IEEE Standard Specifications for Public-Key Cryptography

Sponsor

Microprocessor Standards Committee
of the
IEEE Computer Society

Approved 30 January 2000
IEEE-SA Standards Board

Approved 27 July 2000
American National Standards Institute

Abstract: This standard specifies common public-key cryptographic techniques, including mathematical primitives for secret value (key) derivation, public-key encryption, and digital signatures, and cryptographic schemes based on those primitives. It also specifies related cryptographic parameters, public keys, and private keys. The purpose of this standard is to provide a reference for specifications on a variety of techniques from which applications may select.

Keywords: digital signature, encryption, key agreement, public-key cryptography

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Print: ISBN 0-7381-1956-3 SH94820
PDF: ISBN 0-7381-1957-1 SS94820

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

IEEE is the sole entity that may authorize the use of certification marks, trademarks, or other designations to indicate compliance with the materials set forth herein.

Introduction

(This introduction is not part of IEEE Std 1363-2000, IEEE Standard Specifications for Public-Key Cryptography.)

The P1363 project started as the "Standard for Rivest-Shamir-Adleman, Diffie-Hellman, and Related Public-Key Cryptography," with its first meeting held in January 1994, following a strategic initiative by the Microprocessor Standards Committee to develop standards for cryptography.

P1363's scope broadened with the inclusion of elliptic curve cryptosystems as *related* cryptography, and later the title was changed to the current one to reflect the breadth of the effort. In mid-1996, the working group decided to include three families of techniques, based on three different hard problems—integer factorization, discrete logarithms over finite fields, and elliptic curve discrete logarithms. By late 1996, the set of techniques was fairly stable, with *additional* techniques deferred to the P1363a project.

Most of the next two years saw an increasing intensity of editing, as the Working Group sought to fulfill the scope that remained. In early 1998, the group set a schedule for completion that would bring the document to ballot in late 1998, and final sections of the document were prepared.

The process of developing a standard is always a challenging one, particularly when the subject is as technical as cryptography and the scope is as broad as proposed for P1363. Moreover, as other groups were developing complementary standards at the same time as P1363, close coordination was an essential aspect. Security implies a great deal of caution, and the Working Group was careful in its deliberations not to set any "standards" that might later lead to vulnerabilities (although, as it is pointed out elsewhere, this standard is much more about a framework for specifying public-key techniques, than it is about security, per se).

In addition to this standard, the P1363 project has provided a number of other contributions to the computer security industry. First, it has presented a forum where experts can discuss general issues related to public-key standardization. Second, through its Web page and its call for submissions to P1363a, the project has given a focal point for the presentation of new developments in public-key technology. Third, it has helped facilitate the open discussion of intellectual property issues in this area. And, finally, through its drafts, P1363 has provided reference material to a wide community of cryptographers that otherwise was relegated to textbooks and research papers. For the duration of its existence, the Working Group intends to maintain a Web page containing "errata and latest information" as an additional reference to support P1363 documents (see **http://grouper.ieee.org/groups/1363/index.html**).

The P1363 Working Group is grateful to the many experts who have contributed to this standard, and particularly to those whose development of public-key technology over the past two decades has provided the foundation for information security in the next century.

Participants

The active participants in the IEEE Std 1363-2000 Working Group at the time this standard was completed and balloted were as follows:

Burt Kaliski, *Chair*

Terry S. Arnold, *Vice Chair*

Robert Schlafly, *Secretary*

Michael Markowitz, *Treasurer*

Yiquin Lisa Yin, *Technical Editor*

Benjamin Arazi
Ian Blake
Lily Chen
Louis Finkelstein
Walter Fumy
Donald B. Johnson
Shirley Kawamoto
Tetsutaro Kobayashi
David Kravitz
Pil Joong Lee
Franck Leprevost
Daniel Lieman
Alfred Menezes
Tatsuaki Okamoto
Minghua Qu
Anand Rajan
Leonid Reyzin
Allen Roginsky
Richard Schroeppel
Ari Singer
Jerry Solinas
Kazuo Takaragi
Ashok Vadekar
Scott Vanstone
William Whyte
Robert Zuccherato

In addition, the Working Group would like to thank the following people for their contributions to the standard:

Michel Abdalla
Rich Ankney
David Aucsmith
Paulo S. L. M. Barreto
Mihir Bellare
Tom Berson
Simon Blake-Wilson
Eric Blossom
Uri Blumenthal
Mark Chen
Don Coppersmith
Richard Crandall
Wei Dai
Erik De Win
Jean-Francois Dhem
Whitfield Diffie
Carl Ellison
Amos Fiat
Robert Gallant
John Gilmore
Roger Golliver
Gary L. Graunke
Phillip Griffin
Louis Guillou
Stuart Haber
Shai Halevi
Shouichi Hirose
Robert Hofer
Dale Hopkins
David Hopwood
Russell Housley
Eric Hughes
David Jablon
Marc Joye
Aleksandar Jurisic
Charanjit Jutla
John Kennedy
Katherine T. Kislitzin
Cetin K. Koc
Ray Kopsa
Robert J. Lambert
Peter Landrock
Laurie Law
Chang-Hyi Lee
Jong-In Lim
Moses Liskov
Wenbo Mao
Stephen M. Matyas
Preda Mihailescu
Peter Montgomery
Francois Morain
Peter Neumann
Mark Oliver
Aram Perez
Mohammad Peyravian
Bart Preneel
Jean-Jacques Quisquater
Karen Randall
Richard L. Robertson
Matt Robshaw
Phillip Rogaway
Paul Rubin
Rainer Rueppel
Claus P. Schnorr
Mike Scott
Gadiel Seroussi
Sherry Shannon
Robert D. Silverman
Tim Skorick
David Sowinski
Paul Van Oorschot
Michael J. Wiener
Harold M. Wilensky
Thomas Wu
Susumu Yoshida
Yuliang Zheng

The Working Group apologizes for any inadvertent omissions from the above list. Please note that inclusion of a person's name on the above two lists does not imply that the person agrees with all the materials in this standard.

The following members of the balloting committee voted on this standard:

Carlisle M. Adams
Malcolm J. Airst
Christopher Allen
Terry S. Arnold
Glen Atkins
Simon Blake-Wilson
Lily Chen
John L. Cole
Erik De Win
Dante Del Corso
Stephen L. Diamond
Robert Gallant
Patrick S. Gonia
Julio Gonzalez-Sanz
Gary L. Graunke
Sandor V. Halasz
Jim D. Isaak
Donald B. Johnson
Burt Kaliski

Shirley Kawamoto
David Kravitz
Thomas M. Kurihara
Robert J. Lambert
Pil Joong Lee
Franck Leprevost
Thomas Luedeke
Michael J. Markowitz
Joseph R. Marshall
Serge Mister
Stig Frode Mjolsnes
Paul S. Montague
John E. Montague
Roy Oishi
Minghua Qu
Leonid Reyzin
David Rockwell
Roger Schlafly
Marius Seritan

W. Olin Sibert
Ari Singer
Keith Sollers
Michael Steinacker
Fred J. Strauss
Kazuo Takaragi
Joseph Tardo
Michael D. Teener
Ashok Vadekar
Paul Van Oorschot
Clarence M. Weaver, Jr
Michael J. Weiner
Harold M. Wilensky
Forrest D. Wright
Cheng-Wen Wu
Chung-Huang Yang
Yiqun Lisa Yin
Oren Yuen
Robert Zuccherato

When the IEEE-SA Standards Board approved this standard on 30 January 2000, it had the following membership:

Richard J. Holleman, *Chair*
Donald N. Heirman, *Vice Chair*
Judith Gorman, *Secretary*

Satish K. Aggarwal
Dennis Bodson
Mark D. Bowman
James T. Carlo
Gary R. Engmann
Harold E. Epstein
Jay Forster*
Ruben D. Garzon

James H. Gurney
Lowell G. Johnson
Robert J. Kennelly
E. G. "Al" Kiener
Joseph L. Koepfinger*
L. Bruce McClung
Daleep C. Mohla
Robert F. Munzner

Louis-François Pau
Ronald C. Petersen
Gerald H. Peterson
John B. Posey
Gary S. Robinson
Akio Tojo
Hans E. Weinrich
Donald W. Zipse

*Member Emeritus

Also included is the following nonvoting IEEE-SA Standards Board liaison:

Robert E. Hebner

Jennifer McClain Longman
IEEE Standards Project Editor

Contents

1. Overview 1

1.1 Scope 1
1.2 Purpose 1
1.3 Organization of the document 2

2. References 2

3. Definitions 3

4. Types of cryptographic techniques 7

4.1 General model 7
4.2 Primitives 8
4.3 Schemes 8
4.4 Additional methods 9
4.5 Table summary 9

5. Mathematical conventions 10

5.1 Mathematical notation 11
5.2 Bit strings and octet strings 12
5.3 Finite fields 13
5.4 Elliptic curves and points 14
5.5 Data type conversion 14

6. Primitives based on the discrete logarithm problem 17

6.1 The DL setting 17
6.2 Primitives 19

7. Primitives based on the elliptic curve discrete logarithm problem 27

7.1 The EC setting 27
7.2 Primitives 29

8. Primitives based on the integer factorization problem 37

8.1 The IF setting 37
8.2 Primitives 39

9. Key agreement schemes 46

9.1 General model 46
9.2 DL/ECKAS-DH1 47
9.3 DL/ECKAS-DH2 49
9.4 DL/ECKAS-MQV 50

10. Signature schemes 51

10.1 General model 51

10.2 DL/ECSSA 53
10.3 IFSSA 55

11. Encryption schemes 56

11.1 General model 56
11.2 IFES 57

12. Message-encoding methods 58

12.1 Message-encoding methods for signatures with appendix 59
12.2 Message-encoding methods for encryption 61

13. Key derivation functions 63

13.1 KDF1 64

14. Auxiliary functions 64

14.1 Hash functions 64
14.2 Mask generation functions 65

Annex A (informative) Number-theoretic background 67

A.1 Integer and modular arithmetic: overview 67
A.2 Integer and modular arithmetic: algorithms 72
A.3 Binary finite fields: overview 77
A.4 Binary finite fields: algorithms 85
A.5 Polynomials over a finite field 91
A.6 General normal bases for binary fields 94
A.7 Basis conversion for binary fields 98
A.8 Bases for binary fields: tables and algorithms 104
A.9 Elliptic curves: overview 115
A.10 Elliptic curves: algorithms 121
A.11 Functions for elliptic curve parameter and key generation 131
A.12 Functions for elliptic curve parameter and key validation 134
A.13 Class group calculations 143
A.14 Complex multiplication 147
A.15 Primality tests and proofs 157
A.16 Generation and validation of parameters and keys 162

Annex B (normative) Conformance 170

B.1 General model 170
B.2 Conformance requirements 171
B.3 Examples 173

Annex C (informative) Rationale 177

C.1 General 177
C.2 Keys and domain parameters 178
C.3 Schemes 179

Annex D (informative) Security considerations 182

D.1 Introduction 182
D.2 General principles 182
D.3 Key management considerations 184
D.4 Family-specific considerations 188
D.5 Scheme-specific considerations 198
D.6 Random number generation 208
D.7 Implementation considerations 212

Annex E (informative) Formats 213

E.1 Overview 213
E.2 Representing basic data types as octet strings 213
E.3 Representing outputs of schemes as octet strings 216

Annex F (informative) Bibliography 217

IEEE Standard Specifications for Public-Key Cryptography

1. Overview

1.1 Scope

This standard covers specifications for common public-key cryptographic techniques, including mathematical primitives for secret value (key) derivation, public-key encryption and digital signatures, and cryptographic schemes based on those primitives. Specifications of related cryptographic parameters, public keys, and private keys are also discussed. Classes of computers and communication systems are not restricted.

NOTE—As of the date of this standard's publication, another IEEE project, P1363a, is underway to specify additional techniques. Its intent is to be an amendment to this standard. Its scope is similar to that of this standard, with the addition of identification schemes. (See the note in 1.2 for a description of the purpose of P1363a.)

1.2 Purpose

The transition from paper to electronic media brings with it the need for electronic privacy and authenticity. Public-key cryptography offers fundamental technology addressing this need. Many alternative public-key techniques have been proposed, each with its own benefits. However, there has been no single, comprehensive reference defining a full range of common public-key techniques covering key agreement, public-key encryption, digital signatures, and identification from several families, such as discrete logarithms, integer factorization, and elliptic curves.

It is not the purpose of this standard to mandate any particular set of public-key techniques, or particular attributes of public-key techniques, such as key sizes. Rather, its purpose is to provide a reference for specifications of a variety of techniques from which applications may select.

NOTE—As of the date of this standard's publication, another IEEE project, P1363a, is underway to specify additional techniques. Its intent is to be an amendment to this standard. Its purpose is similar to that of this standard; however, P1363a will focus on newer techniques, while this standard specifies relatively well-established techniques. (See the note in 1.1 for a decription of the scope of P1363a.)

1.3 Organization of the document

This standard contains two parts: the main document and six annexes.

1.3.1 Structure of the main document

- Clause 1 is an overview.
- Clause 2 provides references to other standards.
- Clause 3 defines relevant terms used throughout this standard.
- Clause 4 gives an overview of the types of cryptographic techniques that are described in this standard.
- Clause 5 describes certain mathematical conventions used in the standard, including notation and representation of mathematical objects. It also defines formats to be used in communicating the mathematical objects, as well as primitives for data type conversion.
- Clause 6 through Clause 11 define three families of cryptographic techniques: techniques based on the discrete logarithm problem (DL), the elliptic curve discrete logarithm problem (EC), and the integer factorization problem (IF). Clause 6, Clause 7, and Clause 8 define the setting and primitives used in the DL, EC, and IF families, respectively. Clause 9, Clause 10, and Clause 11 define key agreement schemes, signature schemes, and encryption schemes, respectively.
- Clause 12 defines message-encoding methods for signature and encryption schemes described in Clause 10 and Clause 11.
- Clause 13 defines key derivation functions for key agreement schemes described in Clause 9.
- Clause 14 defines certain auxiliary functions supporting the techniques described in Clause 12 and Clause 13.

1.3.2 Structure of the annexes

The six annexes provide background and helpful information for the users of this standard, as follows:

- Annex A (informative) Number-theoretic background
- Annex B (normative) Conformance
- Annex C (informative) Rationale
- Annex D (informative) Security considerations
- Annex E (informative) Formats
- Annex F (informative) Bibliography

2. References

This standard shall be used in conjunction with the following publications.

FIPS PUB 180-1, Secure Hash Standard, Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce/National Institute of Standards and Technology, April 1995.[1]

[1] FIPS publications are available from the National Technical Information Service (NTIS), U. S. Dept. of Commerce, 5285 Port Royal Rd., Springfield, VA 22161 (http://www.ntis.gov/) and/or from the Federal Information Processing Standards Publications web site at http://www.itl.nist.gov/fipspubs/.

ISO/IEC 10118-3:1998, Information Technology—Security techniques—Hash-functions—
Part 3: Dedicated hash-functions.

NOTES

1—The above references are required for implementing some, but not all, of the techniques in this standard.

2—The mention of any standard in this document is for reference only, and does not imply conformance with that standard. Readers should refer to the relevant standard for full information on conformance with that standard.

3— A bibliography is provided in Annex F.

3. Definitions

For the purposes of this standard, the following terms and definitions apply. The IEEE Standard Dictionary of Electrical and Electronics Terms, Sixth Edition [B71][2] should be referenced for terms not defined in this clause.

3.1 authentication of ownership: The assurance that a given, identified party intends to be associated with a given public key. May also include assurance that the party possesses the corresponding private key.

NOTE—See D.3.2 for more information.

3.2 bit length: *See:* **length**.

3.3 bit string: An ordered sequence of bits (0's and 1's). A bit and a bit string of length 1 are equivalent for all purposes of this standard.

3.4 ciphertext: The result of applying encryption to a message. *Contrast:* **plaintext**. *See also:* **encryption**.

3.5 conformance region: A set of inputs to a primitive or a scheme operation for which an implementation operates in accordance with the specification of the primitive or scheme operation.

NOTE—See B.1 for more information.

3.6 cryptographic families: Three families of techniques, which are presented in this standard, based on the underlying hard problem: discrete logarithm over finite fields (DL), discrete logarithm over elliptic curve groups (EC), and integer factorization (IF).

3.7 decrypt: To produce plaintext from ciphertext. *Contrast:* **encrypt**. *See also:* **ciphertext, encryption, plaintext**.

3.8 digital signature: A digital string for providing authentication. Commonly, in public-key cryptography, it is a digital string that binds a public key to a message in the following way: only the person knowing the message and the corresponding private key can produce the string, and anyone knowing the message and the public key can verify that the string was properly produced. A digital signature may or may not contain the information necessary to recover the message itself. *See also:* **digital signature scheme, public key, public-key cryptography, private key, signature scheme with appendix, signature scheme with message recovery**.

3.9 digital signature scheme: A method for providing authentication. In public-key cryptography, this method can be used to generate a digital signature on a message with a private key in such a way that anyone knowing the corresponding public key can verify that the digital signature was properly produced. *See also:*

[2] The numbers in brackets correspond to those of the bibliography in Annex F.

digital signature, public key, public-key cryptography, private key, signature scheme with appendix, signature scheme with message recovery.

3.10 domain parameters: A set of mathematical objects, such as fields or groups, and other information defining the context in which public/private key pairs exist. More than one key pair may share the same domain parameters. Not all cryptographic families have domain parameters. *See also:* **public/private key pair, valid domain parameters**.

3.11 domain parameter validation: The process of ensuring or verifying that a set of domain parameters is valid. *See also*: **domain parameters, key validation, valid domain parameters**.

3.12 encrypt: To produce ciphertext from plaintext. *Contrast:* **decrypt**. *See also:* **ciphertext, encryption, plaintext**.

3.13 encryption scheme: A method for providing privacy. In public-key cryptography, this method can be used to modify a message with the help of a public key to produce what is known as ciphertext, such that only the holder of the corresponding private key can recover the original message from the ciphertext. *See also:* **ciphertext, plaintext, private key, public key, public-key cryptography**.

3.14 family: *See:* **cryptographic families**.

3.15 field: A setting in which the usual mathematical operations (addition, subtraction, multiplication, and division by nonzero quantities) are possible and obey the usual rules (such as the commutative, associative, and distributive laws). A discrete logarithm (DL) or elliptic curve (EC) scheme is always based on computations in a field.

NOTE—See Koblitz [B95] for a precise mathematical definition.

3.16 finite field: A field in which there are only a finite number of quantities. The discrete logarithm (DL) and elliptic curve (EC) schemes are always implemented over finite fields.

NOTE—See Clause 5 for a description of the particular finite fields used in this standard.

3.17 first bit: The leading bit of a bit string or an octet. For example, the first bit of 0110111 is 0. *Contrast*: **last bit**. *Synonyms:* **most significant bit, leftmost bit**. *See also:* **bit string, octet**.

3.18 first octet: The leading octet of an octet string. For example, the first octet of 1c 76 3b e4 is 1c. *Contrast*: **last octet**. *Synonyms:* **most significant octet, leftmost octet**. *See also:* **octet, octet string**.

3.19 key agreement: A method by which two entities, using each other's public keys and their own private keys, agree on a common secret key that only they know. The secret key is then commonly used in some symmetric cryptography technique. *See also:* **private key, public key, secret key, symmetric cryptography**.

3.20 key confirmation: The assurance provided to each party participating in a key agreement protocol that the other party is capable of computing the agreed-upon key, and that it is the same for both parties. *See also:* **key agreement**.

NOTE—See D.5.1.3 for more information.

3.21 key derivation: The process of deriving a secret key from a secret value. *See also:* **secret key, secret value**.

3.22 key pair: *See:* **public/private key pair**.

3.23 key validation: The process of ensuring or verifying that a key or a key pair is valid. *See also:* **domain parameter validation, public/private key pair, valid key, valid key pair**.

3.24 last bit: The trailing bit of a bit string or an octet. For example, the last bit of 0110111 is 1. *Contrast*: **first bit**. *Synonyms:* **least significant bit, rightmost bit**. *See also:* **first bit, octet**.

3.25 last octet: The trailing octet of an octet string. For example, the last octet of 1c 76 3b e4 is e4. *Contrast:* **first octet**. *Synonyms:* **least significant octet, rightmost octet**. *See also:* **octet, octet string**.

3.26 least significant: *See:* **last bit, last octet**.

3.27 length: **(A)** The length of a bit string is the number of bits in the string. **(B)** The length of an octet string is the number of octets in the string. **(C)** The length in bits of a nonnegative integer n is $\lceil \log_2 (n + 1) \rceil$ (i.e., the number of bits in the integer's binary representation). **(D)** The length in octets of a nonnegative integer n is $\lceil \log_{256} (n + 1) \rceil$ (i.e., the number of digits in the integer's representation base 256). For example, the length in bits of the integer 500 is 9, whereas its length in octets is 2.

3.28 leftmost bit: *See:* **first bit**.

3.29 leftmost octet: *See:* **first octet**.

3.30 message recovery: *See:* **signature scheme with message recovery**.

3.31 message representative: A mathematical value for use in a cryptographic primitive, computed from a message that is input to an encryption or a digital signature scheme. *See also:* **encryption scheme, digital signature scheme**.

3.32 most significant: *See:* **first bit, first octet**.

3.33 octet: A bit string of length 8. An octet has an integer value between 0 and 255 when interpreted as a representation of an integer in base 2. An octet can also be represented by a hexadecimal string of length 2, where the hexadecimal string is the representation of its integer value base 16. For example, the integer value of the octet 10011101 is 157; its hexadecimal representation is 9d. *See also:* **bit string**.

3.34 octet string: An ordered sequence of octets. *See also:* **octet**.

3.35 parameters: *See:* **domain parameters**.

3.36 plaintext: A message before encryption has been applied to it; the opposite of ciphertext. *Contrast:* **ciphertext**. *See also:* **encryption**.

3.37 private key: The private element of the public/private key pair. *See also:* **public/private key pair, valid key**.

3.38 public key: The public element of the public/private key pair. *See also:* **public/private key pair, valid key**.

3.39 public-key cryptography: Methods that allow parties to communicate securely without having prior shared secrets, usually through the use of public/private key pairs. *Contrast:* **symmetric cryptography**. *See also:* **public/private key pair**.

3.40 public/private key pair: A pair of cryptographic keys used in public-key cryptography, consisting of a public key and a private key that correspond to each other by some mathematical relation. The public key is commonly available to a wide audience and can be used to encrypt messages or verify digital signatures. The private key is held by one entity and not revealed to anyone; it is used to decrypt messages encrypted with the public key and/or produce signatures that can be verified with the public key. A public/private key pair can also be used in key agreement. In some cases, a public/private key pair can only exist in the context of domain parameters. *See also:* **digital signature, domain parameters, encryption, key agreement, public-key cryptography, valid key, valid key pair**.

3.41 rightmost bit: *See:* **last bit**.

3.42 rightmost octet: *See:* **last octet**.

3.43 root (of a polynomial): For a polynomial $f(x)$, a value r such that $f(r) = 0$.

3.44 secret key: A key used in symmetric cryptography; commonly needs to be known to all parties involved, but cannot be known to an adversary. *Contrast*: **public/private key pair**. *See also:* **key agreement, shared secret key, symmetric cryptography**.

3.45 secret value: A value that can be used to derive a secret key, but typically cannot by itself be used as a secret key. *See also:* **secret key**.

3.46 shared secret key: A secret key shared by two parties, usually derived as a result of a key agreement scheme. *See also:* **key agreement, secret key**.

3.47 shared secret value: A secret value shared by two parties, usually during a key agreement scheme. *See also:* **key agreement, secret value**.

3.48 signature: *See:* **digital signature**.

3.49 signature scheme with appendix: A digital signature scheme that requires the signed message as input to the verification algorithm. *Contrast:* **signature scheme with message recovery**. *See also:* **digital signature, digital signature scheme**.

3.50 signature scheme with message recovery: A digital signature scheme that contains enough information for recovery of the signed message, thus limiting the possible message size while eliminating the need to transmit the message with the signature and input it to the verification algorithm. *Contrast:* **signature scheme with appendix**. *See also:* **digital signature, digital signature scheme**.

3.51 signature verification: The process of verifying a digital signature. *See also:* **digital signature, digital signature scheme**.

3.52 symmetric cryptography: Methods that allow parties to communicate securely only when they already share some prior secrets, such as the secret key. *Contrast:* **public-key cryptography**. *See also:* **secret key**.

3.53 valid domain parameters: A set of domain parameters that satisfies the specific mathematical definition for the set of domain parameters of its family. While a set of mathematical objects may have the general structure of a set of domain parameters, it may not actually satisfy the definition (e.g., it may be internally inconsistent) and thus not be valid. *See also:* **domain parameters, public/private key pair, valid key, valid key pair, validation**.

3.54 valid key: A key (public or private) that satisfies the specific mathematical definition for the keys of its family, possibly in the context of its set of domain parameters. While some mathematical objects may have the general structure of keys, they may not actually lie in the appropriate set (e.g., they may not lie in the appropriate subgroup of a group, or be out of the bounds allowed by the domain parameters) and thus not be valid keys. *See also:* **domain parameters, public/private key pair, valid domain parameters, valid key pair, validation**.

3.55 valid key pair: A public/private key pair that satisfies the specific mathematical definition for the key pairs of its family, possibly in the context of its set of domain parameters. While a pair of mathematical objects may have the general structure of a key pair, the keys may not actually lie in the appropriate sets (e.g., they may not lie in the appropriate subgroup of a group, or be out of the bounds allowed by the domain parameters), or may not correspond to each other; such a pair is thus not a valid key pair. *See also:* **domain parameters, public/private key pair, valid domain parameters, valid key, validation**.

3.56 validation: *See:* **domain parameter validation, key validation**.

4. Types of cryptographic techniques

This clause gives an overview of the types of cryptographic techniques that are specified in this standard, as well as some requirements for conformance with those techniques. See Annex B for more information on conformance.

4.1 General model

As stated in Clause 1, the purpose of this standard is to provide a reference for specifications of a variety of common public-key cryptographic techniques from which applications may select. Therefore, it is desirable to define these techniques in a framework that allows selection of techniques appropriate for particular applications. Different types of cryptographic techniques can be viewed abstractly according to the following three-level general model:

— *Primitives:* Basic mathematical operations. Historically, they were discovered based on number-theoretic hard problems. Primitives are not meant to achieve security by themselves, but they serve as building blocks for schemes.

— *Schemes:* A collection of related operations combining primitives and additional methods (see 4.4). Schemes can provide complexity-theoretic security, which is enhanced when they are appropriately applied in protocols.

— *Protocols:* Sequences of operations to be performed by multiple parties to achieve some security goal. Protocols can achieve desired security for applications if implemented correctly.

From an implementation viewpoint, primitives can be viewed as low-level implementations (e.g., implemented within cryptographic accelerators, or software modules); schemes can be viewed as medium-level implementations (e.g., implemented within cryptographic service libraries); and protocols can be viewed as high-level implementations (e.g., implemented within entire sets of applications).

General frameworks for primitives, schemes, and additional methods are provided in 4.2, 4.3, and 4.4, and specific techniques are defined in Clause 6 through Clause 14. This standard, however, does not define protocols, mainly because they tend to be application-specific and, hence, are outside the scope of the standard. Nevertheless, the techniques defined in this standard are key components for constructing various cryptographic protocols. Also, Annex D discusses security considerations related to how the techniques can be used in protocols to achieve certain security attributes.

4.2 Primitives

The following types of primitives are defined in this standard:

- Secret value derivation primitives (SVDP): components of key agreement schemes
- Signature primitives (SP) and verification primitives (VP): components of signature schemes
- Encryption primitives (EP) and decryption primitives (DP): components of encryption schemes

Primitives in this standard are presented as mathematical operations, and they are not meant to provide security by themselves. For example, it may be easy to compute message-signature pairs that a signature verification primitive would find consistent with a public key. This is no assurance, however, that the signature was produced on the message by the owner of the corresponding private key. Such assurance can only be provided by a properly implemented signature scheme, which uses the primitive along with other operations.

Primitives assume that their inputs satisfy certain assumptions, as listed with the specification of each primitive. An implementation of a primitive is unconstrained on an input not satisfying the assumptions, as long as it does not adversely affect future operation of the implementation; the implementation may or may not return an error condition. For example, an implementation of a signature primitive may return something that looks like a signature even if its input was not a valid private key. It may also reject the input. It is up to the user of the primitive to guarantee that the input will satisfy the constraints or to include the relevant checks. For example, the user may choose to use the relevant key and domain parameter validation techniques.

The specification of a primitive consists of the following information:

- *Input* to the primitive
- *Assumptions* about the input made in the description of the operation performed by the primitive
- *Output* from the primitive
- *Operation* performed by the primitive, expressed as a series of steps
- *Conformance region recommendations* describing the minimum recommended set of inputs for which an implementation should operate in conformance with the primitive (see Annex B for more information on conformance)

The specifications are functional specifications, not interface specifications. As such, the format of inputs and outputs, and the procedure by which an implementation of a primitive is invoked, are outside the scope of this standard. See Annex E for more information on input and output formats.

4.3 Schemes

The following types of schemes are defined in this standard:

- Key agreement schemes (KAS), in which two parties use their public/private key pairs, and possibly other information, to agree on a shared secret key. The shared secret key may then be used for symmetric cryptography. (See Clause 3 for the definition of symmetric cryptography.)
- Signature schemes with appendix (SSA), in which one party signs a message using its private key, and any other party can verify the signature by examining the message, the signature, and the signer's corresponding public key.

- Encryption schemes (ES), in which any party can encrypt a message using a recipient's public key, and only the recipient can decrypt the message by using its corresponding private key. Encryption schemes may be used for establishing secret keys to be used in symmetric cryptography.

Schemes in this standard are presented in a general form based on certain primitives and additional methods, such as message-encoding methods. For example, an encryption scheme is based on an encryption primitive, a decryption primitive, and an appropriate message-encoding method.

Schemes also include key management operations, such as selecting a private key or obtaining another party's public key. For proper security, a party needs to be assured of the true owners of the keys and domain parameters and of their validity. Generation of domain parameters and keys needs to be performed properly, and, in some cases, validation also needs to be performed. Proper key management, which is outside the scope of this standard, is essential for security. It is addressed in more detail in Annex D.

The specification of a scheme consists of the following information:

- *Scheme options*, such as choices for primitives and additional methods
- One or more *operations*, depending on the scheme, expressed as a series of *steps*
- *Conformance region recommendations* for implementations conforming with the scheme (see Annex B for more information on conformance)

As for primitives, the specifications are functional specifications, not interface specifications. As such, the format of inputs and outputs, and the procedure by which an implementation of a scheme is invoked, are outside the scope of this standard. See Annex E for more information on input and output formats.

4.4 Additional methods

This standard specifies the following additional methods:

- Message-encoding methods, which are components of signature or encryption schemes
    1) Encoding methods for signatures with appendix (EMSA)
    2) Encoding method for encryption (EME)
- Key derivation functions (KDF), which are components of key agreement schemes
- Auxiliary functions, which are building blocks for other additional methods
    1) Mask generation function (MGF), which is used in EME
    2) Hash functions, which are used in EMSA, EME, KDF, and MGF

The specified additional methods are strongly recommended for use in the schemes. The use of an inadequate message-encoding method, key derivation function, or auxiliary function may compromise the security of the scheme in which it is used. Therefore, any implementation that chooses not to follow the recommended additional methods for a particular scheme should perform its own thorough security analysis of the resulting scheme.

4.5 Table summary

Table 1 gives a summary of all the schemes in this standard, together with the primitives and additional methods that are invoked within a scheme.

Table 1—Summary of techniques in the standard

Scheme name	Primitives	Additional methods
Key Agreement Schemes		
DLKAS-DH1	DL/ECSVDP-DH or -DHC	KDF1 (uses a hash function)
DL/ECKAS-DH2	DL/ECSVDP-DH and/or -DHC	KDF1 (uses a hash function)
DLKAS-MQV	DL/ECSVDP-MQV or -MQVC	KDF1 (uses a hash function)
Signature Schemes with Appendix		
DL/ECSSA	DLSP-NR and DLVP-NR OR DLSP-DSA and DLVP- DSA OR ECSP-NR and ECVP-NR OR ECSP-DSA and ECVP-DSA	EMSA1 (uses a hash function)
IFSSA	IFSP-RSA1 and IFVP-RSA1 OR IFSP-RSA2 and IFVP-RSA2 OR IFSP-RW and IFVP-RW	EMSA2 (uses a hash function)
Encryption Schemes		
IFES	IFEP-RSA and IFDP-RSA	EME1 (uses a hash function and a mask generation function)

NOTE—Acronym definitions are as follows:

Families
DL: discrete logarithm
EC: elliptic curve
IF: integer factorization

Schemes
KAS: key agreement scheme
SSA: signature scheme with appendix
ES: encryption scheme

Primitives

SVDP: secret value derivation primitive
SP: signature primitive
VP: verification primitive
EP: encryption primitive
DP: decryption primitive

Additional methods
KDF: key derivation function
EMSA: encoding method for signatures with appendix
EME: encoding method for encryption

Names of techniques
DH: Diffie–Hellman
DHC: Diffie–Hellman with cofactor exponentiation/multiplication
MQV: Menezes–Qu–Vanstone
MQVC: Menezes–Qu–Vanstone with cofactor exponentiation/multiplication
NR: Nyberg–Rueppel
DSA: Digital signature algorithm
RSA: Rivest–Shamir–Adleman
RW: Rabin–Williams

5. Mathematical conventions

This clause describes certain mathematical conventions used in the standard, including notation and representation of mathematical objects. It also contains primitives for data type conversion. Note that the internal representation of mathematical objects is left entirely to the implementation, and may or may not follow the one described below.

5.1 Mathematical notation

The following mathematical notation is used throughout this standard:

0 — Denotes the integer 0, the bit 0, or the additive identity (the element zero) of a finite field. See 5.3 for more on finite fields.

1 — Denotes the integer 1, the bit 1, or the multiplicative identity (the element one) of a finite field. See 5.3 for more on finite fields.

$a \times b$ — The product of *a* and *b*, where *a* and *b* are either both integers, or both finite field elements. When it does not cause confusion, × is omitted and the notation *ab* is used. See 5.3 for more on finite fields.

$a \times P$ — Scalar multiplication of an elliptic curve point *P* by a non-negative integer *a*. When it does not cause confusion, × is omitted and the notation *aP* is used. See 5.4 for more on elliptic curves.

$\lceil x \rceil$ — The smallest integer greater than or equal to the real number *x*. For example, $\lceil 5 \rceil = 5$, $\lceil 5.3 \rceil = 6$, $\lceil -5 \rceil = -5$, and $\lceil -5.3 \rceil = -5$.

$\lfloor x \rfloor$ — The largest integer less than or equal to the real number *x*. For example, $\lfloor 5 \rfloor = 5$, $\lfloor 5.3 \rfloor = 5$, $\lfloor -5 \rfloor = -5$, and $\lfloor -5.3 \rfloor = -6$.

$[a, b]$ — The interval of integers between and including the integers *a* and *b*.

LCM (a, b) — For two positive integers *a* and *b*, denotes the *least common multiple* of *a* and *b* (i.e., the least positive integer that is divisible by both *a* and *b*). See A.1.1 and A.2.2 for an algorithm to compute the LCM.

GCD (a, b) — For two positive integers *a* and *b*, denotes the *greatest common divisor* of *a* and *b* (i.e., the largest positive integer that divides both *a* and *b*). See A.1.1 and A.2.2 for an algorithm to compute the GCD.

$X \oplus Y$ — Bitwise exclusive-or (XOR) of two bit strings or two octet strings *X* and *Y* of the same length.

$X \parallel Y$ — Ordered concatenation of two strings *X* and *Y*. *X* and *Y* are either both bit strings, or both octet strings.

$\log_2 x$ — The logarithmic function of a positive number *x* to the base 2.

$\log_{256} x$ — The logarithmic function of a positive number *x* to the base 256.

$a \bmod n$ — The unique remainder *r*, $0 \le r < n$, when the integer *a* is divided by the positive integer *n*. For example, 23 mod 7 = 2. The operator “mod” has the lowest precedence of all arithmetic operators (e.g., 5 + 8 mod 3 is equal to 13 mod 3, not 5 + 2). See A.1.1 for more details.

$a \equiv b \pmod n$ — The integers *a* and *b* have the same remainder when divided by the positive integer *n*. Pronounced “*a* is congruent to *b* modulo *n*.” This is equivalent to $(a \bmod n) = (b \bmod n)$. See A.1.1 for more details.

$a \not\equiv b \pmod{n}$ The integers a and b have different remainders when divided by the positive integer n. Pronounced "a is not congruent to b modulo n." This is equivalent to $(a \bmod n) \neq (b \bmod n)$.

$a^{-1} \bmod n$ A positive integer $b < n$, if it exists, such that $ab \bmod n = 1$. Pronounced "the (multiplicative) inverse of a modulo n." Also denoted by $1/a \bmod n$. See A.1.1 and A.2.2 for a more detailed description and an algorithm for computing it.

$a/b \bmod n$ An integer a multiplied by the inverse of the integer b modulo n. Equivalent to $ab^{-1} \bmod n$.

$GF(p)$ The finite field of p elements, represented as the integers modulo p, where p is an odd prime number. Also known as a *prime finite field*. See 5.3.1 for more information.

$GF(2^m)$ The finite field containing 2^m elements for some integer $m > 0$. Also known as a *characteristic two finite field*. See 5.3.2 for more information.

$GF(q)$ The finite field containing q elements. For this standard, q will be either a prime number (p) or a power of 2 (2^m).

$\left(\frac{x}{n}\right)$ The Jacobi symbol of the integer x with respect to the integer n. See A.1.4 for a detailed description and A.2.3 for an algorithm to compute the Jacobi symbol.

O The elliptic curve point at infinity. See 5.4 for more information.

$exp(a, b)$ The result of raising a to the power b, where a is an integer or a finite field element, and b is an integer. Also denoted by a^b.

$exp(a)$ The result of raising the number e (where $e = 2.718\ldots$) to the power a, where a is a real number.

NOTE—Throughout this main document, integers and field elements are denoted by lowercase letters, and octet strings and elliptic curve points are denoted by uppercase letters. The one exception is when a public key that can be either a field element or an elliptic curve point is denoted by a lowercase letter in the DL and EC schemes (see 9.2, 9.3, 9.4, and 10.2).

5.2 Bit strings and octet strings

Bit strings and octet strings are ordered sequences. The terms *first* and *last*, *leftmost* and *rightmost*, and *leading* and *trailing* are used to distinguish the ends of these sequences (*first*, *leftmost*, and *leading* are equivalent; *last*, *rightmost*, and *trailing* are equivalent; other publications sometimes use *most significant*, which is synonymous with *leading*, and *least significant*, which is synonymous with *trailing*).

Note that when a string is represented as a sequence, it may be indexed from right to left or from left to right, starting with any index. This does not change the meaning of the terms above. For example, consider the octet string of 4 octets: 1c 76 3b e4. One can represent it as a string $a_0\ a_1\ a_2\ a_3$, with a_0 = 1c, a_1 = 76, a_2 = 3b, and a_3 = e4. In this case, a_0 represents the first octet, and a_3 represents the last octet. Alternatively, one can represent it as a string $a_1\ a_2\ a_3\ a_4$, with a_1 = 1c, a_2 = 76, a_3 = 3b, and a_4 = e_4. In this case, a_1 represents the first octet, and a_4 represents the last octet. Yet another possibility would be to represent it as $a_3\ a_2\ a_1\ a_0$, with a_3 = 1c, a_2 = 76, a_1 = 3b, and a_0 = e4. In this case, a_3 represents the first octet and a_0 represents the last octet. No matter how this string is represented, the value of the first octet is always 1c and the value of the last octet is always e4.

5.3 Finite fields

This subclause describes the kinds of underlying finite fields $GF(q)$ that shall be used, and how they are to be represented for purposes of conversion with the primitives in 5.5. As noted above, the internal representation of objects is left to the implementation, and may be different. If the internal representation is different, conversion to the representation defined here may be needed at certain points in cryptographic operations.

5.3.1 Prime finite fields

A *prime finite field* is a field containing a prime number of elements. If p is an odd prime number, then there is a unique field $GF(p)$ with p elements. For purposes of conversion, the elements of $GF(p)$ shall be represented by the integers 0, 1, 2 ,..., $p - 1$.

A description of the arithmetic of $GF(p)$ is given in A.1 and A.2.

5.3.2 Characteristic two finite fields

A *characteristic two finite field* (also known as a *binary* finite field) is a finite field whose number of elements is a power of 2. If $m \geq 1$, then there is a unique field $GF(2^m)$ with 2^m elements. For purposes of conversion, the elements of $GF(2^m)$ should be represented by bit strings of length m.

There are several ways of performing arithmetic in $GF(2^m)$. The specific rules depend on how the bit strings are interpreted. For purposes of conversion, this interpretation should be made in terms of one of the following *basis representations*.

NOTE—As opposed to the representation method for prime finite fields, which is required, the representation methods for binary finite fields are recommendations. The choice of a particular representation for binary fields affects the primitives and schemes in which conversion from field elements to other objects is used. See D.4.1.3 and D.4.2.3 (and related notes in D.4.1.4 and D.4.2.4) for related security considerations.

5.3.2.1 Polynomial basis over GF (2)

This representation is determined by choosing an irreducible binary polynomial $p(t)$ of degree m. (See A.3 and A.4 for definitions of the above terms and for a description of the arithmetic of a field using this representation.) If the polynomial basis representation over $GF(2)$ is used, then, for purposes of conversion, the bit string

$$(a_{m-1} \ldots a_2\, a_1\, a_0)$$

shall be taken to represent the polynomial

$$a_{m-1}t^{m-1} + \ldots + a_2t^2 + a_1t + a_0$$

where the coefficients a_i are elements of $GF(2)$.

In particular, for purposes of conversion, the additive identity (zero) element of the field is represented by a string of all zero bits, and the multiplicative identity (one) element of the field is represented by a string where all bits but the last are zero, and the last bit is one. (Note, however, that in mathematical expressions in this standard, for typographic convenience, the numeral 0 is used to represent the element zero of a field, and the numeral 1 is used to represent the element one of the field.)

NOTE—In keeping with traditional mathematical notation, the bits in this representation are indexed from right to left, as opposed to the bits in the normal basis representation (see 5.3.2.2), which are indexed from left to right (see also 5.2).

5.3.2.2 Normal basis over GF (2)

This representation is determined by choosing a normal polynomial $p(t)$ of degree m. (See A.3 and A.4 for the definitions of the above terms and for a description of the arithmetic of a field using this representation.) If the normal basis representation over GF (2) is used, then, for purposes of conversion, the bit string

$$(a_0\ a_1\ \ldots\ a_{m-1})$$

shall be taken to represent the element

$$a_0\theta + a_1\theta^{2} + a_2\theta^{2^2} + \ldots + a_{m-1}\theta^{2^{m-1}}$$

where θ is a root of $p(t)$ in $GF\ (2^m)$.

In particular, for purposes of conversion, the additive identity (zero) element of the field is represented by a string of all zero bits, and the multiplicative identity (one) element of the field is represented by a string of one bits. (Note, however, that in mathematical expressions in this standard, for typographic convenience, the numeral 0 is used to represent the element zero of a field, and the numeral 1 is used to represent the element one of the field.)

NOTE—In keeping with traditional mathematical notation, the bits in this representation are indexed from left to right, as opposed to the bits in the polynomial basis representation (see 5.3.2.1), which are indexed from right to left. (See also 5.2.)

5.4 Elliptic curves and points

An elliptic curve E defined over $GF\ (q)$ is a set of points $P = (x_P\ ,\ y_P)$, where x_P and y_P are elements of $GF\ (q)$ that satisfy a certain equation, together with the point at infinity denoted by O. For purposes of this standard, it is specified by two field elements, $a \in GF\ (q)$ and $b \in GF\ (q)$, called the *coefficients* of E. The field elements x_P and y_P are called the x-coordinate of P and the y-coordinate of P, respectively.

If q is an odd prime, then a and b shall satisfy $4\ a^3 + 27\ b^2 \neq 0$ in $GF\ (q)$, and every point $P = (x_P\ ,\ y_P)$ on E (other than the point O) shall satisfy the following equation in $GF\ (q)$:

$$y_P^2 = x_P^3 + ax_P + b$$

If q is a power of 2, then b shall be nonzero in $GF\ (q)$, and every point $P = (x_P\ ,\ y_P)$ on E (other than the point O) shall satisfy the following equation in $GF\ (q)$:

$$y_P^2 + x_P y_P = x_P^3 + ax_P^2 + b$$

See A.9 and A.10 for more on elliptic curves and elliptic curve arithmetic.

5.5 Data type conversion

This subclause describes the primitives that shall be used to convert between different types of objects and strings when such conversion is required in primitives, schemes, or encoding techniques. Representation of mathematical and cryptographic objects as octet strings is not specifically addressed here; rather, it is discussed in Annex E. Figure 1 shows the primitives presented in this subclause.

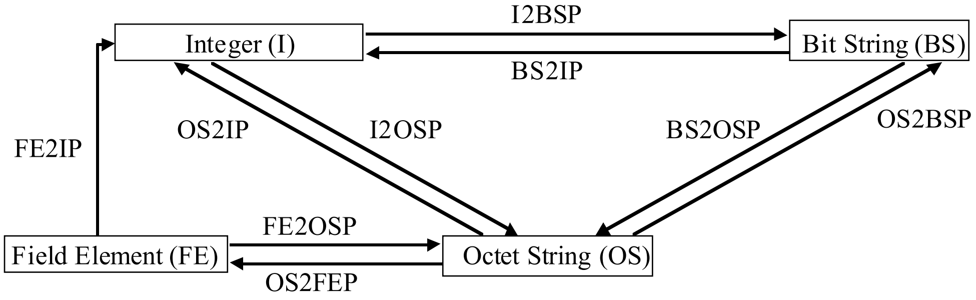

Figure 1—Summary of data type conversion primitives

5.5.1 Converting between integers and bit strings (I2BSP and BS2IP)

In performing cryptographic operations, bit strings sometimes need to be converted to non-negative integers and vice versa.

To convert a non-negative integer x to a bit string of length l (l has to be such that $2^l > x$), the integer shall be written in its unique l-digit representation base 2

$$x = x_{l-1}\, 2^{l-1} + x_{l-2}\, 2^{l-2} + \ldots + x_1\, 2 + x_0$$

where x_i is either 0 or 1 (note that one or more leading digits will be zero if $x < 2^{l-1}$). Then let the bit b_i have the value x_{l-i} for $1 \le i \le \mathrm{l}$. The bit string shall be $b_1\, b_2 \ldots b_l$.

For example, the integer 10945 is represented by a bit string of length 19 as 000 0010 1010 1100 0001.

The primitive that converts integers to bit strings is called the Integer to Bit String Conversion Primitive, or I2BSP. It takes an integer x and the desired length l as input, and outputs the bit string if $2^l > x$. It shall output “error” otherwise.

The primitive that converts bit strings to integers is called the Bit String to Integer Conversion Primitive, or BS2IP. It takes a bit string as input and outputs the corresponding integer. Note that the bit string of length zero (the empty bit string) is converted to the integer 0.

5.5.2 Converting between bit strings and octet strings (BS2OSP and OS2BSP)

To represent a bit string as an octet string, one simply pads enough zeroes on the left to make the number of bits a multiple of eight, and then breaks it up into octets. More precisely, a bit string $b_{l-1}\, b_{l-2} \ldots b_0$ of length l shall be converted to an octet string $M_{d-1}\, M_{d-2} \ldots M_0$ of length $d = \lceil l/8 \rceil$ as follows: for $0 \le i < d - 1$, let the octet $M_i = b_{8i+7}\, b_{8i+6} \ldots b_{8i}$. The leftmost octet M_{d-1} shall have its leftmost $8d - l$ bits set to zero; its rightmost $8 - (8d - l)$ bits shall be $b_{l-1}\, b_{l-2} \ldots b_{8d-8}$.

The primitive that converts bit strings to octet strings is called the Bit String to Octet String Conversion Primitive, or BS2OSP. It takes the bit string as input and outputs the octet string.

The primitive that converts octet strings to bit strings is called the Octet String to Bit String Conversion Primitive, or OS2BSP. It takes an octet string of length d and the desired length l of the bit string as input. OS2BSP shall output the bit string if $d = \lceil l/8 \rceil$ and if the leftmost $8d - l$ bits of the leftmost octet are zero; it shall output “error” otherwise.

5.5.3 Converting between integers and octet strings (I2OSP and OS2IP)

To represent a non-negative integer x as an octet string of length l (l has to be such that $256^l > x$), the integer shall be written in its unique l-digit representation base 256

$$x = x_{l-1}\, 256^{l-1} + x_{l-2}\, 256^{l-2} + \ldots + x_1\, 256 + x_0$$

where $0 \le x_i < 256$ (note that one or more leading digits will be zero if $x < 256^{l-1}$). Then let the octet M_i have the value x_i for $0 \le i \le l$-1. The octet string shall be $M_{l\text{-}1}\ M_{l\text{-}2} \ldots M_0$.

For example, the integer 10945 is represented by an octet string of length 3 as 00 2A C1.

The primitive that converts integers to octet strings is called the Integer to Octet String Conversion Primitive, or I2OSP. It takes an integer x and the desired length l as input, and outputs the octet string if $256^l > x$. It shall output “error” otherwise.

The primitive that converts octet strings to integers is called the Octet String to Integer Conversion Primitive, or OS2IP. It takes an octet string as input and outputs the corresponding integer. Note that the octet string of length zero (the empty octet string) is converted to the integer 0.

5.5.4 Converting between finite field elements and octet strings (FE2OSP and OS2FEP)

An element x of a finite field $GF\ (q)$, for purposes of this standard, is represented by an integer if q is an odd prime (see 5.3.1) or by a bit string if q is a power of two (see 5.3.2). If q is an odd prime, then to represent x as an octet string, I2OSP shall be used with the integer value of x and the length $\lceil \log_{256} q \rceil$ as inputs. If q is a power of two, then to represent x as an octet string, BS2OSP shall be applied to the bit string representing x.

The primitive that converts finite field elements to octet strings is called the Field Element to Octet String Conversion Primitive, or FE2OSP. It takes a field element x and the field size q as inputs, and outputs the corresponding octet string.

To convert an octet string back to a field element, if q is an odd prime, then OS2IP shall be used with the octet string as the input. If q is a power of two, then OS2BSP shall be used with the octet string and the length $\log_2 q$ as inputs.

The primitive that converts octet strings to finite field elements is called the Octet String to Field Element Conversion Primitive, or OS2FEP. It takes the octet string and the field size q as inputs and outputs the corresponding field element. It shall output “error” if OS2BSP or OS2IP outputs “error.”

5.5.5 Converting finite field elements to integers (FE2IP)

In performing cryptographic operations, finite field elements sometimes need to be converted to non-negative integers. The primitive that performs this is called the Field Element to Integer Conversion Primitive, or FE2IP.

An element j of a finite field $GF\ (q)$ shall be converted to a non-negative integer i by the following, or an equivalent, procedure:

1. Convert the element j to an octet string using FE2OSP.
2. Convert the resulting octet string to an integer i using OS2IP.
3. Output i.

Note that if q is an odd prime, j is already represented as an integer and FE2IP merely outputs that representation. If q is a power of two, then FE2IP outputs an integer whose binary representation is the same as the bit string representing j.

6. Primitives based on the discrete logarithm problem

This clause specifies the family of cryptographic primitives based on the discrete logarithm problem over finite fields, also known as the DL family. For background information on this family, see A.1.2, A.3.9, and Diffie and Hellman [B47].

6.1 The DL setting

6.1.1 Notation

The list below introduces the notation used in Clause 6, Clause 9, and Clause 10. It is meant for reference only; for complete definitions of the terms listed, refer to the appropriate text. Some other symbols are also used occasionally and are introduced in the text where appropriate.

q	The size of the field used (part of the DL domain parameters)
r	The prime divisor of $q - 1$ and the order of g (part of the DL domain parameters)
g	An element of the field $GF(q)$ generating a multiplicative subgroup of order r (part of the DL domain parameters)
k	$(q - 1) / r$, the cofactor
s, u, s', u'	DL private keys, integers, corresponding to public keys w, v, w', and v', respectively
w, v, w', v'	DL public keys, elements of $GF(q)$, corresponding to private keys s, u, s', and u', respectively
$(s, w), (u, v)$	DL key pairs, where s and u are private keys, and w and v are the corresponding public keys
z, z_1, z_2	Shared secret values, elements of $GF(q)$, derived by secret value derivation primitives
K	Shared secret key agreed upon by a key agreement scheme
(c, d)	Signature, a pair of integers, computed by a signature primitive
f	Message representative, an integer, computed by a message-encoding operation
M	The message, an octet string, whose signature is computed by a signature scheme

NOTES

1—When keys from two parties are involved in a primitive or scheme, the symbols s, u, w, and v are used to denote the party's own keys, and the symbols s', u', w', and v' are used to denote the other party's keys.

2—Multiplication of field elements, as well as multiplication of integers, is denoted by ×, although this symbol may be omitted when such omission does not cause ambiguity. The notation such as $\exp(w, c)$, where w is a field element and c is an integer, will be used to denote raising w to the power c. The more traditional notation w^c will be used if w is an integer.

3—Throughout this clause, operations on finite field elements and integers are used. Care needs to be exercised to distinguish integers from field elements, as operations on integers and field elements are denoted by the same symbols. (See 5.3 for more information on finite fields.)

6.1.2 DL domain parameters

DL domain parameters are used in every DL primitive and scheme and are an implicit component of every DL key. A set of DL domain parameters specifies a field *GF* (q), where q is a positive odd prime integer p or 2^m for some positive integer m; a positive prime integer r dividing $q - 1$; and a field element g of multiplicative order r (g is called the *generator* of the subgroup of order r). If $q = 2^m$, it also specifies a representation for the elements of *GF* (q) to be used by the conversion primitives (see 5.3 and 5.5). Implicitly, it also specifies the cofactor $k = (q - 1) / r$. If the DLSVDP-DHC or DLSVDP-MQVC primitive is to be applied, then it shall also be the case that GCD (k, r) = 1 (i.e., r does not divide k; see A.1.1).

Depending on the scheme and protocol used, a party may need to generate its own set of domain parameters or use domain parameters provided by another party. If parameters are provided by another party, their authenticity may need to be determined (as discussed in D.3.2), and they may need to be validated (see below). The security issues related to domain parameter generation are discussed in D.3.1 and D.4.1. Suggested methods for generating DL domain parameters are discussed in A.16.1 and A.16.3.

Parties establish DL domain parameters as part of key management for a scheme. Depending on the key management technique, it is possible that the established domain parameters do not satisfy the intended definition, even though they have the same general form (i.e., components q, r, g, and optionally k). To accommodate this possibility, the term *DL domain parameters* shall be understood in this standard as referring to instances of the general form for DL domain parameters. The term *valid DL domain parameters* shall be reserved for DL domain parameters satisfying the definition.

Domain parameter validation is the process of determining whether a set of DL domain parameters is valid. Further discussion of domain parameter validation is contained in D.3.3. Suggested algorithms for DL domain parameter validation are contained in A.16.2 and A.16.4.

There may be more than one set of domain parameters used in a primitive or scheme. The sets of DL domain parameters may be different for different keys, or they may be the same, depending on the requirements of a primitive or scheme. Unlike keys, which are not meant to be shared among users, a set of domain parameters can, and sometimes needs to be, shared. DL domain parameters are often public; the security of the schemes in this standard does not rely on these domain parameters being secret.

6.1.3 DL key pairs

For a given set of DL domain parameters, a *DL key pair* consists of a *DL private key s*, which is an integer in the range [1, $r - 1$] and a DL *public key w*, which is an element of *GF* (q), where $w = \exp(g, s)$. For security, DL private keys need to be generated so that they are unpredictable and stored so that they are inaccessible to an adversary. DL private keys may be stored in any form convenient to the application.

DL key pairs are closely associated with their domain parameters, and can only be used in the context of the domain parameters. A key pair shall not be used with a set of domain parameters that is different from the one for which it was generated. A set of domain parameters may be shared by a number of key pairs (see D.4.1.2 and D.4.1.4, Note 1).

A DL key pair may or may not be generated by the party using it, depending on the trust model. This and other security issues related to DL key pair generation are discussed in D.3.1 and D.4.1. A suggested method for generating DL key pairs is contained in A.16.5.

As is also the case for DL domain parameters, parties establish DL keys as part of key management for a scheme and, depending on the key management technique, it is possible that an established key does not satisfy the intended definition for the key, even though it has the same general form. Accordingly, the terms "DL public key" and "DL private key" shall be understood in this standard as referring to instances of the general form for the key, and the terms *valid DL public key*, *valid DL private key*, and *valid DL key pair* shall be reserved for keys satisfying the definition.

Key validation is the process of determining whether a key is valid. Further discussion of key validation is contained in D.3.3. A suggested algorithm for DL public-key validation is contained in A.16.6. In some cases (when the DLSVDP-DHC or DLSVDP-MQVC primitive is applied), it may be necessary to verify only that the public key is a member of $GF(q)$ (rather than a stronger condition that it is a power of g other than one). The algorithm to verify this weaker condition is also contained in A.16.6. No algorithm is given for DL private-key validation, because, generally, a party controls its own private key and need not validate it. However, private-key validation may be performed if desired.

6.2 Primitives

This subclause describes primitives used in the DL family. Before proceeding with this subclause, the reader should be familiar with the material in Clause 4.

As detailed in Clause 4 and Annex B, an implementation of a primitive may make certain assumptions about its inputs, as listed with the specification of the primitive. For example, if DL domain parameters q, r, and g and a public key w are inputs to a primitive, an implementation may generally assume that the domain and the public key parameters are valid (i.e., that g has order r in $GF(q)$ and $w = \exp(g, s)$ for some integer s in the interval $[1, r - 1]$). The behavior of an implementation is not specified if w is not a power of g and, in such a case, the implementation may or may not output an error condition. It is up to the properly implemented scheme to ensure that only appropriate inputs are passed to a primitive, or to accept the risks of passing inappropriate inputs. See Annex B for more on conforming with a primitive.

6.2.1 DLSVDP-DH

DLSVDP-DH is the Discrete Logarithm Secret Value Derivation Primitive, Diffie-Hellman version. It is based on the work in Diffie and Hellman [B47]. This primitive derives a shared secret value from one party's public key and another party's private key, where the keys have the same set of DL domain parameters. If two parties correctly execute this primitive with corresponding keys as inputs, they will produce the same output. This primitive can be invoked by a scheme to derive a shared secret key; specifically, it may be used with the schemes DLKAS-DH1 and DL/ECKAS-DH2. It assumes that the input keys are valid (see also 6.2.2).

Input:

- The DL domain parameters q, r, and g associated with the keys s and w' (the domain parameters shall be the same for both s and w').
- The party's own private key s.
- The other party's public key w'.

Assumptions: Private key s, DL domain parameters q, r, and g, and public key w' are valid; both keys are associated with the domain parameters.

Output: The derived shared secret value, which is a nonzero field element $z \in GF(q)$

Operation: The shared secret value z shall be computed by the following or an equivalent sequence of steps:

1. Compute a field element $z = \exp(w', s)$.
2. Output z as the shared secret value.

Conformance region recommendation: A conformance region should include the following:

- At least one valid set of DL domain parameters q, r, and g
- At least one valid private key s for each set of domain parameters
- All valid public keys w' associated with the same set of domain parameters as s

NOTES

1—This primitive does not address small subgroup attacks (see D.5.1.6), which may occur valid. To prevent them, a key agreement scheme should validate the public key w' before executing this primitive (see also 6.2.2).

2—When the public key w' and the private key s are valid, w' has order r and s is in the interval [1, r–1]. Thus, the output z in this case is neither the element zero nor the element one.

6.2.2 DLSVDP-DHC

DLSVDP-DHC is the Discrete Logarithm Secret Value Derivation Primitive, Diffie-Hellman version with cofactor exponentiation. It is based on the work in Diffie and Hellman [B47], Kaliski [B88], and Law et. al. [B98]. This primitive derives a shared secret value from one party’s public key and another party’s private key, where the keys have the same set of DL domain parameters. If two parties correctly execute this primitive with corresponding keys as inputs, they will produce the same output. This primitive can be invoked by a scheme to derive a shared secret key; specifically, it may be used with the schemes DLKAS-DH1 and DL/ECKAS-DH2. It does not assume the validity of the input public key, but does assume that the public key is an element of $GF(q)$ (see also 6.2.1).

Input:

- The DL domain parameters q, r, g, and the cofactor k associated with the keys s and w' (the domain parameters shall be the same for both s and w')
- The party’s own private key s
- The other party’s public key w'
- An indication as to whether compatibility with DLSVDP-DH is desired

Assumptions: Private key s and DL domain parameters q, r, g, and k are valid; the private key is associated with the domain parameters; w' is in $GF(q)$; GCD $(k, r) = 1$

Output: The derived shared secret value, which is a nonzero field element $z \in GF(q)$; or “invalid public key”

Operation: The shared secret value z shall be computed by the following or an equivalent sequence of steps:

1. If compatibility with DLSVDP-DH is desired, then compute an integer $t = k^{-1}s \bmod r$; otherwise, set $t = s$.
2. Compute a field element $z = \exp(w', kt)$.
3. If $z = 0$ or $z = 1$, output “invalid public key”; else, output z as the shared secret value.

Conformance region recommendation: A conformance region should include the following:

- At least one valid set of DL domain parameters q, r, g, and k
- At least one valid private key s for each set of domain parameters
- All elements w' in $GF(q)$, where q is from the domain parameters of s

— Compatibility with DLSVDP-DH may be preset by the implementation, or given as an input flag

NOTES

1—This primitive addresses small subgroup attacks, which may occur when the public key w' is not valid (see D.5.1.6). A key agreement scheme only needs to validate that w' is in $GF(q)$ before executing this primitive (see also 6.2.1).

2—The cofactor k depends only on the DL domain parameters and is equal to $(q - 1) / r$. Hence, it can be computed once for a given set of domain parameters and stored as part of the domain parameters. Similarly, in the compatibility case, the value k^{-1} can be computed once and stored with the domain parameters, and the integer t can be computed once for a given private key s.

3—When the public key w' and the private key s are valid, the output z will be an element of a subset of $GF(q)$ that consists of all the powers of g (except for the element one). As a consequence, z cannot be the element zero or the element one in this case. When the public key is invalid, the output will be either "invalid public key" or an element of order r in $GF(q)$; in particular, it will not be in a small subgroup.

4—In the compatibility case, DLSVDP-DHC computes the same output for valid keys as DLSVDP-DH, so an implementation that conforms with DLSVDP-DHC in the compatibility case also conforms with DLSVDP-DH.

6.2.3 DLSVDP-MQV

DLSVDP-MQV is the Discrete Logarithm Secret Value Derivation Primitive, Menezes-Qu-Vanstone version. It is based on the work of Law et al. [B98]. This primitive derives a shared secret value from one party's two key pairs and another party's two public keys, where all the keys have the same set of DL domain parameters. If two parties execute this primitive with corresponding keys as inputs, they will produce the same output. This primitive can be invoked by a scheme to derive a shared secret key; specifically, it may be used with the scheme DLKAS-MQV. It assumes that the input keys are valid (see also 6.2.4).

In this primitive, let $h = \lceil (\log_2 r) / 2 \rceil$. Note that h depends only on the DL domain parameters and, hence, can be computed once for a given set of domain parameters.

Input:

— The DL domain parameters q, r, and g associated with the keys s, (u, v), w', and v' (the domain parameters shall be the same for these keys)

— The party's own first private key s

— The party's own second key pair (u, v)

— The other party's public key w'

— The other party's second public key v'

Assumptions: Private key s, key pair (u, v), public keys w', v', and DL domain parameters q, r, and g are valid; all the keys are associated with the domain parameters

Output: The derived shared secret value, which is a nonzero field element $z \in GF(q)$; or "error"

Operation: The shared secret value z shall be computed by the following or an equivalent sequence of steps:

1. Convert v into an integer i using FE2IP.
2. Compute an integer $t = i \bmod 2^h$. Let $t = t + 2^h$.
3. Convert v′ into an integer i′ using FE2IP.
4. Compute an integer $t' = i' \bmod 2^h$. Let $t' = t' + 2^h$.
5. Compute an integer $e = ts + u \bmod r$.

6. Compute a field element $z = \exp(v' \times \exp(w', t'), e)$.
7. If $z = 1$, output "error" and stop.
8. Output z as the shared secret value.

Conformance region recommendation: A conformance region should include the following:

- At least one valid set of DL domain parameters q, r, and g
- At least one valid private key s for each set of domain parameters
- All valid key pairs (u, v) associated with the same set of domain parameters as s
- All valid public keys w' and v' associated with the same set of domain parameters as s

NOTES

1—This primitive does not address small subgroup attacks (see D.5.1.6), which may occur when the public keys w' and v' are not valid. To prevent them, a key agreement scheme should validate the public keys w' and v' before executing this primitive (see also 6.2.4).

2—When the public keys w' and v' are valid, the output z will be an element of a subset of $GF(q)$ that consists of all the powers of g. It is possible (although very unlikely) that even though all the keys and parameters are valid, z will be the element one. In this case, the primitive will output "error," and will need to be rerun with a different input. Except for this rare case, z will always be defined and will not be the element zero or the element one, as long as the input keys and parameters are valid.

6.2.4 DLSVDP-MQVC

DLSVDP-MQVC is the Discrete Logarithm Secret Value Derivation Primitive, Menezes-Qu-Vanstone version with cofactor exponentiation. It is based on the work in Kaliski [B88] and Law et al. [B98]. This primitive derives a shared secret value from one party's two key pairs and another party's two public keys, where all the keys have the same set of DL domain parameters. If two parties execute this primitive with corresponding keys as inputs, they will produce the same output. This primitive can be invoked by a scheme to derive a shared secret key; specifically, it may be used with the scheme DLKAS-MQV. It does not assume the validity of the input public keys, but does assume that the public key is an element of $GF(q)$ (see also 6.2.3).

In this primitive, let $h = \lceil (\log_2 r) / 2 \rceil$. Note that h depends only on the DL domain parameters and, hence, can be computed once for a given set of domain parameters.

Input:

- The DL domain parameters q, r, g, and the cofactor k associated with the keys s, (u, v), w', and v' (the domain parameters shall be the same for these keys)
- The party's own first private key s
- The party's own second key pair (u, v)
- The other party's public key w'
- The other party's second public key v'
- An indication as to whether compatibility with DLSVDP-MQV is desired

Assumptions: Private key s, key pair (u, v), and DL domain parameters q, r, g, and k are valid; private key s and key pair (u, v) are associated with the domain parameters; w' and v' are in $GF(q)$; GCD $(k, r) = 1$.

Output: The derived shared secret value, which is a nonzero field element $z \in GF(q)$; or "invalid public key"

Operation: The shared secret value z shall be computed by the following or an equivalent sequence of steps:

1. Convert v into an integer i using FE2IP.
2. Compute an integer $t = i \bmod 2^h$. Let $t = t + 2^h$.
3. Convert v' into an integer i' using FE2IP.
4. Compute an integer $t' = i' \bmod 2^h$. Let $t' = t' + 2^h$.
5. If compatibility with DLSVDP-MQV is desired, then compute an integer $e = k^{-1}(ts + u) \bmod r$; otherwise, compute an integer $e = ts + u \bmod r$.
6. Compute a field element $z = \exp(v' \times \exp(w', t'), ke)$.
7. If $z = 0$ or $z = 1$, output "invalid public key"; else, output z as the shared secret value.

Conformance region recommendation: A conformance region should include:

- At least one valid set of DL domain parameters q, r, g, and k
- At least one valid private key s for each set of domain parameters
- All valid key pairs (u, v) associated with the same set of domain parameters as s
- All elements w' and v' in $GF(q)$, where q is from the domain parameters of s
- Compatibility with DLSVDP-MQV may be preset by the implementation, or given as an input flag

NOTES

1—This primitive addresses small subgroup attacks, which may occur when the public keys w' and v' are not valid (see D.5.1.6). A key agreement scheme only needs to validate that w' and v' are in $GF(q)$ before executing this primitive (see also 6.2.3).

2—The cofactor k depends only on the DL domain parameters and is equal to $(q - 1) / r$. Hence, it can be computed once for a given set of domain parameters and stored as part of the domain parameters. Similarly, in the compatibility case, the value k^{-1} can be computed once and stored with the domain parameters. An equivalent way to compute the integer e in this case is as $(tk^{-1}s + k^{-1}u) \bmod r$, where $k^{-1}s \bmod r$ and $k^{-1}u \bmod r$ can be computed once for each given private key s and u.

3—When the public keys w' and v' are valid, the output z will be an element of a subset of $GF(q)$ that consists of all the powers of g. It is possible (although very unlikely) that even though all the keys and parameters are valid, z will be the element one, and the primitive will output "invalid public key." In this case, the primitive will need to be rerun with a different input. Except for this rare case, z will always be defined and will not be the element zero or the element one, as long as the input keys and parameters are valid. When one of the public keys is invalid, the output will be either "invalid public key" or an element of order r in $GF(q)$; in particular, it will not be in a small subgroup.

4—In the compatibility case, DLSVDP-MQVC computes the same output for valid keys as DLSVDP-MQV, so an implementation that conforms with DLSVDP-MQVC in the compatibility case also conforms with DLSVDP-MQV.

6.2.5 DLSP-NR

DLSP-NR is the Discrete Logarithm Signature Primitive, Nyberg-Rueppel version. It is based on the work of Nyberg and Rueppel [B120]. It can be invoked in a scheme to compute a signature on a message representative with the private key of the signer, in such a way that the message representative can be recovered from the signature using the public key of the signer by the DLVP-NR primitive. Note, however, that no DL signature schemes with message recovery are defined in this standard (see C.3.4). DLSP-NR may also be used in a signature scheme with appendix and can be invoked in the scheme DLSSA as part of signature generation.

Input:

- The DL domain parameters q, r, and g associated with the key s
- The signer's private key s
- The message representative, which is an integer f such that $0 \leq f < r$

Assumptions: Private key s and DL domain parameters q, r, and g are valid and associated with each other; $0 \leq f < r$.

Output: The signature, which is a pair of integers (c, d), where $1 \leq c < r$ and $0 \leq d < r$

Operation: The signature (c, d) shall be computed by the following or an equivalent sequence of steps:

1. Generate a key pair (u, v) with the same set of domain parameters as the private key s (see the note below).
2. Convert v into an integer i with primitive FE2IP [recall that v is an element of $GF(q)$].
3. Compute an integer $c = i + f \bmod r$. If $c = 0$, then go to step 1.
4. Compute an integer $d = u - sc \bmod r$.
5. Output the pair (c, d) as the signature.

Conformance region recommendation: A conformance region should include the following:

- At least one valid set of DL domain parameters q, r, and g
- At least one valid private key s for each set of domain parameters
- All message representatives f in the range $[0, r - 1]$, where r is from the domain parameters of s

NOTE—The key pair in step 1 should be a one-time key pair, which is generated and stored by the signer following the security recommendations of D.3.1, D.4.1.2, D.6, and D.7. A new key pair should be generated for every signature. The one-time private key u should be discarded after step 4, as its recovery by an opponent can lead to the recovery of the private key s.

