IEEE Std 802.11i™-2004
[Amendment to IEEE Std 802.11™, 1999 Edition (Reaff 2003)
as amended by
IEEE Stds 802.11a™-1999, 802.11b™-1999,
802.11b™-1999/Cor 1-2001, 802.11d™-2001,
802.11g™-2003, and 802.11h™-2003]

IEEE Standards

802.11i™

IEEE 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

Amendment 6: Medium Access Control (MAC) Security Enhancements

IEEE Computer Society

Sponsored by the
LAN/MAN Standards Committee

This amendment is an approved IEEE Standard. It will be incorporated into the base standard in a future edition.

IEEE

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

23 July 2004

Print: SH95248
PDF: SS95248

IEEE Std 802.11i™-2004
[Amendment to IEEE Std 802.11™, 1999 Edition (Reaff 2003)
as amended by IEEE Stds 802.11a™-1999, 802.11b™-1999,
802.11b™-1999/Cor 1-2001, 802.11d™-2001,
802.11g™-2003, and 802.11h™-2003]

IEEE 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

Amendment 6: Medium Access Control (MAC) Security Enhancements

Sponsor
LAN/MAN Committee
of the
IEEE Computer Society

Approved 24 June 2004

IEEE-SA Standards Board

Abstract: Security mechanisms for IEEE 802.11 are defined in this amendment, which includes a definition of WEP for backward compatibility with the original standard, IEEE Std 802.11, 1999 Edition. This amendment defines TKIP and CCMP, which provide more robust data protection mechanisms than WEP affords. It introduces the concept of a security association into IEEE 802.11 and defines security association management protocols called the 4-Way Handshake and the Group Key Handshake. Also, it specifies how IEEE 802.1X may be utilized by IEEE 802.11 LANs to effect authentication.
Keywords: AES, authentication, CCM, CCMP, confidentiality, countermeasures, data authenticity, EAPOL-Key, 4-Way Handshake, Group Key Handshake, IEEE 802.1X, key management, key mixing, Michael, RC4, replay protection, robust security network, RSN, security, security association, TKIP, WEP

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

Print: ISBN 0-7381-4073-2 SH95248
PDF: ISBN 0-7381-4074-0 SS95248

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

P.O. Box 1331

Piscataway, NJ 08855-1331USA

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

Introduction

[This introduction is not part of IEEE Std 802.11i™-2004, IEEE Standard for Information Technology—Telecommunications and Information Exchange Between Systems—Local and Metropolitan Area Networks—Specific Requirements—Part 11: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications—Amendment 6: Medium Access Control (MAC) Security Enhancements.)]

Enhanced security services and mechanisms for the IEEE 802.11 medium access control (MAC) beyond those features and capabilities provided by the wired equivalent privacy (WEP) mechanism of the base standard, IEEE Std 802.11, 1999 Edition, are defined in this amendment. This amendment retains the WEP feature for purposes of backwards compatibility with existing IEEE 802.11 devices, but WEP is deprecated in favor of the new security features provided in this amendment.

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.

Participants

At the time the draft of this amendment was sent to sponsor ballot, the IEEE 802.11 Working Group had the following officers:

Stuart J. Kerry, *Chair*
Al Petrick and **Harry Worstell,** *Vice-Chairs*
Tim Godfrey, *Secretary*
Brian Mathews, *Publicity Standing Committee*

Tan Teik-Kheong, *Wireless Next Generation Standing Committee*
Terry L. Cole, *Editor*

John Fakatselis, *Chair Task Group e*
Duncan Kitchen, *Vice-Chair Task Group e*
Sheung Li, *Chair Task Group j*
Richard Paine, *Chair Task Group k*
Bob O'Hara, *Chair Task Group m*
Bruce Kramer, *Chair Task Group n*

When the IEEE 802.11 Working Group approved this amendment, the Task Group I had the following membership:

David Halasz, *Chair*
Jesse Walker, *Editor*
Frank Ciotti, *Secretary*

Osama Aboul-Magd
Tomoko Adachi
Jaemin Ahn
Thomas Alexander
Areg Alimian
Richard Allen
Keith Amann
Dov Andelman
Merwyn Andrade
Carl Andren
David Andrus
Butch Anton
Hidenori Aoki
Tsuguhide Aoki
Michimasa Aramaki
Takashi Aramaki
William Arbaugh
Lee Armstrong
Larry Arnett
Hiroshi Asai
Yusuke Asai
Arthur Astrin
Malik Audeh
Geert Awater
Shahrnaz Azizi
Floyd Backes
Jin-Seok Bae
David Bagby
Dennis Baker
Ramanathan Balachander
Jaiganesh Balakrishnan
Boyd Bangerter
John Barr
Simon Barber
Farooq Bari
Michael Barkway
Kevin Barry
Anuj Batra
Burak Baysal
Tomer Bentzion
Mathilde Benveniste
Don Berry
Jan Biermann
Arnold Bilstad
Harry Bims
Bjorn Bjerke
Simon Black
Jan Boer
William Brasier
Jennifer Bray
Phillip Brownlee
Alex Bugeja
Alistair Buttar
Peter Cain
Richard Cam
Nancy Cam-Winget
Bill Carney
Pat Carson
Broady Cash
Jayant Chande
Kisoo Chang
Clint Chaplin
Ye Chen
Hong Cheng
Greg Chesson
Aik Chindapol
Sunghyun Choi
Won-Joon Choi
Woo-Yong Choi
Yang-Seok Choi
Per Christoffersson
Simon Chung
Ken Clements
Sean Coffey
Terry L. Cole
Paul Congdon
W. Steven Conner
Charles Cook
Kenneth Cook
Mary Cramer
Steven Crowley
Nora Dabbous
Rolf De Vegt
Javier del Prado Pavon
Georg Dickmann
Yoshiharu Doi
Brett Douglas
Simon Duggins
Baris Dundar
Bryan Dunn
Roger Durand
Eryk Dutkiewicz
Mary DuVal
Yaron Dycian
Dennis Eaton
Peter Ecclesine
Jonathan Edney
Bruce Edwards
Natarajan Ekambaram
Jason Ellis
Darwin Engwer
Jeff Erwin
Andrew Estrada
Christoph Euscher
Knut Evensen
John Fakatselis
Lars Falk
Steve Fantaske
Paul Feinberg
Alex Feldman
Weishi Feng
Nestor Fesas
Matthew Fischer
Wayne Fisher
Helena Flygare
Brian Ford
Ruben Formoso
Sheila Frankel
John Fuller
James Gardner
Atul Garg
Albert Garrett
Ramez Gerges
Noam Geri

Vafa Ghazi
Monisha Ghosh
James Gilb
Jeffrey Gilbert
Rabinder Gill
Tim Godfrey
Wataru Gohda
Yuri Goldstein
Jim Goodman
Aviv Goren
Andrew Gowans
Rik Graulus
Gordon Gray
Evan Green
Patrick Green
Kerry Greer
Daqing Gu
Rajugopal Gubbi
Sam Guirguis
Srikanth Gummadi
Qiang Guo
Vivek Gupta
Herman Haisch
Steven Halford
Robert Hall
Neil Hamady
Mounir Hamdi
Christopher Hansen
Yasuo Harada
Daniel Harkins
Thomas Haslestad
Amer Hassan
Vann Hasty
James Hauser
Yutaka Hayakawa
Morihiko Hayashi
Haixiang He
Xiaoning He
Robert Heile
Frans Hermodsson
Dave Hetherington
Guido Hiertz
Garth Hillman
Christopher Hinsz
Jun Hirano
Mikael Hjelm
Jin-Meng Ho
Michael Hoghooghi
Allen Hollister
Keith Holt
Satoru Hori
William Horne
Srinath Hosur
Frank Howley
Yungping Hsu
Robert Huang
Dave Hudak
David Hunter
Syang-Myau Hwang
David Hytha
Muhammad Ikram
Daichi Imamura
Kimihiko Imamura
Yasuhiko Inoue
Katsumi Ishii
Stephen Jackson
Eric Jacobsen
Marc Jalfon
KyungHun Jang
Bruno Jechoux
Taehyun Jeon
Moo Ryong Jeong
Daniel Jiang
Kuniko Jimi
Walter Johnson
David Johnston
Jari Jokela
VK Jones
Bobby Jose
Tyan-Shu Jou
Carl Kain
Srinivas Kandala
You Sung Kang
Jeyhan Karaoguz
Kevin Karcz
Pankaj Karnik
Mika Kasslin
Dean Kawaguchi
Patrick Kelly
Richard Kennedy
Stuart Kerry
John Ketchum
Vytas Kezys
Andrew Khieu
Ryoji Kido
Tomohiro Kikuma
Byoung-Jo Kim
Dooseok Kim
Joonsuk Kim
Yongbum Kim
Yongsuk Kim
Young Kim
Youngsoo Kim
Wayne King
John Klein
Guenter Kleindl
Toshiya Kobashi
Keiichiro Koga
Lalit Kotecha
John Kowalski
Bruce Kraemer
Gopal Krishnan
Shuji Kubota
Thomas Kuehnel
Tomoaki Kumagai
Takushi Kunihiro
Thomas M. Kurihara
Denis Kuwahara
Joe Kwak
Paul Lambert
David Landeta
Jim Lansford
Colin Lanzl
Choi Law
Dongjun Lee
Insun Lee
Jae Hwa Lee
Marty Lefkowitz
Onno Letanche
Joseph Levy
Mike Lewis
Pen Li
Quinn Li
Sheung Li
Jie Liang
Wei Lih Lim
Yong Je Lim
Huashih Lin
Sheng Lin
Victor Lin
Stanley Ling
Der-Zheng Liu
I-Ru Liu
Yonghe Liu
Titus Lo
Peter Loc
Patrick Lopez
Hui-Ling Lou
Xiaolin Lu
Luke Ludeman
Yi-Jen Lung
Akira Maeki
Ravishankar Mahadevappa
Doug Makishima
Majid Malek
Rahul Malik
Jouni Malinen
Krishna Malladi
Stefan Mangold
Mahalingam Mani
Jonn Martell
Naotaka Maruyama
Paul Marzec
Brian Mathews
Yoichi Matsumoto
Sudheer Matta
Thomas Maufer
Conrad Maxwell
Stephen McCann
Kelly McClellan
Gary McCoy
William McFarland
Timothy McGovern
Bill McIntosh
Justin McNew
Irina Medvedev
Pratik Mehta
Robert Meier
Graham Melvile
Klaus Meyer
Robert Miller
Partho Mishra
David Mitton
Kenichi Miyoshi
Rishi Mohindra
Peter Molnar
Leo Monteban
Michael Montemurro
Rondal Moore
Tim Moore
Anthony Morelli
Mike Moreton

Yuichi Morioka
Steven Morley
Robert Moskowitz
Joseph Mueller
Syed Mujtaba
Willem Mulder
Peter Murphy
Peter Murray
Andrew Myers
Andrew Myles
Yukimasa Nagai
Katsuyoshi Naka
Makoto Nakahara
Michiharu Nakamura
Seigo Nakao
Hiroyuki Nakase
Sanjiv Nanda
Ravi Narasimhan
Slobodan Nedic
Robert Neilsen
David Nelson
Dan Nemits
Chiu Ngo
Tuan Nguyen
Qiang Ni
Gunnar Nitsche
Erwin Noble
Tzvetan Novkov
Ivan Oakes
Kei Obara
Karen O'Donoghue
Hiroshi Oguma
Jongtaek Oh
Bob O'Hara
Sean O'Hara
Yoshihiro Ohtani
Chandra Olson
Timothy Olson
Hiroshi Ono
Peter Oomen
Lior Ophir
Satoshi Oyama
Richard Paine
Michael Paljug
Stephen Palm
Jong Ae Park
Jonghun Park
Joon Goo Park
Taegon Park
Steve Parker
Glenn Parsons
Vijay Patel
Eldad Perahia
Sebastien Perrot
Al Petrick
Joe Pitarresi
Leo Pluswick
Stephen Pope
James Portaro
Al Potter
Henry Ptasinski
Anuj Puri
Aleksandar Purkovic
Jim Raab
Ali Raissinia
Ajay Rajkumar
Noman Rangwala
Ivan Reede
Stanley Reible
Anthony Reid
Joe Repice
Edward Reuss
Valentine Rhodes
Maximilian Riegel
Edmund Ring
Carlos Rios
Stefan Rommer
Jon Rosdahl
John Sadowsky
Ali Sadri
Kazuyuki Sakoda
Shoji Sakurai
Kenichi Sakusabe
Hemanth Sampath
Sumeet Sandhu
Anil Sanwalka
Ryo Sawai
Tom Schaffnit
Brian Schreder
Sid Schrum
Erik Schylander
Michael Seals
Joe Sensendorf
N. Shankaranarayanan
Donald Shaver
Stephen Shellhammer
Tamara Shelton
Ian Sherlock
Matthew Sherman
Ming Sheu
Shusaku Shimada
Matthew Shoemake
William Shvodian
D. J. Shyy
Thomas Siep
Floyd Simpson
Manoneet Singh
Hasse Sinivaara
Efstratios (Stan) Skafidas
David Skellern
Roger Skidmore
Donald Sloan
Kevin Smart
David Smith
Yoram Solomon
V. Somayazulu
Amjad Soomro
Robert Soranno
Gary Spiess
William Spurgeon
Dorothy Stanley
William Steck
Greg Steele
Adrian Stephens
William Stevens
Carl Stevenson
Fred Stivers
Warren Strand
Paul Struhsaker
Michael Su
Hiroki Sugimoto
Abhaya Sumanasena
Qinfang Sun
SK Sung
Shravan Surineni
Hirokazu Tagiri
Masahiro Takagi
Mineo Takai
Katsumi Takaoka
Daisuke Takeda
Nir Tal
Tsuyoshi Tamaki
Pek-Yew Tan
Teik-Kheong Tan
Wai-Cheung Tang
Takuma Tanimoto
Henry Taylor
James Taylor
Carl Temme
Stephan ten Brink
John Terry
Timothy Thornton
Jerry Thrasher
James Tomcik
Allen Tsai
Jean Tsao
Chih Tsien
Tom Tsoulogiannis
Kwei Tu
David Tung
Sandra Turner
Mike Tzamaloukas
Marcos Tzannes
Yusuke Uchida
Takashi Ueda
Naoki Urano
Hidemi Usuba
Chandra Vaidyanathan
Hans Van Leeuwen
Richard van Leeuwen
Richard Van Nee
Nico van Waes
Allert van Zelst
Madan Venugopal
George Vlantis
Dennis Volpano
Tim Wakeley
Brad Wallace
Thierry Walrant
Vivek Wandile
Huaiyuan Wang
Stanley Wang
Christopher Ware
Fujio Watanabe
Mark Webster
Matthew Welborn
Bryan Wells
Filip Weytjens
Stephen Whitesell
Michael Wilhoyte
Michael Glenn Williams
Peter Williams

Richard Williams
James Wilson
Steven Wilson
Jack Winters
Jin Kue Wong
Timothy Wong
Patrick Worfolk
Harry Worstell
Charles Wright
Gang Wu
Yang Xiao
James Yee
Jung Yee
Kazim Yildiz
Jijun Yin
Kit Yong
Heejung Yu
Hon Yung
Erol Yurtkuran
Zhun Zhong
Glen Zorn
James Zyren

Major contributions were received from the following individuals:

Bernard Aboba
Areg Alimian
Keith Amann
Merwyn Andrade
Arun Ayyagari
Butch Anton
Bob Beach
Simon Black
Simon Blake-Wilson
Nancy Cam-Winget
Clint Chaplin
Greg Chesson
Alan Chickinsky
Frank Ciotti
Donald Eastlake III
Jonathan Edney
Niels Ferguson
Aaron Friedman
Craig Goston
Larry Green
Daniel Harkins
Dan Hassett
Kevin Hayes
Russ Housley
Jin-Meng Ho
Dick Hubbard
Tony Jeffree
Hong Jiang
David Johnston
Asa Kalvade
Kevin Karcz
Paul Lambert
Marty Lefkowitz
Onno Letanche
Jie Liang
Jouni Malinen
Thomas Maufer
Kelly McClellan
Bill McIntosh
Graham Melville
Tim Moore
Leo Monteban
Mike Moreton
Robert Moskowitz
David Nelson
Bob O'Hara
Richard Paine
Henry Ptasinski
Ivan Reede
Carlos Rios
Phil Rogaway
Mike Sabin
Dan Simon
Doug Smith
Mike Sordi
Dorothy Stanley
Fred Stivers
Sandra Turner
Dennis Volpano
Doug Whiting
Albert Young
Glen Zorn
Arnoud Zwemmer

This project was balloted using individual balloting. The following members of the balloting committee voted on this amendment. Balloters may have voted for approval, disapproval, or abstention.

Tomoko Adachi
John Adams
Toru Aihara
James Allen
Keith Amann
Butch Anton
Eladio Arvelo
Colin Ayer
David Bagby
Daniel Bailey
John Barr
Les Baxter
Anader Benyamin-Seeyar
Barbara Bickham
Jan Boer
Gennaro Boggia
Gary Bourque
Ed Callaway
Lon Canaday
Edward Carley
Bill Carney
Clint Chaplin
Amalavoyal Chari
Brendon Chetwynd
Alan Chickinsky
Aik Chindapol
Keith Chow
Terry L. Cole
Christopher Cooke
Todor Cooklev
Todd Cooper
Javier del Prado Pavon
Guru Dutt Dhingra
Thomas Dineen
Lakshminath Dondeti
Vern Dubendorf
Sourav Dutta
Clint Early
Jonathan Edney
Carl Eklund
Michael Fischer
Michele Gammel
Corey Gates
Theodore Georgantas
Andrew Germano
Tim Godfrey
Jose Gutierrez
Chris Guy
David Halasz
Karen Halford
Steven Halford
Christopher Hansen
Robert Heile
Stuart Holoman
Russell Housley
Atsushi Ito
Peeya Iwagoshi
Tony Jeffree
David Johnston
Bobby Jose
Joe Juisai
Thomas M. Kurihara
Srinivas Kandala
Kevin Karcz
Pankaj Karnik
Michael Kelsen
Stuart Kerry
Brian Kiernan
Thomas Kolze
John Kowalski
Joe Kubler
Denis Kuwahara
William Lane
Colin Lanzl
John Lemon
Jie Liang

Jan-Ray Liao
Randolph Little
Gregory Luri
Ryan Madron
Peter Martini
Kelly McClellan
Michael McInnis
Ingolf Meier
George Miao
Yinghua Min
Apurva Mody
Leo Monteban
Mike Moreton
Robert Moskowitz
Oliver Muelhens
Andrew Myles
Paul Nikolich
Erwin Noble
Bob O'Hara
Satoshi Oyama
Leo Pluswick
Richard Paine
Stephen Palm
Roger Pandanda
Subbu Ponnuswamy
Albert Potter
Vikram Punj
Bijan Raahemi
Moshe Ran
Terry Richards
Maximilian Riegel
Calvin Roberts
David Rockwell
Mike Rudnick
Tom Siep
Thomas Sapiano
John Sarallo
Durga Satapathy
George She
Hiroyasu Shimizu
Akihiro Shimura
Matthew Shoemake
Gil Shultz
Yoram Solomon
Amjad Soomro
Kenneth Stanwood
Thomas Starai
Adrian Stephens
Carl Stevenson
Masahiro Takagi
Pek Yew Tan
Joseph Tardo
Jerry Thrasher
Jim Tomcik
Scott Valcourt
Richard van Leeuwen
Hung-yu Wei
Stephen Whitesell
Dave Willow
Harry Worstell
Shugong Xu
Jung Yee
Patrick Yu
Oren Yuen
Arnoud Zwemmer

When the IEEE-SA Standards Board approved this standard on 24 June 2004, it had the following membership:

Don Wright, *Chair*
Steve M. Mills, *Vice Chair*
Judith Gorman, *Secretary*

Chuck Adams
H. Stephen Berger
Mark D. Bowman
Joseph A. Bruder
Bob Davis
Roberto de Boisson
Julian Forster*
Arnold M. Greenspan
Mark S. Halpin
Raymond Hapeman
Richard J. Holleman
Richard H. Hulett
Lowell G. Johnson
Joseph L. Koepfinger*
Hermann Koch
Thomas J. McGean
Daleep C. Mohla
Paul Nikolich
T. W. Olsen
Ronald C. Petersen
Gary S. Robinson
Frank Stone
Malcolm V. Thaden
Doug Topping
Joe D. Watson

*Member Emeritus

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

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

Savoula Amanatidis
IEEE Standards Managing Editor

Contents

IEEE 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

Amendment 6: Medium Access Control (MAC) Security Enhancements

[This amendment is based on IEEE Std 802.11™, 1999 Edition (Reaff 2003), as amended by IEEE Stds 802.11a™-1999, 802.11b™-1999, 802.11b™-1999/Cor 1-2001, 802.11d™-2001, 802.11g™-2003, and 802.11h™-2003.]

NOTE—The editing instructions contained in this amendment define how to merge the material contained herein into the existing base standard and its amendments to form the comprehensive standard.

The editing instructions are shown in ***bold italic***. Four editing instructions are used: change, delete, insert, and replace. ***Change*** is used to make small corrections in existing text or tables. The editing instruction specifies the location of the change and describes what is being changed either by using ~~strikethrough~~ (to remove old material) or underscore (to add new material). ***Delete*** removes existing material. ***Insert*** adds new material without disturbing the existing material. Insertions may require renumbering. If so, renumbering instructions are given in the editing instructions. ***Replace*** is used to make large changes in existing text, subclauses, tables, or figures by removing existing material and replacing it with new material. Editorial notes will not be carried over into future editions.

1. Overview

1.2 Purpose

Change the fifth bullet of 1.2 as follows:

— Describes the requirements and procedures to provide ~~privacy~~confidentiality of user information being transferred over the wireless medium (WM) and authentication of IEEE 802.11 conformant devices.

End of changes to Clause 1.

2. Normative references

Insert the following references at the appropriate locations in Clause 2:

FIPS PUB 180-1-1995, Secure Hash Standard.[1]

FIPS PUB 197-2001, Advanced Encryption Standard (AES).

IEEE P802.1X™-REV, Draft Standard for Local and Metropolitan Area Networks: Port-Based Network Access Control.[2, 3, 4]

IETF RFC 1321, The MD5 Message-Digest Algorithm, April 1992.[5]

IETF RFC 1750, Randomness Recommendations for Security, December 1994.

IETF RFC 2104, HMAC: Keyed-Hashing for Message Authentication, February 1997.

IETF RFC 2202, Test Cases for HMAC-MD5 and HMAC-SHA-1, September 1997.

IETF RFC 3394, Advanced Encryption Standard (AES) Key Wrap Algorithm, September 2002.

IETF RFC 3610, Counter with CBC-MAC (CCM), September 2003.

IETF RFC 3748, Extensible Authentication Protocol (EAP), March 2004.

End of changes to Clause 2.

3. Definitions

Delete the definition "3.40 privacy."

Change the definition for "wired equivalent privacy (WEP)" as follows:

3.49 wired equivalent privacy (WEP): ~~The~~ An optional cryptographic confidentiality algorithm specified by IEEE 802.11 that may be used to provide data confidentiality that is subjectively equivalent to the confidentiality of a wired local area network (LAN) medium that does not employ cryptographic techniques to enhance ~~privacy~~ confidentiality.

Insert the following definitions in alphabetical order into Clause 3, renumbering as necessary:

3.63 additional authentication data (AAD): Data that are not encrypted, but are cryptographically protected.

---

[1]FIPS publications are available from the National Technical Information Service (NTIS), U. S. Dept. of Commerce, 5285 Port Royal Road, Springfield, VA 22161 (http://www.ntis.org/).

[2]IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).

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

[4]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.

[5]Internet RFCs are available from the Internet Engineering Task Force at http://www.ietf.org/.

3.64 authentication, authorization, and accounting (AAA) key: Key information that is jointly negotiated between the Supplicant and the Authentication Server (AS). This key information is transported via a secure channel from the AS to the Authenticator. The pairwise master key (PMK) may be derived from the AAA key.

3.65 authentication and key management (AKM) suite: A set of AKM suite selectors.

3.66 Authentication Server (AS): An entity that provides an authentication service to an Authenticator. This service determines, from the credentials provided by the Supplicant, whether the Supplicant is authorized to access the services provided by the Authenticator. (IEEE P802.1X-REV[6])

3.67 Authenticator: An entity at one end of a point-to-point LAN segment that facilitates authentication of the entity attached to the other end of that link. (IEEE P802.1X-REV)

3.68 Authenticator address (AA): The IEEE 802.1X Authenticator medium access control (MAC) address.

3.69 authorized: To be explicitly allowed.

3.70 big endian: The concept that, for a given multi-octet numeric representation, the most significant octet has the lowest address.

3.71 cipher suite: A set of one or more algorithms, designed to provide data confidentiality, data authenticity or integrity, and/or replay protection.

