IEEE Trial-Use Standard for Wireless Access in Vehicular Environments—Security Services for Applications and Management Messages

1609.2™

Intelligent Transportation Systems Committee

Sponsored by the
IEEE Vehicular Technology Society

IEEE
3 Park Avenue
New York, NY 10016-5997, USA

6 July 2006

IEEE Std 1609.2™-2006

IEEE Std 1609.2™-2006

IEEE Trial-Use Standard for Wireless Access in Vehicular Environments—Security Services for Applications and Management Messages

Sponsor

Intelligent Transportation Systems Committee
of the
IEEE Vehicular Technology Society

Approved 8 June 2006

IEEE-SA Standards Board

Abstract: Secure message formats, and the processing of those secure messages, within the DSRC/WAVE system are defined. The standard covers methods for securing WAVE management messages and application messages, with the exception of vehicle-originating safety messages. It also describes administrative functions necessary to support the core security functions
Keywords: authentication, Dedicated Short Range Communications (DSRC), encryption, vehicular communications, Wireless Access in Vehicular Environments (WAVE)

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

Print: ISBN 0-7381-5008-8 SH95558
PDF: ISBN 0-7381-5009-6 SS95558

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an IEEE Standard is wholly voluntary. The IEEE disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEEE Standard document.

The IEEE does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEEE Standards documents are supplied "**AS IS**."

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.

In publishing and making this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

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 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. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE.

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

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854
USA

Introduction

This introduction is not part of IEEE Std 1609.2, IEEE Trial-Use Standard for Wireless Access in Vehicular Environments—Security Services for Applications and Management Messages.

5.9 GHz Dedicated Short Range Communications for Wireless Access in Vehicular Environments (DSRC/WAVE, hereafter simply WAVE), as specified in a range of standards including those generated by the IEEE P1609 working group, enables vehicle-to-vehicle (V2V), and vehicle-to-infrastructure (V2I) wireless communications. This connectivity makes possible a range of applications that rely on communications between road users, including vehicle safety, public safety, commercial fleet management, tolling, and other operations.

With improved communications come increased risks, and the safety-critical nature of many WAVE applications makes it vital that services be specified that can be used to protect messages from attacks such as eavesdropping, spoofing, alteration, and replay. Additionally, the fact that the wireless technology will be deployed in personal vehicles, whose owners have a right to privacy, means that in as much as possible the security services should respect that right and not leak personal, identifying, or linkable information to unauthorized parties.

With this in mind, at the time that IEEE P1609 was established to develop the standards for the DSRC wireless network stack, the IEEE also established IEEE P1556™ (later renumbered as IEEE 1609.2) to develop standards for the security techniques that will be used to protect the services that use this network stack. These applications face unique constraints. Many of them, particularly safety applications, are time-critical: the processing and bandwidth overhead due to security must be kept to a minimum, to improve responsiveness and decrease the likelihood of packet loss. For many applications, the potential audience consists of all vehicles on the road in North America; therefore, the mechanism used to authenticate messages must be as flexible and scalable as possible, and must accommodate the smooth removal of compromised units from the system. Additionally, as mentioned above, the privacy of privately owned and operated vehicles must be respected as far as technically and administratively feasible.

This document specifies a range of security services for use in the WAVE environment. Mechanisms are provided to authenticate WAVE management messages, to authenticate messages that do not require anonymity, and to encrypt messages to a known recipient. Mechanisms to provide anonymity, particularly anonymous broadcast, will be provided in a separate document.

Notice to users

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: http://standards.ieee.org/reading/ieee/updates/errata/index.html. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations can be accessed at the following URL: http://standards.ieee.org/reading/ieee/interp/index.html.

Patents

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 or patent applications for which a license may be required to implement an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention. A patent holder or patent applicant has filed a statement of assurance that it will grant licenses under these rights without compensation or under reasonable rates and nondiscriminatory, reasonable terms and conditions to applicants desiring to obtain such licenses. The IEEE makes no representation as to the reasonableness of rates, terms, and conditions of the license agreements offered by patent holders or patent applicants. Further information may be obtained from the IEEE Standards Department.

Publication of this trial-use standard for comment and criticism has been approved by the Institute of Electrical and Electronics Engineers. Trial-use standards are effective for 24 months from the date of publication. Comments for revision will be accepted for 18 months after publication. Suggestions for revision should be directed to the Secretary, IEEE-SA Standards Board, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, and should be received no later than 7 January 2007. It is expected that following the 24-month period, this trial-use standard, revised as necessary, shall be submitted to the IEEE-SA Standards Board for approval as a full-use standard.

Participants

The active participants in the IEEE P1609.2 (1556) working group at the time this trial-use standard was developed and balloted were as follows:

Thomas M. Kurihara, *Chair*
Roger J. O'Connor, *Working Group Vice Technical Chair*
Douglas M. Kavner, *Security SWG Chair*
Eric Rescorla, *Primary Security Consultant*
William Whyte, *P1609.2 (1556) Technical Editor*

Scott Andrews
Lee R. Armstrong
Daniel V. Bailey
Jim Bauer
Art Carter
Broady Cash
Ronald K. Char
Emily Clark
J. Kenneth Cook
Khaled Dessouky
Eskafi Farokh
Wayne Fisher
Ramez Gerges
Susan Graham
Gloria Gwynne
Chris Hedges
Russell D. Housley
Mary Ann Ingram
Daniel Jiang
Carl Kain
Pankaj R. Karnik
Douglas M. Kavner
David Kelley
Hariharan Krishnan
Jerremy A. Landt
Jason Liu
Justin McNew
John T. Moring
Sean O'Hara
Peter Oomen
Sam Oyama
Joon Gou Park
Gordon Peredo
Frank Perry
Mohan Pundari
Ed Ring
Tom Schaffnit
Dick Schnacke
Douglas Siesel
Robert T. Soranno
Steve Spenler
Bill Spurgeon
Steve Tengler
Jim Tomcik
Roger Tong
Glenn Turnock
Bryan Wells
Filip Weytjens
Doug Whiting
Chris Wilson
Jijun Yin
Jeffery Zhu

The following members of the individual balloting committee voted on this trial-use standard. Balloters may have voted for approval, disapproval, or abstention.

Toru Aihara
Scott Andrews
Lee R. Armstrong
John R. Barr
Alexei Beliaev
Juan C. Carreon
Yi-ming Chen
Danila Chernetsov
Elizabeth Chesnutt
Kai Moon Chow
Keith Chow
J. Kenneth Cook
Tommy P. Cooper
Thomas J. Dineen
Randall L. Dotson
Marc Emmelmann
Avraham Freedman
Ignacio Marin Garcia
Nikhil Goel
Sergiu R. Goma
Randall C. Groves
Pradeep Gupta
Gloria G. Gwynne
Gary A. Heuston
Werner Hoelzl
Russell D. Housley
Raj Jain
Oh Jongtaek
Avinash Joshi
Pankaj R. Karnik
Piotr Karocki
Douglas M. Kavner
Stuart J. Kerry
Patrick W. Kinney
Jim Kulchisky
Thomas M. Kurihara
Jeremy A. Landt
Jun Liu
William Lumpkins
G. L. Luri
Julius M. Madey
Gary L. Michel
William J. Mitchell
Apurva N. Mody
Yasser L. Morgan
John T. Moring
Ross A. Morris
Andrew F. Myles
Michael S. Newman
Richard H. Noens
Satoshi Obara
Roger J. O'Connor
Chris L. Osterloh
Satoshi Oyama
Subburajan Ponnuswamy
Henry S. Ptasinski
Vikram Punj
Robert A. Robinson
Frank H. Rocchio
Randal D. Roebuck
Michael Scholles
Stephen C. Schwarm
Rich Seifert
John W. Sheppard
Robert T. Soranno,
Luca Spotorno
Thomas E. Starai
Mark A. Tillinghast
Scott A. Valcourt
Christopher G. Ware
William Whyte
Eric V. Woods
Paul R Work
Oren Yuen

When the IEEE-SA Standards Board approved this standard on 8 June 2006, it had the following membership:

Steve M. Mills, *Chair*
Richard H. Hulett, *Vice Chair*
Don Wright, *Past Chair*
Judith Gorman, *Secretary*

Mark D. Bowman
Dennis B. Brophy
William R. Goldbach
Arnold M. Greenspan
Robert M. Grow
Joanna N. Guenin
Julian Forster*
Mark S. Halpin
Kenneth S. Hanus
William B. Hopf
Joseph L. Koepfinger*
David J. Law
Daleep C. Mohla
T. W. Olsen
Glenn Parsons
Ronald C. Petersen
Tom A. Prevost
Greg Ratta
Robby Robson
Anne-Marie Sahazizian
Virginia C. Sulzberger
Malcolm V. Thaden
Richard L. Townsend
Walter Weigel
Howad L. Wolfman

*Member Emeritus

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

Satish K. Aggarwal, *NRC Representative*
Richard DeBlasio, *DOE Representative*
Alan H. Cookson, *NIST Representative*

Michelle Turner
IEEE Standards Program Manager, Document Development

Matthew Ceglia
IEEE Standards Program Manager, Technical Program Development

Contents

IEEE Trial-Use Standard for Wireless Access in Vehicular Environments—Security Services for Applications and Management Messages

1. Overview

1.1 Introduction

Wireless Access in Vehicular Environments (WAVE) is a radio communications system intended to provide interoperable wireless networking services for transportation. These services include those recognized for Dedicated Short-Range Communications (DSRC) by the U.S. National Intelligent Transportation Systems (ITS) Architecture (NITSA) [B21][1] and many others not specifically identified in the architecture. The system enables vehicle-to-vehicle (V2V) and vehicle-to-roadside or vehicle-to-infrastructure (V2I) communications, generally over line-of-sight distances of less than 1000 m, where the vehicles may be moving at speeds up to 140 km/h.

The Physical Layer (PHY) and Medium Access Control (MAC) use elements of the IEEE 802.11™ PHY and MAC and were under development at the time this standard was issued. Channelization and the upper layers of the network stack are defined in IEEE P1609.4™ [B4] and IEEE P1609.3™, respectively. IEEE P1609.1™ [B3] defines an application, the Resource Manager, that uses the network stack for communications. This document, IEEE Std 1609.2, specifies security services for the WAVE networking stack and for applications that are intended to run over that stack. Services include encryption using another party's public key, and non-anonymous authentication.

1.2 Scope

The scope of this standard is to define secure message formats, and the processing of those secure messages, within the DSRC/WAVE system. The standard covers methods for securing WAVE

[1] Numbers in brackets correspond to the bibliography in Annex I.

management messages and application messages, with the exception of vehicle-originating safety messages. It also describes administrative functions necessary to support the core security functions.

1.3 Purpose

The safety-critical nature of many DSRC/WAVE applications makes it vital that services be specified that can be used to protect messages from attacks such as eavesdropping, spoofing, alteration, and replay. Additionally, the fact that the wireless technology will be deployed in personal vehicles, whose owners have a right to privacy, means that in as much as possible the security services must be designed to respect that right and not leak personal, identifying, or linkable information to unauthorized parties. This standard describes security services for WAVE management messages and application messages, with the exception of vehicle-originating safety messages, to meet these requirements. It is anticipated that vehicle-originating safety messages will be added in an amendment to this standard.

1.4 Document organization

The document contains both normative and informative text, and is organized as follows.

Clause 1 reviews the scope and purpose of this standard and introduces the main entities in the system.

Clause 2 contains the normative references. These are documents that shall be referred to in order to produce a compliant implementation of this standard.

Clause 3 provides a list of definitions and commonly used abbreviations.

Clause 4 describes the presentation language that is used in later clauses to define secured message formats.

Clause 5 uses the presentation language of Clause 4 to describe the secure message formats for time or bandwidth constrained applications. These include signed messages, encrypted messages, and digital certificates.

Clause 6 describes other mechanisms that may be used to secure messages.

Clause 7 describes the steps to be taken on sending or receiving one of the secured messages described in Clause 5.

Clause 8 describes how the secure messages are to be used in specific contexts.

Annex A is a Protocol Implementation Conformance Statement (PICS) proforma, to be completed by implementers of this standard to indicate with what parts of the standard their implementation is conformant.

Annex B collects all the secured message formats defined in the document in a single location for convenience.

Annex C gives examples of the structure of some encoded messages for the convenience of implementers.

Annex D provides an informative overview of the entire WAVE system.

Annex E describes additional security considerations. These are typically infrastructure considerations that are not necessary for interoperation of two devices conforming to this standard.

Annex F describes how specific applications might make different uses of the security services described in this document.

Annex G describes how to use the techniques described in this standard in a way that minimizes the size (and so maximizes the probability of successful transmission) of time-critical messages.

Annex H is a copyright statement covering the text from Clause 5 that was excerpted from IETF RFC 2246 [B11].

Annex I contains the bibliography. The documents referenced here may be used to gain a fuller understanding of this standard, but conformance to this standard does not depend on conformance to any of these documents.

1.5 Document conventions

Unless otherwise stated, conventions follow those in IEEE Std 802.11 [B6], including conventions for the ordering of information within data streams.

Numbers are decimal unless otherwise noted. Numbers preceded by 0x are to be read as hexadecimal, so that 0xFF is equivalent to "FF hexadecimal".

Figures are used for illustration and are informative, unless otherwise noted.

2. Normative references

The following referenced documents are indispensable for the application of this standard. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

Federal Information Processing Standard (FIPS) 180-1, Secure Hash Standard.[2]

Federal Information Processing Standard (FIPS) 186-2, Digital Signature Standard.

IEEE P1609.3™ (Draft 19, March 2006), Draft Standard for Wireless Access in Vehicular Environments (WAVE) Networking Services.[3]

IEEE Std 1363™-2000, IEEE Standard Specifications for Public Key Cryptography.[4, 5]

IEEE Std 1363a™-2004, IEEE Standard Specifications for Public Key Cryptography: Additional Techniques.

IETF Request for Comments: 768, User Datagram Protocol.[7]

---

[2] 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.org/).

[3] This IEEE standards project was not approved by the IEEE-SA Standards Board at the time this publication went to press. For information about obtaining a draft, contact the IEEE (http://standards.ieee.org). Upon approved by the IEEE-SA Standards Board, IEEE P1609.3 should be superseded by IEEE Std 1609.3-200X (where 200X is the year of approval).

[4] IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, Piscataway, NJ 08855-1331, USA (http://standards.ieee.org/).

[5] The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[7] Internet RFCs are retrievable by FTP at ds.internic.net/rfc/rfcnnnn.txt (where nnnn is a standard's publication number such as 768), or call InterNIC at 1-800-444-4345 for information about receiving copies through the mail.

IETF Request for Comments: 3629, UTF-8, A Transformation Format of ISO 10646.

NIMA Technical Report TR8350.2, "Department of Defense World Geodetic System 1984, Its Definition and Relationships With Local Geodetic Systems."[8]

NIST Special Publication SP 800-38C, Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality.[9]

3. Definitions, abbreviations, and acronyms

3.1 Definitions

For the purposes of this document, the following terms and definitions apply. *The Authoritative Dictionary of IEEE Standards Terms* [B1] should be referenced for terms not defined in this clause.

3.1.1 advanced encryption standard (AES): A symmetric block cipher adopted by the U.S. government as a standard and used to encrypt and decrypt electronic data.

3.1.2 airlink: A radio-frequency communication interface, from the physical layer of one device to that of a remote device, such as the interface defined by WAVE.

3.1.3 application: In the context of IEEE Std 1609.2, a receiving application aggregates messages sent to one or more endpoints. A sending application sends messages to one or more endpoints.

3.1.4 application context mark (ACM): A code that identifies a specific instance of an application.

3.1.5 application class identifier (ACID): A code that identifies a class of application.

3.1.6 ApplicationID: A structure that contains an ACID and an ACM.

3.1.7 associated ApplicationID: If a certificate or signed message uses the `from_issuer` value in its `ApplicationID`, the associated `ApplicationID` is the first `ApplicationID` other than `from_issuer` to appear in its certificate chain. The associated ApplicationID is used as if it appeared in the original certificate or signed message.

3.1.8 asymmetric cryptographic algorithm: A cryptographic algorithm that uses two related keys— a public key and a private key. The two keys have the property that given the public key it is computationally infeasible to derive the private key.

3.1.9 block cipher: A symmetric encryption algorithm that processes data in blocks, typically of 8 or 16 octets.

3.1.10 broadcast application: An application that transmits information without targeting a specific recipient and without expecting a response. *Contrast*: **transactional application.**

3.1.11 certificate authority (CA) certificate: A certificate that is used to sign other certificates. *Contrast*: message signing certificate.

[8] This technical report is available from http://earth-info.nga.mil/GandG/tr8350/tr8350_2.html.

[9] This special publication is available from http://csrc.nist.gov/publications/nistpubs/.

3.1.12 cache: A collection of data objects such that each object stays in the cache for a limited period of time.

3.1.13 certificate: *See* **digital certificate**.

3.1.14 certificate authority (CA): An entity that issues certificates, and is accepted by other entities to have in place valid procedures to ensure that, if a certificate identifies an entity, it is only issued to that entity, and if a certificate grants permissions, it is only issued to entities who have a right to be granted those permissions.

3.1.15 certificate chain: A collection of certificates that can be put into an ordered list such that each certificate in the list (except the first) was issued by the owner of the certificate before it.

3.1.16 certificate signing request (CSR): A request by an entity to a CA to be issued with a certificate.

3.1.17 certificate signing request signing certificate (CSRSC): A certificate used to sign a certificate signing request (CSR).

3.1.18 certificate revocation list (CRL): A mechanism for the revocation of certificates. *See*: **revocation**.

3.1.19 certificate subject: The entity identified in a certificate; the entity that knows the private key corresponding to the public key in the certificate.

3.1.20 cipher block chaining (CBC): A mode of operation for a block cipher that uses the previous block of ciphertext to randomize the current block of plaintext, masking plaintext patterns.

3.1.21 cipher block chaining message integrity check (CBC-MIC): An application of the CBC mode of operation that produces a message integrity check rather than ciphertext.

3.1.22 control channel (CCH): A radio channel used for exchange of management data and WAVE short messages.

3.1.23 counter mode: A mode of operation of a block cipher where the output of encrypting an incrementing counter is used as a keystream.

3.1.24 cryptographically secure hash function: A hash function is a transformation that converts a variable length input to a fixed length string called the hash value. A cryptographically secure hash function maps an arbitrary-length input into a fixed-length output (the hash value) such that (a) it is computationally infeasible to find an input that maps to a specific hash value and (b) it is computationally infeasible to find two inputs that map to the same hash value.

3.1.25 datagram TLS (DTLS): A modification of TLS appropriate for packets transferred by UDP (or some other unreliable transport mechanism) as opposed to TCP (or some other reliable transport mechanism).

3.1.26 data plane: A set of communication protocols defined to carry application and management data.

3.1.27 decode: To parse an array of octets into a set of its constituent fields. *Contrast*: **decrypt, encode.**

3.1.28 decrypt: To convert unreadable, encrypted data to readable, decrypted data using an decryption algorithm and a key. *Contrast*: **decode, encrypt.**

3.1.29 decryption algorithm: An algorithm that takes as input ciphertext and a key and (if the correct key is provided) produces plaintext.

3.1.30 dedicated short range communication (DSRC): A system for wireless communication between vehicles and the roadside and from one vehicle to another.

3.1.31 digital certificate: A digitally signed document binding a public key to an identity and/or a set of permissions.

3.1.32 digital signature: *See:* **public-key digital signature.**

3.1.33 elliptic curve cryptography (ECC): A form of public-key cryptography based on the problem of finding discrete logarithms in a group defined over elliptic curves.

3.1.34 elliptic curve digital signature algorithm (ECDSA): A digital signature algorithm based on the hard problems of elliptic curve cryptography.

3.1.35 elliptic curve integrated encryption scheme (ECIES): A public-key encryption scheme based on the hard problems of elliptic curve cryptography.

3.1.36 encode: To convert a data structure into an array of octets. *Contrast*: **decode, encrypt.**

3.1.37 encrypt: To convert readable data to unreadable, encrypted data using an encryption algorithm and a key. *Contrast*: **decrypt, encode.**

3.1.38 encryption algorithm: An algorithm that takes as input plaintext and a key and produces ciphertext.

3.1.39 endpoint: The destination of a transmitted message. In the case of the IP stack, an endpoint is uniquely identified by the combination of IPv6 address and port number. In the case of the WSMP stack, an endpoint is uniquely identified by the Application Class ID and Application Context Mark.

3.1.40 expiry: Removing a certificate from the system after a given time.

3.1.41 hardware security module (HSM): A device that executes cryptographic functions in a secure execution environment, preventing unauthorized access to keys and functionality.

3.1.42 hash function: *See:* **cryptographically secure hash function.**

3.1.43 hash value: The output of a hash function.

3.1.44 implementation: Any entity that sends messages over the WAVE stack. An application, or set of applications that use the same radio.

3.1.45 issuing certificate: If the public key from certificate A can be used to verify the signature on certificate B, then A is the *issuing certificate* for B. *Contrast*: **subordinate certificate.**

3.1.46 keypair: A private key and the corresponding public key for an asymmetric cryptosystem.

3.1.47 keystream: A cryptographically secure stream of random data that is XORed with plaintext to produce ciphertext.

3.1.48 management plane: The collection of functions performed in support of the communication functions provided by the data plane, but not directly involved in passing application data.

3.1.49 message integrity check (MIC): A cryptographic checksum generated using a symmetric key.

3.1.50 message integrity check length (MIC length): The length of the MIC generated by a MIC scheme.

3.1.51 message integrity check scheme (MIC scheme): An instantiation of a cryptographic process that produces a MIC.

3.1.52 message signing certificate: A certificate that is used to verify signatures on messages. *Contrast*: **CA certificate.**

3.1.53 networking services: The collection of management plane and data plane functions at the network layer and transport layer.

3.1.54 on-board unit (OBU): A WAVE device that can operate when in motion and supports information exchange with RSUs and other OBUs.

3.1.55 plaintext: Data that has not been cryptographically processed (or has had all the cryptographic processing removed from it).

3.1.56 provider: An entity in the system, typically though not exclusively an RSU, that offers services for consumption by entities known as users, which are typically though not exclusively on-board units (OBUs).

3.1.57 provider service table (PST): The collection of data describing the applications that are registered with and available through a WAVE device.

3.1.58 public-key digital signature: A code generated with an asymmetric cryptographic algorithm. The signing operation takes as input a message to be signed and the signer's private key, and produces a signature. The verification takes as input the purported signer's public key, the message, and the signature, and will produce "valid" if the signature was produced with the corresponding private key and the message and signature have not been altered since generation, and "invalid" otherwise.

3.1.59 public safety on-board unit (PSOBU): An on-board unit (OBU) on a public safety vehicle.

3.1.60 public safety vehicle: A vehicle that is given privileges over ordinary, private vehicles by a government agency to assist it in carrying out functions necessary for safety of life, or other public safety functions.

3.1.61 randomized encryption: An encryption mechanism that has the property that the same message, encrypted twice with the same key, will give different results. Randomized encryption is semantically secure if an attacker who only sees the two ciphertexts will not be able to tell that they were derived from the same plaintext.

3.1.62 registration: The provision of application parameters to a provider or user. These parameters are used by the provider to populate the PST, by the user to decide whether to link to a service advertised by a provider, and by both entities to manage the establishment and maintenance of the link in support of the service.

3.1.63 replay attack: An attack in which an attacker records a valid message and retransmits it at a different time or location, hoping that listeners will still react to it as if it were valid.

3.1.64 revocation: The act of removing a certificate from the system by distributing a signed statement that the certificate is no longer to be trusted.

3.1.65 revoked certificate store: A unit's list of revoked certificates.

3.1.66 roadside unit (RSU): A WAVE device that operates only when stationary and supports information exchange with on-board units (OBUs).

3.1.67 root certificate: A certificate that was issued by itself and that can be used as a trust anchor to validate other certificates.

3.1.68 root certificate store: A unit's store of root certificates.

3.1.69 security manager: The application that is responsible for managing the root certificate store and the revoked certificate store on a unit.

3.1.70 semantically secure randomized encryption: *See:* **randomized encryption**.

3.1.71 service: An application offered by a provider for consumption by a user over an airlink.

3.1.72 service channel (SCH): A secondary channel used for application specific information exchanges.

3.1.73 store: A collection of data objects (typically certificates and CRLs in this standard) such that each object stays in the store until it expires, is superseded, or is revoked.

3.1.74 subordinate certificate: If the public key from certificate A can be used to verify the signature on certificate B, then B is a *subordinate certificate* to A. *Contrast*: **issuing certificate.**

3.1.75 symmetric cryptographic algorithm: A cryptographic algorithm that uses a single key. Anyone who knows a symmetric encryption key can both encrypt and decrypt. Anyone who knows a symmetric authentication key can both generate and check a MIC.

3.1.76 symmetric key: *See:* **symmetric cryptographic algorithm**.

3.1.77 transactional application: An application in which two entities exchange information. *Contrast*: broadcast application.

3.1.78 transport layer security (TLS): An IETF standard providing for secure communications over TCP/IP.

3.1.79 trust anchor: A known, trusted certificate. When a party receives a certificate, it will typically attempt to construct a certificate chain back to a trust anchor. If no trust anchor is available, the party shall use an out-of-band method to check that the certificate under consideration was issued by a legitimate issuer.

3.1.80 unit: An implementation of a WAVE stack, with a single WAVE management entity (WME) and a single security manager.

3.1.81 vehicle host: In-vehicle device running applications that communicate to external WAVE devices using the WAVE system through the OBU vehicle-host interface.

3.1.82 WAVE device: A device that contains a WAVE-conformant medium access control (MAC) and physical layer (PHY) interface to the wireless medium (see IEEE Std 802.11 and IEEE P1609.4).

3.1.83 WAVE management entity (WME): The set of management functions, defined in IEEE P1609.3, required to provide WAVE Networking Services; the management entity that performs these functions.

3.1.84 WAVE routing advertisement (WRA): Network configuration information broadcast by the roadside unit (RSU).

3.1.85 WAVE service advertisement (WSA): A collection of data collected by the WME and used to populate the WSIE.

3.1.86 WAVE service information element (WSIE): A collection of configuration data transmitted by a provider (either OBU or RSU), which includes the Provider Service Table, and in the case of the RSU the WAVE Routing Advertisement, as well as security credentials.

3.1.87 WAVE short message (WSM): A message sent over WSMP and routed to a receiving application using an ACID and an ACM rather than an IP address and port.

3.1.88 WAVE short message protocol (WSMP): A protocol, defined in IEEE P1609.3, for routing WAVE short messages to receiving applications.

3.1.89 WAVE stack: The network stack described in the IEEE 1609 series of standards.

3.1.90 wireless access in vehicular environment (WAVE): A series of standards, developed by the IEEE P1609 working group, for vehicle-to-vehicle and vehicle-to-roadside wireless communications.