6.2.6 DLVP-NR

DLVP-NR is the Discrete Logarithm Verification Primitive, Nyberg-Rueppel version. It is based on the work of Nyberg and Rueppel [B120]. This primitive recovers the message representative that was signed with DLSP-NR, given only the signature and public key of the signer. It can be invoked in a scheme as part of signature verification and, possibly, message recovery. Note, however, that no DL signature schemes with message recovery are defined in this standard (see C.3.4). DLVP-NR may also be used in a signature scheme with appendix and can be invoked in the scheme DLSSA as part of signature verification.

Input:

- The DL domain parameters q, r, and g associated with the key w
- The signer's public key w
- The signature to be verified, which is a pair of integers (c, d)

Assumptions: Public key w and DL domain parameters q, r, and g are valid and associated with each other.

Output: The message representative, which is an integer f such that $0 \leq f < r$; or "invalid"

Operation: The message representative f shall be computed by the following or an equivalent sequence of steps:

1. If c is not in $[1, r - 1]$ or d is not in $[0, r - 1]$, output "invalid" and stop.
2. Compute a field element $j = \exp(g, d) \times \exp(w, c)$.
3. Convert the field element j to an integer i with primitive FE2IP.
4. Compute an integer $f = c - i \bmod r$.
5. Output f as the message representative.

Conformance region recommendation: A conformance region should include the following:

- At least one valid set of DL domain parameters q, r, and g.
- At least one valid public key w for each set of domain parameters.
- All purported signatures (c, d) that can be input to the implementation; this should include at least all (c, d) such that c and d are in the range $[0, r - 1]$, where r is from the domain parameters of w.

6.2.7 DLSP-DSA

DLSP-DSA is the Discrete Logarithm Signature Primitive, DSA version. It is based on the work in Kravitz [B97]. It can be invoked in a scheme to compute a signature on a message representative with the private key of the signer. The message representative cannot be recovered from the signature, but DLVP-DSA can be used in the scheme DLSSA to verify the signature.

Input:

- The DL domain parameters q, r, and g associated with the key s
- The signer's private key s
- The message representative, which is an integer $f \geq 0$

Assumptions: Private key s and DL domain parameters q, r, and g are valid and associated with each other; $f \geq 0$.

Output: The signature, which is a pair of integers (c, d), where $1 \leq c < r$ and $1 \leq d < r$

Operation: The signature (c, d) shall be computed by the following or an equivalent sequence of steps:

1. Generate a key pair (u, v) with the same set of domain parameters as the private key s (see the note below).
2. Convert v into an integer i with primitive FE2IP [recall that v is an element of $GF(q)$].
3. Compute an integer $c = i \bmod r$. If $c = 0$, then go to step 1.
4. Compute an integer $d = u^{-1}(f + sc) \bmod r$. If $d = 0$, then go to step 1.
5. Output the pair (c, d) as the signature.

Conformance region recommendation: A conformance region should include:

- At least one valid set of DL domain parameters q, r, and g
- At least one valid private key s for each set of domain parameters

— A range of message representatives f; this should include at least all $f \geq 0$ with bit length no greater than that of r, where r is from the domain parameters of s

NOTE—The key pair in step 1 should be a one-time key pair, which is generated and stored by the signer following the security recommendations of D.3.1, D.4.1.2, D.6, and D.7. A new key pair should be generated for every signature. The one-time private key u should be discarded after step 4, as its recovery by an opponent can lead to the recovery of the private key s.

6.2.8 DLVP-DSA

DLVP-DSA is the Discrete Logarithm Verification Primitive, DSA version. It is based on the work in Kravitz [B97]. This primitive verifies whether the message representative and the signature are consistent, given the key and the domain parameters. It can be invoked in the scheme DLSSA as part of signature verification.

Input:

— The DL domain parameters q, r, and g associated with the key w

— The signer's public key w

— The message representative, which is an integer $f \geq 0$

— The signature to be verified, which is a pair of integers (c, d)

Assumptions: Public key w and DL domain parameters q, r, and g are valid and associated with each other; $f \geq 0$.

Output: "Valid" if f and (c, d) are consistent given the key and the domain parameters; "invalid" otherwise

Operation: The output shall be computed by the following or an equivalent sequence of steps:

1. If c is not in $[1, r - 1]$ or d is not in $[1, r - 1]$, output "invalid" and stop.
2. Compute integers $h = d^{-1} \bmod r$; $h_1 = f\,h \bmod r$; $h_2 = c\,h \bmod r$.
3. Compute a field element $j = \exp(g, h_1) \times \exp(w, h_2)$.
4. Convert the field element j to an integer i with primitive FE2IP.
5. Compute an integer $c' = i \bmod r$.
6. If $c' = c$, then output "valid"; else, output "invalid."

Conformance region recommendation: A conformance region should include:

— At least one valid set of DL domain parameters q, r, and g.

— At least one valid public key w for each set of domain parameters.

— All message representatives $f \geq 0$ that can be input to the implementation; this should include at least all f with bit length no greater than that of r, where r is from the domain parameters of w.

— All purported signatures (c, d) that can be input to the implementation; this should include at least all (c, d), such that c and d are in the range $[1, r - 1]$, where r is from the domain parameters of w.

7. Primitives based on the elliptic curve discrete logarithm problem

This clause specifies the family of cryptographic primitives based on the discrete logarithm problem over elliptic curve groups, also known as the EC family. For background information on this family, see A.9.3, A.9.4, Koblitz [B94], and Miller [B117].

7.1 The EC setting

7.1.1 Notation

The list below introduces the notation used in Clause 7, Clause 9, and Clause 10. It is meant as a reference guide only; for complete definitions of the terms listed, refer to the appropriate text. Some other symbols are also used occasionally and are introduced in the text where appropriate.

q	The size of the underlying field used (part of the EC domain parameters)
a, b	The coefficients defining the elliptic curve E, elements of $GF\ (q)$ (part of the EC domain parameters)
E	The elliptic curve over the field $GF\ (q)$ defined by a and b
$\#E$	The number of points on the elliptic curve E
r	The prime divisor of $\#E$ and the order of G (part of the EC domain parameters)
G	A curve point generating a subgroup of order r (part of the EC domain parameters)
k	$\#E/r$, the cofactor
s, u, s', u'	EC private keys, integers, corresponding to public keys W, V, W', V', respectively
W, V, W', V'	EC public keys, points on the curve, corresponding to private keys s, u, s', u', respectively
(s, W), (u, V)	EC key pairs, where s and u are the private keys, and W and V are the corresponding public keys
z, z_1, z_2	Shared secret values, elements of $GF\ (q)$, derived by secret value derivation primitives
K	Shared secret key agreed upon by a key agreement scheme
(c, d)	Signature, a pair of integers, computed by a signature primitive
f	Message representative, an integer, computed by a message-encoding operation
M	The message, an octet string, whose signature is computed by a signature scheme

NOTES

1—When keys from two parties are involved in a primitive or scheme, the symbols s, u, W, V are used to denote the party's own keys, and the symbols s', u', W', V' are used to denote the other party's keys.

2—Multiplication of field elements, multiplication of integers, and scalar multiplication of elliptic curve points by integers are denoted by ×, although this symbol may be omitted when such omission does not cause ambiguity. Addition of field elements, addition of integers, and addition of elliptic curve points are denoted by +. Elliptic curve points are generally denoted by capital letters; for an elliptic curve point $P \neq O$, its x-coordinate and y-coordinate are denoted by x_P and y_P, respectively: $P = (x_P, y_P)$.

3—Throughout this clause, operations on finite field elements, integers, and elliptic curve points are used. Care needs to be exercised to distinguish among these, as operations are denoted by the same symbols regardless of operands. (See 5.3 for more information on finite fields, and 5.4 for more information on elliptic curves.)

7.1.2 EC domain parameters

EC domain parameters are used in every EC primitive and scheme and are an implicit component of every EC key. A set of EC domain parameters specifies a field $GF(q)$, where q is a positive odd prime integer p or 2^m for some positive integer m; two elliptic curve coefficients a and b, elements of $GF(q)$, that define an elliptic curve E; a positive prime integer r dividing the number of points on E; and a curve point G of order r. (G is called the generator of a subgroup of order r.) If $q = 2^m$, it also specifies a representation for the elements of $GF(q)$ to be used by the conversion primitives (see 5.3 and 5.5). Implicitly, it also specifies the cofactor $k = \#E/r$ (where $\#E$ is the number of points on E). If key validation is to be performed, or if the ECSVDP-DHC or ECSVDP-MQVC primitive is to be applied, then it shall also be the case that GCD $(k, r) = 1$ (i.e., r does not divide k; see A.1.1).

Depending on the scheme and protocol used, a party may need to generate its own set of domain parameters or use domain parameters provided by another party. If parameters are provided by another party, their authenticity may need to be determined (as discussed in D.3.2), and they may need to be validated (see below). The security issues related to domain parameter generation are discussed in D.3.1 and D.4.2. A suggested method for generating EC domain parameters is contained in A.16.7 (see also A.9.5).

Parties establish EC domain parameters as part of key management for a scheme. Depending on the key management technique, it is possible that the established domain parameters do not satisfy the intended definition, even though they have the same general form (i.e., components q, r, G, and optionally k). To accommodate this possibility, the term “EC domain parameters” shall be understood in this standard as referring to instances of the general form for EC domain parameters. The term *valid EC domain parameters* shall be reserved for EC domain parameters satisfying the definition.

Domain parameter validation is the process of determining whether a set of EC domain parameters is valid. Further discussion of domain parameter validation is contained in D.3.3. A suggested algorithm for EC domain parameter validation is contained in A.16.8.

There may be more than one set of domain parameters used in a primitive or scheme; the sets of EC domain parameters may be different for different keys, or they may be the same, depending on the requirements of a primitive or scheme. Unlike keys, which are not meant to be shared among users, a set of domain parameters can, and sometimes needs to be, shared. EC domain parameters are often public; the security of the schemes in this standard does not rely on these domain parameters being secret.

7.1.3 EC key pairs

For a given set of EC domain parameters, an *EC key pair* consists of an *EC private key s*, which is an integer in the range $[1, r - 1]$ and an *EC public key W*, which is a point on E, where $W = s\,G$. (Note that $W \neq O$, since G has order r and $0 < s < r$.) For security, EC private keys need to be generated so that they are unpredictable, and stored so that they are inaccessible to an adversary. EC private keys may be stored in any form convenient to the application.

EC key pairs are closely associated with their domain parameters, and can only be used in the context of the domain parameters. A key pair shall not be used with a set of domain parameters that is different from the one for which it was generated. A set of domain parameters may be shared by a number of key pairs (see D.4.2.2 and D.4.2.4, Note 1).

An EC key pair may or may not be generated by the party using it, depending on the trust model. This and other security issues related to EC key pair generation are discussed in D.3.1 and D.4.2. A suggested method for generating EC key pairs is contained in A.16.9.

As is the case for EC domain parameters, parties establish EC keys as part of key management for a scheme; and, depending on the key management technique, it is possible that an established key does not satisfy the intended definition for the key, even though it has the same general form. Accordingly, the terms "EC public key" and "EC private key" shall be understood in this standard as referring to instances of the general form for the key, and the terms *valid EC public key*, *valid EC private key,* and *valid EC key pair* shall be reserved for keys satisfying the definition.

Key validation is the process of determining whether a key is valid. Further discussion of key validation is contained in D.3.3. A suggested algorithm for EC public-key validation is contained in A.16.10. In some cases (when the ECSVDP-DHC or ECSVDP-MQVC primitive is applied), it may be necessary to verify only that the public key is a point on the curve (rather than a stronger condition that it is a non-zero multiple of G). The algorithm to verify that is also contained in A.16.10. No algorithm is given for EC private-key validation, because, generally, a party controls its own private key and need not validate it. However, private-key validation may be performed if desired.

7.2 Primitives

This subclause describes primitives used in the EC family. Before proceeding with this subclause, the reader should be familiar with the material of Clause 4.

As detailed in Clause 4 and Annex B, an implementation of a primitive may make certain assumptions about its inputs, as listed with the specification for each primitive. For example, if EC domain parameters q, a, b, r, and G and a public key W are inputs to a primitive, the implementation may generally assume that the domain parameters are valid (i.e., that G has order r on the elliptic curve defined by a and b, and $W = s\,G$ for some integer s in the interval $[1, r – 1]$). The behavior of an implementation is unconstrained in the case that W is not a multiple of G and, in such a case, the implementation may or may not output an error condition. It is up to the properly implemented scheme to ensure that only appropriate inputs are passed to a primitive, or to accept the risks of passing inappropriate inputs. For more on conforming with a primitive, see Annex B.

7.2.1 ECSVDP-DH

ECSVDP-DH is the Elliptic Curve Secret Value Derivation Primitive, Diffie-Hellman version. It is based on the work in Diffie and Hellman [B47], Koblitz [B94], and Miller [B117]. This primitive derives a shared secret value from one party's private key and another party's public key, where both have the same set of EC domain parameters. If two parties correctly execute this primitive, they will produce the same output. This primitive can be invoked by a scheme to derive a shared secret key; specifically, it may be used with the schemes ECKAS-DH1 and DL/ECKAS-DH2. It assumes that the input keys are valid (see also 7.2.2).

Input:

— The EC domain parameters q, a, b, r, and G associated with the keys s and W' (the domain parameters shall be the same for both s and W')

— The party's own private key s

— The other party's public key W'

Assumptions: Private key s, EC domain parameters q, a, b, r, and G, and public key W' are valid; both keys are associated with the domain parameters.

Output: The derived shared secret value, which is a field element $z \in GF(q)$; or "error"

Operation: The shared secret value z shall be computed by the following or an equivalent sequence of steps:

1. Compute an elliptic curve point $P = s\,W'$.
2. If $P = \mathcal{O}$, output "error" and stop.
3. Let $z = x_P$, the x-coordinate of P.
4. Output z as the shared secret value.

Conformance region recommendation: A conformance region should include the following:

— At least one valid set of EC domain parameters q, a, b, r, and G

— At least one valid private key s for each set of domain parameters

— All valid public keys W' associated with the same set of domain parameters as s

NOTES

1—This primitive does not address small subgroup attacks (see D.5.1.6), which may occur when the public key W' is not valid. To prevent them, a key agreement scheme should validate the public key W' before executing this primitive (see also 7.2.2).

2—When the public key W' and the private key s are valid, W' has order r and s is in the interval $[1, r-1]$. Thus, the point P in this case is not the element $\mathcal{O}$.

7.2.2 ECSVDP-DHC

ECSVDP-DHC is the Elliptic Curve Secret Value Derivation Primitive, Diffie-Hellman version with cofactor multiplication. It is based on the work of Diffie and Hellman [B47], Kaliski [B88], Koblitz [B94], Law et al. [B98], and Miller [B117]. This primitive derives a shared secret value from one party's private key and another party's public key, where both have the same set of EC domain parameters. If two parties correctly execute this primitive, they will produce the same output. This primitive can be invoked by a scheme to derive a shared secret key; specifically, it may be used with the schemes ECKAS-DH1 and DL/ECKAS-DH2. It does not assume the validity of the input public key (see also 7.2.1).

Input:

— The EC domain parameters q, a, b, r, G, and the cofactor associated with the keys s and W' (the domain parameters shall be the same for both s and W')

— The party's own private key s

— The other party's public key W'

— An indication as to whether compatibility with ECSVDP-DH is desired

Assumptions: Private key s, EC domain parameters q, a, b, r, G, and k are valid; the private key is associated with the domain parameters; W' is on the elliptic curve defined by a and b over $GF(q)$; GCD $(k, r) = 1$.

Output: The derived shared secret value, which is a field element $z \in GF(q)$; or "invalid public key"

Operation: The shared secret value z shall be computed by the following or an equivalent sequence of steps:

1. If compatibility with ECSVDP-DH is desired, then compute an integer $t = k^{-1}s \bmod r$; otherwise set $t = s$.

2. Compute an elliptic curve point $P = kt\, W'$.
3. If $P = O$, output "invalid public key" and stop.
4. Let $z = x_P$, the x-coordinate of P.
5. Output z as the shared secret value.

Conformance region recommendation: A conformance region should include the following:

- At least one valid set of EC domain parameters q, a, b, r, G, and k
- At least one valid private key s for each set of domain parameters
- All points W' on the curve over $GF\ (q)$ defined by a and b, where q, a, and b are from the domain parameters of s
- Compatibility with ECSVDP-DH may be preset by the implementation, or given as an input flag

NOTES

1—This primitive addresses small subgroup attacks, which may occur when the public key W' is not valid (see D.5.1.6). A key agreement scheme only needs to validate that W' is on the elliptic curve defined by a and b over $GF\ (q)$ before executing this primitive (see also 7.2.1).

2—The cofactor k depends only on the EC domain parameters. Hence, it can be computed once for a given set of domain parameters and stored as part of the domain parameters. Similarly, in the compatibility case, the value k^{-1} can be computed once and stored with the domain parameters, and the integer t can be computed once for a given private key s. Algorithms for computing or verifying the cofactor are included in A.12.3.

3—When the public key W' and the private key s are valid, the point P will be an element of a subset of the elliptic curve that consists of all the multiples of G (except for the element O). As a consequence, z will always be defined in this case. When the public key is invalid, the output will be either "invalid public key" or an element of order r on the elliptic curve; in particular, it will not be in a small subgroup.

4—In the compatibility case, ECSVDP-DHC computes the same output for valid keys as ECSVDP-DH, so an implementation that conforms with ECSVDP-DHC in the compatibility case also conforms with ECSVDP-DH.

7.2.3 ECSVDP-MQV

ECSVDP-MQV is the Elliptic Curve Secret Value Derivation Primitive, Menezes-Qu-Vanstone version. It is based on the work of Law et al. [B98]. This primitive derives a shared secret value from one party's two key pairs and another party's two public keys, where all the keys and values have the same set of EC domain parameters. If two parties correctly execute this primitive, they will produce the same output. This primitive can be invoked by a scheme to derive a shared secret key; specifically, it may be used with the scheme ECKAS-MQV. It assumes that the input keys are valid (see also 7.2.4).

In this primitive, let $h = \lceil (\log_2 r) / 2 \rceil$. Note that h depends only on the EC domain parameters and, hence, can be computed once for a given set of domain parameters.

Input:

- The EC domain parameters q, a, b, r, and G associated keys s, (u, V), W', V' (the domain parameters shall be the same for these keys)
- The party's own first private key s
- The party's own second key pair (u, V), where $V = (x, y)$
- The other party's first public key W'
- The other party's second public key $V' = (x', y')$

Assumptions: Private key s, key pair (u, V), public keys W', V', and EC domain parameters q, a, b, r, and G are valid; all the keys are associated with the domain parameters.

Output: The derived shared secret value, which is a nonzero field element $z \in GF(q)$; or "error"

Operation: The shared secret value z shall be computed by the following or an equivalent sequence of steps:

1. Convert x into an integer i using FE2IP.
2. Compute an integer $t = i \bmod 2^h$. Let $t = t + 2^h$.
3. Convert x' into an integer i' using FE2IP.
4. Compute an integer $t' = i' \bmod 2^h$. Let $t' = t' + 2^h$.
5. Compute an integer $e = ts + u \bmod r$.
6. Compute an elliptic curve point $P = e(V' + t'W')$.
7. If $P = O$, output "error" and stop.
8. Let $z = x_P$, the x-coordinate of P.
9. Output z as the shared secret value.

Conformance region recommendation: A conformance region should include the following:

- At least one valid set of EC domain parameters q, a, b, r, and g
- At least one valid private key s for each set of domain parameters
- All valid key pairs (u, V) associated with the same set of domain parameters as s
- All valid public keys W' and V' associated with the same set of domain parameters as s

NOTES

1—This primitive does not address small subgroup attacks (see D.5.1.6), which may occur when the public keys W' and V' are not valid. To prevent them, a key agreement scheme should validate the public keys W' and V' before executing this primitive (see also 7.2.4).

2—When the public keys W' and V' are valid, the point P will be an element of a subset of the elliptic curve that consists of all the multiples of G. It is possible (although very unlikely) that even though all the keys and parameters are valid, P will be the point O. In this case, the primitive will output "error," and will need to be rerun with a different input. Except for this rare case, z will always be defined as long as the input keys and parameters are valid.

7.2.4 ECSVDP-MQVC

ECSVDP-MQVC is the Elliptic Curve Secret Value Derivation Primitive, Menezes-Qu-Vanstone version with cofactor multiplication. It is based on the work of Kaliski [B88] and Law et al. [B98]. This primitive derives a shared secret value from one party's two key pairs and another party's two public keys, where all the keys and values have the same set of EC domain parameters. If two parties correctly execute this primitive, they will produce the same output. This primitive can be invoked by a scheme to derive a shared secret key; specifically, it may be used with the scheme ECKAS-MQV. It does not assume the validity of the input public keys (see also 7.2.3).

In this primitive, let $h = \lceil (\log_2 r) / 2 \rceil$. Note that h depends only on the EC domain parameters and, hence, can be computed once for a given set of domain parameters.

Input:

- The EC domain parameters q, a, b, r, G, and the cofactor k associated with keys s, (u, V), W', V' (the domain parameters shall be the same for these keys)
- The party's own first private key s
- The party's own second key pair (u, V), where $V = (x, y)$
- The other party's first public key W'
- The other party's second public key $V' = (x', y')$
- An indication as to whether compatibility with ECSVDP-MQV is desired

Assumptions: Private key s, key pair (u, V), and EC domain parameters q, a, b, r, $G,$ and k are valid; private key s and key pair (u, V) are associated with the domain parameters; W' and V' are on the elliptic curve defined by a and b over $GF(q)$; GCD $(k, r) = 1$.

Output: The derived shared secret value, which is a nonzero field element $z \in GF(q)$; or "invalid public key"

Operation: The shared secret value z shall be computed by the following or an equivalent sequence of steps:

1. Convert x into an integer i using FE2IP.
2. Compute an integer $t = i \bmod 2^h$. Let $t = t + 2^h$.
3. Convert x' into an integer i' using FE2IP.
4. Compute an integer $t' = i' \bmod 2^h$. Let $t' = t' + 2^h$.
5. If compatibility with ECSVDP-MQV is desired, then compute an integer $e = k^{-1}(ts + u) \bmod r$; otherwise, compute an integer $e = ts + u \bmod r$.
6. Compute an elliptic curve point $P = ke\,(V' + t'W')$.
7. If P = $\mathcal{O}$, output "invalid public key" and stop.
8. Let $z = x_P$, the x-coordinate of P.
9. Output z as the shared secret value.

Conformance region recommendation: A conformance region should include the following:

- At least one valid set of EC domain parameters q, a, b, r, g, and k
- At least one valid private key s for each set of domain parameters
- All valid key pairs (u, V) associated with the same set of domain parameters as s
- All points W' and V' on the curve over $GF(q)$ defined by a and b, where q, a, and b are from the domain parameters of s
- Compatibility with ECSVDP-MQV may be preset by the implementation, or given as an input flag

NOTES

1—This primitive addresses small subgroup attacks, which may occur when the public keys W' and V' are not valid (see D.5.1.6). A key agreement scheme only needs to validate that W' and V' are in $GF(q)$ before executing this primitive (see also 7.2.3).

2—The cofactor k depends only on the EC domain parameters. Hence, it can be computed once for a given set of domain parameters and stored as part of the domain parameters. Similarly, in the compatibility case, the value k^{-1} can be computed once and stored with the domain parameters. An equivalent way to compute the integer e in this case is as $(tk^{-1}s + k^{-1}u)$ mod r, where $k^{-1}s$ mod r and $k^{-1}u$ mod r can be computed once for each given private key s and u. Algorithms for computing or verifying the cofactor are included in A.12.3.

3—When the public key W' is valid, the point P will be an element of a subset of the elliptic curve that consists of all the multiples of G. It is possible (although very unlikely) that even though all the keys and parameters are valid, P will be the point O, and the primitive will output "invalid public key." In this case, the primitive will need to be rerun with a different input. Except for this rare case, z will always be defined as long as the input keys and parameters are valid. When one of the public keys is invalid, the output will be either "invalid public key" or the x coordinate of a point of order r on the elliptic curve; in particular, it will not be the x coordinate of a point in a small subgroup.

4—In the compatibility case, ECSVDP-MQVC computes the same output for valid keys as ECSVDP-MQV, so an implementation that conforms with ECSVDP-MQVC in the compatibility case also conforms with ECSVDP-MQV.

7.2.5 ECSP-NR

ECSP-NR is the Elliptic Curve Signature Primitive, Nyberg-Rueppel version. It is based on the work of Koblitz [B94], Miller [B117], and Nyberg and Rueppel [B120]. It can be invoked in a scheme to compute a signature on a message representative with the private key of the signer, in such a way that the message representative can be recovered from the signature using the public key of the signer by the ECVP-NR primitive. Note, however, that no EC signature schemes with message recovery are defined in this version of the standard (see C.3.4). ECSP-NR may also be used in a signature scheme with appendix, and can be invoked in the scheme DLSSA as part of signature generation.

Input:

- The EC domain parameters q, a, b, r, and G associated with the key s
- The signer's private key s
- The message representative, which is an integer f such that $0 \leq f < r$

Assumptions: Private key s and EC domain parameters q, a, b, r, and G are valid and associated with each other; $0 \leq f < r$.

Output: The signature, which is a pair of integers (c, d), where $1 \leq c < r$ and $0 \leq d < r$

Operation: The signature (c, d) shall be computed by the following or an equivalent sequence of steps:

1. Generate a key pair (u, V) with the same set of domain parameters as the private key s (see the note below). Let $V = (x_V, y_V)$ ($V \neq \mathsf{O}$ because V is a public key).
2. Convert x_V into an integer i with primitive FE2IP [recall that x_V is an element of $GF(q)$].
3. Compute an integer $c = i + f \bmod r$. If $c = 0$, then go to step 1.
4. Compute an integer $d = u - sc \bmod r$.
5. Output the pair (c, d) as the signature.

Conformance region recommendation: A conformance region should include the following:

- At least one valid set of EC domain parameters q, a, b, r, and G
- At least one valid private key s for each set of domain parameters
- All message representatives f in the range $[0, r - 1]$, where r is from the domain parameters of s

NOTE—The key pair in step 1 should be a one-time key pair, which is generated and stored by the signer following the security recommendations of D.3.1, D.4.2.2, D.6, and D.7. A new key pair should be generated for every signature. The one-time private key u should be discarded after step 4, as its recovery by an opponent can lead to the recovery of the private key s.

7.2.6 ECVP-NR

ECVP-NR is the Elliptic Curve Verification Primitive, Nyberg-Rueppel version. It is based on the work of Koblitz [B94], Miller [B117], and Nyberg and Rueppel [B120]. This primitive recovers the message representative that was signed with ECSP-NR, given only the signature and public key of the signer. It can be invoked in a scheme as part of signature verification and, possibly, message recovery. Note, however, that no EC signature schemes with message recovery are defined in this version of the standard (see C.3.4). ECVP-NR may also be used in a signature scheme with appendix, and can be invoked in the scheme DLSSA as part of signature verification.

Input:

- The EC domain parameters q, a, b, r, and G associated with the key W
- The signer's public key W
- The signature to be verified, which is a pair of integers (c, d)

Assumptions: Public key W and EC domain parameters q, a, b, r, and G are valid and associated with each other.

Output: The message representative, which is an integer f such that $0 \le f < r$; or "invalid"

Operation: The message representative f shall be computed by the following or an equivalent sequence of steps:

1. If c is not in $[1, r - 1]$ or d is not in $[0, r - 1]$, output "invalid" and stop.
2. Compute an elliptic curve point $P = d\ G + c\ W$. If $P = \mathcal{O}$, output "invalid" and stop; otherwise, $P = (x_P, y_P)$.
3. Convert the field element x_P to an integer i with primitive FE2IP.
4. Compute an integer $f = c - i \bmod r$.
5. Output f as the message representative.

Conformance region recommendation: A conformance region should include the following:

- At least one valid set of EC domain parameters q, a, b, r, and G
- At least one valid public key W for each set of domain parameters
- All purported signatures (c, d) that can be input to the implementation; this should include at least all (c, d), such that c and d are in the range $[0, r - 1]$, where r is from the domain parameters of W

7.2.7 ECSP-DSA

ECSP-DSA is the Elliptic Curve Signature Primitive, DSA version. It is based on the work of Koblitz [B94], Kravitz [B97], and Miller [B117]. It can be invoked in a scheme to compute a signature on a message representative with the private key of the signer. The message representative cannot be recovered from the signature, but ECVP-DSA can be used in the scheme ECSSA to verify the signature.

Input:

- The EC domain parameters q, a, b, r, and G associated with the key s
- The signer's private key s
- The message representative, which is an integer $f \geq 0$

Assumptions: Private key s and EC domain parameters q, a, b, r, and G are valid and associated with each other; $f \geq 0$.

Output: The signature, which is a pair of integers (c, d), where $1 \leq c < r$ and $1 \leq d < r$

Operation: The signature (c, d) shall be computed by the following or an equivalent sequence of steps:

1. Generate a key pair (u, V) with the same set of domain parameters as the private key s (see the note below). Let $V = (x_V , y_V)$ ($V \neq O$ because V is a public key).
2. Convert x_V into an integer i with primitive FE2IP [recall that x_V is an element of $GF(q)$].
3. Compute an integer $c = i \bmod r$. If $c = 0$, then go to step 1.
4. Compute an integer $d = u^{-1}(f + sc) \bmod r$. If $d = 0$, then go to step 1.
5. Output the pair (c, d) as the signature.

Conformance region recommendation: A conformance region should include the following:

- At least one valid set of EC domain parameters q, a, b, r, and G
- At least one valid private key s for each set of domain parameters
- A range of message representatives f; this should include at least all $f \geq 0$ with bit length no greater than that of r, where r is from the domain parameters of s

NOTE—The key pair in step 1 should be a one-time key pair, which is generated and stored by the signer following the security recommendations of D.3.1, D.4.2.2, D.6, and D.7. A new key pair should be generated for every signature. The one-time private key u should be discarded after step 4, as its recovery by an opponent can lead to the recovery of the private key s.

7.2.8 ECVP-DSA

ECVP-DSA is the Elliptic Curve Verification Primitive, DSA version. It is based on the work of Koblitz [B94], Kravitz [B97], and Miller [B117]. This primitive verifies whether the message representative and the signature are consistent given the key and the domain parameters. It can be invoked in the scheme ECSSA as part of signature verification.

Input:

- The EC domain parameters q, a, b, r, and G associated with the key W
- The signer's public key W
- The message representative, which is an integer $f \geq 0$
- The signature to be verified, which is a pair of integers (c, d)

Assumptions: Public key W and EC domain parameters q, a, b, r, and G are valid and associated with each other; $f \geq 0$.

Output: “Valid” if f and (c, d) are consistent, given the key and the domain parameters; “invalid” otherwise

Operation: The output shall be computed by the following or an equivalent sequence of steps:

1. If c is not in $[1, r - 1]$ or d is not in $[1, r - 1]$, output “invalid” and stop.
2. Compute integers $h = d^{-1} \bmod r$; $h_1 = f\,h \bmod r$; $h_2 = c\,h \bmod r$.
3. Compute an elliptic curve point $P = h_1G + h_2W$. If $P = \mathsf{O}$, output “invalid” and stop; otherwise, $P = (x_P, y_P)$.
4. Convert the field element x_P to an integer i with primitive FE2IP.
5. Compute an integer $c' = i \bmod r$.
6. If $c' = c$, then output “valid”; else, output “invalid.”

Conformance region recommendation: A conformance region should include the following:

- At least one valid set of EC domain parameters q, a, b, r, and G.
- At least one valid public key W for each set of domain parameters.
- All message representatives $f \geq 0$ that can be input to the implementation; this should include at least all f with bit length no greater than that of r, where r is from the domain parameters of W.
- All purported signatures (c, d) that can be input to the implementation; this should include at least all (c, d) such that c and d are in the range $[1, r - 1]$, where r is from the domain parameters of W.

8. Primitives based on the integer factorization problem

This clause specifies the family of cryptographic primitives based on the integer factorization problem, also known as the IF family. For background information on this family, see A.1.3, A.1.4, and Rivest, Shamir, and Adleman [B129].

There are two types of primitives in this family. While they are largely similar, they are based on slightly different keys and operations. One type is known as RSA, for “Rivest-Shamir-Adleman” (see Rivest, Shamir, Adleman [B129]); the other is known as RW, for “Rabin-Williams” (see Rabin [B128] and Williams [B149]).

8.1 The IF setting

8.1.1 Notation

The list below introduces the notation used in Clause 8, Clause 10, and Clause 11. It is meant as a reference guide only; for complete definitions of the terms listed, refer to the appropriate text. Some other symbols are also used occasionally and are introduced in the text where appropriate.

n The modulus, part of the keys

p, q The two primes whose product is the modulus n

e The public exponent, part of a public key (e is odd for RSA; e is even for RW)

d The private exponent, part of some of the representations of a private key

d_1	$d \bmod (p - 1)$, part of one of the representations of a private key
d_2	$d \bmod (q - 1)$, part of one of the representations of a private key
c	$q^{-1} \bmod p$, part of one of the representations of a private key
(n, e)	An RSA or RW public key
K	An RSA or RW private key
s	Signature, an integer, computed by a signature primitive
g	Encrypted message, an integer, computed by an encryption primitive
f	Message representative, an integer, computed by a message-encoding operation
M	The message, an octet string, which is encrypted in an encryption scheme or whose signature is computed by a signature scheme

NOTE—Multiplication of integers is denoted by ×, although this symbol may be omitted when such omission does not cause ambiguity. When typographically convenient, notation such as exp(f, e), where f and e are integers, will be used to denote raising f to the power e. The more traditional notation f^e may also be used.

8.1.2 Domain parameters in the IF family

Unlike the DL or EC families, the IF family has no domain parameters. The keys contain all the necessary information within themselves. While DL or EC primitives and schemes sometimes require domain parameters to be common to a group of keys, no similar requirement exists for any IF primitive or scheme. Of course, as with other families, parties need to be aware of each other's capabilities in order to work together. For example, some parties may have restrictions on the size of the modulus they can use or generate, while others may work only with a particular fixed public exponent (see Note 5 in D.4.3.4).

8.1.3 Keys in the IF family

Since there are two types of IF primitives, there are also two types of IF keys. They are similar, but the distinctions between them are significant.

8.1.3.1 RSA key pairs

An RSA *public key* consists of a *modulus* n, which is the product of two odd positive prime integers p and q, and an odd *public exponent* e ($3 \le e < n$), which is an integer relatively prime to $(p - 1)$ and $(q - 1)$. The corresponding RSA *private exponent* is an integer d ($1 \le d < n$) such that

$$d\,e \equiv 1 \pmod{\mathrm{LCM}\,(p - 1, q - 1)}$$

Note that there may be more than one private exponent d corresponding to a public key (n, e).

An RSA *private key K* may have multiple representations. The use of one of the following three representations is recommended in this standard:

1. A pair (n, d)
2. A quintuple (p, q, d_1, d_2, c), where $d_1 = d \bmod (p - 1)$, $d_2 = d \bmod (q - 1)$, and $c = q^{-1} \bmod p$
3. A triple (p, q, d)

8.1.3.2 RW key pairs

An RW *public key* consists of a *modulus n*, which is the product of two odd positive prime integers p and q, such that $p \not\equiv q \pmod 8$, and an even *public exponent e* ($2 \le e < n$), which is an integer relatively prime to $(p-1)(q-1)/4$. [Note that these conditions imply that $p \equiv q \equiv 3 \pmod 4$; moreover, one of the primes is congruent to 3 (mod 8) and the other is congruent to 7 (mod 8).] The corresponding RW *private exponent* is an integer d ($1 \le d < n$) such that

$$d\,e \equiv 1 \ (\mathrm{mod\ LCM}\ (p-1, q-1) / 2)$$

Note that there may be more than one private exponent d corresponding to a public key (n, e).

An RW *private key K* may have multiple representations. The use of one of the following three representations is recommended in this standard:

1. A pair (n, d)
2. A quintuple (p, q, d_1, d_2, c), where $d_1 = d \bmod (p-1)$, $d_2 = d \bmod (q-1)$, and $c = q^{-1} \bmod p$
3. A triple (p, q, d)

8.1.3.3 Considerations common to RSA and RW key pairs

An RSA or RW key pair may or may not be generated by the party using it, depending on the trust model. This and other security issues related to RSA and RW key pair generation are discussed in D.3.1 and D.4.3. A suggested method for generating RSA key pairs is contained in A.16.11. A suggested method for generating RW key pairs is contained in A.16.12.

Parties establish RSA and RW keys as part of key management for a scheme. Depending on the key management technique, it is possible that an established key does not satisfy the intended definition for the key, even though it has the same general form (e.g., components n and e). To accommodate this possibility, the terms *RSA public key*, *RW public key*, *RSA private key*, and *RW private key* shall be understood in this standard as referring to instances of the general form for the key. The terms *valid RSA public key*, *valid RW public key*, *valid RSA private key*, *valid RW private key*, *valid RSA key pair*, and *valid RW key pair* shall be reserved for keys satisfying the definitions.

Key validation is the process of determining whether a key is valid. Further discussion of key validation is contained in D.3.3, although no algorithm for IF public-key validation is suggested in Annex A. No algorithm is given for IF private-key validation, because, generally, a party controls its own private key and need not validate it. However, private-key validation may be performed if desired.

For security, the primes p and q need to be generated so that they are unpredictable and inaccessible to an adversary. Whatever form a private key is represented in, it needs to be stored so that it is inaccessible to an adversary. The compromise of p, q, d, d_1, d_2, or c will aid the adversary in recovering the private key. These values should be discarded if not used.

Three representations are provided for IF private keys because of performance tradeoffs between the representations. Use of the quintuple representation tends to result in faster performance for larger sizes of n.

8.2 Primitives

This subclause describes primitives used in the IF family. Before proceeding with this subclause, the reader should be familiar with the material of Clause 4.

As detailed in Clause 4 and Annex B, an implementation of a primitive may make certain assumptions about its inputs, as listed with the specification of the primitive. For example, if an RSA public key (n, e) is an input to a primitive, an implementation may assume that the public key is valid [i.e., that n is a product of two odd primes, $3 \leq e < n$, and e is relatively prime to $(p - 1)(q - 1)$]. The behavior of an implementation is not specified in the case where the input does not satisfy these conditions and, in such a case, the implementation may or may not output an error condition. It is up to the properly implemented scheme to ensure that only appropriate inputs are passed to a primitive, or to accept the risks of passing inappropriate inputs. For more on conforming with a primitive, see Annex B.

The primitives described in this subclause can be combined into four pairs: message representatives encrypted with IFEP-RSA can be decrypted with IFDP-RSA; message representatives signed with IFSP-RSA1, IFSP-RSA2, or IFSP-RW can be recovered with IFVP-RSA1, IFVP-RSA2, or IFVP-RW, respectively. While IFEP-RSA is mathematically identical to IFVP-RSA1, and IFDP-RSA is mathematically identical to IFSP-RSA1, these primitives are used for entirely different purposes and should not be confused. The three signature primitives are similar, but have some important differences (see C.3.6). To aid in defining these primitives, the operation “IF Private-Key Operation” is defined first.

8.2.1 IF private-key operation

IF Private-Key Operation is not a primitive in itself, but rather is used in some of the primitives below. The operation produces the same result, independent of the representation of the private key.

Input:

— An RSA or RW private key K represented as one of the following:

A pair (n, d)
A quintuple (p, q, d_1, d_2, c)
A triple (p, q, d)

— An integer i such that $0 \leq i < n$ (where $n = pq$ if the second or the third representation of K is used), to which the private key operation is to be applied

Assumptions: Private key K is valid; $0 \leq i < n$.

Output: An integer j such that $0 \leq j < n$, the result of the private key operation on i

Operation: The integer j shall be computed by the following or an equivalent sequence of steps:

```
I.  If the first representation $(n, d)$ of $K$ is used
    1. Let $j = \exp(i, d) \bmod n$.
    2. Output $j$.
II. If the second representation $(p, q, d_1, d_2, c)$ of $K$ is used
    1. Let $j_1 = \exp(i, d_1) \bmod p$.
    2. Let $j_2 = \exp(i, d_2) \bmod q$.
    3. Let $h = c\,(j_1 - j_2) \bmod p$.
    4. Let $j = j_2 + h\,q$.
    5. Output $j$.
```

III. If the third representation (p, q, d) of K is used

1. Let $j_1 = \exp(i, d \bmod (p - 1)) \bmod p$.
2. Let $j_2 = \exp(i, d \bmod (q - 1)) \bmod q$.
3. Let $h = q^{-1} \times (j_1 - j_2) \bmod p$.
4. Let $j = j_2 + h\, q$.
5. Output j.

8.2.2 IFEP-RSA

IFEP-RSA is RSA the Encryption Primitive. It is based on the work of Rivest, Shamir, and Adleman [B129]. It is invoked in the scheme IFES as part of encrypting a message, given the message and the public key of the intended recipient. The message can be decrypted within a scheme by invoking IFDP-RSA.

Input:

— The recipient's RSA public key (n, e)

— The message representative, which is an integer f such that $0 \le f < n$

Assumptions: Public key (n, e) is valid; $0 \le f < n$.

Output: The encrypted message representative, which is an integer g such that $0 \le g < n$

Operation: The encrypted message representative g shall be computed by the following or an equivalent sequence of steps:

1. Let $g = \exp(f, e) \bmod n$.
2. Output g.

Conformance region recommendation: A conformance region should include the following:

— At least one valid RSA public key (n, e)

— All message representatives f in the range $[0, n - 1]$

8.2.3 IFDP-RSA

IFDP-RSA is the RSA Decryption Primitive. It is based on the work of Rivest, Shamir, and Adleman [B129]. It is used in the scheme IFES as part of decrypting a message encrypted with the use of IFEP-RSA, given the encrypted message representative and the private key of the recipient.

Input:

— The recipient's RSA private key K

— The encrypted message representative, which is an integer g such that $0 \le g < n$

Assumptions: Private key K is valid; $0 \le g < n$.

Output: The message representative, which is an integer f such that $0 \le f < n$

Operation: The message representative f shall be computed by the following or an equivalent sequence of steps:

1. Perform the IF Private-Key Operation described in 8.2.1 on K and g to produce an integer f.
2. Output f.

Conformance region recommendation: A conformance region should include the following:

— At least one valid RSA private key K

— All encrypted message representatives g in the range $[0, n - 1]$, where n is from the private key K

8.2.4 IFSP-RSA1

IFSP-RSA1 is the RSA Signature Primitive, version 1. It is based on the work of Rivest, Shamir, and Adleman [B129]. It can be invoked in a scheme to compute a signature on a message representative with the private key of the signer, in such a way that the message representative can be recovered from the signature using the public key of the signer by the IFVP-RSA1 primitive. Note, however, that no IF signature schemes with message recovery are defined in this version of the standard (see C.3.7). IFSP-RSA1 may also be used in a signature scheme with appendix, and can be invoked in the scheme IFSSA as part of signature generation.

Input:

— The signer's RSA private key K

— The message representative, which is an integer f such that $0 \leq f < n$

Assumptions: Private key K is valid; $0 \leq f < n$.

Output: The signature, which is an integer s such that $0 \leq s < n$

Operation: The signature s shall be computed by the following or an equivalent sequence of steps:

1. Perform the IF Private-Key Operation described in 8.2.1 on K and f to produce an integer s.
2. Output s.

Conformance region recommendation: A conformance region should include the following:

— At least one valid RSA private key K

— All message representatives f in the range $[0, n - 1]$, where n is from the private key K

8.2.5 IFVP-RSA1

IFVP-RSA1 is the RSA Verification Primitive, version 1. It is based on the work of Rivest, Shamir, and Adleman [B129]. This primitive recovers the message representative that was signed with IFSP-RSA1, given only the signature and public key of the signer. It can be invoked in a scheme as part of signature verification and, possibly, message recovery. Note, however, that no IF signature schemes with message recovery are defined in this version of the standard (see C.3.7). IFVP-RSA1 may also be used in a signature scheme with appendix, and can be invoked in the scheme IFSSA as part of signature verification.

Input:

- The signer's RSA public key (n, e)
- The signature to be verified, which is an integer s

Assumptions: Public key (n, e) is valid.

Output: The message representative, which is an integer f such that $0 \leq f < n$; or "invalid"

Operation: The message representative f shall be computed by the following or an equivalent sequence of steps:

1. If s is not in $[0, n - 1]$, output "invalid" and stop.
2. Let $f = \exp(s, e) \bmod n$.
3. Output f.

Conformance region recommendation: A conformance region should include the following:

- At least one valid RSA public key (n, e)
- All purported signatures s that can be input to the implementation; this should include at least all s in the range $[0, n - 1]$

8.2.6 IFSP-RSA2

IFSP-RSA2 is the RSA Signature Primitive, version 2. It is based on the work of ISO/IEC 9796:1991 [B78] and Rivest, Shamir, and Adleman [B129]. Its output is at least one bit shorter than the RSA modulus n. It can be invoked in a scheme to compute a signature on a message representative of a certain form with the private key of the signer, in such a way that the message representative can be recovered from the signature using the public key of the signer by the IFVP-RSA2 primitive. Note, however, that no IF signature schemes with message recovery are defined in this version of the standard (see C.3.7). IFSP-RSA2 may also be used in a signature scheme with appendix, and can be invoked in the scheme IFSSA as part of signature generation.

Input:

- The signer's RSA private key K
- The message representative, which is an integer f such that $0 \leq f < n$, and $f \equiv 12 \pmod{16}$

Assumptions: Private key K is valid, $0 \leq f < n$, and f is congruent to 12 modulo 16.

Output: The signature, which is an integer s such that $0 \leq s < n/2$

Operation: The signature s shall be computed by the following or an equivalent sequence of steps:

1. Perform the IF Private-Key Operation described in 8.2.1 on K and f to produce an integer t.
2. Let $s = \min(t, n - t)$.
3. Output s.

Conformance region recommendation: A conformance region should include the following:

- At least one valid RSA private key K
- All message representatives f in the range $[0, n - 1]$ such that $f \equiv 12 \pmod{16}$, where n is from the private key K

8.2.7 IFVP-RSA2

IFVP-RSA2 is the RSA Verification Primitive, version 2. It is based on the work of ISO/IEC 9796:1991 [B78] and Rivest, Shamir, and Adleman [B129]. This primitive recovers the message representative that was signed with IFSP-RSA2, given only the signature and the public key of the signer. It can be invoked in a scheme as part of signature verification and, possibly, message recovery. Note, however, that no IF signature schemes with message recovery are defined in this version of the standard (see C.3.7). IFVP-RSA2 may also be used in a signature scheme with appendix, and can be invoked in the scheme IFSSA as part of signature verification.

Input:

- The signer's RSA public key (n, e)
- The signature to be verified, which is an integer s

Assumptions: Public key (n, e) is valid.

Output: The message representative, which is an integer f such that $0 \le f < n$ and $f \equiv 12 \pmod{16}$; or "invalid"

Operation: The message representative f shall be computed by the following or an equivalent sequence of steps:

1. If s is not in $[0, (n - 1) / 2]$, output "invalid" and stop.
2. Compute $t = \exp(s, e) \bmod n$.
3. If $t \equiv 12 \pmod{16}$, then let $f = t$.
4. Else let $f = n - t$. If $f \not\equiv 12 \pmod{16}$, output "invalid" and stop.
5. Output f.

Conformance region recommendation: A conformance region should include the following:

- At least one valid RSA public key (n, e)
- All purported signatures s that can be input to the implementation; this should include at least all s in the range $[0, (n - 1)/2]$

8.2.8 IFSP-RW

IFSP-RW is the RW Signature Primitive. It is based on the work of ISO/IEC 9796:1991 [B78], Rabin [B128], and Williams [B149]. Its output is at least one bit shorter than the RW modulus n. It can be invoked in a scheme to compute a signature on a message representative with the private key of the signer, in such a way that the message representative can be recovered from the signature using the public key of the signer by the IFVP-RW primitive. Note, however, that no IF signature schemes with message recovery are defined in this version of the standard (see C.3.7). IFSP-RW may also be used in a signature scheme with appendix, and can be invoked in the scheme IFSSA as part of signature generation.

Input:

- The recipient's RW private key K
- The message representative, which is an integer f such that $0 \leq f < n$ and $f \equiv 12 \pmod{16}$

Assumptions: Private key K is valid, $0 \leq f < n$, and f is congruent to 12 modulo 16.

Output: The signature, which is an integer s such that $0 \leq s < n/2$

Operation: The signature s shall be computed by the following or an equivalent sequence of steps:

1. If the Jacobi symbol $\left(\frac{f}{n}\right) = +1$, let $u = f$.
2. Else let $u = f/2$ (recall that f is even).
3. Perform the IF Private-Key Operation described in 8.2.1 on K and u to produce an integer t.
4. Let $s = \min(t, n - t)$.
5. Output s.

Conformance region recommendation: A conformance region should include the following:

- At least one valid RW private key K
- All message representatives f in the range $[0, n - 1]$ such that $f \equiv 12 \pmod{16}$, where n is from the private key K

NOTE—See A.2.9 for a potentially more efficient implementation of the primitive, and A.1.4 and A.2.3 for evaluating Jacobi symbols.

8.2.9 IFVP-RW

IFVP-RW is the RW Verification Primitive. It is based on the work of ISO/IEC 9796:1991 [B78], Rabin [B128], and Williams [B149]. This primitive recovers the message representative that was signed with IFSP-RW, given only the signature and the public key of the signer. It can be invoked in a scheme as part of signature verification and, possibly, message recovery. Note, however, that no IF signature schemes with message recovery are defined in this version of the standard (see C.3.7). IFVP-RW may also be used in a signature scheme with appendix, and can be invoked in the scheme IFSSA as part of signature verification.

Input:

- The signer's RW public key (n, e)
- The signature to be verified, which is an integer s

Assumptions: Public key (n, e) is valid.

Output: The message representative, which is an integer f such that $0 \leq f < n$ and $f \equiv 12 \pmod{16}$; or "invalid"

Operation: The message representative f shall be computed by the following or an equivalent sequence of steps.

1. If s is not in $[0, (n - 1) / 2]$, output "invalid" and stop.
2. Compute $t_1 = \exp(s, e) \bmod n$.

3. Compute $t_2 = n - t_1$.
4. If $t_1 \equiv 12 \pmod{16}$, then let $f = t_1$.
5. Else if $t_1 \equiv 6 \pmod{8}$, then let $f = 2t_1$.
6. Else if $t_2 \equiv 12 \pmod{16}$, then let $f = t_2$.
7. Else if $t_2 \equiv 6 \pmod{8}$, then let $f = 2t_2$.
8. Else output “invalid” and stop.
9. Output f.

Conformance region recommendation: A conformance region should include the following:

- At least one valid RW public key (n, e)
- All purported signatures s that can be input to the implementation; this should include at least all s in the range $[0, (n - 1)/2]$

9. Key agreement schemes

The general model for key agreement schemes is given in 9.1. Three specific schemes and their allowable options are given in 9.2, 9.3, and 9.4.

9.1 General model

In a key agreement scheme, each party combines its own private key(s) with the other party’s public key(s) to come up with a secret key. Other (public or private) information known to both parties may also enter the scheme as *key derivation parameters*. If the parties use the corresponding keys and identical key derivation parameters, and the scheme is executed correctly, the parties will arrive at the same secret key (see Note 1 below). A key agreement scheme can allow two parties to derive shared secret keys without any prior shared secret.

A key agreement scheme consists of a key agreement operation, along with supporting key management. Domain parameter and key pair generation for the key agreement schemes are specified further in connection with the DL and EC families (Clause 6 and Clause 7, respectively). Security considerations for key agreement schemes are given in D.5.1.

A key agreement operation has the following form for all the schemes:

1. Establish one or more sets of valid domain parameters with which the parties’ key pairs shall be associated.
2. Select one or more valid private keys (and, in some cases, their corresponding public keys) for the operation, associated with the domain parameters established in step 1 (see Note 2 below).
3. Obtain one or more other party’s purported public keys for the operation (see Note 3 below).
4. (*Optional*) Depending on the cryptographic operations in step 5, choose an appropriate method to validate the public keys and the domain parameters. If any validation fails, output “invalid” and stop. (see Note 4 below).
5. Apply certain cryptographic operations to the private and public keys to produce a shared secret value.

6. For each shared secret key to be agreed on, establish or agree on key derivation parameters, and derive a shared secret key from the shared secret value and the key derivation parameters using a key derivation function (see Note 5 below).

NOTES

1—(*Key confirmation*) By the definition of a key agreement scheme, if the correct keys are used and computation is performed properly, the shared secret keys computed by the two parties will also be the same. However, to verify the identities of the parties and to ensure that they indeed possess the same key, the parties may need to perform a key confirmation protocol. (See D.5.1.3 for more information.)

2—(*Repeated use of a key pair*) A given public/private key pair may be used by either party for any number of key agreement operations, depending on the implementation.

3—(*Authentication of ownership*) The process of obtaining the other party's public key (step 3) may involve authentication of ownership of the public key, as described in D.3.2 and D.5.1.5. This may be achieved by verifying a certificate, or by other means. The means by which the key (or the certificate containing it) is obtained may vary, and may include one party sending the key to the other, or a party obtaining the other party's key from a third party or from a local database.

4—(*Domain parameter and key validation*) Since a key agreement primitive assumes (with some exceptions, depending on the primitive chosen) that the domain parameters and the public key are valid, the result of the primitive is undefined otherwise. Consequently, it is recommended that parties validate the set(s) of domain parameters in step 1 and the public key(s) in step 4, unless the risk of operating on invalid domain parameters or keys is mitigated by other means, as discussed in D.3.3 and D.5.1.6. Examples of "other means" include validation within the supporting key management, or use of –DHC and –MQVC primitives together with simpler validation. In the case of key agreement, it is only necessary that each party is assured of domain parameter and public-key validity prior to use of the shared secret key. Therefore, while it may be more efficient to validate domain parameters once and then validate each key as it is used, there may be instances where domain parameter validation follows public-key validation. In other words, an implementation may perform domain parameter validation and public-key validation in a different order than assumed in the specification (step 1 followed by step 4), with equivalent effect, provided that both occur prior to use of the shared secret key.

5—(*Key derivation parameters*) Depending on the key derivation function, there may be security-related constraints on the set of allowed key derivation parameters. The interpretation of these parameters is left to the implementation. For instance, it may contain key-specific information, protocol-related public information, and supplementary, private information. For security, the interpretation should be unambiguous. (See D.5.1.4 for further discussion.)

6—(*Attributes of the shared secret key*) The attributes of the shared secret key depend on the particular key agreement scheme used, the attributes of the public/private key pairs, the nature of the parameters to the key derivation function, and whether or not key confirmation is performed. (See D.5.1 for further discussion of the attributes of the shared secret key.)

7—(*Error conditions*) The two parties may produce errors under certain conditions, such as the following:

—Private key not found in step 2
—Public key not found in step 3
—Public key not valid in step 4
—Private key or public key not supported in step 5
—Key derivation parameter not supported in step 6

Such error conditions should be detected and handled appropriately by an implementation; however, the specific methods for detecting and handling them are outside of the scope of this standard.

9.2 DL/ECKAS-DH1

DL/ECKAS-DH1 is the Discrete Logarithm and Elliptic Curve Key Agreement Scheme, Diffie-Hellman version, where each party contributes one key pair.

9.2.1 Scheme options

The following options shall be established or otherwise agreed upon between the parties to the scheme:

— A secret value derivation primitive, which shall be: DLSVDP-DH, DLSVDP-DHC, ECSVDP-DH, or ECSVDP-DHC

— For a -DHC secret value derivation primitive, an indication as to whether or not compatibility with the corresponding -DH primitive is desired

— A key derivation function, which should be KDF1 (see 13.1), or a function designated for use with DL/ECKAS-DH1 in an amendment to this standard

The above information may remain the same for any number of executions of the key agreement scheme, or it may be changed at some frequency. The information need not be kept secret.

9.2.2 Key agreement operation

A sequence of shared secret keys, K_1, K_2, …, K_t, shall be generated by each party by performing the following or an equivalent sequence of steps:

1. Establish the valid set of DL or EC domain parameters with which the parties' key pairs shall be associated.
2. Select a valid private key s for the operation, associated with the parameters established in step 1.
3. Obtain the other party's purported public key w' for the operation, associated with the parameters established in step 1.
4. (*Optional*) If the selected secret value derivation primitive is DLSVDP-DHC or ECSVDP-DHC, then validate that w′ is an element in the appropriate group (i.e., in GF (q) for DL or on the elliptic curve for EC; see 6.2.2 and 7.2.2); otherwise, validate that w' is a valid public key. If any validation fails, output "invalid public key" and stop.
5. Compute a shared secret value z from the private key s and the other party's public key w' with the selected secret value derivation primitive (see 9.2.1).
6. Convert the shared secret value z to an octet string Z using FE2OSP.
7. For each shared secret key to be agreed on
    a. Establish or otherwise agree on key derivation parameters P_i for the key.
    b. Derive a shared secret key K_i from the octet string Z and the key derivation parameters P_i with the selected key derivation function (see 9.2.1).

Conformance region recommendation: A conformance region should include the following:

— At least one valid set of domain parameters

— At least one valid private key s for each set of domain parameters

— All valid public keys w' associated with the same set of domain parameters as s; if key validation is performed or a –DHC primitive is used, invalid public keys that are appropriately handled by the implementation may also be included in the conformance region

— A range of key derivation parameters P

NOTE—These schemes, with appropriate restrictions on the scheme options and inputs, may be compatible with the techniques in ANSI X9.42 [B8] (in the DL case) and ANSI X9.63 [B12] (in the EC case).

9.3 DL/ECKAS-DH2

DL/ECKAS-DH2 is the Discrete Logarithm and Elliptic Curve Key Agreement Scheme, Diffie-Hellman version, where each party contributes two key pairs. Particular variants may be derived from the work of ANSI X9.42 [B8], Blake-Wilson, Johnson, and Menezes [B21], Goss [B67], Gunther [B68], Johnson [B85], and Matsumoto, Takashima, and Imai [B105].

9.3.1 Scheme options

The following options shall be established or otherwise agreed upon between the parties to the scheme:

- Two secret value derivation primitives, each of which (independently) shall be: DLSVDP-DH, DLSVDP-DHC, ECSVDP-DH, or ECSVDP-DHC
- For each -DHC secret value derivation primitive, an indication as to whether or not compatibility with the corresponding -DH primitive is desired
- If the two secret value derivation primitives are the same (and have the same compatibility option in the -DHC case), an indication whether compatibility with DL/ECKAS-DH1 is desired
- A key derivation function, which should be KDF1 (see 13.1), or a function designated for use with DL/ECKAS-DH2 in an amendment to this standard

The above information may remain the same for any number of executions of the key agreement function, or it may be changed at some frequency. The information need not be kept secret.

9.3.2 Key agreement operation

A sequence of shared secret keys, K_1, K_2, …, K_t, shall be generated by each party by performing the following or an equivalent sequence of steps:

1. Establish the valid set of DL or EC domain parameters with which the parties' first key pairs shall be associated.
2. Establish the valid set of DL or EC domain parameters with which the parties' second key pairs shall be associated.
3. Select valid first and second private keys s and u for the operation, associated with the domain parameters established in steps 1 and 2, respectively.
4. Obtain the other party's first and second purported public keys w' and v' for the operation, associated with the domain parameters established in steps 1 and 2, respectively.
5. (*Optional*) If the first selected secret value derivation primitive is DLSVDP-DHC or ECSVDP-DHC, then validate that w' is an element in the appropriate group (i.e., in $GF(q)$ for DL or on the elliptic curve for EC; see 6.2.2 and 7.2.2); otherwise, validate that w' is a valid public key. If the second selected secret value derivation primitive is DLSVDP-DHC or ECSDVP-DHC, then validate that v' is an element in the appropriate group; otherwise validate that v' is a valid public key. If any validation fails, output "invalid public key" and stop.
6. Compute a first shared secret value z_1 from s and w' with the first selected secret value derivation primitive (see 9.3.1). Convert z_1 to an octet string Z_1 using FE2OSP.
7. Compute a second shared secret value z_2 from u and v' with the second selected secret value derivation primitive (see 9.3.1). Convert z_2 to an octet string Z_2 using FE2OSP.
8. If compatibility with DL/ECKAS-DH1 is desired, the selected private keys s and u (and associated domain parameters) are the same, and the other party's public keys w' and v' (and associated domain parameters) are the same, then let $Z = Z_1$. Otherwise, let $Z = Z_1 \| Z_2$.

9. For each shared secret key to be agreed on:
    a. Establish or otherwise agree on key derivation parameters P_i for the key.
    b. Derive a shared secret key K_i from the octet string Z and the key derivation parameters P_i with the selected key derivation function (see 9.3.1).

Conformance region recommendation: A conformance region should include the following:

— At least one valid set of domain parameters for the first pair of keys.

— At least one valid private key s for each set of domain parameters for the first pair of keys.

— All valid public keys w' associated with the same set of domain parameters as s.

— At least one valid set of domain parameters for the second pair of keys.

— At least one valid private key u for each set of domain parameters for the second pair of keys.

— All valid public keys v' associated with the same set of domain parameters as u.

— A range of key derivation parameters P.

— If key validation is performed or a -DHC primitive is used, invalid public keys that are appropriately handled by the implementation may also be included in the conformance region.

NOTES

1—If the two secret value derivation primitives are the same, compatibility with DL/ECKAS-DH1 is selected, and each party contributes two identical key pairs, then DL/ECKAS-DH2 will compute the same output as DL/ECKAS-DH1 with the same primitive and key pairs.

2—This scheme, with appropriate restrictions on the scheme options and inputs, may be compatible with techniques in ANSI X9.42 [B8] (in the DL case) and ANSI X9.63 [B12] (in the EC case).

9.4 DL/ECKAS-MQV

DL/ECKAS-MQV is the Discrete Logarithm and Elliptic Curve Key Agreement Scheme, Menezes-Qu-Vanstone version. Each party contributes two key pairs.

9.4.1 Scheme options

The following options shall be established or otherwise agreed upon between the parties to the scheme:

— A secret value derivation primitive, which shall be: DLSVDP-MQV, DLSVDP-MQVC, ECSVDP-MQV, or ECSVDP-MQVC

— For an -MQVC secret value derivation primitive, an indication as to whether or not compatibility with the corresponding -MQV primitive is desired

— A key derivation function, which should be KDF1 (see 13.1), or a function designated for use with DL/ECKAS-MQV in an amendment to this standard

The above information may remain the same for any number of executions of the key agreement scheme, or it may be changed at some frequency. The information need not be kept secret.

9.4.2 Key agreement operation

A sequence of shared secret keys, K_1, K_2, …, K_t, shall be generated by each party by performing the following or an equivalent sequence of steps:

1. Establish the valid set of DL or EC domain parameters with which the parties' two key pairs shall be associated.
2. Select a valid private key s and a valid key pair (u, v) for the operation, associated with the parameters established in step 1.
3. Obtain the other party's two purported public keys w' and v' for the operation, associated with the parameters established in step 1.
4. (*Optional*) If the selected secret value derivation primitive is DLSVDP-MQVC or ECSVDP-MQVC, then validate that w' and v' are elements in the appropriate group (i.e., in $GF(q)$ for DL or on the elliptic curve for EC; see 6.2.4 and 7.2.4); otherwise, validate that w' and v' are valid public keys. If any validation fails, output "invalid public key" and stop.
5. Compute a shared secret value z from the selected private keys s and u and the other party's two public keys w' and v' with the selected secret value derivation primitive (see 9.4.1).
6. Convert the shared secret value z to an octet string Z using FE2OSP.
7. For each shared secret key to be agreed on
    a. Establish or otherwise agree on key derivation parameters P_i for the key.
    b. Derive a shared secret key K_i from the octet string Z and the key derivation parameters P_i with the selected key derivation function (see 9.4.1).

Conformance region recommendation: A conformance region should include the following:

— At least one valid set of domain parameters.

— At least one valid private key s for each set of domain parameters.

— All valid key pairs (u, v) associated with the same set of domain parameters as s.

— All valid public keys w' and v' associated with the same set of domain parameters as s; if key validation is performed or an -MQVC primitive is used, invalid public keys w' and v' that are appropriately handled by the implementation may also be included in the conformance region.

— A range of key derivation parameters P.

NOTE—These schemes, with appropriate restrictions on the scheme options and inputs, may be compatible with techniques in ANSI X9.42 [B8] (in the DL case) and ANSI X9.63 [B12] (in the EC case).

10. Signature schemes

The general model for signature schemes is given in 10.1. Two specific schemes and their allowable options are given in 10.2 and 10.3.

10.1 General model

In a signature scheme, one party, the *signer*, generates a signature for a message with its own private key, and another party, the *verifier*, verifies the signature with the signer's corresponding public key. A signature scheme can provide assurance of the origin and integrity of a message, and can protect against impersonation of parties and modification of messages.

There are two types of signature schemes. In a *signature scheme with appendix*, the signer conveys both the message and the signature to the verifier, who verifies their consistency. In a *signature scheme with message recovery*, the signer conveys only the signature to the verifier, who verifies the signature and, if it is correct, recovers the message from it. This standard defines only signature schemes with appendix (see C.3.4 and C.3.7).

A signature scheme consists of a signature generation operation and a signature verification operation, along with supporting key management. Domain parameter and key pair generation for the signature schemes are specified further in connection with the DL, EC, and IF families (Clause 6, Clause 7, and Clause 8, respectively). Security considerations for signature schemes are given in D.5.2.

A signature generation operation has the following form for all the schemes:

1. Select a valid private key (together with its set of domain parameters, if any) for the operation.
2. Apply certain cryptographic operations to the message and the private key to produce a signature.
3. Output the signature.

A signature verification operation in a signature scheme with appendix has the following form:

1. Obtain the signer's purported public key (together with its set of domain parameters, if any) for the operation (Note 1 below).
2. (*Optional*) Validate the public key and its associated set of domain parameters, if any. If validation fails, output "invalid" and stop (Note 2 below).
3. Apply certain cryptographic operations to the message, the signature, and the public key to verify the signature.
4. Output "valid" or "invalid" according to the result of step 3.

NOTES

1—(*Authentication of ownership*) The process of obtaining the signer's public key (step 1 of verification) may involve authentication of ownership of the public key, as further described in D.3.2 and D.5.2.4. This may be achieved by verifying a certificate or by other means. The means by which the key (or the certificate containing it) is obtained may vary, and may include the signer sending the key to the verifier, or the verifier obtaining the key from a third party or from a local database.

2—(*Domain parameter and key validation*) Since a signature verification primitive assumes that the domain parameters and the public key are valid, the result of the primitive is undefined otherwise. Consequently, it is recommended that the verifier validate the public key (and its associated set of domain parameters, if any) in step 3, unless the risk of operating on invalid domain parameters or keys is mitigated by other means, as discussed in D.3.3 and D.5.2.5. Examples of "other means" include validation within the supporting key management, or assignment of liability to a purported signer for all signatures that can be verified with the signer's public key, whether or not the public key is valid.

3—(*Error conditions*) The signer's and verifier's steps may produce errors under certain conditions, such as the following:

—Private key not found in signer's step 1
—Message or private key not supported in signer's step 2
—Public key not found in verifier's step 1
—Public key not valid in verifier's step 2
—Message, signature, or public key not supported in verifier's step 3

Such error conditions should be detected and handled appropriately by an implementation; however, the specific methods for detecting and handling them is outside of the scope of this standard.

4—(*Repudiation*) A dishonest signer may be interested in the ability to claim that something went wrong and the signature was not authentic, in order to disclaim the liability for the signature. This is known as *repudiation*. (See D.5.2.3 for more on repudiation and ways to address it.)

10.2 DL/ECSSA

DL/ECSSA is the Discrete Logarithm and Elliptic Curve Signature Scheme with Appendix.

10.2.1 Scheme options

The following options shall be established or otherwise agreed upon between the parties to the scheme (the signer and the verifier):

— The signature and verification primitives, which shall be one of the following pairs of primitives: DLSP-NR and DLVP-NR, DLSP-DSA and DLVP-DSA, ECSP-NR and ECVP-NR, or ECSP-DSA and ECVP-DSA

— The message-encoding method for signatures with appendix, which should be EMSA1 (see 12.1.1), or a technique designated for use with DL/ECSSA (and the selected signature primitive) in an amendment to this standard

The above information may remain the same for any number of executions of the signature scheme, or it may be changed at some frequency. The information need not be kept secret.

10.2.2 Signature generation operation

A signature (c, d) shall be generated by a signer from a message M by the following or an equivalent sequence of steps:

1. Select a valid private key s and its associated set of domain parameters for the operation.
2. If the selected signature primitive is DLSP-NR or ECSP-NR, then set the maximum length of the message representative to be l = (length of r in bits) – 1, where r is the order of the base point in the DL or EC set of domain parameters.
3. If the selected signature primitive is DLSP-DSA or ECSP-DSA, then set the maximum length of the message representative to be l = length of r in bits.
4. Use the encoding operation of the selected message-encoding method (see 10.2.1) to produce a message representative f of maximum length l from the message M (f will be a non-negative integer).
5. Apply the selected signature primitive (see 10.2.1) to the integer f and the private key s to generate the signature (c, d).
6. Output the signature.

Conformance region recommendation: A conformance region should include the following:

— At least one valid set of domain parameters

— At least one valid private key s for each set of domain parameters

— A range of messages M

10.2.3 Signature verification operation

A signature (c, d) on a message M shall be verified by a verifier by the following or an equivalent sequence of steps:

1. Obtain the signer’s purported public key w and its associated set of domain parameters for the operation.

2. (*Optional*) Validate the public key w and its associated set of domain parameters. Output "invalid" and stop if validation fails.
3. If the selected verification primitive is DLVP-NR or ECVP-NR
    a. Apply the selected verification primitive (see 10.2.1) to the signature (c, d) and the signer's public key to recover an integer f. If the output of the primitive is "invalid," output "invalid" and stop.
    b. Set the maximum length of the message representative to be l = (length of r in bits) – 1, where r is the order of the base point in the DL or EC set of domain parameters.
    c. Use the verification operation of the selected message-encoding method (see 10.2.1) to verify that the integer f is a correct encoded representative of the message M according to the encoding method and maximum length l. If that is the case, output "valid"; otherwise, output "invalid."
4. If the selected verification primitive is DLVP-DSA or ECVP-DSA
    a. Set the maximum length of the message representative to be l = length of r in bits, where r is the order of the base point in the DL or EC set of domain parameters.
    b. Use the decoding operation of the selected message-encoding method (see 10.2.1) to produce a message representative f of maximum length l from the message M (f will be a non-negative integer).
    c. Apply the selected verification primitive (see 10.2.1) to the integer f, the signature (c, d), and the signer's public key to determine whether they are consistent. If the output of the primitive is "invalid," output "invalid;" otherwise, output "valid."

Conformance region recommendation: A conformance region should include the following:

— At least one valid set of domain parameters.

— At least one valid public key w for each set of domain parameters; if key validation is performed, invalid public keys w that are appropriately rejected by the implementation may also be included in the conformance region.

— All messages M that can be input to the implementation.