3.72 counter mode (CTR) with CBC-MAC [cipher-block chaining (CBC) with message authentication code (MAC)] (CCM): A symmetric key block cipher mode providing confidentiality using CTR and data origin authenticity using CBC-MAC.

NOTE—See IETF RFC 3610.

3.73 decapsulate: To recover an unprotected frame from a protected one.

3.74 decapsulation: The process of generating plaintext data by decapsulating an encapsulated frame.

3.75 EAPOL-Key confirmation key (KCK): A key used to integrity-check an EAPOL-Key frame.

3.76 EAPOL-Key encryption key (KEK): A key used to encrypt the Key Data field in an EAPOL-Key frame.

3.77 encapsulate: To construct a protected frame from an unprotected frame.

3.78 encapsulation: The process of generating the cryptographic payload from the plaintext data. This comprises the cipher text as well as any associated cryptographic state required by the receiver of the data, e.g., initialization vectors (IVs), sequence numbers, message integrity codes (MICs), key identifiers.

3.79 4-Way Handshake: A pairwise key management protocol defined by this amendment. It confirms mutual possession of a pairwise master key (PMK) by two parties and distributes a group temporal key (GTK).

3.80 group: The entities in a wireless network, e.g., an access point (AP) and its associated stations (STAs), or all the STAs in an independent basic service set (IBSS) network.

[6]Information on references can be found in Clause 2.

3.81 Group Key Handshake: A group key management protocol defined by this amendment. It is used only to issue a new group temporal key (GTK) to peers with whom the local station (STA) has already formed security associations.

3.82 group master key (GMK): An auxiliary key that may be used to derive a group temporal key (GTK).

3.83 group temporal key (GTK): A random value, assigned by the broadcast/multicast source, which is used to protect broadcast/multicast medium access control (MAC) protocol data units (MPDUs) from that source. The GTK may be derived from a group master key (GMK).

3.84 group temporal key security association (GTKSA): The context resulting from a successful group temporal key (GTK) distribution exchange via either a Group Key Handshake or a 4-Way Handshake.

3.85 IEEE 802.1X authentication: Extensible Authentication Protocol (EAP) authentication transported by the IEEE 802.1X protocol.

3.86 key counter: A 256-bit (32-octet) counter that is used in the pseudo-random function (PRF) to generate initialization vectors (IVs). There is a single key counter per station (STA) that is global to that STA.

3.87 key data encapsulation (KDE): Format for data other than information elements in the EAPOL-Key Data field.

3.88 key management service: A service to distribute and manage cryptographic keys within a robust security network (RSN).

3.89 little endian: The concept that, for a given multi-octet numeric representation, the least significant octet has the lowest address.

3.90 liveness: A demonstration that the peer is actually participating in this instance of communication.

3.91 message integrity code (MIC): A value generated by a symmetric key cryptographic function. If the input data are changed, a new value cannot be correctly computed without knowledge of the symmetric key. Thus, the secret key protects the input data from undetectable alteration. This is traditionally called a *message authentication code* (MAC), but the acronym MAC is already reserved for another meaning in this amendment.

3.92 Michael: The message integrity code (MIC) for the Temporal Key Integrity Protocol (TKIP).

3.93 nonce: A value that shall not be reused with a given key, including over all reinitializations of the system through all time.

3.94 pairwise: Two entities that are associated with each other, e.g., an access point (AP) and an associated station (STA), or a pair of STAs in an independent basic service set (IBSS) network. This term is used to describe the key hierarchies for keys that are shared only between the two entities in a pairwise association.

3.95 pairwise master key (PMK): The highest order key used within this amendment. The PMK may be derived from an Extensible Authentication Protocol (EAP) method or may be obtained directly from a preshared key (PSK).

3.96 pairwise master key security association (PMKSA): The context resulting from a successful IEEE 802.1X authentication exchange between the peer and Authentication Server (AS) or from a preshared key (PSK).

3.97 pairwise transient key (PTK): A value that is derived from the pairwise master key (PMK), Authenticator address (AA), Supplicant address (SPA), Anthenticator nonce (ANonce), and Supplicant nonce (SNonce) using the pseudo-random function (PRF) and that is split up into as many as five keys, i.e., temporal encryption key, two temporal message integrity code (MIC) keys, EAPOL-Key encryption key (KEK), EAPOL-Key confirmation key (KCK).

3.98 pairwise transient key security association (PTKSA): The context resulting from a successful 4-Way Handshake exchange between the peer and Authenticator.

3.99 pass-phrase: A secret text string employed to corroborate the user's identity.

3.100 per-frame encryption key: A unique encryption key constructed for each medium access control (MAC) protocol data unit (MPDU), employed by some IEEE 802.11 security protocols.

3.101 per-frame sequence counter: For Temporal Key Integrity Protocol (TKIP), the counter that is used as the nonce in the derivation of the per-frame encryption key. For CCM [counter mode (CTR) with cipher-block chaining (CBC) with message authentication code (MAC)] Protocol (CCMP), the per-frame initialization vector (IV).

3.102 pre-robust security network association (pre-RSNA): The type of association used by a pair of stations (STAs) if the procedure for establishing authentication or association between them did not include the 4-Way Handshake.

3.103 pre-robust security network association (pre-RSNA) equipment: A device that is not able to create robust security network associations (RSNAs).

3.104 preshared key (PSK): A static key that is distributed to the units in the system by a method outside the scope of this amendment, always by some out-of-band means.

3.105 pseudo-random function (PRF): A function that hashes various inputs to derive a pseudo-random value. To add liveness to the pseudo-random value, a nonce should be one of the inputs.

3.106 robust security network (RSN): A security network that allows only the creation of robust security network associations (RSNAs). An RSN can be identified by the indication in the RSN Information Element (IE) of Beacon frames that the group cipher suite specified is not wired equivalent privacy (WEP).

3.107 robust security network association (RSNA): The type of association used by a pair of stations (STAs) if the procedure to establish authentication or association between them includes the 4-Way Handshake. Note that the existence of a RSNA by a pair of devices does not of itself provide robust security. Robust security is provided when all devices in the network use RSNAs.

3.108 robust-security-network-association- (RSNA-) capable equipment: A station (STA) that is able to create RSNAs. Such a device can use pre-RSNAs because of configuration. Notice that RSNA-capable does not imply full compliance with the RSNA Protocol Implementation Conformance Statement (PICS). A legacy device that has been upgraded to support Temporal Key Integrity Protocol (TKIP) can be RSNA-capable, but will not be compliant with the PICS if it does not also support CCM [counter mode (CTR) with cipher-block chaining (CBC) with message authentication code (MAC)] Protocol (CCMP).

3.109 robust-security-network-association- (RSNA-) enabled equipment: A station (STA) when it is RSNA-capable and `dot11RSNAEnabled` is set to TRUE.

3.110 robust security network association (RSNA) key management: Key management that includes the 4-Way Handshake, the Group Key Handshake, and the STAKey Handshake.

3.111 security network: A basic service set (BSS) where the station (STA) starting the BSS provides information about the security capabilities and configuration of the BSS by including the robust security network (RSN) information element in Beacon frames.

3.112 selector: An item specifying a list constituent in an IEEE 802.11 Management Message information element.

3.113 STAKey: A symmetric key used to protect direct station-to-station (STA-to-STA) communication in an infrastructure basic service set (BSS).

3.114 STAKey Handshake: A STAKey key management protocol, used to issue a new STAKey to stations (STAs) that have a pairwise transient key security association (PTKSA) with an access point (AP).

3.115 STAKey security association (STAKeySA): The security context for direct station-to-station (STA-to-STA) communication in an infrastructure basic service set (BSS). A STAKeySA includes a STAKey.

3.116 Supplicant: An entity at one end of a point-to-point LAN segment that is being authenticated by an Authenticator attached to the other end of that link. (IEEE P802.1X-REV)

3.117 Supplicant address (SPA): The Supplicant's medium access control (MAC) address**.**

3.118 temporal encryption key: The portion of a pairwise transient key (PTK) or group temporal key (GTK) used directly or indirectly to encrypt data in medium access control (MAC) protocol data units (MPDUs).

3.119 temporal key: The combination of temporal encryption key and temporal message integrity code (MIC) key.

3.120 temporal message integrity code (MIC) key: The portion of a transient key used to ensure the integrity of medium access control (MAC) service data units (MSDUs) or MAC protocol data units (MPDUs).

3.121 transition security network (TSN): A security network that allows the creation of pre-robust security network associations (pre-RSNAs) as well as RSNAs. A TSN can be identified by the indication in the robust security network (RSN) information element of Beacon frames that the group cipher suite in use is wired equivalent privacy (WEP).

End of changes to Clause 3.

4. Abbreviations and acronyms

Insert the following abbreviations in alphabetical order into Clause 4:

AA	Authenticator address
AAA	authentication, authorization, and accounting
AAD	additional authentication data
AES	advanced encryption standard
AKM	authentication and key management
AKMP	Authentication and Key Management Protocol
ANonce	Authenticator nonce
ARP	Address Resolution Protocol
AS	Authentication Server
CBC	cipher-block chaining
CBC-MAC	cipher-block chaining with message authentication code

CCM	CTR with CBC-MAC
CCMP	CTR with CBC-MAC Protocol
CTR	counter mode
EAP	Extensible Authentication Protocol (IETF RFC 3748)
EAPOL	Extensible Authentication Protocol over LANs (IEEE P802.1X-REV)
GMK	group master key
GNonce	group nonce
GTK	group temporal key
GTKSA	group temporal key security association
ICMP	Internet Control Message Protocol
KCK	EAPOL-Key confirmation key
KDE	key data encapsulation
KEK	EAPOL-Key encryption key
LFSR	linear feedback shift register
MIC	message integrity code
NTP	Network Time Protocol (IETF RFC 1305 [B12][7])
OUI	organizationally unique identifier
PAE	port access entity (IEEE P802.1X-REV)
PMK	pairwise master key
PMKID	pairwise master key identifier
PMKSA	pairwise master key security association
PN	packet number
PRF	pseudo-random function
PRNG	pseudo-random number generator
PSK	preshared key
PTK	pairwise transient key
PTKSA	pairwise transient key security association
RADIUS	remote authentication dial-in user service (IETF RFC 2865 [B14])
RSC	broadcast/multicast transmit sequence counter
RSN	robust security network
RSNA	robust security network association
SNAP	Sub-Network Access Protocol
SNonce	Supplicant nonce
SPA	Supplicant address
TKIP	Temporal Key Integrity Protocol
TSC	TKIP sequence counter
TSN	transition security network
TTAK	TKIP-mixed transmit address and key
UCT	unconditional transfer

End of changes to Clause 4.

[7]The numbers in brackets correspond to the numbers of the bibliography in Annex E.

5. General description

5.1 General description of the architecture

5.1.1 How wireless LAN systems are different

5.1.1.4 Interaction with other IEEE 802® layers

Insert the following paragraph at the end of 5.1.1.4:

In a robust security network association (RSNA), IEEE 802.11 provides functions to protect data frames, IEEE 802.1X provides authentication and a Controlled Port, and IEEE 802.11 and IEEE 802.1X collaborate to provide key management. All stations (STAs) in an RSNA have a corresponding IEEE 802.1X entity that handles these services. This amendment defines how an RSNA utilizes IEEE 802.1X to access these services.

After 5.1.1.4, insert 5.1.1.5:

5.1.1.5 Interaction with non-IEEE 802 protocols

An RSNA utilizes non-IEEE 802 protocols for its authentication and key management (AKM) services. Some of these protocols are defined by other standards organizations, such as the Internet Engineering Task Force (IETF).

5.2 Components of the IEEE 802.11 architecture

5.2.2 Distribution system (DS) concepts

After 5.2.2.1, insert 5.2.2.2:

5.2.2.2 RSNA

An RSNA defines a number of security features in addition to wired equivalent privacy (WEP) and IEEE 802.11 authentication. These features include the following:

— Enhanced authentication mechanisms for STAs

— Key management algorithms

— Cryptographic key establishment

— An enhanced data encapsulation mechanism, called CTR [counter mode] with CBC-MAC [cipher-block chaining (CBC) with message authentication code (MAC)] Protocol (CCMP), and, optionally, Temporal Key Integrity Protocol (TKIP)

An RSNA relies on several components external to the IEEE 802.11 architecture.

The first component is an IEEE 802.1X port access entity (PAE). PAEs are present on all STAs in an RSNA and control the forwarding of data to and from the medium access control (MAC). An access point (AP) always implements an Authenticator PAE and implements the EAP Authenticator role, and a STA always implements a Supplicant PAE and implements the Extensible Authentication Protocol (EAP) peer role. In an independent basic service set (IBSS), each STA implements both an Authenticator PAE and a Supplicant PAE and both the EAP Authenticator and EAP peer roles.

A second component is the Authentication Server (AS). The AS may authenticate the elements of the RSNA itself, i.e., the non-AP STAs; and APs may provide material that the RSNA elements can use to authenticate each other. The AS communicates through the IEEE 802.1X Authenticator with the IEEE 802.1X Supplicant on each STA, enabling the STA to be authenticated to the AS and vice versa. An RSNA depends upon the use of an EAP method that supports mutual authentication of the AS and the STA. In certain applications, the AS may be integrated into the same physical device as the AP, or into a STA in an IBSS.

5.3 Logical service interfaces

Change item g) in the list of architectural services in 5.3 as follows:

g) ~~Privacy~~Confidentiality

5.3.1 Station service (SS)

Change item c) in the list of SSs in 5.3.1 as follows:

c) ~~Privacy~~Confidentiality

5.4 Overview of the services

5.4.2 Services that support the distribution service

5.4.2.2 Association

Insert the following paragraph after the second paragraph of 5.4.2.2:

Within a robust security network (RSN), this is different. In an RSNA, the IEEE 802.1X Port determines when to allow data traffic across an IEEE 802.11 link. A single IEEE 802.1X Port maps to one association, and each association maps to an IEEE 802.1X Port. An IEEE 802.1X Port consists of an IEEE 802.1X Controlled Port and an IEEE 802.1X Uncontrolled Port. The IEEE 802.1X Controlled Port is blocked from passing general data traffic between two STAs until an IEEE 802.1X authentication procedure completes successfully over the IEEE 802.1X Uncontrolled Port. Once the AKM completes successfully, data protection is enabled to prevent unauthorized access, and the IEEE 802.1X Controlled Port unblocks to allow protected data traffic. IEEE 802.1X Supplicants and Authenticators exchange protocol information via the IEEE 802.1X Uncontrolled Port. It is expected that most other protocol exchanges will make use of the IEEE 802.1X Controlled Ports. However, a given protocol may need to bypass the authorization function and make use of the IEEE 802.1X Uncontrolled Port.

NOTE—See IEEE P802.1X-REV for a discussion of Controlled Port and Uncontrolled Port.

5.4.2.3 Reassociation

Insert the following paragraph at the end of 5.4.2.3:

No facilities are provided to move an RSNA during reassociation. Therefore, the old RSNA will be deleted, and a new RSNA will need to be constructed.

Change the title of 5.4.3 as follows:

5.4.3 Access control and confidentiality ~~control~~services

Change the second paragraph of 5.4.3 as follows:

Two services are provided to bring the IEEE 802.11 functionality in line with wired local area network (LAN) assumptions: authentication and ~~privacy~~confidentiality. Authentication is used instead of the wired media physical connection. ~~Privacy~~ Data confidentiality is used to provide the confidential aspects of closed wired media.

Insert the following paragraphs at the end of 5.4.3:

In a wireless LAN (WLAN) that does not support RSNA, two services, authentication and confidentiality, are defined. IEEE 802.11 authentication is used instead of the wired media physical connection. WEP encryption was defined to provide the confidentiality aspects of closed wired media.

An RSNA uses the IEEE 802.1X authentication service along with TKIP and CCMP to provide access control. The IEEE 802.11 station management entity (SME) provides key management via an exchange of IEEE 802.1X EAPOL-Key frames. Confidentiality and data integrity are provided by RSN key management together with the TKIP and CCMP.

5.4.3.1 Authentication

Replace the text of 5.4.3.1 with the following:

IEEE 802.11 authentication operates at the link level between IEEE 802.11 STAs. IEEE 802.11 does not provide either end-to-end (message origin to message destination) or user-to-user authentication.

IEEE 802.11 attempts to control LAN access via the authentication service. IEEE 802.11 authentication is an SS. This service may be used by all STAs to establish their identity to STAs with which they communicate, in both extended service set (ESS) and IBSS networks. If a mutually acceptable level of authentication has not been established between two STAs, an association shall not be established.

IEEE 802.11 defines two authentication methods: Open System authentication and Shared Key authentication. Open System authentication admits any STA to the DS. Shared Key authentication relies on WEP to demonstrate knowledge of a WEP encryption key. The IEEE 802.11 authentication mechanism also allows definition of new authentication methods.

An RSNA also supports authentication based on IEEE 802.1X, or preshared keys (PSKs). IEEE 802.1X authentication utilizes the EAP to authenticate STAs and the AS with one another. This amendment does not specify an EAP method that is mandatory to implement. See 8.4.4 for a description of the IEEE 802.1X authentication and PSK usage within an IEEE 802.11 IBSS.

In an RSNA, IEEE 802.1X Supplicants and Authenticators exchange protocol information via the IEEE 802.1X Uncontrolled Port. The IEEE 802.1X Controlled Port is blocked from passing general data traffic between two STAs until an IEEE 802.1X authentication procedure completes successfully over the IEEE 802.1X Uncontrolled Port.

The Open System authentication algorithm is used in both basic service set (BSS) and IBSS RSNAs, although Open System authentication is optional in an RSNA IBSS. RSNA disallows the use of Shared Key authentication.

Management information base (MIB) functions are provided in Annex D to support the standardized authentication schemes.

A STA may be authenticated with many other STAs at any given instant.

5.4.3.2 Deauthentication

Change the text of 5.4.3.2 as follows:

The deauthentication service is invoked when an existing Open System or Shared Key authentication is to be terminated. Deauthentication is an SS.

In an ESS, because authentication is a prerequisite for association, the act of deauthentication shall cause the station to be disassociated. The deauthentication service may be invoked by either authenticated party (non-AP STA or AP). Deauthentication is not a request; it is a notification. Deauthentication shall not be refused by either party. When an AP sends a deauthentication notice to an associated STA, the association shall also be terminated.

In an RSN ESS, Open System authentication is required. In an RSN ESS, deauthentication results in termination of any association for the deauthenticated station. It also results in the IEEE 802.1X Controlled Port for that STA being disabled and deletes the pairwise transient key security association (PTKSA). The deauthentication notification is provided to IEEE 802.1X via the MAC layer.

In an RSNA, deauthentication also destroys any related PTKSA, group temporal key security association (GTKSA), and STAKey security associations (STAKeySAs) that exist in the STA and closes the associated IEEE 802.1X Controlled Port. If pairwise master key (PMK) caching is not enabled, deauthentication also destroys the pairwise master key security association (PMKSA) from which the deleted PTKSA was derived.

In an RSN IBSS, Open System authentication is optional, but a STA is required to recognize Deauthentication frames. Deauthentication results in the IEEE 802.1X Controlled Port for that STA being disabled and deletes the PTKSA.

Change the title and text of 5.4.3.3 as follows:

5.4.3.3 ~~Privacy~~Confidentiality

In a wired LAN, only those stations physically connected to the wire ~~may hear~~ can send or receive LAN traffic. With a wireless shared medium, this is not the case. Any IEEE 802.11-compliant STA ~~may hear~~ can receive all like-PHY IEEE 802.11 traffic that is within range and can transmit to any other IEEE 802.11 STA within range. Thus the connection of a single wireless link (without ~~privacy~~confidentiality) to an existing wired LAN may seriously degrade the security level of the wired LAN.

To bring the ~~functionality~~ security of the wireless LAN up to the level implicit in wired LAN design, IEEE 802.11 provides the ability to ~~encrypt~~ protect the contents of messages. This functionality is provided by the ~~privacy~~confidentiality service. ~~Privacy~~Confidentiality is an SS.

~~IEEE 802.11 specifies an optional privacy algorithm, WEP, that is designed to satisfy the goal of wired LAN "equivalent" privacy. The algorithm is not designed for ultimate security but rather to be "at least as secure as a wire." See Clause 8 for more details.~~

~~IEEE 802.11 uses the WEP mechanism (see Clause 8) to perform the actual encryption of messages. MIB functions are provided to support WEP.~~

~~Note that privacy may only be invoked for data frames and some Authentication Management frames. All stations initially start "in the clear" in order to set up the authentication and privacy services.~~

IEEE 802.11 provides three cryptographic algorithms to protect data traffic: WEP, TKIP, and CCMP. WEP and TKIP are based on the RC4[8] algorithm, and CCMP is based on the advanced encryption standard (AES). A means is provided for STAs to select the algorithm(s) to be used for a given association.

The default ~~privacy~~confidentiality state for all IEEE 802.11 STAs is "in the clear." If the ~~privacy~~confidentiality service is not invoked, all messages shall be sent ~~unencrypted~~unprotected. If this ~~default~~ policy is ~~not~~ unacceptable to ~~one party or the other, data frames shall not be successfully communicated between the LLC entities.~~ the sender, it shall not send data frames; and if the policy is unacceptable to the receiver, it shall discard any received data frames. ~~Unencrypted~~ Unprotected data frames received at a station configured for mandatory ~~privacy~~confidentiality, as well as protected~~encrypted~~ data frames using a key not available at the receiving station, are discarded without an indication to logical link control (LLC) (or without indication to distribution services in the case of "To DS" frames received at an AP). These frames are acknowledged on the WM [if received without frame check sequence (FCS) error] to avoid wasting WM bandwidth on retries.

After 5.4.3.3, insert 5.4.3.4 through 5.4.3.6:

5.4.3.4 Key management

The enhanced confidentiality, data authentication, and replay protection mechanisms require fresh cryptographic keys. The procedures defined in this amendment provide fresh keys by means of protocols called the 4-Way Handshake and Group Key Handshake.

5.4.3.5 Data origin authenticity

The data origin authenticity mechanism defines a means by which a STA that receives a data frame can determine which STA transmitted the MAC protocol data unit (MPDU). This feature is required in an RSNA to prevent one STA from masquerading as a different STA. This mechanism is provided for STAs that use CCMP or TKIP.

Data origin authenticity is only applicable to unicast data frames. The protocols do not guarantee data origin authenticity for broadcast/multicast data frames, as this cannot be accomplished using symmetric keys and public key methods are too computationally expensive.

5.4.3.6 Replay detection

The replay detection mechanism defines a means by which a STA that receives a data frame from another STA can detect whether the data frame is an unauthorized retransmission. This mechanism is provided for STAs that use CCMP or TKIP.

5.6 Differences between ESS and IBSS LANs

Insert the following paragraph at the end of 5.6:

In an IBSS, each STA must enforce its own security policy. In an ESS, an AP can enforce a uniform security policy across all STAs.

[8]Details of the RC4 algorithm are available from RSA Security, Inc. Contact RSA for algorithm details and the uniform RC4 license terms that RSA offers to anyone wishing to use RC4 for the purpose of implementing the IEEE 802.11 WEP option. If necessary, contact the IEEE Standards Department Intellectual Property Rights Administrator for details on how to communicate with RSA.

5.7 Message information contents that support the services

Change the title and text of 5.7.5 as follows:

5.7.5 ~~Privacy~~Confidentiality

For a STA to invoke a ~~the WEP privacy~~ confidentiality algorithm (as controlled by MLME-SETKEYS, see Clause 10, or the related MIB attributes, see Clause 11), the ~~privacy~~ confidentiality service ~~causes MPDU encryption~~ selects the confidentiality algorithm and sets the ~~WEP frame header~~Protected Frame bit appropriately (see Clause 7).

5.7.6 Authentication

Change the first paragraph in 5.7.6 as follows:

For a STA to authenticate with another STA using either Open System or Shared Key authentication, the authentication service causes one or more authentication management frames to be exchanged. The exact sequence of frames and their content are dependent on the authentication scheme invoked. For ~~all~~both of these authentication schemes, the authentication algorithm is identified within the management frame body.