3.2 Abbreviations and acronyms

AES	Advanced Encryption Standard
ACID	application class identifier
ACM	application context mark
ASN.1	Abstract Syntax Notation 1
BER	Basic Encoding Rules (used with ASN.1)
CA	certificate authority
CBC	cipher block chaining
CCH	control channel
CCM	Counter with CBC MIC
CRL	certificate revocation list
CSR	certificate signing request
CSRSC	certificate signing request signing certificate
DIC	DSRC Industry Consortium
DNS	domain name system
DSRC	Dedicated Short Range Communications
DTLS	datagram TLS
ECC	elliptic curve cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
ECIES	Elliptic Curve Integrated Encryption Scheme
FEC	Forward Error Correction
FIPS	Federal Information Processing Standard
GPS	Global Positioning System
HSM	hardware security module
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPv6	Internet Protocol version 6
ITS	Intelligent Transportation Systems
LLC	logical link control
MAC	medium access control
MIC	message integrity check
MIME	Multipurpose Internet Mail Extensions

MLME	MAC Layer Management Entity
MPDU	message protocol data unit
NIST	National Institute for Standards and Technology
NITSA	National Intelligent Transportation Systems Architecture
OBU	on-board unit
PDU	protocol data unit
PHY	Physical Layer
PLME	Physical Layer Management Entity
PSCRC	public safety certificate requesting certificate
PSOBU	public safety on-board unit
PST	Provider Service Table
RFC	Request for Comments
RSU	roadside unit
SAE	Society of Automotive Engineers
S/MIME	secure MIME
SP	special publication
TLS	Transport Layer Security
UDP	User Datagram Protocol
UTF	Unicode Transformation Format
V2I	vehicle-to-infrastructure
V2V	vehicle-to-vehicle
WAVE	Wireless Access in Vehicular Environments
WBSS	WAVE Basic Service Set
WGS	World Geodetic System
WME	WAVE Management Entity
WRA	WAVE Routing Advertisement
WSA	WAVE Service Advertisement
WSIE	WAVE Service Information Element
WSM	WAVE Short Message
WSMP	WAVE Short Message Protocol
XOR	exclusive-OR

3.3 Terminology, applications, implementations, and the security manager

For an overview of the entire WAVE system and an introduction to cryptographic terminology, see Annex D.

The WAVE system involves many different types of entities. In day-to-day operations, the following entities are the most active:

Providers—Units that provide services on one or more service channels.

Users—Units that consume services offered by providers.

Roadside Units (RSUs)—WAVE devices that operate only when stationary and support information exchange with OBUs. RSUs are typically embedded in infrastructure elements such as road signs and traffic signals; they may be moved from one site to another but do not operate when in motion. Providers will usually be RSUs. RSUs are licensed by site and may provide services on one or more service channels.

On-Board Units (OBUs)—WAVE devices that can operate when in motion and support information exchange with RSUs and other OBUs. OBUs are typically embedded in vehicles, though they may also be portable (hand-held). OBUs operate in private vehicles, and a major goal of the security protocols in this document is to enable them to operate without violating a driver's reasonable assumptions about personal privacy. OBUs will usually be users, but may be providers in certain cases.

An important subset of OBUs consists of

Public Safety On-Board Units (PSOBUs)—Network nodes embedded in public safety vehicles. These nodes are permitted by the relevant authorities to operate particular public safety applications such as traffic signal prioritization.

In addition, the following entities support security services:

Certificate Authorities (CAs)—Entities that are able to authorize other entities via the issuance and revocation of certificates.

In this standard:

— An "endpoint" is the source or destination of any message. In the case of the IP stack, an endpoint is uniquely identified by the combination of IPv6 address and port number. In the case of the WSMP stack, an endpoint is uniquely identified by the Application Class ID and Application Context Mark, described in IEEE P1609.3.
— An "application" processes messages received at one or more endpoints, or sends messages to one or more endpoints. A receiving application may aggregate the messages sent to multiple endpoints. A sending application is identified with the endpoint to which it sends its messages.
— An "implementation" refers to any collection of applications on a single instance of the DSRC/WAVE radio stack.

The WME of IEEE P1609.3 is an application in this terminology, in that it processes management messages.

This standard specifies the behavior of a security manager application. This application has the responsibility for managing the root certificate and revoked certificate stores defined in 7.2.2 and 7.2.7. The security manager application receives messages over the WAVE stack. This standard does not specify any messages that are sent by the security manager. There is one security manager per WAVE stack. See 8.3 for a further description.

4. Presentation language

4.1 General

This standard specifies message formats using a presentation language based on the presentation language used by TLS (IETF RFC 2246 [B11]). This clause describes that presentation language and its encoding. A copyright statement for this clause may be found in Annex H.

In this standard, *encoding* is used to denote the process of converting from an internal representation of a structure to a flat octet string containing the same information. *Decoding* is used to denote the process of converting a flat octet string into an internal representation of the structure encoded in that octet string. The process of decoding can fail if the received octet string is not a valid encoding of the expected structure. In this case, the received octet string cannot be parsed.

4.2 Notation conventions

Elements of the presentation language are presented in `this font`. (Courier)

Presentation language statements contain variable names, data types, and functions. Variable names are all lowercase. Multiple words in a variable name are indicated by underscores, as in `variable_name`. Data

types begin with an uppercase letter and may contain a mixture of upper and lowercase. Multiple words in a data type are indicated by uppercase letters, as in `DataType`. The exceptions to this rule are the built-in data types `uint8`, `uint16`, `uint32`, `uint64`, and `opaque`. Functions (such as `select` statements) follow the same conventions as variable names.

For compactness, a field within a structure may be referred to using the `structure.field` notation.

Comments may be included preceded by //.

```
// this is a comment
```

4.3 Basic block size

The basic data block size is one octet. Multiple octet data items are concatenations of octets in network byte order. All lengths are expressed in octets.

4.4 Numbers

The following numeric types are predefined.

— `uint8` is a single-octet encoding of an unsigned 8-bit integer.
— `uint16` is a two-octet big-endian encoding of an unsigned 16-bit integer.
— `uint32` is a four-octet big-endian encoding of an unsigned 32-bit integer.
— `uint64` is an eight-octet big-endian encoding of an unsigned 64-bit integer.
— `sint32` is a four-octet encoding of a 32-bit signed integer, in which (as in BER encoding) the first bit is the sign bit and the remaining 31 bits encode the integer in big-endian form.

4.5 Fixed-length vectors

The syntax for specifying a new type `TNew` that is a fixed-length vector of type `TOld` is

```
TOld  TNew[n];
```

where `n`, the length in octets, is a multiple of the size of `T`. The vector is encoded as `n` octets containing the data.

Example (1):

```
uint8  XY8[2];
```

This defines an `XY8` as a concatenation of `uint8`s of length 2 octets (which should not be read as "a concatenation of two `uint8`s", even though in this case the two are equivalent.). An `XY8` encoding the values (0x12, 0x34) is encoded as the octets `12 34`.

Example (2):

```
uint32  XY32[8];
```

This defines an `XY32` as a concatenation of `uint32`s of length 8 octets, in other words a concatenation of two `uint32`s. An `XY32` encoding the values (0x12, 0x34) is encoded as the octets `00 00 00 12 00 00 00 34`.

4.6 Variable-length vectors

The syntax for specifying a new type `TNew` that is a variable-length vector of type `TOld` and maximum length `n` is

```
TOld  TNew<n>;
```

The vector is encoded in (length, value) form, where the length in octets is encoded as a big-endian integer of the minimum size necessary to encode `n`. In definitions of types with variable length vectors, `n` shall take one of the values `2^8-1`, `2^16-1`, `2^32-1`, or `2^64-1`, denoting that the length is encoded in one, two, four, or eight octets respectively.

Example (1):

```
uint8       AsciiChar;
AsciiChar   Name<2^8-1>;
```

This defines a `name` as a concatenation of `AsciiChars` of length no more than 2^8-1 octets. A `name` encoding the value "abc" is encoded as the octets `03 61 62 63`. The leading octet, `03`, is the length and the remainder is the data.

Example (2):

```
uint8       AsciiChar;
AsciiChar   LongName<2^16-1>;
```

This defines a `long_name` as a concatenation of `AsciiChars` of length no more than 2^{16}-1 octets. A `long_name` encoding the value "abc" is encoded as the octets `00 03 61 62 63`. The leading two octets, `00 03`, are the length and the remainder is the data.

Example (3):

```
uint8       AsciiChar;
AsciiChar   Name<2^8-1>;
Name        OfficerNames<2^8-1>;
```

This defines the `OfficerNames` type as a concatenation of `names` of length no more than 2^8-1 octets. An `OfficerNames` encoding the two `names` "abc" and "def" is encoded as the octets `08 03 61 62 63 03 64 65 66`.

NOTE—In this example, it is up to the implementer to ensure that the individual `names` are short enough for the concatenation of their encodings to fit in 2^8-1 or fewer octets.[10]

4.7 The opaque and opaqueExtLength type

The notation

```
opaque FieldName[n]
```

or

```
opaque FieldName<n>
```

[10] Notes in text, tables, and figures are given for information only and do not contain requirements needed to implement the standard.

is used to denote a block of data that can be fixed or variable-length, exactly `n` bytes long in the fixed-length case (denoted with square brackets) or up to `n` bytes long in the variable-length case (denoted with angle brackets). Data in a field of type `opaque` is not further parsed by the security services. In this standard, `opaque` types carry application data that the security services protect.

The data in a variable-length field of type `opaque` is encoded preceded by its length, which shall be less than the value `n`. The value `n` shall take one of the values `2^8-1`, `2^16-1`, `2^32-1`, or `2^64-1`, denoting that the length is encoded in one, two four, or eight octets respectively.

The notation

```
opaqueExtLength FieldLength<n>
```

is used to denote a variable-length block of data that is not further parsed by the security services. The data in a field of type opaqueExtLength is *not* preceded by its length when encoded. Instead, the length of the field is deduced from external information.

4.8 Enumerated type

A field of type `enum` is a list of labels, each with a unique value, and an optional unlabelled maximum value. An `enum` can only assume the labeled values declared in the definition. An enum is encoded as an integer large enough to hold the maximum value given.

Example (1):

```
enum {usa(0), canada(1), mexico(2)}
    NaftaSignatories;
```

Because the maximum value given is 2, the enum is encoded as a single octet. The label `canada` as a member of `NaftaSignatories` is encoded as the value `01`.

Example (2):

```
enum {usa(0), canada(1), mexico(2), (2^8-1)}
    NaftaSignatories;
```

Because the maximum value given is 2^8-1, the enum is encoded as a single octet. The label `canada` as a member of `NaftaSignatories` is encoded as the value `01`.

Example (3):

```
enum {usa(0), canada(1), mexico(2), (2^16-1)}
    NaftaSignatoriesWithRoomForExpansion;
```

Because the maximum value given is 2^{16}-1, the enum is encoded as two octets. The label `canada` as a member of `NaftaSignatoriesWithRoomForExpansion` is encoded as the value `00 01`.

An `enum` may note that certain fields are reserved for test purposes and shall not be used for operations, even in future versions of this standard. The notation for this is:

```
enum {usa(0), canada(1), mexico(2), reserved(240...255), (2^8-1)}
    NaftaSignatories;
```

4.9 Constructed types

Structured types may be constructed from primitive types for convenience using the `struct` syntax, as in the following example.

```
uint8       AsciiChar;
AsciiChar   Name<2^8-1>;
enum        {chair(0), treasurer(1), secretary(2)}
                OfficerPosition;

struct      {
                Name              name;
                OfficerPosition   position;
            } Officer;
officer     officers<2^16-1>;
```

An organization with Alice as chair, Bob as treasurer, and Carol as secretary might encode its `officers` structure as follows (line breaks included for clarity only):

```
00 19                // length of encoded data is 19 octets
                     // encoded as 2-octet integer
   05                // length of "Alice"
      41 6C 69 63 65 // "Alice", ASCII-encoded
   00                // chair(0)
   03                // length of "Bob"
      42 6F 62       // "Bob", ASCII-encoded
   01                // treasurer(1)
   05                // length of "Carol"
      43 61 72 6F 6C // "Carol", ASCII-encoded
   02                // secretary(2)
```

4.10 The case statement

Defined structures may have variants based on some available knowledge. The selector shall be an enumerated type, as shown below, or a flag selector, discussed in 4.12. For example:

```
uint16      Pressure;
enum        {car(0), motorbike(1)}
                VehicleType;
struct      {
                VehicleType  type;
                select (type) {
                case car:
                    Pressure FrontLeft;
                    Pressure FrontRight;
                    Pressure RearLeft;
                    Pressure RearRight;
                case motorbike:
                    Pressure Front;
                    Pressure Rear;
                }
            } TirePressureInfo;
```

The syntax

```
case (sel_1):  ;
```

is used within a `select` statement to denote that one possible value of the selector (in this case, `sel_1`) has no additional fields associated with it. Alternatively, values of the selector that have no additional fields associated with them may be omitted from the select statement.

The syntax

```
case (sel_1):
case (sel_2):
    Type1  t1;
case (sel_3):
    Type2  t2;
```

is used within a `select` statement to denote that multiple values of the selector (in this case, `sel_1` and `sel_2`) have the same additional fields associated with them (in this case, `t1`). The general rule is that the first semicolon (`;`) appearing after the `case` statement indicates that the `case` statement is terminated by the next `case` statement if one exists, and by the closing brace (`}`) otherwise.

4.11 The extern statement

A structure may depend on information external to it. The notation is given below.

```
uint16      Pressure;
struct      {
                extern  VehicleType type;
                select (type) {
                case car:
                    Pressure FrontLeft;
                    Pressure FrontRight;
                    Pressure RearLeft;
                    Pressure RearRight;
                case motorbike:
                    Pressure Front;
                    Pressure Rear;
                }
            } TirePressureInfo;
```

For those structures in this standard that use external information, the text explains where the external information is to be found.

4.12 Flags

4.12.1 Use of flags field

The flags field provides an alternative mechanism for constructing conditional structures. A `flags` type consists of a declaration of the different values that it may encode. The `if_set()` and `if_not_set()` statements are used to make the contents of a structure conditional on a flags type.

An example of the use of a flags field is

```
flags       {start_included(0), end_included(1)}
                TimeInfoFlags;
uint64      Time;

struct      {
                TimeInfoFlags    time_flags;
                if_set(time_flags, start_included) {
                    Time64       start_time;
                }
                if_set(time_flags, end_included) {
                    Time64       end_time;
                }
             } TimeInfo;
```

The values of the entries in the `flags` declaration need not be consecutive. The maximum value of an entry in the `flags` declaration is 2047.

The `if_set` and `if_not_set` statements shall be used with opening and closing braces to denote the portions of the structure that are conditional on the given flag.

The `if_any_set` and `if_any_not_set` statements may take an arbitrary number of arguments. The first argument is the field to be checked and the remaining arguments are the values to check for. This is simply to allow compact representations, so that

```
if_set(flags, x) {
    uint32          z;
}
if_set(flags, y) {
    uint32          z;
}
```

may be represented as

```
if_any_set(flags, x, y) {
    uint32          z;
}
```

4.12.2 Encoding of flags field

An encoded flags field consists of a single octet, encoding its length in octets, followed by the flag data, which is a big-endian integer containing the flags. The `flags` field should be the minimum necessary length to encode all the flags that have been set.

A flag with value *v* is set in an encoded `flags` vector by ORing into the encoded vector the value (1 << *v*), where "<<" denotes left-shifting. For example, a vector where flags 2 and 5 were set would be the octets `01 24`, and a vector where the flag 8 was set would be the octets `02 01 00`.

If no flags are set, the flags field is encoded as the single octet `00`.

5. Secured Messages

5.1 General

This clause describes secure message formats for use in WAVE systems.

5.2 SecuredMessage type

Many of the specific applications discussed below make use of a single generic secured message format:

```
struct  {
            uint8        protocol_version; // 1 for this version
            MessageType  type;
            select (type) {
            case unsecured :
                opaque           message<2^32-1>;
            case signed :
                SignedMessage    signed_message;
            case encrypted :
                EncryptedMessage encrypted_message;
        } SecuredMessage;
```

```
enum    {   unsecured (0), signed(1), encrypted (2),
            reserved (240...255), (2^8-1)
        }   MessageType;
```

The fields in the SecuredMessage have the following meanings.

— `protocol_version` contains the current version of the protocol. The version described in this document is version 1, represented by the integer 1. There are no major or minor version numbers.
— `type` contains the type of the message. This tells the receiving unit how to interpret the message body. This standard defines the types `unsecured(0)`, which is included for use with WAVE short messages (WSMs); `signed` (1) indicating a signed message; and `encrypted` (2) indicating an encrypted message. Message types from 240 through 255 shall not be assigned and are reserved for private usage in test environments.

If the `type` field is `signed`, then the remaining octets are a `SignedMessage`, defined in 5.3. If the `type` field is `encrypted`, then the message block is an `EncryptedMessage`, defined in 5.13.

5.3 SignedMessage, ToBeSignedMessage, and MessageFlags types

```
struct {
            FullySpecifiedAppID  application;
            MessageFlags         mf;
            opaque               application_data<2^16-1>;
            if_flag_set (mf, use_generation_time) {
                Time64            generation_time;
            }
            if_flag_set (mf, expires) {
                Time64           expiry_time;
            }
            if_flag_set (mf, use_location) {
                3DLocationAndConfidence       transmission_location;
            }
       } ToBeSignedMessage;

struct {
            SignerInfo           signer;
            ToBeSignedMessage    unsigned_message;
            Signature            signature;
       } SignedMessage;

flags  {fragment(0), use_generation_time(1), expires(2),
            use_location(3)} MessageFlags;
```

In the `ToBeSignedMessage` structure:

— `application` identifies the application that will receive the `SignedMessage`. The `ApplicationID` type used here is discussed in more detail in 5.8. The `application.type` field shall take the value `fully_specified` in a `SignedMessage`.
— `mf` may contain the flags:
  — `fragment(0)` indicating that the message has been fragmented using the procedure specified in 8.5.
  — `use_generation_time(1)` indicating that the message will contain the transmission time.
  — `expires(2)` indicating that the message contains a `lifetime` field.
  — `use_location(3)` indicating that the `transmission_location` field shall be included.
— `application_data` contains the data for the application. Unless otherwise indicated, this is not interpreted by the security protocol and is passed unchanged up to the application.

— `generation_time` contains the time at which the message was generated. The `Time64` type is defined in 5.9. An implementation shall not send two messages with the same `generation_time` value.
— `expiry_time`, if present, contains the time at which the message expires.
— `transmission_location` contains the location of the generating unit at the time the message was generated. The `3DLocationAndConfidence` type is defined in 5.19.

In the `SignedMessage` structure:

— `signer` determines the keying material and hash algorithm used to sign the message. The `SignerInfo` type is defined in 5.10.
— `unsigned_message` contains the message data, as above.
— `signature` contains the digital signature itself. The `Signature` type is defined in 5.11.

The signature is calculated as follows: Given a signature algorithm that takes as inputs an octet string *M*, the message to be signed, and a private key *Pr* of the appropriate form for the signing algorithm, encode the `ToBeSignedMessage` as an octet string and use the resulting octet string as the input *M* to the signature algorithm.

The signature is verified as follows: Given a verification algorithm that takes as inputs an octet string *M*, the message to be verified; *S*, the signature on that message; and *Pu*, the public key to be used to verify the message, and given a `SignedMessage` *sm*: extract the encoded `ToBeSignedMessage` from the signed message and run the verification algorithm using the extracted `ToBeSignedMessage` as the input *M*, the `signature` from the `SignedMessage` as the input *S*, and the public key identified in the `SignerInfo` as the input *Pu*.

5.4 SignedWSM and ToBeSignedWSM types

```
struct {
            MessageFlags          mf;
            opaqueExtLength       wsm_payload<2^16-1>;
            if_flag_set (mf, use_generation_time) {
                Time64             generation_time;
            }
            if_flag_set (mf, expires) {
                Time64            expiry_time;
            }
            if_flag_set (mf, use_location) {
                3DLocationAndConfidence       transmission_location;
            }
        } ToBeSignedWSM;

struct {
            SignerInfo            signer;
            ToBeSignedWSM         unsigned_wsm;
            Signature             signature;
        } SignedWSM;
```

The `ToBeSignedWSM` type is a variant of the `ToBeSignedMessage` type. The fields have the same meanings as in the `ToBeSignedMessage` type, except that the `application` field is omitted.

The `SignedWSM` type is a variant of the `SignedMessage` type. The fields have the same meanings as in the `SignedMessage` type, except that the signed data is a `ToBeSignedWSM` structure rather than a `ToBeSignedMessage` structure.

The use of this type is described in 8.2.

5.5 PublicKey, PKAlgorithm, and SymmAlgorithm types

```
struct {

            PKAlgorithm             algorithm;

            select(algorithm){

            case ecdsa_nistp224_with_sha224:
            case ecdsa_nistp256_with_sha256:
                ECPublicKey         public_key;
            case ecies_nistp256:
                SymmAlgorithm       supported_symm_algs<2^8-1>;
                ECPublicKey         public_key;
            }

        } PublicKey;

enum    {   ecdsa_nistp224_with_sha224 (0),
            ecdsa_nistp256_with_sha_256 (1), ecies_nistp256 (2),
            reserved (240..255), (2^8-1)
        }   PKAlgorithm;

enum    {   aes_128_ccm (0), reserved (240..255), (2^8-1) }
            SymmAlgorithm;
```

The `PublicKey` structure encodes a public key and states with what algorithm the public key shall be used.

All implementations of this standard shall support the signing algorithm ECDSA over the two NIST curves p224 and p256, specified with the `PKAlgorithm` values `ecdsa_nistp224_with_sha224` and `ecdsa_nistp256_with_sha256`.

The only supported asymmetric encryption algorithm in this version of this standard is ECIES. All implementations of this standard that support encryption shall support this algorithm over the NIST curve p256, specified with the `PKAlgorithm` value `ecies_nistp256`.

— The `supported_symm_algs` field lists the symmetric algorithms that may be used with the ECIES public key. The only symmetric algorithm currently supported is AES in CCM mode as described in NIST SP 800-38C (see the references in Clause 2), denoted by `aes_128_ccm(0)`. All implementations that support encryption shall support this algorithm.
— `public_key` contains the encoded public key. The `ECPublicKey` type is specified in 5.6.

Additional public-key encryption or signature algorithms may be supported by reserving a `PKAlgorithm` value for them. Values 240-255 shall not be assigned and are reserved for private usage in test environments.

Additional symmetric algorithms may be supported by reserving a `SymmAlgorithm` value for them. Values 240-255 shall not be assigned and are reserved for private usage in test environments.

5.6 ECPublicKey type

An ECDSA or ECIES public key is specified as follows:

```
struct {
            extern uint8   point_size;
            opaque         point[point_size];
        } ECPublicKey;
```

ECC points are expressed in the LSB-compressed format of IEEE Std 1363-2000. The value `point_size` depends on the `PKAlgorithm` associated with the key. If the `PKAlgorithm` is `ecdsa_nistp224_with_sha224`, `point_size` shall be 29. If the PKAlgorithm is `ecdsa_nistp256_with_sha_256` or `ecies_nistp256`, `point_size` shall be 33.

5.7 CertID8 and CertID10 type

```
opaque CertID8[8];
opaque CertID10[10];
```

The `CertID8` and `CertID10` types are used to identify a certificate. The `CertIDX` for a given certificate, where X is 8 or 10, shall be calculated by calculating the SHA-256 hash of the certificate and taking the low-order X bytes of the hash output.

5.8 The ApplicationID and FullySpecifiedAppID types

```
struct {
            AIDType   type;
            select    (type) {
            case fully_specified:
                uint8   acid;
                opaque  acm<2^8-1>;
            case match_any_acm:
                uint8   acid;
            case from_issuer: ;
            }
        } ApplicationID;

enum    {fully_specified (0), match_any_acm(1), from_issuer(2),
            (2^8-1)} AIDType;

struct {
            uint8   acid;
            opaque  acm<2^8-1>;
        } FullySpecifiedAppID;
```

These types are used by the security services to determine whether a message may legitimately address a given application. This is carried out by checking that the `FullySpecifiedAppID` in a signed message is consistent with the `ApplicationID` in the signer's certificate. The process of checking consistency is covered in detail in 7.3.3.5.

In an `ApplicationID`:

— `type` shall take the value `fully_specified(0)`, `match_any_acm(1)` or `from_issuer(1)`. This field indicates which of the other fields are present in the encoded structure.
— `acid`, if present, contains the application class identifier. This field shall only take the values given in IEEE P1609.3.
— `acm`, if present, contains additional information about the receiving application. This data may be used by the security layer to check consistency as specified in 7.3.3.5.

In a `FullySpecifiedAppID`:

— `acid` contains the application class identifier. This field shall only take the values given in IEEE P1609.3.
— `acm` contains additional information about the receiving application. This data may be used by the security layer to check consistency as specified in 7.3.3.5.

NOTE—If the ACID and ACM are present, encoding them following the encoding rules of this document gives the same results as encoding them with the rules presented in IEEE P1609.3.

5.9 Time64 and Time32 types

```
uint64   Time64;
uint32   Time32;
```

This standard supports two time types to allow bandwidth to be saved if a low-precision time is sufficient.

The `Time64` type is a 64-bit integer, encoded in big-endian format, giving the number of microseconds since 00:00:00 UTC, 1 January, 2004.

The `Time32` type is a 32-bit integer, encoded in big-endian format, giving the number of seconds since 00:00:00 UTC, 1 January, 2004.

Subclause 7.3.3.2 discusses the source of the time measurements.

5.10 SignerInfo type

```
struct {
           SignerIdentifierType  type;
           select (type) {
           case certificate:
               WAVECertificate   certificate;
           case certificate_digest:
               CertID8           digest;
           case certificate_chain:
               WAVECertificate   certificates<2^16-1>;
           case self: ;
           }
       } SignerInfo;

enum {certificate(0), certificate_digest(1), certificate_chain (2),
       self (3), (2^8-1)}  SignerIdentifierType;
```

The `SignerInfo` structure allows the recipient of a message to determine which keying material to use to authenticate the message. It may contain either a certificate, a message digest of a certificate, or a certificate chain, as follows:

— If the `type` field contains `certificate(0)` then the `SignerInfo` contains a `certificate`.
— If the `type` field contains `certificate_digest(1)` then the `SignerInfo` contains the `CertID8` corresponding to the relevant certificate. This is the low-order 8 octets of the SHA-256 hash of that certificate.
— If the `type` field contains `certificate_chain(2)` then the `SignerInfo` contains multiple certificates. The first certificate shall be the certificate that signed the message. The other certificates may appear in any order. A compliant application shall support certificate chains containing five certificates, and may support chains that contain more. For more details on certificate chains see 7.3.3.4.