— All purported signatures (c, d) that can be input to the implementation; this should at least include all (c, d) such that c and d are in the range $[1, r – 1]$ for -DSA or $[0, r – 1]$ for -NR, where r is from the domain parameters of w.

NOTES

1—Since message-encoding methods for signatures with appendix are usually based on hash functions, the length of a message that can be signed by this scheme is usually unrestricted or constrained by a very large number.

2—Because of the way verification is performed when the verification primitive is DLVP-DSA or ECVP-DSA, the message-encoding method has to be deterministic when one of these primitives is used. Unlike verification using DLVP-NR or ECVP-NR, where the message representative is an output of the verification primitive, for DLVP-DSA and ECVP-DSA the message representative is one of the inputs to the verification primitive.

3—These schemes, with appropriate restrictions on the scheme options and inputs, may be compatible with techniques in ANSI X9.30:1-1997 [B4] (in the DL case), ANSI X9.62-1998 [B11] (in the EC case), and ISO/IEC DIS 14888-3 [B80].

10.3 IFSSA

IFSSA is the Integer Factorization Signature Scheme with Appendix.

10.3.1 Scheme options

The following options shall be established or otherwise agreed upon between the parties to the scheme (the signer and the verifier):

- The type of key pair for the signer (RSA or RW).
- The signature and verification primitives, which shall be a pair from the following list:

  If the signer's public key is an RSA public key, then the primitives shall be either the pair IFSP-RSA1 and IFVP-RSA1, or the pair IFSP-RSA2 and IFVP-RSA2.

  If the signer's public key is an RW public key, then the primitives shall be the pair IFSP-RW and IFVP-RW.

- The message-encoding method for signatures with appendix, which should be EMSA2 (see 12.1.2), or a technique designated for use with IFSSA in an amendment to this standard. (If the signature primitive is IFSP-RSA2 or IFSP-RW, the message-encoding method shall always produce a message representative congruent to 12 modulo 16.)

The above information may remain the same for any number of executions of the signature scheme, or it may be changed at some frequency. The information need not be kept secret.

10.3.2 Signature generation operation

A signature *s* shall be generated by a signer from a message *M* by the following or an equivalent sequence of steps:

1. Select a valid private key *K* for the operation.
2. Set the maximum length of the message representative to be l = (length of n in bits) – 1, where n is the modulus in the IF private key.
3. Use the encoding operation of the selected message-encoding method (see 10.3.1) to produce a message representative f of maximum length l from the message M (f will be a non-negative integer). If the selected signature primitive is IFSP-RSA2 or IFSP-RW, f will be congruent to 12 modulo 16.
4. Apply the selected signature primitive (see 10.3.1) to the integer f and the selected private key K to generate the signature s.
5. Output the signature.

Conformance region recommendation: A conformance region should include the following:

- At least one valid private key *K*
- A range of messages *M*

10.3.3 Signature verification operation

A signature *s* on a message *M* shall be verified by a verifier by the following or an equivalent sequence of steps:

1. Obtain the signer's purported public key (n, e).
2. (*Optional*) Validate the public key (n, e). Output "invalid" and stop if validation fails.
3. Apply the selected verification primitive (see 10.3.1) to the signature s and the signer's public key to recover an integer f. If the output of the primitive is "invalid," output "invalid" and stop.
4. Set the maximum length of the message representative to be l = (length of n in bits) – 1.
5. Verify that the integer f is a correct encoded representative of the message M using the verification operation of the selected message-encoding method (see 10.3.1) and maximum length l. If that is the case, output "valid;" otherwise, output "invalid."

Conformance region recommendation: A conformance region should include the following:

- At least one valid public key (n, e); if key validation is performed, invalid public keys (n, e) that are appropriately rejected by the implementation may also be included in the conformance region.
- All messages M that can be input to the implementation.
- All purported signatures s that can be input to the implementation; this should include at least all s in the range [0, n – 1] for IFVP-RSA1, or [0, (n – 1)/2] for IFVP-RSA2 and IFVP-RW.

NOTES

1—The upper bound on the length in bits of the message representative is so that the message representative is less than the modulus n, as required by all signature primitives. Since message-encoding methods for signatures with appendix are usually based on hash functions, the length of a message that can be signed by this scheme is usually unrestricted or constrained by a very large number.

2—This scheme, with appropriate restrictions on the scheme options and inputs, may be compatible with techniques in ANSI X9.31-1998 [B7] and ISO/IEC DIS 14888-3 [B80].

11. Encryption schemes

Discussion of encryption schemes, in general, is given in 11.1. A specific scheme and its allowable options are given in 11.2.

11.1 General model

In an encryption scheme, one party, the *sender*, generates a ciphertext for a message with another party's public key. The other party, called the *recipient*, decrypts the ciphertext with the corresponding private key to obtain the original message. An encryption scheme can provide confidentiality of a message.

In addition to providing confidentiality of a message, an encryption scheme may provide for a secure association between the message and a string known as *encoding parameters*. The encoding parameters themselves are not encrypted by the encryption scheme, but are securely associated with the ciphertext by the sender. The recipient, during decryption, may also supply encoding parameters and verify whether or not the sender used the same encoding parameters during encryption. The encoding parameters may be an empty string. See D.5.3.3 for more information on encoding parameters.

An encryption scheme consists of an encryption operation and a decryption operation, along with supporting key management. Domain parameter and key pair generation for the encryption scheme below are specified further in connection with the IF family (see Clause 8). Security considerations for encryption schemes are given in D.5.3.

Since there is only one encryption scheme in this standard, the general form of an encryption scheme, similar to those for key agreement and signature schemes, is not presented here.

11.2 IFES

IFES is the Integer Factorization Encryption Scheme.

11.2.1 Scheme options

The following options shall be established or otherwise agreed upon between the parties to the scheme (the sender and the recipient):

- The message-encoding method for encryption, which should be EME1 (see 12.2.1), or a technique designated for use with IFES in an amendment to this standard

The above information may remain the same for any number of executions of the encryption scheme, or it may be changed at some frequency. The information need not be kept secret.

11.2.2 Encryption operation

The ciphertext *g* shall be generated by a sender from a message *M* and encoding parameters *P* by the following or an equivalent sequence of steps:

1. Obtain the recipient's purported public key (n, e) for the operation (see Note 1 in 11.2.3).
2. (*Optional*) Validate the public key (n, e). Stop if validation fails (see Note 2 in 11.2.3).
3. Set the maximum length of the message representative to be l = (length of n in bits) – 1.
4. Use the encoding operation of the selected message-encoding method (see 11.2.1) to produce a message representative f of maximum length l from the message M and the encoding parameters P (f will be a non-negative integer). If the message is too long, the encoding method may not be able to produce a representative of the selected length. In that case, output "error" and stop.
5. Compute the ciphertext g from f and the recipient's public key with the primitive IFEP-RSA.

Conformance region recommendation: A conformance region should include the following:

- At least one valid RSA public key (n, e); if key validation is performed, invalid public keys (n, e) that are appropriately rejected by the implementation may also be included in the conformance region.
- A range of messages M and encoding parameters P (where the range of encoding parameters may include only the empty string).

11.2.3 Decryption operation

The plaintext *M* shall be recovered from the ciphertext *g* and encoding parameters *P* by the following or an equivalent sequence of steps:

1. Select the private key K for the operation.
2. Compute an integer f from g and the selected private key K with primitive IFDP-RSA.
3. Set the maximum length of the message representative to be l = (length of n in bits) – 1, where n is the modulus in the IF private key.

4. Decode the integer f according to the decoding operation of the selected message-encoding method (see 11.2.1), maximum length l, and the encoding parameters P to produce the message M. If the decoding operation produces an error, output “invalid”; otherwise, output the message M as the plaintext.

Conformance region recommendation: A conformance region should include the following:

- At least one valid RSA private key K
- All ciphertexts g in the range $[0, n – 1]$, where n is from the private key K
- A range of messages M and encoding parameters P (where the range of encoding parameters may include only the empty string)

NOTES

1—(*Authentication of ownership*) The process of obtaining the recipient’s public key (step 1 of encryption) may involve authentication of ownership of the public key, as described in D.3.2 and D.5.3.4. This may be achieved by verifying a certificate, or by other means. The means by which the key (or the certificate containing it) is obtained may vary, and may include the recipient sending the key to the sender, or the sender obtaining the key from a third party or from a local database.

2—(*Key validation*) Since the encryption primitive assumes that the public key is valid, the result of the primitive is undefined if the public key is invalid. Consequently, it is recommended that the sender validate the public key (and its associated set of domain parameters, if any) in step 2, unless the risk of operating on invalid domain parameters or keys is mitigated by other means, as discussed in D.3.3 and D.5.3.5. Examples of “other means” include validation within the supporting key management.

3—(*Error conditions*) The sender’s and recipient’s steps may produce errors under certain conditions, such as the following:

- Public key not found in sender’s step 1
- Public key not valid in sender’s step 2
- Message, encoding parameters, or public key not supported in sender’s steps 4 and 5
- Private key not found in recipient’s step 1
- Message, encoding parameters, or private key not supported in recipient’s steps 2 and 4.

Such error conditions should be detected and handled appropriately by an implementation; however, the specific methods for detecting and handling them are outside the scope of this standard.

4—(*Length of the message*) The upper bound on the length in bits of the encoded message representative is such that, when converted to an integer, it is less than the modulus n, as required by the encryption primitive. Since the message representative has to, at the very least, contain the information contained in the message itself, the length of messages that can be encrypted by this scheme is limited by the modulus length and the selected message-encoding method.

5—(*Compatibility*) This scheme, with appropriate restrictions on the scheme options and inputs, may be compatible with techniques in ANSI X9.44 [B9].

12. Message-encoding methods

This clause describes message-encoding methods used in this document as building blocks for schemes. Unlike cryptographic primitives and schemes, encoding methods do not use keys.

Each message-encoding method consists of an encoding operation, and of either a decoding or a verification operation. An encoding operation encodes the message to produce a non-negative integer, called a *message representative*. It takes the message, the maximum bit length l of the output, and possibly other parameters as input, and produces the integer of bit length no more than l (see 3.1 for the definition of bit length of an integer). A decoding operation gives back the original message given the message representative, the length l, and the same additional parameters as were passed to the encoding operation. A verification operation verifies whether the message representative is a valid encoding of a message given the message representative, the message, the length l, and the same parameters as were passed to the encoding operation.

Different message-encoding methods are needed for different categories of schemes. The use of an inadequate encoding method may compromise the security of the scheme in which it is used. This standard strongly recommends the use of the encoding methods contained in this clause, or any encoding methods defined in an amendment to this standard.

12.1 Message-encoding methods for signatures with appendix

This standard strongly recommends the use of one of the following message-encoding methods for signature schemes with appendix.

12.1.1 EMSA1

EMSA1 is an encoding method for signatures with appendix based on a hash function. It is recommended for use with DLSSA and ECSSA (see 10.2).

The method is parameterized by the following choice:

— A hash function *Hash* with output length *hLen* octets, which shall be SHA-1 (see 14.1.1), or RIPEMD-160 (see 14.1.2), or a technique designated for use with EMSA1 in an amendment to this standard

NOTE—EMSA1 cannot produce message representatives longer than the hash function output; for SHA-1 and RIPEMD-160, the maximum length is 160 bits.

12.1.1.1 Encoding operation

Input:

— A message, which is an octet string M (depending on the hash function chosen, there may be a limitation on the length of M; for SHA-1 and RIPEMD-160, the maximum length is $2^{61} - 1$ octets)

— The maximum bit length l of the message representative

Output: A message representative, which is an integer $f \geq 0$ of bit length at most min(l, 8*hLen*); or “error”

The message representative f shall be computed by the following or an equivalent sequence of steps:

1. If the length of M is greater than the length limitation ($2^{61} - 1$ octets for SHA-1 or RIPEMD-160), output “error” and stop.
2. Compute *Hash* (M) with the selected hash function to produce an octet string H of length *hLen* octets.
3. If $l < 8hLen$, convert H to a bit-string HB of length 8*hLen* bits with the primitive OS2BSP. Remove $8hLen - l$ rightmost bits of HB, and then convert the remaining l bits to an integer f using the primitive BS2IP.
4. Else convert H to an integer f using OS2IP.
5. Output f as the message representative.

12.1.1.2 Verification operation

Input:

- A message, which is an octet string *M* (depending on the hash function chosen, there may be a limitation on the length of *M*; for SHA-1 and RIPEMD-160, the maximum length is $2^{61} - 1$ octets)
- The maximum bit length l of the message representative
- The message representative, which is an integer $f \geq 0$

Output: "Valid" if f is a correct representative of *M*; "invalid" otherwise

The validity indicator shall be computed by the following or an equivalent sequence of steps:

1. Use the Encoding Operation of EMSA1 to compute a message representative g, a non-negative integer. If the encoding operation outputs "error," output "invalid" and stop.
2. If $f = g$, output "valid"; otherwise, output "invalid."

12.1.2 EMSA2

EMSA2 is an encoding method for signatures with appendix based on a hash function, with some additional formatting based on ANSI X9.31-1998 [B7]. It is recommended for use with IFSSA (see 10.3).

The method is parameterized by the following choice:

- A hash function *Hash* with output length 20 octets, which shall be SHA-1 (see 14.1.1) or RIPEMD-160 (see 14.1.2), or a technique designated for use with EMSA2 in an amendment to this standard

NOTE—EMSA2 cannot produce message representatives shorter than 191 bits in length.

12.1.2.1 Encoding operation

Input:

- A message, which is an octet string *M* (depending on the hash function chosen, there may be a limitation on the length of *M*; for SHA-1 and RIPEMD-160, the maximum length is $2^{61} - 1$ octets)
- The maximum bit length l of the message representative

Output: A message representative, which is an integer $f \geq 0$ of bit length at most l; or "error"

The message representative f shall be computed by the following or an equivalent sequence of steps:

1. If the length of *M* is greater than the length limitation ($2^{61} - 1$ octets for SHA-1 or RIPEMD-160), or $l < 191$, output "error" and stop.
2. Compute *Hash* (*M*) with the selected hash function to produce an octet string *H* of length 20 octets.
3. If *M* is of length 0 (i.e., an empty string), let P_1 be a single octet with hexadecimal value 4b. Otherwise, let P_1 be a single octet with hexadecimal value 6b.
4. Let P_2 be an octet string of length $\lfloor (l + 1) / 8 \rfloor - 24$ octets, each with hexadecimal value bb.

5. Let P_3 be a single octet with hexadecimal value 33 if the selected hash function is SHA-1, and let it be 31 if the selected hash function is RIPEMD-160.
6. Let $T = P_1 \| P_2 \| \text{ba} \| H \| P_3 \| \text{cc}$, where ba and cc are single octets represented in hexadecimal.
7. Convert T to an integer f using OS2IP.
8. Output f as the message representative.

12.1.2.2 Verification operation

Input:

- A message, which is an octet string M (depending on the hash function chosen, there may be a limitation on the length of M; for SHA-1 and RIPEMD-160, the maximum length is $2^{61} - 1$ octets)
- The maximum bit length l of the message representative
- The message representative, which is an integer $f \geq 0$

Output: "Valid" if f is a correct representative of M; "invalid" otherwise

The validity indicator shall be computed by the following or an equivalent sequence of steps:

1. If the length of M is greater than the length limitation ($2^{61} - 1$ octets for SHA-1 or RIPEMD-160), or $l < 191$, output "invalid" and stop.
2. Convert f to an octet string T of length $\lfloor (l + 1) / 8 \rfloor$ octets using the primitive I2OSP.
3. If M is of length 0 (i.e., an empty string), let P_1 be a single octet with hexadecimal value 4b. Otherwise, let P_1 be a single octet with hexadecimal value 6b.
4. Verify that the leftmost octet of T is equal to P_1, the next $\lfloor (l + 1) / 8 \rfloor - 24$ octets each have hexadecimal value bb, and the next octet after that has hexadecimal value ba. If not, output "invalid" and stop.
5. Verify that the second rightmost octet has hexadecimal value 33 if the selected hash function is SHA-1, and 31 if the selected hash function is RIPEMD-160. If not, output "invalid" and stop.
6. Verify that the rightmost octet has hexadecimal value cc. If not, output "invalid" and stop.
7. Remove the leftmost $\lfloor (l + 1) / 8 \rfloor - 22$ octets of T and the rightmost 2 octets of T to produce an octet string H' of length 20 octets.
8. Apply the selected hash function to M to produce an octet string H of length 20 octets.
9. If $H = H'$, output "valid"; otherwise, output "invalid."

12.2 Message-encoding methods for encryption

This standard strongly recommends the use of the following message-encoding method for encryption schemes.

12.2.1 EME1

EME1 is an encoding method for encryption based on the "enhanced Optimal Asymmetric Encryption Padding (OAEP)" (see Bellare and Rogaway [B18] and Johnson and Matyas [B86]), a hash function (14.1), and a mask generation function (14.2). It is recommended for use with IFES (11.2). It can produce message representatives of arbitrary length l; however, there are restrictions on the length of the message it can encode that depend on l.

The method is parameterized by the following choices:

- A hash function *Hash* with output length *hLen* octets, which shall be SHA-1 (14.1.1) or RIPEMD-160 (14.1.2), or a technique designated for use with EME1 in an amendment to this standard
- A mask generation function *G*, which shall be MGF1, or a technique designated for use with EME1 in an amendment to this standard

12.2.1.1 Encoding operation

Input:

- The maximum bit length l of the message representative.
- A message, which is an octet string *M* of length $mLen \le \lfloor (l/8) - 2hLen - 1 \rfloor$ octets.
- Encoding parameters, which are an octet string *P* (*P* may be an empty string) (see D.5.3.3 for possible uses). (Depending on the hash function chosen, there may be a limitation on the length of *P*; for SHA-1 and RIPEMD-160, the maximum length is $2^{61} - 1$ octets.)

Output: A message representative, which is an integer $f \ge 0$ of bit length at most l; or "error"

The message representative *f* shall be computed by the following or an equivalent sequence of steps:

1. If the length of *P* is greater than the length limitation ($2^{61} - 1$ octets for SHA-1 or RIPEMD-160), output "error" and stop.
2. Let $oLen = \lfloor l/8 \rfloor$ and $seedLen = hLen$. If $mLen > oLen - hLen - seedLen - 1$, then output "error" and stop.
3. Let *S* be the octet string that consists of $oLen - mLen - hLen - seedLen - 1$ zero octets. Let *T* be the octet string consisting of a single octet with hexadecimal value 01.
4. Let $M' = S \parallel T \parallel M$.
5. Apply the hash function *Hash* to the parameters *P* to produce an octet string *cHash* of length *hLen*.
6. Let $DB = cHash \parallel M'$. The length of *DB* is $oLen - seedLen$ octets.
7. Generate a fresh, random octet string *seed* of length *seedLen* octets. (See D.6 for more on generation of random strings.)
8. Apply the mask generation function *G* to the string *seed* to produce an octet string *dbMask* of length $oLen - seedLen$ octets.
9. Let $maskedDB = DB \oplus dbMask$.
10. Apply the mask generation function *G* to the string *maskedDB* to produce an octet string *seedMask* of length *seedLen* octets.
11. Let $maskedSeed = seed \oplus seedMask$.
12. Let $EM = maskedSeed \parallel maskedDB$.
13. Convert *EM* to an integer *f* with the primitive OS2IP.
14. Output *f* as the message representative.

12.2.1.2 Decoding operation

Input:

- The maximum bit length l of the message representative.
- The message representative, which is an integer $f \geq 0$.
- Encoding parameters, which are an octet string P (P may be an empty string) (see D.5.3.3 for possible uses). (Depending on the hash function chosen, there may be a limitation on the length of P; for SHA-1 and RIPEMD-160, the maximum length is $2^{61} - 1$ octets.)

Output: The message, which is an octet string M; or “error”

The message shall be decoded by the following or an equivalent sequence of steps:

1. If the length of P is greater than the length limitation ($2^{61} - 1$ octets for SHA-1 or RIPEMD-160), output “error” and stop.
2. Let $oLen = \lfloor l/8 \rfloor$ and $seedLen = hLen$. If $oLen < seedLen + hLen + 1$, output “error” and stop.
3. Convert f to an octet string EM of length $oLen$ with the primitive I2OSP. If the primitive outputs “error,” output “error” and stop.
4. Let $maskedSeed$ be the leftmost $seedLen$ octets of EM, and $maskedDB$ be the remaining octets.
5. Apply the mask generation function G to the string $maskedDB$ to produce an octet string $seedMask$ of length $seedLen$ octets.
6. Let $seed = maskedSeed \oplus seedMask$.
7. Apply the mask generation function G to the string $seed$ to produce an octet string $dbMask$ of length $oLen - seedLen$ octets.
8. Let $DB = maskedDB \oplus dbMask$.
9. Apply the hash function $Hash$ to the parameters P to produce an octet string $cHash$ of length $hLen$.
10. Compare the first $hLen$ octets of DB to $cHash$. If they are not equal, output “error” and stop.
11. Let M' be all but the first $hLen$ octets of DB.
12. Let T be the leftmost nonzero octet of M'. If $T \neq$ hexadecimal value 01, output “error” and stop.
13. Remove T and all octets to the left of it (which are all zero) from M' to produce an octet string M.
14. Output the message M.

13. Key derivation functions

This clause describes key derivation functions used in this standard as building blocks for key agreement schemes. A key derivation function computes one or more shared secret keys from shared secret value(s) and other mutually known parameters. The derived secret keys are usually used in symmetric cryptography.

The use of an inadequate key derivation function compromises the security of the key agreement scheme in which it is used. This standard strongly recommends the use of the key derivation functions contained in this clause, or any key derivation functions defined in subsequent amendments to this standard.

13.1 KDF1

KDF1 is a more general version of a similar key derivation function construction in ANSI X9.42 [B8]. It is recommended for use with DL and EC key agreement schemes (Clause 9).

The function is parameterized by the following choice:

— A hash function *Hash* with output length *hLen* octets, which shall be SHA-1 (14.1.1), or RIPEMD-160 (14.1.2), or a technique designated for use with KDF1 in an amendment to this standard

Input:

— A shared secret string, which is an octet string *Z* of length *zLen* octets.

— Key derivation parameters, which are an octet string *P* of length *pLen* octets. (Depending on the hash function chosen, there may be a limitation on *zLen* + *pLen*; for SHA-1 and RIPEMD-160, the maximum of *zLen* + *pLen* is $2^{61} - 1$.)

Output: A shared secret key, which is an octet string *K* of length *hLen* octets; or "error"

The shared secret key *K* shall be computed by the following or an equivalent sequence of steps:

1. If *zLen* + *pLen* exceeds the length limitation ($2^{61} - 1$ for SHA-1 or RIPEMD-160), output "error" and stop.
2. Compute *hash*(*Z* || *P*) with the selected hash function to produce an octet string *K* of *hLen* octets.
3. Output *K* as the shared secret key.

NOTE—The interpretation and usage of the *hLen*-octet output *K* is beyond the scope of this standard. For example, the two parties may agree to use the first 128 bits (with parity adjusted) of *K* as a session key for two-key triple-DES (ANSI X9.52-1998 [B10]).

14. Auxiliary functions

14.1 Hash functions

A hash function is a building block for many techniques described in Clause12, Clause 13, and Clause 14. It takes a variable-length octet string as input and outputs a fixed-length octet string. The length of the input to a hash function is usually unrestricted or constrained by a very large number. The output depends solely on the input—a hash function is deterministic.

14.1.1 SHA-1

SHA-1 is defined in FIPS PUB 180-1 [B55].

Input: An octet string *M* of length *mLen* octets (*mLen* has to be less than 2^{61}).

Output: A hash value, which is an octet string *H* of length 20 octets

The hash value shall be computed by the following or an equivalent sequence of steps:

1. Convert *M* to a bit string *MB* of length 8*mLen* bits with the primitive OS2BSP.
2. Apply the Secure Hash Algorithm, revision 1 (as described in FIPS PUB 180-1) to *MB* to produce a bit string *HB* of length 160 bits.
3. Convert *HB* to an octet string *H* with the primitive BS2OSP. The length of *H* will be 20 octets.
4. Output *H* as the hash value.

14.1.2 RIPEMD-160

RIPEMD-160 is based on the work of Dobbertin, Bosselaers, and Preneel [B49].

Input: An octet string *M* of length *mLen* octets (*mLen* has to be less than 2^{61}).

Output: A hash value, which is an octet string *H* of length 20 octets

The hash value shall be computed by the following or an equivalent sequence of steps:

1. Apply the RIPEMD-160 hash algorithm as described in Clause 7 of ISO/IEC 10118-3:1998 to *M* to produce a bit string *HB* of length 160 bits.
2. Convert *HB* to an octet string *H* with the primitive BS2OSP. The length of *H* will be 20 octets.
3. Output *H* as the hash value.

14.2 Mask generation functions

This subclause describes a mask generation function used in this standard as a building block for EME1. A mask generation function takes as input an octet string and the desired length of the output, and outputs an octet string of that length. The lengths of both the input to and the output of a mask generation function are usually unrestricted or constrained by a very large number. The output depends solely on the input—a mask generation function is deterministic.

14.2.1 MGF1

MGF1 is a mask generation function based on a hash function, following the ideas of Bellare and Rogaway [B18] and Bellare and Rogaway [B19]. It is used with EME1 (12.2.1).

The function is parameterized by the following choice:

— A hash function *Hash* with output length *hLen* octets, which shall be SHA-1 (14.1.1), or RIPEMD-160 (14.1.2), or a technique designated for use with MGF1 in an amendment to this standard

Input:

— An octet string *Z* of *zLen* octets (depending on the hash function chosen, there may be a limitation on *zLen*; for SHA-1 and RIPEMD-160, *zLen* shall be less than or equal to $2^{61} – 5$).

— The desired length of the output, which is a positive integer *oLen*. (*oLen* shall be less than or equal to *hLen* × 2^{32}).

Output: An octet string *mask* of length *oLen* octets; or “error”

The octet string *mask* shall be computed by the following or an equivalent sequence of steps:

1. If *zLen* exceeds the length limitation ($2^{61} - 5$ for SHA-1 or RIPEMD-160), or if $oLen > hLen \times 2^{32}$, output “error” and stop.
2. Let M be the empty string. Let $cThreshold = \lceil oLen/hLen \rceil$.
3. Let $counter = 0$
   - 3.1 Convert *counter* to an octet string C of length 4 octets using I2OSP.
   - 3.2 Compute $Hash(Z \parallel C)$ with the selected hash function to produce an octet string H of *hLen* octets.
   - 3.3 Let $M = M \parallel H$.
   - 3.4 Increment *counter* by one. If $counter < cThreshold$, go to step 3.1 above.
4. Output the leading *oLen* octets of M as the octet string *mask*.

NOTE—Since *counter* is only 32 bits, the maximum length of the output is at most $hLen \times 2^{32}$.

Annex A

(informative)

Number-theoretic background

A.1 Integer and modular arithmetic: overview

A.1.1 Modular arithmetic

Modular reduction

Modular arithmetic is based on a fixed integer $m > 1$, called the *modulus*. The fundamental operation is *reduction modulo m*. To reduce an integer a modulo m, one divides a by m and takes the remainder r. This operation is written

$$r := a \bmod m$$

The remainder must satisfy $0 \leq r < m$.

Examples:

$11 \bmod 8 = 3$
$7 \bmod 9 = 7$
$-2 \bmod 11 = 9$
$12 \bmod 12 = 0$

Congruences

Two integers a and b are said to be *congruent modulo m* if they have the same result upon reduction modulo m. This relationship is written

$$a \equiv b \pmod{m}$$

Two integers are congruent *modulo m* if, and only if, their difference is divisible by m.

Example:

$$11 \equiv 19 \pmod{8}$$

If $r = a \bmod m$, then $r \equiv a \pmod{m}$.

If $a_0 \equiv b_0 \pmod{m}$ and $a_1 \equiv b_1 \pmod{m}$, then

$$a_0 + a_1 \equiv b_0 + b_1 \pmod{m}$$
$$a_0 - a_1 \equiv b_0 - b_1 \pmod{m}$$
$$a_0 a_1 \equiv b_0 b_1 \pmod{m}$$

Integers modulo m

The *integers modulo m* are the possible results of reduction modulo m. Thus, the set of integers modulo m is

$$Z_m = \{0, 1, \ldots, m - 1\}$$

One performs addition, subtraction, and multiplication on the set Z_m by performing the corresponding integer operation and reducing the result modulo m. For example, in Z_7

$$3 = 6 + 4$$
$$5 = 1 - 3$$
$$6 = 4 \times 5$$

Modular exponentiation

If v is a positive integer and g is an integer modulo m, then *modular exponentiation* is the operation of computing $g^v \bmod m$ (also written exp $(g, v) \bmod m$). A.2.1 contains an efficient method for modular exponentiation.

GCDs and LCMs

If m and h are integers, the *greatest common divisor* (GCD) is the largest positive integer d dividing both m and h. If $d = 1$, then m and h are said to be *relatively prime* (or *coprime*). A.2.2 contains an efficient method for computing the GCD.

The *least common multiple* (LCM) is the smallest positive integer l divisible by both m and h. The GCD and LCM are related by

$$\text{GCD}\ (h, m) \times \text{LCM}\ (h, m) = hm$$

(for h and m positive), so that the LCM is easily computed if the GCD is known.

Modular division

The *multiplicative inverse* of h modulo m is the integer k modulo m, such that $hk \equiv 1 \pmod{m}$. The multiplicative inverse of h is commonly written $h^{-1} \pmod{m}$. It exists if h is relatively prime to m and not otherwise.

If g and h are integers modulo m, and h is relatively prime to m, then the *modular quotient g/h* modulo m is the integer $gh^{-1} \bmod m$. If c is the modular quotient, then c satisfies $g \equiv hc \pmod{m}$.

The process of finding the modular quotient is called *modular division.* A.2.2 contains an efficient method for modular division.

A.1.2 Prime finite fields

The field GF (p)

In the case in which m equals a prime p, the set Z_p forms a *prime finite field* and is denoted by $GF\ (p)$.

In the finite field $GF\ (p)$, modular division is possible for any denominator other than 0. The set of nonzero elements of $GF\ (p)$ is denoted by $GF\ (p)^*$.

Orders

The *order* of an element c of $GF(p)^*$ is the smallest positive integer v, such that $c^v \equiv 1 \pmod p$. The order always exists and divides $p - 1$. If k and l are integers, then $c^k \equiv c^l \pmod p$ if, and only if, $k \equiv l \pmod v$.

Generators

If v divides $p - 1$, then there exists an element of $GF(p)^*$ having order v. In particular, there always exists an element g of order $p - 1$ in $GF(p)^*$. Such an element is called a *generator* for $GF(p)^*$, because every element of $GF(p)^*$ is some power of g. In number-theoretic language, g is also called a *primitive root* for p.

Exponentiation and discrete logarithms

Suppose that the element g of $GF(p)^*$ has order v. Then an element h of $GF(p)^*$ satisfies

$$h \equiv g^l \pmod p$$

for some l if, and only if, $h^v \equiv 1 \pmod p$. The exponent l is called the *discrete logarithm* of h (with respect to the base g). The discrete logarithm is an integer modulo v.

DL-based cryptography

Suppose that g is an order-r element of $GF(p)^*$, where r is prime. Then a key pair can be defined as follows:

- The private key s is an integer modulo r.
- The corresponding public key w is an element of $GF(p)^*$ defined by

$$w := g^s \pmod p.$$

It is necessary to compute a discrete logarithm in order to derive a private key from its corresponding public key. For this reason, public-key cryptography based on key pairs of this type relies for its security on the difficulty of the discrete logarithm problem. Thus, it is an example of *DL-based cryptography.* The difficulty of the discrete logarithm problem is discussed in D.4.1.

A.1.3 Composite moduli

RSA moduli

An *RSA modulus* is the product n of two odd (distinct) primes p and q. It is assumed that p and q are large enough that factoring n is computationally infeasible (see *IF-based cryptography* later in this subclause).

A *unit* modulo n is a positive integer less than n and relatively prime to n (i.e., divisible by neither p nor q). The set of all units modulo n is denoted by U_n. The modular product and modular quotient of units are also units. If a is a unit and k an integer, then $a^k \bmod n$ is also a unit.

If a positive integer less than n is selected without knowledge of p or q, it is virtually certain to be a unit. (Indeed, generating a nonunit modulo n is as difficult as factoring n.)

Exponentiation

The *universal exponent* of an RSA modulus $n = pq$ is defined to be

$$\lambda(n) := \mathrm{LCM}(p - 1, q - 1)$$

If c and d are positive integers congruent modulo $\lambda(n)$, then

$$a^c \equiv a^d \pmod{n}$$

for every integer a. It follows that if

$$de \equiv 1 \pmod{\lambda(n)}$$

then

$$a^{de} \equiv a \pmod{n}$$

for every integer a.

IF-based cryptography

Given an RSA modulus n, a pair of integers d, e satisfying $de \equiv 1 \pmod{\lambda(n)}$ can be taken as the components of a key pair. Traditionally, e denotes the public key and d the private key. Given n and e, it is known that finding d is as difficult as factoring n. This, in turn, is believed to be necessary for computing a unit a given $a^e \bmod n$. For this reason, public-key cryptography based on key pairs of this type relies for its security on the difficulty of the integer factorization problem. Thus, it is an example of *IF-based cryptography*. The difficulty of the integer factorization problem is discussed in D.4.3.

A.1.4 Modular square roots

The Legendre symbol

If $p > 2$ is prime, and a is any integer, then the *Legendre symbol* $\left(\frac{a}{p}\right)$ is defined as follows. If p divides a, then $\left(\frac{a}{p}\right) = 0$. If p does not divide a, then $\left(\frac{a}{p}\right)$ equals 1 if a is a square modulo p and –1 otherwise. (Despite the similarity in notation, a Legendre symbol should not be confused with a rational fraction; the distinction must be made from the context.)

The Jacobi symbol

The Jacobi symbol $\left(\frac{a}{n}\right)$ is a generalization of the Legendre symbol. If $n > 1$ is odd with prime factorization

$$n = \prod_{i=1}^{t} p_i^{e_i}$$

and a is any integer, then the Jacobi symbol is defined to be

$$\left(\frac{a}{n}\right) := \prod_{i=1}^{t} \left(\frac{a}{p_i}\right)^{e_i}$$

where the symbols $\left(\frac{a}{p_i}\right)$ are Legendre symbols. (Despite the similarity in notation, a Jacobi symbol should not be confused with a rational fraction; the distinction must be made from the context.)

The values of the Jacobi symbol are ±1 if a and n are relatively prime and 0 otherwise. The values 1 and –1 are achieved equally often (unless n is a square, in which case the value –1 does not occur at all).

Algorithms for computing Legendre and Jacobi symbols are given in A.2.3.

Square roots modulo a prime

Let p be an odd prime, and let g be an integer with $0 \leq g < p$. A *square root modulo p* of g is an integer z with $0 \leq z < p$ and

$$z^2 \equiv g \pmod p$$

The number of square roots modulo p of g is $1+J$, where J is the Jacobi symbol $\left(\frac{g}{p}\right)$.

If $g = 0$, then there is one square root modulo p, namely $z = 0$. If $g \neq 0$, then g has either 0 or 2 square roots modulo p. If z is one square root, then the other is $p - z$.

A procedure for computing square roots modulo a prime is given in A.2.5.

RW moduli

If $n = pq$ is an RSA modulus with $p \equiv q \equiv 3 \pmod 4$ and $p \not\equiv q \pmod 8$, then n is called an *RW modulus*.

Denote by J_n the set of units u such that the Jacobi symbol $\left(\frac{u}{n}\right)$ equals 1. The set J_n comprises precisely half of the units. If f is a unit, then either f or $f/2$ mod n is in J_n.

Exponentiation

Let $\lambda(n)$ be the universal exponent of n (see A.1.3). If c and d are congruent modulo $\lambda(n)/2$, then

$$a^c \equiv \pm a^d \pmod n$$

for every a in J_n. It follows that if

$$de \equiv 1 \pmod{\lambda(n)/2}$$

then

$$a^{de} \equiv \pm a \pmod n$$

for every a in J_n.

IF-based cryptography

Given an RW modulus n, a pair of integers d, e satisfying $de \equiv 1 \pmod{\lambda(n)/2}$ can be taken as the components of a key pair. Traditionally, e denotes the public key and d the private key. The public key e must be even (see 8.1.3.2).

Given n and e, it is known that finding d is as difficult as factoring n. This, in turn, is necessary for computing a unit a given a^e mod n. For this reason, public-key cryptography based on key pairs of this type provides another example of IF-based cryptography. The difficulty of the integer factorization problem is discussed in D.4.3.

A.2 Integer and modular arithmetic: algorithms

A.2.1 Modular exponentiation

Modular exponentiation can be performed efficiently by the *binary method* outlined below.

Input: A positive integer v, a modulus m, and an integer g modulo m

Output: $g^v \bmod m$

```
1.  Let $v = v_r v_{r-1} ... v_1 v_0$ be the binary representation of $v$, where the most significant bit $v_r$ of $v$ is 1.
2.  Set $x \leftarrow g$ .
3.  For $i$ from $r - 1$ downto 0 do
    3.1  Set $x \leftarrow x^2 \bmod m$.
    3.2  If $v_i = 1$, then set $x \leftarrow gx \bmod m$.
4.  Output $x$.
```

There are several modifications that improve the performance of this algorithm. These methods are summarized in Gordon [B65].

A.2.2 The extended Euclidean algorithm

The following algorithm computes efficiently the GCD d of m and h. If m and h are relatively prime, the algorithm also finds the quotient g/h modulo m.

Input: An integer $m > 1$ and integers g and $h > 0$. (If only the GCD of m and h is desired, no input g is required.)

Output: The GCD d of m and h and, if $d = 1$, the integer c with $0 < c < m$ and $c \equiv g/h \pmod{m}$

```
1.  If $h = 1$, then output $d := 1$ and $c := g$ and stop.
2.  Set $r_0 \leftarrow m$ .
3.  Set $r_1 \leftarrow h \bmod m$.
4.  Set $s_0 \leftarrow 0$ .
5.  Set $s_1 \leftarrow g \bmod m$.
6.  While $r_1 > 0$
    6.1  Set $q \leftarrow \lfloor r_0 / r_1 \rfloor$ .
    6.2  Set $r_2 \leftarrow r_0 - qr_1 \bmod m$.
    6.3  Set $s_2 \leftarrow s_0 - qs_1 \bmod m$.
    6.4  Set $r_0 \leftarrow r_1$ .
         Set $r_1 \leftarrow r_2$ .
         Set $s_0 \leftarrow s_1$ .
         Set $s_1 \leftarrow s_2$ .
```

```
7. Output $d := r_0$.
8. If $r_0 = 1$, then output $c := s_0$.
```

If m is prime, the quotient exists provided that $h \not\equiv 0 \pmod{m}$, and can be found efficiently using exponentiation via

$$c := g\, h^{m-2} \bmod m.$$

A.2.3 Evaluating Jacobi symbols

The following algorithm efficiently computes the Jacobi symbol.

Input: An integer a and an odd integer $n > 1$

Output: The Jacobi symbol $\left(\frac{a}{n}\right)$

```
1. Set $x \leftarrow a, y \leftarrow n, J \leftarrow 1$.
2. While $y > 1$
   2.1 Set $x \leftarrow (x \bmod y)$.
   2.2 If $x > y/2$, then
       2.2.1 Set $x \leftarrow y - x$.
       2.2.2 If $y \equiv 3 \pmod 4$, then set $J \leftarrow -J$.
   2.3 If $x = 0$, then set $x \leftarrow 1, y \leftarrow 0, J \leftarrow 0$.
   2.4 While 4 divides $x$
       2.4.1 Set $x \leftarrow x/4$.
   2.5 If 2 divides $x$, then
       2.5.1 Set $x \leftarrow x/2$.
       2.5.2 If $y \equiv \pm 3 \pmod 8$, then set $J \leftarrow -J$.
   2.6 If $x \equiv 3 \pmod 4$ and $y \equiv 3 \pmod 4$ then set $J \leftarrow -J$.
   2.7 Switch $x$ and $y$.
3. Output $J$.
```

If n is equal to a prime p, the Jacobi symbol can also be found efficiently using exponentiation via

$$\left(\frac{a}{p}\right) := a^{(p-1)/2} \bmod p$$

A.2.4 Generating Lucas sequences

Let P and Q be nonzero integers. The *Lucas sequence* V_k for P, Q is defined by

$$V_0 = 2,\ V_1 = P, \text{ and } V_k = PV_{k-1} - QV_{k-2} \text{ for } k \geq 2$$

This recursion is adequate for computing V_k for small values of k. For large k, one can compute V_k modulo an odd integer $n > 2$ using the following algorithm (see Joye and Quisquater [B87]). The algorithm also computes the quantity $Q^{\lfloor k/2 \rfloor}$ mod n; this quantity will be useful in the application given in A.2.5.

Input: An odd integer $n > 2$, integers P and Q, and a positive integer k

Output: V_k mod n and $Q^{\lfloor k/2 \rfloor}$ mod n

```
1. Set $v_0 \leftarrow 2, v_1 \leftarrow P, q_0 \leftarrow 1, q_1 \leftarrow 1$ .
2. Let $k = k_r\ k_{r-1} ... k_1\ k_0$ be the binary representation of $k$, where the leftmost bit $k_r$ of $k$ is 1.
3. For $i$ from $r$ downto 0 do
   3.1 Set $q_0 \leftarrow q_0 q_1$ mod $n$.
   3.2 If $k_i = 1$, then set
          $q_1 \leftarrow q_0\ Q$ mod $n$.
          $v_0 \leftarrow v_0 v_1 - P\ q_0$ mod $n$.
          $v_1 \leftarrow v_1^2 - 2\ q_1$ mod $n$.
       Else set
          $q_1 \leftarrow q_0$ .
          $v_1 \leftarrow v_0 v_1 - P\ q_0$ mod $n$.
          $v_0 \leftarrow v_0^2 - 2\ q_0$ mod $n$.
4. Output $v_0$ and $q_0$.
```

A.2.5 Finding square roots modulo a prime

The following algorithm computes a square root z modulo p of $g \neq 0$.

Input: An odd prime p, and an integer g with $0 < g < p$

Output: A square root modulo p of g if one exists. In case III, the message “no square roots exist” is returned if none exists.

```
I. $p \equiv 3$ (mod 4); that is, $p = 4k + 3$ for some positive integer $k$ (see Lehmer [B100]).
   1. Compute (via A.2.1) and output $z := g^{k+1}$ mod $p$.
II. $p \equiv 5$ (mod 8); that is, $p = 8k + 5$ for some positive integer $k$ (see Atkin [B16]).
   1. Compute $\gamma := (2g)^k$ mod $p$ via A.2.1.
   2. Compute $i := 2g\gamma^2$ mod $p$.
   3. Compute and output $z := g\gamma(i-1)$ mod $p$.
III. $p \equiv 1$ (mod 8) (see Lehmer [B100]).
   1. Set $Q \leftarrow g$ .
   2. Generate a value $P$ with $0 < P < p$ not already chosen.
   3. Compute via A.2.4 the quantities
      $V := V_{(p+1)/2}$ mod $p$   and   $Q_0 := Q^{(p-1)/4}$ mod $p$.
   4. Set $z \leftarrow V/2$ mod $p$.
   5. If ($z^2$ mod $p$) = $g$, then output $z$ and stop.
   6. If $1 < Q_0 < p - 1$, then output the message “no square roots exist” and stop.
   7. Go to step 2 .
```

NOTES

1—To perform the modular division of an integer V by 2 (needed in step 4 of case III), one can simply divide by 2 the integer V or $V + p$ (whichever is even). (The integer division by 2 can be accomplished by shifting the binary expansion of the dividend by one bit.)

2—As written, the algorithm for case III works for all $p \equiv 1 \pmod 4$, although it is less efficient than the algorithm for case II, when $p \equiv 5 \pmod 8$.

3—In case III, a given choice of P will produce a solution if, and only if, $P^2 - 4Q$ is not a quadratic residue modulo p. If P is chosen at random, the probability of this is at least 1/2. Thus, only a few values of P will be required. It may, therefore, be possible to speed up the process by restricting to very small values of P and implementing the multiplications by P in A.2.4 by repeated addition.

4—In case I and case II, the algorithm produces a solution z, provided that one exists. If it is unknown whether a solution exists, then the output z should be checked by comparing $w := z^2 \bmod p$ with g. If $w = g$, then z is a solution; otherwise no solutions exist. In case III, the algorithm performs the determination of whether or not a solution exists.

A.2.6 Finding square roots modulo a power of 2

If $r > 2$ and $a < 2^r$ is a positive integer congruent to 1 modulo 8, then there is a unique positive integer b less than 2^{r-2}, such that $b^2 \equiv a \pmod{2^r}$. The number b can be computed efficiently using the following algorithm. The binary representations of the integers a, b, h are denoted as

$$a = a_{r-1}...a_1a_0$$
$$b = b_{r-1}...b_1b_0$$
$$h = h_{r-1}...h_1h_0$$

Input: An integer $r > 2$, and a positive integer $a \equiv 1 \pmod 8$ less than 2^r

Output: The positive integer b less than 2^{r-2}, such that $b^2 \equiv a \pmod{2^r}$

```
1.  Set $h \leftarrow 1$ .
2.  Set $b \leftarrow 1$ .
3.  For $j$ from 2 to $r - 2$ do
    3.1  If $h_{j+1} \neq a_{j+1}$, then
         3.1.1 Set $b_j \leftarrow 1$ .
         3.1.2 If $2j < r$, then $h \leftarrow (h + 2^{j+1}b - 2^{2j}) \bmod 2^r$.
         3.1.3 Else $h \leftarrow (h + 2^{j+1}b) \bmod 2^r$.
4.  If $b_{r-2} = 1$, then set $b \leftarrow 2^{r-1} - b$ .
5.  Output $b$.
```

A.2.7 Computing the order of a given integer modulo a prime

Let p be a prime, and let g satisfy $1 < g < p$. The following algorithm determines the order of g modulo p. The algorithm is efficient only for small p.

Input: A prime p and an integer g with $1 < g < p$

Output: The order k of g modulo p

```
1. Set $b \leftarrow g$ and $j \leftarrow 1$.
2. Set $b \leftarrow gb \bmod p$ and $j \leftarrow j + 1$.
3. If $b > 1$, then go to step 2.
4. Output $j$.
```

A.2.8 Constructing an integer of a given order modulo a prime

Let p be a prime, and let T divide $p - 1$. The following algorithm generates an element of $GF(p)$ of order T. The algorithm is efficient only for small p.

Input: A prime p and an integer T dividing $p - 1$

Output: An integer u having order T modulo p

```
1. Generate a random integer $g$ between 1 and $p$.
2. Compute via A.2.7 the order $k$ of $g$ modulo $p$.
3. If $T$ does not divide $k$, then go to step 1.
4. Output $u := g^{k/\mathrm{T}} \bmod p$.
```

A.2.9 An implementation of IF signature primitives

The following algorithm is an implementation of the IFSP-RSA1, IFSP-RSA2, and IFSP-RW signature primitives. Assuming e is small, it is more efficient for RW signatures than the primitive given in 8.2.8, because it combines the modular exponentiation and Jacobi symbol calculations. The algorithm requires that the primes p and q be known to the user; thus, the private key can have either the second or third form specified in 8.1.3. (Note that this algorithm cannot be applied to IF signature verification, since it requires knowledge of p and q, which is equivalent to knowledge of the private key.)

One-time computation: If the private key is the triple (p, q, d) (see 8.1.3), then the remaining components, d_1, d_2, c, can be computed via

$c := q^{-1} \bmod p$ via A.2.2
$d_1 := d \bmod (p - 1)$
$d_2 := d \bmod (q - 1)$

NOTE—For the smallest values of e, the values of d_1 and d_2 can be written down explicitly as follows:

RSA: If $e = 3$, then

$d_1 = (2p - 1)/3$ and $d_2 = (2q - 1)/3$

RW: If $e = 2$, then

if $p \equiv 3 \pmod 8$, $q \equiv 7 \pmod 8$ and d odd, or

if $p \equiv 7 \pmod 8$, $q \equiv 3 \pmod 8$ and d even

$d_1 = (p + 1)/4$ and $d_2 = (3q - 1)/4$

if $p \equiv 3 \pmod 8$, $q \equiv 7 \pmod 8$ and d even, or

if $p \equiv 7 \pmod 8$, $q \equiv 3 \pmod 8$ and d odd

$d_1 = (3p - 1)/4$ and $d_2 = (q + 1)/4$

In the RW case, one also computes via A.2.1

$w_1 := \exp(2, p - 1 - d_1) \bmod p$

$w_2 := \exp(2, q - 1 - d_2) \bmod q$

Input: The (fixed) integers p, q, e, c, d_1, d_2, and (for RW) w_1 and w_2; the (per-message) integer f modulo pq. (It is unnecessary to store both p and d_1, if one can be conveniently computed from the other, as in the cases $e = 2$ or $e = 3$. The same remark holds for q and d_2.)

Output: The signature s of f

The steps marked with an asterisk (*) below are to be implemented only for RW and not for RSA.

```
1.   Set $f_1 \leftarrow f \bmod p$.
2.   Compute $j_1 := \exp(f_1, d_1) \bmod p$ via A.2.1.
3.*  Compute $i_1 := \exp(j_1, e) \bmod p$.
4.*  If $i_1 = f_1$, then set $u_1 \leftarrow 0$; else set $u_1 \leftarrow 1$.
5.   Set $f_2 \leftarrow f \bmod q$.
6.   Compute $j_2 := \exp(f_2, d_2) \bmod q$ via A.2.1.
7.*  Compute $i_2 := \exp(j_2, e) \bmod q$.
8.*  If $i_2 = f_2$, then set $u_2 \leftarrow 0$; else set $u_2 \leftarrow 1$.
9.*  If $u_1 \neq u_2$, then compute
       $t_1 := j_1 w_1 \bmod p$
       $t_2 := j_2 w_2 \bmod q$
     and go to step 11.
10.  Set
       $t_1 \leftarrow j_1, t_2 \leftarrow j_2$.
11.  Compute
       $h := (t_1 - t_2)\, c \bmod p$
       $t := t_2 + hq$.
12.  If IFSP-RSA1, output $s := t$. Else (i.e., if IFSP-RSA2 or IFSP-RW), output $s := \min(t, pq - t)$.
```

NOTE—Since the exponents are fixed in steps 2 and 6, the exponentiations can be made more efficient using *addition chains* rather than the binary method in A.2.1 (see Gordon [B65]).

A.3 Binary finite fields: overview

A.3.1 Finite fields

A *finite field* (or *Galois field*) is a set with finitely many elements in which the usual algebraic operations (addition, subtraction, multiplication, division by nonzero elements) are possible, and in which the usual algebraic laws (commutative, associative, distributive) hold. The *order* of a finite field is the number of elements it contains. If $q > 1$ is an integer, then a finite field of order q exists if q is a prime power, and not otherwise.

The finite field of a given order is unique, in the sense that any two fields of order q display identical algebraic structure. Nevertheless, there are often many ways to represent a field. It is traditional to denote the finite field of order q by F_q or $GF\ (q)$; this standard uses the latter notation for typographical reasons. It should be borne in mind that the expressions "the field $GF\ (q)$" and "the field of order q" usually imply a choice of field representation.

Although finite fields exist of every prime-power order, there are two kinds that are commonly used in cryptography.

— When q is a prime p, the field $GF\ (p)$ is called a *prime finite field*. The field $GF\ (p)$ is typically represented as the set of integers modulo p. See A.1.2 for a discussion of these fields.

— When $q = 2^m$ for some m, the field $GF\ (2^m)$ is called a *binary finite field*. Unlike the prime field case, there are many common representations for binary finite fields. These fields are discussed below (A.3.3 through A.3.9).

These are the two kinds of finite fields considered in this standard.

A.3.2 Polynomials over finite fields

A *polynomial over GF (q)* is a polynomial with coefficients in $GF\ (q)$. Addition and multiplication of polynomials over $GF\ (q)$ are defined as usual in polynomial arithmetic, except that the operations on the coefficients are performed in $GF\ (q)$.

A polynomial over the prime field $GF\ (p)$ is commonly called a *polynomial modulo p*. Addition and multiplication are the same as for polynomials with integer coefficients, except that the coefficients of the results are reduced modulo p.

Example: Over the prime field $GF\ (7)$

$$(t^2 + 4t + 5) + (t^3 + t + 3) = t^3 + t^2 + 5t + 1$$

$$(t^2 + 3t + 4)\ (t + 4) = t^3 + 2t + 2$$

A binary polynomial is a polynomial modulo 2.

Example: Over the field $GF\ (2)$

$$(t^3 + 1) + (t^3 + t) = t + 1$$

$$(t^2 + t + 1)\ (t +1) = t^3 + 1$$

A polynomial over $GF\ (q)$ is *reducible* if it is the product of two smaller degree polynomials over $GF\ (q)$; otherwise, it is *irreducible*. For instance, the above examples show that $t^3 + 2t + 2$ is reducible over $GF\ (7)$ and that the binary polynomial $t^3 + 1$ is reducible.

Every nonzero polynomial over $GF\ (q)$ has a unique representation as the product of powers of irreducible polynomials. (This result is analogous to the fact that every positive integer has a unique representation as the product of powers of prime numbers.) The degree-1 factors correspond to the roots of the polynomial.

Polynomial congruences

Modular reduction and congruences can be defined among polynomials over $GF\ (q)$, in analogy to the definitions for integers given in A.1.1. To reduce a polynomial $a\ (t)$ modulo a nonconstant polynomial $m\ (t)$, divide $a\ (t)$ by $m\ (t)$ by long division of polynomials and take the remainder $r\ (t)$. This operation is written

$$r\ (t) := a\ (t) \bmod m\ (t)$$

The remainder $r\ (t)$ must either equal zero or have degree smaller than that of $m\ (t)$.

If $m\ (t) = t - c$ for some element c of $GF\ (q)$, then $a\ (t) \bmod m\ (t)$ is just the constant $a\ (c)$.

Two polynomials, $a\ (t)$ and $b\ (t)$, are said to be *congruent modulo* $m\ (t)$ if they have the same result upon reduction modulo $m\ (t)$. This relationship is written

$$a\ (t) \equiv b\ (t)\ (\text{mod } m\ (t))$$

One can define addition, multiplication, and exponentiation of polynomials (to integral powers) modulo $m\ (t)$ analogously to how they are defined for integer congruences in A.1.1. In the case of a prime field $GF\ (p)$, each of these operations involves both reduction of the polynomials modulo $m\ (t)$ and reduction of the coefficients modulo p.

A.3.3 Binary finite fields

If m is a positive integer, the *binary finite field* $GF\ (2^m)$ consists of the 2^m possible bit strings of length m. Thus, for example

$$GF\ (2^3) = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

The integer m is called the *degree* of the field.

For $m = 1$, the field $GF\ (2)$ is just the set {0, 1} of integers modulo 2. The addition and multiplication operations are given by

+	0	1
0	0	1
1	1	0

×	0	1
0	0	0
1	0	1

Addition

For $m > 1$, addition of two elements is implemented by bitwise addition modulo 2. Thus, for example

$$(11001) + (10100) = (01101)$$

Multiplication

There is more than one way to implement multiplication in $GF\ (2^m)$. To specify a multiplication rule, one chooses a *basis representation* for the field. The basis representation is a rule for interpreting each bit string; the multiplication rule follows from this interpretation.

There are two common families of basis representations: *polynomial basis* representations and *normal basis* representations.

A.3.4 Polynomial basis representations

In a *polynomial basis representation*, each element of GF (2^m) is represented by a different binary polynomial of degree less than m. More explicitly, the bit string ($a_{m\text{-}1} \ldots a_2\, a_1\, a_0$) is taken to represent the binary polynomial

$$a_{m-1}\ t_{m-1} + \ldots + a_2 t\ _2 + a_1 t + a_0$$

The *polynomial basis* is the set

$$B = \{t^{m-1}, \ldots, t^2, t, 1\}$$

The addition of bit strings, as defined in A.3.3, corresponds to addition of binary polynomials.

Multiplication is defined in terms of an irreducible binary polynomial $p(t)$ of degree m, called the *field polynomial* for the representation. The product of two elements is simply the product of the corresponding polynomials, reduced modulo $p(t)$.

There is a polynomial basis representation for GF (2^m) corresponding to each irreducible binary polynomial $p(t)$ of degree m. Irreducible binary polynomials exist of every degree. Roughly speaking, every one out of m binary polynomials of degree m is irreducible.

Trinomials and pentanomials

The reduction of polynomials modulo $p(t)$ is particularly efficient if $p(t)$ has a small number of terms. The irreducibles with the least number of terms are the *trinomials* $t^m + t^k + 1$. Thus, it is a common practice to choose a trinomial for the field polynomial, provided that one exists.

If an irreducible trinomial of degree m does not exist, then the next best polynomials are the *pentanomials* $t^m + t^a + t^b + t^c + 1$. For every m up to 1000, there exists either an irreducible trinomial or pentanomial of degree m. Table A.2 in A.8 provides an example for each m up to 1000. Subclause A.8 also provides algorithms for constructing other irreducibles.

A.3.5 Normal basis representations

Normal bases

A *normal basis* for GF (2^m) is a set of the form

$$B = \{\theta, \theta^2, \theta^{2^2}, \ldots, \theta^{2^{m-1}}\}$$

with the property that no subset of B adds to 0. (In the language of linear algebra, the elements of B are said to be *linearly independent*.) There exist normal bases for GF (2^m) for every positive integer m.

The representation of GF (2^m) via the normal basis B is carried out by interpreting the bit string ($a_0\, a_1\, a_2 \ldots a_{m-1}$) as the element

$$a_0\theta + a_1\theta^2 + a_2\theta^{2^2} + \ldots + a_{m-1}\theta^{2^{m-1}}$$

All of the elements of a normal basis B satisfy the same irreducible binary polynomial $p(t)$. This polynomial is called the *field polynomial* for the basis. An irreducible binary polynomial is called a *normal polynomial* if it is the field polynomial for a normal basis.

Gaussian normal bases

Normal basis representations have the computational advantage that squaring an element can be done very efficiently (see A.4.1). Multiplying distinct elements, on the other hand, can be cumbersome in general. For this reason, it is common to specialize to a class of normal bases, called *Gaussian normal bases*, for which multiplication is both simpler and more efficient.

Gaussian normal bases for $GF(2^m)$ exist whenever m is not divisible by eight. They include the *optimal normal bases*, which have the most efficient multiplication possible in a normal basis.

The *type* of a Gaussian normal basis is a positive integer measuring the complexity of the multiplication operation with respect to that basis. The smaller the type, the more efficient the multiplication. For a given m and T, the field $GF(2^m)$ can have, at most, one Gaussian normal basis of type T. Thus, it is proper to speak of *the* type T Gaussian normal basis over $GF(2^m)$. See Ash, Blake, and Vanstone [B15] for more information on Gaussian normal bases. A table of Gaussian normal bases is given in A.8.1.

The Gaussian normal bases of Types 1 and 2 have the most efficient multiplication rules of all normal bases. For this reason, they are called *optimal* normal bases. The Type 1 Gaussian normal bases are called Type I optimal normal bases, and the Type 2 Gaussian normal bases are called Type II optimal normal bases. See Menezes [B108] for more information on optimal normal bases.

A.3.6 Checking for a Gaussian normal basis

If $m > 1$ is not divisible by eight, the following algorithm (Ash, Blake, and Vanstone [B15]) tests for the existence of a Gaussian normal basis for $GF(2^m)$ of given type. (See also Table A.2 in A.8.1.)

Input: An integer $m > 1$ not divisible by eight; a positive integer T

Output: If a type T Gaussian normal basis for $GF(2^m)$ exists, the message "True"; otherwise "False"

```
1. Set $p \leftarrow \mathrm{T}m + 1$.
2. If $p$ is not prime, then output "False" and stop.
3. Compute via A.2.7 the order $k$ of 2 modulo $p$.
4. Set $h \leftarrow \mathrm{T}m/k$.
5. Compute $d := \mathrm{GCD}(h, m)$ via A.2.2.
6. If $d = 1$, then output "True"; else output "False."
```

Example: Let $m = 4$ and $T = 3$. Then $p = 13$, and 2 has order $k = 12$ modulo 13. Since $h = 1$ is relatively prime to $m = 4$, there does exist a Gaussian normal basis of Type 3 for $GF(2^4)$.

A.3.7 The multiplication rule for a Gaussian normal basis

The following procedure produces the rule for multiplication with respect to a given Gaussian normal basis. (See A.6 for a more general method applicable to all normal bases.)

Input: Integers $m > 1$ and T for which there exists a type T Gaussian normal basis B for $GF(2^m)$

Output: An explicit formula for the first coordinate of the product of two elements with respect to B

```
1. Set $p \leftarrow \mathrm{T}m + 1$.
2. Generate via A.2.8 an integer $u$ having order $T$ modulo $p$.
3. Compute the sequence $F(1), F(2), \ldots, F(p-1)$ as follows:
   3.1 Set $w \leftarrow 1$.
   3.2 For $j$ from 0 to $T - 1$ do
         Set $n \leftarrow w$.
         For $i$ from 0 to $m - 1$ do
               Set $F(n) \leftarrow i$.
               Set $n \leftarrow 2n \bmod p$.
         Set $w \leftarrow uw \bmod p$.
4. If $T$ is even, then set $J \leftarrow 0$; else set
      $J := \sum_{k=1}^{m/2} (a_{k-1} b_{m/2+k-1} + a_{m/2+k-1} b_{k-1})$
5. Output the formula
      $c_0 = J + \sum_{k=1}^{p-2} a_{F(k+1)} b_{F(p-k)}$
```

Example: For the Type 3 normal basis for $GF(2^4)$, the values of F are given by

$F(1) = 0$	$F(5) = 1$	$F(9) = 0$
$F(2) = 1$	$F(6) = 1$	$F(10) = 2$
$F(3) = 0$	$F(7) = 3$	$F(11) = 3$
$F(4) = 2$	$F(8) = 3$	$F(12) = 2$

Therefore, after simplifying one obtains

$$c_0 = a_0 (b_1 + b_2 + b_3) + a_1 (b_0 + b_2) + a_2 (b_0 + b_1) + a_3 (b_0 + b_3)$$

Here c_0 is the first coordinate of the product

$$(*) \qquad (\mathrm{c}_0 c_1 \ldots c_{m-1}) = (a_0\, a_1 \ldots a_{m-1}) \times (b_0\, b_1 \ldots b_{m-1})$$

The other coordinates of the product are obtained from the formula for c_0 by cycling the subscripts modulo m. Thus

$$c_1 = a_1 (b_2 + b_3 + b_0) + a_2 (b_1 + b_3) + a_3 (b_1 + b_2) + a_0 (b_1 + b_0)$$

$$c_2 = a_2 (b_3 + b_0 + b_1) + a_3 (b_2 + b_0) + a_0 (b_2 + b_3) + a_1 (b_2 + b_1)$$

$$c_3 = a_3 (b_0 + b_1 + b_2) + a_0 (b_3 + b_1) + a_1 (b_3 + b_0) + a_2 (b_3 + b_2)$$

A.3.8 A Multiplication algorithm for a Gaussian normal basis

The formulas given in A.3.7 for $c_0, \ldots, c_{m-1}$ can be implemented in terms of a single expression. For

$\underline{u} = (u_0\ u_1\ \ldots\ u_{m-1})$, $\underline{v} = (v_0\ v_1 \ldots v_{m\text{-}1})$

let

$F(\underline{u}, \underline{v})$

be the expression with

$c_0 = F\ (\underline{a}, \underline{b})$

Then, the product (*) can be computed as follows:

```
1.  Set $(u_0\ u_1\ \ldots\ u_{m-1}) \leftarrow (a_0\ a_1\ \ldots\ a_{m-1})$.
2.  Set $(v_0\ v_1\ \ldots\ v_{m-1}) \leftarrow (b_0\ b_1\ \ldots\ b_{m-1})$.
3.  For $k$ from 0 to $m - 1$ do
    3.1 Compute
          $c_k := F\ (\underline{u}, \underline{v})$.
    3.2 Set $u \leftarrow$ LeftShift($u$) and $v \leftarrow$ LeftShift($v$), where LeftShift denotes the
        circular left shift operation.
4.  Output $c := (c_0\ c_1\ \ldots\ c_{m-1})$.
```

In the above example

$F\ (\underline{u}, \underline{v}) := u_0\ (v_1 + v_2 + v_3) + u_1\ (v_0 + v_2) + u_2\ (v_0 + v_1) + u_3\ (v_0 + v_3)$

If

$a := (1000) = \theta$ and $b := (1101) = \theta + \theta^2 + \theta^8$

then

$c_0 = F\ ((1000), (1101)) = 0$
$c_1 = F\ ((0001), (1011)) = 0$
$c_2 = F\ ((0010), (0111)) = 1$
$c_3 = F\ ((0100), (1110)) = 0$

so that

$c = ab = (0010)$

A.3.9 Binary finite fields (continued from A.3.3)

Exponentiation

If k is a positive integer and α is an element of $GF\ (2^m)$, then *exponentiation* is the operation of computing α^k. Subclause A.4.3 contains an efficient method for exponentiation.

Division

If α and $\beta \neq 0$ are elements of the field *GF* (2^m), then the *quotient* α/β is the element γ such that $\alpha = \beta\gamma$.

In the finite field *GF* (2^m), division is possible for any denominator other than zero. The set of nonzero elements of *GF* (2^m) is denoted by *GF* $(2^m)^*$.

Subclause A.4.4 contains an efficient method for division.

Orders

The *order* of an element γ of *GF* $(2^m)^*$ is the smallest positive integer v, such that $\gamma^v = 1$. The order always exists and divides $2^m - 1$. If k and l are integers, then $\gamma^k = \gamma^l$ in *GF* (2^m) if, and only if, $k \equiv l$ (mod v).

Generators

If v divides $2^m - 1$, then there exists an element of *GF* $(2^m)^*$ having order v. In particular, there always exists an element γ of order $2^m - 1$ in *GF* $(2^m)^*$. Such an element is called a *generator* for *GF* $(2^m)^*$ because every element of *GF* $(2^m)^*$ is some power of γ.

Exponentiation and discrete logarithms

Suppose that the element γ of *GF* $(2^m)^*$ has order v. Then an element η of *GF* $(2^m)^*$ satisfies $\eta = \gamma^l$ for some l if, and only if, $\eta^v = 1$. The exponent l is called the *discrete logarithm* of η (with respect to the base γ). The discrete logarithm is an integer modulo v.

DL-based cryptography

Suppose that γ is an order-r element of *GF* $(2^m)^*$, where r is prime. Then a key pair can be defined as follows:

- The private key s is an integer modulo r.
- The corresponding public key w is an element of *GF* $(2^m)^*$ defined by $w := \gamma^s$.

It is necessary to compute a discrete logarithm in order to derive a private key from its corresponding public key. For this reason, public-key cryptography based on key pairs of this type relies for its security on the difficulty of the discrete logarithm problem. Thus, it is an example of *DL-based cryptography.* The difficulty of the discrete logarithm problem is discussed in D.4.1.

Field extensions

Suppose that d and m are positive integers with d dividing m, and let **K** be a field *GF* (2^m). Then there are precisely 2^d elements α of **K** satisfying

$$\alpha^{2^d} = \alpha$$

The set **F** of all such elements forms a field *GF* (2^d). The field **F** is called a *subfield* of **K**, and **K** is called an *extension field* of **F**.

Suppose that γ is an element of *GF* (2^m) of prime order r. The prime r is said to be a *primitive* factor of $2^m - 1$ if r does not divide $2^d - 1$ for any $d < m$ dividing m. It follows from the previous paragraph that r is primitive if, and only if, γ is not an element of any proper subfield of $GF\,(2^m)$. (A *proper* subfield of a field **F** is any subfield except **F** itself.)

Example

Let **K** be the field $GF\,(2^4)$ given by the polynomial basis with field polynomial $p\,(t) := t^4 + t + 1$. Since

$$(t^2 + t)^3 = (t^2 + t + 1)^3 = 1$$

then $\mathbf{F} = \{0, 1, t^2 + t, t^2 + t + 1\}$ forms the field $GF\,(2^2)$.

A.3.10 Parameters for common key sizes

When selecting domain parameters for DL-based cryptography over binary fields, it is necessary to begin by choosing the following:

— The degree m of the field (so that the field has 2^m elements)

— The prime number r, which is to serve as the order of the base point

These two numbers are related by the condition that r must be a primitive divisor of $2^m - 1$ (see A.3.9). Moreover, it is a common practice to choose the two numbers to provide comparable levels of security (see D.4.1.4, Note 1). Each parameter depends on the size of the symmetric keys which the DL system is being used to protect.

Table A.1 below lists several common symmetric key lengths. For each length is given a set of parameters of m and r, which can be used in conjunction with symmetric keys of that length.

Table A.1—Example parameters for DL-based cryptography over binary fields

Key size	*m*	*r*
40	189	207617485544258392970753527
56	384	442499826945303593556473164314770689
64	506	2822551529460330847604262086149015242689
80	1024	7455602825647884208337395736200454918783366342657
112	2068	1628456355410411547509613374150158051231656743052344513161858038422883778013
128	2880	19194878188585855612908061936944281464039294965346491767953330250249208842371201

A.4 Binary finite fields: algorithms

The algorithms in this subclause perform operations in a finite field $GF\,(2^m)$ having 2^m elements. The elements of $GF\,(2^m)$ can be represented via either a polynomial basis modulo the irreducible polynomial $p\,(t)$, or a normal basis $\{\theta, \theta^2, \theta^{2^2}, \ldots, \theta^{2^{m-1}}\}$.

A.4.1 Squaring and square roots

Polynomial Basis

If

$$\alpha = \alpha_{m-1}t^{m-1} + \ldots + \alpha_2 t^2 + \alpha_1 t + \alpha_0$$

then

$$\alpha^2 = \alpha_{m-1}t^{2m-2} + \ldots + \alpha_2 t^4 + \alpha_1 t^2 + \alpha_0 \bmod p(t)$$

To compute $\sqrt{\alpha}$, take α and square $m - 1$ times.

Normal Basis

If α has representation

$$\alpha = (\alpha_0\ \alpha_1\ .\ .\ .\ \alpha_{m-1})$$

then

$$\alpha_2 = (\alpha_{m-1}\ \alpha_0\ \alpha_1\ .\ .\ .\ \alpha_{m-2})$$

and

$$\sqrt{\alpha} = (\alpha_1\ .\ .\ .\ \alpha_{m-2}\ \alpha_{m-1}\ \alpha_0)$$

A.4.2 The squaring matrix

If it is necessary to perform frequent squarings in a fixed polynomial basis, it can be more efficient to use a *squaring matrix*. This is an m-by-m matrix with coefficients in GF (2).