5.7.7 Deauthentication

Change the first paragraph in 5.7.7 as follows:

For a STA to invalidate an active authentication that was established using Open System or Shared Key authentication, the following message is sent:

5.8 Reference model

Replace Figure 11 with the following:

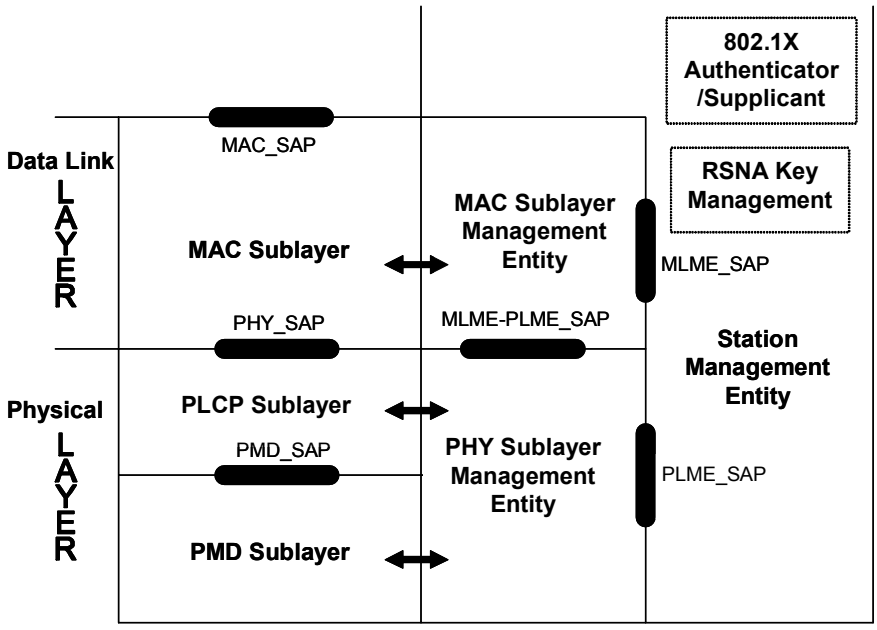


Figure 11—Portion of the ISO/IEC basic reference model covered in this standard

Insert the following paragraph at the end of 5.8:

There is an interface between the IEEE 802.1X Supplicant/Authenticator and the SME shown in Figure 11. This interface is described in IEEE P802.1X-REV.

After 5.8, insert 5.9 through 5.9.5 and renumber the figures as necessary:

5.9 IEEE 802.11 and IEEE 802.1X

An RSNA relies on IEEE 802.1X to provide authentication services and uses the IEEE 802.11 key management scheme defined in 8.5. The IEEE 802.1X access control mechanisms apply to the association between a STA and an AP and to the relationship between the IBSS STA and STA peer. The AP's SME performs the Authenticator and, optionally, the Supplicant and AS roles. In an ESS, a non-AP STA's SME performs the Supplicant role. In an IBSS, a STA's SME takes on both the Supplicant and Authenticator roles and may take on the AS role.

5.9.1 IEEE 802.11 usage of IEEE 802.1X

IEEE 802.11 depends upon IEEE 802.1X to control the flow of MAC service data units (MSDUs) between the DS and STAs by use of the IEEE 802.1X Controlled/Uncontrolled Port model. IEEE 802.1X authentication frames are transmitted in IEEE 802.11 data frames and passed via the IEEE 802.1X Uncontrolled Port. The IEEE 802.1X Controlled Port is blocked from passing general data traffic between two STAs until an IEEE 802.1X authentication procedure completes successfully over the IEEE 802.1X Uncontrolled Port. It is the responsibility of both the Supplicant and the Authenticator to implement port blocking. Each association between a pair of STAs creates a unique pair of IEEE 802.1X Ports, and authentication takes place relative to those ports alone.

IEEE 802.11 depends upon IEEE 802.1X and the 4-Way Handshake and Group Key Handshake, described in Clause 8, to establish and change cryptographic keys. Keys are established after authentication has completed. Keys may change for a variety of reasons, including expiration of an IEEE 802.1X authentication timer, key compromise, danger of compromise, or policy.

5.9.2 Infrastructure functional model overview

This subclause summarizes the system setup and operation of an RSN, in two cases: when an IEEE 802.1X AS is used and when a PSK is used. For an ESS, the AP includes an Authenticator, and each associated STA includes a Supplicant.

5.9.2.1 AKM operations with AS

The following AKM operations are carried out when an IEEE 802.1X AS is used:

a) Prior to any use of IEEE 802.1X, IEEE 802.11 assumes that the Authenticator and AS have established a secure channel. The security of the channel between the Authenticator and the AS is outside the scope of this amendment.

   Authentication credentials must be distributed to the Supplicant and AS prior to association.

b) A STA discovers the AP's security policy through passively monitoring Beacon frames or through active probing (shown in Figure 11a). If IEEE 802.1X authentication is used, the EAP authentication process starts when the AP's Authenticator sends the EAP-Request (shown in Figure 11b) or the STA's Supplicant sends the EAPOL-Start message. EAP authentication frames pass between the Supplicant and AS via the Authenticator and Supplicant's Uncontrolled Ports. This is shown in Figure 11b.

c) The Supplicant and AS authenticate each other and generate a PMK. The PMK is sent from the AS to the Authenticator over the secure channel. See Figure 11b.

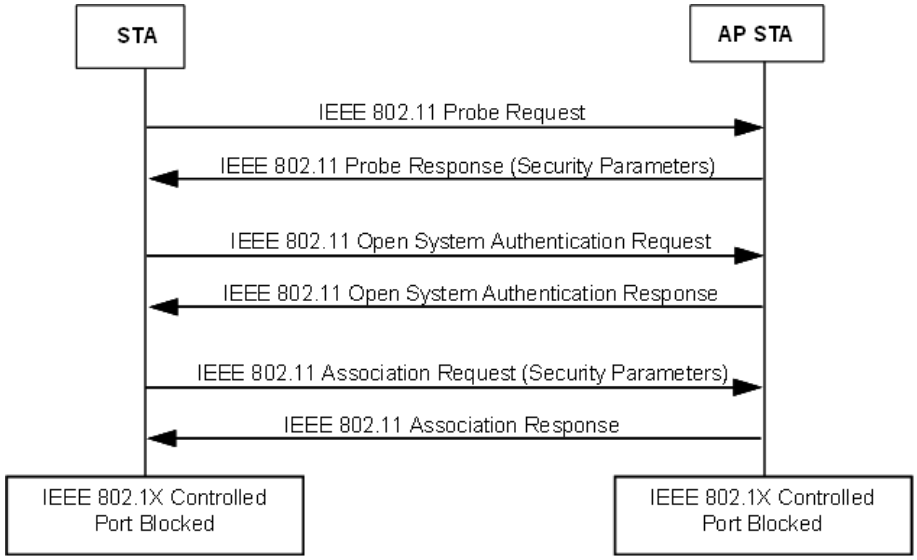


Figure 11a—Establishing the IEEE 802.11 association

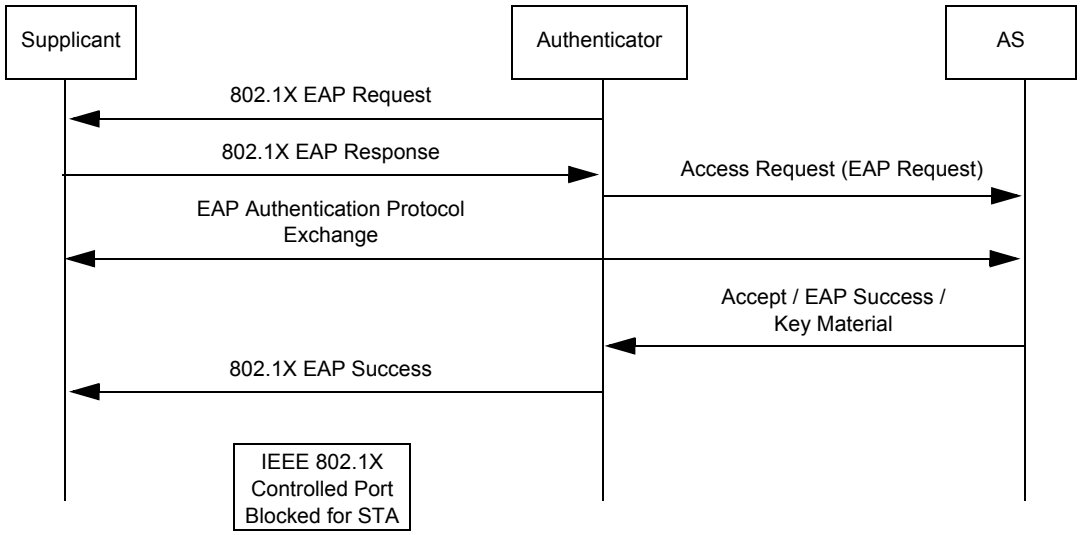


Figure 11b—IEEE 802.1X EAP authentication

A 4-Way Handshake utilizing EAPOL-Key frames is initiated by the Authenticator to do the following:

— Confirm that a live peer holds the PMK.

— Confirm that the PMK is current.

— Derive a fresh pairwise transient key (PTK) from the PMK.

— Install the pairwise encryption and integrity keys into IEEE 802.11.
— Transport the group temporal key (GTK) and GTK sequence number from Authenticator to Supplicant and install the GTK and GTK sequence number in the STA and, if not already installed, in the AP.
— Confirm the cipher suite selection.

Upon successful completion of the 4-Way Handshake, the Authenticator and Supplicant have authenticated each other; and the IEEE 802.1X Controlled Ports are unblocked to permit general data traffic. See Figure 11c.

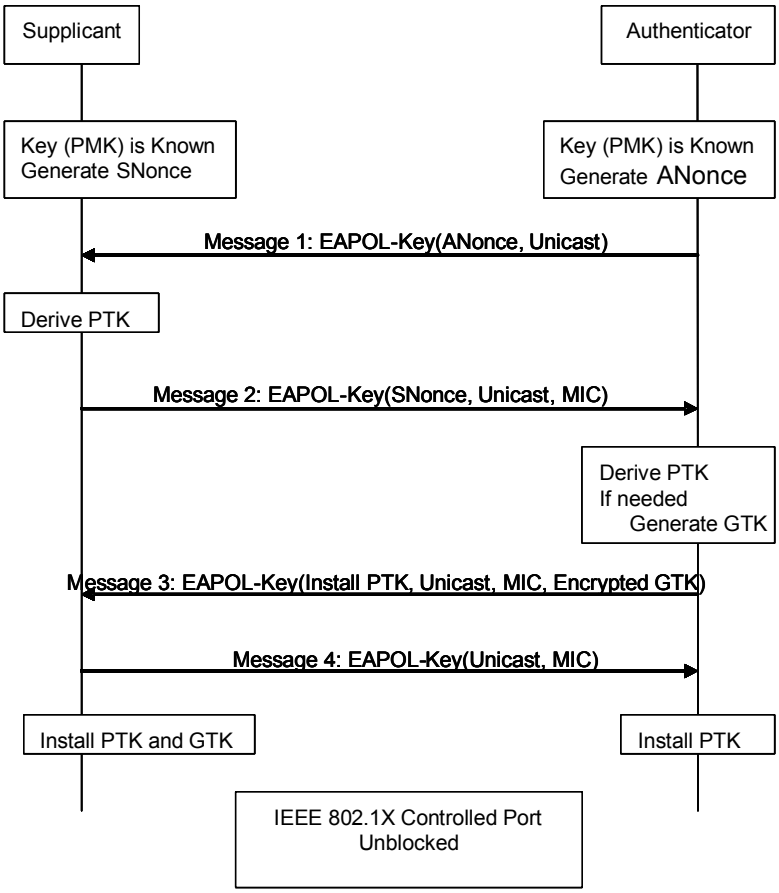


Figure 11c—Establishing pairwise and group keys

If the Authenticator later changes the GTK, it sends the new GTK and GTK sequence number to the Supplicant using the Group Key Handshake to allow the Supplicant to continue to receive broadcast/multicast messages and, optionally, to transmit and receive unicast frames. EAPOL-Key frames are used to carry out this exchange. See Figure 11d.

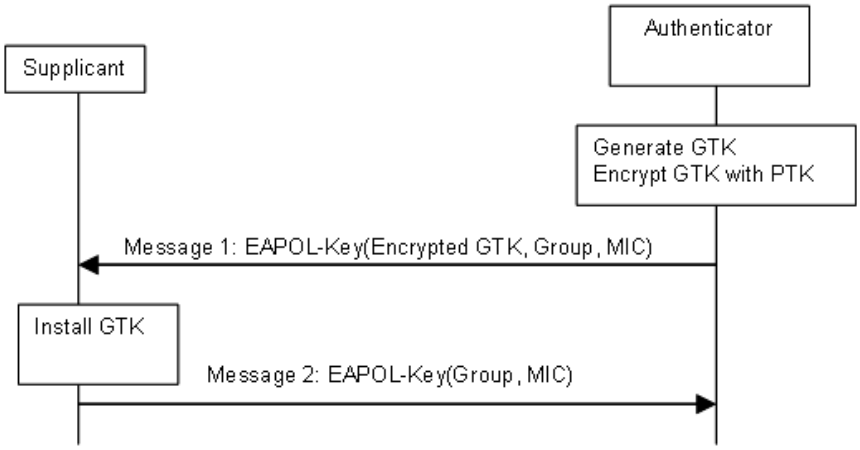


Figure 11d—Delivery of subsequent group keys

5.9.2.2 Operations with PSK

The following AKM operations are carried out when the PMK is a PSK:

— A STA discovers the AP's security policy through passively monitoring Beacon frames or through active probing (shown in Figure 11a). A STA associates with an AP and negotiates a security policy. The PMK is the PSK.

— The 4-Way Handshake using EAPOL-Key frames is used just as with IEEE 802.1X authentication, when an AS is present. See Figure 11c.

— The GTK and GTK sequence number are sent from the Authenticator to the Supplicant just as in the AS case. See Figure 11c and Figure 11d.

5.9.3 IBSS functional model description

This subclause summarizes the system setup and operation of an RSNA in an IBSS. An IBSS RSNA is specified in 8.4.7.

5.9.3.1 Key usage

In an IBSS, the unicast data frames between two STAs are protected with a pairwise key. The key is part of the PTK, which is derived during a 4-Way Handshake.

In an IBSS, the broadcast/multicast data frames are protected by a key, e.g., named B1, that is generated by the STA transmitting the broadcast/multicast frame. To allow other STAs to decrypt broadcast/multicast frames, B1 must be sent to all the other STAs in the IBSS. B1 is sent in an EAPOL-Key frame, encrypted under the EAPOL-Key encryption key (KEK) portion of the PTK, and protected from modification by the EAPOL-Key confirmation key (KCK) portion of the PTK.

In an IBSS, a STA's SME responds to Deauthentication frames from a STA by deleting the PTKSA associated with that STA.

5.9.3.2 Sample IBSS 4-Way Handshakes

In this example (see Figure 11e), there are three STAs: S1, S2, S3. The broadcast/multicast frames sent by S1 are protected by B1; similarly B2 for S2, and B3 for S3.

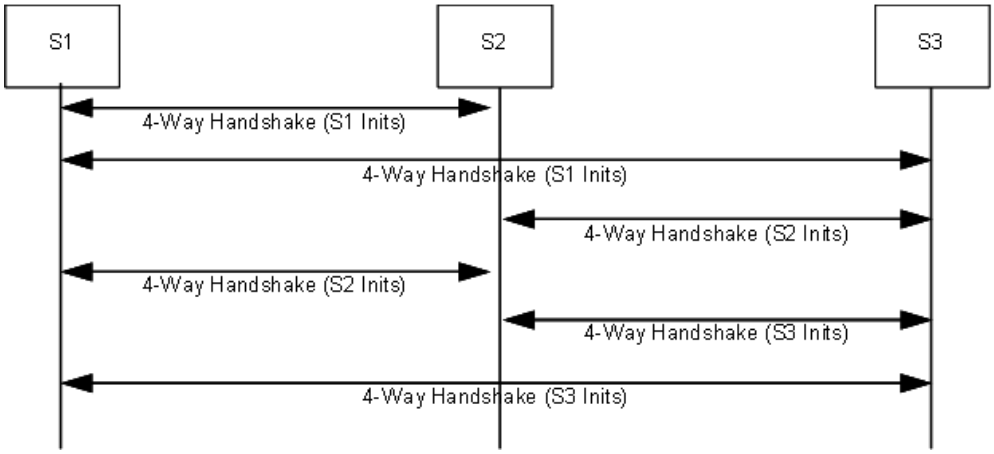


Figure 11e—Sample 4-Way Handshakes in an IBSS

For stations S2 and S3 to decrypt broadcast/multicast frames from S1, B1 must be sent to S2 and S3. This is done using the 4-Way Handshake initially and using the Group Key Handshake for GTK updates.

The 4-Way Handshake from S1 to S2 allows S1 to send broadcast/multicast frames to S2, but does not allow S2 to send broadcast/multicast frames to S1 because S2 has a different transmit GTK. Therefore, S2 needs to initiate a 4-Way Handshake to S1 to allow S1 to decrypt S2's broadcast/multicast frames. Similarly, S2 also needs to initiate a 4-Way Handshake to S3 to enable S3 to receive broadcast/multicast messages from S2.

In a similar manner S3 needs to complete the 4-Way Handshake with S1 and S2 to deliver B3 to S1 and S2.

In this example, there are six 4-Way Handshakes. In general, N STA Supplicants require $N(N–1)$ 4-Way Handshakes.

NOTE—In principle the KCK and KEK from a single 4-Way Handshake can be used for the Group Key Handshake in both directions, but using two 4-Way Handshakes means the Authenticator key state machine does not need to be different between IBSS and ESS.[9]

The Group Key Handshake can be used to send the GTKs to the correct STAs. The 4-Way Handshake is used to derive the pairwise key and to send the initial GTK. Because in an IBSS there are two 4-Way Handshakes between any two STA Supplicants and Authenticators, the pairwise key used between any two STAs is from the 4-Way Handshake initiated by the STA Authenticator with the higher MAC address (see 8.5.1 for the notion of address comparison). The KCK and KEK used for a Group Key Handshake are the KCK and KEK derived by the 4-Way Handshake initiated by the same Authenticator that is initiating the Group Key Handshake.

In an IBSS, a secure link exists between two stations when both 4-Way Handshakes have completed successfully. The Supplicant and Authenticator 4-Way Handshake state machines interact so the IEEE 802.1X variable portValid is not set until both 4-Way Handshakes complete.

If a fourth STA comes within range and its SME decides to initiate a security association with the three peers, its Authenticator initiates 4-Way Handshakes with each of the other three STA Supplicants. Similarly, the original three STA Authenticators in the IBSS need to initiate 4-Way Handshakes to the fourth STA

[9]Notes in text, tables, and figures are given for information only and do not contain requirements needed to implement this amendment.

Supplicant. A STA learns that a peer STA is RSNA-enabled and the peer's security policy (e.g., whether the Authentication and Key Management Protocol (AKMP) is PSK or IEEE 802.1X authentication) from the Beacon or Probe Response frame. The initiation may start for a number of reasons:

a) The fourth STA receives a Beacon or Probe Response frame from a MAC address with which it has not completed a 4-Way Handshake.

b) A STA's SME receives a MLME-PROTECTEDFRAMEDROPPED.indication primitive from a MAC address with which it has not completed a 4-Way Handshake. This could be a multicast/broadcast data frame transmitted by any of the STAs. If the SME wants to set up a security association to the peer STA, but does not know the security policy of the peer, it should send a Probe Request frame to the peer STA to find its security policy before setting up a security association to the peer STA.

c) A STA's SME receives Message 1 of the 4-Way Handshake sent to a STA because the initiator received a broadcast data frame, Beacon frame, or Probe Response frame from that STA. If a STA received a 4-Way Handshake, wants to set up a security association to the peer STA, but does not know the security policy of the peer, it should send a Probe Request frame to the peer STA to find its security policy before setting up a security association to the peer STA.

5.9.3.3 IBSS IEEE 802.1X Example

When IEEE 802.1X authentication is used, each STA will need to include an IEEE 802.1X Authenticator and AS. A STA learns that a peer STA is RSNA-enabled and the peer's security policy (e.g., whether the AKMP is PSK or IEEE 802.1X authentication) from the Beacon or Probe Response frame.

Each STA's Supplicant will send an EAPOL-Start message to every other station to which it wants to authenticate, and each STA's Authenticator will respond with the identity of the credential it wants to use.

The EAPOL-Start and EAP-Request/Identity messages are initiated when a protected data frame (indicated via a MLME-PROTECTEDFRAMEDROPPED.indication primitive), an IEEE 802.1X message, Beacon frame, or Probe Response frame is received from a MAC address with which the STA has not completed IEEE 802.1X authentication. If the SME wants to set up a security association to the peer STA, but does not know the security policy of the peer, it should send a Probe Request frame to the peer STA to find its security policy before setting up a security association to the peer STA.

Although Figure 11f shows the two IEEE 802.1X exchanges serialized, they may occur interleaved.

5.9.4 Authenticator-to-AS protocol

The Authenticator-to-AS authentication definition is out of the scope of this amendment, but, to provide security assurances, the protocol must support the following functions:

a) Mutual authentication between the Authenticator and AS

b) A channel for the Supplicant/AS authentication

c) The ability to pass the generated key from the AS to the Authenticator in a manner that provides authentication of the key source, ensures integrity of the key transfer, and preserves confidentiality of the key from all other parties

Suitable protocols include, but are not limited to, remote authentication dial-in user service (RADIUS) (IETF RFC 2865 [B14]) and Diameter (IETF RFC 3588 [B15]).

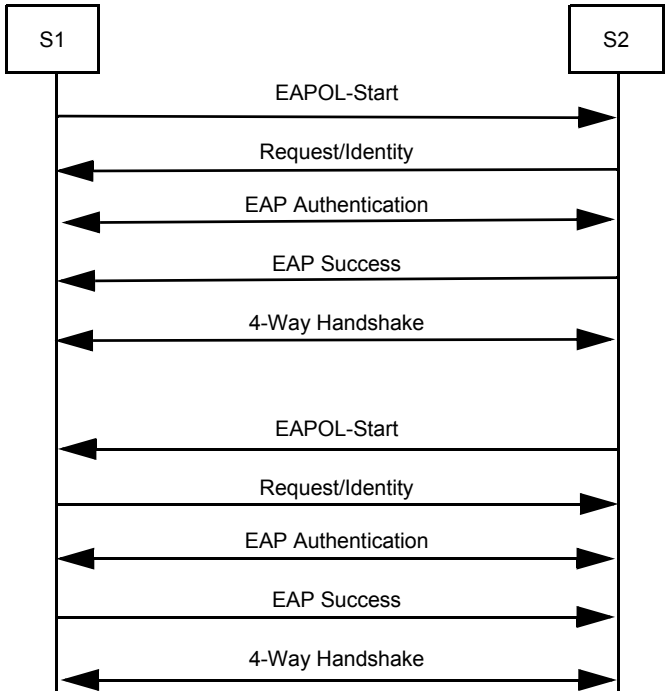


Figure 11f—Example using IEEE 802.1X authentication

5.9.5 PMKSA caching

The Authenticator and Supplicant may cache PMKSAs, which include the IEEE 802.1X state. A PMKSA can be deleted from the cache for any reason and at any time.

The STA may supply a list of PMK or PSK key identifiers in the (Re)Association Request frame. Each key identifier names a PMKSA; the PMKSA may contain a single PMK. The Authenticator specifies the selected PMK or PSK key identifier in Message 1 of the 4-Way Handshake. The selection of the key identifiers to be included within the (Re)Association Request frame and Message 1 of the 4-Way Handshake is out of the scope of this amendment.

End of changes to Clause 5.

6. MAC service definition

6.1 Overview of MAC services

6.1.2 Security services

Change the text of 6.1.2 as follows:

Security services in IEEE 802.11 are provided by the authentication service and the WEP, TKIP, and CCMP mechanisms. The scope of the security services provided is limited to station-to-station data exchange. The ~~privacy~~ confidentiality service offered by an IEEE 802.11 WEP, TKIP, and CCMP implementation is the encryption of the MSDU. For the purposes of this standard, WEP, TKIP, and CCMP are ~~is~~ viewed as ~~a~~

logical services located within the MAC sublayer as shown in the reference model, Figure 11. Actual implementations of the WEP, TKIP, and CCMP services are transparent to the LLC and other layers above the MAC sublayer.

The security services provided by WEP, TKIP, and CCMP in IEEE 802.11 are as follows:

a) Confidentiality;
b) Authentication; and
c) Access control in conjunction with layer management.

During the authentication exchange, both parties ~~A and B~~ exchange authentication information as described in Clause 8.

The MAC sublayer security services provided by WEP, TKIP, and CCMP rely on information from non-layer 2 management or system entities. Management entities communicate information to WEP through a set of MIB attributes. Management entities communicate information to TKIP and CCMP through a set of MAC sublayer management entity (MLME) interfaces and MIB attributes; in particular, the decision tree for TKIP and CCMP defined in 8.7 is driven by MIB attributes.

After 6.1.3, insert 6.1.4 and renumber the figures as necessary:

6.1.4 MAC data service architecture

The MAC data plane architecture (i.e., processes that involve transport of all or part of an MSDU) is shown in Figure 11g. During transmission, an MSDU goes through some or all of the following processes: frame delivery deferral during power save mode, sequence number assignment, fragmentation, encryption, integrity protection, and frame formatting. IEEE 802.1X may block the MSDU at the Controlled Port.