If the `type` field contains `self(3)` then the message is signed by a key contained within itself. This `type` may only be used to sign a certificate request. All other messages shall be signed with one of the other `types`.

5.11 Signature type

```
struct {
            extern PKAlgorithm algorithm;
            select(algorithm) {
            case ecdsa_nistp224_with_sha224:
            case ecdsa_nistp256_with_sha256:
            case ecdsa_with_sha256:
                ECDSASignature ecdsa_signature;
            }
        } Signature;
```

The `signature` field contains the actual signature. In order to parse it, a recipient needs to know the algorithm that it was created with. In the case of a signed message (or CRL) the algorithm is to be found in the `unsigned_certificate.public_key.algorithm` field of the message (or CRL) signing certificate. In the case of a certificate the algorithm is to be found in the `unsigned_certificate.public_key.algorithm` field of the issuing certificate. See 5.15 for the definition of `unsigned_certificate`.

5.12 ECDSASignature type

```
struct {
            extern uint8 curve_order_octets;
            opaque       r[curve_order_octets];
            opaque       s[curve_order_octets];
        }   ECDSASignature;
```

ECDSA signatures are performed as described in FIPS 186-2. The input message is the encoded `unsigned_message` value. An ECDSA signature consists of two values *r* and *s*. These integers shall be converted into octet strings of the same length as the curve order *n* using the procedure of FIPS 186-2. The strings are padded with leading zeros as necessary to obtain the appropriate length. The strings are then concatenated to form the appropriate signature value.

The value `curve_order_octets` is constant for a given `NamedCurve PKAlgorithm` value. It takes the value given in Table 1.

Table 1—curve_order_octets for different values of PKAlgorithm

NamedCurve	curve_order_octets
nistp224	28
nistp256	32

5.13 EncryptedMessage, EncryptedContentInfo, and RecipientInfo types

```
struct {
            EncryptedContentType   content_type;
            SymmAlgorithm          symm_algorithm;
            RecipientInfo          recipients<2^16-1>;
            select(symm_algorithm) {
            case aes_128_ccm:
                AESCCMCiphertext   ciphertext;
            }
        } EncryptedMessage;

struct {
            CertID8                          cert_id;
            extern PKAlgorithm               pk_encryption;
            select (pk_encryption)  {
            case ecies_nistp256:
                ECIESNISTp256EncryptedKey    enc_key;
```

```
            }
        } RecipientInfo;

enum {app_data(0), signed(1), (2^8-1)} EncryptedContentType;
```

The EncryptedMessage type supports encrypting a message to one or more recipients using the recipients' public keys. The message is encrypted with a fresh symmetric key generated by the sender, and the symmetric key is then encrypted for each recipient separately using that recipient's public key. See 7.4 for more details.

The EncryptedMessage type contains the following fields:

— content_type takes the values app_data or signed. If it is app_data, the contents of the encrypted message are application-specific. If it is signed, the contents of the encrypted message are a signed message and shall require further processing by the security services before they can be passed to the application.

— symm_algorithm contains an identifier for the symmetric algorithm that was used to encrypt the message. The SymmAlgorithm type is defined in 5.5. This field is provided primarily for future expansion; Tthe only supported value in this version of the standard is aes_128_ccm, indicating that the message was encrypted with AES using a 128-bit key in CCM mode. The SymmAlgorithm value here shall also appear in the supported_symm_algorithms field of all of the certificates identified in the cert_id fields of the recipients.

— recipients contains one or more RecipientInfos, defined below.

— The application_dataciphertext field contains the encrypted message. The general procedure to be used in generating the encrypted message is discussed given in 7.4.2, and the general procedure for processing received encrypted messages is discussed given in 7.4.3. The specific procedure for encryption and decryption with aes_128_ccm is discussed in 7.4.2.2.

The RecipientInfo type is used to transfer the symmetric key to each recipient separately. It contains the following fields:

— cert_id contains the low-order eight octets of the SHA-256 hash of the recipient's certificate. This shall be the certificate that contains the public key used to encrypt the symmetric key.

— pk_algorithm is set equal to the algorithm field of the encryption key in the referenced certificate. See 5.17 for more information.

— The final field contains the encrypted key. The format of this field depends on pk_algorithm. This standard only supports one pk_algorithm, ecies_nistp256, and only one format for the encrypted key, the ECIESNISp256EncryptedKey type.

5.14 ECIESNISTp256EncryptedKey and AESCCMCiphertext types

```
struct {
            extern uint32     symm_key_len;
            opaque            v[33];
            opaque            c[symm_key_len];
            opaque            t[16];
        }   ECIESNISTp256EncryptedKey;

struct {
            opaque          nonce[12];
            opaque          ccm_ciphertext<2^16-1>;
        }   AESCCMCiphertext;
```

The ECIESNISTp256EncryptedKey type is used to transfer the symmetric key, encrypted using ECIES as specified in IEEE Std 1363a-2004. The specific procedure for encryption and decryption, including the use of the fields in this type, is given in 7.4.2.3. The type contains the following fields:

— symm_key_len is the length of the key for the symmetric algorithm identified in the symm_algorithm field of the containing EncryptedMessage. In this standard, the only symm_algorithm value supported is aes_128_ccm and the corresponding key length shall be 16 octets.
— v is the sender's ephemeral public key, which shall be of length 33 for ecies_nistp256.
— c is the encrypted symmetric key, of length symm_key_length.
— t is the authentication tag, which shall be of length 16 for ecies_nistp256.

The RecipientInfo AESCCMCiphertext type is used to transfer the encrypted ciphertext. The specific procedure for encryption and decryption with aes_128_ccm, including the use of the fields in this type, is given in 7.4.2.2.The type symmetric key to each recipient separately. It contains the following fields:

— nonce field contains a 12-octet nonce.
— ccm_ciphertext field contains the ciphertext.

5.15 WAVECertificate, ToBeSignedWAVECertificate, CertSpecificData, SubjectType, and CRLSeries types

This standard supports different certificate types, corresponding to different roles that signers can play in the system. All certificates use the same basic structure:

```
struct {
            uint8                       certificate_version;
            ToBeSignedWAVECertificate unsigned_certificate;
            Signature                   signature;
        } WAVECertificate;

struct {
            extern uint8            certificate_version;
            SubjectType             subject_type;
            select (subject_type) {
            case ca:
            case crl_signer:
            case wsa_signer:
            case csr_signer:
            case rsu:
            case psobu:
            case obu_identified:
                CertID8             signer_id;
            case root_ca:  ;
            } signer;
            CertSpecificData        scope;
            Time32                  expiration;
            CRLSeries               crl_series;
            PublicKey               public_keys<2^8-1>;
        } ToBeSignedWAVECertificate.

struct {
            extern SubjectType subject_type;
            select(subject_type){
            case root_ca:
            case ca:
            case csr_signer:
                CAScope             scope;
            case crl_signer:
```

```
                CRLSeries           responsible_series<2^16-1>;
            case wsa_signer:
            case wsa_ca:
                WSASignerScope      scope;
            case rsu:
            case psobu:
                IdentifiedScope     scope;
            case obu_identified:
                OBUIdentifiedScope  scope;
            }
        } CertSpecificData;

enum    {  wsa_ca (0), ca(1), wsa_signer(2), rsu(3), psobu(4),
           obu_identified(5), crl_signer(6), csr_signer(8),
           root_ca(9), (2^8-1)
        }  SubjectType;

uint32 CRLSeries;
```

The fields in the `WAVECertificate` structure have the following meaning:

— `certificate_version` contains the version of the certificate format. In this version of the protocol, this field shall be set to 1.
— `unsigned_certificate` is the structure described below.
— `signature` is the signature, calculated by the signer identified in the `signer_id` field of the `unsigned_certificate`, over the encoding of the `unsigned_certificate`.

The fields in the `ToBeSignedWAVECertificate` structure have the following meaning:

— `certificate_version` allows the ToBeSignedWAVECertificate structure to reference the certificate_version field of the WAVECertificate. It is not used in this version of the standard but is included to allow for future extensibility.

— `subject_type` describes what kind of entity owns the certificate. It is used in this standard to determine the scope of the certificate and the means of identifying the signer (the certificate is signed by the key contained within it if the `subject_type` is `root_ca`, and otherwise it is signed by an external public key contained in another certificate).

— `expiration` contains the last `Time32` on which the certificate is valid. If the `expiration` is zero, the certificate does not expire. A valid certificate shall have a non-zero value in at least one of the `expiration` field and the `crl_series` field.

— `scope` contains information that is unique to this certificate `subject_type`. The different `scope` types are described below.

— `crl_series` contains an integer that represents which of multiple CRLs maintained by the CA this certificate shall appear on if it is ever revoked. `crl_series` only have meaning within the context of a given CA. The value 0 in this field indicates that the certificate shall not appear on a CRL (in other words, that its user shall use a series of reissued, short-lived certificates rather than a single long-lived certificate). For more details, see 7.3.3.6. A valid certificate shall have a non-zero value in at least one of the `expiration` field and the `crl_series` field.

— `signer_id` identifies the issuing certificate. The field shall contain the low order 8 octets of the SHA-1 hash of the CA's certificate.

— `public_keys` contains the public keys that belong to the subject of the certificate. This field shall contain either one or two keys; the recipient parses the contents of the `public_keys` field as a public key, and if at the end of that public key there is still data remaining in the `public_keys` field the recipient parses that data as another public key. It is an error if the public_keys field contains any data following a second public key. If this field contains two keys, one shall have its `PKAlgorithm` field set equal to either `ecdsa_nistp256_with_sha256` or

`ecdsa_nistp224_with_sha224` and the other shall have its `PKAlgorithm` field set equal to an `ecies_nistp256`.

5.16 WAVECRL, ToBeSignedCRL, CRLType, and IDAndDate types

A CRL has the following structure:

```
struct {
           uint8         version;
           SignerInfo    signer;
           ToBeSignedCRL unsigned_crl;
           Signature     signature;
       } WAVECRL;

struct {
           CRLType     type;
           CRLSeries   crl_series;
           CertID8     ca_id;
           uint32      crl_serial;
           Time32      start_period;
           Time32      issue_date;
           Time32      next_crl;
           select (type) {
           case (id_only):
               CertID10    entries<2^64-1>;
           case (id_and_expiry):
               IDAndDate   entries<2^64-1>;
           } entries;
       } ToBeSignedCRL;

enum { id_only(0), id_and_expiry(1), (2^8-1) } CRLType;

struct {
           CertID10    id;
           Time32      expiry;
       } IDAndDate;
```

The fields in the `WAVECRL` have the following meaning:

— `version` contains the CRL version. In this version of the standard, this field shall be set to 1.

— `signer` identifies the signing key. It shall not take the value `self`. It may only take the value `certificate_digest` if the certificate that contains the signing public key has `subject_type` equal to `root_ca`.

— `unsigned_crl` contains the unsigned CRL.

— `signature` contains the signature of the signer identified in the `signer` field. The signature is calculated over the contents of the `unsigned_crl` field.

The fields in the `ToBeSignedCRL` have the following meaning:

— `crl_series` represents the CRL series for which this CRL is used. See the discussion of partitioning CRLs in 8.3.3.2 for more details.

— `ca_id` contains the low-order eight octets of the hash of the certificate of the CA for which this CRL is being issued.

— `crl_serial` is a counter that should increment by 1 for every issued CRL by that CRL issuer.

— `start_period` and `issue_date` specify the time period that this CRL covers. The CRL shall include all certificates belonging to that `crl_series` that were revoked between `start_period` and `issue_date`. CRLs from the same issuer for the same `crl_series` may have overlapping time periods. If this is the case, any certificate revoked during the overlap period shall appear on multiple CRLs.

— `next_crl` contains the time when the next CRL is expected to be issued. See 7.3.3.6 for discussion of the behavior of recipients with respect to CRLs.

— `entries` contains identifiers for each revoked certificate.

  — If type is `id_only`, the entry lists only the ID for the certificate, calculated as described in 5.5.

  — If type is `id_and_expiry`, the entry lists the ID for the certificate and the expiry date for that certificate. This allows the recipient to maintain the certificate store more efficiently.

In either case, the entries shall be sorted lexicographically by their `CertID10` field. For a given CRL series, a CRL issuer shall issue either only CRLs of type `id_only` or only CRLs of type `id_and_expiry`.

5.17 WAVECertificateRequest and WAVECertificateResponse types

```
struct {
            SignerInfo info;
            ToBeSignedCSR unsigned_csr;
            Signature signature;
        } WAVECertificateRequest;

struct {
            uint8            csr_version;
            SubjectType      subject_type;
            RequestScopeType request_type;
            select (request_type) {
            case specified_in_request:
                CertSpecificData type_specific_data;
            case specified_by_ca: ;
            }
            PublicKey        public_key;
        } ToBeSignedCSR;

struct {
            Certificate certificate_chain<2^32-1>;
            WAVECRL crl_path<2^32-1>;
        } WAVECertificateResponse;

enum    { specified_in_request(0), specified_by_ca(1), (2^8-1) }
            RequestScopeType;
```

The `WAVECertificateRequest` type shall be used to request a `WAVECertificate`. The `WAVECertificateResponse` type shall be used to return a `WAVECertificate` to the requester.

In the `ToBeSignedCSR`:

— `csr_version` shall be set to 1 for this version of the standard.
— `subject_type` shall specify the type of the entity that is requesting the certificate.
— `request_type` shall be set to `specified_in_request` if the certificate request includes the scope that the requester wishes to have included in the certificate, and to `specified_by_ca` if the certificate scope field shall be filled in by the CA.

— If present, type_specific_data shall specify the desired scope. None of the fields in the type_specific_data field shall take the value from_issuer.
— public_key contains the public key of the enrollee.

In the WAVECertificateRequest:

— signature contains a signature over the encoded unsigned_csr value. The key used to verify the signature depends on the info field. If info.type is equal to self, the signature is generated using the private component corresponding to the public_key value. If info.type is equal to certificate, then the info.certificate field contains a certificate whose subject_type is csr_signer, and the signature is verified using the public key from that certificate. No other signer types are allowed.

In the WAVECertificateResponse:

— certificate_chain contains the certificate path of the new certificate. This path is in order, with the most local certificate (the newly issued one) being first and each successive certificate signing the one before it. The path should be complete with the final certificate being a trust anchor. However, some implementations may choose to deliver less complete paths for space reasons.
— crl_path contains the CRLs necessary to validate the certificate. At minimum, it shall contain the most recent version of the CRL series on which the issued certificate would appear if it were revoked. In addition, CAs should include CRLs corresponding to other CAs in the chain. These CRLs are not ordered.

5.18 GeographicRegion and RegionType types

5.18.1 GeographicRegion type

```
struct {
            RegionType              region_type;
            select(region_type){
            case from_issuer:
            case circle:
                CircularRegion      circular_region;
            case rectangle:
                RectangularRegion   rectangular_region<2^16-1>;
            case polygon:
                PolygonalRegion     polygonal_region;
            case none: ;
            }
        } GeographicRegion;

enum    {from_issuer(0), circle(1), rectangle(2), polygon(3),
            none (4), (2^8-1)} RegionType;
```

This standard supports a number of different region types. Currently, five are defined: from_issuer(0), circle (1), rectangle (2), polygon(3), and none(4). All region types with values < 240 shall be assigned before usage. Values 240–255 shall not be assigned and are reserved for private usage in test environments.

Note that the rectangular_region value is an array of RectangularRegion structures. This is interpreted as a series of rectangles, which may overlap or be disjoint. The permitted region is any point within any of the rectangles. Entities that claim conformance with this standard shall support rectangular_region values that include up to six rectangles.

For a discussion of how to use the various RegionTypes to minimize the size of broadcast messages, see Annex G.

5.18.2 The from_issuer RegionType

If the `RegionType` is `from_issuer`, no other information needs to be included in the `GeographicRegion` structure. The `from_issuer` value denotes that the geographic region of validity of the current certificate is the same as the geographic region of validity of the issuing certificate. Its use is described in 7.3.3.4.1.

5.18.3 CircularRegion type

```
struct {
            2DLocation center;
            uint16                  radius;
        } CircularRegion;
```

A `CircularRegion` is a circle with center at `center` and radius `radius` meters. The allowed region is the interior of the circle.

5.18.4 RectangularRegion type

```
struct {
            2DLocation    upper_left;
            2DLocation    lower_right;
        } RectangularRegion;
```

A `RectangularRegion` is a rectangle formed by connecting in sequence: (`upper_left.latitude`, `upper_left.longitude`), (`lower_right.latitude`, `upper_left.longitude`), (`lower_right.latitude`, `lower_right.longitude`), and (`upper_left.latitude`, `lower_right.longitude`).

5.18.5 PolygonalRegion type

```
2DLocation PolygonalRegion<2^16-1>;
```

The `PolygonalRegion` type defines a region using a series of distinct geographic points. The region is specified by connecting the points in a connect-the-dots arrangement in the order they appear. The polygon is completed by connecting the final point to the first point. The implied lines that make up the sides of the polygon shall not intersect. A CA shall not issue a certificate that contains a polygonal region whose sides intersect. The allowed region is the interior of the polygon.

Entities that comply with this standard shall support polygonal regions with up to 12 points.

5.19 The 2DLocation and 3DLocationAndConfidence types

```
struct {
            sint32        latitude;
            sint32        longitude;
            opaque        elevation_and_confidence[3];
        } 3DLocationAndConfidence;

struct {
            sint32        latitude;
            sint32        longitude;
        } 2DLocation;
```

These types are derived from the location encoding used in the SAE DSRC Common Safety Message Set (Modadugu and Rescorla [B20]). The `2DLocation` type is used to define validity regions for use in certificates and does not include a confidence field. The `3DLocationAndConfidence` type is used to indicate transmission locations and includes a confidence field. The types are to be interpreted as follows:

— The `latitude` and `longitude` fields contain the latitude and longitude as an `sint32` type, encoding the latitude and longitude in integer microdegrees. The encoded integer in the `latitude` field shall be no more than 9 0000 000 and no less than –9 000 0000. The encoded integer in the `longitude` field shall be no more than 180 000 000 and no less than 180 000 000. All values are relative to the WGS84 datum.
— In the `3DLocationAndConfidence` type, the `elevation_and_confidence` field is to be interpreted as follows. The first 20 bits give the big-endian encoding of the ellipsoidal height relative to the WGS84 ellipsoid. The bits are to be interpreted as an unsigned integer representing the height in centimeters above a baseline 1 km below the WGS84 ellipsoid. The remaining 4 bits give the confidence of the latitude and longitude fields, to be interpreted according to Table 2.

Table 2—Interpretation of the Positioning Confidence field

0000	Not equipped
0001	500 m (approx $5*10^{-3}$ decimal degrees)
0010	200 m (approx $2*10^{-3}$ decimal degrees)
0011	100 m (approx 10^{-3} decimal degrees)
0100	50 m (approx $5*10^{-4}$ decimal degrees)
0101	20 m (approx $2*10^{-4}$ decimal degrees)
0110	10 m (approx 10^{-4} decimal degrees)
0111	5 m (approx $5*10^{-5}$ decimal degrees)
1000	2 m (approx $2*10^{-5}$ decimal degrees)
1001	1 m (approx 10^{-5} decimal degrees)
1010	50 cm (approx $5*10^{-6}$ decimal degrees)
1011	20 cm (approx $2*10^{-6}$ decimal degrees)
1100	10 cm (approx 10^{-6} decimal degrees)
1101	5 cm (approx $5*10^{-7}$ decimal degrees)
1110	2 cm (approx $2*10^{-7}$ decimal degrees)
1111	1 cm (approx 10^{-7} decimal degrees)

5.20 Certificate Scopes

5.20.1 The CAScope type

In addition to the standard certificate data, a CA certificate (of type root_`ca(0)` or `ca(8)`) contains a scope field of type `CAScope`.

```
struct {
            SubjectTypesFlags   tf<2^8-1>;
            if_any_set (tf, ca, csr_signer, rsu, psobu,
                obu_identified) {
                ApplicationID       applications<2^16-1>;
            }
            if_any_set (tf, wsa_ca, wsa_signer) {
                AppIDAndPriority    apps_and_priorities<2^16-1>;
            }
            GeographicRegion    region;
        } CAScope;

flags  {  wsa_ca (0), ca(1), wsa_signer(2), rsu(3), psobu(4),
```

```
            obu_identified(5), crl_signer(6), csr_signer(8)
        } SubjectTypeFlags;

    struct {
            AIDType   type;
            select    (type) {
            case fully_specified:
                uint8   acid;
                opaque  acm<2^8-1>;
                uint8   maxPriority;
            case match_any_acm:
                uint8   acid;
                uint8   maxPriority;
            case from_issuer: ;
            }
        } AppIDAndPriority;
```

— `tf` lists the subject types for which this CA can issue certificates. See 5.15 for the `SubjectTypes` supported by this standard and their meaning. The flag value 7 is not allocated in this standard but is reserved for future use. Note that both the `wsa_ca` and the `ca` `SubjectTypeFlags` values allow the certificate to issue certificates with `subject_type` equal to `ca`. CAs for WSA signers and other CAs are only distinguished by their scope field. For further discussion of this, see 7.3.3.4 for ordinary CAs and 8.1.3 for WSA signing CAs.

— If present, `applications` indicates the application types for which this CA can issue certificates. If this field is empty (encoded as `00 00`, indicating a zero-length data field), it indicates that the CA can issue certificates for any application. If this field contains the `from_issuer` `ApplicationID`, it indicates that the CA has the same permissions as its own CA (see 7.3.3.5). If the `subject_type` field in the certificate is `root_ca`, the `applications` field shall not contain `from_issuer`. If the `applications` field contains an `ApplicationID` with `type` = `from_issuer`, that shall be the only `ApplicationID` contained in that field. Compliant implementations shall accept an `applications` field that contains up to 8 entries, and may accept more.

— If present, `apps_and_priorities` indicates which application types the CA can authorize to be advertised and the priorities at which those applications can be authorized. If this field is empty (encoded as `00 00`, indicating a zero-length data field), it indicates that the CA can issue certificates for any application and at any priority. If this field contains the `from_issuer` `ApplicationID`, it indicates that the CA has the same permissions as its own CA (see 7.3.3.5). If the `subject_type` field in the certificate is `root_ca`, the `apps_and_priorities` field shall not contain `from_issuer`. If the `apps_and_priorities` field contains an `AppIDAndPriority` with `type = from_issuer`, that shall be the only `AppIDAndPriority` contained in that field. Compliant implementations shall accept an `apps_and_priorities` field that contains up to 8 entries, and may accept more.

— `region` indicates the area for which the CA is allowed to issue certificates. If this field has `region_type` = `none`, it indicates that the CA is valid worldwide. If this field contains the `from_issuer` region value, it indicates that the CA can issue certificates within the same region that its own CA was entitled to issue certificates for. If the `subject_type` field in the certificate is `root_ca`, the `region` field shall not contain `from_issuer`.

5.20.2 WSASignerScope type

A `type` of `wsa_signer` indicates that the certificate subject (a) may broadcast WSAs advertising a specific set of applications; (b) may only operate within a certain area; and (c) may have a unique identifying name. A `type` of `wsa_ca` indicates that the certificate subject is entitled to issue certificates with `subject_type` `wsa_signer` or `wsa_ca`.

```
    struct {
            opaque              subject_name<2^8-1>;
```

```
            AppIDAndPriority  applications<2^16-1>;
            GeographicRegion  region;
        } IdentifiedScope;
```

— `subject_name` identifies the certificate subject. Its contents are a matter of local policy. They should consist of a human-readable string, encoded according to the UTF-8 encoding rules of IETF RFC 3629. This field may be empty (of length zero, encoded as `00`).

— `applications` contains a list of the applications that the certificate authorizes the WSA Signer to offer. It. The certificate subject shall only generate `SignedMessages` that are consistent with this field, and shall only offer them at a priority less than or equal to the given `maxPriority`. See 8.1 for further discussion. Compliant implementations shall accept an `apps_and_priorities` field that contains up to 8 entries, and may accept more.

— `region` indicates the geographic region within which the certificate subject is allowed to operate. If it contains the `from_issuer` region value, it indicates that the certificate subject can operate anywhere that its CA certificate is accepted.

5.20.3 IdentifiedScope type

A `type` of `rsu` or `psobu` indicates that the certificate subject (a) sends out messages for a restricted class of applications; (b) may only operate within a certain area; (c) may have a unique identifying name.

```
struct {
            opaque            subject_name<2^8-1>;
            ApplicationID     applications<2^16-1>;
            GeographicRegion  region;
        } IdentifiedScope;
```

— `subject_name` identifies the certificate subject. Its contents are a matter of local policy. They should consist of a human-readable string, encoded according to the UTF-8 encoding rules of IETF RFC 3629. This field may be empty (of length zero, encoded as `00`).

— `applications` contains a list of the applications that the certificate authorizes. The certificate subject shall only generate `SignedMessages` that are consistent with this `ApplicationID`. See 7.3.3.5 for further discussion. This field shall not be empty. Compliant implementations shall accept an `applications` field that contains up to 8 entries, and may accept more.

— `region` indicates the geographic region within which the certificate subject is allowed to operate. If it contains the `from_issuer` region value, it indicates that the certificate subject can operate anywhere that its CA certificate is accepted. This field shall not be empty.

NOTE—There is no functional difference in this standard between the `subject_types` `rsu` and `psobu`. However, RSUs and PSOBUs will probably have different certificate hierarchies, and receiving units may want to check that a public safety message was sent by an authenticated public safety vehicle, so the two unit `subject_types` are maintained separately.

5.20.4 CSRSignerScope type

This field is used by a Certificate Signing Request certificate (of type `csr_signer`) to request message signing certificates.

```
struct {
            opaque            subject_name<2^8-1>;
            SubjectType       types<2^8-1>;
            ApplicationID     applications<2^16-1>
            GeographicRegion  region;
        } CSRSigner;
```

— `subject_name` identifies the certificate subject. Its contents are a matter of local policy. They should consist of a human-readable string, encoded according to the UTF-8 encoding rules of IETF RFC 3629. It may be empty (encoded as `00`, a variable-length field of length 0).
— `types` lists the subject types for which the CSR signer can request a certificate. It shall contain one or more of the types `wsa_signer`, `rsu`, `psobu`, and `obu_identified`. No other types shall appear in this field.
— `applications` indicates the application types for which the CSR signer can request a certificate. This field shall not be empty or contain `from_issuer`. Compliant implementations shall accept an `applications` field that contains up to 8 entries, and may accept more.
— `region` indicates the region for which the CSR signer can request a certificate. It shall not be empty. If this field has `region_type` = `none`, it indicates that the CSR may request a certificate for any geographical location. It shall not contain the `from_issuer` region value.