Suppose that the field polynomial is

$$p(t) := t^m + c_{m-1}\ t^{m-1} + \ldots + c_1\ t + c_0$$

Let $d_{0,j} \leftarrow c_j$ for $0 \le j < m$, and compute

$$t^m = d_{0,m-1}\ t^{m-1} + \ldots + d_{0,1}\ t + d_{0,0}$$

$$t^{m+1} = d_{1,m-1}\ t^{m-1} + \ldots + d_{1,1}\ t + d_{1,0}$$

$$\ldots$$

$$t^{2m-2} = d_{m-2,m-1}\ t^{m-1} + \ldots + d_{m-2,1}\ t + d_{m-2,0}$$

by repeated multiplication by t. Define the matrix

$$S := \begin{bmatrix} s_{1,\,1} & \cdots & s_{1,\,m} \\ \vdots & \ddots & \vdots \\ s_{m,\,1} & \cdots & s_{m,\,m} \end{bmatrix}$$

where

$$s_{i,\,j} := \begin{cases} d_{m-2i,m-j} & \text{if } i \le \lfloor m/2 \rfloor \\ 1 & \text{if } i > \lfloor m/2 \rfloor \text{ and } 2i = j + m \\ 0 & \text{otherwise} \end{cases}$$

If $(a_0\ a_1\ \ldots\ a_{m-1})$ represents the field element α, then the representation for α^2 is

$$(b_0\ b_1\ \ldots\ b_{m-1}) = (a_0\ a_1\ \ldots\ a_{m-1})\ S$$

A.4.3 Exponentiation

Exponentiation can be performed efficiently by the *binary method* outlined below.

Input: A positive integer k, a field $GF\ (2^m)$, and a field element α

Output: α^k

```
1. Let $k = k_r\ k_{r-1}\ ...\ k_1\ k_0$ be the binary representation of $k$, where the most significant bit $k_r$ of $k$ is 1.
2. Set $x \leftarrow \alpha$.
3. For $i$ from $r - 1$ downto 0 do
   3.1 Set $x \leftarrow x^2$.
   3.2 If $k_i = 1$, then set $x \leftarrow \alpha\ x$.
4. Output $x$.
```

There are several modifications that improve the performance of this algorithm. These methods are summarized in Gordon [B65].

A.4.4 Division

The quotient α /β can be computed directly (in one step by an algorithm with inputs α and β), or indirectly (by computing the multiplicative inverse β^{-1} and then multiplying it by α). There are two common methods for performing division in a finite field $GF\ (2^m)$, one direct and one indirect.

Method I: the extended Euclidean algorithm

This algorithm produces the quotient directly. (It also can be used for multiplicative inversion of β, and so for indirect division, by using as input 1 in place of α). By $\lfloor r_0(t)/r_1(t) \rfloor$ is meant the quotient upon polynomial division, dropping any remainder.

Input: A field $GF\ (2^m)$, and field elements α and $\beta \neq 0$

Output: $\gamma := \alpha\ /\beta$

```
1. Set $r_0(t) \leftarrow p(t)$.
2. Set $r_1(t) \leftarrow \beta$.
```

```
3.  Set $s_0(t) \leftarrow 0$.
4.  Set $s_1(t) \leftarrow \alpha$.
5.  While $r_1(t) \neq 0$
    5.1  Set $q(t) \leftarrow \lfloor r_0(t)/r_1(t) \rfloor$.
    5.2  Set $r_2(t) \leftarrow r_0(t) + q(t)r_1(t)$.
    5.3  Set $s_2(\mathrm{t}) \leftarrow s_0(\mathrm{t}) + q(t)s_1(t)$.
    5.4  Set $r_0(t) \leftarrow r_1(t)$; set $r_1(t) \leftarrow r_2(t)$.
    5.5  Set $s_0(t) \leftarrow s_1(t)$; set $s_1(t) \leftarrow s_2(t)$.
6.  Output $\gamma := s_0(t)$.
```

NOTES

1—An efficient hardware implementation of this procedure is described in Berlekamp [B20].

2—The extended Euclidean algorithm uses a polynomial basis representation for $GF\ (2^m)$. If a normal basis representation is being used, then one can divide using this algorithm only by converting the inputs α and β to a polynomial basis representation, performing the division, and converting the output γ back to normal basis form. (See A.7.1 for methods of converting between different basis representations.)

Method II: exponentiation

The multiplicative inverse of β can be found efficiently in either basis representation via

$$\beta^{-1} = \beta^k$$

where k is any positive integer satisfying

$$k \equiv -1 \pmod{r}$$

where r is the order of β. In particular, it is always the case that

$$\beta^{-1} = \beta^{2^m - 2}$$

If a general-purpose exponentiation algorithm such as in A.4.3 is used, then the best choice is $k := r - 1$. However, there is also a specialized algorithm of Itoh, Teechai, and Tsujii [B81] for exponentiating to the power $k = 2^m - 2$, which is more efficient than the generic method. The efficiency improvements are especially significant when squaring can be done quickly (e.g., in a normal basis representation). The procedure is given below.

Input: A field $GF\ (2^m)$ and a nonzero field element β

Output: The reciprocal β^{-1}

```
1.  Let $m - 1 = b_r\, b_{r-1} \ldots b_1\, b_0$ be the binary representation of $m - 1$, where the most significant bit
    $b_r$ of $m - 1$ is 1.
2.  Set $\eta \leftarrow \beta$ and $k \leftarrow 1$.
3.  For $i$ from $r - 1$ downto 0 do
    3.1  Set $\mu \leftarrow \eta$.
```

```
        3.2  For $j = 1$ to $k$ do
             3.2.1   Set $\mu \leftarrow \mu^2$.
        3.3  Set $\eta \leftarrow \mu\eta$ and $k \leftarrow 2k$.
        3.4  If $b_i = 1$, then set $\eta \leftarrow \eta^2\beta$ and $k \leftarrow k + 1$.
    4.  Output $\eta^2$.
```

A.4.5 Trace

If α is an element of $GF(2^m)$, the *trace* of α is

$$\mathrm{Tr}(\alpha) = \alpha + \alpha^2 + \alpha^{2^2} + ... + \alpha^{2^{m-1}}$$

The value of Tr (α) is zero for half the elements of $GF(2^m)$, and one for the other half.

The trace can be computed efficiently as follows:

Normal Basis

If α has representation ($\alpha_0\ \alpha_1 ... \alpha_{m-1}$), then

$$\mathrm{Tr}(\alpha) = \alpha_0 \oplus \alpha_1 \oplus ... \oplus \alpha_{m-1}$$

Polynomial Basis

The basic algorithm inputs $\alpha \in GF(2^m)$ and outputs $T = \mathrm{Tr}(\alpha)$.

```
    1.  Set $T \leftarrow \alpha$.
    2.  For $i$ from 1 to $m - 1$ do
        2.1  $T \leftarrow T^2 + \alpha$.
    3.  Output $T$.
```

If many traces are to be computed with respect to a fixed polynomial basis

$$\{t^{m-1}, \ldots, t, 1\}$$

then it is more efficient to compute and store the element

$$\tau = (\tau_{m-1} \ldots \tau_1 \tau_0)$$

where each coordinate

$$\tau_j = \mathrm{Tr}(t^j)$$

is computed via the basic algorithm. Subsequent traces can be computed via

$$\mathrm{Tr}(\alpha) = \alpha \cdot \tau$$

where the "dot product" of the bit strings is given by bitwise AND (or bitwise multiplication).

A.4.6 Half-trace

If m is odd, the *half-trace* of $\alpha \in GF(2^m)$ is

$$\mathrm{HfTr}(\alpha) = \alpha + \alpha^{2^2} + \alpha^{2^4} + ... + \alpha^{2^{m-1}}$$

The following algorithm inputs $\alpha \in GF(2^m)$ and outputs $H = \mathrm{HfTr}(\alpha)$:

```
1.  Set $H \leftarrow \alpha$.
2.  For $i$ from 1 to $(m-1)/2$ do
    2.1  $H \leftarrow H^2$.
    2.2  $H \leftarrow H^2 + \alpha$.
3.  Output $H$.
```

A.4.7 Solving quadratic equations over GF (2^m)

If β is an element of $GF(2^m)$, then the equation

$$z^2 + z = \beta$$

has $2 - 2T$ solutions over $GF(2^m)$, where $T = \mathrm{Tr}(\beta)$. Thus, there are either zero or two solutions. If z is one solution, then the other solution is $z + 1$. In the case $\beta = 0$, the solutions are zero and one.

The following algorithms compute a solution if one exists.

Input: A field $GF(2^m)$ along with a polynomial or normal basis for representing its elements; an element $\beta \neq 0$

Output: An element z for which $z^2 + z = \beta$, if such an element exists

Normal basis

```
1.  Let $(\beta_0\ \beta_1 ... \beta_{m-1})$ be the representation of $\beta$.
2.  Set $z_0 \leftarrow 0$.
3.  For $i = 1$ to $m - 1$ do
    3.1  Set $z_i \leftarrow z_{i-1} \oplus \beta_{i.}$
4.  Output $z \leftarrow (z_0\ z_1 ... z_{m-1})$.
```

Polynomial basis

If m is odd, then compute z := half-trace of β via A.4.6. If m is even, proceed as follows:

```
1.  Choose random $\rho \in GF(2^m)$.
2.  Set $z \leftarrow 0$ and $w \leftarrow \rho$.
3.  For $i$ from 1 to $m - 1$ do
    3.1  Set $z \leftarrow z^2 + w^2\beta$.
    3.2  Set $w \leftarrow w^2 + \rho$.
```

```
4.  If $w = 0$, then go to step 1.
5.  Output $z$.
```

If the latter algorithm is to be used repeatedly for the same field, and memory is available, then it is more efficient to precompute and store ρ and the values of *w*. Any element of trace 1 will serve as ρ, and the values of *w* depend only on ρ and not on β.

Both of the above algorithms produce a solution *z*, provided that one exists. If it is unknown whether a solution exists, then the output *z* should be checked by comparing $\gamma := z^2 + z$ with β. If $\gamma = \beta$, then *z* is a solution; otherwise no solutions exist.

A.5 Polynomials over a finite field

The computations below can take place either over a prime field (having a prime number *p* of elements), or over a binary field (having 2^m elements).

A.5.1 Exponentiation modulo a polynomial

If *k* is a positive integer and $f(t)$ and $m(t)$ are polynomials with coefficients in the field *GF* (*q*), then $f(t)^k$ mod $m(t)$ can be computed efficiently by the *binary method* outlined below.

Input: A positive integer *k*, a field *GF* (*q*), and polynomials $f(t)$ and $m(t)$ with coefficients in *GF* (*q*)

Output: The polynomial $f(t)^k$ mod $m(t)$

```
1.  Let $k = k_r\, k_{r-1} \ldots k_1\, k_0$ be the binary representation of $k$, where the most significant bit $k_r$ of $k$ is 1.
2.  Set $u(t) \leftarrow f(t) \bmod m(t)$.
3.  For $i$ from $r - 1$ downto 0 do
    3.1  Set $u(t) \leftarrow u(t)^2 \bmod m(t)$.
    3.2  If $k_i = 1$, then set $u(t) \leftarrow u(t)\, f(t) \bmod m(t)$.
4.  Output $u(t)$.
```

There are several modifications that improve the performance of this algorithm. These methods are summarized in Gordon [B65].

A.5.2 GCDs over a finite field

If $f(t)$ and $g(t) \neq 0$ are two polynomials with coefficients in the field *GF* (*q*), then there is a unique monic polynomial $d(t)$ of largest degree that divides both $f(t)$ and $g(t)$. The polynomial $d(t)$ is called the *greatest common divisor*, or GCD, of $f(t)$ and $g(t)$. The following algorithm computes the GCD of two polynomials.

Input: A finite field *GF* (*q*) and two polynomials $f(t)$, $g(t) \neq 0$ over *GF* (*q*)

Output: $d(t) = \text{GCD}\,(f(t), g(t))$

```
1.  Set $a(t) \leftarrow f(t)$, $b(t) \leftarrow g(t)$.
2.  While $b(t) \neq 0$
```

```
        2.1  Set $c(t) \leftarrow$ the remainder when $a(t)$ is divided by $b(t)$.
        2.2  Set $a(t) \leftarrow b(t)$.
        2.3  Set $b(t) \leftarrow c(t)$.
    3.  Set $\alpha \leftarrow$ the leading coefficient of $a(t)$.
    4.  Set $d(t) \leftarrow \alpha^{-1}\, a(t)$.
    5.  Output $d(t)$.
```

A.5.3 Factoring polynomials over GF (p) (special case)

Let $f(t)$ be a polynomial with coefficients in the field $GF\ (p)$, and suppose that $f(t)$ factors into distinct irreducible polynomials of degree d. (This is the special case needed in A.14.) The following algorithm finds a random degree-d factor of $f(t)$ efficiently.

Input: A prime $p > 2$, a positive integer d, and a polynomial $f(t)$, which factors modulo p into distinct irreducible polynomials of degree d

Output: A random degree-d factor of $f(t)$

```
    1.  Set $g(t) \leftarrow f(t)$.
    2.  While $\deg(g) > d$
        2.1  Choose $u(t) \leftarrow$ a random monic polynomial of degree $2d - 1$.
        2.2  Compute via A.5.1
               $c(t) := u(t)^{(p^d - 1)/2} \bmod g(t)$.
        2.3  Set $h(t) \leftarrow \mathrm{GCD}(c(t) - 1, g(t))$.
        2.4  If $h(t)$ is constant or $\deg(g) = \deg(h)$, then go back to step 2.1.
        2.5  If $2 \deg(h) > \deg(g)$, then set $g(t) \leftarrow g(t)/h(t)$; else $g(t) \leftarrow h(t)$.
    3.  Output $g(t)$.
```

A.5.4 Factoring polynomials over GF (2) (special case)

Let $f(t)$ be a polynomial with coefficients in the field $GF\ (2)$, and suppose that $f(t)$ factors into distinct irreducible polynomials of degree d. (This is the special case needed in A.14.) The following algorithm finds a random degree-d factor of $f(t)$ efficiently.

Input: A positive integer d and a polynomial $f(t)$, which factors modulo 2 into distinct irreducible polynomials of degree d

Output: A random degree-d factor of $f(t)$

```
    1.  Set $g(t) \leftarrow f(t)$.
    2.  While $\deg(g) > d$
        2.1  Choose $u(t) \leftarrow$ a random monic polynomial of degree $2d - 1$.
        2.2  Set $c(t) \leftarrow u(t)$.
        2.3  For $i$ from 1 to $d - 1$ do
            2.3.1  $c(t) \leftarrow c(t)^2 + u(t) \bmod g(t)$.
        2.4  Compute $h(t) := \mathrm{GCD}(c(t), g(t))$ via A.5.2.
        2.5  If $h(t)$ is constant or $\deg(g) = \deg(h)$, then go back to step 2.1.
        2.6  If $2 \deg(h) > \deg(g)$, then set $g(t) \leftarrow g(t)/h(t)$; else $g(t) \leftarrow h(t)$.
    3.  Output $g(t)$.
```

A.5.5 Checking polynomials over GF (2^r) for irreducibility

If $f(t)$ is a polynomial with coefficients in the field $GF\ (2^r)$, then $f(t)$ can be tested efficiently for irreducibility using the following algorithm.

Input: A polynomial $f(t)$ with coefficients in $GF\ (2^r)$

Output: The message "True" if $f(t)$ is irreducible; the message "False" otherwise

```
1.  Set $d \leftarrow$ degree of $f(t)$.
2.  Set $u(t) \leftarrow t$.
3.  For $i$ from 1 to $\lfloor d/2 \rfloor$ do
    3.1  For $j$ from 1 to $r$ do
             Set $u(t) \leftarrow u(t)^2 \bmod f(t)$;
         Next $j$.
    3.2  Set $g(t) \leftarrow \mathrm{GCD}(u(t) + t, f(t))$.
    3.3  If $g(t) \neq 1$, then output "False" and stop.
4.  Output "True."
```

A.5.6 Finding a root in GF (2^m) of an irreducible binary polynomial

If $f(t)$ is an irreducible polynomial modulo 2 of degree d dividing m, then $f(t)$ has d distinct roots in the field $GF\ (2^m)$. A random root can be found efficiently using the following algorithm.

Input: An irreducible polynomial modulo 2 of degree d, and a field $GF\ (2^m)$, where d divides m

Output: A random root of $f(t)$ in $GF\ (2^m)$

```
1.  Set $g(t) \leftarrow f(t)$ [$g(t)$ is a polynomial over $GF\ (2^m)$].
2.  While $\deg(g) > 1$
    2.1  Choose random $u \in GF\ (2^m)$.
    2.2  Set $c(t) \leftarrow ut$.
    2.3  For $i$ from 1 to $m - 1$ do
        2.3.1  $c(t) \leftarrow c(t)^2 + ut \bmod g(t)$.
    2.4  Set $h(t) \leftarrow \mathrm{GCD}(c(t), g(t))$.
    2.5  If $h(t)$ is constant or $\deg(g) = \deg(h)$, then go back to step 2.1.
    2.6  If $2 \deg(h) > \deg(g)$, then set $g(t) \leftarrow g(t)/h(t)$; else $g(t) \leftarrow h(t)$.
3.  Output $g(0)$.
```

A.5.7 Embedding in an extension field

Given a field **F** = *GF* (2^d), the following algorithm embeds **F** into an extension field **K** = *GF* (2^{de}).

Input: Integers d and e; a (polynomial or normal) basis B for **F** = *GF* (2^d) with field polynomial $p(t)$; a (polynomial or normal) basis for **K** = *GF* (2^{de})

Output: An embedding of **F** into **K**; that is, a function taking each $\alpha \in \mathbf{F}$ to a corresponding element β of **K**

1. Compute via A.5.6 a root $\lambda \in \mathbf{K}$ of $p(t)$.
2. If B is a polynomial basis then output

   $$\beta := a_{m-1}\,\lambda^{m-1} + \ldots + a_2\lambda^2 + a_1\lambda + a_0$$

   where $(a_{m-1} \ldots a_1\, a_0)$ is the bit string representing α with respect to B.
3. If B is a normal basis then output

   $$\beta := a_0\lambda + a_1\lambda^2 + a_2\lambda^{2^2} + \ldots + a_{m-1}\lambda^{2^{m-1}}$$

   where $(a_0\, a_1 \ldots a_{m-1})$ is the bit string representing α with respect to B.

A.6 General normal bases for binary fields

The algorithms in this clause allow computation with any normal basis. (In the case of Gaussian normal bases, the algorithms of A.3 are more efficient.)

A general normal basis is specified by its field polynomial $p(t)$ (see A.3.5). The rule for multiplication with respect to a general normal basis is most easily described via a *multiplication matrix*. The multiplication matrix is an m-by-m matrix with entries in *GF* (2). A description of how to multiply using the multiplication matrix is given in A.6.4.

A.6.1 Checking for a normal basis

The following algorithm checks whether an irreducible polynomial $p(t)$ over *GF* (2) is normal and, if so, outputs two matrices to be used later in deriving the multiplication rule for the corresponding normal basis.

Input: An irreducible polynomial $p(t)$ of degree m over *GF* (2)

Output: If $p(t)$ is normal, two m-by-m matrices over *GF* (2). If not, the message “not normal.”

1. Compute

   $$t = a_{0,0} + a_{0,1}t + a_{0,2}t^2 + \ldots + a_{0,m-1}t^{m-1} \pmod{p(t)}$$
   $$t^2 = a_{1,0} + a_{1,1}t + a_{1,2}t^2 + \ldots + a_{1,m-1}t^{m-1} \pmod{p(t)}$$
   $$t^4 = a_{2,0} + a_{2,1}t + a_{2,2}t^2 + \ldots + a_{2,m-1}t^{m-1} \pmod{p(t)}$$

   ...

   $$t^{2^{m-1}} = a_{m-1,0} + a_{m-1,1}t + a_{m-1,2}t^2 + \ldots + a_{m-1,m-1}t^{m-1} \pmod{p(t)}$$

   modulo 2 via repeated squaring.

2. Set

$$A := \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,m-1} \end{bmatrix}$$

3. Attempt to compute the inverse *B* of the matrix *A* over *GF* (2).
4. If *A* is not invertible, output the message “not normal” and stop; otherwise output *A* and *B*.

A.6.2 Finding a normal basis

If *m* is a multiple of eight, then the Gaussian normal basis construction of A.3 does not apply. In this case (or in any case where a nonGaussian normal basis is desired), it is possible to find a normal basis by searching for a normal polynomial of degree *m* over *GF* (2).

The procedure is to produce an irreducible polynomial of degree *m* over *GF* (2), test it for normality via A.6.1, and repeat until a normal polynomial is found. To produce an irreducible, one can use the table in A.8.1, the random search routine in A.8.2, or one of the techniques in A.8.3 through A.8.5.

The probability of a randomly chosen irreducible polynomial of degree *m* over *GF* (2) being normal is at least 0.2 if $m \le 2000$, and is usually much higher. Thus, one should expect to try no more than about five irreducibles before finding a normal polynomial.

A.6.3 Computing the multiplication matrix

The following algorithm computes the multiplication matrix of a basis *B* that has been deemed “normal” by A.6.1.

Input: A normal polynomial

$$p(t) = t^m + c_{m-1}\, t^{m-1} + \ldots + c_1\, t + c_0$$

over *GF* (2); the matrices *A* and *B* computed by A.6.1.

Output: The multiplication matrix *M* for *p*(*t*)

1. Set

$$C := \begin{bmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \\ c_0 & c_1 & c_2 & \ldots & c_{m-1} \end{bmatrix}$$

2. Compute $D := ACB$ over *GF* (2). (Let $d_{i,j}$ denote the $(i, j)^{\text{th}}$ entry of *D*, for $0 \le i,j < m$.)
3. Set $\mu_{i,j} := d_{j-i,-i}$ for each *i* and *j*, where each subscript is regarded as an integer modulo *m*.

4. Output

$$M := \begin{bmatrix} \mu_{0,0} & \mu_{0,1} & \cdots & \mu_{0,m-1} \\ \mu_{1,0} & \mu_{1,1} & \cdots & \mu_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \mu_{m-1,0} & \mu_{m-1,1} & \cdots & \mu_{m-1,m-1} \end{bmatrix}$$

Example: Let $p(t) = t^4 + t^3 + t^2 + t + 1$. Then

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

so that

$$B = A^{-1} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Also

$$C = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

so that

$$D = ACB = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Therefore

$$M = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

In the case of a Gaussian normal basis, the multiplication matrix M can be written down immediately from the explicit multiplication rule. For $0 \le i < m$ and $0 \le j < m$, $\mu_{i,j}$ is 1 if $a_i b_j$ appears in the formula for c_0, and 0 otherwise.

Example: If B is the Type 3 normal basis over $GF\ (2^4)$, then it was found in A.3.7 that if

$$(c_0\ c_1\ \ldots\ c_{m-1}) = (a_0\ a_1 \ldots\ a_{m-1}) \times (b_0\ b_1\ \ldots\ b_{m-1})$$

then

$$c_0 = a_0\ (b_1 + b_2 + b_3) + a_1\ (b_0 + b_2) + a_2\ (b_0 + b_1) + a_3\ (b_0 + b_3)$$

Thus, the multiplication matrix for B is

$$M = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

A.6.4 Multiplication

The following algorithm implements the multiplication of two elements of a field $GF\ (2^m)$ represented in terms of a normal basis.

Input: The multiplication matrix M for the field $GF\ (2^m)$; field elements $a = (a_0\ a_1\ \ldots\ a_{m-1})$ and $b = (b_0\ b_1\ \ldots\ b_{m-1})$

Output: The product $c = (c_0\ c_1\ \ldots\ c_{m-1})$ of a and b

```
1.  Set $x \leftarrow a$.
2.  Set $y \leftarrow b$.
3.  For $k$ from 0 to $m - 1$ do
    3.1  Compute via matrix multiplication
             $c_k := x\ M\ y^{tr}$
         where $y^{tr}$ denotes the matrix transpose of the vector $y$.
    3.2  Set $x \leftarrow$ LeftShift($x$) and $y \leftarrow$ LeftShift($y$), where LeftShift denotes the
         circular left shift operation.
4.  Output $c = (c_0\ c_1\ \ldots\ c_{m-1})$.
```

Example: Let $\theta \in GF\ (2^4)$ be a root of the normal polynomial $p(t) = t^4 + t^3 + t^2 + t + 1$. Consider multiplying the elements

$$a = (1000) = \theta$$

and

$$b = (1101) = \theta + \theta^2 + \theta^8$$

The multiplication matrix for $p(t)$ was found in A.6.3 to be

$$M = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

Thus

$$c_0 = (1000)M\begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} = 0$$

$$c_1 = (0001)M\begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} = 1$$

$$c_2 = (0010)M\begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} = 1$$

$$c_3 = (0100)M\begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} = 1$$

so that $c = ab = (0111)$.

A.7 Basis conversion for binary fields

In order to use DL and EC protocols over a finite field $\mathbf{F} = GF\,(2^m)$, it is necessary for the users to agree on a common field degree m, but not on a common basis representation of **F**. If the users do not agree on the basis, however, it is necessary for one or both users to perform a *change-of-basis*; i.e., to convert incoming and/or outgoing field elements between one's own representation and that of the other user.

A.7.1 The change-of-basis matrix

Suppose that a user has received a field element that is represented in terms of the basis B_0, while the user himself uses basis B_1 for his own computations. This user must now perform a change-of-basis from B_0 to B_1 on this element. This can be accomplished using a change-of-basis matrix from B_0 to B_1. This is an m-by-m matrix with entries in $GF\,(2)$.

Suppose that Γ is the change-of-basis matrix from B_0 to B_1, and that $\underline{u}$ is the bit string representing $\alpha \in GF\ (2^m)$ in terms of B_0. Then

$$\underline{v} = \underline{u}\ \Gamma$$

is the bit string representing α in terms of B_1.

If Γ is the change-of-basis matrix from B_0 to B_1, then the inverse Γ^{-1} of the matrix over GF (2) is the change-of-basis matrix from B_1 to B_0. Thus, given a bit string $\underline{v}$ representing $\alpha \in GF\ (2^m)$ in terms of B_1, the bit string

$$\underline{u} = \underline{v}\ \Gamma^{-1}$$

represents α in terms of B_0.

If Γ is the change-of-basis matrix from B_0 to B_1, and Γ' is the change-of-basis matrix from B_1 to B_2, then the matrix product $\Gamma'' = \Gamma\ \Gamma'$ is the change-of-basis matrix from B_0 to B_2.

The algorithm for calculating the change-of-basis matrix is given in A.7.3. A special case is treated in A.7.4.

Example: Suppose that B_0 is the Type I optimal normal basis for $GF\ (2^4)$, and B_1 is the polynomial basis with field polynomial $t^4 + t + 1$. Then the change-of-basis matrix from B_0 to B_1 is

$$\Gamma = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

(see A.7.3). If α is the element of $GF\ (2^4)$ represented by (1001) with respect to B_0, then its representation with respect to B_1 is

$$(1001)\ \Gamma = (0100)$$

The inverse of Γ over GF (2) is

$$\Gamma^{-1} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

If β is the element of $GF\ (2^4)$ represented by (1101) with respect to B_1, then its representation with respect to B_0 is

$$(1101)\ \Gamma^{-1} = (0111)$$

A.7.2 The field polynomial of a Gaussian normal basis

In order to compute the change-of-basis matrix for conversion to or from a normal basis B, it is necessary to compute the field polynomial for that basis (see A.3.5).

If $GF(2^m)$ has a type T Gaussian normal basis over $GF(2)$, the following algorithm efficiently produces its field polynomial.

Input: Positive integers m and T for which there exists a type T Gaussian normal basis for $GF(2^m)$ over $GF(2)$

Output: The field polynomial $p(t)$ for the basis

```
I.   $T = 1$ (Type I optimal normal basis).
     1.  Set $p(t) \leftarrow t^m + t^{m-1} + \ldots + t + 1$.
     2.  Output $p(t)$.
II.  T = 2 (Type II optimal normal basis).
     1.  Set $q(t) \leftarrow 1$.
     2.  Set $p(t) \leftarrow t + 1$.
     3.  For $i = 1$ to $m - 1$ do
              $r(t) \leftarrow q(t)$
              $q(t) \leftarrow p(t)$
              $p(t) := t\, q(t) + r(t)$.
     4.  Output $p(t)$.
III. $T \geq 3$ (suboptimal Gaussian normal basis).
     1.  Set $p \leftarrow T m + 1$.
     2.  Generate via A.2.8 an integer $u$ having order $T$ modulo $p$.
     3.  For $k$ from 1 to $m$ do
         3.1 Compute
             $e_k = \sum_{j=0}^{T-1} \exp\left(\frac{2^k u^j \pi i}{p}\right)$
     4.  Compute the polynomial
             $f(t) := \prod_{k=1}^{m} (t - e_k)$
         (The polynomial $f(t)$ has integer coefficients.)
     5.  Output $p(t) := f(t) \bmod 2$.
```

NOTE—The complex numbers e_k must be computed with sufficient accuracy to identify each coefficient of the polynomial $f(t)$. Since each such coefficient is an integer, this means that the error incurred in calculating each coefficient should be less than 1/2.

Example: Let $m = 4$ and $T = 3$. By the example of A.3.6, there exists a Gaussian normal basis of Type 3 for $GF(2^4)$ over $GF(2)$. Now $p = 13$, and $u = 3$ has order $T = 3$ modulo $p = 13$.

The complex numbers e_k are

$$e_1 := e^{2\pi i/13} + e^{6\pi i/13} - e^{5\pi i/13}$$
$$\approx 0.6513878188659973233 + 0.522415803456407715\, i$$

$$e_2 := e^{4\pi i/13} + e^{12\pi i/13} + e^{10\pi i/13}$$
$$\approx -1.1513878188659973233 + 1.7254221884220093641\ i$$

$$e_3 := e^{8\pi i/13} - e^{11\pi i/13} - e^{7\pi i/13}$$
$$\approx 0.6513878188659973233 - 0.522415803456407715\ i$$

$$e_4 := -e^{\pi i/13} - e^{3\pi i/13} - e^{9\pi i/13}$$
$$\approx -1.1513878188659973233 - 1.7254221884220093641\ i$$

Thus

$$f(t) := (t - e_1)\,(t - e_2)\,(t - e_3)\,(t - e_4) = t^4 + t^3 + 2t^2 - 4t + 3$$

so that $p(t) := t^4 + t^3 + 1$ is the field polynomial for the basis.

A.7.3 Computing the change-of-basis matrix

The following algorithm efficiently calculates the change-of-basis matrix from a (polynomial or normal) basis B_0 to one's own (polynomial or normal) basis B_1.

Input: A field degree m; a (polynomial or normal) basis B_0 with field polynomial $p_0(u)$; a (polynomial or normal) basis B_1 with field polynomial $p_1(t)$. (Note that in the case of a Gaussian normal basis, the field polynomial can be computed via A.7.2.)

Output: The change-of-basis matrix Γ from B_0 to B_1

1. Let u be a root of $p_0(u)$ represented with respect to B_1 (u can be computed via A.5.6).
2. Define the elements $e_0, \ldots, e_{m-1}$ via

$$e_j = \begin{cases} t^{m-1-j} & \text{if} \quad B_1 = \{t^{m-1}, \ldots, t^2, t, 1\} \\ \theta^{2^j} & \text{if } B_1 = \{\theta, \theta^2, \theta^{2^2}, \ldots, \theta^{2^{m-1}}\} \end{cases}$$

3. Compute the elements $\gamma_{i,j}$ for $0 \le i < m$, $0 \le j < m$ as follows:

— If B_0 is a polynomial basis, then compute

$$1 = \sum_{j=0}^{m-1} \gamma_{m-1,j} e_j$$

$$u = \sum_{j=0}^{m-1} \gamma_{m-2,j} e_j$$

$$\vdots$$

$$u^{m-2} = \sum_{j=0}^{m-1} \gamma_{1,j} e_j$$

$$u^{m-1} = \sum_{j=0}^{m-1} \gamma_{0,j} e_j$$

by repeated multiplication by u.

— If B_0 is a normal basis, then compute

$$u = \sum_{j=0}^{m-1} \gamma_{0,j} e_j$$

$$u^2 = \sum_{j=0}^{m-1} \gamma_{1,j} e_j$$

$$u^{2^2} = \sum_{j=0}^{m-1} \gamma_{2,j} e_j$$

$$\vdots$$

$$u^{2^{m-1}} = \sum_{j=0}^{m-1} \gamma_{m-1,j} e_j$$

by repeated squaring.

4) Output

$$\Gamma \leftarrow \begin{bmatrix} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,m-1} \\ \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,m-1} \end{bmatrix}$$

Example: Suppose that B_0 is the Type I optimal normal basis for $GF(2^4)$, and B_1 is the polynomial basis modulo $p_1(t) = t^4 + t + 1$. Then a root of $p_0(u) = u^4 + u^3 + u^2 + u + 1$ is given by $u = t^3 + t^2$. By repeated squaring modulo $p_1(t)$

$$u = t^3 + t^2$$
$$u^2 = t^3 + t^2 + t + 1$$
$$u^4 = t^3 + t$$
$$u^8 = t^3$$

so that

$$\Gamma := \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Example: Suppose that B_0 is polynomial basis modulo $p_0(u) = u^5 + u^4 + u^2 + u + 1$, and B_1 is the polynomial basis modulo $p_1(t) = t^5 + t^2 + 1$. Then a root of $p_0(u)$ is given by $u = t + 1$. Thus

$1 = 1$
$u = t + 1$
$u^2 = t^2 + 1$
$u^3 = t^3 + t^2 + t + 1$
$u^4 = t^4 + 1$

so that

$$\Gamma := \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Example: Suppose that B_0 is polynomial basis modulo $p_0(u) = u^5 + u^2 + 1$, and B_1 is the Type II optimal normal basis for $GF\ (2^5)$. Then a root of $p_0(u)$ is given by $u = \theta^2 + \theta^4 + \theta^8 + \theta^{16}$. Thus

$1 = \theta + \theta^2 + \theta^4 + \theta^8 + \theta^{16}$
$u = \theta^2 + \theta^4 + \theta^8 + \theta^{16}$
$u^2 = \theta + \theta^4 + \theta^8 + \theta^{16}$
$u^3 = \theta + \theta^4 + \theta^{16}$
$u^4 = \theta + \theta^2 + \theta^8 + \theta^{16}$

so that

$$\Gamma := \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

NOTE—If both B_0 and B_1 are normal bases, then it is sufficient to perform the first computation

$$u = \sum_{j=0}^{m-1} \gamma_{0,j} e_j$$

and omit the rest. This expression provides the top row of the matrix Γ, and subsequent rows are obtained by right cyclic shifts.

Example: Suppose that B_0 is the Type 3 Gaussian normal basis for $GF\ (2^4)$, and B_1 is the Type I optimal normal basis for $GF\ (2^4)$. Then a root of $p_0(u) = u^4 + u^3 + 1$ is given by $u = \theta + \theta^4 + \theta^8$. Thus

$$\Gamma := \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

A.7.4 Conversion to a polynomial basis

If one is representing elements of $GF(2^m)$ with respect to the normal basis B_1, and it is desired to perform division via the Euclidean algorithm (see A.4.4), then it is necessary to construct a change-of-basis matrix from some polynomial basis B_0 to the basis B_1. If another means of division is not available, it is necessary to construct this matrix using an algorithm other than that of A.7.3 (which requires division in its first step). This can be done as follows:

Input: A field degree m; a normal basis B_1 with field polynomial $p(t)$

NOTE—if B_1 is a Gaussian normal basis, then $p(t)$ can be computed via A.7.2.

Output: The change-of-basis matrix Γ from B_0 to B_1, where B_0 is the polynomial basis with field polynomial $p(t)$

If B_1 is a Type I optimal normal basis, then the change-of-basis matrix is

$$\Gamma := \begin{bmatrix} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,m-1} \\ \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,m-1} \end{bmatrix}$$

where

$$\gamma_{i,j} = \begin{cases} 1 \text{ if } & i = m-1 \\ 1 \text{ if } & 2^j \equiv -(i+2) \pmod{m+1} \\ 0 & \text{otherwise} \end{cases}$$

In the general case, the algorithm of A.7.3 can be used, with step 1 replaced by

1'. Set $u \leftarrow \theta$ with respect to $B_1 = \{\theta, \theta^2, \theta^{2^2}, \ldots, \theta^{2^{m-1}}\}$

A.8 Bases for binary fields: tables and algorithms

A.8.1 Basis table

Table A.2 provides irreducible polynomials modulo 2 that are minimal, in the sense that they have as few terms as possible and that those terms are of the smallest possible degree. More precisely, if an irreducible binary trinomial $t^m + t^k + 1$ exists, then the minimal possible value of k is listed; if no such trinomial exists, then a pentanomial $t^m + t^a + t^b + t^c + 1$ is listed. In the latter case, the value of a is minimal; the value of b is minimal for the given a; and c is minimal for the given a and b. In addition, for each m not divisible by 8 is given the minimal type among Gaussian normal bases for $GF(2^m)$.

The optimal normal bases are Type 1 and Type 2 Gaussian normal bases (see A.3.5). In the cases where both Type 1 and Type 2 bases exist, they are both listed in Table A.2.

Table A.2—Basis table

m	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type
2	1	1,2	252	15	3	502	8, 5, 4	10	752	13, 10, 3	—
3	1	2	253	46	10	503	3	6	753	158	16
4	1	1	254	7, 2, 1	2	504	15, 14, 6	—	754	19	10
5	2	2	255	52	6	505	156	10	755	12, 10, 1	2
6	1	2	256	10, 5, 2	—	506	23	5	756	45	1
7	1	4	257	12	6	507	13, 6, 3	4	757	7, 6, 1	16
8	4, 3, 1	—	258	71	5	508	9	1	758	233	6
9	1	2	259	10, 6, 2	10	509	8, 7, 3	2	759	98	4
10	3	1	260	15	5	510	69	3	760	11, 6, 5	—
11	2	2	261	7, 6, 4	2	511	10	6	761	3	2
12	3	1	262	9, 8, 4	3	512	8, 5, 2	—	762	83	10
13	4, 3, 1	4	263	93	6	513	26	4	763	16, 14, 9	22
14	5	2	264	9, 6, 2	—	514	67	33	764	6, 5, 3	3
15	1	4	265	42	4	515	14, 7, 4	2	765	9, 7, 4	2
16	5, 3, 1	—	266	47	6	516	21	3	766	22, 19, 9	6
17	3	6	267	8, 6, 3	8	517	12, 10, 2	4	767	168	6
18	3	1,2	268	25	1	518	33	14	768	19, 17, 4	—
19	5, 2, 1	10	269	7, 6, 1	8	519	79	2	769	120	10
20	3	3	270	53	2	520	15, 11, 2	—	770	14, 5, 2	5
21	2	10	271	58	6	521	32	32	771	17, 15, 6	2
22	1	3	272	9, 3, 2	—	522	39	1	772	7	1
23	5	2	273	23	2	523	13, 6, 2	10	773	10, 8, 6	6
24	4, 3, 1	—	274	67	9	524	167	5	774	185	2
25	3	4	275	11, 10, 9	14	525	6, 4, 1	8	775	93	6
26	4, 3, 1	2	276	63	3	526	97	3	776	15, 14, 7	—
27	5, 2, 1	6	277	12, 6, 3	4	527	47	6	777	29	16
28	1	1	278	5	2	528	11, 6, 2	—	778	375	21
29	2	2	279	5	4	529	42	24	779	10, 8, 3	2
30	1	2	280	9, 5, 2	—	530	10, 7, 3	2	780	13	13
31	3	10	281	93	2	531	10, 5, 4	2	781	17, 16, 2	16
32	7, 3, 2	—	282	35	6	532	1	3	782	329	3
33	10	2	283	12, 7, 5	6	533	4, 3, 2	12	783	68	2
34	7	9	284	53	3	534	161	7	784	13, 9, 6	—
35	2	2	285	10, 7, 5	10	535	8, 6, 2	4	785	92	2
36	9	1	286	69	3	536	7, 5, 3	—	786	12, 10, 3	1

Table A.2—Basis table (continued)

m	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type
37	6, 4, 1	4	287	71	6	537	94	8	787	7, 6, 3	6
38	6, 5, 1	6	288	11, 10, 1	—	538	195	6	788	17, 10, 3	11
39	4	2	289	21	12	539	10, 5, 4	12	789	5, 2, 1	14
40	5, 4, 3	—	290	5, 3, 2	5	540	9	1	790	9, 6, 1	3
41	3	2	291	12, 11, 5	6	541	13, 10, 4	18	791	30	2
42	7	5	292	37	1	542	8, 6, 1	3	792	9, 7, 3	—
43	6, 4, 3	4	293	11, 6, 1	2	543	16	2	793	253	6
44	5	9	294	33	3	544	8, 3, 1	—	794	143	14
45	4, 3, 1	4	295	48	16	545	122	2	795	7, 4, 1	10
46	1	3	296	7, 3, 2	—	546	8, 2, 1	1	796	9, 4, 1	1
47	5	6	297	5	6	547	13, 7, 4	10	797	12, 10, 4	6
48	5, 3, 2	—	298	11, 8, 4	6	548	10, 5, 3	5	798	53	6
49	9	4	299	11, 6, 4	2	549	16, 4, 3	14	799	25	22
50	4, 3, 2	2	300	5	19	550	193	7	800	9, 7, 1	—
51	6, 3, 1	2	301	9, 5, 2	10	551	135	6	801	217	12
52	3	1	302	41	3	552	19, 16, 9	—	802	15, 13, 9	6
53	6, 2, 1	2	303	1	2	553	39	4	803	14, 9, 2	2
54	9	3	304	11, 2, 1	—	554	10, 8, 7	2	804	75	5
55	7	12	305	102	6	555	10, 9, 4	4	805	8, 7, 2	6
56	7, 4, 2	—	306	7, 3, 1	2	556	153	1	806	21	11
57	4	10	307	8, 4, 2	4	557	7, 6, 5	6	807	7	14
58	19	1	308	15	15	558	73	2	808	14, 3, 2	—
59	7, 4, 2	12	309	10, 6, 4	2	559	34	4	809	15	2
60	1	1	310	93	6	560	11, 9, 6	—	810	159	2
61	5, 2, 1	6	311	7, 5, 3	6	561	71	2	811	12, 10, 8	10
62	29	6	312	9, 7, 4	—	562	11, 4, 2	1	812	29	3
63	1	6	313	79	6	563	14, 7, 3	14	813	10, 3, 1	4
64	4, 3, 1	—	314	15	5	564	163	3	814	21	15
65	18	2	315	10, 9, 1	8	565	11, 6, 1	10	815	333	8
66	3	1	316	63	1	566	153	3	816	11, 8, 2	—
67	5, 2, 1	4	317	7, 4, 2	26	567	28	4	817	52	6
68	9	9	318	45	11	568	15, 7, 6	—	818	119	2
69	6, 5, 2	2	319	36	4	569	77	12	819	16, 9, 7	20
70	5, 3, 1	3	320	4, 3, 1	—	570	67	5	820	123	1
71	6	8	321	31	12	571	10, 5, 2	10	821	15, 11, 2	8

Table A.2—Basis table (continued)

m	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type
72	10, 9, 3	—	322	67	6	572	12, 8, 1	5	822	17	3
73	25	4	323	10, 3, 1	2	573	10, 6, 4	4	823	9	10
74	35	2	324	51	5	574	13	3	824	11, 6, 4	—
75	6, 3, 1	10	325	10, 5, 2	4	575	146	2	825	38	6
76	21	3	326	10, 3, 1	2	576	13, 4, 3	—	826	255	1
77	6, 5, 2	6	327	34	8	577	25	4	827	12, 10, 7	14
78	6, 5, 3	7	328	8, 3, 1	—	578	23, 22,16	6	828	189	1
79	9	4	329	50	2	579	12, 9, 7	10	829	4, 3, 1	10
80	9, 4, 2	—	330	99	2	580	237	3	830	17, 10, 7	14
81	4	2	331	10, 6, 2	6	581	13, 7, 6	8	831	49	2
82	8, 3, 1	1	332	89	3	582	85	3	832	13, 5, 2	—
83	7, 4, 2	2	333	2	24	583	130	4	833	149	2
84	5	5	334	5, 2, 1	7	584	14, 13, 3	—	834	15	2
85	8, 2, 1	12	335	10, 7, 2	12	585	88	2	835	14, 7, 5	6
86	21	2	336	7, 4, 1	—	586	7, 5, 2	1	836	10, 9, 2	15
87	13	4	337	55	10	587	11, 6, 1	14	837	8, 6, 5	6
88	7, 6, 2	—	338	4, 3, 1	2	588	35	11	838	61	7
89	38	2	339	16, 10, 7	8	589	10, 4, 3	4	839	54	12
90	27	2	340	45	3	590	93	11	840	11, 5, 1	—
91	8, 5, 1	6	341	10, 8, 6	8	591	9, 6, 4	6	841	144	12
92	21	3	342	125	6	592	13, 6, 3	—	842	47	5
93	2	4	343	75	4	593	86	2	843	11, 10, 7	6
94	21	3	344	7, 2, 1	—	594	19	17	844	105	13
95	11	2	345	22	4	595	9, 2, 1	6	845	2	8
96	10, 9, 6	—	346	63	1	596	273	3	846	105	2
97	6	4	347	11, 10, 3	6	597	14, 12, 9	4	847	136	30
98	11	2	348	103	1	598	7, 6, 1	15	848	11, 4, 1	—
99	6, 3, 1	2	349	6, 5, 2	10	599	30	8	849	253	8
100	15	1	350	53	2	600	9, 5, 2	—	850	111	6
101	7, 6, 1	6	351	34	10	601	201	6	851	13, 10, 5	6
102	29	6	352	13, 11, 6	—	602	215	5	852	159	1
103	9	6	353	69	14	603	6, 4, 3	12	853	10, 7, 1	4
104	4, 3, 1	—	354	99	2	604	105	7	854	7, 5, 3	18
105	4	2	355	6, 5, 1	6	605	10, 7, 5	6	855	29	8
106	15	1	356	10, 9, 7	3	606	165	2	856	19, 10, 3	-

Table A.2—Basis table (continued)

m	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type
107	9, 7, 4	6	357	11, 10, 2	10	607	105	6	857	119	8
108	17	5	358	57	10	608	19, 13, 6	—	858	207	1
109	5, 4, 2	10	359	68	2	609	31	4	859	17, 15, 4	22
110	33	6	360	5, 3, 2	—	610	127	10	860	35	9
111	10	20	361	7, 4, 1	30	611	10, 4, 2	2	861	14	28
112	5, 4, 3	—	362	63	5	612	81	1	862	349	31
113	9	2	363	8, 5, 3	4	613	19, 10, 4	10	863	6, 3, 2	6
114	5, 3, 2	5	364	9	3	614	45	2	864	21, 10, 6	—
115	8, 7, 5	4	365	9, 6, 5	24	615	211	2	865	1	4
116	4, 2, 1	3	366	29	22	616	19, 10, 3	—	866	75	2
117	5, 2, 1	8	367	21	6	617	200	8	867	9, 5, 2	4
118	33	6	368	7, 3, 2	—	618	295	1,2	868	145	19
119	8	2	369	91	10	619	9, 8, 5	4	869	11, 7, 6	12
120	4, 3, 1	—	370	139	6	620	9	3	870	301	2
121	18	6	371	8, 3, 2	2	621	12, 6, 5	6	871	378	6
122	6, 2, 1	6	372	111	1	622	297	3	872	13, 3, 1	—
123	2	10	373	8, 7, 2	4	623	68	12	873	352	2
124	19	3	374	8, 6, 5	3	624	11, 6, 5	—	874	12, 7, 4	9
125	7, 6, 5	6	375	16	2	625	133	36	875	12, 8, 1	12
126	21	3	376	8, 7, 5	—	626	251	21	876	149	1
127	1	4	377	41	14	627	13, 8, 4	20	877	6, 5, 4	16
128	7, 2, 1	—	378	43	1,2	628	223	7	878	12, 9, 8	15
129	5	8	379	10, 8, 5	12	629	6, 5, 2	2	879	11	2
130	3	1	380	47	5	630	7, 4, 2	14	880	15, 7, 5	—
131	8, 3, 2	2	381	5, 2, 1	8	631	307	10	881	78	18
132	17	5	382	81	6	632	9, 2, 1	—	882	99	1
133	9, 8, 2	12	383	90	12	633	101	34	883	17, 16, 12	4
134	57	2	384	12, 3, 2	—	634	39	13	884	173	27
135	11	2	385	6	6	635	14, 10, 4	8	885	8, 7, 1	28
136	5, 3, 2	—	386	83	2	636	217	13	886	13, 9, 8	3
137	21	6	387	8, 7, 1	4	637	14, 9, 1	4	887	147	6
138	8, 7, 1	1	388	159	1	638	6, 5, 1	2	888	19, 18, 10	—
139	8, 5, 3	4	389	10, 9, 5	24	639	16	2	889	127	4
140	15	3	390	9	3	640	14, 3, 2	—	890	183	5
141	10, 4, 1	8	391	28	6	641	11	2	891	12, 4, 1	2

Table A.2—Basis table (continued)

m	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type
142	21	6	392	13, 10, 6	—	642	119	6	892	31	3
143	5, 3, 2	6	393	7	2	643	11, 3, 2	12	893	11, 8, 6	2
144	7, 4, 2	—	394	135	9	644	11, 6, 5	3	894	173	3
145	52	10	395	11, 6, 5	6	645	11, 8, 4	2	895	12	4
146	71	2	396	25	11	646	249	6	896	7, 5, 3	—
147	14	6	397	12, 7, 6	6	647	5	14	897	113	8
148	27	1	398	7, 6, 2	2	648	13, 3, 1	—	898	207	21
149	10, 9, 7	8	399	26	12	649	37	10	899	18, 15, 5	8
150	53	19	400	5, 3, 2	—	650	3	2	900	1	11
151	3	6	401	152	8	651	14	2	901	13, 7, 6	6
152	6, 3, 2	—	402	171	5	652	93	1	902	21	3
153	1	4	403	9, 8, 5	16	653	10, 8, 7	2	903	35	4
154	15	25	404	65	3	654	33	14	904	12, 7, 2	—
155	62	2	405	13, 8, 2	4	655	88	4	905	117	6
156	9	13	406	141	6	656	7, 5, 4	—	906	123	1
157	6, 5, 2	10	407	71	8	657	38	10	907	12, 10, 2	6
158	8, 6, 5	2	408	5, 3, 2	—	658	55	1	908	143	21
159	31	22	409	87	4	659	15, 4, 2	2	909	14, 4, 1	4
160	5, 3, 2	—	410	10, 4, 3	2	660	11	1	910	15, 9, 7	18
161	18	6	411	12, 10, 3	2	661	12, 11, 4	6	911	204	2
162	27	1	412	147	3	662	21	3	912	7, 5, 1	—
163	7, 6, 3	4	413	10, 7, 6	2	663	107	14	913	91	6
164	10, 8, 7	5	414	13	2	664	11, 9, 8	—	914	4, 2, 1	18
165	9, 8, 3	4	415	102	28	665	33	14	915	8, 6, 3	10
166	37	3	416	9, 5, 2	—	666	10, 7, 2	22	916	183	3
167	6	14	417	107	4	667	18, 7, 3	6	917	12, 10, 7	6
168	15, 3, 2	—	418	199	1	668	147	11	918	77	10
169	34	4	419	15, 5, 4	2	669	5, 4, 2	4	919	36	4
170	11	6	420	7	1	670	153	6	920	14, 9, 6	—
171	6, 5, 2	12	421	5, 4, 2	10	671	15	6	921	221	6
172	1	1	422	149	11	672	11, 6, 5	—	922	7, 6, 5	10
173	8, 5, 2	2	423	25	4	673	28	4	923	16, 14, 13	2
174	13	2	424	9, 7, 2	—	674	11, 7, 4	5	924	31	5
175	6	4	425	12	6	675	6, 3, 1	22	925	16, 15, 7	4
176	11, 3, 2	—	426	63	2	676	31	1	926	365	6

Table A.2—Basis table (continued)

m	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type
177	8	4	427	11, 6, 5	16	677	8, 4, 3	8	927	403	4
178	31	1	428	105	5	678	15, 5, 3	10	928	10, 3, 2	—
179	4, 2, 1	2	429	10, 8, 7	2	679	66	10	929	11, 4, 3	8
180	3	1	430	14, 6, 1	3	680	23, 16, 9	—	930	31	2
181	7, 6, 1	6	431	120	2	681	11, 9, 3	22	931	10, 9, 4	10
182	81	3	432	13, 4, 3	—	682	171	6	932	177	3
183	56	2	433	33	4	683	11, 6, 1	2	933	16, 6, 1	2
184	9, 8, 7	—	434	12, 11, 5	9	684	209	3	934	22, 6, 5	3
185	24	8	435	12, 9, 5	4	685	4, 3, 1	4	935	417	2
186	11	2	436	165	13	686	197	2	936	15, 13, 12	—
187	7, 6, 5	6	437	6, 2, 1	18	687	13	10	937	217	6
188	6, 5, 2	5	438	65	2	688	19, 14, 6	—	938	207	2
189	6, 5, 2	2	439	49	10	689	14	12	939	7, 5, 4	2
190	8, 7, 6	10	440	4, 3, 1	—	690	79	2	940	10, 7, 1	1
191	9	2	441	7	2	691	13, 6, 2	10	941	11, 6, 1	6
192	7, 2, 1	—	442	7, 5, 2	1	692	299	5	942	45	10
193	15	4	443	10, 6, 1	2	693	15, 8, 2	6	943	24	6
194	87	2	444	81	5	694	169	3	944	12, 11, 9	—
195	8, 3, 2	6	445	7, 6, 4	6	695	177	18	945	77	8
196	3	1	446	105	6	696	23, 10, 2	—	946	21, 20, 13	1
197	9, 4, 2	18	447	73	6	697	267	4	947	9, 6, 5	6
198	9	22	448	11, 6, 4	—	698	215	5	948	189	7
199	34	4	449	134	8	699	15, 10, 1	4	949	8, 3, 2	4
200	5, 3, 2	—	450	47	13	700	75	1	950	13, 12, 10	2
201	14	8	451	16, 10, 1	6	701	16, 4, 2	18	951	260	16
202	55	6	452	6, 5, 4	11	702	37	14	952	16, 9, 7	—
203	8, 7, 1	12	453	15, 6, 4	2	703	12, 7, 1	6	953	168	2
204	27	3	454	8, 6, 1	19	704	8, 3, 2	—	954	131	49
205	9, 5, 2	4	455	38	26	705	17	6	955	7, 6, 3	10
206	10, 9, 5	3	456	18, 9, 6	—	706	12, 11, 8	21	956	305	15
207	43	4	457	16	30	707	15, 8, 5	6	957	10, 9, 6	6
208	9, 3, 1	—	458	203	6	708	15	1	958	13, 9, 4	6
209	6	2	459	12, 5, 2	8	709	4, 3, 1	4	959	143	8
210	7	1,2	460	19	1	710	13, 12, 4	3	960	12, 9, 3	—
211	11, 10, 8	10	461	7, 6, 1	6	711	92	8	961	18	16

Table A.2—Basis table (continued)

m	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type
212	105	5	462	73	10	712	5, 4, 3	—	962	15, 8, 5	14
213	6, 5, 2	4	463	93	12	713	41	2	963	20, 9, 6	4
214	73	3	464	19, 18, 13	—	714	23	5	964	103	9
215	23	6	465	31	4	715	7, 4, 1	4	965	15, 4, 2	2
216	7, 3, 1	—	466	14, 11, 6	1	716	183	5	966	201	7
217	45	6	467	11, 6, 1	6	717	16, 7, 1	18	967	36	16
218	11	5	468	27	21	718	165	15	968	9, 5, 2	—
219	8, 4, 1	4	469	9, 5, 2	4	719	150	2	969	31	4
220	7	3	470	9	2	720	9, 6, 4	—	970	11, 7, 2	9
221	8, 6, 2	2	471	1	8	721	9	6	971	6, 2, 1	6
222	5, 4, 2	10	472	11, 3, 2	—	722	231	26	972	7	5
223	33	12	473	200	2	723	16, 10, 4	2	973	13, 6, 4	6
224	9, 8, 3	—	474	191	5	724	207	13	974	9, 8, 7	2
225	32	22	475	9, 8, 4	4	725	9, 6, 5	2	975	19	2
226	10, 7, 3	1	476	9	5	726	5	2	976	17, 10, 6	—
227	10, 9, 4	24	477	16, 15, 7	46	727	180	4	977	15	8
228	113	9	478	121	7	728	4, 3, 2	—	978	9, 3, 1	6
229	10, 4, 1	12	479	104	8	729	58	24	979	178	4
230	8, 7, 6	2	480	15, 9, 6	—	730	147	13	980	8, 7, 6	9
231	26	2	481	138	6	731	8, 6, 2	8	981	12, 6, 5	32
232	9, 4, 2	—	482	9, 6, 5	5	732	343	11	982	177	15
233	74	2	483	9, 6, 4	2	733	8, 7, 2	10	983	230	14
234	31	5	484	105	3	734	11, 6, 1	3	984	24, 9, 3	—
235	9, 6, 1	4	485	17, 16, 6	18	735	44	8	985	222	10
236	5	3	486	81	10	736	13, 8, 6	—	986	3	2
237	7, 4, 1	10	487	94	4	737	5	6	987	16, 13, 12	6
238	73	7	488	4, 3, 1	—	738	347	5	988	121	7
239	36	2	489	83	12	739	18, 16, 8	4	989	10, 4, 2	2
240	8, 5, 3	—	490	219	1	740	135	3	990	161	10
241	70	6	491	11, 6, 3	2	741	9, 8, 3	2	991	39	18
242	95	6	492	7	13	742	85	15	992	17, 15, 13	—
243	8, 5, 1	2	493	10, 5, 3	4	743	90	2	993	62	2
244	111	3	494	17	3	744	13, 11, 1	—	994	223	10
245	6, 4, 1	2	495	76	2	745	258	10	995	15, 12, 2	14
246	11, 2, 1	11	496	16, 5, 2	—	746	351	2	996	65	43

Table A.2—Basis table (continued)

m	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type	*m*	Coeff	GB type
247	82	6	497	78	20	747	10, 6, 4	6	997	12, 6, 3	4
248	15, 14, 10	—	498	155	9	748	19	7	998	101	2
249	35	8	499	11, 6, 5	4	749	7, 6, 1	2	999	59	8
250	103	9	500	27	11	750	309	14	1000	5, 4, 3	—
251	7, 4, 2	2	501	5, 4, 2	10	751	18	6			

A.8.2 Random search for other irreducible polynomials

The basic method for producing an irreducible polynomial modulo 2 of degree *m* is as follows:

- Generate a random polynomial (perhaps subject to certain conditions, such as number of terms).
- Test for irreducibility (via A.5.5 with $r = 1$).
- Repeat until an irreducible is found.

The polynomial $f(t)$ should satisfy the following three conditions, since these are necessary for irreducibility modulo 2:

- $f(t)$ must have an odd number of (nonzero) terms.
- The constant term of $f(t)$ must be 1.
- There must be at least one odd-degree term.

If a random $f(t)$ of degree at most *m* satisfies these conditions, then the probability that $f(t)$ is irreducible is roughly $4/m$. Thus, one can expect to find an irreducible after $\approx m/4$ random tries.

A.8.3 Irreducibles from other irreducibles

If $f(t)$ is an irreducible of degree *m*, then so are the following five polynomials:

$$g(t) = t^m f(1/t)$$
$$h(t) = g(t+1)$$
$$j(t) = t^m h(1/t)$$
$$k(t) = j(t+1)$$
$$\begin{aligned} l(t) &= t^m k(1/t) \\ &= f(t+1) \end{aligned}$$

Example:

$$f(t) = t^5 + t^2 + 1$$
$$g(t) = t^5 + t^3 + 1$$
$$h(t) = t^5 + t^4 + t^3 + t^2 + 1$$

$$j(t) = t^5 + t^3 + t^2 + t + 1$$
$$k(t) = t^5 + t^4 + t^3 + t + 1$$
$$l(t) = t^5 + t^4 + t^2 + t + 1$$

Another construction is as follows. Suppose that $f(t)$ is an irreducible of odd degree m. Define the polynomials $h_0(t)$, $h_1(t)$, $h_2(t)$ by

$$f(t) = h_0(t^3) + t\, h_1(t^3) + t^2\, h_2(t^3)$$

Then

$$g(t) = h_0(t)^3 + t\, h_1(t)^3 + t^2\, h_2(t)^3 + t\, h_0(t)\, h_1(t)\, h_2(t)$$

is an irreducible of degree m.

More generally, if r is an odd number relatively prime to $2^m - 1$, then define $h_0(t), \ldots, h_{r-1}(t)$ by

$$f(t) = \sum_{k=0}^{r-1} t^k h_k(t^r)$$

Define the functions

$$w_k(t) = t^{k/r}\, h_k(t)$$

Let M be the r-by-r matrix with entries

$$M_{i,j} = w_{i-j}(t)$$

where the subscripts of w are taken modulo r. Then, the determinant of M is an irreducible of degree m.

A.8.4 Irreducibles of even degree

If $m = 2d$, then one can produce an irreducible of degree m more efficiently by generating an irreducible of degree d and using the following *degree-doubling* technique.

If

$$p(t) = t^d + t^{d-1} + 1 + \sum_i t^{k_i}$$

is irreducible, then so is

$$P(t) = t^{2d} + t^d + t^{2d-2} + t^{d-1} + 1 + \sum_i (t^{2k_i} + t^{k_i})$$

Note that the sum over i is allowed to be empty.

Examples:

$p(t) = t^3 + t^2 + 1$

$P(t) =t^6 + t^4 + t^3 + t^2 + 1$

$p(t) = t^5 + t^4 + t^3 +t + 1$

$P(t) = t^{10} + t^8 + t^6 + t^5 + t^4 +t^3 + t^2 + t + 1$

$p(t) = t^5 + t^4 + t^2 + t + 1$

$P(t) = t^{10} + t^8 + t^5 + t + 1$

If an irreducible $f(t)$ is not of the form required for degree doubling, then it may be used to construct one that is suitable via the methods of A.8.3. In particular, if $m = 2d$ with d odd, then either $f(t)$ or $f(t + 1)$ is of the suitable form.

Input: An irreducible $f(t)$ of odd degree d

Output: An irreducible $P(t)$ of degree $2d$

1. If $f(t)$ has a degree $d – 1$ term, then set $p(t) \leftarrow f(t)$; else set $p(t) \leftarrow f(t + 1)$.
2. Apply degree doubling to $p(t)$ and output the result.

The degree doubling technique can often be applied more than once. The following algorithm treats a family of cases in which this is possible.

Input: An irreducible $f(t)$ of odd degree d satisfying at least one of the following three conditions:

1. $f(t)$ contains both degree 1 and degree $d – 1$ terms.
2. $f(t)$ contains a degree 1 term and contains an even number of odd-degree terms in all.
3. $f(t)$ contains a degree $d – 1$ term and contains an odd number of odd-degree terms in all.

Output: An irreducible of degree $4d$

1. If $f(t)$ satisfies condition 3 above, then set $p(t) \leftarrow f(t)$; else set $p(t) \leftarrow t^d f(1/t)$.
2. Apply degree doubling to $p(t)$ to produce an irreducible $P(t)$ of degree $2d$.
3. Set $g(t) \leftarrow P(t + 1)$.
4. Set $h(t) \leftarrow t^{2d} g(1/t)$.
5. Apply degree doubling to $h(t)$ and output the result.

A.8.5 Irreducible trinomials

A search for irreducible trinomials should take into account the following restrictions on which trinomials can be irreducible modulo 2.

Suppose the trinomial to be checked is $f(t) = t^m + t^k + 1$, where $m > k > 0$. Note that since $f(t)$ is irreducible if, and only if, $t^m + t^{m–k} + 1$ is irreducible, then only half the possible k need to be checked. For instance, it is sufficient to check for $k \le m/2$. Then $f(t)$ is reducible in all of the following cases:

— m and k are both even.

— m is a multiple of 8.

— $m = hn$ and $k = 2h$ or $(n - 2)h$, for some odd numbers n, h, such that $n \equiv \pm h \pmod 8$.

— $k = 2h$ or $m - 2h$ for some h not dividing m, and $m \equiv \pm 3 \pmod 8$.

— $m = 2n$, where n is odd and $n \equiv k \pmod 4$, *except* for the polynomials $t^{2k} + t^k + 1$ with k a power of 3.
(All of these polynomials are *irreducible*.)

A.9 Elliptic curves: overview

A.9.1 Introduction

A *plane curve* is defined to be the set of points satisfying an equation $F(x, y) = 0$. The simplest plane curves are lines (whose defining equation has degree 1 in x and y) and conic sections (degree 2 in x and y). The next simplest are the cubic curves (degree 3). These include *elliptic curves*, so called because they arose historically from the problem of computing the circumference of an ellipse.

NOTE—See Silverman [B139] for a mathematically precise definition of “elliptic curve.” This standard restricts its attention to cubic plane curves, although other representations could be defined. The coefficients of such a curve must satisfy a side condition to guarantee the mathematical property of *nonsingularity*. The side condition is given below for each family of curves.

In cryptography, the elliptic curves of interest are those defined over finite fields. That is, the coefficients of the defining equation $F(x, y) = 0$ are elements of $GF(q)$, and the points on the curve are of the form $P = (x, y)$, where x and y are elements of $GF(q)$. Examples are given below.

The Weierstrass equation

There are several kinds of defining equations for elliptic curves, but the most common are the *Weierstrass equations*.

— For the prime finite fields $GF(p)$ with $p > 3$, the Weierstrass equation is

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

where a and b are integers modulo p for which $4a^3 + 27b^2 \not\equiv 0 \pmod p$.

— For the binary finite fields $GF(2^m)$, the Weierstrass equation is

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

where a and b are elements of $GF(2^m)$ with $b \neq 0$.

NOTE—There is another kind of Weierstrass equation over $GF(2^m)$, giving what are called *supersingular* curves. However, these curves are cryptographically weak (see Menezes, Okamoto, and Vanstone [B111]); thus, they are omitted from this standard.

Given a Weierstrass equation, the elliptic curve E consists of the solutions (x, y) over $GF(q)$ to the defining equation, along with an additional element called the *point at infinity* (denoted O). The points other than O are called *finite* points. The number of points on E (including O) is called the *order* of E and is denoted by $\#E(GF(q))$.

Example: Let E be the curve

$$y^2 = x^3 + 10\,x + 5$$

over the field $GF\,(13)$. Then the points on E are

$$\{\mathsf{O}, (1,4), (1,9), (3,6), (3,7), (8,5), (8,8), (10,0), (11,4), (11,9)\}$$

Thus, the order of E is $\#E\,(GF\,(13)) = 10$.

Example: Let E be the curve

$$y^2 + xy = x^3 + (t + 1)\,x^2 + 1$$

over the field $GF\,(2^3)$ given by the polynomial basis with field polynomial $t^3 + t + 1 = 0$. Then the points on E are

$$\begin{aligned}&\{\mathsf{O}, ((000), (001))\\&((010), (100)), ((010), (110)), ((011), (100)), ((011), (111)),\\&((100), (001)), ((100), (101)), ((101), (010)), ((101), (111)),\\&((110), (000)), ((110), (110)), ((111), (001)), ((111), (110))\}\end{aligned}$$

Thus, the order of E is $\#E\,(GF\,(2^3)) = 14$.

For more information on elliptic curve cryptography, see Menezes [B109].

A.9.2 Operations on elliptic curves

There is an *addition operation* on the points of an elliptic curve that possesses the algebraic properties of ordinary addition (e.g., commutativity and associativity). This operation can be described geometrically as follows:

Define the *inverse* of the point $P = (x, y)$ to be

$$-P = \left(\begin{array}{l}(x,-y) \text{ if } q = p \text{ prime,}\\ \quad (x, x + y) \text{ if } q = 2^m\end{array}\right.$$

Then, the *sum* $P + Q$ of the points P and Q is the point R, with the property that P, Q, and $-R$ lie on a common line.

The point at infinity

The point at infinity O plays a role analogous to that of the number 0 in ordinary addition. Thus

$$P + \mathsf{O} = P$$

$$P + (-P) = \mathsf{O}$$

for all points P.

Full addition

When implementing the formulas for elliptic curve addition, it is necessary to distinguish between *doubling* (adding a point to itself) and *adding* two distinct points that are not inverses of each other, because the formulas are different in the two cases. Besides this, there are also the special cases involving O. By *full addition* is meant choosing and implementing the appropriate formula for the given pair of points. Algorithms for full addition are given in A.10.1, A.10.2, and A.10.8.

Scalar multiplication

Elliptic curve points can be added but not *multiplied*. It is, however, possible to perform *scalar multiplication*, which is another name for repeated addition of the same point. If n is a positive integer and P a point on an elliptic curve, the scalar multiple nP is the result of adding n copies of P. Thus, for example, $5P = P + P + P + P + P$.

The notion of scalar multiplication can be extended to zero and the negative integers via

$$0P = \mathrm{O}, \qquad (-n)\, P = n\, (-P)$$

A.9.3 Elliptic curve cryptography

Orders

The *order* of a point P on an elliptic curve is the smallest positive integer r such that $rP = \mathrm{O}$. The order always exists and divides the order of the curve $\#E(GF\,(q))$. If k and l are integers, then $kP = lP$ if, and only if, $k \equiv l \pmod r$.

Elliptic curve discrete logarithms

Suppose that the point G on E has prime order r, where r^2 does not divide the order of the curve $\#E(GF\,(q))$. Then, a point P satisfies $P = lG$ for some l if, and only if, $rP = \mathrm{O}$. The coefficient l is called the *elliptic curve discrete logarithm* of P (with respect to the base point G). The elliptic curve discrete logarithm is an integer modulo r.

EC-based cryptography

Suppose that the base point G on E has order r as described in the preceding paragraph. Then a key pair can be defined as follows:

— The private key s is an integer modulo r.

— The corresponding public key W is a point on E defined by $W := sG$.

It is necessary to compute an elliptic curve discrete logarithm in order to derive a private key from its corresponding public key. For this reason, public-key cryptography based on key pairs of this type relies for its security on the difficulty of the elliptic curve discrete logarithm problem. Thus, it is an example of *EC-based cryptography.* The difficulty of the elliptic curve discrete logarithm problem is discussed in D.4.2.

A.9.4 Analogies with DL

The discrete logarithm problem in finite fields $GF\,(q)^*$ and the elliptic curve discrete logarithm are, in some sense, the same abstract problem in two different settings. As a result, the primitives and schemes of DL- and EC-based cryptography are closely analogous to each other. Table A.3 makes these analogies explicit.

Table A.3—Comparison between DL- and EC-based cryptography

	DL	EC
Setting	$GF\ (q)^*$	Curve E over $GF\ (q)$
Basic operation	Multiplication in $GF\ (q)$	Addition of points
Main operation	Exponentiation	Scalar multiplication
Base element	Generator g	Base point G
Base element order	Prime r	Prime r
Private key	s (integer modulo r)	s (integer modulo r)
Public key	w [element of $GF\ (q)$]	W (point on E)

A.9.5 Curve orders

The most difficult part of generating EC parameters is finding a base point of prime order. Generating such a point requires knowledge of the curve order $n = \#E(GF\ (q))$. Since r must divide n, one has the following problem: *given a field* $\mathbf{F} = GF\ (q)$, *find an elliptic curve defined over* $\mathbf{F}$ *whose order is divisible by a sufficiently large prime r*. (Note that "sufficiently large" is defined in terms of the desired security; see A.9.3 and D.4.2.) This subclause discusses this problem.