During reception, a received data frame goes through processes of MPDU header + cyclic redundancy code (CRC) validation, duplicate removal, decryption, defragmentation, integrity checking, and replay detection. The IEEE 802.1X Controlled/Uncontrolled Ports discard the MSDU if the Controlled Port is not enabled or if the MSDU does not represent an IEEE 802.1X frame. TKIP and CCMP MPDU frame order enforcement occurs after decryption, but prior to MSDU defragmentation; therefore, defragmentation will fail if MPDUs arrive out of order.

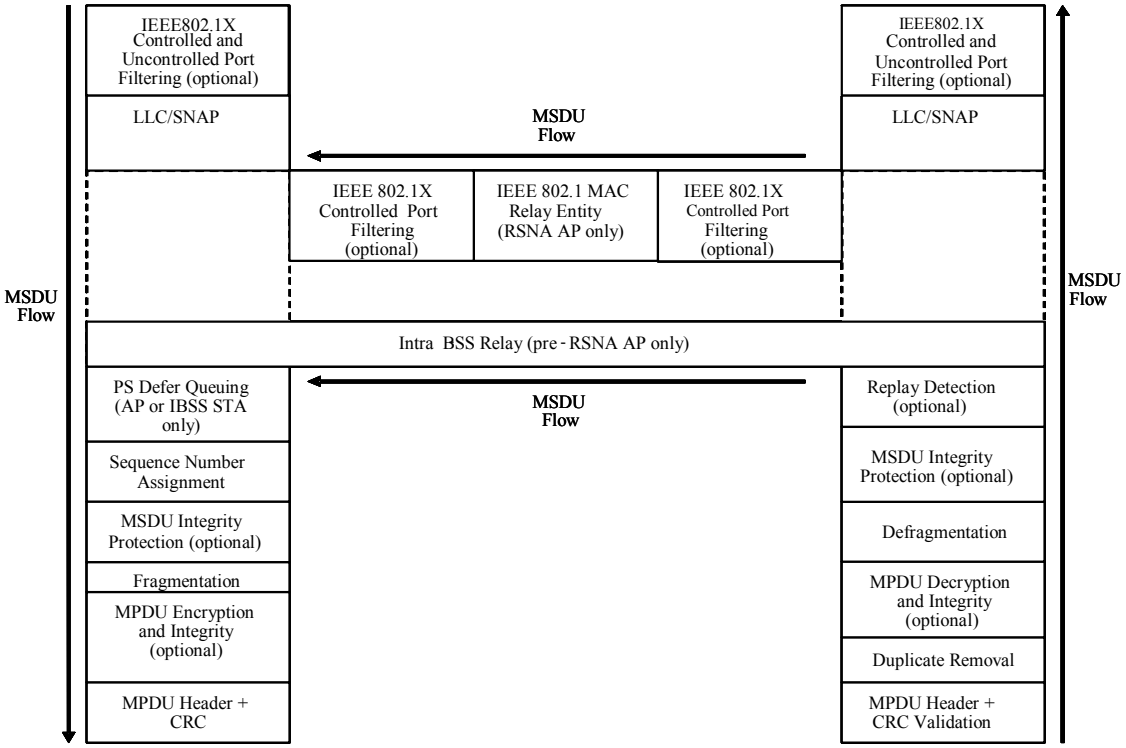


Figure 11g—MAC data plane architecture

End of changes to Clause 6.

7. Frame formats

7.1 MAC frame formats

7.1.3 Frame fields

7.1.3.1 Frame Control field

Change the text of 7.1.3.1 as follows:

The Frame Control field consists of the following subfields: Protocol Version, Type, Subtype, To DS, From DS, More Fragments, Retry, Power Management, More Data, ~~Wired Equivalent Privacy (WEP)~~ Protected Frame, and Order. The format of the Frame Control field is illustrated in Figure 13.

Change Figure 13 as shown:

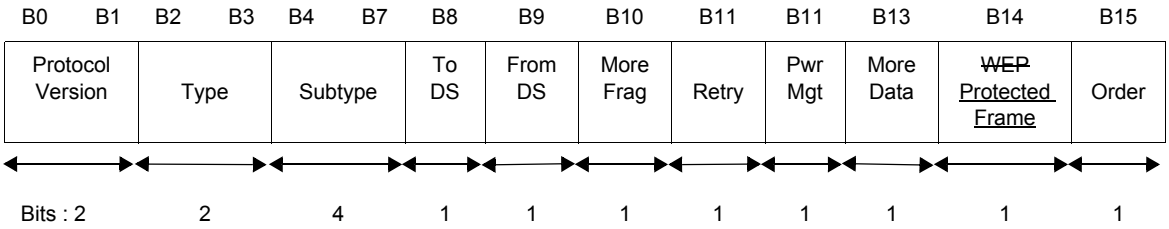


Figure 13—Frame Control field

Change the title and text of 7.1.3.1.9 as follows:

7.1.3.1.9 ~~WEP~~Protected Frame field

The ~~WEP~~ Protected Frame field is 1 bit in length. ~~It~~ The Protected Frame field is set to 1 if the Frame Body field contains information that has been processed by ~~the WEP~~ a cryptographic encapsulation algorithm. The ~~WEP~~ Protected Frame field is set to 1 only within data frames ~~of type Data~~ and within management frames of ~~type Management,~~ subtype Authentication. The ~~WEP~~ Protected Frame field is set to 0 in all other frames. When the ~~WEP bit~~ Protected Frame field is set to 1~~, the Frame Body field is expanded as defined in 8.2.5~~ in a data frame, the Frame Body field is protected utilizing the cryptographic encapsulation algorithm and expanded as defined in Clause 8. Only WEP is allowed as the cryptographic encapsulation algorithm for management frames of subtype Authentication.

7.2 Format of individual frame types

7.2.2 Data frames

Change the first paragraph after the lettered list in 7.2.2 as follows:

The frame body consists of the MSDU or a fragment thereof, and a ~~WEP IV and ICV~~ security header and trailer (if and only if the ~~WEP~~ Protected Frame subfield in the Frame Control field is set to 1). The frame body is null (0 octets in length) in data frames of subtypes Null Function (no data), CF-Ack (no data), CF-Poll (no data), and CF-Ack+CF-Poll (no data).

7.2.3 Management frames

7.2.3.1 Beacon frame format

Insert the order 21 information field in Table 5:

Table 5—Beacon frame body

21	RSN	The RSN information element is only present within Beacon frames generated by STAs that have `dot11RSNAEnabled` set to TRUE.

7.2.3.4 Association Request frame format

Insert the order 8 information field in Table 7:

Table 7—Association Request frame body

8	RSN	The RSN information element is only present within Association Request frames generated by STAs that have `dot11RSNAEnabled` set to TRUE.

7.2.3.6 Reassociation Request frame format

Insert the order 9 information field to Table 9:

Table 9—Reassociation Request frame body

9	RSN	The RSN information element is only present within Reassociation Request frames generated by STAs that have `dot11RSNAEnabled` set to TRUE.

7.2.3.9 Probe Response frame format

Insert the order 21 and 22–n information fields in Table 12:

Table 12—Probe Response frame body

21	RSN	The RSN information element is only present within Probe Response frames generated by STAs that have `dot11RSNAEnabled` set to TRUE.
22–*n*	Requested information elements	Elements requested by the Request information element of the Probe Request frame.

7.2.3.10 Authentication frame format

Insert the following text after the first sentence of 7.2.3.10:

Only Authentication frames with the authentication algorithm set to Open System authentication may be used within an RSNA. RSNA STAs shall not associate if shared authentication was invoked prior to RSN association.

7.3 Management frame body components

7.3.1 Fixed fields

7.3.1.4 Capability Information field

Change the sixth and seventh paragraphs in 7.3.1.4 as follows:

APs set the Privacy subfield to 1 within transmitted Beacon, Probe Response, Association Response, and Reassociation Response management frames if ~~WEP encryption~~ data confidentiality is required for all data type frames exchanged within the BSS. If ~~WEP encryption~~ data confidentiality is not required, ~~APs set~~ the Privacy subfield is set to 0~~ within these management frames~~.

In an RSNA, non-AP STAs in an ESS set the Privacy subfield to 0 within transmitted Association and Reassociation Request management frames. APs ignore the Privacy subfield within received Association and Reassociation Request management frames.

STAs within an IBSS set the Privacy subfield to 1 in transmitted Beacon or Probe Response management frames if ~~WEP encryption~~ data confidentiality is required for all data type frames exchanged within the IBSS. If ~~WEP encryption~~ data confidentiality is not required, STAs in an IBSS set the Privacy subfield ~~is set~~ to 0 within these management frames.

STAs that include the RSN information element in Beacon and Probe Response frames shall set the Privacy subfield to 1 in any frame that includes the RSN information element.

7.3.1.7 Reason Code field

Insert reason codes 12 through 24 and change the final Reserved reason code row in Table 18 as follows:

Table 18—Reason codes

Reason code	Meaning
12	Reserved
13	Invalid information element
14	MIC failure
15	4-Way Handshake timeout
16	Group Key Handshake timeout
17	Information element in 4-Way Handshake different from (Re)Association Request/Probe Response/Beacon frame
18	Invalid group cipher
19	Invalid pairwise cipher
20	Invalid AKMP
21	Unsupported RSN information element version
22	Invalid RSN information element capabilities
23	IEEE 802.1X authentication failed

Table 18—Reason codes *(continued)*

Reason code	Meaning
24	Cipher suite rejected per security policy
25–65 535	Reserved

7.3.1.9 Status Code field

Insert reason codes 27 through 46 and change the final Reserved reason code row in Table 19 as follows:

Table 19—Status codes

Status code	Meaning
27–39	Reserved
40	Invalid information element
41	Invalid group cipher
42	Invalid pairwise cipher
43	Invalid AKMP
44	Unsupported RSN information element version
45	Invalid RSN information element capabilities
46	Cipher suite rejected per security policy
47–65 535	Reserved

7.3.2 Information elements

Replace element identifiers (IDs) 43 through 49 in Table 20 as follows:

Table 20—Element IDs

Information element	Element ID
Reserved	43–47
RSN	48
Reserved	49

After 7.3.2.24, insert 7.3.2.25 through 7.3.2.25.5 and renumber tables and figures as necessary:

7.3.2.25 RSN information element

The RSN information element contains authentication and pairwise cipher suite selectors, a single group cipher suite selector, an RSN Capabilities field, the PMK identifier (PMKID) count, and PMKID list. See Figure 46ta. All STAs implementing RSNA shall support this element. The size of the RSN information element is limited by the size of an information element, which is 255 octets. Therefore, the number of pairwise cipher suites, AKM suites, and PMKIDs is limited.

Element ID 1 octet	Length 1 octet	Version 2 octets	Group Cipher Suite 4 octets	Pairwise Cipher Suite 2 octets	Pairwise Cipher Suite List 4-*m* octets	AKM Suite Count 2 octets	AKM Suite List 4-*n* octets	RSN Capabilities 2 octets	PMKID Count 2 octets	PMKID List 16-*s* octets

Figure 46ta—RSN information element format

In Figure 46ta, *m* denotes the pairwise cipher suite count, *n* the AKM suite count, and *s* is the PMKID count.

All fields use the bit convention from 7.1.1. The RSN information element shall contain up to and including the Version field. All fields after the Version field are optional. If any optional field is absent, then none of the subsequent fields shall be included.

Element ID shall be 48 decimal (30 hex).

Length gives the number of octets in the information field (field(s) following the Element ID and Length fields) of the information element.

The Version field indicates the version number of the RSNA protocol. The range of Version field values a STA supports shall be contiguous. Values 0 and 2 or higher of the Version field are reserved. RSN Version 1 is defined in this amendment.

NOTE—The following represent sample information elements:

802.1X authentication, CCMP pairwise and group cipher suites (WEP-40, WEP-104, and TKIP not allowed).:

30, // information element id, 48 expressed as Hex value
14, // length in octets, 20 expressed as Hex value
01 00, // Version 1
00 0F AC 04, // CCMP as group cipher suite
01 00, // pairwise cipher suite count
00 0F AC 04, // CCMP as pairwise cipher suite
01 00, // authentication count
00 0F AC 01 // 802.1X authentication
00 00 // No capabilities

802.1X authentication, CCMP pairwise and group cipher suites (WEP-40, WEP-104 and TKIP not allowed), preauthentication supported:

30, // information element id, 48 expressed as Hex value
14, // length in octets, 20 expressed as Hex value
01 00, // Version 1
00 0F AC 04, // CCMP as group cipher suite
01 00, // pairwise cipher suite count
00 0F AC 04, // CCMP as pairwise cipher suite
01 00, // authentication count
00 0F AC 01 // 802.1X authentication
01 00 // Preauthentication capabilities

802.1X authentication, Use GTK for pairwise cipher suite, WEP-40 group cipher suites, optional RSN Capabilities field omitted:

30, // information element id, 48 expressed as Hex value
12, // length in octets, 18 expressed as Hex value
01 00, // Version 1
00 0F AC 01, // WEP-40 as group cipher suite
01 00, // pairwise cipher suite count
00 0F AC 00, // Use group key as pairwise cipher suite
01 00, // authentication count
00 0F AC 01 // 802.1X authentication

802.1X authentication, Use CCMP for pairwise cipher suite, CCMP group cipher suites, preauthentication and a PMKID.

30, // information element id, 48 expressed as Hex value
26 // length in octets, 38 expressed as Hex value
01 00, // Version 1
00 0F AC 04, // CCMP as group cipher suite
01 00, // pairwise cipher suite count
00 0F AC 04, // CCMP as pairwise cipher suite
01 00, // authentication count
00 0F AC 01 // 802.1X authentication
01 00 // Preauthentication capabilities
01 00 // PMKID Count
01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 // PMKID

7.3.2.25.1 Cipher suites

The Group Cipher Suite field contains the cipher suite selector used by the BSS to protect broadcast/multicast traffic.

The Pairwise Cipher Suite Count field indicates the number of pairwise cipher suite selectors that are contained in the Pairwise Cipher Suite List field.

The Pairwise Cipher Suite List field contains a series of cipher suite selectors that indicate the pairwise cipher suites contained in the RSN information element.

A suite selector has the format shown in Figure 46tb.

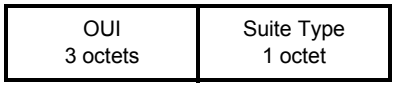


Figure 46tb—Suite selector format

The order of the organizationally unique indentifier (OUI) field shall follow the ordering convention for MAC addresses from 7.1.1.

Table 20da provides the cipher suite selectors defined by this amendment.

Table 20da—Cipher suite selectors

OUI	Suite type	Meaning
00-0F-AC	0	Use group cipher suite
00-0F-AC	1	WEP-40

Table 20da—Cipher suite selectors *(continued)*

OUI	Suite type	Meaning
00-0F-AC	2	TKIP
00-0F-AC	3	Reserved
00-0F-AC	4	CCMP – default in an RSNA
00-0F-AC	5	WEP-104
00-0F-AC	6–255	Reserved
Vendor OUI	Other	Vendor specific
Other	Any	Reserved

The cipher suite selector 00-0F-AC:4 (CCMP) shall be the default cipher suite value.

The cipher suite selectors 00-0F-AC:1 (WEP-40) and 00-0F-AC:5 (WEP-104) are only valid as a group cipher suite in a transition security network (TSN) to allow pre-RSNA devices to join the BSS.

Use of CCMP as the group cipher suite with TKIP as the pairwise cipher suite shall not be supported.

NOTE—If the STAs can support CCMP, then there is no need for a weaker data confidentiality protocol.

The cipher suite selector 00-0F-AC:0 (Use group cipher suite) is only valid as the pairwise cipher suite. An AP may specify the selector 00-0F-AC:0 (Use group cipher suite) for a pairwise cipher suite if it does not support any pairwise cipher suites. If an AP specifies 00-0F-AC:0 (Use group cipher suite) as the pairwise cipher selection, this shall be the only pairwise cipher selection the AP advertises.

If CCMP is enabled, then the AP supports pairwise keys, and thus the suite selector 00-0F-AC:0 (Use group cipher suite) shall not be a valid option.

Table 20db indicates the circumstances under which each cipher suite shall be used.

Table 20db—Cipher suite usage

Cipher suite selector	GTK	PTK
Use group key	No	Yes
WEP-40	Yes	No
WEP-104	Yes	No
TKIP	Yes	Yes
CCMP	Yes	Yes

7.3.2.25.2 AKM suites

The AKM Suite Count field indicates the number of AKM suite selectors that are contained in the AKM Suite List field.

The AKM Suite List field contains a series of AKM suite selectors contained in the RSN information element. In an IBSS, only a single AKM suite selector may be specified because STAs in an IBSS must use the same AKM suite and because there is no mechanism to negotiate the AKMP in an IBSS (see 8.4.4).

Each AKM suite selector specifies an AKMP. Table 20dc gives the AKM suite selectors defined by this amendment.

Table 20dc—AKM suite selectors

OUI	Suite type	Meaning	
		Authentication type	Key management type
00-0F-AC	0	Reserved	Reserved
00-0F-AC	1	Authentication negotiated over IEEE 802.1X or using PMKSA caching as defined in 8.4.6.2 – RSNA default	RSNA key management as defined in 8.5 or using PMKSA caching as defined in 8.4.6.2 – RSNA default
00-0F-AC	2	PSK	RSNA key management as defined in 8.5, using PSK
00-0F-AC	3–255	Reserved	Reserved
Vendor OUI	Any	Vendor specific	Vendor specific
Other	Any	Reserved	Reserved

The AKM suite selector value 00-0F-AC:1 (Authentication negotiated over IEEE 802.1X) with (RSNA key management as defined in 8.5 or using PMKSA caching as defined in 8.4.6.2) shall be the assumed default when the AKM suite selector field is not supplied.

NOTE—The selector value 00-0F-AC:1 specifies only that IEEE 802.1X is used as the authentication transport. IEEE 802.1X selects the authentication mechanism.

The AKM suite selector value 00-0F-AC:2 (PSK) is used when a PSK is used with RSNA key management.

NOTE—Selector values 00-0F-AC:1 and 00-0F-AC:2 can simultaneously be enabled by an Authenticator.

7.3.2.25.3 RSN capabilities

The RSN Capabilities field indicates requested or advertised capabilities. The value of each of the RSN Capabilities fields shall be taken to be 0 if the RSN Capabilities field is not available in the RSN information element.

The length of the RSN Capabilities field is 2 octets. The format of the RSN Capabilities field is as illustrated in Figure 46tc and described after the figure.

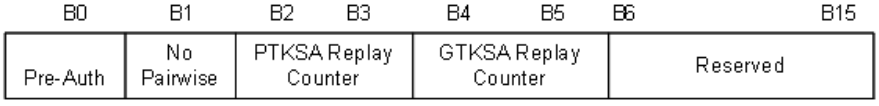


Figure 46tc—RSN Capabilities field format

— Bit 0: Pre-Authentication. An AP sets the Pre-Authentication subfield of the RSN Capabilities field to 1 to signal it supports preauthentication (see 8.4.6.1) and sets the subfield to 0 when it does not support preauthentication. A non-AP STA sets the Pre-Authentication subfield to 0.

— Bit 1: No Pairwise. If a STA can support WEP default key 0 simultaneously with a pairwise key (see 8.5.1), then the STA sets the No Pairwise subfield of the RSN Capabilities field to 0.

  If a STA cannot support WEP default key 0 simultaneously with a pairwise key (see 8.5.1), then the STA sets the No Pairwise subfield of the RSN Capabilities field to 1.

  The No Pairwise subfield describes a capability of a non-AP STA. STAs in an IBSS and APs set the No Pairwise subfield to 0.

  The No Pairwise subfield shall be set only in a TSN and when the pairwise cipher suite selected by the STA is TKIP.

— Bits 2–3: PTKSA Replay Counter. A STA sets the PTKSA Replay Counter subfield of the RSN Capabilities field to the value contained in `dot11RSNAConfigNumberofPTKSAReplay-Counters`. The least significant bit (LSB) of `dot11RSNAConfigNumberofPTKSA-ReplayCounters` is put in bit 2. See 8.3.2.6 and 8.3.3.4.3. The meaning of the value in the PTKSA/GTKSA/STAKeySA Replay Counter subfield is defined in Table 20dd. The number of replay counters per STAKeySA is the same as the number of replay counters per PTKSA or GTKSA.

Table 20dd—PTKSA/GTKSA/STAKeySA replay counters usage

Replay counter value	Meaning
0	1 replay counter per PTKSA/GTKSA/STAKeySA
1	2 replay counters per PTKSA/GTKSA/STAKeySA
2	4 replay counters per PTKSA/GTKSA/STAKeySA
3	16 replay counters per PTKSA/GTKSA/STAKeySA

— Bits 4–5: GTKSA Replay Counter. A STA sets the GTKSA Replay Counter subfield of the RSN Capabilities field to the value contained in `dot11RSNAConfigNumberofGTKSAReplay-Counters`. The LSB of `dot11RSNAConfigNumberofGTKSAReplayCounters` is put in bit 4. See 8.3.2.6 and 8.3.3.4.3. The meaning of the value in the GTKSA Replay Counter subfield is defined in Table 20dd.

— Bits 6–15: Reserved. The remaining subfields of the RSN Capabilities field are reserved and shall be set to 0 on transmission and ignored on reception.

7.3.2.25.4 PMKID

The PMKID Count and List fields shall be used only in the RSN information element in the (Re)Association Request frame to an AP. The PMKID Count specifies the number of PMKIDs in the PMKID List field. The PMKID list contains 0 or more PMKIDs that the STA believes to be valid for the destination AP. The PMKID can refer to

a) A cached PMKSA that has been obtained through preauthentication with the target AP

b) A cached PMKSA from an EAP authentication

c) A PMKSA derived from a PSK for the target AP

See 8.5.1.2 for the construction of the PMKID.

NOTE—A STA can choose not to insert a PMKID in the PMKID List field if the STA does not want to use that PMKSA.

End of changes to Clause 7.

Replace Clause 8 in its entirety with the following text:

8. Security

8.1 Framework

This amendment defines two classes of security algorithms for IEEE 802.11 networks:

— Algorithms for creating and using a RSNA, called *RSNA algorithms*

— Pre-RSNA algorithms

NOTE—This amendment does not prohibit STAs from simultaneously operating pre-RSNA and RSNA algorithms.

8.1.1 Security methods

Pre-RSNA security comprises the following algorithms:

— WEP, described in 8.2.1

— IEEE 802.11 entity authentication, described in 8.2.2

RSNA security comprises the following algorithms:

— TKIP, described in 8.3.2

— CCMP, described in 8.3.3

— RSNA establishment and termination procedures, including use of IEEE 802.1X authentication, described in 8.4

— Key management procedures, described in 8.5

8.1.2 RSNA equipment and RSNA capabilities

RSNA-capable equipment can create RSNAs. When `dot11RSNAEnabled` is true, RSNA-capable equipment shall include the RSN information element in Beacon, Probe Response, and (Re)Association Request frames and in Message 2 and Message 3 of the 4-Way Handshake. Pre-RSNA equipment is not capable of creating RSNAs.

8.1.3 RSNA establishment

A STA's SME establishes an RSNA in one of four ways:

a) When using IEEE 802.1X AKM in an ESS, an RSNA-capable STA's SME establishes an RSNA as follows:

   1) It identifies the AP as RSNA-capable from the AP's Beacon or Probe Response frames.

   2) It shall invoke Open System authentication.

   3) It negotiates cipher suites during the association process, as described in 8.4.2 and 8.4.3.

   4) It uses IEEE 802.1X to authenticate, as described in 8.4.6 and 8.4.7.

   5) It establishes temporal keys by executing a key management algorithm, using the protocol defined by 8.5.

6) It programs the agreed-upon temporal keys and cipher suites to protect into the MAC and invokes protection. See 8.3.2 and 8.3.3 for a description of the RSNA data protection mechanisms.

b) If an RSNA is based on a PSK in an ESS, the STA's SME establishes an RSNA as follows:

1) It identifies the AP as RSNA-capable from the AP's Beacon or Probe Response frames.

2) It shall invoke Open System authentication.

3) It negotiates cipher suites during the association process, as described in 8.4.2 and 8.4.3.

4) It establishes temporal keys by executing a key management algorithm, using the protocol defined by 8.5. It uses the PSK as the PMK.

5) It protects the data link by programming the negotiated cipher suites and the established temporal key into the MAC and then invoking protection.

c) If an RSNA is based on a PSK in an IBSS, the STA's SME executes the following sequence of procedures:

1) It identifies the peer as RSNA-capable from the peer's Beacon or Probe Response frames.

NOTE—STAs may respond to a data MPDU from an unrecognized STA by sending a Probe Request frame to find out whether the unrecognized STA is RSNA-capable.

2) It may optionally invoke Open System authentication.

3) Each STA uses the procedures in 8.5, to establish temporal keys and to negotiate cipher suites. It uses a PSK as the PMK. Note that two peer STAs may follow this procedure simultaneously. See 8.4.9.