5.20.5 OBUIdentifiedScope type

A `SubjectType` of `obu_identified` indicates that the certificate subject (a) transmits messages for a restricted class of applications; (b) is not restricted by its certificate to operating in a given geographic region (although the application that receives the certificate may, of course, enforce geographic restrictions); (c) has a unique identifier or name.

```
struct {
           opaque         cert_specific_data<2^16-1>;
           ApplicationID  applications<2^16-1>;
       } OBUIdentifiedScope;
```

— The contents of the `cert_specific_data` identifier field are determined by the issuing CA (see 8.4.3 for more details). This field may be empty (encoded as `00 00`).
— `applications` contains a list of the applications that the certificate authorizes. The certificate subject shall only generate `SignedMessages` that are consistent with this `ApplicationID`. See 7.3.3.5 for further discussion.

An entity running multiple identified applications may have a single certificate that applies to all of them, or may have multiple certificates, each covering one or more applications.

5.20.6 CRL Signer Certificates

A certificate of type `crl_signer` indicates that the holder of this certificate is entitled to sign CRLs for this certificate authority. The `responsible_series` array contains all of the CRL series for which this CRL can sign. See 8.3.3.1 for discussion of how to process received CRLs.

```
CRLSeries responsible_series<2^16-1>;
```

6. Other secured message formats

The format described in Clause 5 is for use when it is vital to minimize bandwidth, and when both the sending and the receiving applications are built specifically for use within the WAVE system. In other cases it may be necessary to communicate with a remote server that has other interoperability requirements. In this case applications should use existing standards for secured messages, such as S/MIME's Cryptographic Message Syntax (CMS) [B18].

Applications that communicate over UDP/IP also have the option of using an application layer secure session protocol such as DTLS [B20], or a network layer security protocol such as IPSec [B9], [B10], [B13]. Specific examples of these applications are considered in Annex F.

7. Secure message processing

7.1 Required information for security services

A WAVE unit shall make the following information available to all local implementations of the security services:

— Current time
— Estimated error in that time *E*
— Current location
— Estimated error in that location

7.2 Caches and stores

7.2.1 Introduction

This subclause describes the contents of the caches and stores that an entity shall or should maintain. The term "store" is used to denote semi-permanent storage. Items may be removed from a store only when they are superseded and overwritten. The term "cache" denotes temporary storage. Items in a cache may expire and be removed from the cache after a certain time even if they have not been superseded.

7.2.2 Root certificate store

The security manager on each unit shall maintain a store of root certificates to be used for constructing certificate chains when validating signed messages. This store shall be available to all applications local to the unit. A conformant implementation shall support a store that can contain at least eight certificates. See 8.3.4 for further details.

The store shall include the following for each root certificate:

— The certificate itself.
— The SHA-256 hash of the certificate, for use in matching the `signer_id`.

7.2.3 CA certificate cache

An implementation may maintain a cache of recently received CA certificates. These are non-root certificates that are used to sign other certificates, not messages. This cache may be application-specific, or may be shared between all applications on the unit. Maintaining this cache is not necessary, but may improve efficiency. For maximum efficiency, the cache should consist of the following for each certificate:

— The certificate itself.
— The SHA-256 hash of the certificate, for use for use in matching the `signer_id`.
— If the certificate's `application.type` field is `from_issuer`, the application field obtained from the issuer (see 7.3.3.4).
— If the certificate's scope contains a `GeographicRegion` type whose `region_type` field is `from_issuer`, the geographic region obtained from the issuer (see 7.3.3.4).
— The last time that the certificate was used by the recipient (so that the certificate need only be checked against CRLs that have arrived since its last use).

The lifetime of certificates in the cache may be determined by individual implementers. If this cache is maintained, then it shall be updated whenever a valid message is received by adding all the CA certificates from that message (in other words, all the certificates transmitted with the message except for the message signing certificate).

7.2.4 Message signing certificate cache

An implementation may maintain a cache of recently received message signing certificates, thereby reducing the computational overhead that would otherwise be incurred when a certificate is received twice in rapid succession. This cache may be application-specific, or may be shared between all applications on the unit. For maximum efficiency, the cache should include the following:

— The certificate itself.
— The SHA-256 hash of the certificate, for use if a signed message uses the `certificate_digest` form of the `SignerInfo` type (see 5.10).
— If the certificate's `application.type` field is `from_issuer`, the application field obtained from the issuer (see 7.3.3.4).
— If the certificate's scope contains a `GeographicRegion` type whose `region_type` field is `from_issuer`, the geographic region obtained from the issuer (see 7.3.3.4).
— The last time that the certificate was used by the recipient (so that the certificate need only be checked against CRLs that have arrived since its last use).

The time that a certificate remains in the cache before being deleted may be determined by individual implementers.

7.2.5 Recently received message cache

An application may maintain a cache of messages whose `generation_time` differs by no more than a tolerance r from the currently measured time T at the unit. The value r shall be at least 30 seconds. The exact choice of r may be left to the application and may vary depending on local conditions. The tolerance r shall never be less than the estimated error in the current time E.

The contents of the cache shall be either the messages themselves, or, for each message, a tuple of:

— A cryptographically strong hash of the message
— The message's `generation_time` value (which is used to purge stale messages from the cache).

The already received cache shall be emptied of all packets older than $T - r$ no less often than every 30 seconds.

The use of this cache to prevent replay attacks is described in 7.3.3.2.

7.2.6 Potential encrypted recipients certificate store

An application shall maintain a store of certificates whose owners it might want to send encrypted messages to. This store shall typically consist simply of the most recently received certificate for that application, but may contain an arbitrary number of other certificates. The cache should include the following:

— The certificate itself.
— The SHA-256 hash of the certificate.
— If the certificate's `application.type` field is `from_issuer`, the application field obtained from the issuer (see 7.3.3.4).
— If the certificate's scope contains a `GeographicRegion` type whose `region_type` field is `from_issuer`, the geographic region obtained from the issuer (see 7.3.3.4).
— The last time that the certificate was used by the recipient (so that the certificate need only be checked against CRLs that have arrived since its last use).

The cache shall at minimum support retrieving certs by `scope.subject_name` (if such exists) and by their SHA-256 hash.

7.2.7 Revoked certificate store

The security manager on a unit shall maintain a store with a list of revoked certificates. This store shall be available to all applications hosted on the unit.

A CRL issuer may issue a number of different CRL series. The revoked certificate store shall contain:

— For each (issuer, CRLSeries) tuple:
  — If the CRL is of type `id_only`, the `CertID10` for each certificate.
  — If the CRL is of type `id_and_expiry`, the `CertID10` and expiry date for each certificate.
  — The time that the next CRL is expected to issue.

It may also contain:

— The most recent time at which a CRL was received.

If the CRLs for a given issuer and CRLSeries are of type `id_and_expiry`, the implementation shall periodically review the store and remove certificates whose expiry date has passed.

See 8.3.3 for further details.

7.2.8 Incoming fragmented message cache

An application shall maintain a cache of incoming fragmented messages. See 8.5 for further details.

7.3 Signed messages

7.3.1 Overview

All implementations shall support the following procedure to process received signed messages.

Specific uses of signed messages by management entities (the WME and the security manager) are described in Clause 8.

7.3.2 Transmission processing

An entity that creates a signed message for transmission shall have access to the following information and services:

— A signing key.
— The certificates associated with that signing key. These certificates shall contain scopes that authorize the signed message.
— A random number generator.
— A cryptographic implementation, hardware or software, that implements the signing algorithm. This shall include a cryptographically secure random number generator
— The current position and time and the estimated error in that position and time.

An application may choose not to send a `SignedMessage` if the estimated error in the position or time is greater than some application-specific threshold.

To sign a message, the entity shall take the following steps:

a) Fill in the `ToBeSignedMessage` structure and encode it as the `unsigned_value` octet string following the encoding rules specified in Clause 4.

1) If the transmission time is to be included, set the `use_generation_time` flag and include the current time in the `generation_time` field. The transmission time shall be included unless the application uses some alternative means to prevent replay attacks. See 7.3.3.2 for further discussion.
2) If the location is to be included, set the `use_location` flag and include the current location in the `location` field
3) If the message is going to expire, set the `expires` flag and include the expiry time in the `expiry_time` field.
4) Set the `application.type` field to `fully_specified` and include the `acid` and `acm` of the sending application in the `application.acid` and `application.acm` fields.

b) Digitally sign the `unsigned_message` value. The signature is computed over the encoding of the entire `ToBeSignedMessage` structure.
c) Create and encode the `signed_message` value. The signature algorithm is uniquely determined by the type of the key used to generate the signature. There are no in-message indicators for signature algorithm type. The output of this process is a `signed_message`, which can then be passed to the appropriate part of the network stack for transmission.

7.3.3 Reception processing

7.3.3.1 Overview

Entities that may receive signed messages shall have access to the following caches and stores:

— The recently received message cache.
— The root certificate store.
— The revoked certificate store.

Entities shall have access to the following services:

— A cryptographic implementation, hardware or software, that implements the verification algorithm.
— The current position and time.

Additionally, entities may have access to the following information:

— The message certificate cache.
— The "CA certificate cache.

On receiving a `SignedMessage`, an entity (the "receiver") shall use the following process, or its equivalent, in processing it.

a) Decode (in the sense of 4.1) the received octet string to create a populated `SecuredMessage` structure containing a `SignedMessage`. If the received octet string does not represent a correctly encoded `SecuredMessage` containing a `SignedMessage`, discard the octet string.
b) If the `application.type` field in the signed message is not `fully_specified`, discard the message.
c) Perform the message freshness checks described in 7.3.3.2 to ensure that the message is not a replay of a previously received message.
d) If the `transmission_location` is included in the `SignedMessage`, perform the geographical validity checks described in 7.3.3.3.
e) If the `signer.type` field in the received message is `certificate_digest`, use the `signer.digest` to retrieve the certificate from the message certificate cache. If the `signer.digest` field does not map to a known certificate, or if the receiver is not maintaining a message certificate cache, discard the message.
f) Verify that the message signing certificate `subject_type` is `rsu`, `psobu`, or `obu_identified`. If it is not, discard the message.

g) If the message signing certificate does not appear in the message certificate cache, or if the receiver is not maintaining a message certificate cache, or if a CRL has been received since the last time the sender's certificate was used, construct and validate the certificate chain as described in 7.3.3.4. If the certificate chain construction algorithm outputs "failure", discard the message.
h) If the message signing certificate contains a geographic scope restriction and the `transmission_location` is present in the `SignedMessage`:
   1) Verify that the `transmission_location` is within the `GeographicRegion` of the message signing certificate (or the associated `GeographicRegion`, as described in 7.3.3.4.2). If not, discard the message.
   2) If a certificate chain has been constructed, verify that the `transmission_location` is within the `GeographicRegion` of all the certificates in the chain.
i) Denote the `application` field in the message by *aidM*, and the `applications` field in the certificate (or the associated `applications` field, as described in 7.3.3.4.2) by *appC*. Verify that *aidM* is consistent with *appC* as follows:
   1) If *appC* contains an `ApplicationID` *aidC* such that *aidC*.`type` = `match_any_ACM`, and *aidC*.`acid` = *aidM*.`acid`, the consistency check passes.
   2) If *appC* contains an `ApplicationID` *aidC* such that *aidC*.`type` = `fully_specified`, and *aidC*.`acid` = *aidM*.`acid`, and *aidC*.`acm` = *aidM*.`acm`, the consistency check passes.
   3) Otherwise, the consistency check fails. Output "failure" and discard the message.
j) If this is the first time the message signing certificate has been seen, if no message signing certificate cache is being maintained, or if a CRL has been received since the last time the sender's certificate was used, verify that none of the certificates in the certificate chain have been revoked as described in 7.3.3.6. If any certificates have been revoked, discard the message.
k) If a certificate chain has been constructed, verify the signature on each certificate with the public key from its issuing certificate. If any of the verifications fail, discard the message.
l) Verify the signature on the message with the public key from the message signer's certificate. If the signature does not verify, discard the message.
m) If all the previous tests verify, cache any previously unseen certificates along with their associated Application IDs and associated Geographic Regions. The message signer's certificate may also be cached in the potential encrypted recipients certificate cache. Output the following information:
   1) The `application_data` field from the signed message.
   2) The certificate associated with the signed message.
   3) The `generation_time` associated with the signed message.
   4) The `transmission_location` associated with the signed message, or an indication that no `transmission_location` was present.
   5) The ACID and ACM associated with the signed message.

7.3.3.2 Message freshness checks

An application that receives a `SignedMessage` shall ensure that the message is freshly generated, and not a replay of a previous message.

Broadcast applications shall prevent replay by timestamping the messages and checking that a received message is fresh (has a recent timestamp) and is not a duplicate of a previously received message, following the process outlined in this clause.

A transactional application that receives `SignedMessages` shall prevent replay attacks using either the timestamping mechanism described in this clause or an application-specific mechanism based on cryptographic linkage of messages.

An application that uses timestamping to prevent replay shall maintain a cache of messages whose `generation_time` *t* differs from the current time at the unit by no more than a tolerance *r*, as described in 7.2.5. On receiving a `SignedMessage`, the application shall take the following actions:

a) If the `use_generation_time` flag is not set, discard the message as badly formed.
b) Extract the `generation_time` *t* from the message.

c) If $t < T - r$ discard the message as out of window.
d) If $t > T + r$ discard the message as out of window.
e) If a copy of the `SignedMessage` is in the recently received message cache, discard the message as a replay.
f) If the message has not been discarded, add it to the recently received message cache.

There are multiple ways that replay protection via cryptographic linkage can be implemented in a transactional application. The choice of mechanism in this case is application-specific and is not discussed further in this standard.

NOTE 1—The comparison in step e) is over the entire body of the `SignedMessage`, not simply over the application_data field (an efficient way to implement the recently received message cache would be to hash the entire `SignedMessage` and cache only the hash). Therefore, this check will not discard a message with repeated contents if the timestamp and signature are fresh.

NOTE 2—The check described here protects an application from processing a message that it has already received. In addition to this check, the application may have its own freshness criteria that are stricter than the replay criteria described in this clause. For example, it may require a message to be no more than five seconds old. These freshness criteria are an application-specific matter and are out of scope for this document. Additionally, an application may choose to reject a message if the message contents are repeated, even if the message has a new timestamp and signature and so passes the checks described in this clause. This is also out of the scope of this document.

NOTE 3—The following is an example of using a challenge-response mechanism to prevent replay attacks:

1) Both parties include some non-repeating value in their initial messages (either the payload or the header/footer).
2) On sending a message, the transmitter includes a field that cryptographically links it to the preceding message in the transaction.
3) On receiving the message, the receiver checks that cryptographic linkage and discards the received message if the check fails.

There are multiple ways that replay protection via cryptographic linkage can be implemented in a transactional application. The choice of mechanism in this case is application-specific and is not discussed further in this standard.

7.3.3.3 Message location checks

A WAVE unit shall make the current location available to all applications that it hosts. The location information shall include the estimated current 3D location and the estimated horizontal and vertical location accuracy.

An implementation may choose to maintain a geographic tolerance *G*. *G* need not be fixed and may depend on the location accuracy. An application may check whether the `transmission_location` in a received message is greater than *G* from its current location and discard the message if so.

NOTE—This standard does not specify how the current location shall be made available. A device may use a combination of GPS, differential correction, dead reckoning and other techniques as appropriate, so long as at any point the device is able to estimate the accuracy of its location information.

7.3.3.4 Constructing and validating the certificate chain

7.3.3.4.1 Construction

Implementations shall follow a procedure equivalent to the following algorithm to construct the certificate chain. The purpose of the algorithm is to construct a path from the received certificate to a root certificate that is already known to the implementation.

If the `signer.type` field in the `SignedMessage` is `certificate`, run the following algorithm with "the input certificate" set to the certificate from the message, and "the input caches" set to the CA certificate cache and the root certificate store.

a) Set $i = 0$. Initialize `path` as an empty certificate vector. Set C = the input certificate.
b) Do:
   1) Set path[i] = C.
   2) For all certificates in the input caches, see if the low-order 8 bytes of the SHA-256 hash of the cached certificate match the `signer_id` field in the certificate C. If there is a match, call the matching certificate C'. If there is no match, output "failure" and end.
   3) If C' is in the root certificate store, set `path`[i+1] = C' and exit.
   4) Set $C = C$'. Set $i = i$+1.
   5) If i is greater than the maximum path length supported, output "failure" and end.
   6) Return to step b1).

If the `signer.type` field in the `SignedMessage` is `certificate_chain`:

a) Parse the `certificates` field of the signer structure into a series of k certificates.
b) Run the algorithm above with "the input certificate" set equal to the first certificate in the `certificates` field, and "the input caches" set to the CA certificate cache, the root certificate store, and the remaining k–1 certificates from the message.

This procedure outputs either an error or a certificate chain of length i. An output certificate chain shall be validated, as described in 7.3.3.4.2. It shall also be verified by verifying the signatures on all of the individual steps, as described in step j) of the algorithm given in 7.3.3.1.

Conformant implementations shall support constructing a certificate chain of length 5 or less, and may support longer certificate chains.

7.3.3.4.2 Validation

Once the certificate chain has been constructed, the receiver shall take the following actions:

a) Check that all the certificates in the certificate chain except for the message signing certificate and the root certificate have `subject_type` equal to `ca`. If this condition is not met, output "failure" and exit.
b) Check that all the CA certificates in the certificate chain (in other words, all the certificates except for the message signing certificate) contain the `subject_type` of the message signing certificate in their `scope.tf` field. If this condition is not met, output "failure" and exit.
c) For each (issuer, subject) pair in the certificate chain, starting with the root certificate and the certificate that it issued, ensure that the application scope is consistent using the methods of 7.3.3.5. Optionally, also check that the geographic scope is consistent using the methods of 7.3.3.6. This process will output the associated `applications` and `region` field for each certificate. If the consistency check fails for any pair, output "failure" and exit.

7.3.3.5 Application scope consistency check

The check that an `applications` field in a certificate A is consistent with its issuing certificate B consists of the following steps. This algorithm takes as input: the issuing certificate B, the subject certificate A, and, if the `applications` field of B is a single `ApplicationID` with `type = from_issuer`, the *associated* `applications` field for B. The associated `applications` field will have been obtained from a previous run of this check, as described below.

a) If B has `subject_type = root_ca` and the `applications` field of B contains an `ApplicationID` with type = `from_issuer`, the consistency check fails.

b) If the `applications` field of B is a single `ApplicationID` with `type` = `from_issuer`, set *appB* equal to the associated `applications` field that was passed in as an input. Otherwise, set *appB* equal to the `applications` field of B.
c) If the `applications` field in A contains multiple `ApplicationIDs` and one of them has `type` = `from_issuer`, the consistency check fails. (In other words, if A has type `from_issuer` it cannot specify additional applications).
d) If the `applications` field in A contains a single AppplicationID with type = `from_issuer`:
    1) Set the associated `applications` field for A to *appB*.
    2) The consistency check passes. Output "pass" and the associated `applications` field for A.
e) For each `ApplicationID` *aidA* in the `applications` field in A:
    1) If *appB* contains an `ApplicationID` *aidB* such that *aidB*.`type` = `match_any_ACM`, and *aidB*.`acid` = *aidA*.`acid`, the consistency check passes for that *aidA*.
    2) If *appB* contains an `ApplicationID` *aidB* such that *aidB*.`type` = `fully_specified`, and *aidB*.`acid` = *aidA*.`acid`, and *aidB*.`acm` = *aidA*.`acm`, the consistency check passes for that *aidA*.
    3) Otherwise, the consistency check fails for that *aidA*.
f) If the consistency check has failed for any of the `ApplicationIDs` in A, output "fail". Otherwise, output "pass".

7.3.3.6 Geographic scope consistency check

The check that a `region` field in a certificate A is consistent with its issuing certificate B consists of the following steps. This algorithm takes as input: the issuing certificate B, the subject certificate A, and, if the `region` field of B is of `type` = `from_issuer`, the *associated* `region` field for B. The associated `region` field will have been obtained from a previous run of this check, as described below.

a) If B has `subject_type` = `root_ca` and the `region` field of B has type = `from_issuer`, the consistency check fails.
b) If the `region` field of B has `type` = `from_issuer`, set *rB* equal to the associated `region` field that was passed in as an input. Otherwise, set *rB* equal to the `region` field of B.
c) If the `region` field in A has type = `from_issuer`:
    1) Set the associated `region` field for A to *rB*.
    2) The consistency check passes. Output "pass" and the associated `region` field for A.
d) If no part of the region field of A falls outside *rB*, output "pass". Otherwise, output "fail".

7.3.3.7 Certificate revocation list check

When a certificate chain is constructed for the first time, all of the certificates shall be checked against the revoked certificate store to ensure they have not been revoked. If a received message uses a message signing certificate that is in the cache, but a CRL has been received since the cached certificate was last used, the certificate chain shall be constructed and all of the certificates shall be checked against the CRLs that have been received since that certificate was last used.

To check a set of certificates against a set of CRLs, where each CRL is the most recently received with its (`signer_id`, `crl_series`) tuple, an entity shall take the following steps:

a) Select a tolerance *T*.
b) For each certificate:
    1) Mark the certificate as "not revoked".
    2) Set *CertHash* equal to the 10 low-order octets of a SHA-256 digest of the certificate.
    3) For each CRL:
        i) Compare the `signer_id` in the certificate to the `ca_id` in the CRL. If they differ, the CRL does not apply to this certificate. Move to the next CRL.
        ii) Compare the `crl_series` in the certificate to the `crl_series` in the CRL. If they differ, the CRL does not apply to this certificate. Move to the next CRL.

iii) If the CRL expiry date is more than *T* time in the past, mark the certificate as "revoked" and move to the next certificate.
iv) Search through the entries field in the CRL to see if any of the `cert_hash` fields match the *CertHash* calculated in step a2). If there is a match, mark the certificate as "revoked".

c) If any certificate is marked "revoked", output "revoked". Otherwise, output "not revoked".

The tolerance *T* is implementation-specific and may vary. For example, it may depend on the observed frequency with which CRLs with that (`signer_id`, `crl_series`) tuple are received.

Processing of received CRLs by the security manager is described in 8.3.3.1.

7.4 Processing Encrypted Messages

7.4.1 General

An implementation may support sending encrypted messages, receiving encrypted messages, both, or neither. If it supports sending encrypted messages it shall do so using the procedures described in 7.4.2. If it supports receiving encrypted messages it shall do so using the procedures described in 7.4.3.

7.4.2 Transmission Processing

7.4.2.1 General

To send an encrypted message, an implementation takes the following or equivalent steps. The inputs to the encryption process are the identity of the recipient, the message to be encrypted, and a flag stating whether or not the message has already been processed by the security layer.

a) Retrieve the certificate of the desired recipient from the potential encrypted recipients certificate cache. Check that the recipient certificate contains a key for a public-key encryption algorithm. If not, abort.
b) Check that the recipient's certificate has not been revoked.
c) Select a symmetric encryption algorithm from the `supported_symm_algss` field in the recipient's certificate.
d) Generate a random key for this algorithm.
e) Retrieve and encode the message to be encrypted.
f) Encrypt the message.
g) Create a `RecipientInfo` field. Create and encode the `EncryptedMessage` with the `FurtherSecurityProcessing` flag set as appropriate.

7.4.2.2 AES-CCM encryption

To encrypt a message with AES-CCM, the sender shall use the mechanism defined in NIST SP 800-38C. If more than one message is encrypted with the same AES-CCM key, the sender shall use a different nonce for each message.

The formatting mechanism used shall be the one described in Appendix A.2 of NIST SP 800-38C, with the following specific choices:

Control Information and Nonce (A.2.1): There is no associated data, so *Adata* = 0. The MIC length (called "MAC length" in NIST SP 800-38C) shall be 128 bits (16 octets). The octet length of the nonce N shall be 12, leaving three octets to encode the length of the message.

Formatting of the Associated Data (A.2.2): There shall be no associated data.

The counter block generation mechanism used shall be the one described in Appendix A.3 of NIST SP 800-38C.

The input to AES-CCM encryption with no associated data is the nonce *N* and the payload *P* of length *Plen* bits. The output is the ciphertext *C* of length *Clen* = *Plen*+*Tlen* bits.

The output of AES-CCM encryption shall be encoded in an `AESCCMCiphertext` type. The `nonce` field shall be set equal to the 12-octet nonce. The `ciphertext` field shall be set equal to the ciphertext *C*. The length in the ciphertext field shall be set equal to the encoded value of *Clen*/8.

7.4.2.3 Encrypting a key with ECIES

A symmetric key shall be encrypted using the Elliptic Curve Integrated Encryption Scheme as specified in IEEE Std 1363a-2004. The following specific choices are the only ones supported in this standard.

- The secret value derivation primitive shall be ECSVDP-DHC.
- The message encryption method shall be a stream cipher based on KDF2. KDF2 shall be parameterized by the choice
  - *Hash* = SHA-256.
- The message authentication code shall be MAC1. MAC1 shall be parameterized by the choices
  - *Hash* = SHA-256
  - *tBits* = 128.
- Encryption shall use non-DHAES mode.
- The elliptic cuve points shall be converted to octet strings using LSB compressed representation.

The output of this encryption is a triple (*V*, *C*, *T*), where

- *V* is an octet string representing the sender's ephemeral public key.
- *C* is the encrypted symmetric key
- *T* is the authentication tag.

The output of this encryption is a triple (*V*, *C*, *T*), where

7.4.3 Reception Processing

7.4.3.1 General

On receipt of an encrypted message, an implementation takes the following or equivalent steps:

a) Decode (in the sense of 4.1) the received octet string to create a populated `SecuredMessage` structure containing an `EncryptedMesage`. If the received octet string does not represent a correctly encoded `SecuredMessage` containing an `EncryptedMessage`, discard the octet string.
b) For each `RecipientInfo` in the `recipients` field, check whether the `cert_id` in that `RecipientInfo` matches the `cert_id` of any certificate for which the receiving implementation knows the private key. Output "failure" and exit if there is no match. If there is a match, output "failure" and exit if any of the following conditions hold:
   - The matched certificate has expired
   - The matched certificate has been revoked
   - The matched certificate does not contain a public key for an encryption algorithm
   - The `symm_alg` field in the `RecipientInfo` does not match one of the algorithms identified in the supported_symm_algs field in the matched certificate
c) Decrypt the `encrypted_key` using the associated asymmetric decryption algorithm. In the case of this standard, the only supported asymmetric algorithm is ECIES as specified in 7.4.2.3. If the decryption of the symmetric key fails, output "failure" and exit.

d) Decrypt and authenticate the message using the extracted symmetric key. In the case of this standard, the only supported symmetric decryption and authentication method is AES-CCM with 128-bit keys, specified in 7.4.3.2. If the decryption process outputs "failure", output "failure" and exit.
e) If the `content_type` field in the `EncryptedMessage` was set equal to `signed`, process the output decrypted message as a `SignedMessage`. Otherwise, pass the decrypted message to the application.