Basic facts

— If n is the order of an elliptic curve over $GF\ (q)$, then the Hasse bound is

$$q - 2\sqrt{q} + 1 \le n \le q + 2\sqrt{q} + 1$$

Thus, the order of an elliptic curve over $GF\ (q)$ is approximately q.

— If q is a prime p, let n be the order of the curve $y^2 = x^3 + ax + b$, where a and b are both nonzero. Then if $\lambda \neq 0$, the order of the curve $y^2 = x^3 + a\lambda^2 x + b\lambda^3$ is n if λ is a square modulo p, and $2p + 2 - n$ otherwise. (This fact allows one to replace a given curve by one with the same order and satisfying some extra condition, such as $a = p - 3$, which will be used in A.10.4.) In the case $b = 0$, there are four possible orders; in the case $a = 0$, there are six. The formulas for these orders can be found in step 6 of A.14.2.3.

— If $q = 2^m$, let n be the order of the curve $y^2 + xy = x^3 + ax^2 + b$, where a and b are both nonzero. Then if $\lambda \neq 0$, the order of the curve $y^2 + xy = x^3 + (a + \lambda)\ x^2 + b$ is n if λ has trace 0, and $2^{m+1} + 2 - n$ otherwise (see A.4.5). (This fact allows one to replace a given curve by one with the same order and satisfying some extra condition, such as $a = 0$, which will be used in A.10.7.)

— If $q = 2^m$, then the curves $y^2 + xy = x^3 + ax^2 + b$ and $y^2 + xy = x^3 + a^2\ x^2 + b^2$ have the same order.

Near primality

Given a trial division bound $l_{\max}$, the positive integer k is called *smooth* if every prime divisor of k is at most $l_{\max}$. Given large positive integers $r_{\min}$ and $r_{\max}$, u is called *nearly prime* if $u = kr$ for some prime r in the interval $r_{\min} \le r \le r_{\max}$ and some smooth integer k. (The requirement that k be smooth is omitted in most definitions of near primality. It is included here to guarantee that there exists an efficient algorithm to check for near primality.) In the case in which a prime order curve is desired, the bound $l_{\max}$ is set to 1.

NOTE—Since all elliptic curves over $GF\ (q)$ have order at most $u_{\max} = q + 2\ \sqrt{q}\ + 1$, then $r_{\max}$ should be no greater than $u_{\max}$. [If no maximum is desired (e.g., as in ANSI X9.62-1998 [B11]), then one takes $r_{\max} \leftarrow u_{\max}$.] Moreover, if $r_{\min}$ is close to $u_{\max}$, then there will be a small number of possible curves to choose from, so that finding a suitable one will be more difficult. If a prime-order curve is desired, a convenient choice is $r_{\min} = q + \sqrt{q}$.

Finding curves of appropriate order

In order to perform EC-based cryptography, it is necessary to be able to find an elliptic curve. The task is as follows: given a field size q and lower and upper bounds $r_{\min}$ and $r_{\max}$ for base point order, find an elliptic curve E over GF (q) and a prime r in the interval $r_{\min} \le r \le r_{\max}$, which is the order of a point on E.

Since large numbers are difficult to factor in general, a trial division bound $l_{\max}$ is chosen, and the search is restricted to nearly prime curve orders (see A.15.5). There are four approaches to selecting such a curve, as follows:

— Select a curve at random, compute its order directly, and repeat the process until an appropriate order is found.

— Select curve coefficients with particular desired properties, compute the curve order directly, and repeat the process until an appropriate order is found.

— If $q = 2^m$, where m is divisible by a "small" integer d, then select a curve defined over GF (2^d) and compute its order [over GF (2^m)] via A.11.6. Repeat, if possible, until an appropriate order is found.

— Search for an appropriate order, and construct a curve of that order.

The first approach is described in A.12.4 and A.12.6 (except for the point-counting algorithm, which is omitted). The second approach is not described, because its details depend on the particular desired properties. The third approach is given in A.11.4 through A.11.6. The fourth approach is implemented using the *complex multiplication* (CM) method in A.14.

A.9.6 Representation of points

This subclause discusses the issues involved in choosing representations for points on elliptic curves, for purposes of internal computation and for external communication.

Affine coordinates

A finite point on E is specified by two elements x, y in GF (q) satisfying the defining equation for E. These are called the *affine coordinates* for the point. The point at infinity O has no affine coordinates. For purposes of internal computation, it is most convenient to represent O by a pair of coordinates (x, y) not on E. For $q = 2^m$, the simplest choice is O = (0,0). For $q = p$, one chooses O = (0, 0) unless $b = 0$, in which case O = (0, 1).

Coordinate compression

The affine coordinates of a point require $2m$ bits to store and transmit, where q is either 2^m or an m-bit prime. This is far more than is needed, however. For purposes of external communication, therefore, it can be advantageous to *compress* one or both of the coordinates.

The y coordinate can always be compressed. The compressed y coordinate, denoted $\tilde{y}$, is a single bit, defined as follows:

— If q is an odd prime, then $\tilde{y} := y \bmod 2$, where y is interpreted as a positive integer less than q. Put another way, $\tilde{y}$ is the rightmost bit of y.

— If q is a power of 2, then $\tilde{y}$ is the rightmost bit of the field element $y\,x^{-1}$ (except when $x = 0$, in which case $\tilde{y} := 0$).

Compression of x coordinates takes place only when $q = 2^m$. Moreover, the x coordinate of a point can be compressed if, and only if, the point is twice another point. In particular, every point of prime order can be compressed. Therefore, x coordinate compression can be used in all cryptographic algorithms specified in this standard that are based on an elliptic curve over a binary field.

The compressed x coordinate, denoted $\tilde{x}$, is a bit string one bit shorter than the bit string representing x.

The string $\tilde{x}$ is derived from x by dropping a certain bit from the bit string representing x. If **F** is given by a normal basis, or if m is odd, then the rightmost bit is dropped. If **F** is given by a polynomial basis

$$\{t^{m-1}, \ldots, t, 1\}$$

and m is even, then one drops the rightmost bit corresponding to a basis element of trace 1. In other words, one drops the bit in the location of the rightmost 1 in the vector τ defined in A.4.5.

Examples:

a) If $m = 5$ and $x = (11001)$, then $\tilde{x} = (1100)$.

b) If **F** is given by the field polynomial $t^6 + t^5 + 1$, then

$$\tau = (111110)$$

so that the compressed form of $x = (a_5\ a_4\ a_3\ a_2\ a_1\ a_0)$ is $\tilde{x} = (a_5\ a_4\ a_3\ a_2\ a_0)$.

The choice of dropped bit depends only on the representation of the field, not on the choice of point.

NOTES

1—Algorithms for decompressing coordinates are given in A.12.8 through A.12.10.

2—There are many other possible ways to compress coordinates; the methods given here are the ones that have appeared in the literature (see Menezes [B110] and Seroussi [B135]).

3—It is a common usage (adopted, in particular, within this standard) to say that the point P is compressed if its y coordinate is compressed but its x coordinate is not (see E.2.3.1).

Projective coordinates

If division within $GF(q)$ is relatively expensive, then it may pay to keep track of numerators and denominators separately. In this way, one can replace division by α with multiplication of the denominator by α. This is accomplished by the projective coordinates X, Y, and Z given by

$$x = \frac{X}{Z^2}, y = \frac{Y}{Z^3}$$

The projective coordinates of a point are not unique because

$$(X, Y, Z) = (\lambda^2 X, \lambda^3 Y, \lambda Z)$$

for every nonzero $\lambda \in GF(q)$.

The projective coordinates of the point at infinity are $(\lambda^2, \lambda^3, 0)$, where $\lambda \neq 0$.

Other kinds of projective coordinates exist, but the ones given here provide the fastest arithmetic on elliptic curves (see Chudnovsky and Chudnovsky [B38]).

The formulas above provide the method for converting a finite point from projective coordinates to affine. To convert from affine to projective, one proceeds as follows:

$$X \leftarrow x, Y \leftarrow y, Z \leftarrow 1$$

Projective coordinates are well suited for internal computation, but not for external communication because they require so many bits. They are more common over *GF* (*p*), because division tends to be more expensive there.

A.10 Elliptic curves: algorithms

A.10.1 Full addition and subtraction (prime case)

The following algorithm implements a full addition (on a curve modulo *p*) in terms of affine coordinates:

Input: A prime $p > 3$; coefficients a, b for an elliptic curve E: $y^2 = x^3 + ax + b$ modulo p; points $P_0 = (x_0, y_0)$ and $P_1 = (x_1, y_1)$ on E

Output: The point $P_2 := P_0 + P_1$

```
1. If $P_0 = O$, then output $P_2 \leftarrow P_1$ and stop.
2. If $P_1 = O$, then output $P_2 \leftarrow P_0$ and stop.
3. If $x_0 \neq x_1$, then
   3.1 Set $\lambda \leftarrow (y_0 - y_1)/(x_0 - x_1) \bmod p$.
   3.2 Go to step 7.
4. If $y_0 \neq y_1$, then output $P_2 \leftarrow O$ and stop.
5. If $y_1 = 0$, then output $P_2 \leftarrow O$ and stop.
6. Set $\lambda \leftarrow (3x_1^2 + a)/(2y_1) \bmod p$.
7. Set $x_2 \leftarrow \lambda^2 - x_0 - x_1 \bmod p$.
8. Set $y_2 \leftarrow (x_1 - x_2)\,\lambda - y_1 \bmod p$.
9. Output $P_2 \leftarrow (x_2, y_2)$.
```

The above algorithm requires three or four modular multiplications and a modular inversion.

To subtract the point $P = (x, y)$, add the point $-P = (x, -y)$.

A.10.2 Full addition and subtraction (binary case)

The following algorithm implements a full addition [on a curve over *GF* (2^m)] in terms of affine coordinates:

Input: A field *GF* (2^m); coefficients a, b for an elliptic curve E: $\mathrm{y}^2 + xy = x^3 + ax^2 + b$ over *GF* (2^m); points $P_0 = (x_0, y_0)$ and $P_1 = (x_1, y_1)$ on E

Output: The point $P_2 := P_0 + P_1$

```
1.  If $P_0 = O$, then output $P_2 \leftarrow P_1$ and stop.
2.  If $P_1 = O$, then output $P_2 \leftarrow P_0$ and stop.
3.  If $x_0 \neq x_1$ then
    3.1  Set $\lambda \leftarrow (y_0 + y_1)/(x_0 + x_1)$.
    3.2  Set $x_2 \leftarrow a + \lambda^2 + \lambda + x_0 + x_1$.
    3.3  Go to step 7.
4.  If $y_0 \neq y_1$, then output $P_2 \leftarrow O$ and stop.
5.  If $x_1 = 0$, then output $P_2 \leftarrow O$ and stop.
6.  Set
    6.1  $\lambda \leftarrow x_1 + y_1/x_1$.
    6.2  $x_2 \leftarrow a + \lambda^2 + \lambda$.
7.  $y_2 \leftarrow (x_1 + x_2)\, \lambda + x_2 + y_1$.
8.  $P_2 \leftarrow (x_2, y_2)$.
```

The above algorithm requires two general multiplications, a squaring, and a multiplicative inversion.

To subtract the point $P = (x, y)$, add the point $-P = (x, x + y)$.

Further speedup

The two steps

```
    6.1  $\lambda \leftarrow x_1 + y_1/x_1$
    6.2  $x_2 \leftarrow a + \lambda^2 + \lambda$
```

require a multiplication and an inversion. If $GF\,(2^m)$ is represented by a normal basis, this can be improved by replacing the two lines with the following steps:

```
    6.1  $w \leftarrow x_1^2$.
    6.2  $x_2 \leftarrow w + b/w$.
    6.3  Solve the equation $\mu^2 + \mu = x_2 + a$ for $\mu$ via A.4.7 (normal basis case).
    6.4  $z \leftarrow w + y_{1.}$
    6.5  If $z = \mu x_1$, then $\lambda \leftarrow \mu$; else $\lambda \leftarrow \mu + 1$.
```

NOTE—The determination of whether $z = \mu x_1$ can be accomplished by computing one bit of the product μx_1 (any bit for which the corresponding bit of x_1 is 1) and comparing it to the corresponding bit of z.

This routine is more efficient than the ordinary method, because it replaces the general multiplication by a multiplication by the constant b (see Schroeppel et al. [B133]).

A.10.3 Elliptic scalar multiplication

Scalar multiplication can be performed efficiently by the *addition-subtraction method* outlined below.

Input: An integer n and an elliptic curve point P

Output: The elliptic curve point nP

```
1. If $n = 0$, then output O and stop.
2. If $n < 0$, then set $Q \leftarrow (-P)$ and $k \leftarrow (-n)$; else set $Q \leftarrow P$ and $k \leftarrow n$.
3. Let $h_l\, h_{l-1} \ldots h_1\, h_0$ be the binary representation of $3k$, where the most significant bit $h_l$ is 1.
4. Let $k_l\, k_{l-1} \ldots k_1\, k_0$ be the binary representation of $k$.
5. Set $S \leftarrow Q$.
6. For $i$ from $l - 1$ downto 1 do
       Set $S \leftarrow 2S$.
       If $h_i = 1$ and $k_i = 0$, then compute $S \leftarrow S + Q$ via A.10.1 or A.10.2.
       If $h_i = 0$ and $k_i = 1$, then compute $S \leftarrow S - Q$ via A.10.1 or A.10.2.
7. Output $S$.
```

There are several modifications that improve the performance of this algorithm. These methods are summarized in Gordon [B65].

A.10.4 Projective elliptic doubling (prime case)

The projective form of the doubling formula on the curve $y^2 = x^3 + ax + b$ modulo p is

$$2\,(X_1, Y_1, Z_1) = (X_2, Y_2, Z_2)$$

where

$$M = 3X_1^2 + aZ_1^4$$

$$Z_2 = 2Y_1Z_1$$

$$S = 4X_1Y_1^2$$

$$X_2 = M^2 - 2S$$

$$T = 8Y_1^4$$

$$Y_2 = M\,(S - X_2) - T.$$

The algorithm `Double` given below performs these calculations.

Input: A modulus p; the coefficients a and b defining a curve E modulo p; projective coordinates (X_1, Y_1, Z_1) for a point P_1 on E

Output: Projective coordinates (X_2, Y_2, Z_2) for the point $P_2 = 2P_1$

```
1.  $T_1 \leftarrow X_1$.
2.  $T_2 \leftarrow Y_1$.
3.  $T_3 \leftarrow Z_1$.
4.  If $T_2 = 0$ or $T_3 = 0$, then output (1, 1, 0) and stop.
5.  If $a = p - 3$ then
        $T_4 \leftarrow T_3^2$
        $T_5 \leftarrow T_1 - T_4$
        $T_4 \leftarrow T_1 + T_4$
        $T_5 \leftarrow T_4 \times T_5$
        $T_4 \leftarrow 3 \times T_5$ (this step computes $M$)
    else
        $T_4 \leftarrow a$
        $T_5 \leftarrow T_3^2$
        $T_5 \leftarrow T_5^2$
        $T_5 \leftarrow T_4 \times T_5$
        $T_4 \leftarrow T_1^2$
        $T_4 \leftarrow 3 \times T_4$
        $T_4 \leftarrow T_4 + T_5$ (this step computes $M$).
6.  $T_3 \leftarrow T_2 \times T_3$.
7.  $T_3 \leftarrow 2 \times T_3$ (this step computes $Z_2$).
8.  $T_2 \leftarrow T_2^2$.
9.  $T_5 \leftarrow T_1 \times T_2$.
10. $T_5 \leftarrow 4 \times T_5$ (this step computes $S$).
11. $T_1 \leftarrow T_4^2$.
12. $T_1 \leftarrow T_1 - 2 \times T_5$ (this step computes $X_2$).
13. $T_2 \leftarrow T_2^2$.
14. $T_2 \leftarrow 8 \times T_2$ (this step computes $T$).
15. $T_5 \leftarrow T_5 - T_1$.
16. $T_5 \leftarrow T_4 \times T_5$.
17. $T_2 \leftarrow T_5 - T_2$ (this step computes $Y_2$).
18. $X_2 \leftarrow T_1$.
19. $Y_2 \leftarrow T_2$.
20. $Z_2 \leftarrow T_3$.
```

This algorithm requires ten field multiplications and five temporary variables. If a is small enough that multiplication by a can be done by repeated addition, only 9 field multiplications are required. If $a = p - 3$, then only eight field multiplications are required (see Chudnovsky and Chudnovsky [B38]). The proportion of elliptic curves modulo p that can be rescaled so that $a = p - 3$ is about 1/4 if $p \equiv 1 \pmod 4$ and about 1/2 if $p \equiv 3 \pmod 4$ (see A.9.5).

A.10.5 Projective elliptic addition (prime case)

The projective form of the adding formula on the curve $y^2 = x^3 + ax + b$ modulo p is

$$(X_0, Y_0, Z_0) + (X_1, Y_1, Z_1) = (X_2, Y_2, Z_2)$$

where

$$U_0 = X_0Z_1^2$$

$$S_0 = Y_0Z_1^3$$

$$U_1 = X_1Z_0^2$$

$$S_1 = Y_1Z_0^3$$

$$W = U_0 - U_1$$

$$R = S_0 - S_1$$

$$T = U_0 + U_1$$

$$M = S_0 + S_1$$

$$Z_2 = Z_0Z_1W$$

$$X_2 = R^2 - TW^2$$

$$V = TW^2 - 2X_2$$

$$2Y_2 = VR - MW^{3.}$$

The algorithm `Add` given below performs these calculations.

Input: A modulus p; the coefficients a and b defining a curve E modulo p; projective coordinates (X_0, Y_0, Z_0) and (X_1, Y_1, Z_1) for points P_0 and P_1 on E, where Z_0 and Z_1 are nonzero

Output: Projective coordinates (X_2, Y_2, Z_2) for the point $P_2 = P_0 + P_1$, unless $P_0 = P_1$. In this case, the triplet (0, 0, 0) is returned. [The triplet (0, 0, 0) is not a valid projective point on the curve, but rather a marker indicating that routine `Double` should be used.]

```
1. $T_1 \leftarrow X_0$ [this step computes $U_0$ (if $Z_1 = 1$)].
2. $T_2 \leftarrow Y_0$ [this step computes $S_0$ (if $Z_1 = 1$)].
3. $T_3 \leftarrow Z_0$.
4. $T_4 \leftarrow X_1$.
5. $T_5 \leftarrow Y_1$.
6. If $Z_1 \neq 1$, then
        $T_6 \leftarrow Z_1$
```

```
            $T_7 \leftarrow T_6^2$
            $T_1 \leftarrow T_1 \times T_7$ [this step computes $U_0$ (if $Z_1 \neq 1$)]
            $T_7 \leftarrow T_6 \times T_7$
            $T_2 \leftarrow T_2 \times T_7$ [this step computes $S_0$ (if $Z_1 \neq 1$)].
 7.  $T_7 \leftarrow T_3^2$.
 8.  $T_4 \leftarrow T_4 \times T_7$ (this step computes $U_1$).
 9.  $T_7 \leftarrow T_3 \times T_7$.
10.  $T_5 \leftarrow T_5 \times T_7$ (this step computes $S_1$).
11.  $T_4 \leftarrow T_1 - T_4$ (this step computes $W$).
12.  $T_5 \leftarrow T_2 - T_5$ (this step computes $R$).
13.  If $T_4 = 0$, then
            If $T_5 = 0$, output (0,0,0) and stop;
            else output (1, 1, 0) and stop.
14.  $T_1 \leftarrow 2 \times T_1 - T_4$ (this step computes $T$).
15.  $T_2 \leftarrow 2 \times T_2 - T_5$ (this step computes $M$).
16.  If $Z_1 \neq 1$, then
            $T_3 \leftarrow T_3 \times T_6$.
17.  $T_3 \leftarrow T_3 \times T_4$ (this step computes $Z_2$).
18.  $T_7 \leftarrow T_4^2$.
19.  $T_4 \leftarrow T_4 \times T_7$.
20.  $T_7 \leftarrow T_1 \times T_7$.
21.  $T_1 \leftarrow T_5^2$.
22.  $T_1 \leftarrow T_1 - T_7$ (this step computes $X_2$).
23.  $T_7 \leftarrow T_7 - 2 \times T_1$ (this step computes $V$).
24.  $T_5 \leftarrow T_5 \times T_7$.
25.  $T_4 \leftarrow T_2 \times T_4$.
26.  $T_2 \leftarrow T_5 - T_4$.
27.  $T_2 \leftarrow T_2/2$ (this step computes $Y_2$).
28.  $X_2 \leftarrow T_1$.
29.  $Y_2 \leftarrow T_2$.
30.  $Z_2 \leftarrow T_3$.
```

NOTE—The modular division by two in step 27 can be carried out in the same way as in A.2.4.

This algorithm requires sixteen field multiplications and seven temporary variables. In the case $Z_1 = 1$, only eleven field multiplications and six temporary variables are required. (This is the case of interest for elliptic scalar multiplication.)

A.10.6 Projective elliptic doubling (binary case)

The projective form of the doubling formula on the curve $y^2 + xy = x^3 + ax^2 + b$ over $GF(2^m)$ does not use the coefficient b, but rather the field element

$$c := b^{2^{m-2}}$$

computed from b by $m - 2$ squarings (thus, $b = c^4$). The formula is

$$2\,(X_1, \mathrm{Y}_1, Z_1) = (X_2, Y_2, Z_2)$$

where

$$Z_2 = X_1 Z_1^2$$

$$X_2 = (X_1 + cZ_1^2)^4$$

$$U = Z_2 + X_1^2 + Y_1 Z_1$$

$$Y_2 = X_1^4 Z_2 + U X_2 .$$

The algorithm `Double` given below performs these calculations.

Input: A field of 2^m elements; the field elements a and c specifying a curve E over $GF(2^m)$; projective coordinates (X_1, Y_1, Z_1) for a point P_1 on E

Output: Projective coordinates (X_2, Y_2, Z_2) for the point $P_2 = 2P_1$

```
1.  $T_1 \leftarrow X_1$.
2.  $T_2 \leftarrow Y_1$.
3.  $T_3 \leftarrow Z_1$.
4.  $T_4 \leftarrow c$.
5.  If $T_1 = 0$ or $T_3 = 0$, then output (1, 1, 0) and stop.
6.  $T_2 \leftarrow T_2 \times T_3$.
7.  $T_3 \leftarrow T_3^2$
8.  $T_4 \leftarrow T_3 \times T_4$.
9.  $T_3 \leftarrow T_1 \times T_3$ (this step computes $Z_2$).
10. $T_2 \leftarrow T_2 + T_3$.
11. $T_4 \leftarrow T_1 + T_4$.
12. $T_4 \leftarrow T_4^2$.
13. $T_4 \leftarrow T_4^2$ (this step computes $X_2$).
14. $T_1 \leftarrow T_1^2$.
15. $T_2 \leftarrow T_1 + T_2$ (this step computes $U$).
16. $T_2 \leftarrow T_2 \times T_4$.
```

```
17. $T_1 \leftarrow T_1^2$.
18. $T_1 \leftarrow T_1 \times T_3$.
19. $T_2 \leftarrow T_1 + T_2$ (this step computes $Y_2$).
20. $T_1 \leftarrow T_4$.
21. $X_2 \leftarrow T_1$.
22. $Y_2 \leftarrow T_2$.
23. $Z_2 \leftarrow T_3$.
```

This algorithm requires five field squarings, five general field multiplications, and four temporary variables.

A.10.7 Projective elliptic addition (binary case)

The projective form of the adding formula on the curve $y^2 + xy = x^3 + ax^2 + b$ over $GF(2^m)$ is

$$(X_0, Y_0, Z_0) + (X_1, Y_1, Z_1) = (X_2, Y_2, Z_2)$$

where

$$U_0 = X_0 Z_1^2$$
$$S_0 = Y_0 Z_1^3$$
$$U_1 = X_1 Z_0^2$$
$$W = U_0 + U_1$$
$$S_1 = Y_1 Z_0^3$$
$$R = S_0 + S_1$$
$$L = Z_0\, W$$
$$V = RX_1 + LY_1$$
$$Z_2 = LZ_1$$
$$T = R + Z_2$$
$$X_2 = aZ_2^2 + TR + W^3$$
$$Y_2 = TX_2 + VL_2.$$

The algorithm `Add` given below performs these calculations.

Input: A field of 2^m elements; the field elements a and b defining a curve E over $GF(2^m)$; projective coordinates (X_0, Y_0, Z_0) and (X_1, Y_1, Z_1) for points P_0 and P_1 on E, where Z_0 and Z_1 are nonzero

Output: Projective coordinates (X_2, Y_2, Z_2) for the point $P_2 = P_0 + P_1$, unless $P_0 = P_1$. In this case, the triplet (0, 0, 0) is returned. [The triplet (0, 0, 0) is not a valid projective point on the curve, but rather a marker indicating that routine `Double` should be used.]

```
1. $T_1 \leftarrow X_0$ [this step computes $U_0$ (if $Z_1 = 1$)].
2. $T_2 \leftarrow Y_0$ [this step computes $S_0$ (if $Z_1 = 1$)].
3. $T_3 \leftarrow Z_0$.
```

```
4.  $T_4 \leftarrow X_1$.
5.  $T_5 \leftarrow Y_1$
6.  If $a \neq 0$, then
      $T_9 \leftarrow a$.
7.  If $Z_1 \neq 1$, then
      $T_6 \leftarrow Z_1$
      $T_7 \leftarrow T_6^2$
      $T_1 \leftarrow T_1 \times T_7$ [this step computes $U_0$ (if $Z_1 \neq 1$)]
      $T_7 \leftarrow T_6 \times T_7$
      $T_2 \leftarrow T_2 \times T_7$ [this step computes $S_0$ (if $Z_1 \neq 1$)].
8.  $T_7 \leftarrow T_3^2$.
9.  $T_8 \leftarrow T_4 \times T_7$ (this step computes $U_1$).
10. $T_1 \leftarrow T_1 + T_8$ (this step computes $W$).
11. $T_7 \leftarrow T_3 \times T_7$.
12. $T_8 \leftarrow T_5 \times T_7$ (this step computes $S_1$).
13. $T_2 \leftarrow T_2 + T_8$ (this step computes $R$).
14. If $T_1 = 0$, then
          If $T_2 = 0$, output (0, 0, 0) and stop;
          else output (1, 1, 0) and stop.
15. $T_4 \leftarrow T_2 \times T_4$.
16. $T_3 \leftarrow T_1 \times T_3$ [this step computes $L$ (and $Z_2$ if $Z_1 = 1$)].
17. $T_5 \leftarrow T_3 \times T_5$.
18. $T_4 \leftarrow T_4 + T_5$ (this step computes $V$).
19. $T_5 \leftarrow T_3^2$.
20. $T_7 \leftarrow T_4 \times T_5$.
21. If $Z_1 \neq 1$, then
          $T_3 \leftarrow T_3 \times T_6$ [this step computes $Z_2$ (if $Z_1 \neq 1$)].
22. $T_4 \leftarrow T_2 + T_3$ (this step computes $T$).
23. $T_2 \leftarrow T_2 \times T_4$.
24. $T_5 \leftarrow T_1^2$.
25. $T_1 \leftarrow T_1 \times T_5$.
26. If $a \neq 0$, then
          $T_8 \leftarrow T_3^2$
          $T_9 \leftarrow T_8 \times T_9$
          $T_1 \leftarrow T_1 + T_9$.
27. $T_1 \leftarrow T_1 + T_2$ (this step computes $X_2$).
```

```
28. $T_4 \leftarrow T_1 \times T_4$.
29. $T_2 \leftarrow T_4 + T_7$ (this step computes $Y_2$).
30. $X_2 \leftarrow T_1$.
31. $Y_2 \leftarrow T_2$.
32. $Z_2 \leftarrow T_3$.
```

This algorithm requires five field squarings, fifteen general field multiplications and nine temporary variables. If $a = 0$, then only four field squarings, fourteen general field multiplications, and eight temporary variables are required. [About half of the elliptic curves over $GF(2^m)$ can be rescaled so that $a = 0$. They are precisely the curves with order divisible by four (see A.9.5)].

In the case $Z_1 = 1$, only four field squarings, eleven general field multiplications, and eight temporary variables are required. If $a = 0$, then only three field squarings, ten general field multiplications, and seven temporary variables are required. (These are the cases of interest for elliptic scalar multiplication.)

A.10.8 Projective full addition and subtraction

The following algorithm FullAdd implements a full addition in terms of projective coordinates.

Input: A field of q elements; the field elements a and b defining a curve E over $GF(q)$; projective coordinates (X_0, Y_0, Z_0) and (X_1, Y_1, Z_1) for points P_0 and P_1 on E

Output: Projective coordinates (X_2, Y_2, Z_2) for the point $P_2 = P_0 + P_1$

```
1. If $Z_0 = 0$, then output $(X_2, Y_2, Z_2) \leftarrow (X_1, Y_1, Z_1)$ and stop.
2. If $Z_1 = 0$, then output $(X_2, Y_2, Z_2) \leftarrow (X_0, Y_0, Z_0)$ and stop.
3. Set $(X_2, Y_2, Z_2) \leftarrow$ Add[$(X_0, Y_0, Z_0)$, $(X_1, Y_1, Z_1)$].
4. If $(X_2, Y_2, Z_2) = (0, 0, 0)$, then set $(X_2, Y_2, Z_2) \leftarrow$ Double[$(X_1, Y_1, Z_1)$].
5. Output $(X_2, Y_2, Z_2)$.
```

An *elliptic subtraction* is implemented as follows:

Subtract[(X_0, Y_0, Z_0), (X_1, Y_1, Z_1)] = FullAdd[(X_0, Y_0, Z_0), (X_1, U, Z_1)]

where

$$U = \begin{cases} -Y_1 \quad \bmod p & \text{if } q = p \\ X_1 Z_1 + Y_1 & \text{if } q = 2^m \end{cases} .$$

A.10.9 Projective elliptic scalar multiplication

Input: An integer n and an elliptic curve point $P = (X, Y, Z)$

Output: The elliptic curve point $nP = (X^*, Y^*, Z^*)$

```
1. If $n = 0$ or $Z = 0$, then output (1, 1, 0) and stop.
2. Set
```

```
    2.1  $X^* \leftarrow X$
    2.2  $Z^* \leftarrow Z$
    2.3  $Z_1 \leftarrow 1$.
3.  If $n < 0$, then go to step 6.
4.  Set
    4.1  $k \leftarrow n$
    4.2  $Y^* \leftarrow Y$.
5.  Go to step 8.
6.  Set $k \leftarrow (-n)$.
7.  If $q = p$, then set $Y^* \leftarrow -Y \pmod{p}$; else set $Y^* \leftarrow XZ + Y$.
8.  If $Z^* = 1$, then set $X_1 \leftarrow X^*$, $Y_1 \leftarrow Y^*$; else set $X_1 \leftarrow X^*/(Z^*)^2$, $Y_1 \leftarrow Y^* / (Z^*)^3$.
9.  Let $h_l\, h_{l-1} \ldots h_1\, h_0$ be the binary representation of $3k$, where the most significant bit $h_l$ is 1.
10. Let $k_l\, k_{l-1} \ldots k_1\, k_0$ be the binary representation of $k$.
11. For $i$ from $l - 1$ downto 1 do
    11.1 Set $(X^*, Y^*, Z^*) \leftarrow$ Double[$(X^*, Y^*, Z^*)$].
    11.2 If $h_i = 1$ and $k_i = 0$, then set $(X^*, Y^*, Z^*) \leftarrow$ FullAdd[$(X^*, Y^*, Z^*)$, $(X_1, Y_1, Z_1)$].
    11.3 If $h_i = 0$ and $k_i = 1$, then set $(X^*, Y^*, Z^*) \leftarrow$ Subtract[$(X^*, Y^*, Z^*)$, $(X_1, Y_1, Z_1)$].
12. Output $(X^*, Y^*, Z^*)$.
```

There are several modifications that improve the performance of this algorithm. These methods are summarized in Gordon [B65].

A.11 Functions for elliptic curve parameter and key generation

A.11.1 Finding a random point on an elliptic curve (prime case)

The following algorithm provides an efficient method for finding a random point (other than O) on a given elliptic curve over the finite field *GF* (p):

Input: A prime $p > 3$ and the parameters a, b of an elliptic curve E modulo p

Output: A randomly generated point (other than O) on E

```
1.  Choose random $x$ with $0 \le x < p$.
2.  Set $\alpha \leftarrow x^3 + ax + b \bmod p$.
3.  If $\alpha = 0$, then output $(x, 0)$ and stop.
4.  Apply the appropriate technique from A.2.5 to find a square root modulo $p$ of $\alpha$ or determine
    that none exists.
5.  If the result of step 4 indicates that no square roots exist, then go to step 1; otherwise, the output
    of step 4 is an integer $\beta$ with $0 < \beta < p$ such that
        $\beta^2 \equiv a \pmod{p}$.
6.  Generate a random bit $\mu$ and set $y \leftarrow (-1)^{\mu}\, \beta$.
7.  Output $(x, y)$.
```

A.11.2 Finding a random point on an elliptic curve (binary case)

The following algorithm provides an efficient method for finding a random point (other than O) on a given elliptic curve over the finite field $GF\ (2^m)$:

Input: A field $GF\ (2^m)$ and the parameters a, b of an elliptic curve E over $GF\ (2^m)$

Output: A randomly generated point (other than O) on E

1. Choose random x in $GF\ (2^m)$.
2. If $x = 0$, then output $(0,\ b^{2^{m-1}})$ and stop.
3. Set $\alpha \leftarrow x^3 + ax^2 + b$.
4. If $\alpha = 0$, then output $(x, 0)$ and stop.
5. Set $\beta \leftarrow x^{-2}\ \alpha$.
6. Apply the appropriate technique from A.4.7 to find an element z for which $z^2 + z = \beta$ or determine that none exists.
7. If the result of step 6 indicates that no solutions exist, then go to step 1; otherwise, the output of step 6 is a solution z.
8. Generate a random bit μ and set $y \leftarrow (z + \mu)\ x$.
9. Output (x, y).

A.11.3 Finding a point of large prime order

If the order $\#E(GF\ (q)) = u$ of an elliptic curve E is nearly prime, the following algorithm efficiently produces a random point on E whose order is the large prime factor r of $u = kr$. (See A.9.5 for the definition of *nearly prime*.)

Input: A prime r; a positive integer k not divisible by r; an elliptic curve E over the field $GF\ (q)$

Output: If $\#E(GF\ (q))= kr$, a point G on E of order r. If not, the message "wrong order."

1. Generate a random point P (not O) on E via A.11.1 or A.11.2.
2. Set $G \leftarrow kP$.
3. If $G = O$, then go to step 1.
4. Set $Q \leftarrow rG$.
5. If $Q \neq O$, then output "wrong order" and stop.
6. Output G.

A.11.4 Curve orders over small binary fields

If d is "small" (i.e., it is feasible to perform 2^d arithmetic operations), then the order of the curve $y^2 + xy = x^3 + ax^2 + b$ over $GF\ (2^d)$ can be calculated directly as follows: Let

$$\mu = (-1)^{\mathrm{Tr}(a)}$$

For each nonzero $x \in GF\ (2^d)$, let

$$\lambda\ (x) = \mathrm{Tr}\ (x + b/x^2)$$

Then

$$\#E(GF(2^d)) = 2^d + 1 + \mu \sum_{x \neq 0} (-1)^{\lambda(x)}$$

A.11.5 Curve orders over extension fields

Given the order of an elliptic curve E over a finite field $GF\ (2^d)$, the following algorithm computes the order of E over the extension field $GF\ (2^{de})$:

Input: Positive integers d and e; an elliptic curve E defined over $GF\ (2^d)$; the order w of E over $GF\ (2^d)$

Output: The order u of E over $GF\ (2^{de})$

```
1.  Set $P \leftarrow 2^d + 1 - w$ and $Q \leftarrow 2^d$.
2.  Compute via A.2.4 the Lucas sequence element $V_e$.
3.  Compute $u := 2^{de} + 1 - V_e$.
4.  Output $u$.
```

A.11.6 Curve orders via subfields

The algorithms of A.11.4 and A.11.5 allow construction of elliptic curves with known orders over $GF\ (2^m)$, provided that m is divisible by an integer d that is small enough for A.11.4. The following algorithm finds such curves with nearly prime orders when such exist. (See A.9.5 for the definition of *nearly prime*.)

Input: A field $GF\ (2^m)$; a subfield $GF\ (2^d)$ for some (small) d dividing m; lower and upper bounds $r_{\min}$ and $r_{\max}$ for the base point order

Output: Elements a, $b \in GF\ (2^m)$ specifying an elliptic curve E, along with the nearly prime order $n = \#E(GF\ (2^m))$, if one exists; otherwise, the message "no such curve"

```
1.  Select elements $a_0$, $b_0 \in GF\ (2^d)$ such that $b_0$ has not already been selected. (If all the possible
    values for $b_0$'s have already been tried, then output the message "no such curve" and stop.) Let
    $E$ be the elliptic curve $y^2 + xy = x^3 + a_0 x^2 + b_0$.
2.  Compute the order $w = \#E(GF\ (2^d))$ via A.11.4.
3.  Compute the order $u = \#E(GF\ (2^m))$ via A.11.5.
4.  Test $u$ for near-primality via A.15.5.
5.  If $u$ is nearly prime, then set $\lambda \leftarrow 0$ and $n \leftarrow u$ and go to step 9.
6.  Set $u' = 2^{m+1} + 2 - u$.
7.  Test $u'$ for near-primality via A.15.5.
8.  If $u'$ is nearly prime, then set $\lambda \leftarrow 1$ and $n \leftarrow u'$, else go to step 1.
9.  Find the elements $a_1$, $b_1 \in GF\ (2^m)$ corresponding to $a_0$ and $b_0$ via A.5.7.
10. If $\lambda = 0$, then set $\tau \leftarrow 0$. If $\lambda = 1$ and $m$ is odd, then set $\tau \leftarrow 1$. Otherwise, find an element $\tau \in$
    $GF\ (2^m)$ of trace 1 by trial and error using A.4.5.
11. Set $a \leftarrow a_1 + \tau$ and $b \leftarrow b_1$.
12. Output $n$, $a$, $b$.
```

NOTE—It follows from the basic facts of A.9.5 that any a_0 can be chosen at any time in step 1.

A.12 Functions for elliptic curve parameter and key validation

A.12.1 The MOV condition

The *MOV condition* ensures that an elliptic curve is not vulnerable to the reduction attack of Menezes, Okamoto, and Vanstone [B111].

Before performing the algorithm, it is necessary to select an *MOV threshold*. This is a positive integer B such that taking discrete logarithms over $GF\ (q^B)$ is judged to be at least as difficult as taking elliptic discrete logarithms over $GF\ (q)$. It follows from the complexity discussions of D.4.1 and from Menezes [B110] that B should be large enough so that

$$T\ (mB) \geq m$$

where

$$2^m \leq q < 2^{m+1}$$

and

$$T(n) = \frac{8}{3}\left(3n(\log_2(n\ 1\text{n}\ 2))^2\right)^{1/3} \ -17.135872$$

(As before, $\ln x$ denotes the natural logarithm of x.) The constant in this formula follows from Dodson and Lenstra [B50] and Odlyzko [B121]. Table A.4 gives the smallest value of B satisfying the above condition, for each q whose bit length m is between 128 and 512.

Table A.4—MOV thresholds

m	*B*	*m*	*B*	*m*	*B*
128–142	6	281–297	14	407–420	22
143–165	7	298–313	15	421–434	23
166–186	8	314–330	16	435–448	24
187–206	9	331–346	17	449–462	25
207–226	10	347–361	18	463–475	26
227–244	11	362–376	19	476–488	27
245–262	12	377–391	20	489–501	28
263–280	13	392–406	21	502–512	29

Once an appropriate B has been selected, the following algorithm checks the MOV condition for the choice of field size q and base point order r by verifying that q^i is not congruent to 1 modulo r for any $i \leq B$.

Input: A MOV threshold B; a prime-power q; a prime r

Output: The message “True” if the MOV condition is satisfied for an elliptic curve over $GF\ (q)$ with a base point of order r; the message “False” otherwise

```
1.  Set $t \leftarrow 1$.
2.  For $i$ from 1 to $B$ do
    2.1  Set $t \leftarrow tq \bmod r$.
    2.2  If $t = 1$, then output "False" and stop.
3.  Output "True."
```

A.12.2 The Weil pairing

The *Weil pairing* is a function $< P, Q >$ of pairs P, Q of points on an elliptic curve E. It can be used to determine whether P and Q are multiples of each other. It will be used in A.12.3.

Let $l > 2$ be prime, and let P and Q be points on E with $lP = lQ = O$. The following procedure computes the Weil pairing:

— Given three points (x_0, y_0), (x_1, y_1), (u, v) on E, define the function $f((x_0, y_0), (x_1, y_1), (u, v))$ by

$$\begin{array}{ll} u - x_1 & \text{if } x_0 = x_1 \text{ and } y_0 = -y_1 \\ (3x_1^2 + a)(u - x_1) - 2y_1(v - y_1) & \text{if } x_0 = x_1 \text{ and } y_0 = y_1 \\ (x_0 - x_1)v - (y_0 - y_1)u - (x_0 y_1 - x_1 y_0) & \text{if } x_0 \neq x_1 \end{array}$$

if E is the curve $y^2 = x^3 + ax + b$ over $GF(p)$, and by

$$\begin{array}{ll} u + x_1 & \text{if } x_0 = x_1 \text{ and } y_0 = x_1 + y_1 \\ x_1^3 + (x_1^2 + y_1)u + x_1 v & \text{if } x_0 = x_1 \text{ and } y_0 = y_1 \\ (x_0 + x_1)v + (y_0 + y_1)u + (x_0 y_1 + x_1 y_0) & \text{if } x_0 \neq x_1 \end{array}$$

if E is the curve $y^2 + xy = x^3 + ax^2 + b$ over $GF(2^m)$.

— Given points A, B, C on E, let

$$g(A, B, C) := f(A, B, C) / f(A + B, -A - B, C)$$

— The *Weil function h* is computed via the following algorithm.

Input: A prime $l > 2$; a curve E; finite points D, C on E with $lD = lC = O$

Output: The Weil function $h(D, C)$

```
1.  Set $A \leftarrow D$, $B \leftarrow D$, $h \leftarrow 1$, $n \leftarrow l$.
2.  Set $n \leftarrow \lfloor n/2 \rfloor$.
3.  Set $h \leftarrow g(B, B, C)^n \times h$.
4.  Set $B \leftarrow 2B$.
5.  If $n$ is odd then
    5.1  If $n = 1$
             then $h \leftarrow g(A, -A, C) \times h$;
             else $h \leftarrow g(A, B, C) \times h$.
    5.2  Set $A \leftarrow A + B$.
```

6. If $n > 1$, then go to step 2.
7. Output h.

— Given points R and S on E, let

$$j(R, S) := h(R, S) / h(S, R).$$

— Given points P and Q on E with $lP = lQ = \mathrm{O}$, the Weil pairing $< P, Q >$ is computed as follows:

Choose random points T, U on E and let

$$V = P + T, \quad W = Q + U.$$

Then

$$< P, Q > = j(T, U)\, j(U, V)\, j(V, W)\, j(W, T).$$

If, in evaluating $< P, Q >$, one encounters $f((x_0, y_0), (x_1, y_1), (u, v)) = 0$, then the calculation fails. In this (unlikely) event, repeat the calculation with newly chosen T and U.

In the case $l = 2$, the Weil pairing is easily computed as follows: $< P, Q >$ equals 1 if $P = Q$, and –1 otherwise.

Define $< P, \mathrm{O} > =1$ for all points P.

A.12.3 Verification of cofactor

Let E be an elliptic curve over $GF(q)$ of order $u = kr$, where r is the prime order of the base point G. The integer k is called the *cofactor.* The DL and EC key agreement schemes provide an option for exponentiation or scalar multiplication of the shared secret value by the cofactor. In order to implement this option, it is necessary to know the cofactor k.

In the DL case, the cofactor is simply $k = (q - 1)/r$. In the EC case, a simple formula exists only if $r > 4\sqrt{q}$ (which is typically the case). In this case, k can be computed directly from the verified parameters q and r (see A.16.8) via the formula

$$k := \left\lfloor \frac{(\sqrt{q} + 1)^2}{r} \right\rfloor$$

If $r \leq 4\sqrt{q}$, then k cannot be computed from q and r alone. Therefore, the value of k must be sent along with the EC parameters, and should be verified by the recipient. This verification is accomplished by a rather complex algorithm, as described below.

Torsion points

If l is a prime, then an *l-torsion point* is a finite point T, such that $lT = \mathrm{O}$. The following algorithm (a subroutine of the cofactor verification procedure to be given below) generates an l-torsion point or reports that none exists. The algorithm is probabilistic, but the probability of error can be made as small as desired. (A parameter β is included for this purpose.)

Input: The EC parameters q, a, and b; the putative order $u = \#E(GF(q))$; a prime $l \neq r$; the largest positive integer v such that l^v divides u; and a positive integer β

Output: An l-torsion point T and a positive integer $\alpha \leq v$, such that $T = l^\alpha P$ for some point P, or the message "no torsion point found"

```
1.  Set $h \leftarrow kr/l^v$.
2.  Set $\mu \leftarrow 0$.
3.  Set $\mu \leftarrow \mu + 1$.
4.  If $\mu > \beta$, then output “no torsion point found” and stop.
5.  Generate a random finite point $R$ via A.11.1 or A.11.2.
6.  Compute $S := hR$.
7.  If $S = \mathsf{O}$ then go to step 3.
8.  Set $\alpha \leftarrow 1$.
9.  Set $\alpha \leftarrow \alpha + 1$.
10. It $\alpha > v$, then output “no torsion point found” and stop.
11. Set $T \leftarrow S$.
12. Set $S \leftarrow lS$.
13. If $S \neq \mathsf{O}$, then go to step 9.
14. Output $T$ and $\alpha$.
```

Cofactor verification procedure

Input: The EC parameters q, a, b, r, and k; an upper bound ε for the probability of error

Output: “Cofactor confirmed” or “cofactor rejected”

```
1.  Set $u \leftarrow kr$.
2.  If $u < (\sqrt{q} - 1)^2$ or $u > (\sqrt{q} + 1)^2$, then output “cofactor rejected” and stop.
3.  Factor
        $k := l_1^{v_1} \ldots l_s^{v_s}$
    where the $l_i$’s are distinct primes and each $v_i$ is positive.
4.  For $i$ from 1 to $s$ do
    4.1  Set $l \leftarrow l_i$, $v \leftarrow v_i$.
    4.2  Set
         $\beta := \left\lceil \frac{\log(s/\varepsilon)}{v \log l} \right\rceil$.
    4.3  Set $j \leftarrow 0$.
    4.4  If $q \equiv 1 \pmod{l}$, then set $e \leftarrow 0$, $f \leftarrow 0$, $U \leftarrow \mathsf{O}$, $V \leftarrow \mathsf{O}$.
    4.5  Set $j \leftarrow j + 1$.
    4.6  If $j > v\beta$, then output “cofactor rejected” and stop.
    4.7  Generate an $l$-torsion point $T$ and associated integer $\alpha$ via the subroutine.
    4.8  If the output of the subroutine is “no torsion point found,” then output “cofactor rejected”
         and stop.
    4.9  If $q \not\equiv 1 \pmod{l}$, then set $\rho \leftarrow \alpha$; else
```

```
        4.9.1   If $\alpha \leq e$, then go to step 4.10.
        4.9.2   Compute the Weil pairing $w := < T, V >$ via A.12.2.
        4.9.3   If $\alpha \geq f$, then go to step 4.9.6.
        4.9.4   If $w \neq 1$, then set $e \leftarrow \alpha$, $U \leftarrow T$.
        4.9.5   Go to step 4.9.8.
        4.9.6   If $w \neq 1$, then set $e \leftarrow f$, $U \leftarrow V$.
        4.9.7   Set $f \leftarrow \alpha$, $V \leftarrow T$.
        4.9.8   Set $\rho \leftarrow e + f$.
        4.9.9   If $\rho < v$, then go to step 4.5.
    5.  Output "cofactor confirmed."
```

A.12.4 Constructing verifiably pseudo-random elliptic curves (prime case)

It is common to use an elliptic curve selected at random from the curves of appropriate order (see A.9.5). The following algorithm produces a set of elliptic curve parameters for such a curve over a field $GF(p)$, along with sufficient information for others to verify that the curve was indeed chosen pseudo-randomly. (The algorithm is consistent with the one given in ANSI X9.62-1998 [B11]).

See 5.5 for the conversion routines BS2IP and I2BSP.

It is assumed that the following quantities have been chosen:

- A prime modulus p
- Lower and upper bounds $r_{\min}$ and $r_{\max}$ for the order of the base point
- A cryptographic hash function H with output length B bits, where

$$B \geq \left\lceil \frac{1}{2} \log_2(r_{\min}) \right\rceil$$

- The bit length L of inputs to H, satisfying $L \geq B$

The following notation is adopted below:

$$v = \lceil \log_2 p \rceil$$
$$s = \lfloor (v-1)/B \rfloor$$
$$w = v - B\,s - 1$$

Input: A prime modulus p; lower and upper bounds $r_{\min}$ and $r_{\max}$ for r; a trial division bound $l_{\max} < r_{\min}$

Output: A bit string X; EC parameters $q = p$, a, b, r, and G

```
    1.  Choose an arbitrary bit string $X$ of bit length $L$.
    2.  Compute $h := H(X)$.
    3.  Let $W_0$ be the bit string obtained by taking the $w$ rightmost bits of $h$.
    4.  Convert the length-$L$ bit string $X$ to an integer $z$ via BS2IP.
    5.  For $i$ from 1 to $s$ do
        5.1  Convert the integer $(z + i) \bmod (2^L)$ to a length-$L$ bit string $X_i$ via I2BSP.
        5.2  Compute $W_i := H(X_i)$.
    6.  Let $W$ be the bit string obtained by the concatenation of $W_0, W_1, \ldots, W_s$ as follows:
          $W = W_0 \| W_1 \| \ldots \| W_s$.
    7.  Convert the length-$(v - 1)$ bit string $W$ to an integer $c$ via BS2IP.
    8.  If $c = 0$ or $4c + 27 \equiv 0 \pmod{p}$, then go to step 1.
    9.  Choose integers $a, b \in GF(p)$ such that
          $cb^2 \equiv a^3 \pmod{p}$.
```

(The simplest choice is $a = c$ and $b = c$. However, one may want to choose differently for performance reasons; e.g., the condition $a = p - 3$ used in A.10.4.)

```
10. Compute the order $u$ of the elliptic curve $E$ over $GF(p)$ given by $y^2 = x^3 + ax + b$. (This can be
    done using the techniques described in ANSI X9.TG-17 [B14].)
11. Test $u$ for near-primality via A.15.5.
12. If $u$ is not nearly prime, then go to step 1; otherwise, the output of A.15.5 consists of the
    integers $k, r$.
13. Generate a point $G$ on $E$ of order $r$ via A.11.3.
14. Output $X, a, b, r, G$.
```

A.12.5 Verification of elliptic curve pseudo-randomness (prime case)

The following algorithm verifies the validity of a set of elliptic curve parameters. In addition, it determines whether an elliptic curve over $GF(p)$ was generated using the method of A.12.4.

The quantities B, L, v, s, and w, and the hash function H, are as in A.12.4. See 5.5 for the conversion routines BS2IP and I2BSP.

Input: A bit string X of length L; EC parameters $q = p$, a, b, r, and $G = (x, y)$

Output: “True” or “False”

```
1.  Compute $h := H(X)$.
2.  Let $W_0$ be the bit string obtained by taking the $w$ rightmost bits of $h$.
3.  Convert the bit string $X$ to an integer $z$ via BS2IP.
4.  For $i$ from 1 to $s$ do
    4.1  Convert the integer $(z + i) \bmod (2^L)$ to a length-$L$ bit string $X_i$ via I2BSP.
    4.2  Compute $W_i := H(X_i)$.
5.  Let $W$ be the bit string obtained by the concatenation of $W_0, W_1, \ldots, W_s$ as follows:
      $W = W_0 \| W_1 \| \ldots \| W_s$
6.  Convert the length-$(v - 1)$ bit string $W$ to an integer $c$ via BS2IP.
7.  Perform the following checks:
    7.1  $c > 0$
```

```
        7.2  $(4c + 27 \bmod p) > 0$
        7.3  $cb^2 \equiv a^3$ (mod p)
        7.4  $G \neq \mathrm{O}$
        7.5  $y^2 \equiv x^3 + ax + b \pmod{p}$
        7.6  $rG = \mathrm{O}$.
    8.  If all the checks in step 7 work, then output "True"; otherwise output "False."
```

A.12.6 Constructing verifiably pseudo-random elliptic curves (binary case)

It is common to use an elliptic curve selected at random from the curves of appropriate order (see A.9.5). The following algorithm produces a set of elliptic curve parameters for such a curve over a field $GF(2^m)$, along with sufficient information for others to verify that the curve was indeed chosen pseudo-randomly. (The algorithm is consistent with the one given in ANSI X9.62-1998 [B11].)

See 5.5 for the conversion routines BS2IP, I2BSP, BS2OSP, and OS2FEP.

It is assumed that the following quantities have been chosen:

- A field $GF(2^m)$
- Lower and upper bounds $r_{\min}$ and $r_{\max}$ for the order of the base point
- A cryptographic hash function H with output length B bits, where

$$B \geq \left\lceil \frac{1}{2}\log_2(r_{\min}) \right\rceil$$

- The bit length L of inputs to H, satisfying $L \geq B$.

The following notation is adopted below:

$s = \lfloor (m-1)/B \rfloor$
$w = m - \mathrm{B}\ s$

Input: A field $GF(2^m)$; lower and upper bounds $r_{\min}$ and $r_{\max}$ for r; a trial division bound $l_{\max} < r_{\min}$

Output: A bit string X; EC parameters $q = 2^m$, a, b, r, and G

```
    1.  Choose an arbitrary bit string $X$ of bit length $L$.
    2.  Compute $h := H(X)$.
    3.  Let $W_0$ be the bit string obtained by taking the $w$ rightmost bits of $h$.
    4.  Convert the length-$L$ bit string $X$ to an integer $z$ via BS2IP.
    5.  For $i$ from 1 to $s$ do
        5.1  Convert the integer $(z + i) \bmod (2^L)$ to a length-$L$ bit string $X_i$ via I2BSP.
        5.2  Compute $W_i := H(X_i)$.
    6.  Let $W$ be the bit string obtained by the concatenation of $W_0, W_1, \ldots, W_s$ as follows:
            $W = W_0 \parallel W_1 \parallel \ldots \parallel W_s$
    7.  Convert the length-$m$ bit string $W$ to a field element $b$ via BS2OSP and OS2FEP.
```

8. If $b = 0$, then go to step 1.
9. Let a be an arbitrary element in $GF\ (2^m)$. (The simplest choice is $a = 0$, which also allows for the efficient implementation given in A.10.7. However, one may want to choose differently for other reasons, such as other performance issues or the availability of suitable curves.)
10. Compute the order u of the elliptic curve E over $GF\ (2^m)$ given by $y^2 + xy = x^3 + ax^2 + b$. (This can be done using the techniques described in ANSI X9.TG-17 [B14]. See also Menezes [B109].)
11. Test u for near-primality via A.15.5.
12. If u is not nearly prime, then go to step 1; otherwise, the output of A.15.5 consists of the integers k, r.
13. Generate a point G on E of order r via A.11.3.
14. Output X, a, b, r, G.

A.12.7 Verification of elliptic curve pseudo-randomness (binary case)

The following algorithm verifies the validity of a set of elliptic curve parameters. In addition, it determines whether an elliptic curve over $GF\ (2^m)$ was generated using the method of A.12.6.

The quantities B, L, s, and w, and the hash function H, are as in A.12.6. See 5.5 for the conversion routines BS2IP, I2BSP, BS2OSP, and OS2FEP.

Input: A bit string X of length L; EC parameters $q = 2^m$, a, b, r, and $G = (x, y)$

Output: “True” or “False”

1. Compute $h := H\ (X)$.
2. Let W_0 be the bit string obtained by taking the w rightmost bits of h.
3. Convert the bit string X to an integer z via BS2IP.
4. For i from 1 to s do

    4.1 Convert the integer $(z + i) \bmod (2^L)$ to a length-L bit string X_i via I2BSP.

    4.2 Compute $W_i := H\ (X_i)$.

5. Let W be the bit string obtained by the concatenation of $W_0, W_1, \ldots, W_s$ as follows:

    $W = W_0 \parallel W_1 \parallel \ldots \parallel W_s$

6. Convert the length-m bit string W to the field element b' via BS2OSP and OS2FEP.
7. Perform the following checks:

    7.1 $b \neq 0$

    7.2 $b = b'$

    7.3 $G \neq O$

    7.4 $y^2 + xy = x^3 + ax^2 + b$

    7.5 $rG = O$.

8. If all the checks in step 7 work, then output “True”; otherwise output “False.”

A.12.8 Decompression of y coordinates (prime case)

The following algorithm recovers the y coordinate of an elliptic curve point from its compressed form.

Input: A prime number p; an elliptic curve E defined modulo p; the x coordinate of a point (x, y) on E; the compressed representation $\tilde{y}$ of the y coordinate

Output: The y coordinate of the point

1. Compute $g := x^3 + ax + b \bmod p$.
2. Find a square root z of g modulo p via A.2.5. If the output of A.2.5 is "no square roots exist," then return an error message and stop.
3. Let $\tilde{z}$ be the rightmost bit of z (in other words, $z \bmod 2$).
4. If $\tilde{z} = \tilde{y}$, then $y \leftarrow z$, else $y \leftarrow p - z$.
5. Output y.

NOTE—When implementing the algorithm from A.2.5, the existence of modular square roots should be checked. Otherwise, a value may be returned even if no modular square roots exist.

A.12.9 Decompression of y coordinates (binary case)

The following algorithm recovers the y coordinate of an elliptic curve point from its compressed form.

Input: A field $GF(2^m)$; an elliptic curve E defined over $GF(2^m)$; the x coordinate of a point (x, y) on E; the compressed representation $\tilde{y}$ of the y coordinate

Output: The y coordinate of the point

1. If $x = 0$, then compute $y := \sqrt{b}$ via A.4.1 and go to step 7.
2. Compute the field element $\alpha := x^3 + ax^2 + b$ in $GF(2^m)$.
3. Compute the element $\beta := \alpha (x^2)^{-1}$ via A.4.4.
4. Find a field element z such that $z^2 + z = \beta$ via A.4.7. If the output of A.4.7 is "no solutions exist," then return an error message and stop.
5. Let $\tilde{z}$ be the rightmost bit of z.
6. Compute $y := (z + \tilde{z} + \tilde{y})\, x$.
7. Output y.

NOTES

1—When implementing the algorithm from A.4.7, the existence of solutions to the quadratic equation should be checked. Otherwise, a value may be returned even if no solutions exist.

2—If both coordinates are compressed, the x coordinate must be decompressed first and then the y coordinate (see A.12.10).

A.12.10 Decompression of x coordinates (binary case)

The following algorithm recovers the x coordinate of an elliptic curve point from its compressed form. The statement of the algorithm assumes that the point has prime order, because that is the case of cryptographic interest; in fact, however, that condition is unnecessarily restrictive (see A.9.6).

In addition to the EC parameters and the compact representation of the point, it is necessary to possess the trace Tr (a) of the coefficient a of the curve (see A.4.5). If **F** is represented by a polynomial basis

$$B = \{t^{m-1}, \ldots, t, 1\}$$

and m is even, it is also necessary to possess the smallest positive integer s such that Tr (t^s) = 1, because this indicates which bit has been dropped during compression. If a given set of EC parameters is to be reused, these two quantities can be computed once and stored with the parameters.

Input: A field **F** of 2^m elements; an elliptic curve E defined over **F**; the compact form $\tilde{x}$ of the x coordinate of a point P on E of prime order; the trace of the coefficient a of the curve; if **F** is represented by a polynomial basis $\{t^{m-1}, \ldots, t, 1\}$ and m is even, the smallest positive integer s such that Tr (t^s) = 1

Output: The x coordinate of P

*If **F** is represented by a normal basis or m is odd*

```
1. Set $x^* := (\tilde{x} \parallel 0)$.
2. If $\mathrm{Tr}(x^*) = \mathrm{Tr}(a)$, then set $x \leftarrow x^*$; else set $x := (\tilde{x} \parallel 1)$.
3. Output $x$.
```

*If **F** is represented by a polynomial basis and m is even*

```
1. Write $\tilde{x} = (u_{m-1} \ldots u_1)$.
2. Let $x^*$ be the field element
     $x^* := (w_{m-1} \ldots w_0)$
   where
     $w_j = u_j$ for $j > s$
     $w_s = 0$
     $w_j = u_{j+1}$ for $0 \le j < s$.
   That is, $x^*$ is the polynomial
     $u_{m-1} t^{m-1} + \ldots + u_{s+1} t^{s+1} + u_s t^{s-1} + \ldots + u_1$.
3. If $\mathrm{Tr}(x^*) = \mathrm{Tr}(a)$, then output $x \leftarrow x^*$; else output $x \leftarrow x^* + t^s$.
```

A.13 Class group calculations

The computations listed in this clause are necessary for the complex multiplication technique described in A.14.

A.13.1 Overview

A reduced symmetric matrix is one of the form

$$S = \begin{bmatrix} A & B \\ B & C \end{bmatrix}$$

where the integers A, B, C satisfy the following conditions:

- GCD $(A, 2B, C) = 1$.
- $|2B| \le A \le C$.
- If either $A = |2B|$ or $A = C$, then $B \ge 0$.

The matrix S will be abbreviated as $[A, B, C]$ when typographically convenient.

The determinant $D := AC - B^2$ of S will be assumed throughout this clause to be positive and *squarefree* (i.e., containing no square factors).

Given D, the *class group* $H(D)$ is the set of all reduced symmetric matrices of determinant D. The *class number* $h(D)$ is the number of matrices in $H(D)$.

The class group is used to construct the *reduced class polynomial*. This is a polynomial $w_D\ (t)$ with integer coefficients of degree $h(D)$. The reduced class polynomial is used in A.14 to construct elliptic curves with known orders.

A.13.2 Class group and class number

The following algorithm produces a list of the reduced symmetric matrices of a given determinant D (see Buell [B33]).

Input: A squarefree determinant $D > 0$

Output: The class group $H\ (D)$

```
1.  Let $s$ be the largest integer less than $\sqrt{D/3}$.
2.  For $B$ from 0 to $s$ do
    2.1  List the positive divisors $A_1, \ldots, A_r$ of $D + B^2$ that satisfy $2B \le A \le \sqrt{D + B^2}$.
    2.2  For $i$ from 1 to $r$ do
         2.2.1  Set $C \leftarrow (D + B^2)/A_i$.
         2.2.2  If GCD $(A_i, 2B, C) = 1$, then
                  List $[A_i, B, C]$
                  If $0 < 2B < A_i < C$, then list $[A_i, -B, C]$.
3.  Output list.
```

Example: $D = 71$. The values of B that need to be checked are $0 \le B < 5$.

- $B = 0$ gives $A = 1$, leading to [1,0,71].

- $B = 1$ gives $A = 2,3,4,6,8$, leading to [3, ±1,24] and [8, ±1,9].
- $B = 2$ gives $A = 5$, leading to [5, ±2, 15].
- $B = 3$ gives $A = 8$, but no reduced matrices.
- $B = 4$ gives no divisors A in the right range.

Thus the class group is

$$\mathrm{H}\,(71) = \{[1,0,71], [3, \pm 1,24], [8, \pm 1,9], [5, \pm 2, 15]\}$$

and the class number is $h\,(71) = 7$.

A.13.3 Reduced class polynomials

Let

$$F(z) = 1 + \sum_{j=1}^{\infty} (-1)^j (z^{(3j^2 - j)/2} + z^{(3j^2 + j)/2})$$
$$= 1 - z - z^2 + z^5 + z^7 - z^{12} - z^{15} + \ldots$$

and

$$\theta = \exp\left(\frac{-\sqrt{D} + Bi}{A}\pi\right)$$

Let

$$\mathbf{f}_0(A, B, C) = \theta^{-1/24} F(-\theta)/F(\theta^2)$$
$$\mathbf{f}_1(A, B, C) = \theta^{-1/24} F(\theta)/F(\theta^2)$$
$$\mathbf{f}_2(A, B, C) = \sqrt{2}\ \theta^{1/12} F(\theta^4)/F(\theta^2)$$

NOTE—Since

$$|\theta| < e^{-\pi\sqrt{3}/2} \approx 0.0658287$$

the series $F\,(z)$ used in computing the numbers $\mathbf{f}_J(A, B, C)$ converges as quickly as a power series in $e^{-\pi\sqrt{3}/2}$.

If $[A, B, C]$ is a matrix of determinant D, then its *class invariant* is

$$\mathbf{C}(A, B, C) = (N\,\lambda^{-BL}\,2^{-I/6}\,(\mathbf{f}_J\,(A, B, C))^K)^G$$

where

$$G = \mathrm{GCD}(D,3)$$

$$I = \begin{cases} 3 & \text{if } D \equiv 1, 2, 6, 7 \pmod 8 \\ 0 & \text{if } D \equiv 3 \pmod 8 \text{ and } D \not\equiv 0 \pmod 3 \\ 2 & \text{if } D \equiv 3 \pmod 8 \text{ and } D \equiv 0 \pmod 3 \\ 6 & \text{if } D \equiv 5 \pmod 8 \end{cases}$$

$$J = \begin{cases} 0 & \text{for } AC \text{ odd} \\ 1 & \text{for } C \text{ even} \\ 2 & \text{for } A \text{ even} \end{cases}$$

$$K = \begin{cases} 2 \text{ if } & D \equiv 1, 2, 6 \pmod 8 \\ 1 \text{ if } & D \equiv 3, 7 \pmod 8 \\ 4 \text{ if } & D \equiv 5 \pmod 8 \end{cases}$$

$$L = \begin{cases} A - C + A^2C & \text{if } AC \text{ odd or } D \equiv 5 \pmod 8 \text{ and } C \text{ even} \\ A + 2C - AC^2 & \text{if } D \equiv 1, 2, 3, 6, 7 \pmod 8 \text{ and } C \text{ even} \\ A - C + 5AC^2 & \text{if } D \equiv 3 \pmod 8 \text{ and } A \text{ even} \\ A - C - AC^2 & \text{if } D \equiv 1, 2, 5, 6, 7 \pmod 8 \text{ and } A \text{ even} \end{cases}$$

$$M = \begin{cases} (-1)^{(A^2-1)/8} & \text{if } A \text{ odd} \\ (-1)^{(C^2-1)/8} & \text{if } A \text{ even} \end{cases}$$

$$N = \begin{cases} 1 & \text{if } D \equiv 5 \pmod 8\text{, or } D \equiv 3 \pmod 8 \text{ and } AC \text{ odd, or } D \equiv 7 \pmod 8 \text{ and } AC \text{ even} \\ M & \text{if } D \equiv 1, 2, 6 \pmod 8 \text{ or } D \equiv 7 \pmod 8 \text{ and } AC \text{ odd} \\ -M & \text{if } D \equiv 3 \pmod 8 \text{ and } AC \text{ even} \end{cases}$$

$$\lambda = e^{\pi i K/24}$$

If $[A_1, B_1, C_1]$,..., $[A_h, B_h, C_h]$ are the reduced symmetric matrices of (positive squarefree) determinant D, then the *reduced class polynomial* for D is

$$w_D(t) = \prod_{j=I}^{h} (t - \mathbf{C}(A_j, B_j, C_j))$$

The reduced class polynomial has integer coefficients.

NOTE—The above computations must be performed with sufficient accuracy to identify each coefficient of the polynomial $w_D(t)$. Since each such coefficient is an integer, this means that the error incurred in calculating each coefficient should be less than half.

Example:

$$w_{71}(t) = \left(t - \frac{1}{\sqrt{2}}\mathbf{f}_0\ (1,0,71)\right)$$

$$\left(t - \frac{e^{-i\pi/8}}{\sqrt{2}}\mathbf{f}_1\ (3,1,24)\right)\left(t - \frac{e^{i\pi/8}}{\sqrt{2}}\mathbf{f}_1\ (3,-1,24)\right)$$

$$\left(t - \frac{e^{-23i\pi/24}}{\sqrt{2}}\mathbf{f}_2\ (8,1,9)\right)\left(t - \frac{e^{23i\pi/24}}{\sqrt{2}}\mathbf{f}_2\ (8,-1,9)\right)$$

$$\left(t + \frac{e^{-5i\pi/12}}{\sqrt{2}}\mathbf{f}_0\ (5,2,15)\right)\left(t + \frac{e^{5i\pi/12}}{\sqrt{2}}\mathbf{f}_0\ (5,-2,15)\right)$$

$$= (t - 2.13060682983889533005591468688942503...)$$

$$(t - (0.95969178530567025250797047645507504...) +$$

$$(0.34916071001269654799855316293926907...)\ i)$$

$$(t - (0.95969178530567025250797047645507504...) -$$

$$(0.34916071001269654799855316293926907...)\ i)$$

$$(t + (0.75613568804001789053564010985317772...) +$$

$$(0.07375086316308890052407649445676775...)\ i)$$

$$(t + (0.75613568804001789053564010985317772...) -$$

$$(0.07375086316308890052407649445676775...)\ i)$$

$$(t + (0.26885951218510002700028771004666102...) -$$

$$(0.84108577401329800103648634224905292...)\ i)$$

$$(t + (0.26885951218510002700028771004666102...) +$$

$$(0.84108577401329800103648634224905292...)\ i)$$

$$= t^7 - 2t^6 - t^5 + t^4 + t^3 + t^2 - t - 1$$

A.14 Complex multiplication

A.14.1 Overview

If E is a non-supersingular elliptic curve over $GF\ (q)$ of order u, then

$$Z = 4q - (q + 1 - u)^2$$

is positive by the Hasse bound (see A.9.5). Thus, there is a unique factorization

$$Z = DV^2$$

where D is squarefree (i.e., contains no square factors). Thus, for each non-supersingular elliptic curve over $GF\,(q)$ of order u, there exists a unique squarefree positive integer D such that

$$(*) \qquad 4q = W^2 + DV^2$$

$$(**) \qquad u = q + 1 \pm W$$

for some W and V.

It is said that E has *complex multiplication* by D (or, more properly, by $\sqrt{-D}$). D is called a *CM discriminant* for q.

If one knows D for a given curve E, one can compute its order via (*) and (**). As will be demonstrated below, one can construct the curves with CM by small D. Therefore, one can obtain curves whose orders u satisfy (*) and (**) for small D. The near-primes are plentiful enough that one can find curves of nearly prime order with small enough D to construct.

Over $GF\,(p)$, the CM technique is also called the *Atkin-Morain method* (see Morain [B118]); over $GF\,(2^m)$, it is also called the *Lay-Zimmer method* (see Lay and Zimmer [B99]). Although it is possible [over $GF\,(p)$] to choose the order first and then the field, it is preferable to choose the field first, because there are fields in which the arithmetic is especially efficient.