4) It protects the data link by programming the negotiated cipher suites and the established temporal key and then invoking protection.

d) An RSNA-capable STA's SME using IEEE 802.1X AKM in an IBSS establishes an RSNA as follows:

1) It identifies the peer as RSNA-capable from the peer's Beacon or Probe Response frames.

NOTE—STAs may respond to a data MPDU from an unrecognized STA by sending a Probe Request frame to find out whether the unrecognized STA is RSNA-capable.

2) It may optionally invoke Open System authentication.

3) Each station uses IEEE 802.1X to authenticate with the AS associated with the other STA's Authenticator, as described in 8.4.6 and 8.4.7. Hence two authentications are happening at the same time.

4) Each STA's SME establishes temporal keys by executing a key management algorithm, using the protocol defined in 8.5. Hence two such key management algorithms are happening in parallel between any two STA's Supplicants and Authenticators.

5) Both STAs use the agreed-upon temporal key portion of the PTK and pairwise cipher suite from one of the exchanges to protect the link. Each STA uses the GTK established by the exchange it initiated to protect the multicast and broadcast frames it transmits.

The time a security association takes to set up shall be less than the MIB variable `dot11RSNAConfigSA-Timeout`. The security association setup starts when initiated by the SME and completes when the MLME-SETPROTECTION.request primitive is invoked. The action the STA takes on the timeout is a policy decision. Some options include retrying the security association setup or trying another STA. This timeout allows recovery when one of the STAs setting up a security association fails to respond correctly to setting up the security association. It also allows recovery in IBSS when one of the two security associations fails because of a security association timeout.

8.1.4 RSNA assumptions and constraints (informative)

An RSNA assumes the following:

a) Each STA can generate cryptographic-quality random numbers. This assumption is fundamental, as cryptographic methods require a source of randomness. See H.6 for suggested hardware and software methods to achieve randomness suitable for this purpose.

b) When IEEE 802.1X authentication is used, the specific EAP method used performs mutual authentication. This assumption is intrinsic to the design of RSN in IEEE 802.11 LANs and cannot be removed without exposing both the STAs to man-in-the-middle attacks. EAP-MD5 is an example of an EAP method that does not meet this constraint (see IETF RFC 3748). Furthermore, the use of EAP authentication methods where server and client credentials cannot be differentiated reduces the security of the method to that of a PSK due to the fact that malicious insiders can masquerade as servers and establish a man-in-the-middle attack.

   In particular, the mutual authentication requirement implies an unspecified prior enrollment process (e.g., a long-lived authentication key or establishment of trust through a third party such as a certification authority), as the STA must be able to identify the ESS or IBSS as a trustworthy entity and vice versa. The STA shares authentication credentials with the AS utilized by the selected AP or, in the case of PSK, the selected AP. The service set identifier (SSID) provides an unprotected indication that the selected AP's authentication entity shares credentials with the STA. Only the successful completion of the IEEE 802.1X EAP or PSK authentication, after association, can validate any such indication that the AP is connected to an authorized network or service provider.

c) The mutual authentication method must be strong, meaning impersonation attacks are computationally infeasible when based on the information exposed by the authentication. This assumption is intrinsic to the design of RSN.

d) The AP and AS have a trustworthy channel between them that can be used to exchange cryptographic keys without exposure to any intermediate parties.

e) An IEEE 802.1X AS never exposes the common symmetric key to any party except the AP with which the STA is currently communicating. This is a very strong constraint. It implies that the AS itself is never compromised. It also implies that the IEEE 802.1X AS is embedded in the AP or that the AP is physically secure and the AS and the AP lie entirely within the same administrative domain. This assumption follows from the fact that if the AP and the AS are not co-located or do not share pairwise key encryption keys directly, then it is impossible to assure the mobile STA that its key, which is distributed by the AS to the AP, has not been compromised prior to use.

f) Similarly, a STA never shares with a third party a common symmetric key that it shares with a peer. Doing so destroys the utility of the key for detecting MPDU replay and forgery.

g) The STA's Supplicant and the Authenticator generate a different, fresh PTK for each session between the pair. This assumption is fundamental, as reuse of any PTK would enable compromise of all the data protected by that key.

h) The destination STA chosen by the transmitter is the correct destination. For example, Address Resolution Protocol (ARP) and Internet Control Message Protocol (ICMP) are methods of determining the destination STA MAC address that are not secure from attacks by other members of the ESS. One of the possible solutions to this problem might be for the STA to send or receive only frames whose final destination address (DA) or source address (SA) are the AP and for the AP to provide a network layer routing function. However, such solutions are outside the scope of this amendment.

8.2 Pre-RSNA security methods

Except for Open System authentication, all pre-RSNA security mechanisms have been deprecated, as they fail to meet their security goals. New implementations should support pre-RSNA methods only to aid migration to RSNA methods.

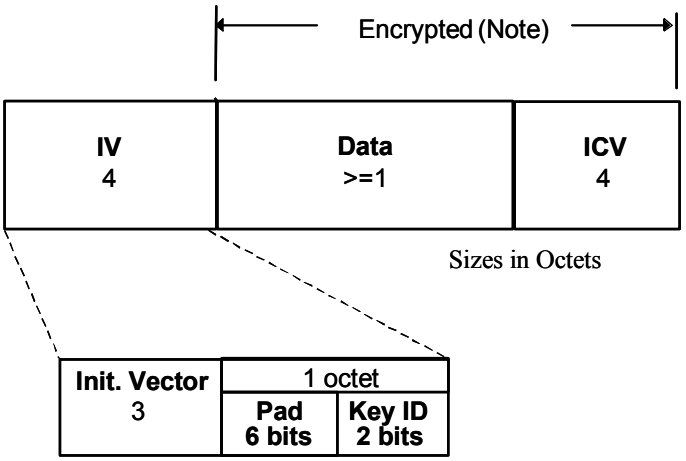
8.2.1 Wired equivalent privacy (WEP)

8.2.1.1 WEP overview

WEP-40 was defined as a means of protecting (using a 40-bit key) the confidentiality of data exchanged among authorized users of a wireless LAN from casual eavesdropping. Implementation of WEP is optional. The same algorithms have been widely used with a 104-bit key instead of a 40-bit key in fielded implementations; this is called WEP-104. The WEP encapsulation and decapsulation mechanics are the same whether a 40-bit or a 104-bit key is used. Therefore, subsequently, WEP can refer to either WEP-40 or WEP-104.

8.2.1.2 WEP MPDU format

Figure 43[10] depicts the encrypted frame body as constructed by the WEP algorithm.



NOTE—The encipherment process has expanded the original MSDU by 8 octets, 4 for the IV field and 4 for the ICV field. The ICV is calculated on the Data field only.

Figure 43—Construction of expanded WEP MPDU

The WEP ICV field shall be 32 bits in length. The expanded frame body shall start with a 32-bit IV field. This field shall contain three subfields: a 3-octet subfield that contains the initialization vector (IV), a 2-bit Key ID subfield, and a 6-bit Pad subfield. The ordering conventions defined in 7.1.1 apply to the IV field and its subfields and to the ICV field. The Key ID subfield contents select one of four possible secret key values for use in decrypting this frame body. When key-mapping keys are used, the Key ID field value is ignored.

Interpretation of these bits is discussed further in 8.2.1.3. The contents of the Pad subfield shall be 0. The Key ID subfield occupies the 2 most significant bits (MSBs) of the last octet of the IV field, while the Pad subfield occupies the 6 LSBs of this octet.

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

8.2.1.3 WEP state

WEP uses encryption keys only; it performs no data authentication. Therefore, it does not have data integrity keys. WEP uses two types of encryption keys: key-mapping keys and default keys.

A key-mapping key is an unnamed key corresponding to a distinct transmitter address-receiver address <TA,RA> pair. Implementations shall use the key-mapping key if it is configured for a <TA,RA> pair. In other words, the key-mapping key shall be used to WEP-encapsulate or -decapsulate MPDUs transmitted by TA to RA, regardless of the presence of other key types. When a key-mapping key for an address pair is present, the WEP Key ID subfield in the MPDU shall be set to 0 on transmit and ignored on receive.

A default key is an item in a four-element MIB array called `dot11WEPDefaultKeys`, named by the value of a related array index called `dot11WEPDefaultKeyID`. If a key-mapping key is not configured for a WEP MPDU's <TA,RA> pair, WEP shall use a default key to encapsulate or decapsulate the MPDU. On transmit, the key selected is the element of the `dot11DefaultKeys` array given by the index `dot11WEPDefaultKeyID`—a value of 0, 1, 2, or 3—corresponding to the first, second, third, or fourth element, respectively, of `dot11WEPDefaultKeys`. The value the transmitter encodes in the WEP Key ID subfield of the transmitted MPDU shall be the `dot11WEPDefaultKeyID` value. The receiver shall use the Key ID subfield of the MPDU to index into `dot11WEPDefaultKeys` to obtain the correct default key. All WEP implementations shall support default keys.

NOTE—Many implementations also support 104-bit WEP keys. These are used exactly like 40-bit WEP keys: a 24-bit WEP IV is prepended to the 104-bit key to construct a 128-bit WEP seed, as explained in 8.2.1.4.3. The resulting 128-bit WEP seed is then consumed by the RC4 stream cipher.

This construction based on 104-bit keys affords no more assurance than the 40-bit construction, and its implementation and use are in no way condoned by this amendment. Rather, the 104-bit construction is noted only to document de facto practice.

The default value for all WEP keys shall be null. WEP implementations shall discard the MSDU and generate an MA-UNITDATA-STATUS.indication with transmission status indicating that a frame may not be encapsulated with a null key in response to any request to encapsulate an MPDU with a null key.

8.2.1.4 WEP procedures

8.2.1.4.1 WEP ICV algorithm

The WEP ICV shall be computed using the CRC-32, as defined in 7.1.3.6, calculated over the plaintext MPDU Data (PDU) field.

8.2.1.4.2 WEP encryption algorithm

A WEP implementation shall use the RC4 stream cipher from RSA Security, Inc., as its encryption and decryption algorithm. RC4 uses a pseudo-random number generator (PRNG) to generate a key stream that it exclusive-ORs (XORs) with a plaintext data stream to produce cipher text or to recover plaintext from a cipher text.

8.2.1.4.3 WEP seed construction

A WEP implementation shall construct a per-MPDU key, called a *seed*, by concatenating an encryption key to an IV.

For WEP-40, bits 0–39 of the WEP key correspond to bits 24–63 of the seed, and bits 0–23 of the IV correspond to bits 0–23 of the seed, respectively. The bit numbering conventions in 7.1.1 apply to the seed. The seed shall be the input to RC4, in order to encrypt or decrypt the WEP Data and ICV fields.

NOTE—For WEP-104, bits 0–103 of the WEP key correspond to bits 24–127 of the seed, and bit 0–23 of the IV correspond to bits 0–23 of the seed, respectively.

The WEP implementation encapsulating an MPDU's plaintext data should select a new IV for every MPDU it WEP-protects. The IV selection algorithm is unspecified. The algorithm used to select the encryption key used to construct the seed is also unspecified.

The WEP implementation decapsulating an MPDU shall use the IV from the received MPDU's Init Vector subfield. See 8.2.1.4.5 for the specification of how the decapsulator selects the key to use to construct the per-MPDU key.

8.2.1.4.4 WEP MPDU encapsulation

WEP shall apply three transformations to the plaintext MPDU to effect the WEP encapsulation. WEP computes the ICV over the plaintext data and appends this after the MPDU data. WEP encrypts the MPDU plaintext data and ICV using RC4 with a seed constructed as specified in 8.2.1.4.3. WEP encodes the IV and key identifier into the IV field, prepended to the encrypted Data field.

Figure 43a depicts the WEP encapsulation process. The ICV shall be computed and appended to the plaintext data prior to encryption, but the IV encoding step may occur in any order convenient for the implementation.

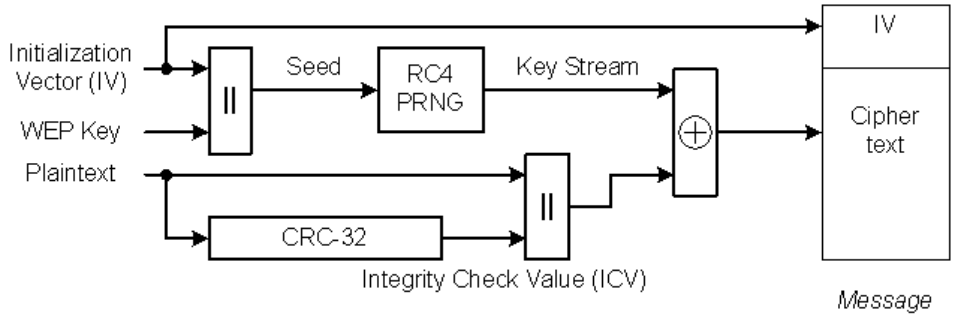


Figure 43a—WEP encapsulation block diagram

8.2.1.4.5 WEP MPDU decapsulation

WEP shall apply three transformations to the WEP MPDU to decapsulate its payload. WEP extracts the IV and key identifier from the received MPDU. If a key-mapping key is present for the <TA,RA> pair, then this shall be used as the WEP key. Otherwise, the key identifier is extracted from the Key ID subfield of the WEP IV field in the received MPDU, identifying the default key to use.

WEP uses the constructed seed to decrypt the Data field of the WEP MPDU; this produces plaintext data and an ICV. Finally WEP recomputes the ICV and bit-wise compares it with the decrypted ICV from the MPDU. If the two are bit-wise identical, then WEP removes the IV and ICV from the MPDU, which is accepted as valid. If they differ in any bit position, WEP generates an error indication to MAC management. MSDUs with erroneous MPDUs (due to inability to decrypt) shall not be passed to LLC.

Figure 43b depicts a block diagram for WEP decapsulation. Unlike encapsulation, the decapsulation steps shall be in the indicated order.

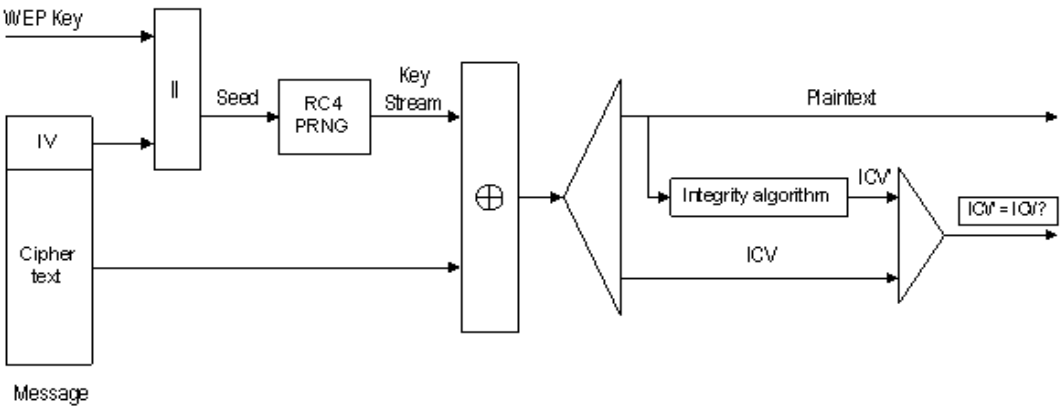


Figure 43b—WEP decapsulation block diagram

8.2.2 Pre-RSNA authentication

8.2.2.1 Overview

In an ESS, a non-AP STA and an AP must both complete an IEEE 802.11 authentication exchange prior to association. Such an exchange is optional in an independent BSS network.

All management frames of subtype Authentication shall be unicast, as IEEE 802.11 authentication is performed between pairs of STAs, i.e., broadcast/multicast authentication is not allowed. Management frames of subtype Deauthentication are advisory and may be sent as group addressed frames.

Shared Key authentication is deprecated and should not be implemented except for backward compatibility with pre-RSNA devices.

8.2.2.2 Open System authentication

Open System authentication is a null authentication algorithm. Any STA requesting Open System authentication may be authenticated if `dot11AuthenticationType` at the recipient STA is set to Open System authentication. A STA may decline to authenticate with another requesting STA. Open System authentication is the default authentication algorithm for pre-RSNA equipment.

Open System authentication utilizes a two-message authentication transaction sequence. The first message asserts identity and requests authentication. The second message returns the authentication result. If the result is "successful," the STAs shall be declared mutually authenticated.

In the description in 8.2.2.2.1 and 8.2.2.2.2, the STA initiating the authentication exchange is referred to as the *requester*, and the STA to which the initial frame in the exchange is addressed is referred to as the *responder*.

8.2.2.2.1 Open System authentication (first frame)

— Message type: Management
— Message subtype: Authentication
— Information items:
  • Authentication Algorithm Identification = "Open System"
  • STA Identity Assertion (in SA field of header)

- Authentication transaction sequence number = 1
- Authentication algorithm dependent information (none)

— Direction of message: From requester to responder

8.2.2.2.2 Open System authentication (final frame)

— Message type: Management

— Message subtype: Authentication

— Information items:

- Authentication Algorithm Identification = "Open System"
- Authentication transaction sequence number = 2
- Authentication algorithm dependent information (none)
- The result of the requested authentication as defined in 7.3.1.9

— Direction of message: From responder to requester

If `dot11AuthenticationType` does not include the value "Open System," the result code shall not take the value "successful."

8.2.2.3 Shared Key authentication

Shared Key authentication seeks to authenticate STAs as either a member of those who know a shared secret key or a member of those who do not.

Shared Key authentication can be used if and only if WEP has been selected.

This mechanism uses a shared key delivered to participating STAs via a secure channel that is independent of IEEE 802.11. This shared key is set in a write-only MIB attribute with the intent to keep the key value internal to the STA.

A STA shall not initiate a Shared Key authentication exchange unless its `dot11PrivacyOption-Implemented` attribute is true.

In the description in 8.2.2.3.1 through 8.2.2.3.5, the STA initiating the authentication exchange is referred to as the *requester*, and the STA to which the initial frame in the exchange is addressed is referred to as the *responder.*

8.2.2.3.1 Shared Key authentication (first frame)

— Message type: Management

— Message subtype: Authentication

— Information items:

- STA Identity Assertion (in SA field of header)
- Authentication Algorithm Identification = "Shared Key"
- Authentication transaction sequence number = 1
- Authentication algorithm dependent information (none)

— Direction of message: From requester to responder

8.2.2.3.2 Shared Key authentication (second frame)

Before sending the second frame in the Shared Key authentication sequence, the responder shall use WEP to generate a string of octets to be used as the authentication challenge text.

— Message type: Management
— Message subtype: Authentication
— Information items:
  • Authentication Algorithm Identification = "Shared Key"
  • Authentication transaction sequence number = 2
  • Authentication algorithm dependent information = The authentication result
  • The result code of the requested authentication as defined in 7.3.1.9.

    If the status code is not "successful," this shall be the last frame of the transaction sequence; and the content of the challenge text field is unspecified.

    If the status code is "successful," the following additional information items shall have valid contents:

    Authentication algorithm dependent information = The challenge text

    This authentication result shall be of fixed length of 128 octets. The field shall be filled with octets generated by the WEP PRNG. The actual value of the challenge field is unimportant, but the value shall not be a static value.
— Direction of message: From responder to requester

8.2.2.3.3 Shared Key authentication (third frame)

The requester shall copy the challenge text from the second frame into the third frame. The third frame shall be transmitted after encapsulation by WEP, as defined in 8.2.1, using the shared key.

— Message type: Management
— Message subtype: Authentication
— Information items:
  • Authentication Algorithm Identification = "Shared Key"
  • Authentication transaction sequence number = 3
  • Authentication algorithm dependent information = The challenge text from the second frame
— Direction of message: From requester to responder

8.2.2.3.4 Shared Key authentication (final frame)

The responder shall WEP-decapsulate the third frame as described in 8.2.1. If the WEP ICV check is successful, the responder shall compare the decrypted contents of the Challenge Text field with the challenge text sent in second frame. If they are the same, then the responder shall respond with a successful status code in the final frame of the sequence. If the WEP ICV check fails or challenge text comparison fails, the responder shall respond with an unsuccessful status code in final frame.

— Message type: Management
— Message subtype: Authentication
— Information items:
  • Authentication Algorithm Identification = "Shared Key"
  • Authentication transaction sequence number = 4
  • Authentication algorithm dependent information = The authentication result

- The result code of the requested authentication as defined in 7.3.1.9.

  This is a fixed length item with values "successful" and "unsuccessful."

— Direction of message: From responder to requester

8.2.2.3.5 Shared key MIB attributes

To transmit a management frame of subtype Authentication, with an Authentication Transaction Sequence Number field value of 2, the MAC shall operate according to the following decision tree:

if `dot11PrivacyOptionImplemented` is "false" **then**
the MMPDU is transmitted with a sequence of 0 octets in the Challenge Text field and a status code value of 13
else
the MMPDU is transmitted with a sequence of 128 octets generated using the WEP PRNG and a key whose value is unspecified and beyond the scope of this amendment and a randomly chosen IV value (note that this will typically be selected by the same mechanism for choosing IV values for transmitted data MPDUs) in the Challenge Text field and a status code value of 0 (the IV used is immaterial and is not transmitted). Note that there are cryptographic issues involved in the choice of key/IV for this process as the challenge text is sent unencrypted and, therefore, provides a known output sequence from the PRNG.
endif

To receive a management frame of subtype Authentication, with an Authentication Transaction Sequence Number field value of 2, the MAC shall operate according to the following decision tree:

if the Protected Frame subfield of the Frame Control field is 1 **then**
respond with a status code value of 15
else
if `dot11PrivacyOptionImplemented` is "true" **then**
if there is a mapping in `dot11WEPKeyMappings` matching the MSDU's TA **then**
if that key is null **then**
respond with a frame whose Authentication Transaction Sequence Number field is 3 that contains the appropriate authentication algorithm number, a status code value of 15, and no Challenge Text field, without encrypting the contents of the frame
else
respond with a frame whose Authentication Transaction Sequence Number field is 3 that contains the appropriate authentication algorithm number, a status code value of 0, and the identical Challenge Text field, encrypted using that key, and setting the Key ID subfield in the IV field to 0
endif
else
if `dot11WEPDefaultKeys`[`dot11WEPDefaultKeyID`] is null **then**
respond with a frame whose Authentication Transaction Sequence Number field is 3 that contains the appropriate authentication algorithm number, a status code value of 15, and no Challenge Text field, without encrypting the contents of the frame
else
respond with a frame whose Authentication Transaction Sequence Number field is 3 that contains the appropriate authentication algorithm number, a status code value of 0, and the identical Challenge Text field, WEP-encapsulating the frame under the key `dot11WEPDefaultKeys`[`dot11WEPDefaultKeyID`], and setting the Key ID subfield in the IV field to `dot11WEPDefaultKeyID`
endif

endif

else

respond with a frame whose Authentication Transaction Sequence Number field is 3 that contains the appropriate authentication algorithm number, a status code value of 13, and no Challenge Text field, without encrypting the contents of the frame

endif

endif

When receiving a management frame of subtype Authentication, with an Authentication Transaction Sequence Number field value of 3, the MAC shall operate according to the following decision tree:

if the Protected Frame subfield of the Frame Control field is 0 **then**

respond with a status code value of 15

else

if `dot11PrivacyOptionImplemented` is "true" **then**

if there is a mapping in `dot11WEPKeyMappings` matching the MSDU's TA **then**

if that key is null **then**

respond with a frame whose Authentication Transaction Sequence Number field is 4 that contains the appropriate authentication algorithm number and a status code value of 15 without encrypting the contents of the frame

else

WEP-decapsulate with that key, incrementing `dot11WEPICVErrorCount` and responding with a status code value of 15 if the ICV check fails

endif

else

if `dot11WEPDefaultKeys[dot11WEPDefaultKeyID]` is null **then**

respond with a frame whose Authentication Transaction Sequence Number field is 4 that contains the appropriate authentication algorithm number and a status code value of 15 without encrypting the contents of the frame

else

WEP-decapsulate with `dot11WEPDefaultKeys[dot11WEPDefaultKeyID]`, incrementing `dot11WEPICVErrorCount` and responding with a status code value of 15 if the ICV check fails

endif

endif

else

respond with a frame whose Authentication Transaction Sequence Number field is 4 that contains the appropriate authentication algorithm number and a status code value of 15

endif

endif

The attribute `dot11PrivacyInvoked` shall not take the value of true if the attribute `dot11PrivacyOptionImplemented` is false. Setting `dot11WEPKeyMappings` to a value that includes more than `dot11WEPKeyMappingLength` entries is illegal and shall have an implementation-specific effect on the operation of the confidentiality service. Note that `dot11WEPKeyMappings` may contain from zero to `dot11WEPKeyMappingLength` entries, inclusive.

The values of the attributes in the aPrivacygrp should not be changed during the authentication sequence, as unintended operation may result.

8.3 RSNA data confidentiality protocols

8.3.1 Overview

This amendment defines two RSNA data confidentiality and integrity protocols: TKIP and CCMP. Implementation of CCMP shall be mandatory in all IEEE 802.11 devices claiming RSNA compliance. Implementation of TKIP is optional for an RSNA. A design aim for TKIP was that the algorithm should be implementable within the capabilities of most devices supporting only WEP, so that many such devices would be field-upgradeable by the supplier to support TKIP.