7.4.3.2 AES-CCM decryption

The sender shall decrypt a message with AES-CCM following NIST SP 800-38C. The formatting mechanism used shall be the one described in Appendix A.2 of NIST SP 800-38C. There shall be no associated data. The MIC length *Tlen* shall be 16 octets.

The inputs to AES-CCM decryption are the purported ciphertext *C* of length *Clen* bits and the nonce *N*. These are extracted from the `AESCCMCiphertext` as follows:

a) The nonce *N* shall be set equal to the contents of the `nonce` field.
b) The ciphertext C shall be set equal to the contents of the `ciphertext` field.
c) The ciphertext length *Clen* shall be set equal to eight times the encoded length of the `ciphertext` field.

7.5 Processing Signed and Encrypted Messages

All implementations that support sending encrypted messages shall also support sending signed and encrypted messages. Likewise, all implementations that support receiving encrypted messages shall also support receiving signed and encrypted messages.

To send a signed and encrypted message, an implementation shall take the following or equivalent steps:

a) Create a signed message from the original message, as described in 7.3.2.
b) Create an encrypted message from the encoded `SignedMessage`, as described in 7.4.2. The inputs to the encryption process are the encoded signed message, the identity of the recipient, and a flag indicating that the message has already been processed by a security layer. The output `EncryptedMessage` shall have `content_type` field equal to `signed`.

On receipt of a signed and encrypted message, an implementation takes the following steps:

a) Decrypt the message, as described in 7.4.3. If this outputs "failure", output "failure" and exit.
b) Since the `content_type` field is `signed`, process the enclosed encrypted message as described in 7.3.3. If this outputs "failure", output "failure" and exit.
c) Forward the output message to the application for processing.

8. Specific uses of secured messages

8.1 Secured WSAs

8.1.1 General

In the WAVE system, a provider offers a service by means of WAVE Service Announcements (WSAs), announcing the availability of a WAVE Basic Service Set (WBSS). A WSA is transmitted in a WAVE Service Information Element (WSIE). If a user determines that one of the services on offer is of interest, they may join the WBSS, enabling them to use the desired service. The WSAs are secured to protect

potential users against joining invalid WBSSes, in other words from responding to WSAs that were not sent by valid providers.

The entity on the provider that generates, and on the user that processes, the WSA is known as the WAVE Management Entity (WME).

IEEE P1609.3 gives specifications for the construction of WSIEs by providers, and for the processing of received WSIEs by users. These specifications omit details of the security processing. This clause describes that security processing.

In outline, a WSIE is constructed as follows.

a) The WME constructs a PstEntry field for each application that is to be offered. This lists information about the application, including the ACID, the ACM, and an Application Priority.
b) The WME constructs a ProviderServiceTable (PST). This consists of the PstEntry for each application on offer and a CitEntry for each service channel that services are offered on.
c) The WME constructs a WSA. This consists of the PST and an optional WAVE Routing Advertisement (WRA).
d) The WME signs the WSA to create a Secured WSA.
e) The MLME adds a header containing timing quality to the Secured WSA to create the WSIE.

All implementations shall support receiving a secured WSA following the procedures of 8.1.3. An implementation may also support sending a secured WSA following the procedures of 8.1.2.

8.1.2 Sending a secured WSA

8.1.2.1 General

All WSIEs shall contain a secured WSA. A secured WSA is a `SecuredMessage` containing a `SignedMessage`, where the `application_data` field in the `SignedMessage` is the original, unsecured WSA. The `generation_time` field in the `SignedMessage` shall be the time the WSA was signed, not the time the WSIE was transmitted. The WME shall have a policy that sets the maximum lifetime of a secured WSA. This lifetime shall be no more than five seconds, to reduce the potential damage from attacks based on replaying a secured WSA. Before the expiry of the current secured WSA, or whenever the contents of the (unsecured) WSA change, the WME shall generate a fresh secured WSA to replace the out of date or altered one. The WME shall ensure that the transmitted WSIE always contains the most recently secured WSA.

The signature shall be generated by a signing key that is reserved for signing WSAs.

8.1.2.2 Format of secured WSA

IEEE P1609.3 describes a secured WSA as consisting of:

— The Security Header
— The WSA itself
— The Security Footer

In the terminology of this standard, the Security Header consists of:

— The `protocol_version` and `type` fields of the `SecuredMessage` structure.
— The `signer` field of the `SignedMessage` structure.
— The `application` and `mf` fields of the `ToBeSignedMessage` structure.

The WSA is carried in the `application_data` field of the `ToBeSignedMessage` structure.

The Security Footer consists of:

— The generation_time (if appropriate), expiry_time (if appropriate) and transmission_location fields of the ToBeSignedMessage structure.
— The signature field of the SignedMessage structure.

The WME shall set the fields as follows:

— protocol_version is 1.
— type is signed.
— application consists of the secured WSIE AID (24) and the ACM 0 (encoded as 01 00).
— The expires flag and the use_location flag are set, i.e. the flags field is 01 06.
— application_data contains the WSA.
— transmission_location is generated as standard for SignedMessages. See 7.3.2 for more details.
— generation_time is the time the secured WSA was generated.
— expiry_time is the time that the secured WSA expires.
— signer shall either have type set to certificate(0) and contain the WME's WSIE Certificate, or have type set to certificate_chain(2) and contain the WME's certificate chain. If type is set to certificate_chain, the certificate chain shall include at least the WME's WSIE certificate and its issuer's certificate. The WME's WSIE certificate is described in 8.1.2.3.
— The signature is generated by the WME using a signing key.

8.1.2.3 WSIE certificate

The certificate included in the signer field of the secured WSA is a standard WAVECertificate. The certificate shall have the following field values set to be considered a valid certificate for signing WSIEs.

— The version is 1.
— The subject_type is wsa_signer(3).
— The type_specific_data field includes a IdentifiedScope.
— The IdentifiedScope may have the subject_name field be empty (encoded as 00).

The WME shall only sign WSAs that it is authorized to sign by the scope field of its certificate. The WME is authorized to sign the WSA if it is authorized to sign every PstEntry in the ProviderServiceTable in the WSA. The WME is authorized to sign a given PstEntry if one of the following conditions hold:

— The scope of the WSA Signer certificate contains an AppIDAndPriority such that all of the following hold:
  — The application.type field is fully_specified.
  — The application.acid field is the same as the ACID in the PstEntry.
  — The application.acm field is the same as the ACM in the PstEntry.
  — The maxPriority field is greater than or equal to the priority in the PstEntry.

or

— The scope of the WSA Signer certificate contains an AppIDAndPriority such that all of the following hold:
  — The application.type field is match_any_acm.
  — The application.acid field is the same as the ACID in the PstEntry,
  — The maxPriority field is greater than or equal to the priority in the PstEntry.

8.1.3 WSIE processing on reception

8.1.3.1 General

A user shall maintain a cache of all Secured WSAs whose `generation_time` is within a tolerance *t* of the time as known to the user. The tolerance *t* may vary according to local conditions. In particular, *t* may vary according to how accurate the user believes its own clock to be. The tolerance *t* shall be no less than 5 seconds and no more than 30 seconds.

A user may also maintain a cache of the *n* most recently received valid WSIE certificates. Whether or not to maintain this cache, and the size of *n* if the cache exists, are implementation-specific decisions.

When a user extracts a Secured WSA from a received WSIE, it shall process it as follows:

a) Decode (in the sense of 4.1) the Secured WSA to create a populated `SecuredMessage` structure containing a `SignedMessage`. If the received octet string does not represent a correctly encoded `SecuredMessage` containing a `SignedMessage`, discard the octet string.
b) Check that the fields are set according to the description in 8.1.2.2. If not, discard the Secured WSA.
c) If the `application.type` field in the signed message is not `fully_specified`, discard the message.
d) If the `use_generation_time` flag is not set in the SignedMessage, discard the message.
e) Perform the timestamping-based message freshness checks described in 7.3.3.2 to ensure that the message is not a replay of a previously received message. In this case, the tolerance *r* shall be no more than 5 seconds or the estimated error in the current local time, whichever is greater.
f) If the `transmission_location` is included in the `SignedMessage`, perform the geographical validity checks described in 7.3.3.3.
g) If the `signer.type` field in the received message is `certificate_digest`, use the `signer.digest` to retrieve the certificate from the message certificate cache. If the `signer.digest` field does not map to a known certificate, or if the receiver is not maintaining a message certificate cache, discard the message.
h) Verify that the WSA signing certificate has `subject_type` `wsa_signer`. If it does not, discard the message.
i) If the message signing certificate does not appear in the message certificate cache, or if the receiver is not maintaining a message certificate cache, or if a CRL has been received since the last time the sender's certificate was used, construct and validate the certificate chain as described in 7.3.3.4, except that when validating the certificate chain as described in 7.3.3.4, replace the application scope consistency check of step c) in 7.3.3.4.2 with the AppIDAndPriority consistency check in 8.1.3.2. If the certificate chain construction algorithm outputs "failure", discard the message.
j) If the message signing certificate contains a geographic scope restriction and the `transmission_location` is present in the `SignedMessage`:
    1) Verify that the `transmission_location` is within the `GeographicRegion` of the message signing certificate (or the associated `GeographicRegion`, as described in 7.3.3.4.2). If not, discard the message.
    2) If a certificate chain has been constructed, verify that the `transmission_location` is within the `GeographicRegion` of all the certificates in the chain.
k) Verify that the application permissions in the WSA are consistent with the permissions in the certificate using the consistency checks described in 8.1.2.3. If the consistency check fails, output "failure" and discard the message.
l) If this is the first time the message signing certificate has been seen, if no message signing certificate cache is being maintained, or if a CRL has been received since the last time the sender's certificate was used, verify that none of the certificates in the certificate chain have been revoked as described in 7.3.3.6. If any certificates have been revoked, discard the message.
m) If a certificate chain has been constructed, verify the signature on each certificate with the public key from its issuing certificate. If any of the verifications fail, discard the message.
n) Verify the signature on the message with the public key from the message signer's certificate. If the signature does not verify, discard the message.

o) If all the previous tests verify, cache any previously unseen certificates along with their associated Application IDs and associated Geographic Regions. The message signer's certificate may also be cached in the potential encrypted recipients certificate cache. Output the following information:
   1) The `application_data` field from the signed message
   2) The certificate associated with the signed message.
   3) The `generation_time` associated with the signed message.
   4) The `transmission_location` associated with the signed message, or an indication that no `transmission_location` was present.
   5) The ACID and ACM associated with the signed message.

If all of these steps have been carried out without discarding the Secured WSA, the Secured WSA is considered valid and the WME may continue processing the WSA contained within the Secured WSA.

The WME shall extract the WSIE certificate from the Secured WSA. If the WME generates a WME-Notification.indication as a result of processing a WSA, the WME-Notification.indication shall include the WSIE certificate from the Secured WSA that contained that WSA.

8.1.3.2 AppIDAndPriority scope consistency check

The check that an `apps_and_priorities` field in a certificate A is consistent with its issuing certificate B consists of the following steps. This algorithm takes as input: the issuing certificate B, the subject certificate A, and, if the `apps_and_priorities` field of B is a single `AppIDAndPriority` with `type = from_issuer`, the *associated* `apps_and_priorities` field for B. The associated `apps_and_priorities` field will have been obtained from a previous run of this check, as described below.

a) If B has `subject_type = root_ca` and the `apps_and_priorities` field of B contains an `AppIDAndPriority` with type = `from_issuer`, the consistency check fails.
b) If the `apps_and_priorities` field of B is a single `AppIDAndPriority` with `type = from_issuer`, set *appB* equal to the associated `apps_and_priorities` field that was passed in as an input. Otherwise, set *appB* equal to the `apps_and_priorities` field of B.
c) If the `apps_and_priorities` field in A contains multiple `AppIDAndPrioritys` and one of them has `type = from_issuer`, the consistency check fails. (In other words, if A has type `from_issuer` it cannot specify additional apps_and_priorities).
d) If the `apps_and_priorities` field in A contains a single `AppIDAndPriority` with type = `from_issuer`:
   1) Set the associated `apps_and_priorities` field for A to *appB*.
   2) The consistency check passes. Output "pass" and the associated `apps_and_priorities` field for A.
e) For each `AppIDAndPriority` *aidA* in the `apps_and_priorities` field in A:
   1) If *appB* contains an `AppIDAndPriority` *aidB* such that *aidB*.`type = match_any_ACM`, and *aidB*.`acid` = *aidA*.`acid`, and *aidB*.`maxPriority` is greater than or equal to *aidA*.`maxPriority`, the consistency check passes for that *aidA*.
   2) If *appB* contains an `AppIDAndPriority` *aidB* such that *aidB*.`type = fully_specified`, and *aidB*.`acid` = *aidA*.`acid`, and *aidB*.`acm` = *aidA*.`acm`, and *aidB*.`maxPriority` is greater than or equal to *aidA*.`maxPriority`, the consistency check passes for that *aidA*.
   3) Otherwise, the consistency check fails for that *aidA*.
f) If the consistency check has failed for any of the `AppIDAndPrioritys` in A, output "fail". Otherwise, output "pass".

8.2 Secured WSMs

8.2.1 Overview

The WAVE Short Message (WSM) format is defined in IEEE P1609.3. The WSM information passed between the WSMP stack and the LLC is shown in Figure 1, with lengths in octets.

1	1	1	1	1	1	1	var.	2	var
WSM Version	Security Type	Channel Number	Data Rate	TxPwr_ Level	Application Class Identification	ACM Field Length	ACM Contents	WSM Length	WSM Data

Figure 1—WAVE Short Message format

An application sends a WAVE short message by generating a WSM-WaveShortMessage.request primitive:

WSM-WaveShortMessage.request
(
ChannelInfo,
WsmVersion,
SecurityType,
ApplicationClassIdentifier,
ApplicationContextMark,
TransmissionPriority,
Length,
Data,
Peer MAC address
)

The WSMP generates WSM-WaveShortMessage.indication primitive to deliver a WSM to a registered application:

WSM-WaveShortMessage.indication
(
ChannelInfo,
WsmVersion,
SecurityType,
ApplicationClassIdentifier,
ApplicationContextMark,
TransmissionPriority,
Length,
Data,
Peer MAC address
)

A `SignedMessage` contains the ApplicationClassIdentifier and ApplicationContextMark, as well as an indication of the length, in its headers. However, this information is also included in the parameters of the request and indication primitives. This standard defines the SignedWSM format to save bandwidth by avoiding duplication of this information.

In outline, a sender signs a `SignedWSM` by calculating the signature on a standard `ToBeSignedMessage` structure that includes the contents of the `ToBeSignedWSM` and the ApplicationClassIdentifier, ApplicationContextMark, and Length data from the WSM headers. The `Data` field of the WSM contains the `SignedWSM`, and the `SecurityType` field is set to `signed(1)`.

On receiving a WSM with `SecurityType` field set equal to `signed(1)`, the receiver uses the `ToBeSignedWSM` from the `SignedWSM` and the `ApplicationClassIdentifier` and `ApplicationContextMark` from the WSM headers to reconstruct the `ToBeSignedMessage` structure and verify the signature. Apart from this straightforward bit-manipulation, the processing is the same as for the standard `SignedMessage`.

An encrypted WSM is simply a WSM where the `SecurityType` field is `encrypted(2)` and the `Data` field is an `EncryptedMessage`.

A signed and encrypted WSM has an `EncryptedMessage` in the `Data` field. This `EncryptedMessage` decrypts to a `SignedWSM`, which is then processed as described in this clause.

All implementations shall support receiving signed WSMs. An implementation may also support any or all of sending signed WSMs, sending encrypted WSMs, and receiving encrypted WSMs. An implementation that supports sending (resp. receiving) signed WSMs and encrypted WSMs shall also support sending (resp. receiving) WSMs that are both signed and encrypted.

8.2.2 Signed WSM

8.2.2.1 Sending a signed WSM

An implementation that sends a signed WSM shall follow this procedure.

Input:

— ChannelInfo, determined according to IEEE P1609.3
— WsmVersion, determined according to IEEE P1609.3
— ApplicationClassIdentifier
— ApplicationContextMark, encoded so that the first two octets represent the length and the remaining two octets represent the contents of the ApplicationContextMark
— TransmissionPriority, determined according to IEEE P1609.3
— Data
— Length of data *l*
— Peer MAC address
— Indication of whether transmission time, location, and expiry are to be included

Procedure:

a) Construct a `ToBeSignedMessage` as described in 7.3.2
   1) The `application` field shall have type set to fully_specified
   2) The `acid` and `acm` fields shall be identical to the `ApplicationClassIdentifier` and `ApplicationContextMark` inputs to this procedure
   3) The `application_data` shall consist of the input data, encoded so as to be preceded by a two-octet representation of its length *l* (that is to say, standard encoding for the contents of an `opaque` field)
b) Sign the `ToBeSignedMessage` to obtain a `SignedMessage`
c) Construct a `SignedWSM` from the `SignedMessage`
   1) Remove the `application` field from the `SignedMessage` and append the data that follows the `application` field to the data that precedes the `application` field
   2) Remove the two-octet encoded length from the start of the `application_data`. Append the data that follows the encoded `application_data` length to the data that precedes the encoded `application_data` length
d) Generate a WSM-WaveShortMessage.request service primitive with the parameters set as follows:
   1) ChannelInfo = the input ChannelInfo
   2) WsmVersion = the input WsmVersion
   3) SecurityType = `signed(1)`
   4) ApplicationClassIdentifier = the `acid` field in the `SignedMessage`
   5) ApplicationContextMark = the data from the `acm` field in the `SignedMesage`

6) TransmissionPriority = the input TransmissionPriority
7) Length = the length of the `SignedWSM`
8) Data = the `SignedWSM`
9) Peer MAC address = the input Peer MAC address

8.2.2.2 Receiving a signed WSM

An implementation that receives a WSM-WAVEShortMessage.indication with `SecurityType` field set to `signed(1)` shall take the following steps:

a) Parse the Data parameter of the WSM-WAVEShortMessage.indication as a `SignedWSM` *s*. To do this it is necessary to calculate the length of the `application_data` field in the `SignedWSM`. This can be done as follows.
   — The `SignerInfo` field identifies the signing algorithm used, and hence the length of the `signature` field. Denote the length of the signature field in octets by l_s. If the signing algorithm is `ecdsa_nistp224_with_sha224`, l_s = 56. If the signing algorithm is `ecdsa_nistp256_with_sha256`, l_s = 64.
   — The length of the `application_data` field in the `SignedWSM` is the number of octets from the start of the `application_data` field to the end of the `SignedWSM`, minus l_s. Denote this length by *l*.
b) Construct an `ApplicationID` *a*, where
   — `type` = `fully_specified`.
   — `acid` = the `ApplicationClassIdentifier` field in the WSM-WAVEShortMessage.indication.
   — `acm` = the (length, data) encoding of the `ApplicationContextMark` field in the WSM-WAVEShortMessage.indication.
c) Construct a `SignedMessage` *m* consisting of:
   — The `signer` field from the `SignedWSM` *s*, concatenated with
   — The encoded `ApplicationID` *a*, concatenated with
   — The message flags from the `SignedWSM` *s*, concatenated with
   — The length of the `application_data`, *l*, concatenated with
   — The application _data itself and the remaining fields from the `SignedWSM` *s*.
d) Process the `SignedMessage` *m* as described in 7.3.3.

8.2.3 Encrypted WSM

8.2.3.1 Sending an encrypted WSM

An implementation that sends an encrypted WSM shall take the following steps:

a) Construct an `EncryptedMessage` as described in 7.4.2.
b) Generate a WSM-WaveShortMessage.request service primitive with the following parameters set as indicated:
   — `SecurityType` = `encrypted(2)`.
   — `Data` = the `EncryptedMessage`.

All other parameters shall be set as appropriate for that application, as described in IEEE P1609.3.

8.2.3.2 Sending a signed and encrypted WSM

An implementation that sends a signed and encrypted WSM shall take the following steps:

a) Construct a `SignedWSM` as described in 8.2.2.1, steps a)–b).
b) Construct an `EncryptedMessage` as described in 7.4.2, where
   — The input to the encryption process is the `SignedWSM`.
   — The `content_type` field in the `EncryptedMessage` is `signed`.

c) Generate a WSM-WaveShortMessage.request service primitive with the following parameters set as indicated:
   — `SecurityType` = `encrypted(2)`.
   — `ApplicationClassIdentifier` = the `acid` field in the `SignedMessage`.
   — `ApplicationContextMark` = the data from the `acm` field in the `SignedMesage`.
   — `Data` = the `EncryptedMessage`.

All other parameters shall be set as appropriate for that application, as described in IEEE P1609.3

8.2.3.3 Receiving an encrypted WSM

An implementation that receives a WSM-WAVEShortMessage.indication with SecurityType field set to `encrypted(2)` shall take the following steps:

a) Parse the Data field as an `EncryptedMessage` . If this parsing fails, discard the message.
b) Decrypt the EncryptedMesasge as described in 7.4.3.
c) If the `content_type` field in the `EncryptedMessage` is app_data, return the data to the application for further processing. If the content_type is `signed`, continue with this algorithm. Otherwise, output "failure" and discard the message.
d) Parse the decrypted data as a `SignedWSM` s. If the parsing fails, output "failure" and discard the message.
e) Continue to process the `SignedWSM` as described in 8.2.2.2, step b) onward.

8.3 Security Manager

8.3.1 Overview

Each implementation shall contain a Security Manager application. This Security Manager shall maintain the root certificate store and the CRL store for use by all applications.

This clause describes mechanisms for distribution of long-lived system information. Doing this ensures that necessary information is available when processing time-critical messages, without requiring that it is distributed at the same time. This allows the system to load-balance by sending out the long-lived information when the network traffic is low.

Management application messages will often be generated and signed by a central server, but transmitted using RSUs or OBUs. As such, the checks on transmission location and time that are usually performed on signed messages do not need to be performed on these messages, although the recipient shall still check that the messages are valid at the time and location at which they were received.

All management application messages shall be signed. A receiving unit shall verify the signature and permissions on the message before acting on its contents.

8.3.2 Application class identification

WSMP messages to the WAVE Security Manager shall be identified by the ACID `security management(23)`.

8.3.3 Certificate revocation lists

8.3.3.1 CRL processing

The CRL format is given in 5.16.

A CRL is processed by the security manager. It shall be transmitted as a WSM. The ACID shall be 23 and the ACM shall be the 32-bit integer 0, encoded as `04 00 00 00 00`. The entire `AlgorithmID` structure is therefore encoded as `23 04 00 00 00`.

When the security manager receives a CRL, it shall take the following actions to verify the signature on the CRL.

a) If the `signer.type` field is `self`, reject the CRL.

b) If the `signer.type` field is `certificate_digest`:
   1) Search the root certificate store for a certificate whose digest equals the `signer.digest` field. If no such certificate is found, reject the CRL.
   2) Use the key from the root certificate to verify the signature on the `unsigned_crl` field of the CRL. If the verification fails, reject the CRL.
   3) Accept the CRL.

c) If the `signer.type` field is `certificate` or `certificate_chain`:
   1) Construct and validate a certificate chain following the procedures of 7.3.3.4. In constructing this certificate chain, the application scope consistency check shall be replaced with the following checks:
      i) A check that the first certificate in the chain has `subject_type` equal to `ca`, `root_ca`, or `crl_signer` and that all the other certificates in the chain have `subject_type` equal to `ca` or `root_ca`.
      ii) If the first certificate in the chain has `subject_type` equal to `crl_signer`, check that the `responsible_series` field of that certificate is equal to the `crl_series` field in the CRL.

      If either of these checks fail, reject the CRL.
   2) Verify the signatures on all the certificates in the chain.
   3) Check the `subject_type` of the first certificate in the chain:
      i) If the first certificate in the chain has `subject_type` equal to `ca` or `root_ca`, calculate the SHA-256 hash of the certificate and check that the `ca_id` in the CRL is equal to the low-order eight octets of the hash value. If this check fails, reject the CRL.
      ii) If the first certificate in the chain has `subject_type` equal to `crl_signer`, calculate the SHA-256 hash of its issuing certificate (the second certificate in the chain) and check that the `ca_id` in the CRL is equal to the low-order eight octets of the hash value. If this check fails, reject the CRL.
   4) Verify the signature on the `unsigned_crl` field of the CRL with the public key from the first certificate in the chain. If the signature fails to verify, reject the CRL.
   5) Accept the CRL.

Once the CRL is accepted, the security manager shall take the following steps:

a) For every certificate on the CRL:
   1) If the certificate does not already appear in the revoked certificate store, add it to the revoked certificate store.

8.3.3.2 Partitioned CRLs

In order to keep down the size of individual CRLs, a CA might partition the certificates that it issues into groups and assign each group to a different CRL series. By limiting the number of certificates in a given group (to, e.g., 10 000), the CA can put a bound on the maximum size of any CRL. In this case, a receiver would only need the relevant CRL for the certificate it was trying to verify rather than all CRLs from a given CA. The `crl_series` value is used for this purpose. The number of CRL series and the partitioning of CRLs into CRL series is a local matter for the CA.

8.3.3.3 Revoked certificate store management

The Security Manager shall periodically review the revoked certificate store and remove from it any certificates that have passed their expiry date.

8.3.4 Root certificate update

A root certificate update shall be processed by the security manager. It may be transmitted over the WSMP stack or over the IP stack. The certificate update message shall be either a WAVESecuredMessage containing a SignedMessage or a signed WSM. The ACID shall be 23 and the ACM shall be the 32-bit integer 1, encoded as `04 00 00 00 01`. The `application_data` field shall consist of the updated root certificate.

8.4 Certificate requests

8.4.1 Overview

Applications may make over-the-air requests for certificates using the `WAVECertificateRequest` message described in 5.17. The certificate request message shall be transferred by UDP per IETF RFC 768, as there is no ACID/ACM identifying the certificate authority. The certificate response message shall also be transferred by UDP. There are two usage models for the certificate request.

First, the certificate request may be self-signed. This means that the key that signs the message is the key that the user is requesting a certificate for. In this case, the CA that receives the certificate request shall perform additional out-of-band checks to ensure that the signing key is actually in the possession of the entity identified in the certificate request.

Second, the certificate request may be signed by a Certificate Signing Request Certificate. This is a certificate with `subject_type` `csr_signer` and a scope of type `CSRSignerScope`. This certificate allows the requester to request a message signing certificate of any type that is allowed by the `CSRSignerScope`.