There are two basic steps involved: finding an appropriate order, and constructing a curve having that order. More precisely, one begins by choosing the field size q, the minimum point order $r_{\min}$, and trial division bound $l_{\max}$. Given those quantities, D is called *appropriate* if there exists an elliptic curve over $GF\,(q)$ with CM by D and having nearly prime order.

Step 1 (A.14.2 and A.14.3 Finding a nearly prime order)

Find an appropriate D. When one is found, record D, the large prime r, and the positive integer k, such that $u = kr$ is the nearly prime curve order.

Step 2 (A.14.4 and A.14.5 Constructing a curve and point)

Given D, k, and r, construct an elliptic curve over $GF\,(q)$ and a point of order r.

A.14.2 Finding a nearly prime order over GF (p)

A.14.2.1 Congruence conditions

A squarefree positive integer D can be a CM discriminant for p only if it satisfies the following congruence conditions. Let

$$K = \left\lfloor \frac{(\sqrt{p}+1)^2}{r_{\min}} \right\rfloor$$

— If $p \equiv 3 \pmod 8$, then $D \equiv 2, 3$, or $7 \pmod 8$.

— If $p \equiv 5 \pmod 8$, then D is odd.

— If $p \equiv 7 \pmod 8$, then $D \equiv 3, 6$, or $7 \pmod 8$.

— If $K = 1$, then $D \equiv 3 \pmod 8$.

— If $K = 2$ or 3, then $D \not\equiv 7 \pmod 8$.

Thus, the possible squarefree values of D are as follows:

If $K = 1$, then

$D = 3, 11, 19, 35, 43, 51, 59, 67, 83, 91, 107, 115, \ldots.$

If $p \equiv 1 \pmod 8$ and $K = 2$ or 3, then

$D = 1, 2, 3, 5, 6, 10, 11, 13, 14, 17, 19, 21, \ldots.$

If $p \equiv 1 \pmod 8$ and $K \geq 4$, then

$D = 1, 2, 3, 5, 6, 7, 10, 11, 13, 14, 15, 17, \ldots.$

If $p \equiv 3 \pmod 8$ and $K = 2$ or 3, then

$D = 2, 3, 10, 11, 19, 26, 34, 35, 42, 43, 51, 58, \ldots.$

If $p \equiv 3 \pmod 8$ and $K \geq 4$, then

$D = 2, 3, 7, 10, 11, 15, 19, 23, 26, 31, 34, 35, \ldots.$

If $p \equiv 5 \pmod 8$ and $K = 2$ or 3, then

$D = 1, 3, 5, 11, 13, 17, 19, 21, 29, 33, 35, 37, \ldots.$

If $p \equiv 5 \pmod 8$ and $K \geq 4$, then

$D = 1, 3, 5, 7, 11, 13, 15, 17, 19, 21, 23, 29, \ldots.$

If $p \equiv 7 \pmod 8$ and $K = 2$ or 3, then

$D = 3, 6, 11, 14, 19, 22, 30, 35, 38, 43, 46, 51, \ldots.$

If $p \equiv 7 \pmod 8$ and $K \geq 4$, then

$D = 3, 6, 7, 11, 14, 15, 19, 22, 23, 30, 31, 35, \ldots.$

A.14.2.2 Testing for CM discriminants (prime case)

Input: A prime p and a squarefree positive integer D satisfying the congruence conditions from A.14.2.1

Output: If D is a CM discriminant for p, an integer W such that

$$4p = W^2 + DV^2$$

for some V. (In the cases $D = 1$ or 3, the output also includes V.) If not, the message “not a CM discriminant.”

1. Apply the appropriate technique from A.2.5 to find a square root modulo p of $-D$ or determine that none exists.
2. If the result of step 1 indicates that no square roots exist, then output “not a CM discriminant” and stop. Otherwise, the output of step 1 is an integer B modulo p.
3. Let $A \leftarrow p$ and $C \leftarrow (B^2 + D) / p$.
4. Let $S \leftarrow \begin{bmatrix} A & B \\ B & C \end{bmatrix}$ and $U \leftarrow \begin{bmatrix} 1 \\ 0 \end{bmatrix}$.
5. Until $|2B| \le A \le C$, repeat the following steps:

   5.1 Let $\delta \leftarrow \left\lfloor \frac{B}{C} + \frac{1}{2} \right\rfloor$.

   5.2 Let $T \leftarrow \begin{bmatrix} 0 & -1 \\ 1 & \delta \end{bmatrix}$.

   5.3 Replace U by $T^{-1}U$.

   5.4 Replace S by $T^t\, S\, T$, where T^t denotes the transpose of T.
6. If $D = 11$ and $A = 3$, let $\delta \leftarrow 0$ and repeat 5.2, 5.3, 5.4.
7. Let X and Y be the entries of U; that is,

   $$U = \begin{pmatrix} X \\ Y \end{pmatrix}.$$
8. If $D = 1$ or 3, then output $W \leftarrow 2X$ and $V \leftarrow 2Y$ and stop.
9. If $A = 1$, then output $W \leftarrow 2X$ and stop.
10. If $A = 4$, then output $W \leftarrow 4X + BY$ and stop.
11. Output “not a CM discriminant.”

A.14.2.3 Finding a nearly prime order (prime case)

Input: A prime p; a trial division bound $l_{\max}$; lower and upper bounds $r_{\min}$ and $r_{\max}$ for base point order

Output: A squarefree positive integer D; a prime r in the interval $r_{\min} \le r \le r_{\max}$; a smooth integer k such that $u = kr$ is the order of an elliptic curve modulo p with complex multiplication by D

1. Choose a squarefree positive integer D, not already chosen, satisfying the congruence conditions of A.14.2.1.
2. Compute via A.2.3 the Jacobi symbol $J = \left(\frac{-D}{P}\right)$. If $J = -1$, then go to step 1.
3. List the odd primes l dividing D.
4. For each l, compute via A.2.3 the Jacobi symbol $J = \left(\frac{p}{l}\right)$. If $J = -1$ for some l, then go to step 1.
5. Test via A.14.2.2 whether D is a CM discriminant for p. If the result is “not a CM discriminant,” go to step 1. (Otherwise, the result is the integer W, along with V if $D = 1$ or 3.)
6. Compile a list of the possible orders, as follows:

- If $D = 1$, the orders are

  $p + 1 \pm W, p + 1 \pm V;$

- If D = 3, the orders are

  $p + 1 \pm W, p + 1 \pm (W + 3V)/2, p + 1 \pm (W - 3V)/2;$

- Otherwise, the orders are $p + 1 \pm W$.

7. Test each order for near-primality via A.15.5. If any order is nearly prime, output (D, k, r) and stop.

8. Go to step 1.

Example: Let $p = 2^{192} - 2^{64} - 1$. Then

$$p = 4X^2 - 2XY + \frac{1+D}{4} Y^2 \text{ and } p + 1 - (4X - Y) = r$$

where $D = 235$

$X = -31037252937617930835957687234$

$Y = 5905046152393184521033305113$

and r is the prime

$r = 6277101735386680763835789423337720473986773608255189015329.$

Thus, there is a curve modulo p of order r having complex multiplication by D.

A.14.3 Finding a nearly prime order over GF (2^m)

A.14.3.1 Testing for CM discriminants (binary case)

Input: A field degree d and a squarefree positive integer $D \equiv 7 \pmod 8$

Output: If D is a CM discriminant for 2^d, an odd integer W such that

$2^{d+2} = W^2 + DV^2$

for some odd V. If not, the message “not a CM discriminant.”

1. Compute via A.2.6 an integer B such that $B^2 \equiv -D \pmod{2^{d+2}}$.

2. Let $A \leftarrow 2^{d+2}$ and $C \leftarrow (B^2 + D) / 2^{d+2}$. (Note that the variables A and C will remain positive throughout the algorithm.)

3. Let $S \leftarrow \begin{bmatrix} A & B \\ B & C \end{bmatrix}$ and $U \leftarrow \begin{bmatrix} 1 \\ 0 \end{bmatrix}$.

4. Until $|2B| \leq A \leq C$, repeat the following steps:

   4.1 Let $\delta \leftarrow \left\lfloor \frac{B}{C} + \frac{1}{2} \right\rfloor$.

4.2 Let $T \leftarrow \begin{bmatrix} 0 & -1 \\ 1 & \delta \end{bmatrix}$.

4.3 Replace U by $T^{-1}U$.

4.4 Replace S by $T^t\, S\, T$, where T^t denotes the transpose of T.

5. Let X and Y be the entries of U; that is

$$U = \begin{bmatrix} X \\ Y \end{bmatrix}.$$

6. If $A = 1$, then output $W \leftarrow X$ and stop.

7. If $A = 4$ and Y is even, then output $W \leftarrow (4X + BY) / 2$ and stop.

8. Output “not a CM discriminant.”

A.14.3.2 Finding a nearly prime order (binary case)

Input: A field degree d; a trial division bound $l_{\max}$; lower and upper bounds $r_{\min}$ and $r_{\max}$ for base point order

Output: A squarefree positive integer D; a prime r in the interval $r_{\min} \le r \le r_{\max}$; a smooth integer k such that $u = kr$ is the order of an elliptic curve over $GF\,(2^d)$ with complex multiplication by D

1. Choose a squarefree positive integer $D \equiv 7 \pmod 8$, not already chosen.
2. Compute $H \leftarrow$ the class group for D via A.13.2.
3. Set $h \leftarrow$ the number of elements in H.
4. If d does not divide h, then go to step 1.
5. Test via A.14.3.1 whether D is a CM discriminant for 2^d. If the result is “not a CM discriminant,” go to step 1; otherwise, the result is the integer W.
6. The possible orders are $2^d + 1 \pm W$.
7. Test each order for near-primality via A.15.5. If any order is nearly prime, output (D, k, r) and stop.
8. Go to step 1.

Example: Let $q = 2^{155}$. Then

$$4q = X^2 + DY^2 \quad \text{and} \quad q + 1 - X = 4r$$

where

$D =$	942679
$X =$	229529878683046820398181
$Y =$	–371360755031779037497 and
r	is the prime

$$r = 11417981541647679048466230373126290329356873447.$$

Thus, there is a curve over $GF\,(q)$ of order $4r$ having complex multiplication by D.

A.14.4 Constructing a curve and point (prime case)

A.14.4.1 Constructing a curve with prescribed CM (prime case)

Given a prime p and a CM discriminant D, the following technique produces an elliptic curve $y^2 \equiv x^3 + a_0\,x + b_0 \pmod p$ with CM by D. (Note that there are at least two possible orders among curves with CM by D. The curve constructed here will have the proper CM, but not necessarily the desired order. This curve will be replaced in A.14.4.2 by one of the desired order.)

For nine values of D, the coefficients of E can be written down at once, as shown in Table A.5.

Table A.5—Coefficients of E for certain values of D

D	a_0	b_0
1	1	0
2	–30	56
3	0	1
7	–35	98
11	–264	1694
19	–152	722
43	–3440	77658
67	–29480	1948226
163	–8697680	9873093538

For other values of D, the following algorithm may be used:

Input: A prime modulus p and a CM discriminant $D > 3$ for p

Output: a_0 and b_0 such that the elliptic curve

$$y^2 \equiv x^3 + a_0 x + b_0 \pmod p$$

has CM by D.

1. Compute $w(t) \leftarrow w_D(t) \bmod p$ via A.13.3.
2. Let W be the output from A.14.2.2.
3. If W is even, then use A.5.3 with $d = 1$ to compute a linear factor $t - s$ of $w_D(t)$ modulo p. Let

   $$V := (-1)^D\, 2^{4I/K}\, s^{24/(GK)} \bmod p$$

   where G, I, and K are as in A.13.3. Finally, let

   $$a_0 := -3(V + 64)(V + 16) \bmod p$$

   $$b_0 := 2(V + 64)^2\,(V - 8) \bmod p.$$
4. If W is odd, then use A.5.3 with $d = 3$ to find a cubic factor $g\,(t)$ of $w_D(t)$ modulo p. Perform the following computations, in which the coefficients of the polynomials are integers modulo p:

$$V(t) := \begin{cases} -t^{24} \bmod g(t) & \text{if } 3 \text{ does not divide } D \\ -256t^{8} \bmod g(t) & \text{if } 3 \text{ divides } D \end{cases}$$

$$a_1(t) := -3(V(t) + 64)\,(V(t) + 256) \bmod g(t)$$

$$b_1(t) := 2(V(t) + 64)^2\,(V(t) - 512) \bmod g(t)$$

$$a_3(t) := a_1(t)^3 \bmod g(t)$$

$$b_2(t) := b_1(t)^2 \bmod g(t).$$

Now let σ be a nonzero coefficient from $a_3(t)$, and let τ be the corresponding coefficient from $b_2(t)$. Finally, let

$$a_0 := \sigma\tau \bmod p$$

$$b_0 := \sigma\tau^2 \bmod p.$$

5. Output (a_0, b_0).

Example: If $D = 235$, then

$$w_D(t) = t^6 - 10\,t^5 + 22\,t^4 - 24\,t^3 + 16\,t^2 - 4\,t + 4$$

If $p = 2^{192} - 2^{64} - 1$, then

$$w_D(t) \equiv (t^3 - (5 + \phi)t^2 + (1 - \phi)t - 2)\,(t^3 - (5 - \phi)t^2 + (1 + \phi)t - 2) \pmod{p}$$

where $\phi = 1254098248316315745658220082226751383299177953632927607231$. The resulting coefficients are

$$a_0 = -2089023816294079213892272128$$

$$b_0 = -36750495627461354054044457602630966837248$$

Thus the curve $y^2 \equiv x^3 + a_0x + b_0$ modulo p has CM by $D = 235$.

A.14.4.2 Choosing the curve and point (prime case)

Input: EC parameters p, k, and r, and coefficients a_0, b_0 produced by A.14.4.1

Output: A curve E modulo p and a point G on E of order r, or a message "wrong order"

1. Select an integer ξ with $0 < \xi < p$.
2. If $D = 1$, then set $a \leftarrow a_0\xi \bmod p$ and $b \leftarrow 0$; if $D = 3$, then set $a \leftarrow 0$ and $b \leftarrow b_0\xi \bmod p$. Otherwise, set $a \leftarrow a_0\xi^2 \bmod p$ and $b \leftarrow b_0\xi^3 \bmod p$.
3. Look for a point G of order r on the curve

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

   via A.11.3.

4. If the output of A.11.3 is “wrong order,” then output the message “wrong order” and stop.

5. Output the coefficients a, b, and the point G.

The method of selecting ξ in the first step of this algorithm depends on the kind of coefficients desired. Two examples follow:

- If $D \neq 1$ or 3, and it is desired that $a = -3$ (see A.10.4), then take ξ to be a solution of the congruence $a_0\xi^2 \equiv -3 \pmod{p}$, provided one exists. If one does not exist, or if this choice of ξ leads to the message “wrong order,” then select another curve as follows. If $p \equiv 3 \pmod{4}$ and the result was “wrong order,” then choose $p - \xi$ in place of ξ; the result leads to a curve with $a = -3$ and the right order. If no solution ξ exists, or if $p \equiv 1 \pmod{4}$, then repeat A.14.4.1 with another root of the reduced class polynomial. The proportion of roots leading to a curve with $a = -3$ and the right order is roughly one-half if $p \equiv 3 \pmod{4}$, and one-quarter if $p \equiv 1 \pmod{4}$.
- If there is no restriction on the coefficients, then choose ξ at random. If the output is the message “wrong order,” then repeat the algorithm until a set of parameters a, b, G is obtained. This will happen for half the values of ξ, unless $D = 1$ (one-quarter of the values) or $D = 3$ (one-sixth of the values).

A.14.5 Constructing a curve and point (binary case)

A.14.5.1 Constructing a curve with prescribed CM (binary case)

Input: A field $GF(2^m)$; a CM discriminant D for 2^m; the desired curve order u

Output: a and b, such that the elliptic curve

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

over $GF(2^m)$ has order u.

```
1. Compute $w(t) \leftarrow w_D(t) \bmod 2$ via A.13.3.
2. Use A.14.3.1 to find the smallest divisor $d$ of $m$ greater than $(\log_2 D) - 2$, such that $D$ is a CM
   discriminant for $2^d$.
3. Compute $p(t)$ := a degree $d$ factor modulo 2 of $w(t)$. (If $d = h$, then $p(t)$ is just $w(t)$ itself. If $d <$
   $h$, $p(t)$ is found via A.5.4.)
4. Compute $\alpha$ := a root in $GF(2^m)$ of $p(t) = 0$ via A.5.6.
5. If 3 divides $D$, then
      set $b \leftarrow \alpha$;
      else set $b \leftarrow \alpha^3$.
6. If $u$ is divisible by 4, then set $a \leftarrow 0$;
      else if $m$ is odd, then set $a \leftarrow 1$;
      else generate (by trial and error using A.4.5) a random element $a \in GF(2^m)$ of trace 1.
7. Output $(a, b)$.
```

Example: If D = 942679, then

$$\begin{gathered}w_D(t) \equiv 1 + t^2 + t^6 + t^{10} + t^{12} + t^{13} + t^{16} + t^{17} + t^{20} + t^{22} + t^{24} + t^{27} + t^{30} + t^{33} + t^{35} + t^{36} + t^{37} + t^{41} + t^{42} + t^{43} \\
+ t^{45} + t^{49} + t^{51} + t^{54} + t^{56} + t^{57} + t^{59} + t^{61} + t^{65} + t^{67} + t^{68} + t^{69} + t^{70} + t^{71} + t^{72} + t^{74} + t^{75} + t^{76} + t^{82} + t^{83} + \\
t^{87} + t^{91} + t^{93} + t^{96} + t^{99} + t^{100} + t^{101} + t^{102} + t^{103} + t^{106} + t^{108} + t^{109} + t^{110} + t^{114} + t^{117} + t^{119} + t^{121} + t^{123} + \\
t^{125} + t^{126} + t^{128} + t^{129} + t^{130} + t^{133} + t^{134} + t^{140} + t^{141} + t^{145} + t^{146} + t^{147} + t^{148} + t^{150} + t^{152} + t^{154} + t^{155} + \\
t^{157} + t^{158} + t^{160} + t^{161} + t^{166} + t^{167} + t^{171} + t^{172} + t^{175} + t^{176} + t^{179} + t^{180} + t^{185} + t^{186} + t^{189} + t^{190} + t^{191} + \\
t^{192} + t^{195} + t^{200} + t^{201} + t^{207} + t^{208} + t^{209} + t^{210} + t^{211} + t^{219} + t^{221} + t^{223} + t^{225} + t^{228} + t^{233} + t^{234} + t^{235} + \\
t^{237} + t^{238} + t^{239} + t^{241} + t^{242} + t^{244} + t^{245} + t^{248} + t^{249} + t^{250} + t^{252} + t^{253} + t^{255} + t^{257} + t^{260} + t^{262} + t^{263} + \\
t^{264} + t^{272} + t^{273} + t^{274} + t^{276} + t^{281} + t^{284} + t^{287} + t^{288} + t^{289} + t^{290} + t^{292} + t^{297} + t^{299} + t^{300} + t^{301} + t^{302} + \\
t^{304} + t^{305} + t^{306} + t^{309} + t^{311} + t^{312} + t^{313} + t^{314} + t^{317} + t^{318} + t^{320} + t^{322} + t^{323} + t^{325} + t^{327} + t^{328} + t^{329} + \\
t^{333} + t^{335} + t^{340} + t^{341} + t^{344} + t^{345} + t^{346} + t^{351} + t^{353} + t^{354} + t^{355} + t^{357} + t^{358} + t^{359} + t^{360} + t^{365} + t^{366} + \\
t^{368} + t^{371} + t^{372} + t^{373} + t^{376} + t^{377} + t^{379} + t^{382} + t^{383} + t^{387} + t^{388} + t^{389} + t^{392} + t^{395} + t^{398} + t^{401} + t^{403} + \\
t^{406} + t^{407} + t^{408} + t^{409} + t^{410} + t^{411} + t^{416} + t^{417} + t^{421} + t^{422} + t^{423} + t^{424} + t^{425} + t^{426} + t^{429} + t^{430} + t^{438} + \\
t^{439} + t^{440} + t^{441} + t^{442} + t^{443} + t^{447} + t^{448} + t^{450} + t^{451} + t^{452} + t^{453} + t^{454} + t^{456} + t^{458} + t^{459} + t^{460} + t^{462} + \\
t^{464} + t^{465} + t^{466} + t^{467} + t^{471} + t^{473} + t^{475} + t^{476} + t^{481} + t^{482} + t^{483} + t^{484} + t^{486} + t^{487} + t^{488} + t^{491} + t^{492} + \\
t^{495} + t^{496} + t^{498} + t^{501} + t^{503} + t^{505} + t^{507} + t^{510} + t^{512} + t^{518} + t^{519} + t^{529} + t^{531} + t^{533} + t^{536} + t^{539} + t^{540} + \\
t^{541} + t^{543} + t^{545} + t^{546} + t^{547} + t^{548} + t^{550} + t^{552} + t^{555} + t^{556} + t^{557} + t^{558} + t^{559} + t^{560} + t^{563} + t^{565} + t^{566} + \\
t^{568} + t^{580} + t^{585} + t^{588} + t^{589} + t^{591} + t^{592} + t^{593} + t^{596} + t^{597} + t^{602} + t^{604} + t^{606} + t^{610} + t^{616} + t^{620} \pmod 2.\end{gathered}$$

This polynomial factors into four irreducibles over GF (2), each of degree 155. One of these is

$$\begin{gathered}p(t) = 1 + t + t^2 + t^6 + t^9 + t^{10} + t^{11} + t^{13} + t^{14} + t^{15} + t^{16} + t^{18} + t^{19} + t^{22} + t^{23} + t^{26} + t^{27} + t^{29} + t^{31} + t^{49} + t^{50} \\
+ t^{51} + t^{54} + t^{55} + t^{60} + t^{61} + t^{62} + t^{64} + t^{66} + t^{70} + t^{72} + t^{74} + t^{75} + t^{80} + t^{82} + t^{85} + t^{86} + t^{88} + t^{89} + t^{91} + t^{93} + \\
t^{97} + t^{101} + t^{103} + t^{104} + t^{111} + t^{115} + t^{116} + t^{117} + t^{118} + t^{120} + t^{121} + t^{123} + t^{124} + t^{126} + t^{127} + t^{128} + t^{129} + \\
t^{130} + t^{131} + t^{132} + t^{134} + t^{136} + t^{137} + t^{138} + t^{139} + t^{140} + t^{143} + t^{145} + t^{154} + t^{155}.\end{gathered}$$

If t is a root of $p(t)$, then the curve

$$y^2 + xy = x^3 + t^3$$

over GF (2^{155}) has order $4r$, where r is the prime

$$r = 11417981541647679048466230373126290329356873447$$

A.14.5.2 Choosing the curve and point (binary case)

Input: A field size GF (2^m); an appropriate D; the corresponding k and r from A.14.3.2

Output: A curve E over GF (2^m) and a point G on E of order r

1. Compute a and b via A.14.5.1 with $u = kr$.

2. Find a point G of order r via A.11.3.

3. Output the coefficients a, b, and the point G.

A.15 Primality tests and proofs

Techniques for generating and testing primes are also provided in ANSI X9.80 [B13].

A.15.1 A Probabilistic primality test

If n is a large positive integer, the following probabilistic algorithm (the *strong probable prime test* or the *Miller-Rabin test*) will determine whether n is prime or composite, with arbitrarily small probability of error (see Knuth [B93]).

Input: An odd integer $n > 2$ to be tested; a positive integer t for the number of trials

Output: The message “prime” or “composite”

```
1. Compute $v$ and odd $w$ such that $n – 1 = 2^v w$.
2. For $j$ from 1 to $t$ do
    2.1 Choose random $a$ in the interval $0 < a < n$.
    2.2 Set $b \leftarrow a^w \bmod n$.
    2.3 If $b = 1$ or $n – 1$, go to step 2.6.
    2.4 For $i$ from 1 to $v – 1$ do
        2.4.1 Set $b \leftarrow b^2 \bmod n$.
        2.4.2 If $b = n – 1$ go to step 2.6.
        2.4.3 If $b = 1$, output “composite” and stop.
        2.4.4 Next $i$.
    2.5 Output “composite” and stop.
    2.6 Next $j$.
3. Output “prime.”
```

If the algorithm outputs “composite,” then n is composite. If the algorithm outputs “prime” then n is almost certainly prime.

A.15.2 Testing a randomly generated integer for primality

If a random k-bit integer n has tested “prime” after t trials of the Miller-Rabin test, then the probability that n is composite is at most

$$P_{k,t} = 2^{t+4} k (2^{-\sqrt{tk}}) \sqrt{\frac{k}{t}}$$

(provided that $t > 1$ and $k \geq 88$; see Damgard, Landrock, and Pomerance [B43]).

To achieve a probability of error less than 2^{-100} for a random k-bit integer, one should choose the number t of trials according to Table A.6.

The values of t in Table A.6 can be lowered in many cases (see Burthe [B35] and Damgard, Landrock, and Pomerance [B43]).

Table A.6—Number of trials for the Miller–Rabin test

k	*t*	*k*	*t*	*k*	*t*
160	34	202-208	23	335-360	12
161-163	33	209-215	22	361-392	11
164-166	32	216-222	21	393-430	10
167-169	31	223-231	20	431-479	9
170-173	30	232-241	19	480-542	8
174-177	29	242-252	18	543-626	7
178-181	28	253-264	17	627-746	6
182-185	27	265-278	16	747-926	5
186-190	26	279-294	15	927-1232	4
191-195	25	295-313	14	1233-1853	3
196-201	24	314-334	13	1854-up	2

A.15.3 Validating primality of a given integer

In *generating* parameters, one can take random numbers and test them for primality according to A.15.2. When *verifying* parameters, however, one cannot treat the putative prime number as random. In this case, the Miller-Rabin test should be run $t = 50$ times in order to achieve a probability of error less than 2^{-100}.

A.15.4 Proving primality

Another application of the complex multiplication algorithm of A.14 is *proving* the primality of an integer deemed “prime” by a probabilistic test, such as in A.15.1. The *Goldwasser-Kilian-Atkin algorithm* produces a *primality certificate:* a collection

$$\mathsf{C} = \{C_1, \ldots C_s\}$$

in which each component C_i consists of positive integers

$$C_i = (p_i, r_i, a_i, b_i, x_i, y_i)$$

where

— for all i, (x_i, y_i) is a point of order r_i on the elliptic curve $y^2 \equiv x^3 + a_i x + b_i \pmod{p_i}$

— $\sqrt{r_i} > \sqrt[4]{p_i} + 1$ for all i

— $p_1 = n$

— $p_{i+1} = r_i$ for $1 \le i < s$

— $r_s < l_{\max}^2$

— r_s is proved prime by trial division.

If a primality certificate exists for n, then n is prime by the following result (see Goldwasser and Kilian [B60]):

Let p and r be positive integers greater than 3 with $\sqrt{r} > \sqrt[4]{p} + 1$ *. Let a, b, x, and y be integers modulo p, such that (x, y) is a point of order r on the elliptic curve*

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

Then, p is prime if r is.

The GKA algorithm can be implemented as follows:

Input: A large odd positive integer n that has tested “prime” in A.15.1, and a trial division bound $l_{\max}$

Output: A primality certificate for n, or an error message

```
1.  Set C ← {}.
2.  Set $i \leftarrow 0$.
3.  Set $r \leftarrow n$.
4.  While $r > l_{\max}^2$ do
    4.1  Set $i \leftarrow i + 1$.
    4.2  Set $p \leftarrow r$.
    4.3  Set $r_{\min} \leftarrow (\sqrt[4]{p} + 1)^2$.
    4.4  Find positive integers $D$, $k$, $r$ (via A.14.2.3) such that
         —  $r \geq r_{\min}$
         —  $r$ tests “prime” in A.15.3
         —  $kr$ is an order of an elliptic curve over $GF(p)$ with CM by $D$.
    4.5  Set $C_i \leftarrow (p, r, D, k)$.
    4.6  Append $C_i$ to C.
5.  $s \leftarrow i$.
6.  Confirm primality of $r$ by trial division. (If it is found that $r$ is composite, then output an error
    message and stop.)
7.  For $i$ from 1 to $s$ do
    7.1  Recover $(p, r, D, k) \leftarrow C_i$.
    7.2  Find via A.14.4.2 a curve
           $E : y^2 \equiv x^3 + ax + b \pmod{p}$
         over $GF(p)$ and a point $(x, y)$ on $E$ of order $r$.
    7.3  Set $C_i \leftarrow (p, r, a, b, x, y)$.
8.  Output C.
```

A.15.5 Testing for near primality

Given lower and upper bounds $r_{\min}$ and $r_{\max}$, and a trial division bound $l_{\max}$, the following procedures test an integer u for near primality in the sense of A.9.5. Let

$$K := \lfloor u/r_{\min} \rfloor$$

— If $K = 1$, then u is nearly prime if and only if it is prime.

— If $K \geq 2$, then the following algorithm tests u for near primality. Note that one can always take $l_{\max} \leq K$.

Input: Positive integers u, $l_{\max}$, $r_{\min}$, and $r_{\max}$

Output: If u is nearly prime, a prime r in the interval $r_{\min} \leq r \leq r_{\max}$, and a smooth integer k such that $u = kr$. If u is not nearly prime, the message "not nearly prime."

```
1. Set $r \leftarrow u$, $k \leftarrow 1$.
2. For $l$ from 2 to $l_{\max}$ do
   2.1 If $l$ is composite, then go to step 2.3.
   2.2 While ($l$ divides $r$)
       2.2.1 Set $r \leftarrow r / l$ and $k \leftarrow k\ l$.
       2.2.2 If $r < r_{\min}$, then output "not nearly prime" and stop.
   2.3 Next $l$.
3. If $r > r_{\max}$, then output "not nearly prime" and stop.
4. Test $r$ for primality via A.15.3 (and A.15.4 if desired).
5. If $r$ is prime, then output $k$ and $r$ and stop.
6. Output "not nearly prime."
```

A.15.6 Generating random primes

The following algorithm generates random primes within a given interval.

The primes p found by this algorithm also satisfy the condition that $p - 1$ is relatively prime to a specified odd positive integer f. Such a condition is required for generating IF parameters. In the case of RSA, f is set equal to the public exponent. In the case of RW, f is set equal to the largest odd divisor of the public exponent. For applications where such a condition is not needed, one takes $f = 1$, since this choice leads to a vacuous condition.

Input: A lower bound $p_{\min}$ for p; an upper bound $p_{\max}$ for p; an odd positive integer f

Output: A random prime p from the range $p_{\min} \leq p \leq p_{\max}$, for which $p - 1$ is relatively prime to f

```
1. Let $k_{\min} := \lceil (p_{\min} - 1) / 2 \rceil$ and $k_{\max} := \lfloor (p_{\max} - 1) / 2 \rfloor$.
2. Generate a random integer $k$ from the range $k_{\min} \leq k \leq k_{\max}$.
3. Set $p \leftarrow 2k + 1$.
4. Compute $d :=$ GCD $(p - 1, f)$ via A.2.2
```

```
5.  If $d = 1$, then
    5.1  Test $p$ for primality via A.15.2 (and A.15.4 if desired).
    5.2  If $p$ is prime, then output $p$ and stop.
6.  Go to step 2.
```

A.15.7 Generating random primes with congruence conditions

The following algorithm differs from the preceding one only in that it imposes the additional condition on the prime p that $p \equiv a \pmod{r}$ for some specified a and r that are relatively prime. The remarks made at the beginning of A.15.6 apply here as well.

Input: Positive integers $r > 2$ and a that are relatively prime to each other; a lower bound $p_{\min}$ for p; an upper bound $p_{\max}$ for p; an odd positive integer f

Output: A random prime p from the range $p_{\min} \le p \le p_{\max}$ satisfying $p \equiv a \pmod{r}$, and for which $p - 1$ is relatively prime to f

```
1.  If $a$ is odd, set $b \leftarrow a$; else set $b \leftarrow a + r$. If $r$ is odd, set $s \leftarrow 2r$; else set $s \leftarrow r$.
2.  Let $k_{\min} := \lceil (p_{\min} - b) / s \rceil$ and $k_{\max} := \lfloor (p_{\max} - b) / s \rfloor$.
3.  Generate a random integer $k$ from the range $k_{\min} \le k \le k_{\max}$.
4.  Set $p \leftarrow sk + b$.
5.  Compute $d :=$ GCD $(p - 1, f)$ via A.2.2.
6.  If $d = 1$, then
    6.1  Test $p$ for primality via A.15.2 (and A.15.4 if desired).
    6.2  If $p$ is prime, then output $p$ and stop.
7.  Go to step 3.
```

NOTE—If r and p are to have roughly the same bit length, it is possible that a prime will not be found. More precisely, the expected number of primes for a given r is at most $(p_{\max} - p_{\min}) / (r \log p_{\max})$. If this quantity is very small, one should give up after a few iterations and restart the algorithm with a new value of r.

A.15.8 Strong primes

A large prime p is called *strong* if

— $p \equiv 1 \pmod{r}$ for some large prime r

— $p \equiv -1 \pmod{s}$ for some large prime s

— $r \equiv 1 \pmod{t}$ for some large prime t

The following algorithm efficiently produces a strong prime p, such that $p - 1$ is relatively prime to the specified odd positive integer z. (If the latter condition is not wanted, the choice $z = 1$ leads to a vacuous condition.)

Input: A lower bound $p_{\min}$ for p; an upper bound $p_{\max}$ for p; the desired bit lengths $L(r)$, $L(s)$, and $L(t)$; an odd positive integer z

Output: A random strong prime p from the interval $p_{\min} \le p \le p_{\max}$, where the primes r, s, and t have lengths $L(r)$, $L(s)$, and $L(t)$, respectively, where $p - 1$ is relatively prime to z

1. Generate a random prime t from the interval $2^{L(t)-1} \le t \le 2^{L(t)} - 1$ using A.15.6 with $f = 1$.
2. Generate a random prime r from the interval $2^{L(r)-1} \le r \le 2^{L(r)} - 1$ satisfying $r \equiv 1 \pmod{t}$, using A.15.7 with $f = 1$.
3. Generate a random prime s from the interval $2^{L(s)-1} \le s \le 2^{L(s)} - 1$ using A.15.6 with $f = 1$.
4. Compute $u := 1/s \bmod r$ via A.2.2.
5. Compute $v := 1/r \bmod s$ via A.2.2.
6. Compute $a \leftarrow su - rv \bmod rs$.
7. Generate a random prime p from the interval $p_{\min} \le p \le p_{\max}$ satisfying $p \equiv a \pmod{rs}$ using A.15.7 with $f = z$.
8. Output p.

If the strong prime p is also to satisfy an additional congruence condition $p \equiv h \pmod{m}$, then the last two steps of the algorithm should be replaced by the following steps:

7. Compute $b := 1 / (rs) \bmod m$ via A.2.2.
8. Compute $k := 1 / m \bmod rs$ via A.2.2.
9. Compute $c \leftarrow akm + bhrs \bmod mrs$.
10. Generate a random prime p from the interval $p_{\min} \le p \le p_{\max}$ satisfying $p \equiv c \pmod{mrs}$ using A.15.7 with $f = z$.
11. Output p.

A.16 Generation and validation of parameters and keys

A.16.1 An algorithm for generating DL parameters (prime case)

Input: Lower and upper bounds $p_{\min}$ and $p_{\max}$ for the modulus p; lower and upper bounds $r_{\min}$ and $r_{\max}$ for the generator order r; whether or not the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC

Output: DL parameters $q = p$, r, and g, satisfying $p_{\min} \le p \le p_{\max}$ and $r_{\min} \le r \le r_{\max}$; the cofactor k, if desired

1. Generate a random prime r from the interval $r_{\min} \le r \le r_{\max}$ using A.15.6 with $f = 1$.
2. Generate a random prime p from the interval $p_{\min} \le p \le p_{\max}$ satisfying the condition $p \equiv 1 \pmod{r}$, using A.15.7 with $f = 1$.
3. Let $k = (p - 1) / r$.
4. If the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC, check if r divides k; if so, go to step 1.
5. Choose an integer h, not already chosen, satisfying $1 < h < p - 1$.
6. Compute

   $g := h^k \bmod p$

   via A.2.1.

```
7.  If $g = 1$ then go to step 5.
8.  Output $p$, $r$, $g$. If desired, also output $k$.
```

A.16.2 An algorithm for validating DL parameters (prime case)

Input: DL parameters $q = p$, r, and g; the cofactor k (optional); whether or not the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC

Output: “True” or “False”

```
1.  Check that $p$ is an odd integer and $p > 2$. Check primality of $p$ via A.15.3.
2.  Check that $r$ is an odd integer and $r > 2$. Check primality of $r$ via A.15.3.
3.  Check that $g$ is an integer such that $1 < g < p$.
4.  Check that $g^r \equiv 1 \pmod{p}$.
5.  If $k$ is supplied, check that $k$ is an integer such that $kr = p - 1$.
6.  If the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC, check that $r$ does not
    divide $k$. (If $k$ is not supplied, first set $k \leftarrow (p - 1)/r$.)
7.  Output “True” if all checks work, and “False” otherwise.
```

NOTE—A method for verifiably pseudo-random generation of DL parameters is provided in ANSI X9.30:1-1997 [B4], ANSI X9.42 [B8], and FIPS PUB 185 [B56]. (See D.4.1.4 for more information.)

A.16.3 An algorithm for generating DL parameters (binary case)

Input: Lower and upper bounds $r_{\min}$ and $r_{\max}$ for the generator order r; a list of possible field degrees $m_1, \ldots, m_l$; whether polynomial or normal basis is desired; whether or not the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC

Output: DL parameters $q = 2^m$, r, and g (and the cofactor k, if desired), satisfying $r_{\min} \le r \le r_{\max}$ and $m = m_i$ for some i, if such exist; otherwise, the message “no such parameters”

```
1.  For $i$ from 1 to $l$ do
    1.1  Set $m \leftarrow m_i$.
    1.2  If $m$ does not appear in Table A.1 in A.3.10, then go to step 1.5.
    1.3  Set $r$ to the value in Table A.1 that corresponds to $m$.
    1.4  If $r_{\min} \le r \le r_{\max}$, then go to step 4.
    1.5  Next $i$.
2.  For $i$ from 1 to $l$ do
    2.1  Set $m \leftarrow m_i$.
    2.2  Obtain the known primitive prime factors of $2^m - 1$ (see Note 1 below).
    2.3  Remove those factors $r$ from the list that are not between $r_{\min}$ and $r_{\max}$.
    2.4  If the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC, remove those
         prime factors $r$ from the list for which $r^2$ divides $2^m - 1$.
    2.5  If any factors remain, then let $r$ be one of them and go to step 4.
    2.6  Next $i$.
```

3. Output the message “no such parameters” and stop.
4. Set $k \leftarrow (2^m - 1)/r$.
5. Choose a polynomial or normal basis for the field $GF\ (2^m)$ (see Note 2 below).
6. Choose an element h of $GF\ (2^m)$ not already chosen.
7. Compute $g := h^k$ in $GF\ (2^m)$ via A.4.3.
8. If $g = 1$, then go to step 6.
9. Output 2^m, r, g, and k.

NOTES

1—See A.3.9 for the definition of a primitive prime factor of $2^m - 1$.

The Cunningham tables are published collections of factors of $2^m - 1$ for m up to 1200, even m up to 2400, and $m \equiv 4$ (mod 8) up to 4800. They are available in print (see Brillhart et al. [B31]); an up-to-date version is available via FTP from Oxford University at ftp://sable.ox.ac.uk/pub/math/cunningham/.

If $m \leq 1200$ is odd, then the primitive prime factors of $2^m - 1$ are listed in Cunningham Table 2–, in the entry $n = m$. If $m \leq 2400$ is even, but not divisible by four, then the primitive prime factors of $2^m - 1$ are listed in Cunningham Table 2+ (Odd), in the entry $n = m/2$. If $m \leq 4800$ is divisible by four, but not by eight, then the primitive prime factors of $2^m - 1$ are listed in Cunningham Table 2LM, in the two entries for $n = m/2$. If $m \leq 2400$ is divisible by eight, then the primitive prime factors of $2^m - 1$ are listed in Cunningham Table 2+ (4k), in the entry $n = m/2$.

In the FTP version, the last three tables are combined into one, called 2+; but the entries are listed in the same notation.

2—A polynomial basis can be obtained from the basis table given in A.8.1 if $m \leq 1000$. Alternatively, irreducible polynomials over $GF\ (2)$ can be obtained using the methods given in A.8.2 through A.8.5.

If a normal basis is desired, and if $m \leq 1000$ is not divisible by eight, then a Gaussian normal basis can be obtained from the basis table given in A.8.1. Alternatively, normal polynomials over $GF\ (2)$ can be obtained via A.6.2.

A.16.4 An algorithm for validating DL parameters (binary case)

Input: DL parameters $q = 2^m$, r, and g; the cofactor k (optional); the representation for the elements of $GF\ (2^m)$; whether or not the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC

Output: “True” or “False”

1. If the representation for the field elements is given by a polynomial basis, check that m is a positive integer and that the polynomial is of degree m. Check that it is irreducible via A.5.5.
2. If the representation for the field elements is given by a Gaussian normal basis of type T, check that m is a positive integer not divisible by eight and that T is a positive integer; check that such a basis exists via A.3.6.
3. If the representation for the field elements is given by a general normal basis, check that m is a positive integer and that the polynomial is of degree m. Check that it is irreducible via A.5.5 and that it is normal via A.6.1.
4. Check that r is an odd integer and $r > 2$. Check primality of r via A.15.3.
5. Check via A.2.7 that the order of 2 modulo r is m [otherwise, it will be less than m, which means that r is not a primitive factor of $2^m - 1$ (see A.16.3 and A.3.9)].
6. Check that g is an element of $GF\ (2^m)$ and that $g \neq 0$ and $g \neq 1$ in $GF\ (2^m)$.
7. Check that $g^r = 1$ in $GF\ (2^m)$.
8. If k is supplied, check that k is an integer such that $kr = 2^m - 1$.

9. If the parameters will be used for DLSVDP-DHC or DLSVDP-MQVC, check that r does not divide k. (If k is not supplied, first set $k \leftarrow (2^m - 1)/r$.)
10. Output "True" if the checks work, and "False" otherwise.

A.16.5 An algorithm for generating DL keys

Input: Valid DL parameters q, r, and g

Output: A public key w; a private key s

1. Generate an integer s in the range $0 < s < r$ designed to be hard for an adversary to guess (e.g., a random integer).
2. Compute w by exponentiation: $w := g^s$ in $GF\ (q)$.
3. Output w and s.

A.16.6 Algorithms for validating DL public keys

The following algorithm verifies if a DL public key is valid:

Input: Valid DL parameters q, r, and g; the public key w

Output: "True" or "False"

1. If $q = p$ is an odd prime, check that w is an integer such that $1 < w < p$.
2. If $q = 2^m$ is a power of two, check that w is an element of $GF\ (2^m)$ and that $w \neq 0$ and $w \neq 1$ in $GF\ (2^m)$.
3. Check that $w^r = 1$ in $GF\ (q)$.
4. Output "True" if the checks work, and "False" otherwise.

The following algorithm does not verify the validity of a DL public key, but merely checks if it is in the multiplicative group of $GF\ (q)$. It may be used in conjunction with DLSVDP-DHC and DLSVDP-MQVC primitives.

Input: Valid DL parameters q, r, and g; the public key w

Output: "True" or "False"

1. If $q = p$ is an odd prime, check that w is an integer such that $0 < w < p$.
2. If $q = 2^m$ is a power of two, check that w is an element of $GF\ (2^m)$ and that $w \neq 0$ in $GF\ (2^m)$.
3. Output "True" if the checks work, and "False" otherwise.

A.16.7 An algorithm for generating EC parameters

Input: A field size q (where q is an odd prime p or 2^m); lower and upper bounds $r_{\min}$ and $r_{\max}$ for the base point order r; a trial division bound $l_{\max}$; the representation for the elements of $GF\ (2^m)$ if $q = 2^m$ [see Note 2 in A.16.3 for information on methods for choosing a basis for $GF\ (2^m)$]; whether or not the parameters will be used for key validation, ECSVDP-DHC or ECSVDP-MQVC

Output: EC parameters q, a, b, r, and G; the cofactor k if desired

1. Use the techniques of A.14 to find a positive integer k, a prime r in the interval $r_{\min} \le r \le r_{\max}$, a curve E over $GF\,(q)$ of order kr, and a point G of order r.
2. If q is prime and $q = r$, then the curve is anomalous and subject to the reduction attack described in Semaev [B134], Smart [B140], and Satoh and Araki [B130]. Go to step 1.
3. If the parameters will be used for key validation, ECSVDP-DHC or ECSVDP-MQVC, check if r divides k. If so, go to step 1.
4. Check the MOV condition via A.12.1. If the output is "False," then go to step 1.
5. Output q, a, b, r, and G (and k if desired).

A.16.8 An algorithm for validating EC parameters

Input: EC parameters q (where q is an odd prime p or 2^m), a, b, r, and G; the cofactor k (optional); the representation for the elements of $GF\,(2^m)$ if $q = 2^m$; whether or not the parameters will be used for key validation, ECSVDP-DHC or ECSVDP-MQVC; whether or not the curve is verifiably pseudo-random and, if so, the parameters given in A.12.5 (if q odd) or A.12.7 (if q even)

Output: "True" if validation was possible; otherwise "False"

1. If k is not supplied
    - 1.1 If $r \le 4\sqrt{q}$ and the parameters will be used for key validation, ECSVDP-DHC or ECSVDP-MQVC, then output "False" and stop.
    - 1.2 Go to step 4.
2. If $r < \sqrt{q} + 1$ and the parameters will be used for key validation, ECSVDP-DHC or ECSVDP-MQVC, then verify that r does not divide k. If this is **not** the case (i.e., if r divides k) then output "False" and stop.
3. Check that the cofactor k is a positive integer. Verify k via A.12.3. If the result is "cofactor rejected," then output "False" and stop.
4. Check that r is an odd integer and that $r > 2$. Check primality of r via A.15.3.
5. If $q = p$, then
    - 5.1 Check that p is an odd integer and $p > 2$. Check primality of p via A.15.3.
    - 5.2 Check that a and b are integers such that $0 \le a < p$ and $0 \le b < p$.
    - 5.3 If the curve is verifiably pseudo-random, then check this via A.12.5.
    - 5.4 Check that $4a^3 + 27b^2 \bmod p$ is not 0.
    - 5.5 Check that $G \ne \mathcal{O}$; let $G = (x, y)$.
    - 5.6 Check that x and y are integers such that $0 \le x < p$ and $0 \le y < p$.
    - 5.7 Check that $y^2 \equiv x^3 + ax + b \pmod{p}$.
    - 5.8 Check that $rG = \mathcal{O}$.
    - 5.9 Check that the curve is not an instance of one of the following excluded cases:
        - 5.9.1 If the output of the algorithm given in A.12.1 is "False," then the curve is excluded because it is subject to the MOV reduction attack described in Menezes, Okamoto, and Vanstone [B111].

5.9.2 If $r = p$, then the curve is excluded because it is subject to the reduction attack described in Semaev [B134], Smart [B140], and Satoh and Araki [B130] (see D.4.2 for more information).

6. If $q = 2^m$ then

6.1 Check the representation for the field elements, as follows:

6.1.1 If the representation for the field elements is given by a polynomial basis, check that m is a positive integer and that the polynomial is of degree m. If the polynomial is not listed in Table A.2, then check that it is irreducible via A.5.5.

6.1.2 If the representation for the field elements is given by a Gaussian normal basis of type T, check that m is a positive integer not divisible by eight and that T is a positive integer. If the value for T does not appear in the entry for m in Table A.2, then check that such a basis exists via A.3.6.

6.1.3 If the representation for the field elements is given by a general normal basis, check that m is a positive integer and that the polynomial is of degree m. Check that it is irreducible via A.5.5 and that it is normal via A.6.1.

6.2 Check that a and b are elements of $GF(2^m)$.

6.3 If the curve is verifiably pseudo-random, then check this via A.12.7.

6.4 Check that $b \neq 0$ in $GF(2^m)$.

6.5 Check that $G \neq \mathcal{O}$; let $G = (x, y)$.

6.6 Check that x and y are elements of $GF(2^m)$.

6.7 Check that $y^2 + xy = x^3 + ax^2 + b$ in $GF(2^m)$.

6.8 Check that $rG = \mathcal{O}$.

6.9 Check that the curve is not an instance of the following excluded case:

6.9.1 If the output of the algorithm given in A.12.1 is "False," then the curve is excluded because it is subject to the MOV reduction attack described in Menezes, Okamoto, and Vanstone [B111].

7. Output "True" if the checks given in steps 4 through 6 work, and "False" otherwise.

NOTE—The condition $r > 4\sqrt{q}$ is the usual practice and is required by the ISO and ANSI standards dealing with EC parameter validation. Thus, in the typical environment, it will always be the case that $r > 4\sqrt{q}$. For an environment in which this is known to be the case, step 1.1 and step 2 may be omitted.

A.16.9 An algorithm for generating EC keys

Input: Valid EC parameters q, a, b, r, and G

Output: A public key W; a private key s

1. Generate an integer s in the range $0 < s < r$ designed to be hard for an adversary to guess (e.g., a random integer).
2. Compute the point W by scalar multiplication: $W = sG$.
3. Output W and s.

A.16.10 Algorithms for validating EC public keys

The following algorithm verifies if an EC public key is valid.

Input: Valid EC parameters q, a, b, r, G, and k such that r does not divide k; a public key W

Output: “True” or “False”

1. Check that $W \neq O$. Let $W = (x, y)$.
2. If $q = p$, check that x and y are integers, such that $0 \leq x < p$ and $0 \leq y < p$ and $y^2 \equiv x^3 + ax + b$ (mod p).
3. If $q = 2^m$, check that x and y are elements of $GF(2^m)$ and that $y^2 + xy = x^3 + ax^2 + b$ in $GF(2^m)$.
4. Check that $rW = O$.
5. Output “True” if the checks work, and “False” otherwise.

The following algorithm does not verify the validity of an EC public key, but merely checks if it is a nonidentity point on the elliptic curve specified by the parameters. It may be used in conjunction with ECSVDP-DHC and ECSVDP-MQVC primitives.

Input: Valid EC parameters q, a, b, r, and G; a public key W.

Output: “True” or “False.”

1. Check that $W \neq O$. Let $W = (x, y)$.
2. If $q = p$, check that x and y are integers, such that $0 \leq x < p$ and $0 \leq y < p$ and $y^2 \equiv x^3 + ax + b$ (mod p).
3. If $q = 2^m$, check that x and y are elements of $GF(2^m)$ and that $y^2 + xy = x^3 + ax^2 + b$ in $GF(2^m)$.
4. Output “True” if the checks work, and “False” otherwise.

A.16.11 An algorithm for generating RSA keys

An *RSA modulus* is a product n of two (large) primes p and q. Given a public verification (or encryption) exponent e, the following algorithm efficiently produces such an RSA modulus, along with the secret signing (or decryption) exponent d.

Common choices for e include the Fermat primes 3, 5, 17, 257, and 65537, because these choices lead to particularly efficient exponentiations using the binary method of A.2.1. If a pseudo-random value of e is desired, it should be generated independently of p and q.

The primes produced may be randomly generated, or they may be strong primes (see A.15.8). A large, randomly generated prime is virtually certain to be strong enough in practice.

Input: The desired bit length L for the modulus; an odd public exponent $e > 1$

Output: An RSA modulus n of bit length L; the secret exponent d

1. Generate a prime p from the interval

   $2^{M-1} \leq p \leq 2^M - 1$

where $M = \lfloor (L + 1)/2 \rfloor$, using A.15.6 with $f = e$ (for random primes) or A.15.8 with $z = e$ (for strong primes).

2. Generate a prime q from the interval

$$\left\lfloor \frac{2^{L-1}}{p} + 1 \right\rfloor \le q \le \left\lfloor \frac{2^{L}}{p} \right\rfloor$$

using A.15.6 with $f = e$ (for random primes) or A.15.8 with $z = e$ (for strong primes).

3. Set $n := pq$.
4. Compute $l :=$ LCM $(p - 1, q - 1)$ via A.1.1 and A.2.2.
5. Compute $d := e^{-1} \bmod l$ via A.2.2.
6. Output n and d.

A.16.12 An algorithm for generating RW keys

A *Rabin-Williams (RW) modulus* is a product n of two (large) primes $p \equiv 3 \pmod 8$ and $q \equiv 7 \pmod 8$. Given a public verification exponent e, the following algorithm efficiently produces such an RW modulus, along with the secret signing exponent d.

The usual choice for the verification exponent is $e = 2$, because this choice means that signature verification can be implemented via a modular squaring rather than a modular exponentiation. If a pseudo-random value of e is desired, it should be generated independently of p and q.

The primes produced may be randomly generated, or they may be strong primes (see A.15.8). A large, randomly generated prime is virtually certain to be strong enough in practice.

Input: The desired bit length L for the modulus; an even public exponent e

Output: An RW modulus n of bit length L; the secret exponent d

1. Let $x > 0$ be the odd integer usch that $e = 2^k x$ for some integer $k > 0$.
2. Generate a prime $p \equiv 3 \pmod 8$ from the interval

   $$2^{M-1} \le p \le 2^M - 1$$

   where $M = \lfloor (L + 1)/2 \rfloor$, using A.15.7 with $f = x$ (for random primes) or A.15.8 with $z = x$ (for strong primes).

3. Generate a prime $q \equiv 7 \pmod 8$ from the interval

   $$\left\lfloor \frac{2^{L-1}}{p} + 1 \right\rfloor \le q \le \left\lfloor \frac{2^{L}}{p} \right\rfloor$$

   using A.15.7 with $f = x$ (for random primes) or A.15.8 with $z = x$ (for strong primes).

4. Set $n := pq$.
5. Compute $l :=$ LCM $(p - 1, q - 1) / 2$ via A.1.1 and A.2.2.
6. Compute $d := e^{-1} \bmod l$ via A.2.2.
7. Output n and d.

Annex B

(normative)

Conformance

The purpose of this annex is to provide implementers with a consistent language for claiming conformance with parts of this standard. Note, however, that this annex does not provide the means for verifying that a particular implementation indeed operates as claimed (this is sometimes called *implementation validation*). Therefore, conformance claims made by an implementation are merely claims, unless their accuracy can be assured by other means. Such other means may include, for example, implementation validation or assignment of legal liability to the implementer claiming conformance. They are outside the scope of this standard.

Note also that conformance for the purposes of this standard is a matter of functional correctness, not secure implementation; for the latter, implementers should refer to the security considerations in Annex D.

An implementation may claim conformance with one or more primitives, schemes, or scheme operations specified in this standard, as further described in this annex.

An implementation shall not claim conformance with this standard as a whole.

Refer to Clause 4 for background on primitives and schemes. Specific primitives and schemes are defined in Clause 6 through Clause 10.

B.1 General model

A claim of conformance is an assertion by an implementation that it operates in accordance with some specification, over some set of inputs. Thus, a claim of conformance has, fundamentally, two parts:

- The specification with which conformance is claimed
- A set of inputs, or *conformance region*, for which the specification is defined, and over which conformance is claimed

For the purposes of this standard, the specification may be that of a primitive, a scheme operation, or a scheme. (An implementation may claim conformance with a scheme by claiming conformance with each operation in the scheme.) For a primitive, the inputs are those stated in the specification; for a scheme operation, the term "input" refers both to initial inputs, such as messages, and inputs obtained during a step of the operation, such as domain parameters, keys, and key derivation parameters. Recommended conformance regions are given in the specifications.

The set of inputs for which a specification is defined depends on the particular primitive or scheme. For a primitive, the set consists of all inputs that satisfy the assumptions stated for the primitive. For a scheme operation, the set includes at least those inputs that satisfy the assumptions for any primitives invoked by the operation. If the operation includes key validation or domain parameter validation, then the specification may also be defined for certain inputs that do not satisfy the assumptions for a primitive invoked by the scheme. Thus, for example, the specification of a scheme operation may be defined for invalid, as well as valid, keys when key validation is included in the scheme, even though the specification of a primitive invoked by the scheme is not defined for invalid keys. This is because the behavior of the scheme with key validation is defined as follows on invalid keys: the keys are rejected.

The minimum behavioral requirements for claiming conformance over a conformance region are as follows:

a) On all inputs in the conformance region, the implementation shall perform steps identical or equivalent to those specified.

b) On all other inputs it accepts, the behavior of the implementation shall not interfere with correct operation on inputs in the conformance region. The behavior is otherwise unconstrained.

Acceptable behaviors in item b) include the following:

- Operating in accordance with the specification (if the specification is defined for the input)
- Rejecting the input
- Performing steps similar to those specified
- Performing some other noninterfering operation.

Since primitives are intended for low-level software or hardware implementation, it may be inconvenient for an implementation of a primitive to check whether an input is supported. Consequently, while an implementation of a primitive may reject some unsupported inputs, it is not expected that an implementation of a primitive will reject every unsupported input. Primitives are not intended to provide security apart from schemes, so such checking is appropriately deferred to the schemes. It is expected that an implementation of a scheme will reject many or even all unsupported inputs, depending on whether key and domain parameter validation is included. For more discussion on the risks of not rejecting unsupported inputs, see D.3.3.

An implementation may claim conformance over more than one conformance region, or with more than one specification.

NOTES

1—In the interest of interoperability, a conformance region should be sufficiently broad to support a range of possible applications. It is expected that implementation profiles for various applications will give minimum interoperability criteria, in terms of specifications and associated conformance region constraints. For a similar reason, a conformance region should be documented explicitly. (In some cases, however, the documentation may be implicit to some extent; for instance, the domain parameters may be unambiguously specified, but secret.)

2—Although an implementation's behavior is unconstrained on inputs outside the conformance region (except for not interfering with the behavior on inputs in the conformance region), it is recommended, in the interest of robustness, that an implementation include checks that prevent failure when specified assumptions are not satisfied. For instance, an implementation should include checks that prevent division by zero, or infinite loops, even if those checks are not necessary when the specified assumptions are satisfied.

3—The concept of "equivalence" (as in "perform steps... equivalent to") should be understood in the sense of indistinguishability. A conformant implementation of a scheme or primitive may perform steps identical to those specified for the scheme or primitive, in the sense of performing those steps exactly as specified; or it may perform similar steps that produce the same observable behavior.

For instance, if a step calls for generating a random number, the implementation may generate a pseudo-random number. Under the usual cryptographic assumption that the pseudo-random generator is indistinguishable from a truly random generator, the implementation is equivalent to the specification at that step.

Similarly, an implementation may choose to apply restrictions that exclude certain rare events. For instance, an implementation may exclude DL or EC private keys that are equal to one, and instead generate private keys in the range $[2, r-1]$. An implementation with such a restriction will be indistinguishable from the specification, and may still claim conformance. On the other hand, an implementation that generates private keys in the range [1,1000] could not claim conformance, because its behavior would be observably different from the specification.

As another example, an implementation of IFES-RSA might output an error message when the output of the encryption primitive equals its input, which is a rare event.

B.2 Conformance requirements

An implementation claiming conformance with a primitive or scheme specified in this standard shall meet the requirements specified in the subclauses that are specified in Table B.1 and Table B.2, in addition to the general criteria in B.1. Requirements are to be understood in the context of Clause 2, Clause 3, Clause 4, and Clause 5.

Table B.1—Required subclauses for conformance with primitives

Primitive	Subclauses
DLSVDP-DH	4.2, 6.1, 6.2.1
DLSVDP-DHC	4.2, 6.1, 6.2.2
DLSVDP-MQV	4.2, 6.1, 6.2.3
DLSVDP-MQVC	4.2, 6.1, 6.2.4
DLSP-NR	4.2, 6.1, 6.2.5
DLVP-NR	4.2, 6.1, 6.2.6
DLSP-DSA	4.2, 6.1, 6.2.7
DLVP-DSA	4.2, 6.1, 6.2.8
ECSVDP-DH	4.2, 7.1, 7.2.1
ECSVDP-DHC	4.2, 7.1, 7.2.2
ECSVDP-MQV	4.2, 7.1, 7.2.3
ECSVDP-MQVC	4.2, 7.1, 7.2.4
ECSP-NR	4.2, 7.1, 7.2.5
ECVP-NR	4.2, 7.1, 7.2.6
ECSP-DSA	4.2, 7.1, 7.2.7
ECVP-DSA	4.2, 7.1, 7.2.8
IFEP-RSA	4.2, 8.1, 8.2.2
IFDP-RSA	4.2, 8.1, 8.2.1, 8.2.3
IFSP-RSA1	4.2, 8.1, 8.2.1, 8.2.4
IFVP-RSA1	4.2, 8.1, 8.2.5
IFSP-RSA2	4.2, 8.1, 8.2.1, 8.2.6
IFVP-RSA2	4.2, 8.1, 8.2.7
IFSP-RW	4.2, 8.1, 8.2.1, 8.2.8
IFVP-RW	4.2, 8.1, 8.2.9

Table B.2—Required subclauses for conformance with schemes

Scheme	Operation	Subclauses
DL/ECKAS-DH1	Key agreement	4.3, 9.1, 9.2
DL/ECKAS-DH2	Key agreement	4.3, 9.1, 9.3
DL/ECKAS-MQV	Key agreement	4.3, 9.1, 9.4
DL/ECSSA	Signature generation	4.3, 10.1, 10.2.1, 10.2.2
	Signature verification	4.3, 10.1, 10.2.1, 10.2.3
IFSSA	Signature generation	4.3, 10.1, 10.3.1, 10.3.2
	Signature verification	4.3, 10.1, 10.3.1, 10.3.3
IFES	Encryption	4.3, 11.1, 11.2.1, 11.2.2
	Decryption	4.3, 11.1, 11.2.1, 11.2.3

An implementation may claim conformance with a primitive, a scheme, or a scheme operation. For a scheme or scheme operation, conformance requirements for the selected primitive or primitives, and for additional techniques such as encoding methods or key derivation functions, are also assumed. When documenting conformance with a scheme, these scheme options shall be noted explicitly. In addition, the documentation shall indicate whether the implementation includes key validation or domain parameter validation and, if so, what is validated (i.e., what properties of keys and parameters are assured by the validation). An implementation claiming conformance with a scheme shall satisfy the requirements for each operation in the scheme.

The following is a template for a claim of conformance:

> "Conforms with IEEE Std 1363-2000 (technique/options) over the region where (constraints on inputs)."

The "technique/options" component identifies the primitive, scheme, or scheme operation, together with any underlying techniques, such as the encoding method or hash function, and any additional choices, such as whether and how domain parameter or key validation is performed. The "constraints on inputs" component identifies the conformance region. The method of expressing these components is left to the implementation. Some examples are given in B.3.

B.3 Examples

This clause gives some examples of claims of conformance with the primitives and scheme operations in this standard.

B.3.1 DLSP-DSA

A hardware cryptographic module claims conformance with DLSP-DSA. The two parts of its conformance claim are as follows:

— **Specification:** DLSP-DSA, as given in 6.2.7

— **Conformance region**: Inputs of the form

The DL domain parameters q, r, and g associated with the key s

The signer's private key s

The message representative, which is an integer $f \geq 0$

where the DL domain parameters and the private key are valid and associated, subject to additional conditions that constrain the conformance region.

The following is an example of additional conditions (this follows the recommendations in 6.2.7):

— The DL field order q is a 512 bit to 1024 bit prime.

— The DL subgroup order r is a 160 bit prime.

— The message representative f is, at most, 160 bits long.

A module claiming conformance under these conditions may document its conformance as follows:

> *Conforms with IEEE Std 1363-2000 DLSP-DSA over the region where the DL field order q is a 512 bit to 1024 bit prime, the DL subgroup order r is a 160 bit prime, the domain parameters and the private key are valid and associated, and the message representative f is at most 160 bits long.*

Such a module may also claim conformance over a subset of the region just stated. For instance, it may claim conformance with the region specified in the Digital Signature Standard (ANSI X9.30:1-1997 [B4] or FIPS PUB 186 [B56]), where the DL field order q is a 512 bit, 576 bit, …, or 1024 bit prime, and the subgroup order r and message representative f are as already stated.

B.3.2 DLSSA signature verification

A software application claims conformance with the DLSSA signature verification operation. The two parts of its conformance claim are as follows:

— **Specification:** DLSSA signature verification, as given in 10.2.3, with a particular signature verification primitive and encoding method, and optionally with particular domain parameter and key validation methods

— **Conformance region:** Inputs of the form

The DL domain parameters q, r, and g associated with the key w
The signer's purported public key w
The message M
The purported signature (c, d)

subject to additional conditions that constrain the conformance region, some of which may depend on the specification.

Examples of the particulars for DLSSA signature verification are as follows:

— Signature verification primitive: DLSP-DSA

— Encoding method: EMSA1, with SHA-1 hash function

— No domain parameter or key validation

With this specification, the behavior is undefined for invalid domain parameters and keys. An implementation may thus claim conformance over only a conformance region consisting of valid domain parameters and keys. Examples of the conditions that constrain the conformance region in this case are as follows:

— Domain parameters and public key are valid and associated.

— The DL field order q is a 512 bit to 1024 bit prime.

— The DL subgroup order r is a 160 bit prime.

— Message M is any that can be input to the implementation, at most 100 Mbytes long.

— The purported signature (c, d) is any that can be input to the implementation, including at least all those such that c and d are in the range [1, r – 1].

A module claiming conformance under these conditions may document its conformance as follows (with shorthand for the specification):

> *Conforms with IEEE Std 1363-2000 DLSSA/DLVP-DSA/EMSA1/SHA-1 signature verification operation with no explicit domain parameter or key validation over the region where the DL field order q is a 512 bit to 1024 bit prime, the DL subgroup order r is a 160 bit prime, the domain parameters and public key are valid and associated, the message M is at most 100 Mbytes long, and the purported signature (c, d) is any that can be input to the implementation, including at least all those such that c and d are in the range [1, r - 1].*

Other examples of the particulars are as follows:

- Signature verification primitive: DLSP-DSA
- Encoding method: EMSA1, with SHA-1 hash function
- "Canonical seeded hash" domain parameter validation ANSI X9.30:1-1997 [B4], and key validation (A.16.6).

With this specification, the behavior is defined as follows for invalid domain parameters and public keys: they are rejected. (Indeed, domain parameters that are otherwise valid according to the definitions in this standard, but for which the seed is incorrect, are also rejected.) An implementation may thus claim conformance over a conformance region consisting of valid and invalid domain parameters and keys. The same example conditions as given above may be followed here, except for the condition that the domain parameters and public key are valid and associated.

An example conformance statement for this case is as follows:

> *Conforms with IEEE Std 1363-2000 DLSSA/DLVP-DSA/EMSA1/SHA-1 signature verification operation with "canonical seeded hash" domain parameter validation and key validation over the region where the DL field order q is a 512 bit to 1024 bit prime, the DL subgroup order r is a 160 bit prime, the message M is at most 100 Mbytes long, and the purported signature (c, d) is any that can be input to the implementation, including at least all those such that c and d are in the range [1, r – 1].*

B.3.3 IFSP-RSA2

A hardware cryptographic module claims conformance with IFSP-RSA2. The two parts of its conformance claim are as follows:

- **Specification:** IFSP-RSA2, as given in 8.2.6
- **Conformance region:** Inputs of the form
  - The signer's RSA private key K
  - The message representative, which is an integer f such that $0 \leq f < n$

where the private key K is valid and such that $f \equiv 12 \pmod{16}$, subject to additional conditions that constrain the conformance region.

Examples of additional conditions are as follows (this follows the recommendations in 8.2.6):

- Size of the modulus n in the private key is 512 to 2048 bits.
- Message representative f is in the range $[0, n – 1]$.

A module claiming conformance under these conditions may document its conformance as follows:

> *Conforms with IEEE Std 1363-2000 IFSP-RSA2 over the region where the private key K is valid and the size of the modulus n is 512 to 2048 bits, and the message representative f ≡ 12 (mod 16) is in the range [0, n – 1].*

The module may also claim conformance over a subset of the region just stated. For instance, it may claim conformance with the region where the modulus size is 1024 to 2048 bits.

B.3.4 IFSSA signature verification

A software application claims conformance with the IFSSA signature verification operation. The two parts of its conformance claim are as follows:

— **Specification:** IFSSA signature verification, as given in 10.3.3, with a particular signature verification primitive and encoding method and, optionally, with a particular key validation method

— **Conformance region:** Inputs of the form

The signer's purported public key (*n*, *e*)
The message *M*
The purported signature *s*

subject to additional conditions that constrain the conformance region, some of which may depend on the specification.

Examples of the particulars for IFSSA signature verification are as follows:

— Signature verification primitive: IFVP-RSA2

— Encoding method: EMSA2, with SHA-1 hash function

— No domain parameter or key validation

With this specification, the behavior is undefined for invalid keys. An implementation may thus claim conformance only over a conformance region consisting of valid keys. Examples of the conditions that constrain the conformance region in this case are as follows:

— Public key (*n*, *e*) is valid.

— Size of the modulus *n* in the public key is 512 to 2048 bits.

— Message *M* is any that can be input to the implementation, at most 100 Mbytes long.

— The purported signature *s* is any that can be input to the implementation, including at least all *s* in the range [0, (*n* – 1)/2].

The application may document its conformance as follows (with shorthand for the specification):

> *Conforms with IEEE Std 1363-2000 IFSSA / IFVP-RSA2 / EMSA2 / SHA-1 signature verification operation with no explicit key validation over the region where the public key (n, e) is valid, the size of the modulus n is 512 to 2048 bits, the message M is at most 100 Mbytes long, and the purported signature s is any that can be input to the implementation, including at least all s in the range [0, (n – 1)/2].*

Annex C

(informative)

Rationale

This annex is presented in the form of questions and answers.

C.1 General

C.1.1 Why are there three families of cryptographic techniques?

All three of the families (DL, EC, and IF) are established in the marketplace. They have different advantages and disadvantages, such as performance, key size, code size, availability of cryptoanalytic research, patent coverage, and use in other standards. The goal of this standard is not to restrict users to a single choice, but rather to provide a framework that would allow selection of methods appropriate for particular applications. Since implementers of this standard need not implement it in its entirety, they have the option to choose the techniques that best suit their needs.

C.1.2 Why are primitives and schemes separated?

In this standard, primitives are presented as mathematical operations, while schemes are presented in a general form based on certain primitives and additional techniques, such as encoding methods. Historically, primitives were discovered based on number-theoretic one-way functions. Encoding methods and other components of the scheme were added later to achieve particular security attributes, and tend to focus more on bit-string manipulation. The general form of the scheme lays down a good framework to allow alternative techniques to be included in a given scheme in future versions of the standard. An implementation can conform to either a primitive or a scheme (see Annex B). Hardware components, in particular, may find it advantageous to conform to a primitive, because primitives tend to be less flexible and change less than encoding methods and other components of a scheme. Note, however, that primitives alone are not intended to provide security (see Clause 4).

C.1.3 How were the decisions made regarding the inclusion of individual schemes?

Three types of schemes are defined in this standard: key agreement, digital signatures, and public-key encryption. In practice, these are the most basic yet important functions in a public-key cryptosystem. Other types of schemes (e.g., identification schemes) may be included in future versions of this standard.