NOTE—Use of any of the confidentiality algorithms depends on local policies. The confidentiality and integrity mechanisms of TKIP are not as robust as those of CCMP. TKIP is designed to operate within the hardware limitations of a broad class of pre-RSNA devices. TKIP is suitable for firmware-only, hardware-compatible upgrade of fielded equipment. RSNA devices should only use TKIP when communicating with devices that are unable or not configured to communicate using CCMP.

8.3.2 Temporal Key Integrity Protocol (TKIP)

8.3.2.1 TKIP overview

The TKIP is a cipher suite enhancing the WEP protocol on pre-RSNA hardware. TKIP modifies WEP as follows:

a) A transmitter calculates a keyed cryptographic message integrity code (MIC) over the MSDU SA and DA, the MSDU priority (see 8.3.2.3), and the MSDU plaintext data. TKIP appends the computed MIC to the MSDU data prior to fragmentation into MPDUs. The receiver verifies the MIC after decryption, ICV checking, and defragmentation of the MPDUs into an MSDU and discards any received MSDUs with invalid MICs. TKIP's MIC provides a defense against forgery attacks.

b) Because of the design constraints of the TKIP MIC, it is still possible for an adversary to compromise message integrity; therefore, TKIP also implements countermeasures. The countermeasures bound the probability of a successful forgery and the amount of information an attacker can learn about a key.

c) TKIP uses a per-MPDU TKIP sequence counter (TSC) to sequence the MPDUs it sends. The receiver drops MPDUs received out of order, i.e., not received with increasing sequence numbers. This provides replay protection. TKIP encodes the TSC value from the sender to the receiver as a WEP IV and extended IV.

d) TKIP uses a cryptographic mixing function to combine a temporal key, the TA, and the TSC into the WEP seed. The receiver recovers the TSC from a received MPDU and utilizes the mixing function to compute the same WEP seed needed to correctly decrypt the MPDU. The key mixing function is designed to defeat weak-key attacks against the WEP key.

TKIP defines additional MIB variables; see Annex D.

8.3.2.1.1 TKIP encapsulation

TKIP enhances the WEP encapsulation with several additional functions, as depicted in Figure 43c.

a) TKIP MIC computation protects the MSDU Data field and corresponding SA, DA, and Priority fields. The computation of the MIC is performed on the ordered concatenation of the SA, DA, Priority, and MSDU Data fields. The MIC is appended to the MSDU Data field. TKIP discards any MIC padding prior to appending the MIC.

b) If needed, IEEE 802.11 fragments the MSDU with MIC into one or more MPDUs. TKIP assigns a monotonically increasing TSC value to each MPDU, taking care that all the MPDUs generated from the same MSDU have the same value of extended IV (see 8.3.2.2).

c) For each MPDU, TKIP uses the key mixing function to compute the WEP seed.

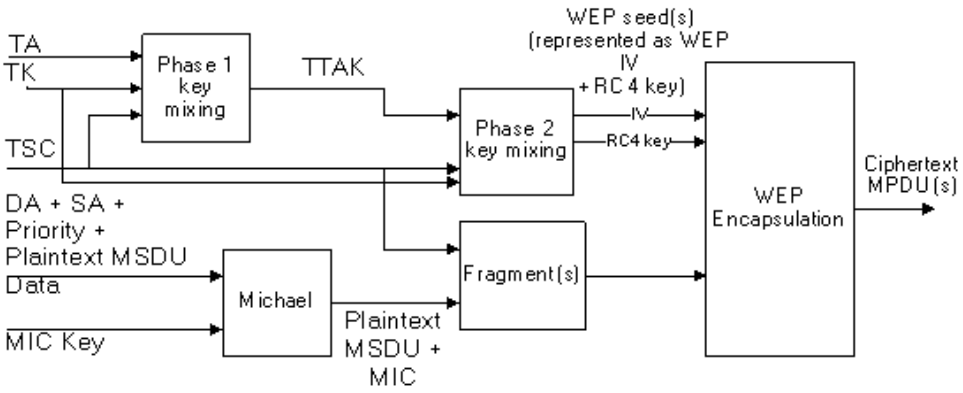


Figure 43c—TKIP encapsulation block diagram

d) TKIP represents the WEP seed as a WEP IV and RC4 key and passes these with each MPDU to WEP for generation of the ICV (see 7.1.3.6), and for encryption of the plaintext MPDU, including all or part of the MIC, if present. WEP uses the WEP seed as a WEP default key, identified by a key identifier associated with the temporal key.

NOTE—When the TSC space is exhausted, the choices available to an implementation are to replace the temporal key with a new one or to end communications. Reuse of any TSC value compromises already sent traffic. Note that retransmitted MPDUs reuse the TSC without any compromise of security. The TSC is large enough, however, that TSC space exhaustion should not be an issue.

In Figure 43c, the TKIP-mixed transmit address and key (TTAK) denotes the intermediate key produced by Phase 1 of the TKIP mixing function (see 8.3.2.5).

8.3.2.1.2 TKIP decapsulation

TKIP enhances the WEP decapsulation process with the following additional steps:

a) Before WEP decapsulates a received MPDU, TKIP extracts the TSC sequence number and key identifier from the WEP IV and the extended IV. TKIP discards a received MPDU that violates the sequencing rules (see 8.3.2.6) and otherwise uses the mixing function to construct the WEP seed.

b) TKIP represents the WEP seed as a WEP IV and RC4 key and passes these with the MPDU to WEP for decapsulation.

c) If WEP indicates the ICV check succeeded, the implementation reassembles the MPDU into an MSDU. If the MSDU defragmentation succeeds, the receiver verifies the TKIP MIC. If MSDU defragmentation fails, then the MSDU is discarded.

d) The MIC verification step recomputes the MIC over the MSDU SA, DA, Priority, and MSDU Data fields (but not the TKIP MIC field). The calculated TKIP MIC result is then compared bit-wise against the received MIC.

e) If the received and the locally computed MIC values are identical, the verification succeeds, and TKIP shall deliver the MSDU to the upper layer. If the two differ, then the verification fails; the receiver shall discard the MSDU and shall engage in appropriate countermeasures.

Figure 43d depicts this process.

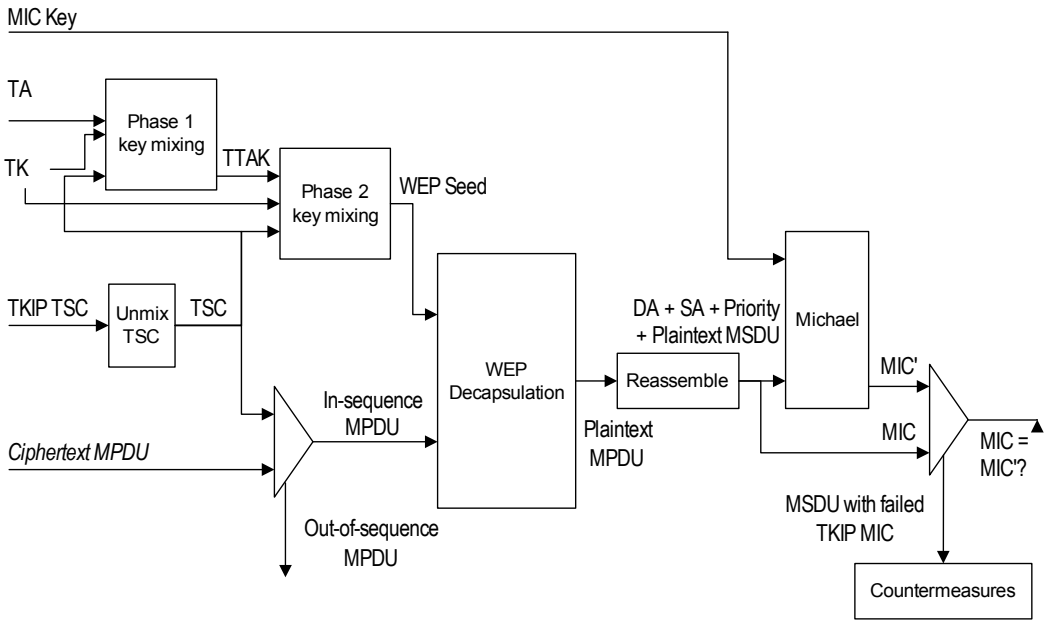


Figure 43d—TKIP decapsulation block diagram

8.3.2.2 TKIP MPDU formats

TKIP reuses the pre-RSNA WEP MPDU format. It extends the MPDU by 4 octets to accommodate an extension to the WEP IV, denoted by the Extended IV field, and extends the MSDU format by 8 octets to accommodate the new MIC field. TKIP inserts the Extended IV field immediately after the WEP IV field and before the encrypted data. TKIP appends the MIC to the MSDU Data field; the MIC becomes part of the encrypted data.

Once the MIC is appended to the MSDU data, the added MIC octets are considered part of the MSDU for subsequent fragmentation.

Figure 43e depicts the layout of the encrypted MPDU when using TKIP. Note the figure only depicts the case when the MSDU can be encapsulated in a single MPDU.

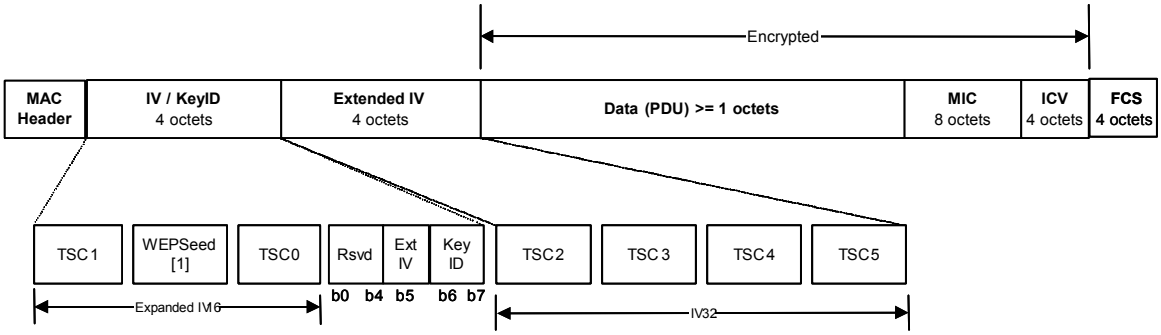


Figure 43e—Construction of expanded TKIP MPDU

The ExtIV bit in the Key ID octet indicates the presence or absence of an extended IV. If the ExtIV bit is 0, only the nonextended IV is transferred. If the ExtIV bit is 1, an extended IV of 4 octets follows the original IV. For TKIP the ExtIV bit shall be set, and the Extended IV field shall be supplied. The ExtIV bit shall be 0 for WEP frames. The Key ID field shall be set to the key index supplied by the MLME-SETKEYS.request primitive for the key used in encapsulation of the frame.

TSC5 is the most significant octet of the TSC, and TSC0 is the least significant. Octets TSC0 and TSC1 form the IV sequence number and are used with the TKIP Phase 2 key mixing. Octets TSC2–TSC5 are used in the TKIP Phase 1 key hashing and are in the Extended IV field. When the lower 16-bit sequence number rolls over (0xFFFF → 0x0000), the extended IV value, i.e., the upper 32 bits of the entire 48-bit TSC, shall be incremented by 1.

NOTE—The rationale for this construction is as follows:

— Aligning on word boundaries eases implementation on legacy devices.
— Adding 4 octets of extended IV eliminates TSC exhaustion as a reason to rekey.
— Key ID octet changes. Bit 5 indicates that an extended IV is present. The receiver/transmitter interprets the 4 octets following the Key ID as the extended IV. The receiving/transmitting STA also uses the value of octets TSC0 and TSC1 to detect that the cached TTAK must be updated.

The Extended IV field shall not be encrypted.

WEPSeed[1] is not used to construct the TSC, but is set to (TSC1 | 0x20) & 0x7f.

TKIP shall encrypt all the MPDUs generated from one MSDU under the same temporal key.

8.3.2.3 TKIP MIC

Flaws in the IEEE 802.11 WEP design cause it to fail to meet its goal of protecting data traffic content from casual eavesdroppers. Among the most significant WEP flaws is the lack of a mechanism to defeat message forgeries and other active attacks. To defend against active attacks, TKIP includes a MIC, named Michael. This MIC offers only weak defenses against message forgeries, but it constitutes the best that can be achieved with the majority of legacy hardware. TKIP uses different MIC keys depending on the direction of the transfer as described in 8.6.1 and 8.6.2.

Annex H contains an implementation of the TKIP MIC. It also provides test vectors for the MIC.

8.3.2.3.1 Motivation for the TKIP MIC

Before defining the details of the MIC, it is useful to review the context in which this mechanism operates. Active attacks enabled by the original WEP design include the following:

— Bit-flipping attacks
— Data (payload) truncation, concatenation, and splicing
— Fragmentation attacks
— Iterative guessing attacks against the key
— Redirection by modifying the MPDU DA or RA field
— Impersonation attacks by modifying the MPDU SA or TA field

The MIC makes it more difficult for any of these attacks to succeed.

All of these attacks remain at the MPDU level with the TKIP MIC. The MIC, however, applies to the MSDU, so it blocks successful MPDU-level attacks. TKIP applies the MIC to the MSDU at the transmitter and verifies it at the MSDU level at the receiver. If a MIC check fails at the MSDU level, the implementation shall discard the MSDU and invoke countermeasures (see 8.3.2.4).

Figure 43f depicts different peer layers communicating.

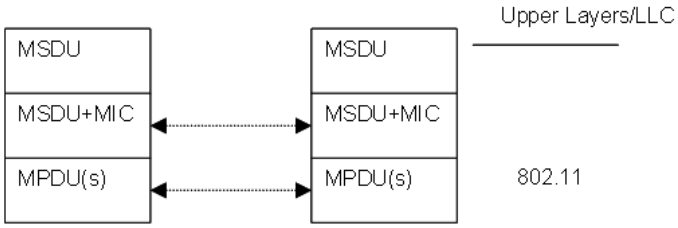


Figure 43f—TKIP MIC relation to IEEE 802.11 processing (informative)

This figure depicts an architecture where the MIC is logically appended to the raw MSDU in response to the MA-UNITDATA.request primitive. The TKIP MIC is computed over

— The MSDU DA
— The MSDU SA
— The MSDU Priority (reserved for future use)
— The entire unencrypted MSDU data (payload)

The DA field, SA field, three reserved octets, and a 1-octet Priority field are used only for calculating the MIC. The Priority field shall be 0 and reserved for future use. The fields in Figure 43g are treated as an octet stream using the conventions described in 7.1.1.

6	6	1	3	M	1	1	1	1	1	1	1	1	octets
DA	SA	Priority	0	Data	M 0	M 1	M 2	M 3	M 4	M 5	M 6	M 7	

Figure 43g—TKIP MIC processing format

TKIP appends the MIC at the end of the MSDU payload. The MIC is 8 octets in size for Michael. The IEEE 802.11 MAC then applies its normal processing to transmit this MSDU-with-MIC as a sequence of one or more MPDUs. In other words, the MSDU-with-MIC can be partitioned into one or more MPDUs, the WEP ICV is calculated over each MPDU, and the MIC can even be partitioned to lie in two MPDUs after fragmentation. The TKIP MIC augments, but does not replace, the WEP ICV. Because the TKIP MIC is a weak construction, TKIP protects the MIC with encryption, which makes TKIP MIC forgeries more difficult. The WEP ICV helps to prevent false detection of MIC failures that would cause countermeasures to be invoked.

The receiver reverses this procedure to reassemble the MSDU; and, after the MSDU has been logically reassembled, the IEEE 802.11 MAC verifies the MIC prior to delivery of the MSDU to upper layers. If the MIC validation succeeds, the MAC delivers the MSDU. If the MIC validation fails, the MAC shall discard the MSDU and invoke countermeasures (see 8.3.2.4).

NOTE—TKIP calculates the MIC over the MSDU rather than the MPDU, because doing so increases the implementation flexibility with pre-existing WEP hardware.

It should be noted that a MIC alone cannot provide complete forgery protection, as it cannot defend against replay attacks. Therefore, TKIP provides replay detection by TSC sequencing and ICV validation. Furthermore, if TKIP is utilized with a GTK, an insider STA can masquerade as any other STA belonging to the group.

8.3.2.3.2 Definition of the TKIP MIC

Michael generates a 64-bit MIC. The Michael key consists of 64 bits, represented as an 8-octet sequence, $k_0...k_7$. This is converted to two 32-bit words, K_0 and K_1. Throughout this subclause, all conversions between octets and 32-bit words shall use the little endian conventions, given in 7.1.1.

Michael operates on each MSDU including the Priority field, 3 reserved octets, SA field, and DA field. An MSDU consists of octets $m_0...m_{n-1}$ where n is the number of MSDU octets, including SA, DA, Priority, and Data fields. The message is padded at the end with a single octet with value 0x5a, followed by between 4 and 7 zero octets. The number of zero octets is chosen so that the overall length of the padded MSDU is a multiple of four. The padding is not transmitted with the MSDU; it is used to simplify the computation over the final block. The MSDU is then converted to a sequence of 32-bit words $M_0 ...M_{N-1}$, where $N = \lceil (n+5)/4 \rceil$, and where $\lceil a \rceil$ means to round a up to the nearest integer. By construction, $M_{N-1} = 0$ and $M_{N-2} \neq 0$.

The MIC value is computed iteratively starting with the key value (K_0 and K_1) and applying a block function b for every message word, as shown in Figure 43h. The algorithm loop runs a total of N times (i takes on the values 0 to N–1 inclusive), where N is as above, the number of 32-bit words composing the padded MSDU. The algorithm results in two words (l and r), which are converted to a sequence of 8 octets using the least-significant-octet-first convention:

— M0 = l & 0xff

— M1 = (l/0x100) & 0xff

— M2 = (l/0x10000) & 0xff

— M3 = (l/0x1000000) & 0xff

— M4 = r & 0xff

— M5 = (r/0x100) & 0xff

— M6 = (r/0x10000) & 0xff

— M7 = (r/0x1000000) & 0xff

This is the MIC value. The MIC value is appended to the MSDU as data to be sent.

Input: Key ($K0$, $K1$) and padded MSDU (represented as 32-bit words) $M0...MN-1$
Output: MIC value ($V0$, $V1$)
MICHAEL(($K0$, $K1$) , ($M0$,... ,MN))
(l,r) ← ($K0$, $K1$)
for i = 0 **to** N–1 **do**
l ← l ⊕ M-i
(l, r) ← b(l, r)
return (l,r)

Figure 43h—Michael message processing

Figure 43i defines the Michael block function b. It is a Feistel-type construction with alternating additions and XOR operations. It uses <<< to denote the rotate-left operator on 32-bit values, >>> for the rotate-right operator, and XSWAP for a function that swaps the position of the 2 least significant octets. It also uses the position of the two most significant octets in a word.

Input: (*l*,*r*)
Output: (l,r)
b(L,R)
$r \leftarrow r \oplus (l <<< 17)$
$l \leftarrow (l + r) \bmod 232$
$r \leftarrow r \oplus XSWAP(l)$
$l \leftarrow (l + r) \bmod 232$
$r \leftarrow r \oplus (l <<< 3)$
$l \leftarrow (l + r) \bmod 232$
$r \leftarrow r \oplus (l >>> 2)$
$l \leftarrow (l + r) \bmod 232$
return (*l*, *r*)

Figure 43i—Michael block function

8.3.2.4 TKIP countermeasures procedures

The TKIP MIC trades off security in favor of implementability on pre-RSNA devices. Michael provides only weak protection against active attacks. A failure of the MIC in a received MSDU indicates a probable active attack. A successful attack against the MIC would mean an attacker could inject forged data frames and perform further effective attacks against the encryption key itself. If TKIP implementation detects a probable active attack, TKIP shall take countermeasures as specified in this subclause. These countermeasures accomplish the following goals:

— MIC failure events *should* be logged as a security-relevant matter. A MIC failure is an almost certain indication of an active attack and warrants a follow-up by the system administrator.

— The rate of MIC failures *must* be kept below two per minute. This implies that STAs and APs detecting two MIC failure events within 60 s must disable all receptions using TKIP for a period of 60 s. The slowdown makes it difficult for an attacker to make a large number of forgery attempts in a short time.

— As an additional security feature, the PTK and, in the case of the Authenticator, the GTK should be changed.

Before verifying the MIC, the receiver shall check the FCS, ICV, and TSC for all related MPDUs. Any MPDU that has an invalid FCS, an incorrect ICV, or a TSC value that is less than or equal to the TSC replay counter shall be discarded before checking the MIC. This avoids unnecessary MIC failure events. Checking the TSC before the MIC makes countermeasure-based denial-of-service attacks harder to perform. While the FCS and ICV mechanisms are sufficient to detect noise, they are insufficient to detect active attacks. The FCS and ICV provide error detection, but not integrity protection.

A single counter or timer shall be used to log MIC failure events. These failure events are defined as follows:

— For an Authenticator:
  — Detection of a MIC failure on a received unicast frame.
  — Receipt of Michael MIC Failure Report frame.
— For a Supplicant:
  — Detection of a MIC failure on a received unicast or broadcast/multicast frame.
  — Attempt to transmit a Michael MIC Failure Report frame.

The number of MIC failures is accrued independent of the particular key context. Any single MIC failure, whether detected by the Supplicant or the Authenticator and whether resulting from a group MIC key failure or a pairwise MIC key failure, shall be treated as cause for a MIC failure event.

The Supplicant uses a single Michael MIC Failure Report frame to report a MIC failure event to the Authenticator. A Michael MIC Failure Report is an EAPOL-Key frame with the following Key Information field bits set to 1: MIC bit, Error bit, Request bit, Secure bit. The Supplicant protects this message with the current PTK; the Supplicant uses the KCK portion of the PTK to compute the IEEE 802.1X EAPOL MIC.

MLME-MICHEALMICFAILURE.indication primitive is used by the IEEE 802.11 MAC to attempt to indicate a MIC failure to the local IEEE 802.1X Supplicant or Authenticator. MLME-EAPOL.request primitive is used by the Supplicant to send the EAPOL-Key frame containing the Michael MIC Failure Report. MLME-EAPOL.confirm primitive indicates to the Supplicant when an IEEE 802.11 MAC acknowledgment (ACK) has been received for this EAPOL-Key frame.

The first MIC failure shall be logged, and a timer initiated to enable enforcement of the countermeasures. If the MIC failure event is detected by the Supplicant, it shall also report the event to the AP by sending a Michael MIC Failure Report frame.

If a subsequent MIC failure occurs within 60 s of the most recent previous failure, then a STA whose IEEE 802.1X entity has acted as a Supplicant shall deauthenticate (as defined in 11.3.3) itself or deauthenticate all the STAs with a security association if its IEEE 802.1X entity acted as an Authenticator. For an IBSS STA, both Supplicant and Authenticator actions shall be taken. Furthermore, the device shall not receive or transmit any TKIP-encrypted data frames, and shall not receive or transmit any unencrypted data frames other than IEEE 802.1X messages, to or from any peer for a period of at least 60 s after it detects the second failure. If the device is an AP, it shall disallow new associations using TKIP during this 60 s period; at the end of the 60 s period, the AP shall resume normal operations and allow STAs to (re)associate. If the device is an IBSS STA, it shall disallow any new security associations using TKIP during this 60 s period. If the device is a Supplicant, it shall first send a Michael MIC Failure Report frame prior to revoking its PTKSA and deauthenticating itself.

The `aMICFailTime` attribute shall contain the `sysUpTime` value at the time the MIC failure was logged.

8.3.2.4.1 TKIP countermeasures for an Authenticator

The countermeasures used by an Authenticator are depicted in Figure 43j and described as follows:

a) For an Authenticator's STA that receives a frame with a MIC error,
   1) Discard the frame.
   2) Increment the MIC failure counter, `dot11RSNAStatsTKIPLocalMIC-Failures.`
   3) Generate a MLME-MICHAELMICFAILURE.indication primitive.

b) For an Authenticator that receives a MLME-MICHAELMICFAILURE.indication primitive or a Michael MIC Failure Report frame,
   1) If it is a Michael MIC Failure Report frame, then increment `dot11RSNAStatsTKIP-RemoteMICFailures`.
   2) If this is the first MIC failure within the past 60 s, initialize the countermeasures timer.
   3) If less than 60 s have passed since the most recent previous MIC failure, the Authenticator shall deauthenticate and delete all PTKSAs for all STAs using TKIP. If the current GTKSA uses TKIP, that GTKSA shall be discarded, and a new GTKSA constructed, but not used for 60 s. The Authenticator shall refuse the construction of new PTKSAs using TKIP as one or more of the ciphers for 60 s. At the end of this period, the MIC failure counter and timer shall be reset, and creation of PTKSAs accepted as usual.
   4) If the Authenticator is using IEEE 802.1X authentication, the Authenticator shall transition the state of the IEEE 802.1X Authenticator state machine to the INITIALIZE state. This will restart the IEEE 802.1X state machine. If the Authenticator is instead using PSKs, this step is omitted.