8.4.2 Certificate request generation

An application that generates a self-signed certificate request may fill in the fields according to local policy. It shall ensure that the scope in the request is consistent with the scope of the CA that will receive the request.

An application that generates a certificate request signed by a CSR signing certificate shall ensure that the scope in the request is consistent with the scope of the CSR signing certificate. This shall amount to checking the following conditions:

— If there is a `subject_name` field in the CSR signing certificate, it is identical to the `subject_name` field in the `ToBeSignedCSR`.
— If there is an `application` field in the CSR signing certificate, the `application` field in the scope of the certificate request is consistent with it as established by the methods of 7.3.3.5.
— If there is a `region` field in the CSR signing certificate, the `region` field in the certificate request represents a region that is completely contained within the `region` in the CSR signing certificate.

8.4.3 Certificate request reception processing

Once a CA receives a CSR, it shall determine whether the requesting unit is eligible to be issued the requested certificate. If the CSR is self-signed, the process for deciding whether or not to issue the certificate is a policy matter and out of scope for this standard.

If the CSR is signed by a valid, non-revoked certificate of type `csr_signer`, the CA shall perform the following checks.

— If there is a `subject_name` field in the CSR signing certificate, it is identical to the `subject_name` field in the `ToBeSignedCSR`.
— If there is an `application` field in the CSR signing certificate, the `application` field in the scope of the certificate request is consistent with it as established by the methods of 7.3.3.5.
— If there is a `region` field in the CSR signing certificate, the `region` field in the certificate request represents a region that is completely contained within the `region` in the CSR signing certificate.

If the certificate can be issued, the CA formats the appropriate `UnsignedCertificate` structure and signs it to generate a `Certificate` value. The CA is not bound by the enrollee's subject type or scope requests. In particular, it may replace any part of the scope with `from_issuer` as appropriate. If issuing a certificate with `subject_type obu_identified`, it shall select the contents of the `cert_specific_data` identifier field according to its own local policies.

The CA may issue a certificate of any type or scope of its choosing, except that:

— It shall not issue a certificate with an invalid `GeographicScope`, in other words:
  — A certificate with a `RectangularRegion` where the `upper_left` value is south or east of the `lower_right` value.
  — A certificate with a `PolygonalRegion` where the region is not simply connected, in other words where two or more of the edges formed by connecting the vertices cross.
— It shall not issue a certificate that would fail the application scope consistency checks of 7.3.3.5.
— It shall not issue a certificate requested by use of a CSR signing certificate such that the application scope of the issued certificate is not consistent with the application scope of the CSR signing certificate.
— It shall not issue a certificate containing a `GeographicScope` if any part of the issued certificate's geographic scope falls outside the CA's geographic scope.

A conformant CA shall sign certificates with keys of type `ecdsa_nistp256_with_sha256`.

8.4.4 Certificate request response

If the CA issues a certificate, it shall respond to the certificate request with a `WAVECertficateResponse` message as described in 5.17.

The `certificate_chain` shall contain the certificate chain of the new certificate. This path is in order, with the most local certificate (the newly issued one) being first and each successive certificate signing the one before it. The path should be complete with the final certificate being a trust anchor. However, some implementations may choose to deliver less complete paths for space reasons.

The `crl_path` shall contain the CRLs necessary to validate the certificate. At minimum, it shall contain the most recent version of the CRL series on which the issued certificate would appear if it were revoked. In addition, CAs should include CRLs corresponding to other CAs in the chain. These CRLs are not ordered.

This certificate response is not a signed message. If transmitted as a WSM, it shall be identified using the ACID 23 and the ACM consisting of the 32-bit integer 2, encoded as `00 00 00 02`.

On receiving a `WAVECertificateResponse`, an application shall:

— Construct and validate the certificate chain as described in 7.3.3.4.

— Verify that the crl_path field contains a CRL such that the ca_id field is the same as the signer_id field in the issued certificate, and both the CRL and the issued certificate have the same value in the crl_series field.

If both of these checks pass, the application shall accept the certificate response.

8.5 Fragmented messages

8.5.1 Overview

In order to maintain the system, some system data (such as CRLs or updated root certificates) needs to be made available to implementations that receive signed messages. This data may be too large to fit in a single transmission. This clause provides a framework for fragmenting and reassembling large messages. Currently, the only supported fragmentation mechanism is the naïve mechanism in which a message is split into *k* equal-size blocks, and *k*+1 symbols are transmitted in which the first *k* symbols are the message blocks and the final symbol is the XOR of all of the message blocks. A recipient who receives any *k* distinct symbols will be able to reconstruct all *k* of the original message blocks.

A conformant implementation shall support reconstructing messages of at least 8 blocks.

8.5.2 Fragmented message syntax

Source symbols are transmitted using the following structure:

```
struct {
            opaque     update_digest[20];
            FECScheme  scheme;
            select (scheme) {
            case (xor_symbols) :
                uint16   k;
                uint16   index;
            }
            opaque symbol<2^16-1>;
        } EncodedSymbol;

enum { xor_symbols(0), (2^8-1) } FECScheme;
```

— `update_digest` is a SHA-256 hash of the original source message, which allows recipients to disambiguate different fragments for reassembly.
— `scheme` identifies the forward error correction scheme under consideration. In this version of the standard it shall always take the value `xor_symbols(0)`.
— `k` identifies how many symbols the message has been split into.
— `index` identifies the index of the symbol transmitted in the `EncodedSymbol`. If `index` is in the range 0 to `k`–1, the encoded symbol is the corresponding segment of the message. If `index` is `k`, the encoded symbol is the XOR of all of the message blocks.
— `symbol` contains the symbol.

8.5.3 Transmitting a fragmented message

This subclause only describes fragmentation of a signed message, such as a `SignedMessage` or a CRL.

To create a fragmented message, a sender shall take the following steps:

a) Create the message contents.
b) Split the message into *k* blocks of size *b*, where *b* is the maximum size of the `application_data` field in a `SignedMessage` that fits into a single WAVE MPDU and *k*,

the number of blocks, is determined by *b* and the length of the original message. If the final block is fewer than *b* bytes in length, pad with zero bytes.

c) Create the additional symbol by XORing all of the *k* blocks.

d) For each symbol, create and encode an `EncodedSymbol` message.

e) For each `EncodedSymbol`, create a `SignedMessage` such that
   — The `application` field is the same as the `application` field in the original message (in the case of a CRL, `acid 23` and `acm` encoded as `04 00 00 00 00`).
   — The flag `fragment(0)` is set.
   — The `application_data` field is the encoded `EncodedSymbol`.
   — The same certificate signs the `SignedMessage` as signed the original message.

The same steps apply if the contents of the original message have been encrypted.

8.5.4 Receiving a fragmented message

Applications that might receive fragmented messages shall maintain a cache of received message fragments.

A unit that receives a `SignedMessage` or signed WSM with the `fragment` flag set in the `mf` field shall take the following actions.

a) Verify the SignedMessage or signed WSM as usual following 7.3.3 or 8.2.2.2, respectively. If the verification fails, discard the fragment.

b) Parse the `application_data` field as an `EncodedSymbol`. If this parsing fails, discard the fragment.

c) Check whether an `EncodedSymbol` with the same `update_digest` and `index` is already present in the fragmented message cache. If it is, discard the symbol and exit. Otherwise, store the encoded symbol and continue.

d) If fewer than k `EncodedSymbols` with the same `update_digest` and distinct `index` values have been received, await the next received `SignedMessage` and return to step a).

e) If one of the k received `EncodedSymbols` has index value equal to k, calculate the symbol with the missing `index` as the XOR of all the symbol fields received with that `update_digest`.

f) Calculate the reconstructed message as the concatenation of all `symbols` with `index` < k in numerical order of `index`.

g) Process the reconstructed message as normal, including verification.

Annex A

(normative)

Protocol Implementation Conformance Statement (PICS) proforma[11]

A.1 General

The supplier of a protocol implementation that is claimed to conform to IEEE Std 1609.2 shall complete the following protocol implementation conformance statement (PICS) proforma.

A completed PICS proforma is the PICS for the implementation in question. The PICS is a statement of which capabilities and options of the protocol have been implemented. The PICS can have a number of uses, including use

a) By the protocol implementer, as a checklist to reduce the risk of failure to conform to the standard through oversight.
b) By the supplier and acquirer, or potential acquirer, of the implementation, as a detailed indication of the capabilities of the implementation, stated relative to the common basis for understanding provided by the standard PICS proforma.
c) By the user, or potential user, of the implementation, as a basis for initially checking the possibility of interworking with another implementation (note that, while interworking can never be guaranteed, failure to interwork can often be predicted from incompatible PICS proformas).
d) By a protocol tester, as the basis for selecting appropriate tests against which to assess the claim for conformance of the implementation.

A.2 Abbreviations and special symbols

A.2.1 Symbols for Status column

M	mandatory
O	optional
O.<n>	optional, but support of at least one of the group of options labeled by the same <n> is required
pred:	conditional symbol, including predicate identification

A.3 Instructions for completing the PICS proforma

A.3.1 General structure of the PICS proforma

The first parts of the PICS proforma, Implementation identification and Protocol summary, are to be completed as indicated with the information necessary to identify fully both the supplier and the implementation.

[11] *Copyright release for PICS proformas:* Users of this standard may freely reproduce the PICS proforma in this annex so that it can be used for its intended purpose and may further publish the completed PICS.

The main part of the PICS proforma is a fixed questionnaire, divided into subclauses, each containing a number of individual items. Answers to the questionnaire items are to be provided in the rightmost column, either by simply marking an answer to indicate a restricted choice (usually Yes or No) or by entering a value or a set or a range of values. (Note that there are some items where two or more choices from a set of possible answers may apply. All relevant choices are to be marked in these cases.)

Each item is identified by an item reference in the first column. The second column contains the question to be answered. The third column contains the reference or references to the material that specifies the item in the main body of this standard. The remaining columns record the status of each item, i.e., whether support is mandatory, optional, or conditional, and provide the space for the answers (see also A.3.4). Marking an item as supported is to be interpreted as a statement that all relevant requirements of the subclauses and normative annexes, cited in the References column for the item, are met by the implementation.

A supplier may also provide, or be required to provide, further information, categorized as either Additional Information or Exception Information. When present, each kind of further information is to be provided in a further subclause of items labeled A<I> or X<I>, respectively, for cross-referencing purposes, where <I> is any unambiguous identification for the item (e.g., simply a numeral). There are no other restrictions on its format or presentation.

The PICS proforma for a station consists of A.4.1 through A.4.4 inclusive, and at least one of A.4.5, A.4.6, or A.4.7 corresponding to the physical layer (PHY) implemented.

A completed PICS proforma, including any Additional Information and Exception Information, is the PICS for the implementation in question.

NOTE—Where an implementation is capable of being configured in more than one way, a single PICS may be able to describe all such configurations. However, the supplier has the choice of providing more than one PICS, each covering some subset of the implementation's capabilities, if this makes for easier and clearer presentation of the information.

A.3.2 Additional information

Items of Additional Information allow a supplier to provide further information intended to assist in the interpretation of the PICS. It is not intended or expected that a large quantity of information will be supplied, and a PICS can be considered complete without any such information. Examples of such Additional Information might be an outline of the ways in which an (single) implementation can be set up to operate in a variety of environments and configurations, or information about aspects of the implementation that are outside the scope of this standard but have a bearing upon the answers to some items.

References to items of Additional Information may be entered next to any answer in the questionnaire, and may be included in items of Exception Information.

A.3.3 Exception information

It may happen occasionally that a supplier will wish to answer an item with mandatory status (after any conditions have been applied) in a way that conflicts with the indicated requirement. No preprinted answer will be found in the Support column for this. Instead, the supplier shall write the missing answer into the Support column, together with an X<I> reference to an item of Exception Information, and shall provide the appropriate rationale in the Exception Information item itself.

An implementation for which an Exception Information item is required in this way does not conform to this standard.

NOTE—A possible reason for the situation described above is that a defect in this standard has been reported, a correction for which is expected to change the requirement not met by the implementation.

A.3.4 Conditional status

The PICS proforma contains a number of conditional items. These are items for which both the applicability of the item itself, and its status if it does apply, mandatory or optional, are dependent upon whether or not certain other items are supported.

Where a group of items is subject to the same condition for applicability, a separate preliminary question about the condition appears at the head of the group, with an instruction to skip to a later point in the questionnaire if the N/A answer is selected. Otherwise, individual conditional items are indicated by a conditional symbol in the Status column.

A conditional symbol is of the form "<pred>:<S>", where "<pred>" is a predicate as described below, and "<S>" is one of the status symbols M or O.

If the value of the predicate is true, the conditional item is applicable, and its status is given by S: the support column is to be completed in the usual way. Otherwise, the conditional item is not relevant and the N/A answer is to be marked.

A predicate is one of the following:

a) An item-reference for an item in the PICS proforma: the value of the predicate is true if the item is marked as supported, and is false otherwise.

b) A Boolean expression constructed by combining item-references using the boolean operator OR: the value of the predicate is true if one or more of the items is marked as supported, and is false otherwise. For compactness, item-references combined with a comma are considered to be combined with the OR operator.

Each item referenced in a predicate, or in a preliminary question for grouped conditional items, is indicated by an asterisk in the Item column.

A.4 PICS proforma—IEEE Std 1609.2[12]

A.4.1 Implementation identification

Supplier	
Contact point for queries about the PICS	
Implementation Name(s) and Version(s)	
Other information necessary for full identification, e.g., name(s) and version(s) of the machines and/or operating systems(s), system names	

NOTE 1—Only the first three items are required for all implementations. Other information may be completed as appropriate in meeting the requirement for full identification.

NOTE 2—The terms *Name* and *Version* should be interpreted appropriately to correspond with a supplier's terminology (e.g., Type, Series, Model).

A.4.2 Protocol summary

Identification of protocol standard	IEEE Std 1609.2
Identification of amendments and corrigenda to this PICS proforma that have been completed as part of this PICS	Amd. : Corr. : Amd. : Corr. :
Have any exception items been required? (See A.3.3; the answer Yes means that the implementation does not conform to IEEE Std 1609.2)	□Yes □No
Date of statement (dd/mm/yy)	

[12] Copyright release for PICS proforma: Users of this standard may freely reproduce the PICS proforma in this annex so that it can be used for its intended purpose and may further publish the completed PICS.

A.4.3 Security configuration (top-level)

This presents a list of the top-level security functionality that an implementation may claim to support.

Item	Security configuration (top-level)	Reference	Status	Suppport
S1	Generate SignedMessage	7.3.2	O	□Yes □No
S2	Receive SignedMessage	7.3.3	M	□Yes □No
S3	Generate Signed WSM	8.2.2.1	O	□Yes □No
S4	Receive Signed WSM	8.2.2.2	M	□Yes □No
S5	Generate EncryptedMessage	7.4.2	O	□Yes □No
S6	Receive EncryptedMessage	7.4.3	O	□Yes □No
S7	Generate Encrypted WSM	8.2.3.1	O	□Yes □No
S8	Receive Encrypted WSM	8.2.3.3	O	□Yes □No
S9	Generate Encrypted and Signed Message	7.5	S1, S7:M	□Yes □No
S10	Receive Encrypted and Signed Message	7.5	S8:M	□Yes □No
S11	Generate Encrypted and Signed WSM	8.2.3.2	S3, S7:M	□Yes □No
S12	Receive Encrypted and Signed WSM	8.2.3.3	S8:M	□Yes □No
S13	Generate WAVECertificateRequest	8.4.2	O	□Yes □No
S14	Receive WAVECertificateRequest	8.4.3	O	□Yes □No
S15	Issue WAVECertificateResponse	8.4.4	S14:M	□Yes □No
S16	Receive WAVECertificateResponse	8.4.4	S13:M	□Yes □No
S17	Issue WAVECRL	8.3.3	O	□Yes □No
S18	Receive WAVECRL	8.3.3	M	□Yes □No
S19	Issue Secured WSA	8.1.2	O	□Yes □No
S20	Receive Secured WSA	8.1.3	M	□Yes □No
S21	Issue fragmented messages	8.5.3	O	□Yes □No
S22	Receive fragmented messages	8.5.4	M	□Yes □No

Item	Security configuration (top-level)	Reference	Status	Suppport
S23	Maintain root certificate store	7.2.2	M	□Yes □No
S24	Maintain CA certificate store	7.2.3	M	□Yes □No
S25	Maintain message signing certificate cache	7.2.4	O	□Yes □No
S26	Maintain recently received message cache	7.2.5	M	□Yes □No
S27	Maintain potential encrypted recipients certificate store	7.2.6	S7:M	□Yes □No
S28	Maintain revoked certificate store	7.2.7	M	□Yes □No
S29	Maintain incoming fragmented message cache	7.2.8	M	□Yes □No

A.4.4 Security configuration (detailed)

This presents a detailed checklist of the functionality that an implementation may claim to support.

In this table, to "encode" a type means to create an octet string containing the components of that type, starting from the encoded fields within that type. To "decode" means to separate the encoded octet string representing a type into octet strings representing the individual fields of that type. In other words, "encode" and "decode" refer to syntactic parsing rather than semantic comprehension.

Item	Security configuration (detailed)	Reference	Status	Suppport
S30	Encode SecuredMessage	5.2	S1 OR S7: M	□Yes □No
S31	Decode SecuredMessage	5.2	M	□Yes □No
S32	Encode SignedMessage	5.3	S1:M	□Yes □No
S33	Decode SignedMessage	5.3	M	□Yes □No
S34	Encode ToBeSignedMessage	5.3	S1:M	□Yes □No
S34.1	with fragment		O	□Yes □No
	with use_generation_time		S1:M	□Yes □No
	with expires		S1:M	□Yes □No
	with use_location		S1:M	□Yes □No
S35	Decode ToBeSignedMessage	5.3	M	□Yes □No

Item	Security configuration (detailed)	Reference	Status	Suppport
S35.1	with fragment		M	□Yes □No
S35.2	with use_generation_time		M	□Yes □No
S35.3	with expires		M	□Yes □No
S35.4	with use_location		M	□Yes □No
S36	Encode ToBeSignedWSM	5.4	S3:M	□Yes □No
S36.1	with fragment		O	□Yes □No
S36.2	with use_generation_time		S3:M	□Yes □No
S36.3	with expires		S3:M	□Yes □No
S36.4	with use_location		S3:M	□Yes □No
S37	Decode ToBeSignedWSM	5.4	M	□Yes □No
S37.1	with fragment		M	□Yes □No
S37.2	with use_generation_time		M	□Yes □No
S37.3	with expires		M	□Yes □No
S37.4	with use_location		M	□Yes □No
S38	Encode EncryptedMessage	5.13	O	□Yes □No
S38.1	with app_data content		S39:M	□Yes □No
S38.2	with signed content		S39:M	□Yes □No
S39	Decode EncryptedMessage	5.13	O	□Yes □No
S39.1	with app_data content		S39:M	□Yes □No
S39.2	with signed content		S39:M	□Yes □No
S40	Encode RecipientInfo	5.13	S39:M	□Yes □No
S41	Decode RecipientInfo	5.13	S39:M	□Yes □No
S42	Encode ECIESNISTp256EncryptedKey	5.14	S39:M	□Yes □No
S43	Decode ECIESNISTp256EncryptedKey	5.14	S39:M	□Yes □No

Item	Security configuration (detailed)	Reference	Status	Suppport
S44	Encode AESCCMCiphertext	5.14	S39:M	□Yes □No
S45	Decode AESCCMCiphertext	5.14	S39:M	□Yes □No
S46	Calculate CertID8	5.5	M	□Yes □No
S47	Calculate CertID10	5.5	M	□Yes □No
S48	Encode ApplicationID	5.8	S1, S3, S15, S17, S19: M	□Yes □No
S48.1	with fully_specified		S48:M	□Yes □No
S48.2	with match_any_acm		S48:M	□Yes □No
S48.3	with from_issuer		S48:M	□Yes □No
S49	Decode ApplicationID	5.8	M	□Yes □No
S49.1	with fully_specified		M	□Yes □No
S49.2	with match_any_acm		M	□Yes □No
S49.3	with from_issuer		M	□Yes □No
S50	Encode Time64	5.9	S1 OR S3:M	□Yes □No
S51	Decode Time64	5.9	M	□Yes □No
S52	Encode Time32	5.9	S17:M	□Yes □No
S53	DecodeTime32	5.9	M	□Yes □No
S54	Encode SignerInfo	5.10	S1, S3, S15, S17, S19: M	□Yes □No
S54.1	with certificate		S54:M	□Yes □No
S54.2	with certificate_digest		S54:M	□Yes □No
S54.3	with certificate_chain		S54:M	□Yes □No
S54.4	with self		S54:M	□Yes □No
S55	Decode SignerInfo	5.10	M	□Yes □No

Item	Security configuration (detailed)	Reference	Status	Suppport
S55.1	with certificate		M	□Yes □No
S55.2	with certificate_digest		M	□Yes □No
S55.3	with certificate_chain		M	□Yes □No
S55.4	with self		M	□Yes □No
S56	Maximum certificate chain length supported	5.10, 7.3.3.4	<5:X 5:M >5:O	
S57	Encode WAVECertificate	5.15	S15:M	□Yes □No
S58	Decode WAVECertificate	5.15	M	□Yes □No
S59	Encode ToBeSignedWAVECertificate	5.15	S15:M	□Yes □No
S59.1	with root_ca		S59:O:1	□Yes □No
S59.2	with ca		S59:O:1	□Yes □No
S59.3	with wsa_signer		S59:O:1	□Yes □No
S59.4	with csr_signer		S59:O:1	□Yes □No
S59.5	with rsu		S59:O:1	□Yes □No
S59.6	with psobu		S59:O:1	□Yes □No
S59.7	with obu_identified		S59:O:1	□Yes □No
S60	Decode ToBeSignedWAVECertificate	5.15	M	□Yes □No
S60.1	with root_ca		M	□Yes □No
S60.2	with ca		M	□Yes □No
S60.3	with wsa_signer		M	□Yes □No
S60.4	with csr_signer		S57:M	□Yes □No
S60.5	with rsu		M	□Yes □No
S60.6	with psobu		M	□Yes □No
S60.7	with obu_identified		M	□Yes □No
S61	Encode CertSpecificData	5.15	S59 OR	□Yes □No

Item	Security configuration (detailed)	Reference	Status	Suppport
			S71:M	
S62	Decode CertSpecificData	5.15	M	□Yes □No
S63	Encode CRLSeries	5.15	S17:M	□Yes □No
S64	Decode CRLSeries	5.15	M	□Yes □No
S65	Encode WAVECRL	5.16	S17:M	□Yes □No
S66	Decode WAVECRL	5.16	M	□Yes □No
S67	Encode ToBeSignedCRL	5.16	S17:M	□Yes □No
S67.1	with id_only		S67:M	□Yes □No
S67.2	with id_and_expiry		S67:M	□Yes □No
S68	Decode ToBeSignedCRL	5.16	M	□Yes □No
S68.1	with id_only		M	□Yes □No
S68.2	with id_and_expiry		M	□Yes □No
S69	Encode IDAndDate	5.16	S17:M	□Yes □No
S70	Decode IDAndDate	5.16	M	□Yes □No
S71	Encode WAVECertificateRequest	5.17	S13:M	□Yes □No
S72	Decode WAVECertificateRequest	5.17	S14:M	□Yes □No
S73	Encode ToBeSignedCSR	5.17	S71:M	□Yes □No
S73.1	with specified_in_request		S71:M	□Yes □No
S73.2	with specified_by_ca		S71:M	□Yes □No
S74	Decode ToBeSignedCSR	5.17	S57:M	□Yes □No
S74.1	with specified_in_request		S74:M	□Yes □No
S74.2	with specified_by_ca		S74:M	□Yes □No
S75	Encode WAVECertificateResponse	5.17	S57:M	□Yes □No
S76	Decode WAVECertificateResponse	5.17	S71:O	□Yes □No
S77	Encode PublicKey	5.5	S13 OR	□Yes □No

Item	Security configuration (detailed)	Reference	Status	Suppport
			S15 :M	
S77.1	with ecdsa_nistp224_with_sha224		S13:O:1	□Yes □No
S77.2	with ecdsa_nistp256_with_sha256		S13:O:1 S15:M	□Yes □No
S77.3	with ecies_nistp256		S39:M	□Yes □No
S78	Decode PublicKey	5.5	M	□Yes □No
S78.1	with ecdsa_nistp224_with_sha224		M	□Yes □No
S78.2	with ecdsa_nistp256_with_sha256		M	□Yes □No
S78.3	with ecies_nistp256		S39:M	□Yes □No
S79	Encode ECPublicKey	5.6	S77:M	□Yes □No
S79.1	with ecdsa_nistp224_with_sha224		S77.1:M	□Yes □No
S79.2	with ecdsa_nistp256_with_sha256		S77.2:M	□Yes □No
S79.3	with ecies_nistp256		S77.3:M	
S80	Decode ECPublicKey	5.6	M	□Yes □No
S80.1	with ecdsa_nistp224_with_sha224		M	□Yes □No
S80.2	with ecdsa_nistp256_with_sha256		M	□Yes □No
S80.3	with ecies_nistp256			
S81	Encode Signature	5.11	S1, S3, S15, S17, S19: M	□Yes □No
S81.1	with ecdsa_nistp224_with_sha224		S77.1:M	□Yes □No
S81.2	with ecdsa_nistp256_with_sha256		S77.2:M	□Yes □No
S82	Decode Signature	5.11	M	□Yes □No
S82.1	with ecdsa_nistp224_with_sha224		M	□Yes □No
S82.2	with ecdsa_nistp256_with_sha256		M	□Yes □No
S83	Encode ECDSASignature	5.12	S83:M	□Yes □No
S83.1	with ecdsa_nistp224_with_sha224		S77.1:M	□Yes □No

Item	Security configuration (detailed)	Reference	Status	Suppport
S83.2	with ecdsa_nistp256_with_sha256		S77.2::M	□Yes □No
S84	Decode ECDSASignature	5.12	M	□Yes □No
S84.1	with ecdsa_nistp224_with_sha224		M	□Yes □No
S84.2	with ecdsa_nistp256_with_sha256		M	□Yes □No
S85	Sign with ECDSA and SHA-224 over NIST curve p224	5.3, 7.3.2	S77.1:M	□Yes □No
S86	Sign with ECDSA and SHA-256 over NIST curve p256	5.3, 7.3.2	S77.2:M	□Yes □No
S87	Cryptographically secure random number generator	7.3.2	M	□Yes □No
S88	Verify with ECDSA and SHA-224 over NIST curve p224	5.3	M	□Yes □No
S89	Verify with ECDSA and SHA-256 over NIST curve p256	5.3	M	□Yes □No
S90	Encrypt an AES key with ECIES over NIST curve p256	5.14, 7.4.2.2	S39:M	□Yes □No
S91	Decrypt an AES key with ECIES over NIST curve p256	5.14, 7.4.3	S39:M	□Yes □No
S92	Encrypt data with AES-CCM	5.14, 7.4.2.3	S39:M	□Yes □No
S93	Decrypt data with AES-CCM	5.14, 7.4.3.2	S39:M	□Yes □No
S94	Encode GeographicRegion	5.18.1	O	□Yes □No
S94.1	with from_issuer		S94:O:2	□Yes □No
S94.2	with circle		S94:O:2	□Yes □No
S94.3	with rectangle		S94:O:2	□Yes □No
S94.4	with polygon		S94:O:2	□Yes □No
S94.5	with none		S94:O:2	□Yes □No
S95	Decode GeographicRegion	5.18.1	M	□Yes □No
S95.1	with from_issuer		M	□Yes □No