The main purpose of this standard is to specify the basic public-key techniques in a common way. Additional techniques remain to be specified, which could eventually be added to the standard. However, many of those additional techniques require further development before being standardized, whereas the basic techniques in the current version are relatively more established. To facilitate the completion of the work on basic techniques, while also providing a forum for discussing additional techniques, the Working Group decided to have two separate projects: P1363 and P1363a. P1363a will supplement the more established techniques already covered in the P1363 document, and it is intended that the two documents will be merged in a future version.

The selection of schemes in the current version of the standard was based on the above decision. Some reasons for the selection of each individual scheme can be found in C.3.

C.1.4 Why are constraints on key sizes not specified?

Key sizes are chosen in accordance with security requirements and are application-dependent. As the focus of this standard is specifications of public-key techniques, not security certification, no constraints are given on the key size for any of the schemes. However, recommendations on key sizes for each family of techniques are included in Annex D.

C.1.5 Why are message-encoding methods for encryption and signature needed?

The mathematical structure of certain “raw” cryptographic algorithms can lead to a number of delicate attacks, such as chosen ciphertext attacks (where one obtains the decryption of a ciphertext by asking for the decryption of an apparently unrelated ciphertext), or chosen message attacks (where one obtains a signature of a message after requesting signatures for apparently unrelated messages). An appropriate message-encoding method provides a systematic way of thwarting all such attacks. Quite a few encoding methods for encryption and signature have appeared in academic literature and other standards. The methods included in this standard are well-established techniques. It is highly recommended to use these methods when implementing the schemes specified in this standard. More discussions on security properties of message-encoding methods are given in Annex D.

C.1.6 Why are key derivation functions needed?

A key derivation function is needed in the DL/EC key agreement schemes. One reason is that the output from a DL/EC secret value derivation primitive may not be appropriate to use directly as a session key, due to a bias in some of the bits of the derived secret value. A good key derivation function helps to remove such bias and to use all the entropy in the output of the secret value derivation primitive. Another reason is that, in many cases, it is useful to be able to derive more than one key from a secret value. In addition, key derivation functions, as defined in this standard, also take as input key derivation parameters, which allow one to specify additional information related to the derived key, as well as the parties in the key agreement scheme.

C.1.7 Why are data formats for input/output, keys, and domain parameters not normative?

Some data formats for input/output, keys, and domain parameters are specified in the informative Annex E. The choice of a particular data format does not affect security, as long as the choice is unambiguous and known to all parties involved, and provides for adequate and unambiguous representation of the relevant data. Two implementations with different choices of data formats may not be interoperable. They will need to convert between each other's formats to achieve interoperability. The data formats are not normative because some applications may not require interoperable formats.

C.2 Keys and domain parameters

C.2.1 Why have two types of fields for the DL and EC families?

Modular arithmetic is already implemented in many systems and may have performance advantages in software. On the other hand, arithmetic in finite fields of characteristic two may have performance advantages in hardware. There are also security issues related to the two types of fields (see Annex D for more discussion). In addition, certain techniques in both cases are covered by patents, and a particular application will have to weigh these tradeoffs when deciding what type of field to use.

C.2.2 Why allow multiple representations for GF (2^m)?

While there is a single most commonly used representation for *GF* (*p*), there are multiple natural, commonly used representations for *GF* (2^m). There are performance tradeoffs in the use of each representation. When choosing a representation, one needs to consider whether hardware or software performance is more important, as well as time complexity, memory complexity, and relevant patents.

While it would have been possible to standardize one representation and require that applications that want to do computations in a different representation convert from one to the other, the expense of basis conversion may be unacceptable in certain applications. The aim instead was to provide a flexible framework within which applications will decide how to interoperate. Therefore, recommendations on two particular representations are provided. It is possible that other representations for *GF* (2^m), as well as *GF* (*p*), will be provided in P1363a.

C.3 Schemes

C.3.1 For the DL and EC families, why have three key agreement schemes (-DH1, -DH2, and -MQV)?

The schemes DL/ECKAS-DH1 are based on the traditional Diffie-Hellman key agreement schemes, in which each party contributes one key pair. The scheme DL/ECKAS-DH2 is based on the so-called “unified model” for key agreement, in which each party contributes two key pairs. The schemes DL/ECKAS-MQV also utilize two key pairs from each party, but are more computationally efficient than DL/ECKAS-DH2. Certain security attributes can be achieved in different ways by the schemes, as discussed in Annex D.

The reason all three schemes are in this standard is that there are tradeoffs in efficiency, depending on how the security attributes are achieved with each of the schemes. There may also be differences in patent coverage. When choosing a scheme, one needs to consider the desired security attributes, communication, time and memory complexity, and relevant patents.

C.3.2 For the DL and EC families, why have the “compatibility” option for the DHC and MQVC primitives?

The “compatibility” option provides flexibility to implementers. The “compatible” versions of DHC and MQVC are likely to achieve greater interoperability with existing DH and MQV implementations, but the “incompatible” versions may have some implementation advantages, as they may be easier to layer on top of existing DH and MQV implementations. (In particular, it may be convenient in some architectures to view the cofactor multiplication as an additional step applied after ordinary DH and MQV.) Concerns about interoperability can be addressed in profiles and other standards based on this standard.

C.3.3 For the EC and DL families, why have both DSA and NR signature schemes with appendix?

The scheme DLSSA-DSA is a generalization of the U.S. Government Digital Signature Standard (DSS) specified in ANSI X9.30:1-1997 [B4] and FIPS PUB 186 [B56], and it has withstood extensive cryptanalysis. The scheme ECSSA-DSA is the elliptic curve analog of DLSSA-DSA, and a similar scheme called ECDSA is specified in ANSI X9.62-1998 [B11].

Note that ANSI X9.30:1-1997 [B4], FIPS PUB 186 [B56] and ANSI X9.62-1998 [B11] mandate the use of the hash function SHA-1 with DSS (and ECDSA), and they have restrictions on the sizes of the DL and EC parameters. However, this standard attempts to provide a more flexible specification by allowing other suitable hash functions and a larger range of key sizes. A particular application may find that its key sizes need to be shorter or longer than DSS allows, and that a different hash function better suits its needs. Implementations of DLSSA-DSA and ECSSA-DSA can be compliant with ANSI X9.30:1-1997 [B4], ANSI X9.62-1998 [B11], or FIPS PUB 186 [B56].

The schemes DL/ECSSA-NR have the advantages of better performance and smaller code size (in particular, because they do not require computation of a multiplicative inverse in a finite field). Also, DL/ECSP-NR and DL/ECVP-NR, the basic primitives for the schemes, allow for message recovery, whereas DL/ECSP-DSA and DL/ECVP-DSA do not. Should a signature scheme with message recovery be necessary for the DL or EC system, it can be constructed based on DL/ECSP-NR and DL/ECVP-NR, although such a scheme is not currently specified in the standard (see C.3.4).

C.3.4 For the DL and EC families, why are there no signature schemes with message recovery?

A signature scheme with message recovery requires the use of an encoding method which, in particular, adds redundancy to a message. The Working Group was not aware of an encoding method that would be acceptable for standardization for the DL and EC family. Should an application require a DL or EC signature scheme with message recovery, it may use an encoding method of its choice with the NR signature and verification primitives (see Annex B). It is also possible a suitable encoding method will be established during the P1363a effort (see C.1.3). Note that ISO/IEC 9796-4 [B79] (currently in draft stage) defines a signature scheme with message recovery based on the Nyberg-Rueppel primitive.

C.3.5 For the DL and EC families, why are there no encryption schemes?

A DL or EC encryption scheme, in general, requires the use of an encoding method that provides the security properties as described in D.5.3. In addition, since EC keys commonly used in practice are fairly short and, hence, can be shorter than the data to be encrypted, an encoding method should be designed to also handle this size difference. Several proposals on DL and EC encryption schemes were presented to the P1363 Working Group to address these issues. However, none of the proposals was considered mature enough, and it was decided that they would benefit from further study before being standardized. It is likely that a suitable encryption scheme for the DL and EC families will be established during the P1363a effort (see C.1.3). (Note that ANSI X9.63 [B12] defines encryption schemes for the EC family.)

C.3.6 For the IF family, why have both RSA and RW signature schemes?

Both RSA and RW are established signature schemes based on the integer factorization problem. The RSA signature schemes are well understood and widely used. The RW signature verification is generally faster than the RSA signature verification if the public exponent of two is used; however, the RSA signature generation has a code-size and speed advantage, because there is no need to compute the Jacobi symbol. Both RSA and RW signatures are described in the appendix to ISO/IEC 9796:1991 [B78].

Note that there are two signature and verification primitives (IFSP/IFVP) for RSA, namely RSA1 and RSA2. RSA1, which specifies just the “raw” RSA operation, has been widely used in various implementations and protocols, such as PKCS #1 ([B126] and [B127]), SET ([B104]), and S/MIME (Dusse et al. [B51] and Dusse et al. [B52]). RSA2 has an extra simple step to adjust the most significant bit of the signature to zero and, hence, can save one bit (and potentially one byte for some key sizes) compared with RSA1. RSA2 is a component in ANSI X9.31-1998 [B7] and ISO/IEC 9796: 1991 [B78].

C.3.7 For the IF family, why are there no signature schemes with message recovery?

Unlike the DL and EC cases, there is a well-established encoding method for the IF case of signatures with message recovery, specified in ISO/IEC 9796:1991 [B78]. However, work by Coron, Naccache, and Stern [B42] and Coppersmith, Halevi, and Jutla [B40] demonstrated practical chosen-message attacks against this method. The Working Group decided that the attacks warranted exclusion of this method from the standard. The Working Group was not aware of another encoding method that would have been ready for standardization for the IF family. Should an application require an IF signature scheme with message recovery, it may use an encoding method of its choice with the IF signature and verification primitives (see Annex B). It is also possible that a suitable encoding method will be established during the P1363a effort (see C.1.3). Note that ISO/IEC JTC1 SC27 (the subcommittee responsible for the ISO/IEC 9796 series of standards) is currently considering modifications to the encoding method of ISO/IEC 9796:1991 [B78], as well as other methods, which will be issued in the ISO/IEC 9796 series.

C.3.8 For the IF family, why are there no key agreement schemes?

In general, key agreement can be achieved using techniques based on integer factorization. For example, each party can transport a secret value using an IF encryption scheme to the other party, and a shared secret key can then be derived using the two secret values. However, such methods are better described as protocols rather than as schemes in the model given in Clause 4. This may be a semantic distinction between “protocol” and “scheme.” Under the model in Clause 4, a key agreement scheme is a collection of related operations by which parties can agree on one or more secret keys in some protocol. While one can use IF encryption and decryption operations to agree upon secret keys, those operations are more naturally described in an encryption scheme. It is possible that key agreement techniques based on the IF family will be specified during the P1363a effort.

Annex D

(informative)

Security considerations

D.1 Introduction

This annex addresses security considerations for the cryptographic techniques that are defined in this standard. It is not the intent of this annex to teach everything about security or cover all aspects of security for public-key cryptography. Rather, the goal of this annex is to provide guidelines for implementing the techniques specified in this standard. Moreover, since cryptography is a rapidly changing field, the information provided here is necessarily limited to the published state-of-the-art as of the time the standard was drafted in August 1998. Implementers should, therefore, review the information against more recent results at the time of implementation; the Working Group website may contain additional relevant information (see http://grouper.ieee.org/groups/1363/index.html).

Security recommendations (in the form of "should") are given throughout this annex. It should be understood, however, that there may be choices other than the ones recommended here that achieve a given level of security. Furthermore, as discussed in D.2, the definition of security depends on the types of attack that are relevant to an implementation. If an attack is not relevant, then some recommendations may not apply. Thus, while the recommendations given here enable security, they should not necessarily be taken as requirements. Nevertheless, it is expected that other standards based on this standard may upgrade some of the recommendations to requirements.

The considerations are presented in six parts. General security principles applying to all the families and schemes are presented in D.2. Key management considerations that also apply to all the families and schemes are summarized in D.3. Family-specific considerations are given in D.4, and scheme-specific considerations are given in D.5. As generation of random numbers is a common tool needed to implement this standard, random number generation considerations are summarized in D.6. Implementation considerations, relevant to all the families and schemes, are given in D.7.

For readers who are interested in extensive and in-depth discussions on security and cryptography, see Menezes, van Oorschot, and Vanstone [B112], Schneier [B132], Stallings [B143], and Stinson [B145] in Annex F.

D.2 General principles

The storage and transmission of data in modern computer and communications systems is vulnerable to a number of threats, including unauthorized disclosure and modification, as well as impersonation of parties in the system. Parties' assurances of where data is from, whether data is in its original form, who has access to data, and whether a party claiming a certain identity is authentic, all depend on appropriate protection of data.

Policy measures, such as requiring a password for access to a system; physical measures, such as concealing wires; and legal measures, such as prohibiting eavesdropping by statute, all provide a degree of defense against the threats just mentioned. But such approaches are all indirect, as they address only the implementation or use of a system—not the data itself. Direct protection, involving operations on the data, is the purpose of cryptography.

As noted in Clause 4, three common means of data protection provided by public-key cryptography are key agreement schemes, encryption schemes, and digital signature schemes. Implemented and combined appropriately, they can address the threats just mentioned. With either a key agreement scheme or an encryption scheme, for instance, parties can establish a shared secret key. The parties can then prevent unauthorized disclosure by applying a symmetric encryption algorithm to the data using the shared secret key, so that parties not possessing the key cannot recover the data. A party can also encrypt data directly with an encryption scheme. Similarly, with a signature scheme, parties can detect unauthorized modification as well as impersonation.

The need for the protection of data and identities against the threats mentioned above leads to consideration of the following question: "What is a secure system?" There is no easy answer to this question. As the *Handbook of Applied Cryptography* states, "… security manifests itself in many ways according to the situation and requirement" (see Section 1.2 of Menezes, van Oorschot, and Vanstone [B112]). The following paragraphs explore some of the general principles that help answer this question.

In a typical security analysis, an *opponent* (also called an *adversary*)—the source of the threats mentioned above—is assumed to have certain abilities. Under the usual model, called *Kerckhoffs' assumption,* Kerckhoffs [B92], an opponent will have full knowledge of system design, including the specification of all cryptographic operations (as, for instance, this standard would provide). Usually, the opponent will be able to use more tools than just "passive wiretapping" in order to read messages exchanged between parties. For example, the opponent may also be able to modify the message between parties. The opponent should also be considered to have some computational resources. The opponent may also be trusted by other parties to some extent. For instance, legitimate parties may be willing to perform selected cryptographic operations requested by the opponent, or the opponent's keys may be accepted as legitimate by other parties. In other words, the opponent may appear to other parties as a legitimate party. However, the opponent generally will not know all of the secrets in the system, such as long-term private keys, and it is this knowledge that distinguishes the opponent from the legitimate parties. Any attempt by the opponent to violate the security of a system in some manner is called an *attack*.

Under the assumption that such an opponent is present, the legitimate parties need to ensure that certain *security properties* (such as data confidentiality or data integrity) are satisfied. Just as there are different classes of opponents, there are different desirable security properties. A higher level of security is likely to be desirable for more valuable data, for instance, or for data with a long lifetime. Thus, as a general principle, security properties should be selected consistent with the value of the data being protected and the set of opponents envisioned. Since the cost of providing security generally varies according to the security properties, decisions related to security are appropriately framed as cost-benefit analyses.

In light of the above discussion, security may be defined as the *assurance of trust in the face of opponents with certain knowledge and resources*. Thus, the opponent, and the types of attack it can mount, is a required component of the definition of security.

A designer's objective, then, is to choose security techniques that provide certain desired security properties when faced with an opponent considered reasonable for a particular system. The cost-benefit analysis that can be applied to make decisions related to security can also be applied when evaluating the capabilities of a potential opponent. For example, an opponent is unlikely to spend more on breaking the system than it will obtain by doing so. However, it is important not to overestimate the opponent's costs, because an opponent may be using stolen, borrowed, or free resources (such as spare cycles of multiple computers). Indeed, in the case of free resources, the attack is always worth doing in a cost-benefit analysis, assuming the cost is really zero. Also, a large one-time investment for an opponent may pay off over time if the opponent breaks multiple systems. For example, if many DL-based systems use one particular finite field, a large investment in accelerated hardware modules for solving the DL problem in that specific field may pay off for the opponent, especially because the computation time per logarithm goes down if multiple discrete logarithms need to be computed. It is also important not to underestimate the opponent's benefit—it may be unclear how much publicity, personal satisfaction, or furthering of political goals may be worth to a potential opponent. Finally, one must also consider risk in terms of an opponent's probability of success. Even a chance of one in one million may be unacceptably large in some situations, so it is not only the cost of guaranteed success that is relevant.

In support of this process of choosing appropriate techniques when faced with a potential opponent, this annex summarizes the choices related to cryptographic techniques in this standard. Note, however, that this standard (and this annex in particular) focuses on individual implementation of individual cryptographic techniques, and not on entire systems. In particular, this standard defines cryptographic techniques from the point of view of a single party, rather than protocols in which multiple parties participate and multiple cryptographic operations are performed. It is expected that techniques in this standard will be combined into protocols by other standards or implementers. However, such combinations, especially ones where multiple cryptographic operations (e.g., both encryption and signature) on the same data are performed, require additional security analyses.

D.3 Key management considerations

This clause provides general information on key management. Proper key management is an essential component of cryptographic security. It is needed, in particular, to ensure integrity, authenticity and, where appropriate, secrecy of domain parameters and keys.

D.3.1 Generation of domain parameters and keys

A set of domain parameters may be generated by one of the parties that intend to use it, by a third party, or jointly by any subset of these. A capability to audit the domain parameter generation may be provided to other parties by generating random components of the domain parameters using one-way function(s) of random seed(s), and publishing the seed(s). Any party can then verify that the domain parameters indeed correspond to the seed(s). This provides some degree of assurance that the party generating the parameters did not pick them specifically to possess some particular special property. The property in question should be rare enough so that it is infeasible to obtain it simply by trying different seeds. The one-way function should be such that it is hard to find a seed that produces parameters with that property (e.g., the property should not be closely related to the one-way function). See A.12.4 through A.12.7 for examples of parameter generation that provide for auditing. Additional information for auditing domain parameter generation is provided by family in D.4.

A public/private key pair may be generated as follows:

- By the owner of the keys (this is the most commonly used method).
- Jointly by the owner of the keys and by another party, such as a certifying authority. Depending on the technique used, the other party may not need to be entrusted with any secrets by the owner of the key pair. This method may be used to ensure that the key pair is not picked by the owner to have some particular special property, and may eliminate the need for a separate verification that the party possesses its private key (see D.3.2). Chen and Williams [B36] provide an example of this method.
- By a third party that can be trusted with the private key (in particular, it should be trusted to keep the private key secret and not use its knowledge of the private key to its advantage). If the third party is trusted to perform the key generation properly, this method may be used to ensure that the key pair is not picked by the owner to have some particular special property, and may eliminate the need for a separate verification that the party possesses its private key (see D.3.2).

A capability to audit the key generation may be provided using methods similar to those for parameter generation, as described above. However, because (as opposed to parameters) the private key is meant to be a secret, the random seed should be kept secret (as securely as the private key), because revealing it would reveal the private key. The actual auditing, therefore, must be done only by trusted parties, or in situations when it is appropriate to reveal the private key.

Private keys should never be shared among parties. Every private key should be generated independently of other private keys. If random numbers for use in producing the private keys are generated properly, accidental sharing of private keys is extremely unlikely to occur (see D.6 for more on random number generation). Two or more dishonest parties, however, may collude in order to make their private keys the same (or make them stand in some relation known to the colluding parties). Dishonest parties may do so, for example, for the purposes of claiming the failure of a key generation procedure and for repudiating signatures produced with those keys (see D.5.2.3 for more on repudiation). Such concerns may be addressed by establishing trust in the implementation of key generation, or by involving a trusted party in the key generation process, as described above. It may be possible for an authority to check that no two public keys (and, hence, private keys) are equal in a limited system. However, this check will not reveal a pair of keys that stand in some special relation to each other, and it may be impractical to implement in some systems.

More considerations for key and parameter generation are given by family in D.4.

D.3.2 Authentication of ownership

Authentication of ownership of a public key is the assurance that a given, identified party intends to be associated with the public key (and the corresponding set of domain parameters, if any). It may also include assurance that the party possesses the corresponding private key; this is commonly known as Proof of Possession (POP). The latter assurance is stronger, in the sense that it prevents an opponent from misleading other parties into believing that the results of operations performed with the private key are associated with the opponent, rather than with the actual owner of the private key.

Further considerations related to authentication are given by type of scheme in D.5.

Authentication of ownership may be conveyed as part of the supporting key management (e.g., in a *certificate*, which is a message signed by a certifying authority indicating that a particular party, typically identified by a name, owns a given public key) (see [Section 13.4.2 of B112]). Anyone who knows the certifying authority's public key and trusts the certifying authority can verify the signature on the certificate and, thereby, authenticate the party's ownership of the public key. Certifying authorities may issue certificates to other certifying authorities, so that trust in a single "root" certifying authority's public key can extend through a chain of certificates to the public keys of a large community. Other attributes, such as key cryptoperiod or usage restrictions, may also be bound to a public key in this manner (see D.3.6 and Kaliski [B82]). To gain assurance that a party possesses a private key and, thereby, to convey this assurance, a certifying authority should perform an appropriate protocol verifying the possession before issuing the certificate. Such methods may be found in Adams and Myers [B1], (Sections 10.3.3, 10.4, and 13.4.2 of Menezes, van Oorschot, and Vanstone [B112], and Solo et al. [B141]).

Another way of conveying authentication of ownership is for a party to issue its own certificate (i.e., to sign the message containing the public key with a separate private key). This approach is helpful when the public key has a short cryptoperiod and it is impractical to obtain a certificate from a certifying authority. However, this approach does not necessarily provide assurance that the party possesses the corresponding private key.

A public key should be securely associated with its set of domain parameters, if any. This may be accomplished by including the set of domain parameters in the certificate. The set of domain parameters may also be embedded into the system if, for example, the system implementation only performs operations with a single set of domain parameters. Domain parameters should be authenticated and protected from unauthorized modification in the same manner as a public key.

In general, a party's possession of a private key should be authenticated as part of authenticating the party's association with a public key. Situations in which it may be acceptable not to do so are given by type of scheme in D.5.

D.3.3 Validation of domain parameters and keys

Most primitives specified in this standard are not defined when an input set of domain parameters or a key is invalid (see 4.2 and Annex B). Implementations should, therefore, appropriately address this possibility. The risks of operating on invalid domain parameters or keys vary depending on the scheme used and the particular implementation, and are addressed by type of scheme in D.5.

DL and EC domain parameters and keys may be validated explicitly before being input to a primitive using, for example, the techniques given in A.16. Note that no techniques for private-key validation are provided because, generally, a party controls its own private key and need not validate it. However, private-key validation may be performed if desired. Also note that no techniques for IF public-key validation are provided in this standard; however, see Gennaro, Micciancio, and Rabin [B58] and Liskov and Silverman [B103] for some proposed techniques.

Alternatively, domain parameters may be validated within the infrastructure by an authority. For example, a certifying authority may validate domain parameters and keys as part of issuing a certificate; certificate verification will then imply validity of the domain parameters or keys it contains. Generation of keys or parameters by a trusted authority or jointly with it (see D.3.1) may provide assurance of their validity. As another means of domain parameter validation, a standards body may publish a set of domain parameters, in effect serving as a trusted third party assuring their validity. Such domain parameters have been published, for example, in ANSI X9.30:1-1997 [B4], ANSI X9.42 [B8], and FIPS PUB 186 [B56] for the DL family, and in ANSI X9.62-1998 [B11], ANSI X9.63 [B12], NIST [B119], and Standards for Efficient Cryptography [B144] for the EC family.

The above methods ensure that keys satisfy their definitions (i.e., are valid). They do not, however, ensure that the keys were generated in a secure manner. Such assurance, in addition to the assurance of validity, may be provided by ensuring that only appropriately validated modules can be used for generating keys (see D.7). (In contrast, domain parameters may potentially be generated in unvalidated modules, provided that the parameters are validated, since no secrets are involved.)

Public-key validation and POP (see D.3.2) can be considered as duals. Public-key validation shows that the party's public key is valid, but not necessarily that the party possesses the corresponding private key. POP demonstrates that the party possesses the private key, but not necessarily that the public key is valid (although certain methods for POP may demonstrate both). Both public-key validation and POP should be considered in high-assurance applications, unless the risk of operating with invalid keys or without assurance that a party possesses a private key is mitigated by other means.

Some additional ways to address the risks of operating on invalid domain parameters and keys are given by type of scheme in D.5.

D.3.4 Cryptoperiod and protection lifetime of domain parameters and keys

The *cryptoperiod* of a set of domain parameters or a key is the period during which operations may be performed with the set of domain parameters or the key. A set of domain parameters or a key should have an appropriate cryptoperiod to limit the amount of data whose protection is at risk (in the event of cryptanalytic attack or physical compromise of a private key), and the amount of data available for cryptanalysis. The cryptoperiod of a set of domain parameters should be at least as long as the cryptoperiod of keys associated with the set of domain parameters. The cryptoperiod of a public key used for signature verification should be at least as long as the cryptoperiod of the corresponding private key for signature generation; the cryptoperiod of a private key used for decryption should be at least as long as the cryptoperiod of the corresponding public key for encryption.

Keys with long cryptoperiods are known as *long-term* or *static*; keys with short cryptoperiods are known as *short-term* or *ephemeral*. These terms are most commonly used in the context of key agreement schemes (see D.5.1). Note that whether a key is static or ephemeral is independent of whether or not it is authenticated.

The *protection lifetime* of a key is the amount of time that the key is needed to protect data, and it is generally longer than the cryptoperiod of a key. For example, even after the cryptoperiod of an encrypting key expires, the data protected by it may still be sensitive; similarly, the data signed by a signing key may need to be protected from unauthorized modification long after the cryptoperiod of the signing key has expired. Hence, the security measures for a key (including the appropriate key sizes) should be picked primarily according to its protection lifetime, rather than its cryptoperiod.

In general, there is a distinction between the risk of an opponent learning a private key through physical compromise, versus an opponent learning a private key through cryptanalysis. Cryptanalysis may be defended against by appropriate key sizes and by limiting the number of operations performed with a given key and, thus, the availability of data to the cryptanalyst. The risk of physical compromise of a given key and the value of the key to the opponent may be reduced by giving a private key a short cryptoperiod and erasing the key thereafter.

Implications of domain parameter and key cryptoperiods are described further by type of scheme in D.5.

The cryptoperiod of domain parameters and public keys may be conveyed as part of the supporting key management (e.g., as certificate attributes). The cryptoperiod should be securely associated with the parameters and keys and protected from unauthorized modification (see D.3.6). To limit the impact of a successful cryptanalytic attack or a physical compromise, provisions should also be made in supporting key management for early termination of a set of domain parameters or a key (e.g., through certificate revocation) (see ITU-T Recommendation X.509 (1997 E) [B82], Section 13.6.3 of Menezes, van Oorschot, and Vanstone [B112], and Solo et al. [B142]).

D.3.5 Usage restrictions

A set of domain parameters or a key should have appropriate restrictions on its use to prevent misapplication of the set of domain parameters or the key as input to an operation, as well as misinterpretation of the results of an operation.

Examples of usage restrictions include

- Restrictions on the type of scheme; (e.g., only a signature scheme, or only an encryption scheme)
- Restrictions on scheme options (e.g., only a particular encoding method or hash function for a signature scheme)
- Restrictions on messages processed (e.g., only payment orders of a certain format, up to a certain value)

As a prudent security practice, the use of a particular key should be restricted to a single scheme with a single specific set of scheme options. Otherwise, the variability of data associated with the key may aid an opponent. If a single key is to be used for a variety of schemes or scheme options, additional security analysis is required. A set of domain parameters may be identified as intended to be used with a single specified scheme, with a set of specified schemes, or with all schemes to which it applies in a given family.

(The granularity of what constitutes a "scheme"— for instance, in terms of scheme options—depends on the implementation and on what is considered to constitute an attack.)

Usage restrictions should be securely associated with the parameters and keys and protected from unauthorized modification (see D.3.6). To accomplish this, they may be conveyed as part of the supporting key management (e.g., as certificate attributes) see e.g., ITU-T Recommendation X.509 (1997 E) [B82]. They may also be embedded (explicitly or implicitly) into a system. For example, in a closed system that is only capable of producing and verifying one specific type of signature, there may be no need to convey restrictions on the type of scheme or scheme options as part of key management.

D.3.6 Storage and distribution methods

The storage and distribution methods for domain parameters and keys should provide appropriate protection.

It is presumed that domain parameters and keys will be identified somehow (perhaps by the owner's identity) and possibly associated with certain attributes, such as ownership, cryptoperiod, and usage restrictions. It is important to protect the integrity of this identification and association when keys or domain parameters are stored and distributed. Specifically, it should be difficult for an opponent to modify the key or set of domain parameters, its identification, or any attributes that are associated with the key or set of domain parameters. The specific protections depend on the component.

— A set of domain parameters should be protected from unauthorized modification, but generally need not be protected from disclosure.

— A public key should be protected from unauthorized modification, but generally need not be protected from disclosure.

— A private key should be protected from unauthorized modification and, with the possible exception of its associated set of domain parameters, if any, protected from unauthorized disclosure.

— Identification information and attributes associated with a key or set of domain parameters should be protected from unauthorized modification.

The security properties for a storage or distribution method should be commensurate with the security properties intended to be provided by a set of domain parameters or key. A typical protection method for domain parameters and public keys is to bind them with identification information and attributes in a certificate, which may be stored or distributed; the signature on the certificate protects against unauthorized modification. Another method, commonly used for "root" public keys and domain parameters (such as public keys of root certifying authorities; see D.3.2), is to embed them in software or hardware in such a way that they become an inherent part of the system and are protected from unauthorized modification by the same mechanisms as the system itself (see D.7).

When a key is associated with a set of domain parameters, the set of domain parameters may either be stored and distributed explicitly with the key, or referenced implicitly. When the set of domain parameters is referenced implicitly, the reference and the set itself should be protected from unauthorized modification. For instance, suppose a key references a shared set of domain parameters. The reference to the shared set should be unambiguous, and the set itself should be protected from unauthorized modification.

D.4 Family-specific considerations

This subclause gives further information on security parameters for each of the three families, as well as generation of domain parameters (if any) and key pairs.

D.4.1 DL Family

D.4.1.1 Security parameters

The primary security parameters for the DL family are the length in bits of the field order *q*, and the length in bits of the subgroup order *r*. A common minimum field order length is 1024 bits and a common minimum subgroup order length is 160 bits (ANSI X9.44 [B8]) (see Note 1 of D.4.1.4).

The field type (prime versus binary) may affect the security of the schemes (see Note 2 of D.4.1.4).

D.4.1.2 Generation method

Considerations for generating domain parameters in the DL family include the following:

— *Prime case*: The field order *q* (i.e., the prime *p*) should be generated randomly or pseudo-randomly among those field orders with a sufficiently large subgroup, with a prime generation method that has a sufficiently low probability of producing a nonprime. For a canonical method of domain parameter generation that can be audited, the field order should be generated as a one-way function of a random seed (see Note 3 of D.4.1.4).

— *Binary case*: The field order *q* may be predetermined. In a canonical method of domain parameter generation that can be audited, the field order should be selected from a set of allowed values (see A.16.3 and A.16.4 for one such method) (see Note 4 of D.4.1.4).

— The subgroup order *r* may have any value consistent with the other parameters, provided it is sufficiently large and prime (and primitive in the case of binary fields; see Note 5 of D.4.1.4). It may be predetermined, generated randomly or pseudo-randomly, or derived from the field order. Prime generation or primality testing for the subgroup order may admit the possibility of error, provided the probability of error is sufficiently low (see Note 5 of D.4.1.4).

— The base *g* may have any value consistent with the other domain parameters (see Note 6 of D.4.1.4).

Considerations for generating key pairs in the DL family include the following:

— The private key *s* should be generated at random from the range $[1, r - 1]$, or a sufficiently large subset of it (see Note 7 of D.4.1.4).

The domain parameters *q*, *r*, and *g* (and the field representation in the binary case) may be shared among key pairs. The private key *s* should not be shared (see Note 8 of D.4.1.4).

D.4.1.3 Other considerations

The field representation (in the binary case) is not known to affect security, although, since the field representation is a domain parameter, it should be protected from unauthorized modification, along with the other domain parameters (see Note 9 of D.4.1.4).

D.4.1.4 Notes

1. *Security parameters*: The security of schemes in the DL family against attacks whose goal is to solve the discrete logarithm problem depends on the difficulties of general-purpose discrete logarithm methods and of collision-search methods. In turn, the difficulty of general-purpose discrete logarithm methods depends on the length in bits of the field order *q*; and the difficulty of collision-search methods depends on the length in bits of the subgroup order *r*.

For large prime fields, the fastest general-purpose discrete logarithm method today is the generalized number field sieve (GNFS) (see Gordon [B64], Section 3.6.5 of Menezes, van Oorschot, and Vanstone [B112], and Schirokauer [B131]), which has asymptotic running time exp $(((64/9)^{1/3} + o(1))\ (\ln q)^{1/3}\ (\ln \ln q)^{2/3})$, where $o(1)$ denotes a number that goes to zero as q grows. For more on estimating the complexity of GNFS for large prime fields, see Note 1 in D.4.3.4; the table given there also applies, except that the memory requirements for the linear algebra are $\log_2 q$ times greater for the discrete logarithm problem than they are for the integer factorization problem. (See http://www.rsasecurity.com/rsalabs/challenges/factoring/index.html for the status of solving the integer factorization problem. The result can be used as an estimate for an appropriate field order in the discrete logarithm system.) For large binary fields, a similar method (Section 3.6.5 of Menezes, van Oorschot, and Vanstone [B112] and Gordon and McCurley [B66]) has asymptotic running time within a constant factor of $\exp(1.587\ (\ln q)^{1/3}\ (\ln \ln q)^{2/3})$. In particular, the method is faster for binary fields than for prime fields.

For both types of field, the Pollard rho method (Pollard [B125] and Section 3.6.3 of Menezes, van Oorschot, and Vanstone [B112]) and the related Pollard lambda and other collision-search methods (e.g., van Oorschot and Wiener [B122]) can compute discrete logarithms after, on average, $(\pi\, r/4)^{1/2}$ field multiplications. The memory requirements of such methods are relatively small. (See ECC challenge link from http://www.certicom.com/research.html for the status of solving the elliptic curve discrete logarithm problem. The result can be used as an estimate for an appropriate subgroup order in the discrete logarithm system.)

The length of the field order and the length of the subgroup order should be selected so that both the general-purpose methods and the collision-search methods have sufficient running time. Often, the parameters are selected so that the difficulty of both methods is about the same. It does not have to be the same, however. For a variety of reasons, such as availability of hardware, an implementation may choose a larger field or subgroup. As noted, a common minimum field order length is 1024 bits, and a common minimum subgroup order length is 160 bits.

When a set of domain parameters is shared among parties, the size should also take into account the number of key pairs associated with the set, since the total running time for computing k discrete logarithms with the same set of domain parameters is only about $k^{1/2}$ times the running time of computing a single discrete logarithm, provided that more memory is available (van Oorschot and Wiener [B122]). In general, a set of domain parameters that is shared among a large number of key pairs should have larger security parameters than one that is shared among a small number of key pairs. A prudent practice is to pick security parameters such that computing even a single discrete logarithm is considered infeasible.

2. *Field type: prime versus binary*: The field type may affect the security of the schemes, because the running time of general-purpose discrete logarithm methods differs between prime fields and binary fields. Also, since the cost of finite field operations may differ between the two types of field, the actual running time of collision-search methods may vary by a small factor. Furthermore, since there is only one binary field for each field order length, there are fewer alternatives to choose from should some binary field be cryptanalyzed, than should a single prime field be cryptanalyzed.

3. *Prime generation*: The field order q (i.e., the prime p) should be generated randomly or pseudo-randomly, because this provides resistance to special-purpose discrete logarithm methods, such as the Special Number Field Sieve, or those given in Gordon [B63]. The prime generation method should have a sufficiently low probability that a nonprime is generated, so that the probability of generating an invalid set of domain parameters is small. A small but nonzero probability of error (e.g., $< 2^{-100}$) is acceptable, since it makes no difference in practice; methods with a small probability of error are generally simpler than those with zero error. See D.6 for more on generating random numbers and A.15 for more on generating primes.

A desired security level can also be provided when the field order has a special form; such a choice requires further security analysis by the implementer.

Computing the field order as a one-way function of a random seed provides a degree of auditing, because it makes it difficult to select a prime with some predetermined rare property. It may not entirely prevent the party generating the prime from producing primes with special properties, since the party can try many different seeds, looking for one that yields the desired property, if the property is likely enough to occur. However, it will make it difficult for a party to produce primes that are vulnerable to a special-purpose discrete logarithm method, such as the Special Number Field Sieve, provided that the prime is large enough. (A party might seek to introduce such a vulnerability in the interest of determining users' private keys.) It will also make it difficult to mount an attack on DSA, in which the opponent picks a particular subgroup order r in order for two messages to have the same signature (Vaudenay [B146]). (This attack is of concern if the message representative is not shorter than the subgroup order r.) See ANSI X9.30:1-1997 [B4], ANSI X9.42 [B8], and FIPS PUB 186 [B56] for an auditable method of generating primes by incremental search.

4. *Field order generation, binary case*: The field order q may be predetermined, since there is only one binary field for each field order length, and random generation is not meaningful. For the same reason, in canonical domain parameter generation, the field order is selected from a list of allowed values, not generated at random.

5. *Subgroup order*: The subgroup order r may have any value consistent with the other domain parameters (provided it is a primitive divisor of $q - 1$ in the case of binary fields, see A.3.9), because the difficulty of the collision-search methods for the discrete logarithm problem depends only on the size of the subgroup order, not on its particular value, as long as it is prime. Selecting a field order that is larger than the size of the hash value employed in a signature scheme, as is done in ANSI X9.62-1998 [B11], has the benefit of thwarting Vaudenay's attack (Vaudenay [B146]). A small but nonzero probability of error in prime generation or primality testing (e.g., $< 2^{-100}$) is acceptable, since it makes no difference in practice. The subgroup order r should be a primitive divisor of $q - 1$ in the case of binary fields (see A.3.9) because, otherwise, the subgroup is contained entirely in a proper subfield $GF(2^d)$ of $GF(2^m)$, in which case the opponent need only solve the DL problem in the smaller field $GF(2^d)$.

6. *Base*: The base g may have any value consistent with the other domain parameters, because the difficulty of the discrete logarithm problem does not depend on which base is employed for a given subgroup. All bases are equivalent in the sense that if an opponent can compute discrete logarithms with respect to one base, say h, the opponent can compute discrete logarithms with respect to any other base for the same subgroup, say g, by taking the ratio

$$\log_g w = \log_h w / \log_h g \bmod r$$

Computing the base g as a one-way function of a random seed provides a degree of auditing, because it makes it difficult to select a predetermined base, such as one that depends on another user's public key, as in Vaudenay [B146]. In particular, it thwarts attacks in which the opponent ensures that two parties have different (but related) bases, while thinking that they have the same set of parameters [B146]. Such attacks may also be prevented by ensuring that both parties have the same set of parameters—for example, by verifying the authenticity of the parameters, as described in D.3.2.

7. *Private keys*: The private key should be generated at random from the range $[1, r–1]$, because this maximizes the difficulty of recovering the private key by collision-search methods. A desired level of security can also be provided when the private key is restricted to a large enough subset of the range (e.g., is shorter than the subgroup order, has low weight, or has some other structure). Such choices require further security analysis by the implementer. (See D.6 for more on random number generation.)

8. *Domain parameters, public-key and private-key sharing*: A set of domain parameters may be shared, because by definition, the set of domain parameters may be common to any number of keys. A private key should not be shared.

9. *Field representation*: The field representation is not known to affect security; its only cryptographic role is the conversion of field elements to integers in the signature and verification primitives, and to octet strings in the key agreement schemes. The choice of field representation for domain parameters and keys is otherwise a matter of data formatting. In the single cryptographic role mentioned, no reason is currently known why the field representation should have any impact on security. However, it is important that the parties to a scheme agree on the representation in which conversion is to take place because, otherwise, an opponent may be able to trick the parties into operating with a different representation for conversion in an attempt to forge signatures or obtain information about keys.

D.4.2 EC family

D.4.2.1 Security parameters

The primary security parameter for the EC family is the length in bits of the subgroup order *r*. A common minimum subgroup order length is 161 bits (ANSI X9.62-1998 [B11]) (see Note 1 of D.4.2.4).

The field type (prime versus binary) may slightly affect the security of the schemes (see Note 2 of D.4.2.4).

D.4.2.2 Generation method

Considerations for generating domain parameters in the EC family include the following:

- *Prime case*: The field order *q* (i.e., the prime *p*) may have any value that provides a sufficiently large subgroup; it may be predetermined, or generated randomly or pseudo-randomly. Prime generation for the field order may admit the possibility of error, provided the probability of error is sufficiently low (see Note 3 of D.4.2.4).
- *Binary case*: The field order *q* may be predetermined (see Note 4 of D.4.2.4).
- The elliptic curve may be any curve satisfying the definition in 5.4, as well as the MOV and anomalous criteria (see Note 1 of D.4.2.4) that provide a sufficiently large prime-order subgroup. It may be predetermined, or generated randomly or pseudo-randomly, perhaps subject to certain conditions that simplify the computation of the number of points on the elliptic curve. For example, a 169 bit Koblitz curve may be estimated to be about 13 times faster to solve than a canonical random 169 bit curve; although in practice, the additional calculations may result in a smaller speedup. This corresponds to a decrease of about 3.5 bits of symmetric key exhaustion security, or 7 bits of elliptic curve security. For a canonical random method of domain parameter generation that can be audited, the elliptic curve coefficients should be generated as a one-way function of a random seed (see A.12.4 through A.12.7 for one such method) (see Note 5 of D.4.2.4).
- The subgroup order *r* may have any value consistent with the other domain parameters, provided that it is sufficiently large and prime. Prime generation or primality testing for the subgroup order may admit the possibility of error, provided the probability of error is sufficiently low (see Note 6 of D.4.2.4).
- The base point *G* may have any value consistent with the other domain parameters (see Note 7 of D.4.2.4).

Considerations for generating key pairs in the EC family include the following:

- The private key *s* should be generated at random from the range [1, $r - 1$], or a sufficiently large subset of it (see Note 8 of D.4.2.4).

The domain parameters may be shared among key pairs. The private key *s* should not be shared (see Note 9 of D.4.2.4).

D.4.2.3 Other considerations

The field representation (in the binary case) is not known to affect security; although, because the field representation is a domain parameter, it should be protected from unauthorized modification, along with the other domain parameters (see Note 10 of D.4.2.4).

D.4.2.4 Notes

1. *Security parameters*: The security of schemes in the EC family against many attacks whose goal is to solve the elliptic curve discrete logarithm problem depends on the difficulty of collision-search methods, which are the fastest methods known today for computing discrete logarithms on an arbitrary elliptic curve. The difficulty of collision-search methods depends, in turn, on the length in bits of the subgroup order *r*. The Pollard rho (Pollard [B125]) and the related Pollard lambda, as well as other collision-search methods (e.g., van Oorschot and Wiener [B122]), can compute elliptic curve discrete logarithms after, on average, $(\pi r/4)^{1/2}$ elliptic curve additions. The memory requirements of such methods are relatively small. For special classes of curves, improved collision-search methods are available. In particular, curves not satisfying the MOV condition (see A.12.1) and anomalous curves over prime fields (those where the order *r* of the base point is equal to the field order *q* and, hence, by the Hasse bound (A.9.5), equal to the curve order; see Semaev [B134], Smart [B140], Satoh and Araki [B130]), are considered insecure and are, thus, avoided in this standard. Collision-search methods may also be sped up by a factor of about $(m/e)^{1/2}$ for curves over a binary field $GF(2^m)$ whose coefficients lie in a smaller subfield $GF(2^e)$ (see Gallant, Lambert, and Vanstone [B57] and Wiener [B148]). This improvement is not considered significant enough to warrant avoidance of such curves, but it may suggest higher key sizes than for general curves. (See ECC challenge link from http://www.certicom.com/research.html for more information on the status of solving the elliptic curve discrete logarithm problem.)

   Table D.1 provides an estimate for the running time of collision-search methods for different sizes of *r*. The estimate is given in MIPS-years, which is the approximate amount of computation that a machine capable of performing one million arithmetic instructions per second would perform in one year (about 3×10^{13} arithmetic instructions).

Table D.1—Estimated crypotographic strength for EC generator order sizes

Size of generator order *r* (Bits)	Processing time (MIPS-years)
128	4.0×10^5
172	3×10^{12}
234	3×10^{21}
314	2×10^{33}

There is some variation among published estimates of running time due to the particular definition of a MIPS-year and to the difficulty of estimating actual processor utilization. (How many arithmetic instructions a modern processor performs in a second when running an actual piece of code depends not only on the clock rate, but also heavily on the processor architecture, the amount and speed of caches and RAM, and the particular piece of code.) Thus, the estimates given here may differ from others in the literature, although the relative order of growth remains the same. The length of the field order and the length of the subgroup order should be selected so that the collision-search methods have sufficient difficulty. A common minimum subgroup order length is 161 bits.

When a set of domain parameters is shared among parties, the size should also take into account the number of key pairs associated with the set, because the total running time for computing k elliptic curve discrete logarithms with the same set of domain parameters is only about $k^{1/2}$ times the running time of computing a single elliptic curve discrete logarithm, provided that more memory is available (van Oorschot and Wiener [B122]). In general, a set of domain parameters that is shared among a large number of key pairs should have larger security parameters than one that is shared among a small number of key pairs. A prudent practice is to pick security parameters such that computing even a single elliptic curve discrete logarithm is considered infeasible.

2. *Field type: prime versus binary*: The field type may slightly affect the security of the schemes, since the cost of finite field operations may differ between the two types of fields, and the actual running time of collision-search methods may vary by a small factor. Furthermore, since there is only one binary field for each field order length, there are fewer alternatives to choose from should some binary field be cryptanalyzed, than should a single prime field be cryptanalyzed. In either case, however, there are many elliptic curves to choose from, and no method is known that cryptanalyzes all elliptic curves over a given field at the same time.

3. *Prime generation*: No special primes are known where computing elliptic curve discrete logarithms might be substantially easier. A small but nonzero probability of error in prime generation or primality testing (e.g., $< 2^{-100}$) is acceptable, since it makes no difference in practice; methods with a small probability of error are generally simpler than those with zero error. (See A.15 for more on generating primes.)

4. *Field order generation, binary case*: The field order q may be predetermined, because there is only one binary field for each field order length.

5. *Elliptic curve*: The elliptic curve may be any curve satisfying the definition in 5.4, as well as MOV and anomalous criteria (Note 1) that provide a sufficiently large prime-order subgroup, as no other special classes of elliptic curves are known where computing elliptic curve discrete logarithms might be substantially easier. The elliptic curve may be predetermined, or generated randomly or pseudo-randomly, perhaps subject to certain conditions that simplify the computation of the number of points on the elliptic curve. Among the methods for curve generation described in A.9.5, Method 1 (selecting a random curve and computing its order) imposes no additional conditions on the curve, and the other three methods impose additional conditions. In particular, Method 3 (selecting a curve over a subfield) imposes additional conditions that make the discrete logarithm computation on the curve subject to the speed-up due to Gallant, Lambert, and Vanstone [B57] and Wiener and Zuccherato [B148], as further described in Note 1.

   Computing the elliptic curve coefficients (or other inputs to elliptic curve generation) as a one-way function of a random seed provides a degree of auditing, because it makes it difficult to select an elliptic curve with some predetermined rare property. It may not entirely prevent the party generating the elliptic curve from producing curves with special properties, since the party can try many different seeds, looking for one that yields the desired property, if the property is likely enough to occur. However, it will make it difficult to mount an attack in which the opponent picks a particular subgroup order r in order for two messages to have the same signature. (See Vaudenay [B146] for such an attack on DSA, and also Note 6.) (See A.12.4 through A.12.7 for an example of such a method.)

6. *Subgroup order*: The subgroup order r may have any value consistent with the other domain parameters, since the difficulty of the collision-search methods for the elliptic curve discrete logarithm problem depends only on the size of the subgroup order, not on its particular value, provided that the subgroup order is prime. Selecting a field order that is larger than the size of the hash value employed in a signature scheme, as is done in ANSI X9.62-1998 [B11], has the benefit of thwarting Vaudenay's attack (Vaudenay [B146]). A small but nonzero probability of error in prime generation or primality testing (e.g., $< 2^{-100}$) is acceptable, since it makes no difference in practice.

7. *Base*: See Note 6 of D.4.1.4, replacing "base g" with "base point G."

8. *Private keys*: See Note 7 of D.4.1.4.

9. *Domain parameters, public-key and private-key sharing*: See Note 8 of D.4.1.4.

10. *Field representation*: See Note 9 of D.4.1.4.

D.4.3 IF family

D.4.3.1 Security parameter

The primary security parameter for the IF family is the length in bits of the modulus n. A common minimum length is 1024 bits (ANSI X9.31-1998 [B7]) (see Note 1 of D.4.3.4).

The key type (RSA versus RW) is not known to affect the security of the schemes with the recommended encoding methods (see Note 2 of D.4.3.4).

D.4.3.2 Generation method

Considerations for generating keys in the IF family include the following:

— The lengths of the primes p and q should be approximately half the length of the modulus n, although other lengths may also provide sufficient security. See A.16.11 and A.16.12 for examples (see Note 3 of D.4.3.4).

— The primes p and q should be generated randomly or pseudo-randomly with a prime generation method that has a sufficiently low probability of producing a non-prime. If auditing of key generation is required, then the primes should be generated as a one-way function of a random secret seed (see Note 4 of D.4.3.4).

— The public exponent e may have any value consistent with the modulus and key type; it may be predetermined, or generated randomly or pseudo-randomly (see Note 5 of D.4.3.4).

— The private exponent d should be derived from the public exponent e or generated at random (see Note 6 of D.4.3.4).

The public exponent e may be shared among keys. The modulus n, the primes p and q, and the private exponent d should not be shared (see Note 7 of D.4.3.4).

D.4.3.3 Other considerations

The private-key representation does not affect security in general, although the effectiveness of certain physical attacks may vary according to the representation. The private key should be stored securely, regardless of the representation, as discussed in D.7 (Note 8 of D.4.3.4).

D.4.3.4 Notes

1. *Modulus size*: The security of schemes in the IF family against attacks whose goal is to recover the private key depends on the difficulty of integer factorization by general-purpose methods, which, in turn, depends on the length in bits of the modulus n. For large moduli, the fastest general-purpose factoring method today is the GNFS (see Buchmann, Loho, and Zayer [B32] and Buhler, Lenstra, and Pomerance [B34]), which has asymptotic running time $\exp(((64/9)^{1/3} + o(1)) (\ln n)^{1/3} (\ln \ln n)^{2/3})$, where $o(1)$ denotes a number that goes to zero as n grows. Previously, the fastest general-purpose method was the multiple polynomial quadratic sieve (MPQS) (Silverman [B138]), which has a running time of about $O(\exp (\ln n \ln \ln n))$. See Section 3.2 of Menezes, van Oorschot, and Vanstone [B112] and Odlyzko [B121] for more discussion on integer factorization. (See http://www.rsasecurity.com/rsalabs/challenges/factoring/index.html for more information on the status of solving the integer factorization problem.)

   In order to compute the complexity of the GNFS accurately for a particular n, one needs to know the precise value of the $o(1)$ term for that n. In particular, the complexity of the GNFS is difficult to measure accurately for numbers of cryptographic interest, because they are larger than GNFS can currently factor. The standard practice is to estimate the complexity by extrapolating from running times observed for factoring smaller numbers. Following this practice, ANSI X9.31-1998 [B7] provides estimates for GNFS factorization at various modulus sizes, which are reproduced in Table D.2 below. In this table, *processing time* is the total computational effort, given in MIPS-years (a MIPS-year is the approximate amount of computation that a machine capable of performing one million arithmetic instructions per second would perform in one year [about 3×10^{13} arithmetic instructions]); *memory requirements* consider the amount of processor memory for a sieving process, which may be distributed among many processors, and for linear algebra operations, which would typically be on a single processor; and *storage requirements* is the amount of disk space.

   Table D.2—Estimated cryptographic strength for IF modulus sizes

Modulus size	Processing time (MIPS-years)	Memory requirements (bytes)		Storage requirements (bytes)
		Sieving process	*Linear algebra*	
512	4.0×10^5	1.3×10^8	2.0×10^{10}	5.0×10^{10}
1024	3×10^{12}	3×10^{11}	1×10^{14}	2×10^{14}
2048	3×10^{21}	8×10^{15}	3×10^{18}	7×10^{18}
4096	2×10^{33}	8×10^{21}	3×10^{24}	7×10^{24}

   There is some variation among published estimates of running time due to the particular definition of a MIPS-year and to the difficulty of estimating actual processor utilization. (How many arithmetic instructions a modern processor performs in a second when running an actual piece of code depends not only on the clock rate, but also heavily on the processor architecture, the amount and speed of caches and RAM, and the particular piece of code.) Thus, the estimates given here may differ from others in the literature, although the relative order of growth remains the same. The length of the modulus should be selected so that integer factorization methods have sufficient difficulty; as mentioned earlier, a common minimum length is 1024 bits.

2. *Key type: RSA versus RW*: The key type (which varies only for the signature schemes) is not known to affect the security of the schemes with recommended and correctly implemented encoding methods. Although root extraction for an even exponent is provably as difficult as integer factorization, only certain encoding methods [e.g., PSS (Bellare and Rogaway [B19])] result in

schemes whose security can be proven equivalent to integer factorization; the ones recommended here do not. However, certain attack strategies on signature schemes (Joye and Quisquater [B84]) have been shown to yield the factorization of the modulus in the RW case, but not in the RSA case. Root extraction for an odd exponent is not known to be equivalent to integer factorization. (Boneh and Venkatesan [B30] have recently shown strong evidence that the problems are not equivalent for low-exponent RSA.) Nevertheless, integer factorization is the only known approach for forging signatures or recovering messages for the schemes with recommended encoding methods, regardless of key type.

3. *Prime size*: The lengths in bits of the primes p and q should be approximately half the length of the modulus n, because this maximizes the difficulty of certain special-purpose integer factorization methods. Such methods include the Pollard rho method (Pollard [B124]), with asymptotic expected running time $O(p^{1/2})$; the Pollard $p - 1$ method (Pollard [B123]), with running time $O(p')$, where p' is the largest prime factor of $p - 1$; and the $p + 1$ method (Williams [B150]), with running time $O(p')$, where p' is the largest prime factor of $p + 1$. The elliptic curve method (ECM) (Lenstra [B101]) is superior to these; its asymptotic running time is $O(\exp\ (2\ln\ p\ \ln\ \ln\ p)^{1/2})$. All these methods are slower than GNFS (see Note 1) for factoring a large IF modulus with prime factors that are approximately half the length of the modulus. These methods may be effective, however, when one of the prime factors is small.

   A desired security level can also be provided when the lengths of the primes are not approximately half the length of the modulus, as long as the primes are large enough to resist the special-purpose methods just mentioned. Such a choice requires further security analysis by the implementer. (For further discussion, see Shamir [B136]; see also Gilbert et al. [B59] for a security analysis.)

4. *Prime generation*: The primes should be generated randomly or pseudo-randomly, because this provides resistance to exhaustive search and other special-purpose attacks. (See D.6 for more on random number generation.) The prime generation method should have a sufficiently low probability that a nonprime is generated, so that the probability of generating an invalid key is small. A small but nonzero probability of error (e.g., $< 2^{-100}$) is acceptable, since it makes no difference in practice; methods with a small probability of error are generally simpler than those with zero error. Other criteria may be added to prime generation to ensure that primes meet certain properties. Acceptable prime-generation methods include those described in A.15.6 and A.15.8 with either probabilistic primality testing (probability of nonprime output of less than 2^{-100}; see A.15.1 through A.15.2) or primality proving (see A.15.4); and recursive construction of primes with an implicit proof of primality, such as Maurer [B107], Mihailescu [B116], and Shawe-Taylor [B137]. (See also ANSI X9.31-1998 [B7], ANSI X9.80 [B13], and Sections 4.1–4.4 of Menezes, van Oorschot, and Vanstone [B112].)

   Computing primes as a one-way function of a random seed provides a degree of auditing, because it makes it difficult to select a prime with some predetermined rare property. It may not entirely prevent a user from producing primes with special properties, since a user can try many different seeds, looking for one that yields the desired property, if the property is likely enough to occur. However, it will make it difficult for a user to produce primes that are vulnerable to a special-purpose factoring method such as the $p - 1$ method, provided that the prime is large enough. (A malicious user may wish to do this in the interest of later disavowing a signature.) The seed should be kept secret, because revealing it would reveal p and q and, thus, the private key. Therefore, the actual auditing must be done only by trusted parties or in situations when it is appropriate to reveal the private key. See ANSI X9.31-1998 [B7] for an auditable method of generating primes by incremental search.

5. *Public exponent selection*: The public exponent may have any value consistent with the modulus and key type, because the value of the public exponent is not known to affect the security of the schemes with recommended and correctly implemented encoding methods. IF schemes with nonrecommended (or incorrectly implemented) encoding methods, or implementations that leak a fraction of the bits of the private exponent or the primes, may be vulnerable to certain attacks when

the public exponent is small (see Boneh, Durfee, and Frankel [B28], Coppersmith et al. [B39], and Hastad [B70]). Moreover, it has been shown that for very small public exponents such as $e = 3$, the RSA problem may not be equivalent to the integer factorization problem [B30] (see Note 2). However, the recommended encoding methods and appropriate protection of the private key (see D.7) prevent these attacks. Typical public exponent values are $e = 2$ for RW keys and $e = 3$ or $2^{16} + 1$ for RSA keys. A larger public exponent may, nevertheless, offer an additional line of defense, as it can mitigate concerns about implementation failures in the encoding methods or in the underlying random number generation for an IF encryption scheme.

Restricting the allowed set of public exponents system-wide, and ensuring that system components refuse to perform operations with public exponents that do not satisfy the restriction, may provide a measure of protection against protocol attacks that exploit the ability of an opponent to pick an arbitrary public exponent (e.g., Chen and Hughes [B37]). However, such attacks are generally better defended against by picking appropriate protocols.

A more common use for a system-wide restriction is picking a specific public exponent system-wide in order to avoid having to store and transmit it with the public key. For example, a system using RW keys may have a system-wide policy that $e = 2$ for all keys, in which case just n, not (n, e), needs to be transmitted and stored for public keys. The system-wide e should be protected from unauthorized modification the same way any public key would be.

6. *Private exponent selection*: The private exponent should be derived from the public exponent or generated at random (see D.6), so that with high probability, it will have approximately the same size as the modulus. In particular, the private exponent should be substantially longer than one quarter of the modulus length. This makes it difficult to recover the private exponent by certain methods, such as Boneh and Durfee [B27] and Wiener [B147]. (Note that ANSI X9.31-1998 [B7] requires that for a 1024 bit modulus, the private exponent is at least $2^{512+128s}$, where s is an integer ≥ 0.) A desired difficulty can also be provided when the private exponent is significantly shorter than the modulus or has some other structure, such as low weight; such choices require further security analysis by the implementer.

7. *Modulus, prime, and exponent sharing*: A modulus should not be shared because it is possible, given two public keys with the same modulus and one of the corresponding private keys, to determine the other private key. A prime should not be shared, because moduli with one prime factor in common can be factored by taking their GCD. A private exponent should not be shared, because it is the only private component of the private key. Sharing of any of these quantities is unlikely to occur if primes are generated at random and the private exponent is derived from the public exponent or generated at random. A public exponent may be shared because it is public and may have any value consistent with the modulus and key type.

8. *Private-key representation*: The choice of private-key representation does not affect security against cryptanalytic attack, although security against certain implementation attacks may vary according to the representation. For instance, the Bellcore fault-analysis attack (Boneh, DeMillo, and Lipton [B26]) is more feasible if the private key contains the prime factors of the modulus, because a single undetected bit error in an exponentiation modulo one of the primes will compromise the private key. (See D.7 for more on implementation attacks.)

D.5 Scheme-specific considerations

This clause gives further information on security implications of choices for each of the three types of scheme, including primitives; key derivation function or encoding method; key derivation or encoding parameters; and authentication, validation, and cryptoperiod specifics for keys in the schemes.

D.5.1 Key agreement schemes

Key agreement schemes are generally used to ensure that nobody but the two legitimate parties involved in the protocol can compute the shared secret key. Note that a basic key agreement scheme does not ensure that the parties are correct about each other's identities, or that both parties can actually compute their shared secret keys. Key confirmation (see D.5.1.3) is often used to provide such additional assurances.

In addition, key agreement schemes are sometimes used to provide forward secrecy. See D.5.1.7 for more on forward secrecy and how the key agreement schemes in this standard can provide it.

The application of key agreement schemes in a key establishment protocol is a particularly challenging task, as illustrated by the various attacks on key agreement protocols that have been observed over the years. Apparently slight changes, such as the ordering of messages, may well introduce unintended security weaknesses. Conversely, slight differences may yield security benefits. For instance, requiring each party to commit to its short-term key by sending a hash of the short-term key, prior to exchanging the short-term keys, can remove some potential for an opponent to manipulate a protocol. Although the recommendations in this subclause cover many of the general principles of key agreement, the implementer is encouraged to consult recent literature on key agreement protocols for further information.

D.5.1.1 Primitives

Secret value derivation primitive choices include the following (the particular choices vary among the three key agreement schemes):

— DLSVDP-DH, DLSVDP-DHC, DLSVDP-MQV, DLSVDP-MQVC

— ECSVDP-DH, ECSVDP-DHC, ECSVDP-MQV, ECSVDP-MQVC

The choice between DL and EC affects security to the extent that the difficulties of the underlying problem differ (see D.4). Attacking the underlying problem is the best currently known method for attacking the primitives. [For ECSVDP-DH and ECSVDP-DHC, attacking the primitive is equivalent to attacking the underlying problem in the sense that if one takes exponential time, then the other takes exponential time as well (Boneh and Lipton [B29]).] The choice of -DH versus -DHC, or -MQV versus -MQVC, affects security with respect to key validation (see D.5.1.6).

D.5.1.2 Key derivation function

The only recommended key derivation function is KDF1.

A key derivation function for the key agreement schemes in this standard (i.e., the three DL/ECKAS schemes) should produce keys that are computationally indistinguishable from randomly generated keys. (See D.6 for more on computational indistinguishability.) KDF1 is considered to have this property, provided that the length of the key derivation parameters is the same for all keys computed from a given shared secret value.

The security of KDF1 depends on the underlying hash function; SHA-1 and RIPEMD-160 are allowed. The length of the hash function's intermediate chaining value (for iterative constructions such as SHA-1 and RIPEMD-160) governs the security of KDF1 against certain forms of attack. Typical theoretical attacks for distinguishing KDF1 from a random source involve on the order of 2^{80} operations, assuming a 160 bit hash function, and these attacks do not necessarily yield any particular key.

D.5.1.3 Key confirmation

Key confirmation is the assurance provided to each party participating in the key agreement protocol that the other party is actually capable of computing the agreed-upon key, and that it is the same for both parties. Note that key agreement does not necessarily, by itself, provide key confirmation. For example, an opponent may present a public key that is identical or related to some other party's public key, to make it appear as though a derived key is shared with the opponent, when in fact it is shared with the other party. This is known as the *unknown key-share* attack (see Blake-Wilson and Menezes [B22], Diffie, van Oorschot, and Wiener [B48], Kaliski [B89], Law et al. [B98], and Menezes, Qu, and Vanstone [B113]).

An unknown key share attack may be a concern in situations where a party assumes that another party has certain privileges as a result of a successful key agreement protocol, and the other party is not further identified in the protocol. If an opponent can make it appear as though a derived key is shared with the opponent, when in fact it is shared with the other party, then the opponent may be able to impersonate the other party in subsequent messages encrypted with the derived key (even though the opponent does not know the derived key). In typical protocols, this is a theoretical concern, since exchanged messages (e.g., funds transfers) should generally identify the parties.

In general, if an opponent can obtain a certificate for another party's public key, the opponent can trivially mount an unknown key share attack by substituting its certificate for the other party's certificate. The CA can prevent this kind of attack by checking for duplicate public keys, or by requiring a proof of possession of the corresponding private key.

For variants of DH2, where one party's ephemeral key is combined with another party's static key, an opponent can also mount an unknown key share attack by substituting powers or multiples of the combined static and ephemeral public keys, obtaining a certificate for the new static key, and substituting the new certificate (Menezes, Qu, and Vanstone [B113]). A CA can prevent this attack by requiring a proof of possession of the private key, but cannot prevent it by checking for duplicates.

For MQV, an opponent can mount an unknown key share attack with certain substitutions, as further described in Kaliski [B89]. A CA cannot prevent the attack either by checking for duplicates or by requiring a proof of possession of the private key. As a result, if the unknown key share attack is a concern, protocol steps such as key confirmation are essential.

In general, to avoid unknown key share attacks and other possible attacks, the parties should perform a key confirmation protocol that securely associates the names of the parties with the knowledge of the shared secret key. Such protocols usually involve computations of cryptographic functions based on the identities of the parties and the derived keys (see e.g., ANSI X9.42 [B8], ANSI X9.62-1998 [B11], Blake-Wilson, Johnson, and Menezes [B21], and Menezes, Qu, and Vanstone [B113]).

D.5.1.4 Key derivation parameters

The purpose of key derivation parameters is to distinguish one derived key from another. As such, the parameters should have an unambiguous interpretation. If no other key confirmation is performed, they should identify the parties exchanging the key and specify the purpose of the derived key (e.g., its type and position within a sequence of derived keys of the given type).

For KDF1, the set of all possible key derivation parameters for a given shared secret value should be prefix-free. [A string P is called a *prefix* of a string S if S begins with P (i.e., $S = P \parallel R$ for some string R); a set of strings is *prefix-free* if it does not contain a pair of strings P, S such that P is a prefix of S.] Otherwise, due to the nature of hash functions used in KDF1, an opponent may be able to derive new shared secret keys based on the same shared secret value and new key derivation parameters, without knowing the shared secret value. If all key derivation parameters are the same length, or if they are BER or DER encodings of ASN.1 structures, they will satisfy the prefix-free requirement.

D.5.1.5 Authentication of ownership

If it is desired to verify that a particular party has the ability to compute a given derived key, the party's ownership of a public key from which the derived key is computed should be authenticated. Table D.3 summarizes the typical means of associating the derived key with a particular party, in terms of which public key should be authenticated.

Table D.3—Summary of typical means of associating public keys with parties in a key agreement scheme

Scheme	Authenticated public key
DH1	The party's public key
DH2	Either of the parties' public keys
MQV	The party's first public key

In general, as noted in D.3.2 and D.5.1.2, the party's possession of the corresponding private key should also be authenticated by means of a key confirmation protocol or otherwise. The risk of not doing so is the same as the lack of key confirmation, as further discussed in D.5.1.3.

A situation in which it may be acceptable not to authenticate a party's possession of a private key (directly or through key confirmation) is one in which the key derivation parameters include information identifying the party. The identifying information avoids any ambiguity in the sharing of the derived key.

D.5.1.6 Validation of domain parameters and keys

As discussed in D.3.3, using invalid keys or domain parameters as inputs to a primitive carries with it certain risks. Specifically, for the key agreement schemes, the risks are outlined below. Note that the risks will vary with the particular implementation:

- Failure of the implementation, resulting in an error state
- Reduction in size of the key space for derived keys (see, e.g., Jablon [B83])
- Compromise of information about a private key that is combined with the invalid public key in a secret value derivation primitive (Lim and Lee [B102])

An attack in which an opponent deliberately provides an invalid public key is sometimes referred to as a *small-subgroup attack*. If an opponent's public key is not valid, the opponent may be able to use a small subgroup of the underlying group to confine the shared secret value or to obtain information about the legitimate party's private key. (If both public keys are valid, then the shared secret value ranges over the subgroup of size r generated by the generator g; otherwise, it may be outside the subgroup of size r, possibly in a subgroup of small size.)

These risks can be mitigated by ensuring the validity of domain parameters and public keys. However, validation does not address certain other risks. A key may be valid and still be insecure (e.g., if the random number generation for the key generation was performed improperly, or if the private key is stored insecurely or deliberately disclosed). Therefore, an implementer should consider other ways in which the key may be insecure, and then decide how to appropriately mitigate the risk. FIPS PUB 140-1 implementation validation and random number generation are typical ways to address some of these concerns (see D.6.1 and D.7).

An alternative to validating the public key is to select a secret value derivation primitive with cofactor multiplication (i.e., -DHC or -MQVC). Provided that the associated set of domain parameters is valid, and the public key is validated as an element of the underlying group (i.e., element of the finite field or point on the elliptic curve), a -DHC or -MQVC primitive will operate appropriately on invalid public keys; no further validation is necessary.

A situation in which it may be acceptable not to validate a public key is one in which the public key is authenticated (typically by the other party in a key agreement operation), and the private key with which the public key is combined in a secret value derivation primitive has a short cryptoperiod. The authentication prevents an outside opponent from substituting an invalid public key in an attempt to reduce the key space. The short cryptoperiod of the private key mitigates the potential for an adversary to benefit from compromising information about the private key.

The amount of information about the private key that the opponent can possibly compromise by using an invalid public key may be limited if the group has few elements of order less than r (e.g., if $q = 2r + 1$ in the DL case or if the cofactor k is small in the EC case), as described in Lim and Lee [B102]. However, an opponent may still be able to reduce the variability of the shared secret value by confining it to a small subgroup.

D.5.1.7 Cryptoperiod and protection lifetime

In a key agreement scheme, if an opponent learns a party's private key or keys, then the opponent may be able to recover derived keys produced from the private key or keys. The longer the cryptoperiod, the more derived keys are potentially vulnerable to recovery. The private keys used for key agreement schemes should be erased after their cryptoperiods expire, in order to limit a private key's potential vulnerability to physical compromise.

By giving certain private keys a short cryptoperiod and erasing them after they expire, it is possible to overcome the risk of recovery of derived keys due to the compromise of parties' cryptographic states. This prevents a passive opponent who merely recorded past communications encrypted with the shared secret keys from decrypting them some time in the future by compromising the parties' cryptographic state. This property is called *forward secrecy*. *One-party forward secrecy* means that an opponent's knowledge of that party's cryptographic state after a key agreement operation does not enable the opponent to recover derived keys. *Two-party forward secrecy* (Gunther [B68]) (see also Diffie, van Oorschot, and Wiener [B48]), means that an opponent's knowledge of *both* parties' cryptographic state does not enable recovery of previously derived keys.

Table D.4 summarizes the typical means for achieving forward secrecy with respect to one party or both parties with the key agreement schemes, in terms of which private keys should have a short cryptoperiod (i.e., be short-term).

Table D.4—Summary of typical means for achieving forward secrecy in a key agreement scheme

Scheme	Short-term private key (one-party forward secrecy)	Short-term private keys (two-party forward secrecy)
DH1	Party's private key	Both parties' private keys
DH2	Either party's private key	Both parties' second private keys (see note below)
MQV	Party's second private key	Both parties' second private keys

Here, "short-term" means that a private key is generated immediately prior to a secret value derivation operation, and destroyed as soon as possible thereafter.