Note that a Supplicant's STA may deauthenticate with a reason code of MIC failure if it is an ESS STA. The Authenticator shall not log the deauthenticate as a MIC failure event to prevent denial-of-service attacks through deauthentications. The Supplicant's STA must report the MIC failure event through the Michael MIC Failure Report frame in order for the AP to log the event.

The requirement to deauthenticate all STAs using TKIP will include those using CCMP as a pairwise cipher if they are also using TKIP as the group cipher.

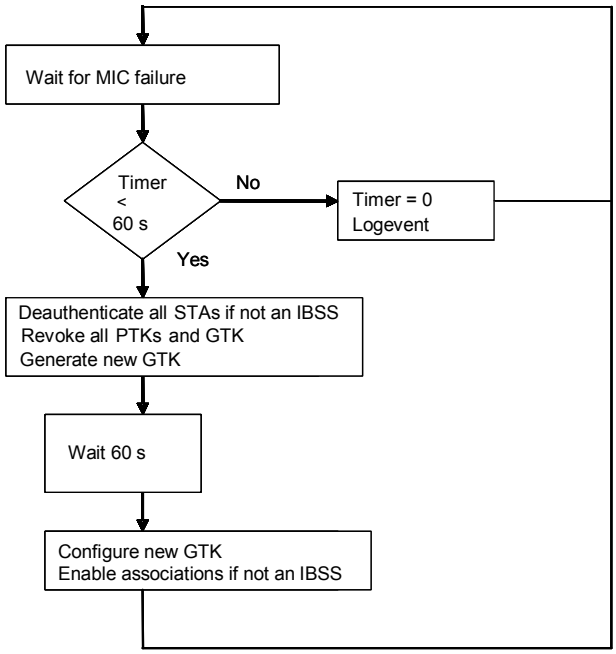


Figure 43j—Authenticator MIC countermeasures

8.3.2.4.2 TKIP countermeasures for a Supplicant

The countermeasures used by a Supplicant are depicted in Figure 43k and described as follows:

a) For a Supplicant's STA that receives a frame with a MIC error,
   1) Increment the MIC failure counter, `dot11RSNAStatsTKIPLocalMIC-Failures`.
   2) Discard the offending frame.
   3) Generate a MLME-MICHAELMICFAILURE.indication primitive.

b) For a Supplicant that receives an MLME-MICHAELMICFAILURE.indication primitive from its STA,
   1) Send a Michael MIC Failure Report frame to the AP.
   2) If this is the first MIC failure within the past 60 s, initialize the countermeasures timer.
   3) If less than 60 s have passed since the most recent previous MIC failure, delete the PTKSA and GTKSA. Deauthenticate from the AP and wait for 60 s before (re)establishing a TKIP association with the same AP. A TKIP association is any IEEE 802.11 association that uses TKIP for its pairwise or group cipher suite.

c) If a non-AP STA receives a deauthenticate frame with the reason code "MIC failure," it cannot be certain that the frame has not been forged, as it does not contain a MIC. The STA may attempt association with this, or another, AP. If the frame was genuine, then it is probable that attempts to associate with the same AP requesting the use of TKIP will fail because the AP will be conducting countermeasures.

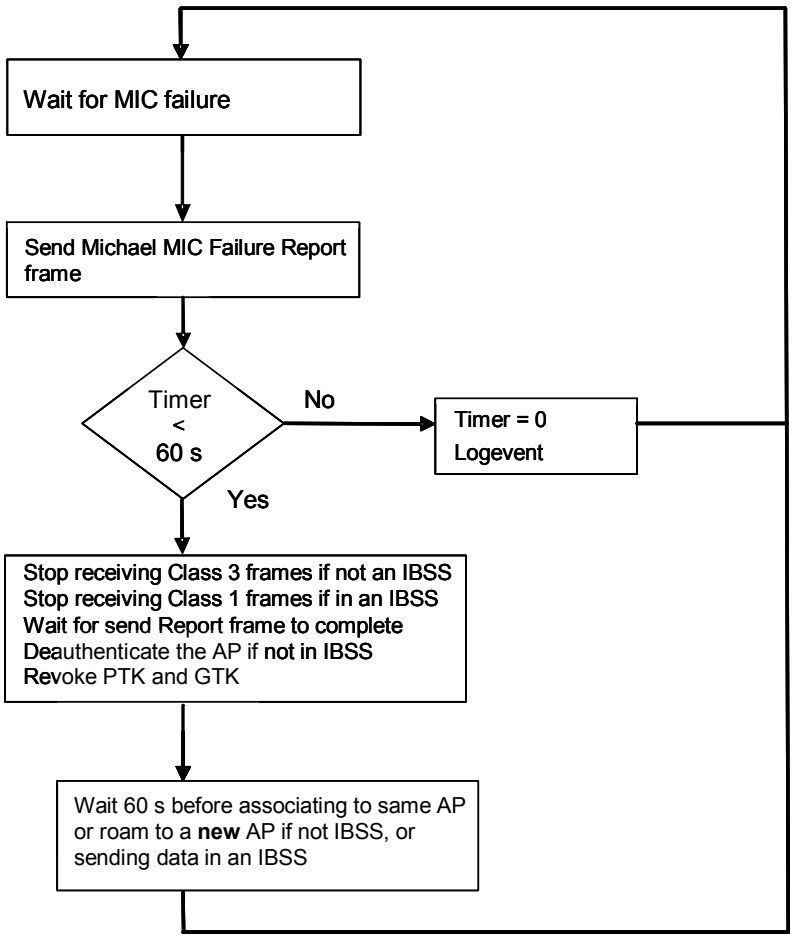


Figure 43k—Supplicant MIC countermeasures

8.3.2.5 TKIP mixing function

Annex H defines a C-language reference implementation of the TKIP mixing function. It also provides test vectors for the mixing function.

The mixing function has two phases. Phase 1 mixes the appropriate temporal key (pairwise or group) with the TA and TSC. A STA may cache the output of this phase to reuse with subsequent MPDUs associated with the same temporal key and TA. Phase 2 mixes the output of Phase 1 with the TSC and temporal key (TK) to produce the WEP seed, also called the *per-frame key*. The WEP seed may be precomputed before it is used. The two-phase process may be summarized as follows:

```
TTAK := Phase1 (TK, TA, TSC)
WEP seed := Phase2 (TTAK, TK, TSC)
```

8.3.2.5.1 S-Box

Both Phase 1 and Phase 2 rely on an S-box, defined in this subclause. The S-box substitutes one 16-bit value with another 16-bit value. This function may be implemented as a table look up.

NOTE—The S-box is a nonlinear substitution. The table look-up can be organized as either a single table with 65 536 entries and a 16-bit index (128K octets of table) or two tables with 256 entries and an 8-bit index (1024 octets for both tables). When the two smaller tables are used, the high-order octet is used to obtain a 16-bit value from one table, the

low-order octet is used to obtain a 16-bit value from the other table, and the S-box output is the XOR (⊕) of the two 16-bit values. The second S-box table is an octet-swapped replica of the first.

```
#define _S_(v16)     (Sbox[0][Lo8(v16)] ^ Sbox[1][Hi8(v16)])

/* 2-byte by 2-byte subset of the full AES S-box table */
const u16b Sbox[2][256]=       /* Sbox for hash (can be in ROM)     */
{ {
   0xC6A5,0xF884,0xEE99,0xF68D,0xFF0D,0xD6BD,0xDEB1,0x9154,
   0x6050,0x0203,0xCEA9,0x567D,0xE719,0xB562,0x4DE6,0xEC9A,
   0x8F45,0x1F9D,0x8940,0xFA87,0xEF15,0xB2EB,0x8EC9,0xFB0B,
   0x41EC,0xB367,0x5FFD,0x45EA,0x23BF,0x53F7,0xE496,0x9B5B,
   0x75C2,0xE11C,0x3DAE,0x4C6A,0x6C5A,0x7E41,0xF502,0x834F,
   0x685C,0x51F4,0xD134,0xF908,0xE293,0xAB73,0x6253,0x2A3F,
   0x080C,0x9552,0x4665,0x9D5E,0x3028,0x37A1,0x0A0F,0x2FB5,
   0x0E09,0x2436,0x1B9B,0xDF3D,0xCD26,0x4E69,0x7FCD,0xEA9F,
   0x121B,0x1D9E,0x5874,0x342E,0x362D,0xDCB2,0xB4EE,0x5BFB,
   0xA4F6,0x764D,0xB761,0x7DCE,0x527B,0xDD3E,0x5E71,0x1397,
   0xA6F5,0xB968,0x0000,0xC12C,0x4060,0xE31F,0x79C8,0xB6ED,
   0xD4BE,0x8D46,0x67D9,0x724B,0x94DE,0x98D4,0xB0E8,0x854A,
   0xBB6B,0xC52A,0x4FE5,0xED16,0x86C5,0x9AD7,0x6655,0x1194,
   0x8ACF,0xE910,0x0406,0xFE81,0xA0F0,0x7844,0x25BA,0x4BE3,
   0xA2F3,0x5DFE,0x80C0,0x058A,0x3FAD,0x21BC,0x7048,0xF104,
   0x63DF,0x77C1,0xAF75,0x4263,0x2030,0xE51A,0xFD0E,0xBF6D,
   0x814C,0x1814,0x2635,0xC32F,0xBEE1,0x35A2,0x88CC,0x2E39,
   0x9357,0x55F2,0xFC82,0x7A47,0xC8AC,0xBAE7,0x322B,0xE695,
   0xC0A0,0x1998,0x9ED1,0xA37F,0x4466,0x547E,0x3BAB,0x0B83,
   0x8CCA,0xC729,0x6BD3,0x283C,0xA779,0xBCE2,0x161D,0xAD76,
   0xDB3B,0x6456,0x744E,0x141E,0x92DB,0x0C0A,0x486C,0xB8E4,
   0x9F5D,0xBD6E,0x43EF,0xC4A6,0x39A8,0x31A4,0xD337,0xF28B,
   0xD532,0x8B43,0x6E59,0xDAB7,0x018C,0xB164,0x9CD2,0x49E0,
   0xD8B4,0xACFA,0xF307,0xCF25,0xCAAF,0xF48E,0x47E9,0x1018,
   0x6FD5,0xF088,0x4A6F,0x5C72,0x3824,0x57F1,0x73C7,0x9751,
   0xCB23,0xA17C,0xE89C,0x3E21,0x96DD,0x61DC,0x0D86,0x0F85,
   0xE090,0x7C42,0x71C4,0xCCAA,0x90D8,0x0605,0xF701,0x1C12,
   0xC2A3,0x6A5F,0xAEF9,0x69D0,0x1791,0x9958,0x3A27,0x27B9,
   0xD938,0xEB13,0x2BB3,0x2233,0xD2BB,0xA970,0x0789,0x33A7,
   0x2DB6,0x3C22,0x1592,0xC920,0x8749,0xAAFF,0x5078,0xA57A,
   0x038F,0x59F8,0x0980,0x1A17,0x65DA,0xD731,0x84C6,0xD0B8,
   0x82C3,0x29B0,0x5A77,0x1E11,0x7BCB,0xA8FC,0x6DD6,0x2C3A,
  },
  {  /* second half of table is byte-reversed version of first! */
   0xA5C6,0x84F8,0x99EE,0x8DF6,0x0DFF,0xBDD6,0xB1DE,0x5491,
   0x5060,0x0302,0xA9CE,0x7D56,0x19E7,0x62B5,0xE64D,0x9AEC,
   0x458F,0x9D1F,0x4089,0x87FA,0x15EF,0xEBB2,0xC98E,0x0BFB,
   0xEC41,0x67B3,0xFD5F,0xEA45,0xBF23,0xF753,0x96E4,0x5B9B,
   0xC275,0x1CE1,0xAE3D,0x6A4C,0x5A6C,0x417E,0x02F5,0x4F83,
   0x5C68,0xF451,0x34D1,0x08F9,0x93E2,0x73AB,0x5362,0x3F2A,
   0x0C08,0x5295,0x6546,0x5E9D,0x2830,0xA137,0x0F0A,0xB52F,
   0x090E,0x3624,0x9B1B,0x3DDF,0x26CD,0x694E,0xCD7F,0x9FEA,
   0x1B12,0x9E1D,0x7458,0x2E34,0x2D36,0xB2DC,0xEEB4,0xFB5B,
   0xF6A4,0x4D76,0x61B7,0xCE7D,0x7B52,0x3EDD,0x715E,0x9713,
   0xF5A6,0x68B9,0x0000,0x2CC1,0x6040,0x1FE3,0xC879,0xEDB6,
   0xBED4,0x468D,0xD967,0x4B72,0xDE94,0xD498,0xE8B0,0x4A85,
   0x6BBB,0x2AC5,0xE54F,0x16ED,0xC586,0xD79A,0x5566,0x9411,
   0xCF8A,0x10E9,0x0604,0x81FE,0xF0A0,0x4478,0xBA25,0xE34B,
   0xF3A2,0xFE5D,0xC080,0x8A05,0xAD3F,0xBC21,0x4870,0x04F1,
   0xDF63,0xC177,0x75AF,0x6342,0x3020,0x1AE5,0x0EFD,0x6DBF,
   0x4C81,0x1418,0x3526,0x2FC3,0xE1BE,0xA235,0xCC88,0x392E,
```

```
    0x5793,0xF255,0x82FC,0x477A,0xACC8,0xE7BA,0x2B32,0x95E6,
    0xA0C0,0x9819,0xD19E,0x7FA3,0x6644,0x7E54,0xAB3B,0x830B,
    0xCA8C,0x29C7,0xD36B,0x3C28,0x79A7,0xE2BC,0x1D16,0x76AD,
    0x3BDB,0x5664,0x4E74,0x1E14,0xDB92,0x0A0C,0x6C48,0xE4B8,
    0x5D9F,0x6EBD,0xEF43,0xA6C4,0xA839,0xA431,0x37D3,0x8BF2,
    0x32D5,0x438B,0x596E,0xB7DA,0x8C01,0x64B1,0xD29C,0xE049,
    0xB4D8,0xFAAC,0x07F3,0x25CF,0xAFCA,0x8EF4,0xE947,0x1810,
    0xD56F,0x88F0,0x6F4A,0x725C,0x2438,0xF157,0xC773,0x5197,
    0x23CB,0x7CA1,0x9CE8,0x213E,0xDD96,0xDC61,0x860D,0x850F,
    0x90E0,0x427C,0xC471,0xAACC,0xD890,0x0506,0x01F7,0x121C,
    0xA3C2,0x5F6A,0xF9AE,0xD069,0x9117,0x5899,0x273A,0xB927,
    0x38D9,0x13EB,0xB32B,0x3322,0xBBD2,0x70A9,0x8907,0xA733,
    0xB62D,0x223C,0x9215,0x20C9,0x4987,0xFFAA,0x7850,0x7AA5,
    0x8F03,0xF859,0x8009,0x171A,0xDA65,0x31D7,0xC684,0xB8D0,
    0xC382,0xB029,0x775A,0x111E,0xCB7B,0xFCA8,0xD66D,0x3A2C,
  }
};
```

8.3.2.5.2 Phase 1 Definition (Figure 43l)

The inputs to Phase 1 of the temporal key mixing function shall be a temporal key (*TK*), the TA, and the TSC. The temporal key shall be 128 bits in length. Only the 32 MSBs of the TSC and all of the temporal key are used in Phase 1. The output, *TTAK*, shall be 80 bits in length and is represented by an array of 16-bit values: $TTAK_0$ $TTAK_1$ $TTAK_2$ $TTAK_3$ $TTAK_4$.

The description of the Phase 1 algorithm treats all of the following values as arrays of 8-bit values: $TA_0..TA_5$, $TK_0..TK_{15}$. The *TA* octet order is represented according to the conventions from 7.1.1, and the first 3 octets represent the OUI.

The XOR (⊕) operation, the bit-wise-and (&) operation, and the addition (+) operation are used in the Phase 1 specification. A loop counter, *i*, and an array index temporary variable, *j*, are also employed.

One function, *Mk*16, is used in the definition of Phase 1. The function *Mk*16 constructs a 16-bit value from two 8-bit inputs as $Mk16(X,Y) = (256 \cdot X)+Y$.

Two steps make up the Phase 1 algorithm. The first step initializes *TTAK* from *TSC* and *TA*. The second step uses an S-box to iteratively mix the keying material into the 80-bit *TTAK*. The second step sets the PHASE1_LOOP_COUNT to 8.

Input: transmit address *TA*0... *TA*5, Temporal Key *TK*0..*TK*15, and *TSC*0..*TSC*5
Output: intermediate key *TTAK*0..*TTAK*4
PHASE1-KEY-MIXING(*TA*0... *TA*5, *TK*0..*TK*15, *TSC*0..*TSC*5)
    PHASE1_STEP1:
    *TTAK*0 ← *MK16(TSC3, TSC2)*
    *TTAK*1 ← *MK16(TSC5, TSC4)*
    *TTAK*2 ← *Mk*16(*TA*1, *TA*0)
    *TTAK*3 ← *Mk*16(*TA*3, *TA*2)
    *TTAK*4 ← *Mk*16(*TA*5, *TA*4)
    PHASE1_STEP2:
    for *i* = 0 to PHASE1_LOOP_COUNT-1
        j ← 2·(*i* & 1)
        *TTAK*0 ← *TTAK*0 + S[*TTAK*4 ⊕ *Mk*16(*TK*1+*j*, *TK*0+*j*)]
        *TTAK*1 ← *TTAK*1 + S[*TTAK*0 ⊕ *Mk*16(*TK*5+*j*, *TK*4+*j*)]
        *TTAK*2 ← *TTAK*2 + S[*TTAK*1 ⊕ *Mk*16(*TK*9+*j*, *TK*8+*j*)]
        *TTAK*3 ← *TTAK*3 + S[*TTAK*2 ⊕ *Mk*16(*TK*13+*j*, *TK*12+*j*)]
        *TTAK*4 ← *TTAK*4 + S[*TTAK*3 ⊕ *Mk*16(*TK*1+*j*, *TK*0+*j*)] + *i*

Figure 43l—Phase 1 key mixing

NOTES

1—The TA is mixed into the temporal key in Phase 1 of the hash function. Implementations can achieve a significant performance improvement by caching the output of Phase 1. The Phase 1 output is the same for 2^{16} = 65 536 consecutive frames from the same temporal key and TA. Consider the simple case where a STA communicates only with an AP. The STA will perform Phase 1 using its own address, and the *TTAK* will be used to protect traffic sent to the AP. The STA will perform Phase 1 using the AP address, and it will be used to unwrap traffic received from the AP.

2—The cached *TTAK* from Phase 1 will need to be updated when the lower 16 bits of the TSC wrap and the upper 32 bits need to be updated.

8.3.2.5.3 Phase 2 definition (see Figure 43m)

The inputs to Phase 2 of the temporal key mixing function shall be the output of Phase 1 (*TTAK*) together with the temporal key and the TSC. The *TTAK* is 80-bits in length. Only the 16 LSBs of the TSC are used in Phase 2. The temporal key is 128 bits. The output is the WEP seed, which is a per-frame key, and is 128 bits in length. The constructed WEP seed has an internal structure conforming to the WEP specification. In other words, the first 24 bits of the WEP seed shall be transmitted in plaintext as the WEP IV. As such, these 24 bits are used to convey lower 16 bits of the TSC from the sender (encryptor) to the receiver (decryptor). The rest of the TSC shall be conveyed in the Extended IV field. The temporal key and *TTAK* values are represented as in Phase 1. The WEP seed is treated as an array of 8-bit values: $WEPSeed_0 \ldots WEPSeed_{15}$. The TSC shall be treated as an array of 8-bit values: $TSC_0\ TSC_1\ TSC_2\ TSC_3\ TSC_4\ TSC_5$.

The pseudo-code specifying the Phase 2 mixing function employs one variable: *PPK*, which is 96 bits long. The *PPK* is represented as an array of 16-bit values: $PPK_0..PPK_5$. The pseudo-code also employs a loop counter, *i*. As detailed in this subclause, the mapping from the 16-bit *PPK* values to the 8-bit *WEPseed* values is explicitly little endian to match the endian architecture of the most common processors used for this application.

The XOR (⊕) operation, the addition (+) operation, the AND (&) operation, the OR (|) operation, and the right bit shift (>>) operation are used in the specification of Phase 2.

The algorithm specification relies on four functions:

— The first function, *Lo*8, references the 8 LSBs of the 16-bit input value.

— The second function, *Hi*8, references the 8 MSBs of the 16-bit value.

— The third function, *RotR*1, rotates its 16-bit argument 1 bit to the right.

— The fourth function, *Mk*16, is already used in Phase 1, defined by $Mk16(X,Y) = (256 \cdot X)+Y$, and constructs a 16-bit output from two 8-bit inputs.

NOTE—The rotate and addition operations in STEP2 make Phase 2 particularly sensitive to the endian architecture of the processor, although the performance degradation due to running this algorithm on a big endian processor should be minor.

Phase 2 comprises three steps:

— STEP1 makes a copy of *TTAK* and brings in the TSC.

— STEP2 is a 96-bit bijective mixing, employing an S-box.

— STEP3 brings in the last of the temporal key *TK* bits and assigns the 24-bit WEP IV value.

The WEP IV format carries 3 octets. STEP3 of Phase 2 determines the value of each of these three octets. The construction was selected to preclude the use of known RC4 weak keys. The recipient can reconstruct the 16 LSBs of the TSC used by the originator by concatenating the third and first octets, ignoring the second octet. The remaining 32 bits of the TSC are obtained from the Extended IV field.

```
Input: intermediate key TTAK0... TTAK4, TK, and TKIP sequence counter TSC
Output: WEP Seed WEPSeed0...WEPSeed15
PHASE2-KEY-MIXING(TTAK0... TTAK4, TK0... TK15, TSC0... TSC5)
        PHASE2_STEP1:
                PPK0 ← TTAK0
                PPK1 ← TTAK1
                PPK2 ← TTAK2
                PPK3 ← TTAK3
                PPK4 ← TTAK4
                PPK5 ← TTAK4 + Mk16(TSC1, TSC0)
        PHASE2_STEP2:
                PPK0 ← PPK0 + S[PPK5 ⊕ Mk16(TK1, TK0)]
                PPK1 ← PPK1 + S[PPK0 ⊕ Mk16(TK3, TK2)]
                PPK2 ← PPK2 + S[PPK1 ⊕ Mk16(TK5, TK4)]
                PPK3 ← PPK3 + S[PPK2 ⊕ Mk16(TK7, TK6)]
                PPK4 ← PPK4 + S[PPK3 ⊕ Mk16(TK9, TK8)]
                PPK5 ← PPK5 + S[PPK4 ⊕ Mk16(TK11, TK10)]
                PPK0 ← PPK0 + RotR1(PPK5 ⊕ Mk16(TK13, TK12))
                PPK1 ← PPK1 + RotR1(PPK0 ⊕ Mk16(TK15, TK14))
                PPK2 ← PPK2 + RotR1(PPK1)
                PPK3 ← PPK3 + RotR1(PPK2)
                PPK4 ← PPK4 + RotR1(PPK3)
                PPK5 ← PPK5 + RotR1(PPK4)
        PHASE2_STEP3:
                WEPSeed0 ← TSC1
                WEPSeed1 ← (TSC1 | 0x20) & 0x7F
                WEPSeed2 ← TSC0
                WEPSeed3 ← Lo8((PPK5 ⊕ Mk16(TK1, TK0)) >> 1)
                for i = 0 to 5
                        WEPSeed4+(2·i) ← Lo8(PPKi)
                        WEPSeed5+(2·i) ← Hi8(PPKi)
                end
        return WEPSeed0...WEPSeed15
```

Figure 43m—Phase 2 key mixing

8.3.2.6 TKIP replay protection procedures

TKIP implementations shall use the TSC field to defend against replay attacks by implementing the following rules:

a) Each MPDU shall have a unique TKIP TSC value.

b) Each transmitter shall maintain a single TSC (48 bit counter) for each PTKSA, GTKSA, and STAKeySA.

c) The TSC shall be implemented as a 48-bit monotonically incrementing counter, initialized to 1 when the corresponding TKIP temporal key is initialized or refreshed.

d) The WEP IV format carries the 16 LSBs of the 48-bit TSC, as defined by the TKIP mixing function (Phase 2, STEP3). The remainder of the TSC is carried in the Extended IV field.

e) A receiver shall maintain a separate set of TKIP TSC replay counters for each PTKSA, GTKSA, and STAKeySA.

f) TKIP replay detection takes place after the MIC verification and any reordering required by ACK processing. Thus, a receiver shall delay advancing a TKIP TSC replay counter until an MSDU passes the MIC check, to prevent attackers from injecting MPDUs with valid ICVs and TSCs, but invalid MICs.