Item	Security configuration (detailed)	Reference	Status	Suppport
S95.2	with circle		M	□Yes □No
S95.3	with rectangle		M	□Yes □No
S95.4	with polygon		M	□Yes □No
S95.5	with none		M	□Yes □No
S96	Encode CircularRegion	5.18.3	S94.2:M	□Yes □No
S97	Decode CircularRegion	5.18.3	M	□Yes □No
S98	Encode RectangularRegion	5.18.4	S94.3:M	□Yes □No
S99	Decode RectangularRegion	5.18.4	M	□Yes □No
S100	Maximum number of RectangularRegions supported in the rectangular_region field of a received GeographicRegion	5.18.1	<6:X 6:M >6:O	
S101	Encode PolygonalRegion	5.18.5	S94.4:M	□Yes □No
S102	Decode PolygonalRegion	5.18.5	M	□Yes □No
S103	Maximum number of vertices supported in a received PolygonalRegion	5.18.5	<12:X 12:M >12:O	
S104	Encode 2DLocation	5.19	S94:M	□Yes □No
S105	Decode 2DLocation	5.19	M	□Yes □No
S106	Encode 3DLocationAndConfidence	5.19	M	□Yes □No
S107	Decode 3DLocationAndConfidence	5.19	M	□Yes □No
S108	Maximum supported confidence when sending 3DLocationAndConfidence	5.19		□500m □200m □100m □50m □20m □10m □5m □2m □1m □50cm □20cm □10cm □5cm □2cm □1cm

Item	Security configuration (detailed)	Reference	Status	Suppport
S109	Encode CAScope	5.20.1	:M	□Yes □No
S110	Decode CAScope	5.20.1	M	□Yes □No
S111	Encode AppIDAndPriority	5.20.1	S61:M	□Yes □No
S111.1	with fully_specified		S111:M	□Yes □No
S111.2	with match_any_acm		S111:M	□Yes □No
S111.3	with from_issuer		S111:M	□Yes □No
S112	Decode AppIDAndPriority	5.20.1	M	□Yes □No
S112.1	with fully_specified		M	□Yes □No
S112.2	with match_any_acm		M	□Yes □No
S112.3	with from_issuer		M	□Yes □No
S113	Maximum number of ApplicationIDs accepted in the applications field of a received CAScope	5.20.1	<8:X 8:M >8:O	
S114	Maximum number of AppIDAndPrioritys accepted in the apps_and_priorities field of a received CAScope	5.20.1	<8:X 8:M >8:O	
S115	Encode WSASignerScope	5.20.2	S61:M	□Yes □No
S116	Decode WSASignerScope	5.20.2	M	□Yes □No
S117	Maximum number of AppIDAndPrioritys accepted in the apps_and_priorities field of a received WSASignerScope	5.20.2	<8:X 8:M >8:O	
S118	Encode IdentifiedScope	5.20.3	S61:M	□Yes □No
S119	Maximum number of ApplicationIDs accepted in the applications field of an IdentifiedScope	5.20.3	<8:X 8:M >8:O	
S120	Encode CSRSignerScope	5.20.4	M	□Yes □No
S121	Maximum number of ApplicationIDs accepted in the applications field of a CSRSignerScope	5.20.4	<8:X 8:M >8:O	
S122	Encode OBUIdentifiedScope	5.20.5	M	□Yes □No

Item	Security configuration (detailed)	Reference	Status	Suppport
S123	Maximum number of ApplicationIDs accepted in the applications field of a OBUIdentifiedScope	5.20.5	<8:X 8:M >8:O	
S124	Maximum number of certificates in root certificate store	7.2.2	<32:X 32:M >32:O	
S125	CA certificate cache: One or both of the following may be specified	7.2.3		
S125.1	Maximum number of certificates in CA certificate cache			
S125.2	Lifetime of a certificate in CA certificate cache			
S126	Is CA certificate cache per-application or shared among applications?	7.2.3		□Per-app □Shared
S127	Message signing certificate cache: One or both of the following may be specified	7.2.3		
S127.1	Maximum number of certificates in message signing certificate cache			
S127.2	Lifetime of a certificate in message signing certificate cache			
S128	Is message signing certificate cache per-application or shared among applications?	7.2.3		□Per-app □Shared
S129	Default tolerance *r* for time window to prevent replay attacks. Measured in seconds.	7.2.5, 7.3.3.2	>=30:M	
S130	Is potential encrypted recipients certificate store maintained?	7.2.6	O	□Yes □No
S131	Maximum number of certificates in revoked certificate store	7.2.7		
S132	Time that fragments of a message are maintained for before being discarded if the message is incomplete	7.2.8		
S133	Correctly generate signed message	7.3.2	S1:M	□Yes □No
S133.1	with and without use_generation_time		S1:M	□Yes □No
S133.2	with and without use_location		S1:M	□Yes □No
S133.3	with and without expires		S1:M	□Yes □No

Item	Security configuration (detailed)	Reference	Status	Suppport
S134	Correctly perform validity checks on signed message	7.3.3.1, 7.3.3.2, 7.3.3.3, 7.3.3.4, 7.3.3.4.1, 7.3.3.4.2, 7.3.3.5, 7.3.3.6, 7.3.3.7	M	□Yes □No
S134.1	with and without use_generation_time		M	□Yes □No
S134.2	with and without use_location		M	□Yes □No
S134.3	with and without expires		M	□Yes □No
S134.4	with signer.type = certificate_digest		M	□Yes □No
S134.5	with signer.type = certificate		M	□Yes □No
S134.6	with signer.type = certificate_chain		M	□Yes □No
S134.7	support from_issuer value in applications and region fields.		M	□Yes □No
S135	Correctly generate encrypted message	7.4.2, 7.4.2.2, 7.4.2.3,	S5:M	□Yes □No
S136	Decrypt received encrypted message	7.4.3	S6:M	□Yes □No
S137	Generate SecuredWSA	8.1.2, 8.1.2.2, 8.1.2.3	XXX:O	□Yes □No
S138	SecuredWSA time tolerance *t* in seconds	8.1.3.1	*t*<5:X 5<=*t*<=30: M 30<*t*:X	
S139	Receive secured WSA	7.3.3, 8.1.3.2 8.4.3 8.4.48.5.2 8.5.38.5.4	M	□Yes □No
S140	Generate SignedWSM	8.2.2	O	□Yes □No
S141	Receive SignedWSM	8.2.3	M	□Yes □No
S142	Process received CRL	8.3.3.1, 8.3.3.3,	M	□Yes □No

Item	Security configuration (detailed)	Reference	Status	Suppport
S143	Support root certificate update	8.3.4	M	□Yes □No
S144	Maximum number of fragments supported for fragmented message	8.5	<8:X 8:M >8:O	

Annex B

(normative)

Summary of message formats

This annex collects the message formats of the main body of this document in one place for ease of reference. The types are sorted alphabetically by name.

```
struct {
            sint32        latitude;
            sint32        longitude;
        }   2DLocation;

struct {
            sint32        latitude;
            sint32        longitude;
            opaque        elevation_and_confidence[3];
        }   3DLocationAndConfidence;

struct {
            opaque         nonce[12];
            opaque         ccm_ciphertext<2^16-1>;
        }   AESCCMCiphertext;

enum    {fully_specified (0), match_any_acm(1), from_issuer(2),
            (2^8-1)
        }   AIDType;

struct {
            AIDType   type;
            select    (type) {
            case fully_specified:
                uint8   acid;
                opaque  acm<2^8-1>;
                uint8   maxPriority;
            case match_any_acm:
                uint8   acid;
                uint8   maxPriority;
            case from_issuer: ;
            }
        }   AppIDAndPriority;

struct {
            AIDType   type;
            select    (type) {
            case fully_specified:
                uint8   acid;
                opaque  acm<2^8-1>;
            case match_any_acm:
                uint8   acid;
            case from_issuer: ;
            }
        }   ApplicationID;

struct {
            SubjectTypesFlags   tf<2^8-1>;
            if_any_set (tf, ca, csr_signer, rsu, psobu,
                obu_identified) {
                ApplicationID       applications<2^16-1>;
            }
```

```
            if_any_set (tf, wsa_ca, wsa_signer) {
                AppIDAndPriority    apps_and_priorities<2^16-1>;
            }
            GeographicRegion    region;
        }   CAScope;

    struct {
            extern SubjectType subject_type;
            select(subject_type){
            case root_ca:
            case ca:
            case csr_signer:
                CAScope             scope;
            case crl_signer:
                CRLSeries           responsible_series<2^16-1>;
            case wsa_signer:
            case rsu:
            case psobu:
                IdentifiedScope     scope;
            case obu_identified:
                OBUIdentifiedScope  scope;
            }
        }   CertSpecificData;

    opaque CertID8[8];

    opaque CertID10[10];

    struct {
            2DLocation center;
            uint16                  radius;
        }   CircularRegion;

    uint32 CRLSeries;

    enum    { id_only(0), id_and_expiry(1), (2^8-1) } CRLType;

    struct {
            opaque              subject_name<2^8-1>;
            SubjectType         types<2^8-1>;
            ApplicationID       applications<2^16-1>
            GeographicRegion    region;
        }   CSRSigner;

    struct {
            extern uint8 curve_order_octets;
            opaque[curve_order_octets] r;
            opaque[curve_order_octets] s;
        }   ECDSASignature;

    struct {
            extern uint32     symm_key_len;
            opaque            v[33];
            opaque            c[symm_key_len];
            opaque            t[16];
        }   ECIESNISTp256EncryptedKey;

    struct {
            extern uint8   point_size
            opaque         point[point_size];
        }   ECPublicKey;

    struct {
            opaque     update_digest[20];
```

```
            FECScheme  scheme;
            select (scheme) {
            case (xor_symbols) :
                uint16   k;
                uint16   index;
            }
            opaque symbol<2^16-1>;
        }   EncodedSymbol;

enum    { app_data(0), signed(1), (2^8-1)} EncryptedContentType;

struct {
            EncryptedContentType   content_type;
            SymmAlgorithm          symm_algorithm;
            RecipientInfo          recipients<2^16-1>;
            select(symm_algorithm) {
            case aes_128_ccm:
                AESCCMCiphertext   ciphertext;
            }
        }   EncryptedMessage;

enum    { xor_symbols(0), (2^8-1) } FECScheme;

struct {
            uint8   acid;
            opaque  acm<2^8-1>;
        }   FullySpecifiedAppID;

struct {
            RegionType              region_type;
            select(region_type){
            case from_issuer:
            case circle:
                CircularRegion      circular_region;
            case rectangle:
                RectangularRegion   rectangular_region<2^16-1>;
            case polygon:
                PolygonalRegion     polygonal_region;
            case none: ;
            }
        }   GeographicRegion;

struct {
            CertID10    id;
            Time32      expiry;
        }   IDAndDate;

struct {
            opaque            subject_name<2^8-1>;
            ApplicationID     applications<2^16-1>;
            GeographicRegion  region;
        }   IdentifiedScope;

flags   {fragment(0), use_generation_time(1), expires(2),
            use_location(3)} MessageFlags;

enum    { unsecured (0), signed(1), encrypted (2), (2^8-1) }
            MessageType;

struct {
            opaque cert_specific_data<2^16-1>;
            ApplicationID  applications<2^16-1>;
        }   OBUIdentifiedScope;
```

```
enum   {   ecdsa_nistp224_with_sha224 (1),
           ecdsa_nistp256_with_sha_256 (2), ecies_nistp256 (1),
           reserved (240..255), (2^8-1)
       }   PKAlgorithm;

2DLocation PolygonalRegion<2^16-1>;

struct {
           PKAlgorithm             algorithm;
           select(algorithm){
           case ecdsa_nistp224_with_sha224:
           case ecdsa_nistp256_with_sha256:
               ECPublicKey         public_key;
           case ecies:
               SymmAlgorithm       supported_symm_algs<2^8-1>;
               ECPublicKey         public_key;
           }
       }   PublicKey;

struct {
           CertID8                          cert_id;
           extern PKAlgorithm               pk_encryption;
           select (pk_encryption)  {
           case ecies_nistp256:
               ECIESNISTp256EncryptedKey  enc_key;
           }
       } RecipientInfo;

struct {
           2DLocation    upper_left;
           2DLocation    lower_right;
       }   RectangularRegion;

enum   { from_issuer(0), circle(1), rectangle(2), polygon(3),
           none (4), (2^8-1)} RegionType;

enum   { specified_in_request(0), specified_by_ca(1), (2^8-1) }
           RequestScopeType;

struct {
           uint8        protocol_version; // 1 for this version
           MessageType  type;
           select (type) {
           case unsecured :
               opaque           message<2^32-1>;
           case signed :
               SignedMessage signed_message;
           case encrypted :
               EncryptedMessage encrypted_message;
       }   SecuredMessage;

struct {
           extern PKAlgorithm algorithm;
           select(algorithm) {
           case ecdsa_nistp224_with_sha224:
           case ecdsa_nistp256_with_sha256:
               ECDSASignature ecdsa_signature;
           }
       }   Signature;

struct {
           SignerInfo           signer;
           ToBeSignedMessage    unsigned_message;
           Signature            signature;
```

```
        }   SignedMessage;

    struct {
                SignerInfo          signer;
                ToBeSignedWSM       unsigned_wsm;
                Signature           signature;
        }   SignedWSM;

    enum    {   certificate(0), certificate_digest(1),
                certificate_chain (2), self (4), (2^8-1)}
                SignerIdentifierType;

    struct {
                SignerIdentifierType  type;
                select (type) {
                case certificate:
                    WAVECertificate   certificate;
                case certificate_digest:
                    CertID8           digest;
                case certificate_chain:
                    WAVECertificate   certificates<2^16-1>;
                case self: ;
                }
        }   SignerInfo;

    enum    {  wsa_ca (0), ca(1), wsa_signer(2), rsu(3), psobu(4),
               obu_identified(5), crl_signer(6), csr_signer(8),
               root_ca(9), (2^8-1)
        }  SubjectType;

    flags   {  wsa_ca (0), ca(1), wsa_signer(2), rsu(3), psobu(4),
               obu_identified(5), crl_signer(6), csr_signer(8)
        }  SubjectTypeFlags;


    enum    {   aes_128_ccm (0), reserved (240..255), (2^8-1) }
                SymmAlgorithm;

    uint32 Time32;

    uint64 Time64;

    struct {
                CRLType     type;
                CRLSeries   crl_series;
                CertID8     ca_id;
                uint32      crl_serial;
                Time32      start_period;
                Time32      issue_date;
                Time32      next_crl;
                select (type) {
                case (id_only):
                    CertID10    entries<2^64-1>;
                case (id_and_expiry):
                    IDAndDate   entries<2^64-1>;
                } entries;
        }   ToBeSignedCRL;

    struct {
                uint8            csr_version;
                SubjectType      subject_type;
                RequestScopeType request_type;
                select (request_type) {
                case specified_in_request:
```

```
                CertSpecificData type_specific_data;
            case specified_by_ca: ;
            }
            PublicKey        public_key;
        }   ToBeSignedCSR;

    struct {
            FullySpecifiedAppID  application;
            MessageFlags         mf;
            opaque               application_data<2^16-1>;
            if_flag_set (mf, use_generation_time) {
                Time64            generation_time;
            }
            if_flag_set (mf, expires) {
                Time64           expiry_time;
            }
            if_flag_set (mf, use_location) {
                3DLocationAndConfidence       transmission_location;
            }
        }   ToBeSignedMessage;

    struct {
            uint8                    certificate_version;
            SubjectType              subject_type;
            select (subject_type) {
            case ca:
            case crl_signer:
            case wsa_signer:
            case csr_signer:
            case rsu:
            case psobu:
            case obu_identified:
                CertID8              signer_id;
            case root_ca:  ;
            } signer;
            CertSpecificData         scope;
            Time32                   expiration;
            CRLSeries                crl_series;
            PublicKey                public_key<2^8-1>;
        }   ToBeSignedWAVECertificate.

    struct {
            MessageFlags         mf;
            opaqueExtLength      application_data<2^16-1>;
            if_flag_set (mf, use_generation_time) {
                Time64           generation_time;
            }
            if_flag_set (mf, expires) {
                Time64           expiry_time;
            }
            if_flag_set (mf, use_location) {
                3DLocationAndConfidence       transmission_location;
            }
        }   ToBeSignedWSM;

    struct {
            ToBeSignedWAVECertificate unsigned_certificate;
            Signature                 signature;
        }   WAVECertificate;

    struct {
            SignerInfo info;
            ToBeSignedCSR unsigned_csr;
            Signature signature;
```

```
        }   WAVECertificateRequest;

    struct {
            Certificate certificate_chain<2^32-1>;
            OrdinaryCRL crl_path<2^32-1>;
        }   WAVECertificateResponse;

    struct {
            uint8         version;
            SignerInfo    signer;
            ToBeSignedCRL unsigned_crl;
            Signature     signature;
        }   WAVECRL;
```

Annex C

(informative)

Examples of message structures

C.1 General

This annex gives examples of the structures of some encoded messages within the system as an aid to implementers.

C.2 CA certificate

Figure C.1 is an example of the structure of a certificate for a CA that is authorized to issue certificates for a single application running on an identified OBU. The application is identified in the CA certificate. The ACM for the application is assumed to be 10 bytes long. The CA is entitled to issue certificates anywhere in the world and so uses the `GeographicRegion` identifier `none`.

Length	Field			
1	certificate_version =			
1	unsigned_certificate	subject_type = ca		
8		signer_id		
2		scope	tf = obu_identified (5) (encoding = 01 20)	
2			applications	length of applications field
1				type
1				ACID
1				ACM length
10				ACM
1			region	region_type = none
4		expiration		
4		crl_series		
1		public_key	length of public key field	
1			algorithm = ecdsa_nistp256_with_sha256	
33			public_key	point (length derived from algorithm)
32	signature	ecdsa_signature	r	
32			s	

Total length: 135 octets

Figure C.1—CA signing certificate

C.3 OBU signing certificate

Figure C.2 is an example of the structure of a certificate for an OBU that is authorized to run a single application. The application permissions are inherited from the CA, so the `application.type` field in the OBU certificate is set equal to `from_issuer`. The OBU `subject_name` data is assumed to be 8 bytes long.

Length	Field			
1	certificate_version = 1			
1	unsigned_certificate	subject_type = obu_identified		
8		signer_id		
1		scope	subject_name length	
8			subject_name	
2			applications	length of applications field
1				type = from_issuer
4		expiration		
4		crl_series		
1		public_key	length of public key field	
1			algorithm = ecdsa_nistp224_with_sha224	
29			public_key	point (length derived from algorithm)
32	signature	ecdsa_signature	r	
32			s	

Total length: 125 octets

Figure C.2—OBU signing certificate

C.4 RSU signing and encryption certificate

Figure C.3 is an example of the structure of a certificate for an RSU that is authorized to run a single application and can both send signed messages and receive encrypted messages. The application permissions are inherited from the CA, so the `application.type` field in the RSU certificate is set equal to `from_issuer`. The RSU `subject_name` data is assumed to be 10 bytes long. The `region` scope is rectangular region consisting of three rectangles.

Length	Field					
1	certificate_ version = 1					
1	unsigned_ certificate	subject_type = rsu				
8		signer_id				
1		scope	subject_name length			
8			subject_name			
2			applications	length of field		
1				type = from_issuer		
1			region	region_type = rectangle		
2				rectangular_ region	length of field	
4				[1]	upper_ left	latitude
4					upper_ left	longitude
4					lower_ right	longitude
4					lower_ right	longitude
4				[2]	upper_ left	latitude
4					upper_ left	longitude
4					lower_ right	longitude
4					lower_ right	longitude
4				[3]	upper_ left	latitude
4					upper_ left	longitude
4					lower_ right	longitude
4					lower_ right	longitude
4		expiration				
4		crl_series				
1		public_key	length of field			
1			algorithm = ecies_nistp256			
33			public_key	point (length derived from algorithm)		
32	signature	ecdsa_ signature	r			
32			s			

Total length: 180 octets

Figure C.3—RSU signing and encryption certificate

C.5 Signed Message

Figure C.4 is an example of the structure of a signed message that might be created by the above OBU. The signed message is a `SignedMessage` type contained in a `SecuredMessage` type. The payload of the `SignedMessage` is taken to be 32 octets. The certificate is the certificate given in C.3; details of the certificate fields are therefore omitted from this subclause. The message flags `use_generation_time` and `use_location` are set; the flag `expiry_time` is not.

Length	Field			
1	protocol_version = 1			
1	type = signed			
1	signed_message	signer	type = certificate	
125			certificate (see C.3 for details of fields)	
1		unsigned_message	application	acid
1				length of ACM
10				ACM
2			mf (encoded as 01 0a)	
2			length of application_data	
32			application_data	
8			generation_time	
4			transmission_location	latitude
4				longitude
3				elevation_ and_confidence
28		signature	ecdsa_signature	r
28				s

Total length: 251 octets

Figure C.4—Signed message

C.6 SignedWSM

Figure C.5 is an example of the structure of a SignedWSM that might be created by the above OBU. As before, the payload of the SignedMessage is taken to be 32 octets, and the length of the ACM is taken to be 10 octets. The certificate is the certificate given in C.3; details of the certificate fields are omitted.

Length	Field			
1	WSM version			
1	Security Type = signed(1)			
1	Channel Number			
1	Data Rate			
1	TxPwr_Level			
1	Application Class Identification			
1	ACM Field Length			
10	ACM			
2	WSM Length			
1	WSM Data	signer	type = certificate	
125			certificate (see C.3 for details of fields)	
2		unsigned_wsm	mf (encoded as 01 0a)	
32			application_ data	
8			transmission_ time	
4			transmission_ location	latitude
4				longitude
3				elevation_ and_confidence
28		signature	ecdsa_ signature	r
28				s

Total length: 254 octets

Figure C.5—Signed WSM

C.7 Encrypted Message

Figure C.6 is an example of the structure of an encrypted message that might be sent to the above RSU. The encrypted message is an `EncryptedMessage` type contained in a `SecuredMessage` type. The plaintext length is taken to be 32 octets. The length of the ciphertext is therefore 48 octets (the length of the plaintext plus the 16-octet authentication tag length). The encrypted contents are application data, rather than signed data. The message is sent to two recipients.

Length	Field		
1	protocol_version		
1	type = encrypted		
1	encrypted_ message	content_type = app_data	
2		recipients	length of recipients field
8		[1]	cert_id
33			v
16			c
16			t
8		[2]	cert_id
33			v
16			c
16			t
12		ciphertext	nonce
2			length of ccm_ciphertext
48			ccm_ciphertext

Total length: 213 octets

Figure C.6—Encrypted Message

Annex D

(informative)

General description

D.1 Introduction

The Wireless Access in Vehicular Environments (WAVE) system is designed to enable vehicle-to-vehicle and vehicle-to-roadside communications. These communications provide transportation services such as, alerting drivers to potential hazards and notifying them of services of interest even at high speed or in high traffic density. This document addresses the security requirements of the system. This annex provides informative material to help the understanding of the normative material in the main body of this standard. Subclause D.2 provides an overview of the WAVE system as a whole, introducing the various roles played by entities in the system and describing the WAVE radio stack. Subclause D.3 provides an overview of the security challenges that face this system, and the cryptographic techniques that can be used to address those challenges.

D.2 WAVE system

D.2.1 Entities in the system

The WAVE system involves many different types of entities. In day-to-day operations the following are the most active:

— **Providers**—Units that provide services on one or more service channels.
— **Users**—Units that consume services offered by providers.
— **Roadside units** (RSUs)—WAVE devices that operate only when stationary and support information exchange with OBUs. RSUs are typically embedded in infrastructure elements such as road signs and traffic signals; they may be moved from one site to another but do not operate when in motion. Providers will usually be RSUs. RSUs are licensed by site and may provide services on one or more service channels.
— **On-Board Units** (OBUs)—WAVE devices that can operate when in motion and support information exchange with RSUs and other OBUs. OBUs are typically embedded in vehicles, though they may also be portable (hand-held). OBUs operate in private vehicles, and a major goal of the security protocols in this document is to enable them to operate without violating a driver's reasonable assumptions about personal privacy. OBUs will usually be users, but may be providers in certain cases.

An important subset of OBUs consists of

— **Public Safety On-Board Units** (PSOBUs)—Network nodes embedded in public safety vehicles. These nodes are permitted by the relevant authorities to operate particular public safety applications such as traffic signal prioritization.

In addition, the following entities support security services:

— **Certificate Authorities** (CAs)—Entities that are able to authorize other entities via the issuance and revocation of certificates.

D.2.2 WAVE radio stack

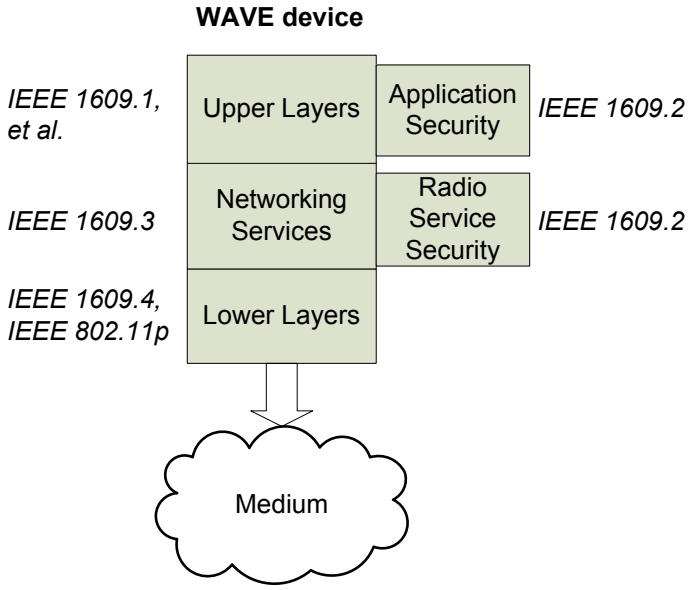

Figure D.1—WAVE radio stack

Figure D.1 illustrates the WAVE radio stack.