Key cryptoperiod and authentication are independent considerations; a public key may be authenticated even if the corresponding private key is short-term. The STS protocol is an example of this approach (see p. 568 of Diffie [B46], Diffie, van Oorschot, and Wiener [B48], and Section 12.6 of Menezes, van Oorschot, and Vanstone [B112]).

NOTE—In the DH2 scheme, it is also possible to achieve two-party forward secrecy if both parties' first private keys are short-term; however, conventionally, it is the second private keys that are short-term for two-party forward secrecy. Two-party forward secrecy is not achieved if one party's first key and the other party's second key are short-term while the other keys are long-term.

D.5.2 Signature schemes

Signature schemes are generally used to provide authenticity of data—an assurance that only the party possessing the corresponding private key is capable of producing a signature that verifies as valid with a given public key. A commonly used theoretical definition for the security of signature schemes is given in Goldwasser, Micali, and Rivest [B62].

There are two types of signature schemes: signature schemes with appendix, and signature schemes with message recovery. Signature schemes with appendix require the message to be transmitted in addition to the signature; without knowing the message signed, one cannot verify the signature. Signature schemes with message recovery produce signatures that contain the message within them. The verifier does not need to know the message in order to verify the signature; if the signature is valid, the message signed is recovered during the signature verification process. While signature schemes with appendix may be used to sign messages of practically any length (subject only to the limitations of the encoding method), signature schemes with recovery are generally used for messages that are short enough. Note that no signature schemes with recovery are defined in this standard (see C.3.4 and C.3.7) and, hence, no security considerations are given here for such signature schemes.

D.5.2.1 Primitives

Signature and verification primitive choices include the following pairs (the particular choices vary among the three signature schemes):

— DLSP-NR and DLVP-NR, DLSP-DSA and DLVP-DSA, ECSP-NR and ECVP-NR, ECSP-DSA and ECVP-DSA, IFSP-RSA1 and IFVP-RSA1, IFSP-RSA2 and IFVP-RSA2, IFSP-RW and IFVP-RW

The choice between DL, EC, and IF affects security to the extent that the difficulties of the underlying problems differ (see D.4). Although none of the primitives is known to be equivalent to the underlying problem, attacking the underlying problem is the best currently known method for attacking the primitives. Note that the DL and EC signature primitives use randomness and, thus, require an appropriate random number generator for security (see D.6 for more on random number generation).

D.5.2.2 Encoding methods

The recommended encoding methods for signatures with appendix are EMSA1 (for DL/ECSSA) and EMSA2 (for IFSSA).

An encoding method for a signature scheme with appendix in this standard (i.e., DL/ECSSA or IFSSA) should have the following properties, stated informally:

— It should be difficult to find a message with a given message representative (the *one-way* property).

- It should be difficult to find two messages with the same representative (*collision resistance*).
- The encoding method should have minimal mathematical structure that could interact with the selected signature primitive (e.g., if the signature primitive is multiplicative, the encoding method should not be).
- The encoding method may identify the hash function (and other options) within the message representative in order to prevent an opponent from tricking a verifier into operating with a different hash function than the signer intended—perhaps a broken hash function; however, this may also be accomplished by key usage restrictions (see D.3.5).

EMSA1 is considered to have these properties (except for identifying the hash function) for DL/ECSSA. EMSA2 is considered to have these properties for IFSSA. (Although EMSA1 does not identify the hash function, in practice it is typically combined with a single hash function, SHA-1, which amounts to a key usage restriction, and removes any risk of ambiguity.)

The security of EMSA1 and EMSA2 depends on the underlying hash function; SHA-1 and RIPEMD-160 are allowed. The length of the hash function output governs the security of both encoding methods. Typical attacks for finding a message with a given representative involve on the order of 2^{160} operations, assuming a 160 bit hash function; attacks for finding two messages with the same representative involve 2^{80} operations.

If the maximum length l of the message representative is less than the output length of the hash function, then EMSA1 will simply truncate the hash function output. Therefore, for DL and EC signature schemes, if the length of the generator order r (and, hence, the maximum length of the message representative) is less than 160 bits, the security of the encoding method will be limited by 2^l and $2^{l/2}$, rather than 2^{160} and 2^{80}, respectively. In addition, for DLSP-DSA and ECSP-DSA, if the length of r is not greater than 160, then an opponent may pick r in such a way as to cause two different messages to have the same signature Vaudenay [B146]. This can be prevented by increasing the length of r beyond 160 bits, or by auditing domain parameter generation (see Note 3 in D.4.1.4 and Note 5 in D.4.2.4). It is customary for the length of r and the length of the hash function output to be approximately equal, so the message representative input to the primitive appears to be a random value between 0 and $r - 1$. If the length of r is greater than the length of the hash value, then this appearance will no longer hold; however, this is not known to result in any security risk.

A further discussion of desired theoretical properties for signature schemes may be found in Bellare and Rogaway [B19].

D.5.2.3 Repudiation

Signature schemes have a special security concern, similar to that for handwritten signatures. Namely, a dishonest signer may be interested in using deliberately weak methods in order to produce a signature that is accepted, but later be able to claim that the signature was forged by someone else who “broke” the cryptosystem. This may allow the signer to later *repudiate* the signature. Since signers are often in some way liable for the statements they sign, this will allow a dishonest signer to avoid the liability. In fact, a dishonest signer may not be using weak methods, but merely be able to claim that the methods were weak by claiming, for example, that the key was leaked or that the random number generator was weak.

Systems in which this is a concern should consider what conditions may be accepted as a basis for repudiation, and then ensure that none of the conditions is present before accepting a signature. For example, if repudiation is possible on the basis that the key size was too small, systems should set policies by which signatures with keys under a certain size are not accepted. If repudiation is possible on the basis that a key was not authentic or invalid, systems should ensure authenticity and validity of keys and parameters (see D.3.2, D.3.3, D.5.2.4, and D.5.2.5). If repudiation is possible on the basis that the implementation was insecure, implementation validation may be appropriate (see D.7). Implementations may also ensure that users are not able to copy or view their own private keys and, thus, cannot deliberately leak them to other parties (see also D.3.1 for more on key-sharing).

System-wide policies may also help address this concern. For example, if signers generate their own keys, a system may choose to prohibit repudiation on the basis that a key was invalid, reasoning that honest signers should not generate invalid keys in the first place. They may also choose to prohibit any repudiation as long as public keys are securely associated with the signers, by asserting that it is the signers' responsibility to ensure that their keys are secure. In the latter case, a signer has a motivation to perform key validation after key generation, if there is any risk that the keys were not generated correctly (e.g., an error occurred).

When performing security analysis of a system that uses signatures, one needs to take into consideration not only adversaries whose goal it is to forge signatures, but also adversaries whose goal it is to repudiate them.

D.5.2.4 Authentication of ownership

If it is desired to verify that a particular party has the ability to generate a given signature (e.g., to prevent future repudiation of the signature by the party) (see D.5.2.3), then the party's ownership of the public key with which the signature is verified should be authenticated. In general, as noted in D.3.2, the party's possession of the corresponding private key should also be authenticated. The risk of not doing so is that an opponent may present a public key that is identical or related to some other party's public key, to make it appear as though a message was signed by the opponent, when in fact it was signed by the other party. Note, of course, that the opponent can always sign any message it knows with its own key. Thus, this risk is of concern in the following two cases:

- The signature is stored in such a way that it cannot be modified by the opponent, and the opponent is trying to show that it generated the signature.
- A portion of the message is secret, and the verifier is relying on the signature as assurance that the purported signer knows the secret. (Note that the use of signature schemes for verification of knowledge of secrets is not discussed in this standard, because such verification is usually accomplished by a protocol, rather than by a scheme. [Protocols and schemes are discussed in Clause 4.] Such use requires additional security analysis by the implementer.)

If the message on which the signature is computed includes the signer's name, an opponent cannot make it appear as though the opponent is the actual signer, because the opponent's name is different. (This is essentially how a signature-based proof of possession protocol works.)

D.5.2.5 Validation of domain parameters and keys

As discussed in D.3.3, using invalid keys or domain parameters as inputs to a primitive carries with it certain risks. Specifically, for the signature schemes, the risks are outlined below. Note that the risks will vary with the particular implementation.

- Failure of the implementation, resulting in an error state
- Potential signature forgery
- Repudiation of signatures based on the argument that the key is invalid and, hence, insecure

The risk of implementation failure can be addressed by appropriate implementation handling of error cases, or by ensuring the validity of the domain parameters and public keys as described in D.3.3.

The risk of signature forgery can be mitigated by ensuring the validity of the parameters and keys. However, validation does not address certain other risks. A key may be valid and still be insecure (e.g., if the random number generation for the key generation was performed improperly, or if the private key is stored insecurely or deliberately disclosed). Therefore, an implementer should consider other ways in which the key may be insecure, and then decide how to appropriately mitigate the risk. FIPS PUB 140-1 implementation validation and random number generation are typical ways to address some of these concerns (see D.6.1 and D.7). Public-key and parameter validation will ensure that no invalid public keys and parameters are used to verify a signature.

The risk of repudiation and ways to address it are described in more detail in D.5.2.3.

D.5.2.6 Cryptoperiod and protection lifetime

In a signature scheme, if an opponent learns a signer's private key, the opponent can generate new signatures that can be verified with the corresponding public key. The longer the cryptoperiod of a private key in a signature scheme, the more data is available to a cryptanalyst. The longer the private key is not erased, the longer it is potentially vulnerable to physical compromise. However, if the private key is erased and its owner needs to repudiate a signature that was forged by an opponent, the private key is not available as evidence. (The owner may need to use the private key to show, for example, that the private key was somehow weak and, therefore, forgery was possible.)

If a verifier requires a secure timestamp (Section 13.8.1 of Menezes, van Oorschot, and Vanstone [B112]) on signed messages and only accepts signatures that have timestamps prior to a certain date, then the opponent is limited to generating new signatures before that date. Hence, if secure timestamps are used, the protection lifetime of the private key can be limited by a specific date, provided that the date is securely associated with the corresponding public key (see D.3.6). Note, however, that secure timestamping is more than just a date field added by the signer. The existence of the signature on the message at the claimed date must be independently confirmed by the verifier or by a third party.

D.5.3 Encryption schemes

Encryption schemes are generally used to provide confidentiality of data. A theoretical definition of "semantic security" (or, equivalently, "indistinguishability" or "polynomial security") is commonly used as the basis for the meaning of "confidentiality" (see Goldwasser and Micali [B61], Section 8.7 of Menezes, van Oorschot, and Vanstone [B112], Micali, Rackoff, and Sloan [B114], and Yao [B151]). Semantic security provides that it should be computationally infeasible to recover any information about the plaintext (except its length) from the ciphertext without knowing the private key. This implies, in particular, that it should also be infeasible to find out whether two ciphertexts correspond to the same, or in some way related, plaintexts, or whether a given ciphertext is an encryption of a given plaintext.In addition, encryption schemes are sometimes used to provide integrity of data in the form of plaintext-awareness—an assurance that nobody could generate a valid ciphertext without knowing the corresponding plaintext (Bellare and Rogaway [B18]). (Data integrity in this context is different than for a signature scheme, where it ensures that a message has not been modified since it was signed by an identified signer. Here, data integrity ensures that a message has not been modified since it was encrypted by a possibly unidentified sender. Plaintext-awareness means that the sender "knew" the message.) In particular, plaintext-awareness implies security against an adaptively chosen ciphertext attack (an attack in which the opponent requests decryptions of specially constructed ciphertexts in order to be able to decrypt other ciphertexts). The encryption scheme in this standard is believed to provide plaintext-awareness and, hence, security against adaptively chosen ciphertext attack.

See Bellare et al. [B17] for more on relations among notions of security and modes of attack for encryption schemes.

D.5.3.1 Primitives

There is only one pair of encryption and decryption primitives for encryption schemes in this standard

— IFEP-RSA and IFDP-RSA

Although the primitives are not known to be equivalent to the underlying problem of factoring, attacking the underlying problem is the best currently known method for attacking the primitives.

D.5.3.2 Encoding method

The only recommended encoding method for encryption is EME1.

An encoding method for an encryption scheme in this standard (i.e., IFES) should have the following properties, stated informally:

- Representatives of different messages should be unrelated.
- The encoding method, through incorporation of randomness or otherwise, should ensure that representatives of the same message produced at different times are unrelated, and that it is difficult to determine (without the private key), given a ciphertext and a plaintext, whether the ciphertext is an encryption of the plaintext.
- Message representatives should have some verifiable structure, so that it is difficult to produce a ciphertext that decrypts to a valid message representative (plaintext awareness).
- The encoding method should have minimal mathematical structure that could interact with the encryption primitive (e.g., the encoding method should not be multiplicative, because IFEP-RSA is).

EME1 is considered to have these properties for IFES. (One motivation for using EME1 as opposed to other encoding methods is a result by Bleichenbacher [B23].) It also has the additional property of securely associating optional encoding parameters with a message, where the encoding parameters are not encrypted but are protected from modification (see D.5.3.3).

The security of EME1 depends on the underlying hash function, mask generation function, and random number generator for generating the seed. SHA-1 and RIPEMD-160 are allowed for the hash function, and MGF1 (with SHA-1 or RIPEMD-160) is allowed for the mask generation function. See D.6 for recommendations on random number generation. The length of the hash function output governs the security of the encoding method.

A further discussion of desired theoretical properties may be found in Bellare and Rogaway [B18].

D.5.3.3 Encoding parameters

The purpose of encoding parameters in an encryption scheme is to associate control information with a message. The parameters should have an unambiguous interpretation. Their content depends on the implementation, and they may be omitted (i.e., the parameters may be an empty string). The parameters are not encrypted by the encryption scheme, but are securely associated with the ciphertext and protected from modification. Whether they have been modified or not is verified during the decryption process. See Johnson and Matyas [B86] for information on the encoding parameters.

For EME1, the length of the encoding parameters can vary from one encryption operation to another.

D.5.3.4 Authentication of ownership

If it is desirable to verify that a particular party has the ability to decrypt a given ciphertext, the party's ownership of the public key with which the ciphertext is produced should be authenticated.

D.5.3.5 Validation of domain parameters and keys

As discussed in D.3.3, using invalid keys as inputs to a primitive carries with it certain risks. Specifically, for the encryption schemes, the risks are outlined below. Note that the risks will vary with the particular implementation

- Failure of the implementation, resulting in an error state
- Loss of confidentiality of the data

The risk of implementation failure can be addressed by appropriate implementation handling of error cases, or by ensuring the validity of the domain parameters and public keys as described in D.3.3.

The risk of loss of confidentiality can be mitigated by ensuring the validity of the parameters and keys. However, validation does not address certain other risks. A key may be valid and still be insecure (e.g., if the random number generation for the key generation was performed improperly, or if the private key is stored insecurely or deliberately disclosed). Even if the key is secure, the recipient may (because of insecure implementation or deliberately) leak the content of the message. Therefore, an implementer should consider other ways in which the key or the message content may be insecure, and then decide how to appropriately mitigate the risk. FIPS PUB 140-1 implementation validation and random number generation are typical ways to address some of these concerns (see D.6.1 and D.7). Public-key validation will ensure that no messages are encrypted with invalid keys. Indeed, public-key validation is one of the precautions a message sender may apply directly, whereas the other countermeasures mentioned here require the sender to rely on representations by another party (e.g., that the implementation is secure).

D.5.3.6 Cryptoperiod and protection lifetime

In an encryption scheme, if an opponent learns a recipient's private key, then the opponent can recover all messages encrypted with the corresponding public key. The longer the cryptoperiod of a public key in an encryption scheme, the more messages are potentially vulnerable to recovery. The longer the private key is not erased, the longer it is potentially vulnerable to physical compromise. However, once the private key is erased, no data encrypted with the corresponding public key can be decrypted.

The protection lifetime in an encryption scheme should be determined by how long the data encrypted with the public key remains sensitive.

D.6 Random number generation

As indicated throughout this standard, and this annex in particular, generation of random numbers (or, more generally, random bit strings) is a tool commonly used in cryptography. Proper generation of random bit strings for cryptographic purposes is essential to security, particularly because secrets, such as private keys, are commonly derived from such strings. Failure of a random number generator (resulting in, for example, generation of predictable or repeating keys) is likely to make a system extremely vulnerable to an opponent. As opposed to random number generation in some other settings, where security is not a concern, cryptographic random number generation cannot generally be accomplished by simply invoking the software subroutine "random" in one's favorite software library.

This subclause describes some of the security concerns related to generating random bit strings. A more detailed exposition is given in [Chapter 5, B112]. ISO and ANSI have recently started studying random number generation. (The ANSI X9F1 Working Group has a work item numbered X9.82; in the ISO, ISO/IEC JTC1 SC 27 WG 2 is studying the issue.) The reader may find the future results of this work useful.

Random strings in cryptography are commonly generated by the following method: collect enough data from a random source to get a seed for a pseudo-random bit generator, and then use the pseudo-random bit generator to get the string of the desired length. D.6.1 and D.6.2 address both steps of this two-step process.

D.6.1 Random seed

An opponent can be assumed to have full knowledge of the pseudo-random bit generator (see D.2), except for its seed input. The random seed is, thus, the only component of the random number generation process that the opponent may not know. Therefore, implementers should ensure that the seed is collected from a source that has sufficient variability and cannot be accessed, guessed, or influenced in a predictable way by the opponent. In short, the seed should be unpredictable. If the result of the random number generation process is to be kept secret, the seed should be kept secret as well.

Ideally, each bit produced by the random source should be evenly distributed and independent of all the other bits. In such a case, there are 2^n equally probable random seeds of length n, and the opponent has no advantage in guessing the seed. However, in reality, this is often not the case. As random sources usually have biased and correlated bits, opponents often have a better chance of guessing the seed of length n than 1 in 2^n. When evaluating a random source, it is essential to establish that an opponent with appropriately large computational resources, and full knowledge of the nature of the random source, has a negligible probability of guessing a seed output by the random source. Note that the opponent should be considered to have as many attempts at guessing as the opponent's potential computational resources would allow.

Informally, variability of a random source is measured in bits of randomness (also called "bits of variability") per bits of output. For example, if a random source is said to have variability of "one bit per byte," this generally means that there are about 2^k different strings of more or less equal probability of length $8k$ bits. Thus, in order to get about 160 bits of randomness from such a source, one needs 1280 bits of data.

If the random bits are biased or correlated, it is common to run them through a cryptographic hash function in order to "distill" the randomness. This method relies on the assumption that the cryptographic hash function will produce a distribution on the outputs that is reasonably close to uniform, because the hash function is not correlated to the biases of the random source. In the example above, the 1280 bits of data could be given as an input to, for example, SHA-1 to come up with a 160 bit output that has (presumably) about 160 bits of randomness. Note that the hash function output should not be longer that the number of bits of randomness believed to be provided by the data.

If the random seed needs to be kept secret, it is essential that the source of the random data be protected from eavesdropping. For example, if one is using the timing of a user's keystrokes as the source of the random data, it is important that the keystrokes cannot be observed by the opponent through, for example, a hidden camera installed in the room.

Whether or not the seed needs to be kept secret, it is also essential that the source of random data be protected from manipulation by the opponent. For example, if system loads or network statistics are used for random data, the opponent having access to the system or the network may be able to manipulate them in order to reduce their variability, even if not able to observe them directly.

A common precaution is to combine many available sources of randomness and to use a hash function, as described above, in order to "mix" them. If, for example, one collects enough data for 160 bits of variability, each from three different sources, then even if two of the sources fail, the remaining source will provide about 160 bits of variability in the hash function output. It is generally safer to collect more data rather than less data from each of the sources, even if only a short seed is needed.

Some sources of randomness that may be used include the following:

— Photon polarization detected at 45° out of phase.

— Elapsed time between emissions of particles from a radioactive source.

— Quantum effects in a semiconductor, such as a noisy diode or a noisy resistor.

— Frequency fluctuations of free-running oscillators.

— Fluctuations in the amount a metal insulator semiconductor capacitor is charged during a fixed period of time.

— Fluctuations in read times caused by air turbulence within a sealed disk drive dedicated to this task (Davis, Ihaka, and Fenstermacher [B44]).

— The difference between the output of two microphones in two different points in a room (provided their amplification is normalized to minimize the difference signal).

— The noise within the signal of a microphone or video-camera in a room (tends to provide less variability and is easier to predict and observe than the previous method).

— The electronic noise of an A-D converter with an unplugged microphone. (The variability depends heavily on the particular converter and environment.)

— Variation in mouse movements or keystrokes and their timing. (For example, the user may be asked to sign his or her name with the mouse; note that accessibility of correct high-resolution timing data for mouse movements and key strokes varies among different systems.)

— The system clock (which tends to provide very little variability and can be assumed to be known to the opponent, except for the last few digits, but may be used in combination with other sources). Note also that the resolution of the system clock that is available to the application varies among different systems.

— Operating system statistics that cannot be observed by a potential opponent (and tend to provide little variability).

When using any of these sources, it is essential not to overestimate the amount of variability that is inaccessible to the opponent. For example, if the user is asked to sign his or her name, the signature may be known to the opponent, and only the deviations from the usual signature should be considered to provide variability. Or if the room sounds in a particular frequency range are used as a source of randomness, the opponent may be able to inject noises in that frequency range in order to bias the result. A more detailed analysis of some of these, as well as other, sources is provided in Eastlake, Crocker, and Schiller [B53].

The quality of a particular source of randomness for generating random seeds should be tested by one or more statistical tests. While statistical tests do not provide a guarantee that the generator is “good,” they help detect some weaknesses the generator may have. A number of tests are described in Section 5.4 of Menezes, van Oorschot, and Vanstone [B112] and Section 4.11.1 of FIPS PUB 140-1 [B54]. (FIPS PUB 140-1 [B54] also describes a continuous random number generator test in Section 4.11.2.) A particular test that detects many weaknesses is given in Maurer [B106] (see also Coron and Naccache [B41]). Note that while statistical tests may detect certain weaknesses, they do not guarantee unpredictability of the random bit source.

It is important not to confuse random-looking but public sources of information for secret sources of randomness. For example, USENET feeds, TV broadcasts, tables of pseudo-random numbers, or digits of π should be assumed to be accessible to the opponent and should not be relied upon as sources of random seeds.

D.6.2 Pseudo-random bit generation

Pseudo-random bit generators are deterministic algorithms that, given a seed as input, produce a pseudo-random bit string of a desired length. (Deterministic here means that they do not use any randomness other than that given by the input seed.) Because they are algorithms and should be assumed publicly known, the unpredictability of their output relies heavily on the unpredictability of the input seed.

A pseudo-random bit generator is generally used to output more bits than the seed provides; therefore, its outputs are not random, even if the seeds are random. In fact, the distribution of the outputs cannot possibly be close to uniform, because the number of possible outputs is limited by the number of possible seeds. For example, if a pseudo-random bit generator uses a 160 bit seed as input and returns a 300 bit output, only 2^{160} out of the possible 2^{300} (or one in 2^{140}) 300 bit strings can be potentially output by the generator.

Thus, the appropriate security condition to impose on a pseudo-random bit generator is not that it behave like a true random source. Rather, the condition is that it be indistinguishable from a true random source by any statistical test that can be computed with resources available to an opponent. More precisely, no computation that can be performed with a feasible amount of computational resources should be able to distinguish a truly random string from the output of a pseudo-random bit generator with probability significantly better than one half. (It is, of course, easy to distinguish with probability exactly one half by simply randomly guessing which one is the random string and which one is the output of the generator.) This condition is sometimes called *indistinguishability* (Yao [B151]).

A condition that is equivalent to indistinguishability is the following: given all but one bit of the output of the pseudo-random bit generator, it is infeasible to predict what the remaining bit is with probability significantly better than one half. In particular, it is infeasible to predict what the next bit of the output will be, even when given all the previous bits. This condition is sometimes called *unpredictability* (Blum and Micali [B25]).

It is essential that pseudo-random bit generators used in cryptography reasonably satisfy the above conditions. Some such generators are defined in ANSI X9.17-1985 [B3], Blum, Blum, and Shub [B24], FIPS PUB 186 [B56], Micali and Schnorr [B115]; see Sections 5.3 and 5.5 of Menezes, van Oorschot, and Vanstone [B112] for their descriptions. The formal basis for the belief of security varies according to the generator. The security of some (ANSI X9.17-1985 [B3] and FIPS PUB 186 [B56]) is based on heuristic assumptions, such as the unpredictability of outputs of DES or SHA-1 when a key or other input is unknown. The security of others (Blum, Blum, and Shub [B24] and Micali and Schnorr [B115]) is based on reduction from a hard problem, such as integer factorization. Some attacks on some of the above generators, and suggestions for improving pseudo-random generator design, are contained in Kelsey et al. [B91].

The above two conditions imply that it is infeasible to guess the seed given just the output of the generator. This is why pseudo-random generators may provide assurance that a set of parameters or keys was generated from a seed (rather than the seed reconstructed after the generation), as detailed in D.3.1 and D.4.

The above two conditions also imply that the output of the pseudo-random generator, if long enough, is very unlikely to repeat in any reasonable time. Thus, keys, if generated properly, are very unlikely to be repeated or accidentally shared among users.

Statistical tests described in the previous clause for testing the quality of the seed may also be used to test the quality of the pseudo-random bit generator. However, some generators commonly used outside of cryptography (such as the linear congruential generator) may pass many of the statistical tests, while being entirely predictable and, therefore, insecure for cryptographic purposes.

D.7 Implementation considerations

This standard (and this annex in particular) focuses on cryptographic methods and the algorithms used to implement them. Many engineering-related considerations are just as important in a secure implementation. This subclause gives a sampling of the problems an implementer needs to consider. It is not meant to be comprehensive, as solutions to these problems are outside the scope of this standard. For more on secure implementation, see FIPS PUB 180-1 [B54] and Book 2, Appendix P of SET Secure Electronic Transaction Specification [B104].

As with other security considerations, sources of the potential threats and their capabilities need to be examined. For example, opponents may be able to induce errors in systems by operating them under unexpected conditions, whether physical [such as change in temperature, power source characteristics, and electromagnetic radiation; e.g., TEMPEST issues (Anderson and Kuhn [B2])], or computational (such as illformed protocol messages or multiple requests overloading a server). Such errors may leak sensitive information. (See e.g., the fault-analysis attack of Boneh, DeMillo, and Lipton [B26], or the story of the "internet worm" in Hafner and Markoff [B69] and Kehoe [B90], or at http://www1.minn.net/~darbyt/worm/worm.html.) Implementations, therefore, should make sure that they handle unexpected or erroneous inputs and environments appropriately at all levels where an adversary may be present.

Implementations should also consider what information they may be giving to an adversary by indirect means. For example, by measuring the time a computation takes, an adversary may be able recover some information about a private key (see Dhem et al. [B45] and Kocher [B96]). By analyzing the error messages sent by an implementation, an adversary may also be able to recover information (e.g., Bleichenbacher [B23]). Therefore, an implementation of any operation that uses secrets (such as key agreement, signature production, or decryption) should limit the information it provides in case of error. Even partial information about a secret may lead to recovery of the entire secret (e.g., Boneh, Durfee, and Frankel [B28]).

Due consideration should also be given to protecting the secrets. They should be stored securely in such a way that unauthorized copying of even a portion of a secret is prevented. All the copies of the secrets should be accounted for. For example, care should be taken that, in a system with paging, a copy of a page containing the secret is not made on disk by the operating system without the knowledge of the implementation. Implementations should also be aware of the possibility of eavesdropping by, for example, the use of electromagnetic radiation emanating from a video monitor (Anderson and Kuhn [B2]).

The authenticity and validity of the machinery (whether hardware or software) performing the cryptographic operations need to be ensured. Otherwise, an adversary may be able to plant a so-called "Trojan horse" within the implementation by substituting pieces of code, parameters, keys, or root certificates that are imbedded in the implementation. Protecting the system itself from unauthorized modification is as important as protecting cryptographic parameters and keys. Implementation validation is addressed in more detail in FIPS PUB 140-1 [B54].

Annex E

(informative)

Formats

E.1 Overview

As outlined in Clause 4, the specifications presented in this standard are functional specifications, rather than interface specifications. Therefore, this standard does not specify how the mathematical and cryptographic objects (such as field elements, keys, or outputs of schemes) are to be represented for purposes of communication or storage. This annex provides references to other relevant standards and defines some recommended primitives for that purpose. While the use of this annex is optional, it is recommended for interoperability.

As octet strings are arguably the most common way to represent data electronically for purposes of communication, this annex focuses on representing objects as octet strings. One way to accomplish this is to represent data structures in Abstract Syntax Notation 1 (ASN.1) (see ISO/IEC 8824-1:1998 [B72], ISO/IEC 8824-2:1998 [B73], ISO/IEC 8824-3:1998 [B74], and ISO/IEC 8824-4:1998 [B75]), and then to use encoding rules, such as Basic Encoding Rules (BER), Distinguished Encoding Rules (DER), or others (see ISO/IEC 8825-1:1998 [B76] and ISO/IEC 8825-2:1998 [B77]) to represent them as octet strings. This annex does not specify ASN.1 constructs for use in this standard, because the generality of this standard would make such constructs very complex. It is likely that particular implementations utilize only a small part of the options available in this standard, and would be better served by simpler ASN.1 constructs. When the use of ASN.1 is desired, ASN.1 constructs defined in the following standards or draft standards may be adapted for use:

- ANSI X9.42 [ANS98b] for DL key agreement
- ANSI X9.63 [ANS98f] for EC key agreement and EC encryption for key transport
- ANSI X9.57 [ANS97c] for DL DSA signatures
- ANSI X9.62 [ANS98e] for EC DSA signatures
- ANSI X9.31 [ANS98a] for IF signatures
- ANSI X9.44 [ANS98c] for IF encryption for key transport

E.2 gives recommendations on representing basic mathematical objects as octet strings, and E.3 gives recommendations on representing the outputs of encryption and signature schemes as octet strings.

E.2 Representing basic data types as octet strings

When integers, finite field elements, elliptic curve points, or binary polynomials need to be represented as octet strings, it should be done as described in this annex. Other primitives for converting between different data types (including bit strings) are defined in Clause 5.

E.2.1 Integers (I2OSP and OS2IP)

Integers should be converted to/from octet strings using primitives I2OSP and OS2IP, as defined in 5.5.3.

E.2.2 Finite field elements (FE2OSP and OS2FEP)

Finite field elements should be converted to/from octet strings using primitives FE2OSP and OS2FEP, as defined in 5.5.4.

E.2.3 Elliptic curve points (EC2OSP and OS2ECP)

Elliptic curve points should be converted to/from octet strings using primitives EC2OSP and OS2ECP, as defined below.

An elliptic curve point P (which is not the point at infinity $\mathcal{O}$) can be represented in either *compressed* or *uncompressed* form. (For internal calculations, it may be advantageous to use other representations; e.g., the projective coordinates of A.9.6. Also see A.9.6 for more information on point compression.) The uncompressed form of P is given simply by its two coordinates. The compressed form is presented below. The octet string format is defined to support both compressed and uncompressed points.

E.2.3.1 Compressed elliptic curve points

The *compressed form* of an elliptic curve point P $\neq \mathcal{O}$ is the pair $(x_{P\,,}\ \tilde{y}_P)$, where x_P is the x-coordinate of P, and $\tilde{y}_P$ is a bit that is computed as follows:

a) If the field size q is an odd prime, then $\tilde{y}_P = y_P \bmod 2$ (where y_P is taken as an integer in $[0, q - 1]$ and not as a field element). Put another way, $\tilde{y}_P$ is the rightmost bit of y_P.

b) If the field size q is a power of 2 and $x_P = 0$, then $\tilde{y}_P = 0$.

c) If the field size q is a power of 2 and $x_P \neq 0$, then $\tilde{y}_P$ is the rightmost bit of the field element $y_P x_P^{-1}$.

Rules b) and c) apply for any of the basis representations given in 5.3.2.

Procedures for *point decompression* (i.e., recovering y_P given x_P and $\tilde{y}_P$) are given in A.12.8 (for q, which is an odd prime) and A.12.9 (for q, which is a power of two).

E.2.3.2 Elliptic curve points as octet strings—EC2OSP and OS2ECP

The point $\mathcal{O}$ should be represented by an octet string containing a single 0 octet. The rest of this subclause discusses octet string representation of a point $P \neq \mathcal{O}$. Let the x-coordinate of P be x_P and the y-coordinate of P be y_P. Let $(x_p,\ \tilde{y}_P)$ be the compressed representation of P. (See E.2.3.1 for information on point compression.)

An octet string PO representing P should have one of the following three formats: *compressed*, *uncompressed*, or *hybrid*. (The hybrid format contains information of both compressed and uncompressed form.) PO should have the following general form

$$PO = PC \parallel X \parallel Y$$

where

PC is a single octet of the form $00000UC\tilde{Y}$ defined as follows:

— Bit U is 1 if the format is uncompressed or hybrid; 0 otherwise.

— Bit C is 1 if the format is compressed or hybrid; 0 otherwise.

— Bit $\tilde{Y}$ is equal to the bit $\tilde{y}_P$ if the format is compressed or hybrid; 0 otherwise.

X is the octet string of length $\lceil \log_{256} q \rceil$ representing x_P according to FE2OSP (see 5.5.4).

Y is the octet string of length $\lceil \log_{256} q \rceil$ representing y_P of P according to FE2OSP (see 5.5.4) if the format is uncompressed or hybrid; Y is an empty string if the format is compressed.

The primitive that converts elliptic curve points to octet strings is called the Elliptic Curve Point to Octet String Conversion Primitive, or EC2OSP. It takes an elliptic curve point P, the size q of the underlying field, and the desired format (compressed, uncompressed, or hybrid) as input and outputs the corresponding octet string.

The primitive that converts octet strings to elliptic curve points is called the Octet String to Elliptic Curve Point Conversion Primitive, or OS2ECP. It takes the octet string and the field size q as inputs and outputs the corresponding elliptic curve point, or "error." It should use OS2FEP to get x_P. It should use OS2FEP to get y_P if the format is uncompressed. It should use point decompression (see E.2.3.1) to get y_P if the format is compressed. It can get y_P by either of these two means if the format is hybrid. It should output "error" in the following cases:

— If the first octet is 00000000 and the octet string length is not 1

— If the first octet is 00000100, 00000110, or 00000111 and the octet string length is not $1+ 2 \lceil \log_{256} q \rceil$

— If the first octet is 00000010, 00000011 and the octet string length is not $1 + \lceil \log_{256} q \rceil$

— If the first octet is any value other than the six values listed above

— If an invocation of OS2FEP outputs "error"

— If an invocation of the point decompression algorithm outputs "error"

NOTE—The first five bits of the first octet PC are reserved and may be used in future formats defined in an amendment to, or in future version of, this standard. It is essential that they be set to zero and checked for zero in order to distinguish this format from other formats.

E.2.4 Polynomials over GF (2) (PN2OSP and OS2PNP)

Polynomials over GF (2) should be converted to/from octet string using primitives PN2OSP and OS2PNP, as defined below.

The coefficients of a polynomial $p(t)$ over GF (2) are elements of GF (2) and are, therefore, represented as bits: the element zero of GF (2) is represented by the bit 0, and the element 1 of GF (2) is represented by the bit 1 (see 5.5.4). Let e be the degree of $p(t)$ and

$$p(t) = a_e\, t^e + a_{e-1}\, t^{e-1} + \ldots + a_1\, t + a_0$$

where $a_e = 1$. To represent $p(t)$ as an octet string, the bits representing its coefficients should be concatenated into a single bit string: $a = a_e \,\|\, a_{e-1} \,\|\, \ldots \,\|\, a_1 \,\|\, a_0$. The bit string a should then be converted into an octet string using BS2OSP (see 5.5.2).

The primitive that converts polynomials over GF (2) to octet strings is called the Polynomial to Octet String Conversion Primitive, or PN2OSP. It takes a polynomial $p(t)$ as input and outputs the octet string.

The primitive that converts octet strings to polynomials over GF (2) is called Octet String to Polynomial Conversion Primitive or OS2PNP. It takes the octet string as input and outputs the corresponding polynomial over GF (2). Let l be the length of the input octet string. OS2PNP should use OS2BSP (see 5.5.2) with the octet string and the length $8l$ as inputs. It should output "error" if OS2BSP outputs "error." The output of

OS2BSP should be parsed into 8l coefficients, one bit each. Note that at most, seven leftmost coefficients may be zero. The leftmost zero coefficients should be discarded.

E.3 Representing outputs of schemes as octet strings

The signature and encryption schemes in this standard do not produce octet strings as their outputs. When a scheme output (e.g., a signature) needs to be represented as an octet string, it may be done as defined in this subclause.

E.3.1 Output data format for DL/ECSSA

For DL/ECSSA, the output of the signature generation function (see 10.2.2) is a pair of integers (c, d). Let r denote the order of the generator (g or G) in the DL or EC settings (see 6.1 and 7.1), and let $l = \lceil \log_{256} r \rceil$ (i.e., l is the length of r in octets). The output (c, d) may be formatted as an octet string as follows: convert the integers c and d to octet strings C and D, respectively, of length l octets each, using the primitive I2OSP, and output the concatenation $C \| D$. To parse the signature, split the octet string into two components C and D, of length l each, and convert them to integers c and d, respectively, using OS2IP. Note that it is essential that both C and D be of length l, even if it means that they have leading zero octets.

NOTE—The output of DL/ECSSA may also be formatted according to the following method, described in more detail in ANSI X9.57-1997 [B6] and ANSI X9.62-1998 [B11]: combine c and d into an ASN.1 structure (ISO/IEC 8824-1:1998 [B72]), and encode the structure using some encoding rules, such as BER or DER (ISO/IEC 8825-1:1998 [B76]).

E.3.2 Output data format for IFSSA

For IFSSA, the output of the signature generation function (see 10.3.2) is an integer s. Let $k = \lceil \log_{256} n \rceil$ denote the length of the modulus n in octets in the IF setting (see 8.1). The output s may be formatted as an octet string by simply converting it to an octet string of length k octets using the primitive I2OSP.

E.3.3 Output data format for IFES

For IFES, the output of the encryption function (see 11.2.2) is an integer g. Let $k = \lceil \log_{256} n \rceil$ denote the length of the modulus n in octets in the IF setting (see 8.1). The output g may be formatted as an octet string by simply converting it to an octet string of length k octets using the primitive I2OSP.

Annex F

(informative)

Bibliography

[B1] Adams, C., and Myers, M. “Certificate Management Message Formats,” Internet Engineering Task Force (IETF), PKIX Working Group, work in progress. Available at http://www.ietf.org/ids.by.wg/pkix.html.

[B2] Anderson, R., and Kuhn, M. “Soft Tempest: Hidden Data Transmission Using Electromagnetic Emanations,” D. Aucsmith, Ed., Second International Workshop on Information Hiding (IH ’98), *Lecture Notes in Computer Science 1525* (1998), Springer-Verlag.

[B3] ANSI X9.17-1985, *Financial Institution Key Management* (wholesale).

[B4] ANSI X9.30:1-1997, *Public-key Cryptography for the Financial Services Industry: Part 1: The Digital Signature Algorithm (DSA)* (revision of X9.30:1-1995).

[B5] ANSI X9.30:2-1997, *Public-key Cryptography for the Financial Services Industry: Part 2: The Secure Hash Algorithm (SHA-1)* (revision of X9.30:2-1993).

[B6] ANSI X9.57-1997, *Public-key Cryptography for the Financial Services Industry: Certificate Management*.

[B7] ANSI X9.31-1998, *Digital Signatures Using Reversible Public-key Cryptography for the Financial Services Industry (rDSA).*

[B8] ANSI X9.42, *Public-key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Diffie-Hellman and MQV Algorithms* (draft), 1998.

[B9] ANSI X9.44, *Key Management Using Reversible Public-key Cryptography for the Financial Services Industry* (draft), 1998.

[B10] ANSI X9.52-1998, *Cryptography for the Financial Services Industry: Triple Data Encryption Algorithm Modes of Operation.*

[B11] ANSI X9.62-1998, *Public-key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA).*

[B12] ANSI X9.63, *Public-key Cryptography for the Financial Services Industry: Elliptic Curve Key Agreement and Transport Protocols* (draft), 1998.

[B13] ANSI X9.80, *Public-key Cryptography for the Financial Services Industry: Prime Number Generation and Validation Methods* (draft), 1998.

[B14] ANSI X9.TG-17, *Public-key Cryptography for the Financial Services Industry: Technical Guideline on Elliptic Curve Arithmetic* (to appear).

[B15] Ash, D., Blake, I., and Vanstone, S. “Low Complexity Normal Bases,” *Discrete Applied Mathematics 25* (1989), pp. 191-210.

[B16] Atkin, O. “Square Roots and Cognate Matters Modulo $p = 8n + 5$,” Internet communication to number theory mailing list, November 1992, archived at http://listserv.nodak.edu/scripts/wa.exe?A2=ind9211&L=nmbrthry&O=T&P=562.

[B17] Bellare, M., Desai, D., Pointcheval, D., and Rogaway, P. “Relations Among Notions of Security for Public-key Encryption Schemes,” H. Krawczyk, Ed., *Advances in Cryptology*, CRYPTO ‘98, *Lecture Notes in Computer Science 1462* (1998), Springer-Verlag, pp. 26-45. Full version available at http://www-cse.ucsd.edu/users/mihir/.

[B18] Bellare, M., and Rogaway, P. “Optimal Asymmetric Encryption—How to Encrypt with RSA,” A. De Santis, Ed., *Advances in Cryptology*, EUROCRYPT ‘94, *Lecture Notes in Computer Science 950* (1995), Springer-Verlag, pp. 92-111. Revised version available at http://www-cse.ucsd.edu/users/mihir/.

[B19] Bellare, M., and Rogaway, P. “The Exact Security of Digital Signatures: How to Sign with RSA and Rabin,” U. M. Maurer, Ed., *Advances in Cryptology*, EUROCRYPT ‘96, *Lecture Notes in Computer Science 1070* (1996), Springer-Verlag, pp. 399-416. Revised version available at http://www-cse.ucsd.edu/users/mihir/.

[B20] Berlekamp, E. *Algebraic Coding Theory*, McGraw-Hill, 1968, pp. 36-44.

[B21] Blake-Wilson, S., Johnson, D., and Menezes, A. “Key Agreement Protocols and Their Security Analysis,” M. Darnell, Ed., Cryptography and Coding: Sixth IMA International Conference, *Lecture Notes in Computer Science 1355* (1997), Springer-Verlag, pp. 30-45. Full version available at http://www.cacr.math.uwaterloo.ca/.

[B22] Blake-Wilson, S., and Menezes, A. “Unknown Key-Share Attacks on the Station-to-Station (STS) Protocol,” H. Imai and Y. Zheng, eds., Public-key Cryptography: Second International Workshop on Practice and Theory in Public-key Cryptography, PKC ’99, *Lecture Notes in Computer Science 1560* (1999), pp. 154-170. Also available as Technical Report CORR 98-42 at http://www.cacr.math.uwaterloo.ca/.

[B23] Bleichenbacher, D. “Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1,” H. Krawczyk, Ed., *Advances in Cryptology*, CRYPTO ‘98, *Lecture Notes in Computer Science 1462* (1998), Springer-Verlag, pp. 1-12.

[B24] Blum, L., Blum, M., and Shub, M. “A Simple Unpredictable Pseudo-random Number Generator,” *SIAM Journal on Computing 15* (1986), pp. 364-383.

[B25] Blum, M., and Micali, S. “How to Generate Cryptographically Strong Sequences of Pseudo-random Bits,” *SIAM Journal on Computing 13* (1984), pp. 850-864.

[B26] Boneh, D., DeMillo, R. A., and Lipton, R. J. “On the Importance of Checking Cryptographic Protocols for Faults,” W. Fumy, Ed., *Advances in Cryptology*, EUROCRYPT ‘97, *Lecture Notes in Computer Science 1223* (1997), Springer-Verlag, pp. 37-51.

[B27] Boneh, D., and Durfee, G. “Cryptoanalysis of RSA with Private Key *d* Less Than $N^{0.292}$,” J. Stern, Ed., *Advances in Cryptology*, EUROCRYPT ‘99, *Lecture Notes in Computer Science 1592* (1999), Springer-Verlag, pp. 1-11.

[B28] Boneh, D., Durfee, G., and Frankel, Y. “An attack on RSA Given a Small Fraction of the Private Key Bits,” K. Ohta and D. Pei, eds, *Advances in Cryptology*, ASIACRYPT ’98, *Lecture Notes in Computer Science 1514* (1998), Springer-Verlag, pp. 25-34.

[B29] Boneh, D., and Lipton, R. “Algorithms for Black Box Fields and Their Application to Cryptography,” N. Koblitz, Ed., *Advances in Cryptology*, CRYPTO ‘96, *Lecture Notes in Computer Science 1109* (1996), Springer-Verlag, pp. 283-297.

[B30] Boneh, D., and Venkatesan, R. “Breaking RSA May Not Be Equivalent to Factoring,” K. Nyberg, Ed., *Advances in Cryptology*, EUROCRYPT ‘98, *Lecture Notes in Computer Science 1403* (1998), Springer-Verlag, pp. 59-71.

[B31] Brillhart, J., Lehmer, D. H., Selfridge, J. L., Tuckerman, B., and Wagstaff, S. S. “Factorizations of b^n ± 1, b = 2,3, 5,6,7,10,11,12, up to High Powers,” 2nd ed., American Math. Soc., 1988.

[B32] Buchmann, J., Loho, J., and Zayer, J. “An Implementation of the General Number Field Sieve,” D. R. Stinson, Ed., *Advances in Cryptology*, CRYPTO ‘93, *Lecture Notes in Computer Science 773* (1994), Springer-Verlag, pp. 159-165.

[B33] Buell, D. *Binary Quadratic Forms: Classical Theory and Modern Computations*, Springer-Verlag, 1989.

[B34] Buhler, J. P., Lenstra, H. W., Jr., and Pomerance, C. “Factoring Integers with the Number Field Sieve,” A. K. Lenstra and H.W. Lenstra, Jr., eds., *The Development of the Number Field Sieve, Lecture Notes in Mathematics 1554* (1993), Springer-Verlag, pp. 50-94.

[B35] Burthe, R. J., Jr. “Further Investigations with the Strong Probable Prime Test,” *Mathematics of Computation 65* (1996), pp. 373-381.

[B36] Chen, L., and Williams, C. “Public-key Sterilization,” unpublished draft, August 1998.

[B37] Chen, M., and Hughes, E. “Protocol Failures Related to Order of Encryption and Signature: Computation of Discrete Logarithms in RSA Groups,” C. Boyd and E. Dawson, eds., Third Australian Conference on Information Security and Privacy, ACISP ’98, *Lecture Notes in Computer Science 1438* (1998).

[B38] Chudnovsky, D.V., and Chudnovsky, G.V. “Sequences of Numbers Generated by Addition in Formal Groups and New Primality and Factorizations Tests,” *Advances in Applied Mathematics 7* (1987), pp. 385-434.

[B39] Coppersmith, D., Franklin, M., Patarin, J., and Reiter, M. “Low-exponent RSA with Related Messages,” U. M. Maurer, Ed., *Advances in Cryptology*, EUROCRYPT ‘96, *Lecture Notes in Computer Science 1070* (1996), Springer-Verlag, pp. 1-9.

[B40] Coppersmith, D., Halevi, S., and Jutla, C. “ISO 9796-1 and the New Forgery Strategy,” (working draft). Presented at the Rump Session of CRYPTO ‘99. Available at
http://grouper.ieee.org/groups/1363/.

[B41] Coron, J-S., and Naccache, D. “An Accurate Evaluation of Maurer’s Universal Test,” S. Tavares and H. Meijer, Eds., Selected Areas in Cryptography, SAC ’98, *Lecture Notes in Computer Science 1556* (1998), Springer-Verlag.

[B42] Coron J-S., Naccache, D., and Stern, J. P. “On the Security of RSA Padding,” M. J. Wiener, Ed., *Advances in Cryptology*, CRYPTO ‘99, *Lecture Notes in Computer Science 1666* (1999), Springer-Verlag, pp. 1-18.

[B43] Damgard, I., Landrock, P., and Pomerance, C. “Average Case Error Estimates for the Strong Probable Prime Test,” *Mathematics of Computation 61* (1993), pp. 177-194.

[B44] Davis, D., Ihaka, R., and Fenstermacher, P. “Cryptographic Randomness from Air Turbulence in Disk Drives,” Yvo G. Desmedt, Ed., *Advances in Cryptology*, CRYPTO ‘94, *Lecture Notes in Computer Science 839* (1994), Springer-Verlag, pp. 114-120.

[B45] Dhem, J-F., Koeune, F., Leroux, P-A., Mestré, P., Quisquater, J-J., and Willems, J-L.”A Practical Implementation of the Timing Attack,” J.-J. Quisquater and B. Schneier, Eds., CARDIS ’98 Third Smart Card Research and Advanced Application Conference, *Lecture Notes in Computer Science 1820*, Springer-Verlag (to appear).

[B46] Diffie, W. “The First Ten Years of Public-key Cryptology,” *Proceedings of the IEEE 76* (1988), pp. 560-577.

[B47] Diffie, W., and Hellman, M. “New Directions in Cryptography,” *IEEE Transactions on Information Theory 22* (1976), pp. 644-654.

[B48] Diffie, W., van Oorschot, P. C., and Wiener M. J. “Authentication and Authenticated Key Exchanges,” *Designs, Codes, and Cryptography 2* (1992), pp. 107-125.

[B49] Dobbertin, H., Bosselaers, A., and Preneel, B. “RIPEMD-160: a Strengthened Version of RIPEMD,” D. Gollmann, Ed., Fast Software Encryption, Third International Workshop, *Lecture Notes in Computer Science 1039* (1996), Springer-Verlag, pp. 71-82. A corrected and updated version is available at http://www.esat.kuleuven.ac.be/~bosselae/ripemd160.html.

[B50] Dodson, B., and Lenstra, A. K. “NFS with Four Large Primes: An Explosive Experiment,” D. Coppersmith, Ed., *Advances in Cryptology*, CRYPTO ‘95, *Lecture Notes in Computer Science 963* (1995), Springer-Verlag, pp. 372-385.

[B51] Dusse, S., Hoffman, P., Ramsdell, B., Lundblade, L., and Repka, L. “RFC2311: S/MIME Version 2 Message Specification,” Internet Activities Board, March 1998. Available at http://www.rfc-editor.org/. See also http://www.ietf.org/html.charters/smime-charter.html and http://www.ietf.org/ids.by.wg/smime.html for latest developments and drafts.

[B52] Dusse, S., Hoffman, P., Ramsdell, B., and Weinstein, J. “RFC2312: S/MIME Version 2 Certificate Handling,” Internet Activities Board, March 1998. Available at http://www.rfc-editor.org/. See also http://www.ietf.org/html.charters/smime-charter.html and http://www.ietf.org/ids.by.wg/smime.html for latest developments and drafts.

[B53] Eastlake, D., Crocker, S., and Schiller, J. “RFC1750: Randomness Recommendations for Security,” Internet Activities Board, December 1994. Available at http://www.rfc-editor.org/.

[B54] FIPS PUB 140-1, *Security Requirements for Cryptographic Modules*, Federal Information Processing Standards Publication 140-1, U.S. Department of Commerce, National Institute of Standards and Technology (NIST), National Technical Information Service, Springfield, Virginia, April, 1994 (supersedes FIPS PUB 140). Available at http://www.itl.nist.gov/fipspubs/.

[B55] FIPS PUB 180-1, *Secure Hash Standard*, Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce, National Institute of Standards and Technology (NIST), National Technical Information Service, Springfield, Virginia, April 1995 (supersedes FIPS PUB 180). Available at http://www.itl.nist.gov/fipspubs/.

[B56] FIPS PUB 186, *Digital Signature Standard*, Federal Information Processing Standards Publication 186, U.S. Department of Commerce, National Institute of Standards and Technology (NIST), National Technical Information Service, Springfield, Virginia, 1994. Available at http://www.itl.nist.gov/fipspubs/.

[B57] Gallant, R., Lambert, R., and Vanstone, S. "Improving the Parallelized Pollard Lambda Search on Binary Anomalous Curves," *Mathematics of Computation* (to appear).

[B58] Gennaro, R., Micciancio, D., and Rabin, T. "An Efficient Noninteractive Statistical Zero-knowledge Proof System for Quasi-safe Prime Products," *Proceedings of the Fifth ACM Conference on Computer and Communications Security* (CCS-5), 1998, pp. 67-72. Available at http://www.acm.org/pubs/articles/proceedings/commsec/288090/p67-gennaro/p67-gennaro.pdf.

[B59] Gilbert, H., Gupta, D., Odlyzko, A., and Quisquater, J-J. "Attacks on Shamir's 'RSA for Paranoids'," *Information Processing Letters*, Vol.68, no.4 (November 30, 1998), pp.197-199. Also available at http://www.research.att.com/~amo/doc/crypto.html.

[B60] Goldwasser, S., and Kilian, J. "Almost All Primes Can Be Quickly Certified," *Proceedings of the 18th Annual ACM Symposium on Theory of Computing* (1986), pp. 316-329.

[B61] Goldwasser, S., and Micali, S. "Probabilistic Encryption," *Journal of Computer and System Sciences 28* (1984), pp. 270-299.

[B62] Goldwasser, S., Micali, S., and Rivest, R. L. "A Digital Signature Scheme Secure Against Adaptive Chosen-message Attacks," *SIAM Journal on Computing 17* (1988), pp. 281-308.

[B63] Gordon, D. M. "Designing and Detecting Trapdoors for Discrete Log Cryptosystems," E. F. Brickell, Ed., *Advances in Cryptology*, CRYPTO '92, *Lecture Notes in Computer Science 740* (1993), Springer-Verlag, pp. 66-75.

[B64] Gordon, D. M. "Discrete Logarithms in *GF* (*p*) Using the Number Field Sieve," *SIAM Journal on Discrete Mathematics 6* (1993), pp. 124-138.

[B65] Gordon, D. M. "A Survey of Fast Exponentiation Methods," *Journal of Algorithms 27* (1998), pp. 129-146.

[B66] Gordon, D. M., and McCurley, K. S. "Massively Parallel Computations of Discrete Logarithms," E. F. Brickell, Ed., *Advances in Cryptology*, CRYPTO '92, *Lecture Notes in Computer Science 740* (1993), Springer-Verlag, pp. 312-323.

[B67] Goss, K. C. "Cryptographic Method and Apparatus for Public-key Exchange With Authentications," U. S. Patent 4,956,863, 11 Sept. 1990.

[B68] Gunther, C. G. "An Identity-based Key-exchange Protocol;" J-J. Quisquater and J. Vandewalle, Ed., *Advances in Cryptology*, EUROCRYPT '89, *Lecture Notes in Computer Science 434* (1990), Springer-Verlag, pp. 29-37.

[B69] Hafner, K., and Markoff, J. *Cyberpunk: Outlaws and Hackers on the Computer Frontier* (updated edition), Touchstone Books, 1995.

[B70] Hastad, J. "Solving Simultaneous Modular Equations of Low Degree," *SIAM Journal on Computing 17* (1988), pp. 336-341.

[B71] IEEE Standard Dictionary of Electrical and Electronics Terms, Sixth edition.

[B72] ISO/IEC 8824-1:1998, *Information Technology—Abstract Syntax Notation One (ASN.1): Specification of Basic Notation*. Equivalent to ITU-T Rec. X.680 (1997).

[B73] ISO/IEC 8824-2:1998, *Information Technology—Abstract Syntax Notation One (ASN.1): Information Object Specification*. Equivalent to ITU-T Rec. X.681 (1997).

[B74] ISO/IEC 8824-3:1998, *Information Technology—Abstract Syntax Notation One (ASN.1): Constraint Specification*. Equivalent to ITU-T Rec. X.682 (1997).

[B75] ISO/IEC 8824-4:1998, *Information Technology—Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 Specifications*. Equivalent to ITU-T Rec. X.683 (1997).

[B76] ISO/IEC 8825-1:1998, *Information Technology—ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER)*. Equivalent to ITU-T Rec. X.690 (1997).

[B77] ISO/IEC 8825-2:1998, *Information Technology—ASN.1 Encoding Rules: Specification of Packed Encoding Rules (PER)*. Equivalent to ITU-T Rec. X.691 (1997).

[B78] ISO/IEC 9796:1991, *Information Technology—Security techniques—Digital signature scheme giving message recovery.*

[B79] ISO/IEC 9796-4, *Information Technology—Security techniques—Digital signature schemes giving message recovery—Part 4: Methods based on the Discrete Logarithm* (draft), 1998.

[B80] ISO/IEC DIS 14888-3, *Information technology—Security techniques—Digital signature with appendix—Part 3: Certificate-based mechanisms*, Draft International Standard, 1998.

[B81] Itoh, T., Teechai, O., and Tsujii, S. “A Fast Algorithm for Computing Multiplicative Inverses in *GF* (2^t) Using Normal Bases,” *J. Society for Electronic Communications (Japan) 44* (1986), pp. 31-36.

[B82] ITU-T, Recommendation X.509 (1997 E), *Information Technology—Open Systems Interconnection—The Directory: Authentication Framework*, International Telecommunications Union, June 1997.

[B83] Jablon, D., “Strong Password-Only Authenticated Key Exchange,” *Computer Communication Review*, Vol. 26, no., 5, pp. 5-26, ACM, Oct. 1996. Available at http://www.IntegritySciences.com/links.html#Jab96.

[B84] Joye, M., and Quisquater, J-J. “On Rabin-type Signatures” (working draft). Presented at the Rump Session of CRYPTO ‘99. Available from http://grouper.ieee.org/groups/1363/.

[B85] Johnson, D. B. Unpublished communication to the ANSI X9F1 and IEEE P1363 Working Groups.

[B86] Johnson, D. B., and Matyas, S. M. “Asymmetric Encryption: Evolution And Enhancements,” *CryptoBytes*, Vol. 2, no. 1 (Spring 1996), RSA Laboratories, ftp://ftp.rsa.com/pub/cryptobytes/crypto2n1.pdf.

[B87] Joye, M., and Quisquater, J-J. “Efficient Computation of Full Lucas Sequences,” *Electronics Letter*s, Vol. 32 (1996), pp. 537-538. Corrected version available at http://www.dice.ucl.ac.be/crypto/publications.html.

[B88] Kaliski, B. S., Jr., “Compatible Cofactor Multiplication for Diffie-Hellman Primitives,” *Electronics Letters*, Vol. 34, no. 25 (December 10, 1998), pp. 2396-2397.

[B89] Kaliski, B. S., Jr., “An Unknown Key Share Attack on the MQV Key Agreement Protocol,” submitted to *ACM Transactions on Information and Systems Security.* Also presented at the RSA Conference 2000 Europe, Munich, Germany, Apr. 2000.

[B90] Kehoe, B. P., *Zen and the Art of the Internet: A Beginner's Guide*, Fourth ed., Prentice Hall Computer Books, 1995.

[B91] Kelsey, J., Schneier, B., Wagner, D., and Hall, C. "Cryptanalytic Attacks on Pseudo-random Number Generators," S. Vaudenay, Ed., *Fast Software Encryption*, Fifth International Workshop Proceedings, *Lecture Notes in Computer Science 1372* (1998), Springer-Verlag, pp. 168-188.

[B92] Kerckhoffs, A. "La Cryptographie Militaire," *Journal des Sciences Militaires*, 9th Series, February 1883, pp. 161-191.

[B93] Knuth, D. E. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 2nd edition, Addison-Wesley, 1981, p. 379.

[B94] Koblitz, N. "Elliptic Curve Cryptosystems," *Mathematics of Computation 48* (1987), pp. 203-209.

[B95] Koblitz, N. *A Course in Number Theory and Cryptography*, 2nd edition, Springer-Verlag, 1994.

[B96] Kocher, P. C. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," N. Koblitz, Ed., *Advances in Cryptology*, CRYPTO '96, *Lecture Notes in Computer Science 1109* (1996), Springer-Verlag, pp. 104-113.

[B97] Kravitz, D. W. "Digital Signature Algorithm," U.S. Patent 5,231,668, July 1993.

[B98] Law, L., Menezes, A., Qu, M., Solinas, J., and Vanstone, S. "An Efficient Protocol for Authenticated Key Agreement," *Technical Report CORR 98-05*, Dept. of C & O, University of Waterloo, Canada, March 1998 (revised August 28, 1998). Available at http://www.cacr.math.uwaterloo.ca/.

[B99] Lay, G., and Zimmer, H. "Constructing Elliptic Curves With Given Group Order Over Large Finite Fields," Algorithmic Number Theory: First International Symposium, *Lecture Notes in Computer Science 877* (1994), Springer-Verlag, pp. 250-263.

[B100] Lehmer, D. H. "Computer Technology Applied to the Theory of Numbers," *Studies in Number Theory*, W. J. LeVeque, Ed., Mathematical Association of America, 1969.

[B101] Lenstra, H. W., Jr., "Factoring Integers With Elliptic Curves," *Annals of Mathematics 126* (1987), pp. 649-673.

[B102] Lim, C. H., and Lee, P. J. "A Key Recovery Attack on Discrete Log-based Schemes Using a Prime Order Subgroup," B. S. Kaliski, Jr., Ed., *Advances in Cryptology*, CRYPTO '97, *Lecture Notes in Computer Science 1294* (1997), Springer-Verlag, pp. 249-263.

[B103] Liskov, M., and Silverman, R. D. "A Statistical Limited-Knowledge Proof for Secure RSA Keys," submitted to *Journal of Cryptology*, 1998.

[B104] MasterCard International, Inc., and Visa International Service Association, *SET Secure Electronic Transaction Specification*, May 1997. Available at http://www.setco.org/.

[B105] Matsumoto, T., Takashima, Y., and Imai, H. "On Seeking Smart Public-key Distribution Systems," *The Transactions of the IECE of Japan E69* (1986), pp. 99-106.

[B106] Maurer, U. M. "A Universal Statistical Test for Random Bit Generators," A.J. Menezes and S. A. Vanstone, Eds., *Advances in Cryptology*, CRYPTO '90, *Lecture Notes in Computer Science 537* (1991), Springer-Verlag, pp. 409-420.

[B107] Maurer, U. M. “Fast Generation of Prime Numbers and Secure Public-key Cryptographic Parameters,” *Journal of Cryptology 8* (1995), pp. 123-155.

[B108] Menezes, A., Ed., *Applications of Finite Fields*. Kluwer Academic Publishers, 1993.

[B109] Menezes, A., *Elliptic Curve Public-key Cryptosystems*. Kluwer Academic Publishers, 1993.

[B110] Menezes, A., “Elliptic Curve Cryptosystems,” *CryptoBytes*, Vol. 1, no. 2 (Summer 1995), RSA Laboratories, ftp://ftp.rsa.com/pub/cryptobytes/crypto1n2.pdf.

[B111] Menezes, A., Okamoto, T., and Vanstone, S. “Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field,” *IEEE Transactions on Information Theory 39* (1993), pp. 1639-1646.

[B112] Menezes, A., van Oorschot, P., and Vanstone, S. *Handbook of Applied Cryptography*, CRC Press, Boca Raton, Florida, 1996.

[B113] Menezes, A., Qu, M., and Vanstone, S. “Some New Key Agreement Protocols Providing Implicit Authentication,” workshop record, 2nd Workshop on Selected Areas in Cryptography (SAC ’95), Ottawa, Canada, May 1995, pp. 22-32.

[B114] Micali, S., Rackoff, C., and Sloan, B. “The Notion of Security for Probabilistic Cryptosystems,” *SIAM Journal on Computing 17* (1988), pp. 412-426.

[B115] Micali, S., and Schnorr, C. P. “Efficient, Perfect Polynomial Random Number Generators,” *Journal of Cryptology 3* (1991), pp. 157-172.

[B116] Mihailescu, P. “Fast Generation of Provable Primes Using Search in Arithmetic Progressions,” Yvo G. Desmedt, Ed., *Advances in Cryptology*, CRYPTO ‘94, *Lecture Notes in Computer Science 839* (1994), Springer-Verlag, pp. 282-293.

[B117] Miller, V. S. “Use of Elliptic Curves in Cryptography,” H. C. Williams, Ed., *Advances in Cryptology*, CRYPTO ’85, *Lecture Notes in Computer Science 218* (1986), Springer-Verlag, pp. 417-426.

[B118] Morain, F. “Building Cyclic Elliptic Curves Modulo Large Primes,” D. W. Davies, Ed., *Advances in Cryptology*, EUROCRYPT ‘91, *Lecture Notes in Computer Science 547* (1991), Springer-Verlag, pp. 328-336.

[B119] National Institute of Standards and Technology (NIST), “Recommended Elliptic Curves for Federal Government Use” (draft), 1999. Available at http://csrc.nist.gov/encryption/.

[B120] Nyberg, K., and Rueppel, R. “A New Signature Scheme Based on the DSA Giving Message Recovery,” *Proceedings of First ACM Conference on Computer and Communications Security* (1993), ACM Press, pp. 58-61.

[B121] Odlyzko, A. M. “The Future of Integer Factorization,” *CryptoBytes*, Vol. 1, no. 2 (Summer 1995), RSA Laboratories, ftp://ftp.rsa.com/pub/cryptobytes/crypto1n2.pdf.

[B122] van Oorschot, P., and Wiener, M. “Parallel Collision Search With Applications to Hash Functions and Discrete Logarithms,” *Proceedings of Second ACM Conference on Computer and Communications Security* (1994), ACM Press, pp. 210-218.

[B123] Pollard, J. M. “Theorems on Factorization and Primality Testing,” *Proceedings of the Cambridge Philosophical Society 76* (1974), pp. 521-528.

[B124] Pollard, J. M. “A Monte Carlo Method for Factorization,” *BIT 15* (1975), pp. 331-334.

[B125] Pollard, J. M. “Monte Carlo Methods for Index Computation (mod *p*),” M*athematics of Computation 32* (1978), pp. 918-924.

[B126] Public-key Cryptography Standards (PKCS), *PKCS #1 v1.5: RSA Encryption Standard*, November1993. Available at http://www.rsasecurity.com/rsalabs/pkcs/.

[B127] Public-key Cryptography Standards (PKCS), *PKCS #1 v2.0: RSA Cryptography Standard*, 1998. Available at http://www.rsasecurity.com/rsalabs/pkcs/.

[B128] Rabin, M. O. “Digitalized Signatures and Public-key Functions as Intractable as Factorization,” *Massachusetts Institute of Technology Laboratory for Computer Science Technical Report 212* (MIT/LCS/TR-212), 1979.

[B129] Rivest, R. L., Shamir, A., and Adleman, L. M. “A Method for Obtaining Digital Signatures and Public-key Cryptosystems,” *Communications of the ACM 21* (1978), pp. 120-126.

[B130] Satoh, T., and Araki, K. “Fermat Quotients and the Polynomial Time Discrete Log Algorithm for Anomalous Elliptic Curves,” *Commentarii Mathematici Universitatis Sancti Pauli 47* (1998), pp. 81-92. Errata: ibid. 48 (1999), pp. 211-213.

[B131] Schirokauer, O. “Discrete Logarithms and Local Units,” *Philosophical Transactions of the Royal Society of London A*, 345 (1993), pp. 409-423.

[B132] Schneier, B. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd ed., John Wiley and Sons, 1995.

[B133] Schroeppel, R., Orman, H., O’Malley, S., and Spatscheck, O. “Fast Key Exchange With Elliptic Curve Systems,” *Univ. of Arizona Computer Science Technical Report 95-03* (1995). Also appears in D. Coppersmith, Ed., *Advances in Cryptology*, CRYPTO ‘95, *Lecture Notes in Computer Science 963* (1995), Springer-Verlag, pp. 43-56.

[B134] Semaev, I. A., “Evaluation of discrete logarithms in a group of *p*-torsion points of an elliptic curve in characteristic *p*,” *Mathematics of Computation 67*, pp. 353-356, 1998.

[B135] Seroussi, G. “Compact Representation of Elliptic Curve Points Over F_{2^n},” Hewlett-Packard Laboratories Technical Report HPL-98-94R1 (1998). Available at http://www.hpl.hp.com/techreports/.

[B136] Shamir, A. “RSA for Paranoids,” *CryptoBytes*, Vol. 1, no. 3 (Autumn 1995), RSA Laboratories, ftp://ftp.rsa.com/pub/cryptobytes/crypto1n3.pdf.

[B137] Shawe-Taylor, J. “Generating Strong Primes,” *Electronics Letters 22*, July 1986, pp. 875-877.

[B138] Silverman, R. D. “The Multiple Polynomial Quadratic Sieve,” *Mathematics of Computation 48* (1987), pp. 329-339.

[B139] Silverman, J. *The Arithmetic of Elliptic Curves*. Springer-Verlag, 1986.

[B140] Smart, N. P. “The Discrete Logarithm Problem on Elliptic Curves of Trace One,” *J. Cryptology*, Vol. 12, no. 3, pp. 193-196, 1999.

[B141] Solo, D., Adams, C., Kemp, D., and Myers, M. “Internet X.509 Certificate Request Message Format,” Internet Engineering Task Force (IETF), PKIX Working Group, work in progress. Available at http://www.ietf.org/ids.by.wg/pkix.html.

[B142] Solo, D., Housley, R., Ford, W., and Polk, T. “Internet X.509 Public-key Infrastructure Certificate and CRL Profile,” Internet Engineering Task Force (IETF), PKIX Working Group, work in progress. Available at http://www.ietf.org/ids.by.wg/pkix.html.

[B143] Stallings, W. *Cryptography and Network Security: Principles and Practice*, 2nd ed., Prentice-Hall, 1998.

[B144] Standards for Efficient Cryptography, “GEC1: Recommended Elliptic Curve Domain Parameters” (draft), September 1999. Available at http://www.secg.org/drafts.htm.

[B145] Stinson, D. R. *Cryptography: Theory and Practice*. CRC Press, 1995.

[B146] Vaudenay, S. “Hidden Collisions on DSS,” N. Koblitz, Ed., *Advances in Cryptology*, CRYPTO ‘96, *Lecture Notes in Computer Science 1109* (1996), Springer-Verlag, pp. 83-88.

[B147] Wiener, M. J. “Cryptanalysis of Short RSA Secret Exponents,” *IEEE Transactions on Information Theory 36* (1990), pp. 553-558

[B148] Wiener, M. J., and Zuccherato, R. “Faster Attacks on Elliptic Curve Cryptosystems,” S. Tavares and H. Meijer, Eds., *Selected Areas in Cryptography*, SAC ’98, *Lecture Notes in Computer Science 1556* (1998), Springer-Verlag.

[B149] Williams, H. C. “A Modification of the RSA Public-key Encryption Procedure,” *IEEE Transactions on Information Theory 26* (1980), pp. 726-729.

[B150] Williams, H. C. “A p + 1 Method of Factoring,” *Mathematics of Computation 39* (1982), pp. 225-234.

[B151] Yao, A. C. “Theory and Applications of Trapdoor Functions,” *Proceedings of the IEEE 23rd Annual Symposium on Foundations of Computer Science*, FOCS ’92, pp. 80-91.