The system supports operations on a control channel (CCH) and multiple service channels (SCH). The PHY layer is specified in IEEE P802.11p [B2], which is in turn based on IEEE Std 802.11 [B6]. The MAC layer, including the management of queues for the CCH and SCH, is specified in IEEE P1609.4 [B4]. The LLC layer is specified in IEEE Std 802.2™ [B5].

Above the LLC layer, the network stack splits into two: the WSMP stack (IEEE P1609.3) and the UDP/IP stack (IETF RFC 768 and IETF RFC 2460 [B14]). IEEE P1609.3 describes the operation of the Wave Short Message Protocol (WSMP) stack, and also describes networking services for applications running over either of these stacks. WSMs are in general used for broadcast applications. The UDP/IP stack is in general used for transactional applications, that is applications that mainly carry out unicast data exchange. A Provider notifies Users of the services it offers using the WAVE Service Announcement (WSA), which details what the services are and what service channel(s) they are provided on. These services are managed by the WAVE Management Entity (WME), which can be considered as touching both the application layer (where it generates notifications) and the MAC layer (where it generates WSAs).

The security services of the current standard are used within the stack as follows:

— Applications running over UDP may use the secured message formats of this standard to protect application data.
— Applications running over WSMP may use the secured message formats or the secured WSM types of this standard to protect application data.
— The WME uses the secured WSIE type to protect WSAs, preventing attackers from offering fake services.

D.3 Communications security overview

D.3.1 Introduction

Data transmitted over any network is subject to a range of threats. For example, an attacker may intercept a message and read its contents, or alter the message and thereby change the recipient's behavior, or replay it at a later time or in a different location to masquerade as the original sender. The first of these attacks is mitigated by the security service known as *confidentiality* (*encrypting* a message for a specific recipient such that only that recipient can use it). The second and third attacks, based on tricking a recipient into accepting incorrect message contents, are mitigated by the security services known as *authenticity* (confirmation of origin of the message) and *integrity* (confirmation that the message has not been altered in transit). This standard specifies mechanisms to provide confidentiality, authenticity and integrity to WAVE communications.

In addition to the requirement to protect against these standard threats, the WAVE system adds an additional requirement, that of *anonymity* for end-users.

D.3.2 Cryptographic services

D.3.2.1 Introduction

The security services of confidentiality, authenticity and integrity are provided by cryptographic mechanisms. There are three main families of cryptographic mechanisms, *secret-key* or *symmetric* algorithms, *public-key* or *asymmetric* algorithms, and *hash functions*. This subclause introduces those mechanisms and discusses their use and related key management issues.

D.3.2.2 Symmetric algorithms

In a symmetric system, two entities (traditionally called Alice and Bob) who want to communicate, and they both know the same secret data, known as a *key*. To provide confidentiality, Alice uses the key to scramble or *encrypt* her message; Bob, who knows the same key, can unscramble or *decrypt* it, but an attacker who does not know the key cannot decrypt. To provide authenticity and integrity, Alice uses the same or a different key to generate a cryptographic checksum or Message Integrity Check (MIC) on the message; Bob, who knows the same authentication key, can check the MIC. The MIC will only pass the check if the message and MIC have not been altered in transit, and if Bob is using the correct key to perform the check. An attacker who does not know the key can alter the contents of the message, but cannot generate the correct MIC. The presence of a valid MIC on a message from Alice to Bob therefore provides a guarantee that the message did, in fact, originate from Alice and has not been altered in transit.

A message may be encrypted only, authenticated only, or both encrypted and authenticated. In the latter case, it may be first encrypted and then authenticated, or first authenticated and then encrypted. This standard uses the AES-CCM mechanism, which fits the authenticate-then-encrypt model.

NOTE—Since either Alice or Bob could have produced a valid MIC on a message between them, the MIC on its own is not enough to prove to a third party which one of Alice and Bob originated the message. However, if a message is correctly MICed, and Bob received it and did not originate it, he can be certain that it actually originated from Alice.

D.3.2.3 Asymmetric algorithms

In an asymmetric algorithm, keys come in pairs. The two keys in a *keypair*, known as the *public key* and the *private key*, are mathematically related so that it is extremely difficult to determine the private key given only the public key. This means that the public key can be distributed widely without jeopardizing the security of the private key.

To send an encrypted message to Bob, Alice obtains Bob's public encryption key and uses it to encrypt the message. Bob, who knows the corresponding private decryption key, can decrypt the message. Bob's private decryption key is known only to Bob, so no-one else who intercepts or overhears the message can decrypt it.

To send an authenticated, integrity-checked message to Bob, Alice uses her own private signing key to generate a cryptographic checksum on the message. A cryptographic checksum generated by a private key is known as a digital signature, or simply a signature (contrast with the cryptographic checksum generated by a symmetric key, which is known as a MIC). On receipt of the message, Bob obtains Alice's public verification key, the dual of her private signing key. The verification process takes as input the message, the signature, and the verification key. It will output "valid" only if the signature was generated by the corresponding private signing key and the message and signature were not altered in transit. In contrast to the symmetric MIC, a signature can only be produced by one party. It can therefore be used to prove to a third party that a message from Alice to Bob actually originated from Alice.

Digital signatures are particularly useful for securing communications with parties that have not previously been encountered, such as when broadcasting to a dynamically changing population. This makes them particularly suitable for use in the WAVE environment, where (for example) a provider on an RSU will offer services to the large number of vehicles that pass that RSU, some of which it will never have encountered before.

In the WAVE context, broadcast messages (e.g., WSIE broadcasts, vehicle safety messages such as extended brake lights, approaching public safety vehicle warnings) are signed. Broadcast messages will not in general be encrypted. Transactional messages may be protected with asymmetric or symmetric algorithms.

D.3.2.4 Hash functions

A cryptographically secure hash function maps an arbitrary-length input into a fixed-length output (the hash value) such that (a) it is computationally infeasible to find an input that maps to a specific hash value and (b) it is computationally infeasible to find two inputs that map to the same hash value. Hash functions can be used to generate identifiers for information, such that it is computationally infeasible to find alternative information that will give the same identifier. When calculating a digital signature, it is standard practice to first hash the message itself and then apply the digital signature algorithm to the resulting hash value. This standard makes use of the SHA-1 hash function, defined in FIPS 180-1, to generate identifiers for certificates and fragmented messages.

D.3.2.5 Digital certificates and authorization

The preceding description has assumed that Bob is certain that Alice's public key in fact belongs to Alice. If Alice and Bob only know each other online, it is difficult to be sure of this: an attacker could in theory be manipulating their communications. However, if Bob knows Alice offline, Alice can use offline means (such as physically transferring the key, or confirming it by a phone call) to give Bob confidence that he is actually using her correct public key.

Now, if Alice knows Carol offline but Bob doesn't, Alice can effectively introduce Carol to Bob by digitally signing Carol's public key with her, Alice's, key. Because Bob knows Alice, and knows that Alice doesn't frivolously sign keys, he can be sure that Carol's key in fact belongs to Carol. In this case the signed attestation that Carol's key belongs to Carol is known as a *digital certificate*, and Alice is said to be acting as a *certificate authority* or CA. Extending this concept, say Bob knows the certificate of a *root CA*. The root CA does not certify end entities, but only other CAs. If Carol presents both her certificate and her CA's certificate, and if her CA's certificate was issued by the root CA that Bob knows, then Bob can construct a *certificate chain* back to that root CA and so trust Carol's certificate.

In general, if Bob knows a single *root certificate*, then that certificate can be used to sign end-entity certificates or to sign *CA certificates*. A certificate that is used only to sign other certificates is known as a *CA certificate*. A certificate that is used to sign messages is referred to in this standard as a *message signing certificate*. If certificate A signed certificate B, then A is known as an *issuing certificate* and B is known as a *subordinate certificate*.

Certificates may be used to link information other than identity information to a public key. In this standard, most certificates link a public key with a set of permissions (to run certain applications, or take certain actions). To take advantage of the permissions, a party needs to demonstrate that they know the private key corresponding to the public key in the certificate before they may take actions corresponding to the permissions in the certificate.

For example:

- Carol sends Bob a message containing
  - Her certificate, which states that she can take action A,
  - A request to take action A, and
  - Her signature on the current time and the request.
- Bob verifies the signature on Carol's certificate. This shows that the owner of the private key can take action A. He then uses Carol's public key, from her certificate, to verify her signature on the current time and request for action. This proves that Carol is the owner of the private key and that she generated the message now, rather than at some point in the past.
- Bob allows Carol to take action A.

In this standard, the construction and use of signed messages is based on this model.

A widely-used certificate profile is found in IETF RFC 3315 [B16]. However, for bandwidth reasons, this standard defines its own certificate format.

D.3.2.6 Combining asymmetric and symmetric cryptography

Symmetric algorithms in general operate much faster than asymmetric algorithms. However, they lack some of the flexibility of asymmetric algorithms. There is no symmetric protocol like the usage model above, which allows Carol to send a single message that Bob will accept if they have not previously been in contact. (The best symmetric protocols require Carol to send a message to a third party, following which she can send a single message to Bob that he will accept).

The fact that asymmetric cryptography allows rapid session setup is of great value in WAVE. However, once a secure session is set up it is useful to take advantage of the superior performance of symmetric cryptography. A typical approach is for the parties to use asymmetric cryptography to agree only on a symmetric key, and then to use the symmetric key to protect data. This model is supported by the encrypted message formats presented in this document.

D.3.3 Anonymity

The term "anonymity" describes the design goal that, as far as possible, broadcasttransmissions from a vehicle operated by a private citizen should not leak information that can be used to identify that vehicle to unauthorized recipients. (Public safety vehicles, which are representing some state authority, do not generally have this requirement for anonymity).

A vehicle may use broadcast or transactional applications. In both cases, the use of these applications should not compromise anonymity.

In the case of broadcast applications, the messages should minimize data that uniquely identifies the vehicle or that would allow a recipient to link messages—that is, to determine that multiple messages from

dispersed locations and times have come from the same vehicle. More precisely, the chance that an attacker can link messages must should drop off rapidly with the distance and time between the transmission of the two messages. This requirement must be satisfied consistent with also requiring messages to be authenticated, in other words preventing an attacker with a radio unit from inserting messages into the system that did not actually originate with a vehicle. Mechanisms for providing anonymous authenticated broadcast messages are not given in this standard at this time.

In the case of transactional applications, the vehicle may choose to reveal its identity, or at least reveal linkable data, to a trusted respondent. This standard defines mechanisms that the vehicle can use to identify and authenticate itself to such a respondent. However, it the vehicle will will not wish to reveal this data identifying information to an eavesdropper. Therefore, all data exchanged by transactional applications needs to be encrypted. The encryption applied needs to be *semantically secure*, meaning that even if a series of plaintexts contain a static identifier this pattern will not be visible in the corresponding series of ciphertexts. This standard specifies semantically secure encryption mechanisms for use with transactional applications, allowing such applications to be used in an anonymous fashion. Alternative mechanisms may also be used so long as they also prevent an eavesdropper from recognizing static or identifying information.

Additionally, the headers in a transmitted packet might reveal information about the sender, for example by including a fixed source MAC address. A truly anonymous system must remove this anonymity-compromising information, or at least reduce its usefulness to an attacker. The current standard is focused on protecting message payloads and does not in its current version provide anonymizing techniques for the message headers.

Annex E

(informative)

Additional security considerations

E.1 OBUs

E.1.1 Keying material

Keys should not be shared between applications with different ACIDs.

Where practical, keys should be protected from exposure by embedding them in a tamper-resistant Hardware Security Module (HSM). This HSM should support both keypair generation and import of keypairs from an external trusted source.

All software/firmware upgrades for the HSM should be signed. On receipt of a software/firmware upgrade message, the HSM should check that the signature verifies, that it can form a chain of certificates back to a known root certificate, and that all certificates in that chain have permissions to sign software/firmware upgrades (or to grant such permissions). It should only install the upgrades if all these checks pass.

E.2 Root certificates

An OBU should have enough root certificates installed on it to enable it to verify all safety messages it is likely to receive. This will typically include:

— A root certificate for public safety and identified vehicle safety messages.
— A root certificate to allow firmware and root certificate update.
— Depending on the mechanism that is chosen to provided anonymity with authentication, OBUs may also be provisioned with a root certificate for anonymous vehicle safety messages.

E.3 Public safety vehicles

E.3.1 Certification and verification

A public safety vehicle is considered herein as one that is certified by the appropriate government agency, local or federal, to operate certain applications such as traffic signal prioritization and preemption. For the purposes of this document, public safety vehicles can therefore be considered to include buses and other public transit vehicles as well as emergency response. This standard does not mandate the certificate hierarchy that shall be used. Many configurations are possible. At one extreme a single national CA could certify all public safety vehicles. At another extreme, the federal department of transportation (DoT) could certify state DoTs, which in turn could certify different departments, which in turn could certify depots, which in turn could certify vehicles. The only restriction placed on certification hierarchies by this standard is that conformant implementations shall be able to construct a certificate chain of length up to 5.

This standard also does not dictate a mechanism to ensure that vehicles traveling in a country other than their country of origin can process domestic public safety messages. Mechanisms include the following, which may or may not be politically acceptable.

— All national DoTs are certified by a root CA. This root CA's key is included in all vehicles.

— All national DoTs have their own root certificate. All these root certificates are included in all vehicles.
— All national DoTs have their own root certificate, with the borders of the country encoded in its geographic scope field. If the vehicle detects that it does not have a root public safety certificate for its current location, it requests a root certificate update.

E.3.2 Public safety certificate lifetime

Because a compromised public safety vehicle might have the ability to cause considerably greater traffic disruption than a single compromised end-user vehicle, it is important to remove those vehicles from the system in a timely fashion. Distributing a CRL to all vehicles that might encounter a compromised PSOBU is likely to be impractical. PSOBU certificates should therefore have a limited lifetime, on the order of a day.

A PSOBU may demonstrate its authorization to obtain and refresh its certificate either by physical presence or by use of a CSR Signer certificate.

In the case of physical presence, the OBU issues a self-signed certificate request while in a location where a human operator can check that the vehicle requesting certification is physically present and manually authorize certificate issuance.

In the case of the CSR Signer certificate, the OBU shall first obtain a CSR signing certificate. Thereafter it may request a message signing certificate by periodically sending a certificate request signed by the CSR signing certificate. The fact that the OBU possesses a CSR signing certificate demonstrates that it is authorized to receive a message signing certificate with the appropriate scope. By requesting a new message signing certificate near the expiration time of its current message signing certificate, the OBU can guarantee that it will always have a fresh message signing certificate, thereby enabling the use of short-lived certificates rather than revocation to enable vehicles on the road to ignore compromised units. The CA that issues the message signing certificate shall manage CSR signing certificate revocation information, but distributing this revocation information to CAs is much simpler than distributing it to vehicles on the road.

An implementation should generate a new keypair every time it requests a public safety certificate.

E.4 RSUs

E.4.1 Keying material

When an RSU is acting as a provider, keys should not be shared between applications with different ACIDs.

E.4.2 Location

An RSU that is equipped with GPS should monitor its location and disable operation if it determines that it is outside its certified region.

E.5 Other considerations

E.5.1 Randomness

Signing with ECDSA involves selecting a random value. The ECDSA algorithm can leak information about the private key if this random value is biased towards a particular interval. The random number

generator used for ECDSA shall therefore be cryptographically secure, as shall the random number generator used to generate symmetric keys.

E.5.2 Cryptographic key sizes

ECDSA keys for end entities should be defined over fields of size at least 224 bits. ECDSA keys for CAs should be defined over fields of size at least 256 bits. ECIES keys should be defined over field of size at least 256 bits. The only symmetric algorithm supported in this document is AES with 128-bit keys.

Annex F

(informative)

Threat model

This annex outlines threats against different specific applications, to illustrate the design choices made in this standard. The application descriptions are given as examples only and are not intended to dictate how the applications will in practice be implemented.

F.1 Attacker taxonomy

To aid analysis, this annex considers four classes of attacker:

— **Class 1**: Has a radio transmitter/receiver but no keying material.
— **Class 2**: Has a valid OBU or RSU.
— **Class 3**: Has extracted the keying material from an OBU or RSU, allowing them to either pretend to be that OBU or RSU in multiple locations simultaneously, or force the revocation of that keying material.
— **Class 4**: Insider at an organization with security administrative functions (HSM manufacturer, CA, vehicle manufacturer, government, etc.).

Many messages in the system are vulnerable to being forged or replayed. The use of the `SignedMessage` type and processing rules in this standard prevents a class 1 attacker from forging messages, or a class 2 attacker from sending a large volume of fraudulent messages. The damage that a class 3 attacker can cause is limited if there is a mechanism to remove compromised certificates from the system, either by certificate expiry (short-lived certificates, as recommended for public safety vehicles) or by certificate revocation. Both of these mechanisms are supported by this standard. Finally, to limit the damage that a class 4 attacker can cause, it is recommended that organizations with security administrative functions have sensible and thorough codes of conduct and audit procedures. The details of these codes and procedures are beyond the scope of this standard.

F.2 BasicSafetyMessage

The BasicSafetyMessage in SAE Draft J2735 [B22] is a vehicle application message, transmitted multiple times per second, that contains location, Velocity, and other telematic information. This will enable other vehicles on the road to build up a picture of local traffic conditions.

The primary threats against these messages are as follows:

— An attacker might forge them, causing traffic disruption.
— An attacker might use valid heartbeat messages to track a private vehicle.

To prevent the first attack, BasicSafetyMessages may be signed on transmission and verified on reception using the `SignedMessage` format of this standard. However, this standard does not provide a mechanism for anonymous signing and so does not address the second threat.

F.3 Tolling

There are many possible implementations of a tolling scheme, so the description below should be regarded as an illustration.

Consider a straightforward challenge-response protocol. The tolling point continually sends out signed challenges that contain an identifier, a public key and a nonce. A vehicle approaching the tolling point signs the challenge and returns it encrypted under the tolling point's public key as a signed and encrypted message using the format of this standard.. The tolling point checks that the signature verifies and that the vehicle's certificate has not been revoked. If the checks pass it allows the vehicle to pass through. Otherwise, it activates a standard backup system, such as cameras to capture the vehicle's license plate.

The threats include the following:

— A vehicle masquerades as a toll plaza to trick another vehicle into paying the first vehicle's toll.
— Eavesdropping on secrets such as credit card information while in transmission.
— An eavesdropper uses static identifiers such as cookies to track the vehicle.
— Insider attackers using system information to forge tolling tags.

The security choices in the above brief description have the following motivations:

Signing the challenges prevents attackers from masquerading as a tolling point.

Including the nonce in the challenge and the response prevents replay attacks. The nonce need not be random, and may in fact simply be the time. If the challenge and response use the SignedMessage format, the generation_time field and the procedures outlined in 7.3.3.2 will protect against replay attacks without the need to include an additional nonce field.

A system based on a shared secret (such as a static identifier) could be used instead of a system based on signing. However, a system based on a shared identifier is difficult to add new service providers to, because either the identifier must be shared with all service providers, or all service providers must be online at all times to verify identifiers that they have issued. If the identifier is shared with all service providers, the risk of an insider attack greatly increases. Signing the response, rather than using a shared secret, means that someone who sees a single response is not able to forge a different response.

Encrypting the signed response prevents an eavesdropper from using the tolling points to construct a tracking system. The toll operators are able to track use of their roads, but third-parties cannot decrypt the data transmitted over the air.

F.4 General Internet access

It is anticipated that future versions of the WAVE standards will require OBUs to frequently change MAC and IP address frequently. Such OBUs will straightforwardly support long-lived IP sessions from moving vehicles. This has the advantage that the long-lived IP addresses cannot be used as a tracking mechanism. This standard does not consider tracking stationary vehicles to be a threat.

However, a service provider who provides internet access should also provide a means to protect the data exchanged between a stationary vehicle and a remote Internet device, both authenticating the data and protecting it from eavesdropping. Here, there are three obvious mechanisms to consider: IEEE Std 802.11i [B8] at the link layer, IPSec [B9], [B10], [B12], [B13] at the network layer, and TLS or DTLS at the application layer. Of these, IEEE Std 802.11i [B8] cannot run exactly as specified because IEEE P802.11p [B2] does not use standard IEEE 802.11 Associations. TLS and DTLS are application layer protocols and do not automatically protect all data over the IP link. The most generically secure mechanism for in-vehicle Internet Access applications is therefore IPSec.

F.5 Roadside e-commerce

The class of roadside e-commerce applications refers to applications that interact with a driver on the road to offer some local service. The canonical example of a roadside e-commerce application is an extension of drive-through meal ordering that allows a driver to order a meal before reaching the restaurant where it is collected.

For clarity, consider the example of an application that presents a driver with a menu of choices through a simplified web browser, using cookies to maintain state between RSUs to work around the difficulties with maintaining a long-lived internet session while in motion. The browser may have a local store of the driver's credit card information, activated with a PIN when the driver starts the vehicle, to remove any need for the driver to enter credit card information on the road.

The threats include the following:

— "Phishing" to direct traffic to a fake website and gather credit card information.
— Eavesdropping on secrets such as credit card information while in transmission.
— Replaying an order to get the goods at the original driver's expense.
— Using static identifiers such as cookies to track the vehicle.

Countermeasures for these threats include the following:

— Serve pages with DTLS to reduce the threat of eavesdropping, replaying and tracking.
— Authenticate the server DNS address in the PST and check this against the DNS name in the server's DTLS certificate to reduce the threat from phishing.

Annex G

(informative)

Bandwidth considerations and opportunities for optimization

G.1 Overview

This annex considers mechanisms that could be used to reduce bandwidth and optimize performance.

G.2 Geographic scope identifiers

Many political or administrative regions have complicated shapes that are difficult to encode in a compact form. For example, many counties in the U.S. have 50 or more corners. Including a full description of these borders in a message might increase the size to a point where the packet loss rate becomes unacceptable. This is a particular concern for public safety vehicles, which typically operate within politically defined boundaries and send many time-critical messages.

This issue can be addressed in the following ways:

— Simplify the description of the region—for example, authorize a public safety vehicle to operate within a rectangular or circular region, or omit the curves on a stretch of boundary that is essentially a straight line. Note that this approach may not be acceptable for political reasons.
— Break the region into a series of simpler overlapping subregions that together cover the base region. Provide a certificate for each subregion. The application that emits the signed messages decides which of these certificates to use depending on its current location.
— Use the `from_issuer` identifier to indicate that the appropriate region is described in the next certificate up the chain.

All of these approaches are permitted by this standard.

G.3 Application identifiers

PSOBUs and RSUs could in theory have a single certificate with permissions for all applications. However, the number of bits on the air for a specific application could be reduced by instead using one certificate per application, where the issuing CA certificate contains permissions to issue certificates for that application only. For instance, a traffic signal RSU might be enhanced to provide a maximum speed warning. It may be simpler to issue a second certificate rather than reissue the initial one. Whether to issue multiple certificates or a single certificate with multiple application IDs is a local policy issue.

This standard also provides for a reduction of bits over the air by the use of the `from_issuer` field. Use of the `from_issuer` field indicates that the application identifier is inherited from the next certificate up the chain.

G.4 Use of the certificate_digest SignerInfo type

If the `signer.type` field of a `SignedMessage` or `SignedWSM` is equal to `certificate_digest`, the message includes a reference to the signing certificate (in the form of an eight-octet digest) rather than the certificate itself. An application that sends messages very frequently may choose, depending on environmental conditions, to include its certificate only in certain messages and identify the certificate using its digest in other messages. This reduces bandwidth by almost the size of the

certificate, typically 100 octets. However, the drawback is that a receiving application will not be able to verify a message unless it has received the certificate itself; if it receives any messages in which the certificate is identified by a digest before it has received a message that includes the certificate, those messages will have to be discarded. Therefore this approach should only be used if the application has confidence that its messages that include the full certificate will be successfully received.

Annex H

(informative)

Copyright statement for Clause 4

Much of the material in Clause 4 is derived from IETF RFC 2246 [B11] and is used subject to the following copyright statement.

Full Copyright Statement

Annex I

(informative)

Bibliography

[B1] IEEE 100, *The Authoritative Dictionary of IEEE Standard Terms*, Seventh Edition.

[B2] IEEE P802.11p, Draft Standard for Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements—Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Wireless Access in Vehicular Environments (WAVE).[13]

[B3] IEEE P1609.1, Draft Standard for Wireless Access in Vehicular Environments—WAVE Resource Manager.

[B4] IEEE P1609.4/Draft 6, November 2005, Draft Standard for Wireless Access in Vehicular Environments (WAVE) Multi-Channel Operation.

[B5] IEEE Std 802.2 (ISO/IEC 8802-2: 1998), Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements—Part 2: Logical link control.

[B6] IEEE Std 802.11, Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements—Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications.

[B7] IEEE Std 802.11a™-1999, Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements—Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: High-speed Physical Layer in the 5 GHz Band.

[B8] IEEE Std 802.11i™-2004, Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements—Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Amendment 6: Medium Access Control (MAC) Security Enhancements.

[B9] IETF Request for Comments: 1826, IP Authentication Header.[15]

[B10] IETF Request for Comments: 1827, IP Encapsulating Security Payload (ESP).

[13] Numbers preceded by P are IEEE authorized standards projects that were not approved by the IEEE-SA standards board at the time this publication went to press. For information about obtaining drafts, contact the IEEE (http://standards.ieee.org). Upon approval by the IEEE-SA Standards Board, the approved IEEE standards will supersede the related drafts (e.g., IEEE P1234 will be available as IEEE Std 1234-200X, where 200X is the year of approval).

[15] RFCs can be downloaded from http://www.ietf.org.

[B11] IETF Request for Comments: 2246, The TLS Protocol Version 1.0.

[B12] IETF Request for Comments: 2401, Security Architecture for the Internet Protocol.

[B13] IETF Request for Comments: 2409, The Internet Key Exchange (IKE).

[B14] IETF Request for Comments: 2460, Internet Protocol, Version 6 (IPv6) Specification.

[B15] IETF Request for Comments: 2463, Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification.

[B16] IETF Request for Comments: 3280, Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.

[B17] IETF Request for Comments: 3315, Dynamic Host Configuration Protocol for IPv6 (DHCPv6).

[B18] IETF Request for Comments: 3369, Cryptographic Message Syntax (CMS).

[B19] IETF Request for Comments: 3450, Asynchronous Layered Coding (ALC) Protocol Instantiation.

[B20] Modadugu, N., and Rescorla, E., *The Design and Implementation of Datagram TLS*, Proceedings of NDSS 2004, February 2004.

[B21] The National ITS Architecture: A Framework for Integrated Transportation in the 21st Century. Version 5.0, November 2003.

[B22] SAE Draft J2735 v05, Dedicated Short Range Communications (DSRC) Message Set Dictionary, November 2005.