NOTE—This works because if an attacker modifies the TSC, then the encryption key is modified and hence both the ICV and MIC will ordinarily decrypt incorrectly, causing the received MPDU to be dropped.

g) For each PTKSA, GTKSA, and STAKeySA, the receiver shall maintain a separate replay counter for each frame priority and shall use the TSC recovered from a received frame to detect replayed frames, subject to the limitations on the number of supported replay counters indicated in the RSN Capabilities field, as described in 7.3.2.25. A replayed frame occurs when the TSC extracted from a received frame is less than or equal to the current replay counter value for the frame's priority. A transmitter shall not reorder frames with different priorities without ensuring that the receiver supports the required number of replay counters. The transmitter shall not reorder frames within a replay counter, but may reorder frames across replay counters. One possible reason for reordering frames is the IEEE 802.11 MSDU priority.

   IEEE 802.11 does not define a method to signal frame priority.

h) A receiver shall discard any MPDU that is received out of order and shall increment the value of `dot11RSNAStatsTKIPReplays` for this key.

8.3.3 CTR with CBC-MAC Protocol (CCMP)

This subclause specifies the CCMP, which provides confidentiality, authentication, integrity, and replay protection. CCMP is mandatory for RSN compliance.

8.3.3.1 CCMP overview

CCMP is based on the CCM of the AES encryption algorithm. CCM combines CTR for confidentiality and CBC-MAC for authentication and integrity. CCM protects the integrity of both the MPDU Data field and selected portions of the IEEE 802.11 MPDU header.

The AES algorithm is defined in FIPS PUB 197. All AES processing used within CCMP uses AES with a 128-bit key and a 128-bit block size.

CCM is defined in IETF RFC 3610. CCM is a generic mode that can be used with any block-oriented encryption algorithm. CCM has two parameters (M and L), and CCMP uses the following values for the CCM parameters:

— $M = 8$; indicating that the MIC is 8 octets.

— $L = 2$; indicating that the Length field is 2 octets, which is sufficient to hold the length of the largest possible IEEE 802.11 MPDU, expressed in octets.

CCM requires a fresh temporal key for every session. CCM also requires a unique nonce value for each frame protected by a given temporal key, and CCMP uses a 48-bit packet number (PN) for this purpose. Reuse of a PN with the same temporal key voids all security guarantees.

Annex H provides a test vector for CCM.

8.3.3.2 CCMP MPDU format

Figure 43n depicts the MPDU when using CCMP.

CCMP processing expands the original MPDU size by 16 octets, 8 octets for the CCMP Header field and 8 octets for the MIC field. The CCMP Header field is constructed from the PN, ExtIV, and Key ID subfields. PN is a 48-bit PN represented as an array of 6 octets. PN5 is the most significant octet of the PN, and PN0 is the least significant. Note that CCMP does not use the WEP ICV.

The ExtIV subfield (bit 5) of the Key ID octet signals that the CCMP Header field extends the MPDU header by a total of 8 octets, compared to the 4 octets added to the MPDU header when WEP is used. The ExtIV bit (bit 5) is always set to 1 for CCMP.

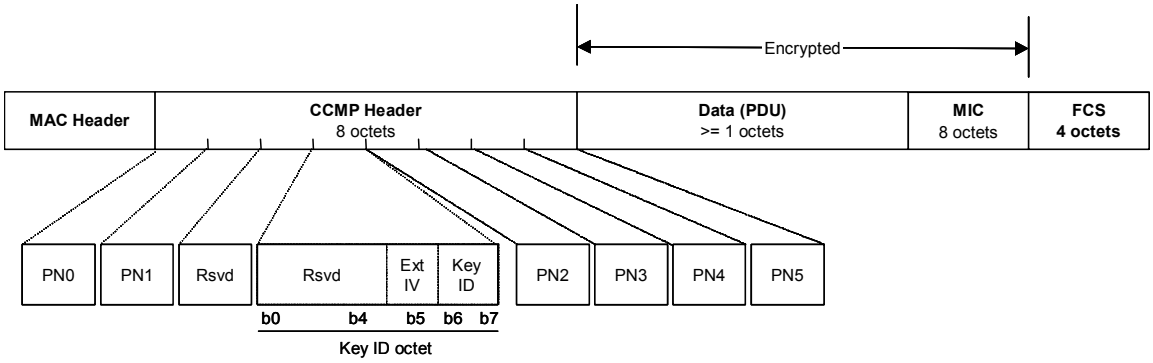


Figure 43n—Expanded CCMP MPDU

Bits 6–7 of the Key ID octet are for the Key ID subfield.

The reserved bits shall be set to 0 and shall be ignored on reception.

8.3.3.3 CCMP encapsulation

The CCMP encapsulation process is depicted in Figure 43o.

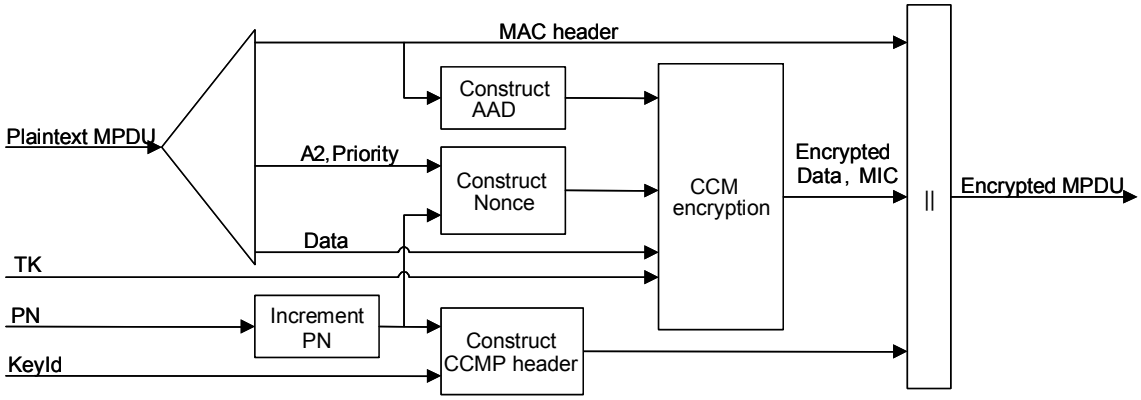


Figure 43o—CCMP encapsulation block diagram

CCMP encrypts the payload of a plaintext MPDU and encapsulates the resulting cipher text using the following steps:

a) Increment the PN, to obtain a fresh PN for each MPDU, so that the PN never repeats for the same temporal key. Note that retransmitted MPDUs are not modified on retransmission.

b) Use the fields in the MPDU header to construct the additional authentication data (AAD) for CCM. The CCM algorithm provides integrity protection for the fields included in the AAD. MPDU header fields that may change when retransmitted are muted by being masked to 0 when calculating the AAD.

c) Construct the CCM Nonce block from the PN, A2, and the Priority field of the MPDU where A2 is MPDU Address 2. The Priority field has a reserved value set to 0.

d) Place the new PN and the key identifier into the 8-octet CCMP header.

e) Use the temporal key, AAD, nonce, and MPDU data to form the cipher text and MIC. This step is known as CCM originator processing.

f) Form the encrypted MPDU by combining the original MPDU header, the CCMP header, the encrypted data and MIC, as described in 8.3.3.2.

The CCM reference describes the processing of the key, nonce, AAD, and data to produce the encrypted output. See 8.3.3.3.1 through 8.3.3.3.5 for details of the creation of the AAD and nonce from the MPDU and the associated MPDU-specific processing.

8.3.3.3.1 PN processing

The PN is incremented by a positive number for each MPDU. The PN shall never repeat for a series of encrypted MPDUs using the same temporal key.

8.3.3.3.2 Construct AAD

The format of the AAD is shown in Figure 43p.

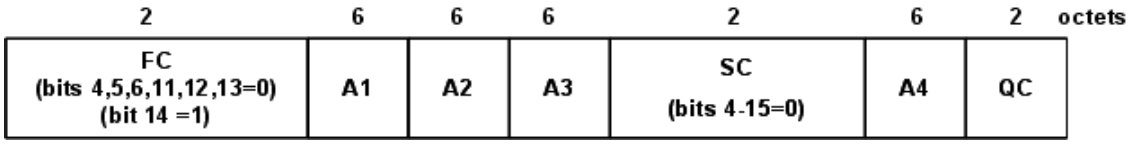


Figure 43p—AAD construction

The AAD is constructed from the MPDU header. The AAD does not include the header Duration field, because the Duration field value can change due to normal IEEE 802.11 operation (e.g., a rate change during retransmission). For similar reasons, several subfields in the Frame Control field are masked to 0. AAD construction is performed as follows:

a) FC – MPDU Frame Control field, with
   1) Subtype bits (bits 4 5 6) masked to 0
   2) Retry bit (bit 11) masked to 0
   3) PwrMgt bit (bit 12) masked to 0
   4) MoreData bit (bit 13) masked to 0
   5) Protected Frame bit (bit 14) always set to 1
b) A1 – MPDU Address 1 field.
c) A2 – MPDU Address 2 field.
d) A3 – MPDU Address 3 field.
e) SC – MPDU Sequence Control field, with the Sequence Number subfield (bits 4–15 of the Sequence Control field) masked to 0. The Fragment Number subfield is not modified.
f) A4 – MPDU Address field, if present in the MPDU.
g) QC – Quality of Service Control field, if present, a 2-octet field that includes the MSDU priority; this field is reserved for future use.

The length of the AAD is 22 octets when no A4 field and no QC field exist and 28 octets long when the MPDU includes the A4 field.

8.3.3.3.3 Construct CCM nonce

The Nonce field occupies 13 octets, and its structure is shown in Figure 43q.

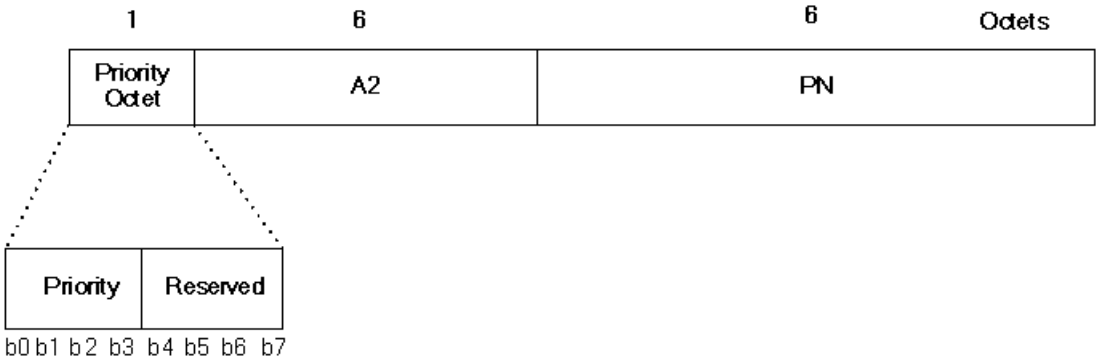


Figure 43q—Nonce construction

The Nonce field has an internal structure of Priority Octet || A2 || PN ("||" is concatenation), where

— This Priority Octet field shall be 0 and reserved for future use with IEEE 802.11 frame prioritization.

— MPDU address A2 field occupies octets 1–6. This shall be encoded with the octets ordered with A2 octet 0 at octet index 1 and A2 octet 5 at octet index 6.

— The PN field occupies octets 7–12. The octets of PN shall be ordered so that PN0 is at octet index 12 and PN5 is at octet index 7.

8.3.3.3.4 Construct CCMP header

The format of the 8-octet CCMP header is given in 8.3.3.2. The header encodes the PN, Key ID, and ExtIV field values used to encrypt the MPDU.

8.3.3.3.5 CCM originator processing

CCM is a generic authenticate-and-encrypt block cipher mode, and in this amendment, CCM is used with the AES block cipher.

There are four inputs to CCM originator processing:

a) *Key:* the temporal key (16 octets).

b) *Nonce:* the nonce (13 octets) constructed as described in 8.3.3.3.3.

c) *Frame body:* the frame body of the MPDU (1–2296 octets; 2296 = 2312 – 8 MIC octets – 8 CCMP header octets).

d) *AAD:* the AAD (22–30 octets) constructed from the MPDU header as described in 8.3.3.3.2.

The CCM originator processing provides authentication and integrity of the frame body and the AAD as well as confidentiality of the frame body. The output from the CCM originator processing consists of the encrypted data and 8 additional octets of encrypted MIC (see Figure 43n).

8.3.3.4 CCMP decapsulation

Figure 43r depicts the CCMP decapsulation process.

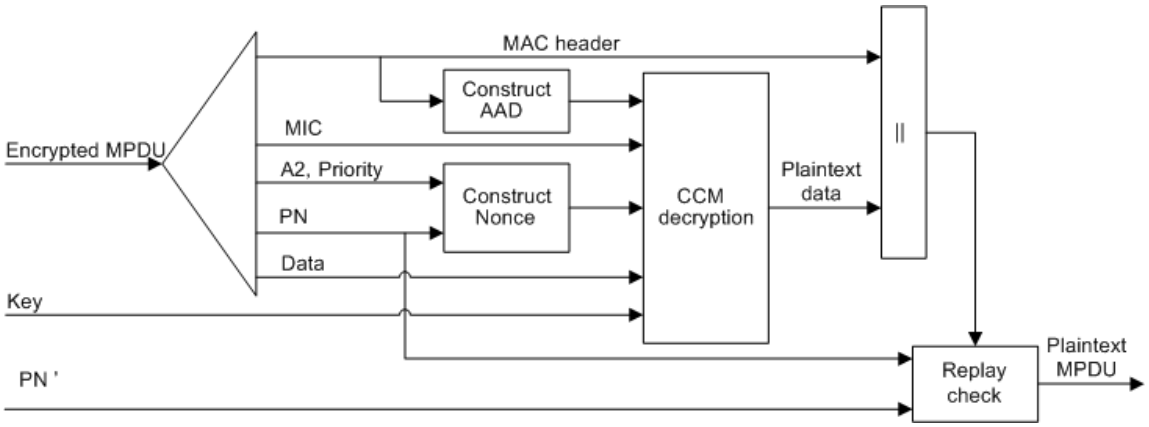


Figure 43r—CCMP decapsulation block diagram

CCMP decrypts the payload of a cipher text MPDU and decapsulates a plaintext MPDU using the following steps:

a) The encrypted MPDU is parsed to construct the AAD and nonce values.

b) The AAD is formed from the MPDU header of the encrypted MPDU.

c) The nonce value is constructed from the A2, PN, and Priority Octet fields (reserved and set to 0).

d) The MIC is extracted for use in the CCM integrity checking.

e) The CCM recipient processing uses the temporal key, AAD, nonce, MIC, and MPDU cipher text data to recover the MPDU plaintext data as well as to check the integrity of the AAD and MPDU plaintext data.

f) The received MPDU header and the MPDU plaintext data from the CCM recipient processing may be concatenated to form a plaintext MPDU.

g) The decryption processing prevents replay of MPDUs by validating that the PN in the MPDU is greater than the replay counter maintained for the session.

See 8.3.3.4.1 through 8.3.3.4.3 for details of this processing.

8.3.3.4.1 CCM recipient processing

CCM recipient processing must use the same parameters as CCM originator processing.

There are four inputs to CCM recipient processing:

— *Key:* the temporal key (16 octets).

— *Nonce:* the nonce (13 octets) constructed as described in 8.3.3.3.3.

— *Encrypted frame body:* the encrypted frame body from the received MPDU. The encrypted frame body includes an 8-octet MIC (9–2304 octets).

— *AAD:* the AAD (22–30 octets) that is the canonical MPDU header as described in 8.3.3.3.2.

The CCM recipient processing checks the authentication and integrity of the frame body and the AAD as well as decrypting the frame body. The plaintext is returned only if the MIC check is successful.

There is one output from error-free CCM recipient processing:

— *Frame body:* the plaintext frame body, which is 8 octets smaller than the encrypted frame body.

8.3.3.4.2 Decrypted CCMP MPDU

The decapsulation process succeeds when the calculated MIC matches the MIC value obtained from decrypting the received encrypted MPDU. The original MPDU header is concatenated with the plaintext data resulting from the successful CCM recipient processing to create the plaintext MPDU.

8.3.3.4.3 PN and replay detection

To effect replay detection, the receiver extracts the PN from the CCMP header. See 8.3.3.2 for a description of how the PN is encoded in the CCMP header. The following processing rules are used to detect replay:

a) The PN values sequentially number each MPDU.

b) Each transmitter shall maintain a single PN (48-bit counter) for each PTKSA, GTKSA, and STAKeySA.

c) The PN shall be implemented as a 48-bit monotonically incrementing non-negative integer, initialized to 1 when the corresponding temporal key is initialized or refreshed.

d) A receiver shall maintain a separate set of PN replay counters for each PTKSA, GTKSA, and STAKeySA. The receiver initializes these replay counters to 0 when it resets the temporal key for a peer. The replay counter is set to the PN value of accepted CCMP MPDUs.

e) For each PTKSA, GTKSA, and STAKeySA, the recipient shall maintain a separate replay counter for each IEEE 802.11 MSDU priority and shall use the PN recovered from a received frame to detect replayed frames, subject to the limitation of the number of supported replay counters indicated in the RSN Capabilities field (see 7.3.2.25). A replayed frame occurs when the PN extracted from a received frame is less that or equal to the current replay counter value for the frame's MSDU priority. A transmitter shall not use IEEE 802.11 MSDU priorities without ensuring that the receiver supports the required number of replay counters. The transmitter shall not reorder frames within a replay counter, but may reorder frames across replay counters. One possible reason for reordering frames is the IEEE 802.11 MSDU priority.

f) The receiver shall discard MSDUs whose constituent MPDU PN values are not sequential. A receiver shall discard any MPDU that is received with its PN less than or equal to the replay counter and shall increment the value of `dot11RSNAStatsCCMPReplays` for this key.

8.4 RSNA security association management

8.4.1 Security associations

8.4.1.1 Security association definitions

IEEE 802.11 uses the notion of a security association to describe secure operation. Secure communications are possible only within the context of a security association, as this is the context providing the state—cryptographic keys, counters, sequence spaces, etc.—needed for correct operation of the IEEE 802.11 cipher suites.

A security association is a set of policy(ies) and key(s) used to protect information. The information in the security association is stored by each party of the security association, must be consistent among all parties, and must have an identity. The identity is a compact name of the key and other bits of security association information to fit into a table index or an MPDU. There are four types of security associations supported by an RSN STA:

— PMKSA: A result of a successful IEEE 802.lX exchange, preshared PMK information, or PMK cached via some other mechanism.

— PTKSA: A result of a successful 4-Way Handshake.

— GTKSA: A result of a successful Group Key Handshake or successful 4-Way Handshake.

— STAKeySA: A result of a successful STAKey Handshake.

8.4.1.1.1 PMKSA

When the PMKSA is the result of a successful IEEE 802.1X authentication, it is derived from the EAP authentication and authorization parameters provided by the AS. This security association is bidirectional. In other words, both parties use the information in the security association for both sending and receiving. The PMKSA is created by the Supplicant's SME when the EAP authentication completes successfully or the PSK is configured. The PMKSA is created by the Authenticator's SME when the PMK is created from the keying information transferred from the AS or the PSK is configured. The PMKSA is used to create the PTKSA. PMKSAs are cached for up to their lifetimes. The PMKSA consists of the following elements:

— PMKID, as defined in 8.5.1.2. The PMKID identifies the security association.
— Authenticator MAC address.
— PMK.
— Lifetime, as defined in 8.5.1.2.
— AKMP.
— All authorization parameters specified by the AS or local configuration. This can include parameters such as the STA's authorized SSID.

8.4.1.1.2 PTKSA

The PTKSA is a result of the 4-Way Handshake. This security association is also bidirectional. The PTKSA is used to create the key hierarchy. PTKSAs are cached for the life of the PMKSA. Because the PTKSA is tied to the PMKSA, it only has the additional information from the 4-Way Handshake. There shall be only one PTKSA with the same Supplicant and Authenticator MAC addresses. There is state created between Message 1 and Message 3 of a 4-Way Handshake. This does not create a PTKSA until Message 3 is validated on the Supplicant and Message 4 is validated by the Authenticator. The PTKSA consists of the following elements:

— PTK
— Pairwise cipher suite selector
— Supplicant MAC address
— Authenticator MAC address

8.4.1.1.3 GTKSA

The GTKSA results from a successful 4-Way Handshake or the Group Key Handshake and is unidirectional. In an ESS, there is one GTKSA, used exclusively for encrypting broadcast/multicast MPDUs that are transmitted by the AP and for decrypting broadcast/multicast transmissions that are received by the STAs. In an IBSS, each STA defines its own GTKSA, which is used to encrypt its broadcast/multicast transmissions, and stores a separate GTKSA for each peer STA so that encrypted broadcast/multicast traffic received from other STAs may be decrypted. A GTKSA is created by the Supplicant's SME when Message 3 of the 4-Way Handshake is received or when Message 1 of the Group Key Handshake is received. The GTKSA is created by the Authenticator's SME when the SME changes the GTK and has sent the GTK to all STAs with which it has a PTKSA. A GTKSA consists of the following elements:

— Direction vector (whether the GTK is used for transmit or receive).
— Group cipher suite selector.
— GTK.
— Authenticator MAC address.
— All authorization parameters specified by local configuration. This can include parameters such as the STA's authorized SSID.

When the GTK is used to encrypt unicast traffic (the selectable cipher suite is "Use group key"), the GTKSA is bidirectional.

8.4.1.1.4 STAKeySA

The STAKeySA is a result of the STAKey Handshake. This security association is unidirectional from the initiator to the peer. There shall be only one STAKeySA with the same initiator and peer MAC addresses. Creation of a new STAKeySA with the same initiator and peer MAC addresses will cause deletion of the existing STAKeySA. The STAKeySA is created when Message 1 of the STAKey Handshake is validated. The STAKeySA consists of the following elements:

— STAKey
— Pairwise cipher suite selector
— Initiator MAC address
— Peer MAC address

8.4.1.2 Security association life cycle

A STA can operate in either an ESS or in an IBSS, and a security association has a distinct life cycle for each.

8.4.1.2.1 Security association in an ESS

In an ESS there are two cases:

— Initial contact between the STA and the ESS
— Roaming by the STA within the ESS

A STA and AP establish an initial security association via the following steps:

a) The STA selects an authorized ESS by selecting among APs that advertise an appropriate SSID.
b) The STA then uses IEEE 802.11 Open System authentication followed by association to the chosen AP. Negotiation of security parameters takes place during association.

   NOTES

   1—It is possible for more than one PMKSA to exist. As an example, a second PMKSA may come into existence through PMKSA caching. A STA might leave the ESS and flush its cache. Before its PMKSA expires in the AP's cache, the STA returns to the ESS and establishes a second PMKSA from the AP's perspective.

   2—An attack altering the security parameters will be detected by the key derivation procedure.

   3—IEEE 802.11 Open System authentication provides no security, but is included to maintain backward compatibility with the IEEE 802.11 state machine (see 5.5).

c) The AP's Authenticator or the STA's Supplicant initiates IEEE 802.1X authentication. The EAP method used by IEEE 802.1X will support mutual authentication, as the STA needs assurance that the AP is a legitimate AP.

   NOTES

   1—Prior to the completion of IEEE 802.1X authentication and the installation of keys, the IEEE 802.1X Controlled Port in the AP will block all data frames. The IEEE 802.1X Controlled Port returns to the unauthorized state and blocks all data frames before invocation of an MLME-DELETEKEYS.request primitive. The IEEE 802.1X Uncontrolled Port allows IEEE 802.1X frames to pass between the Supplicant and Authenticator. Although IEEE 802.1X does not require a Supplicant Controlled Port, this amendment assumes that the Supplicant has a Controlled Port in order to provide the needed level of security. Supplicants without a Controlled Port compromise RSN security and should not be used.

   2—Any secure network cannot support promiscuous association, e.g., an unsecured operation of IEEE 802.11. A trust relationship must exist between the STA and the AS of the targeted SSID prior to association and secure operation, in order for the association to be trustworthy. The reason is that an attacker can deploy a rogue AP

just as easily as a legitimate network provider can deploy a legitimate AP, so some sort of prior relationship is necessary to establish credentials between the ESS and the STA.

d) The last step is key management. The authentication process creates cryptographic keys shared between the IEEE 802.1X AS and the STA. The AS transfers these keys to the AP, and the AP and STA use one key confirmation handshake, called the 4-Way Handshake, to complete security association establishment. The key confirmation handshake indicates when the link has been secured by the keys and is ready to allow normal data traffic.

A STA roaming within an ESS establishes a new PMKSA by one of three schemes:

— In the case of (re)association followed by IEEE 802.1X or PSK authentication, the STA repeats the same actions as for an initial contact association, but its Supplicant also deletes the PTKSA when it roams from the old AP. The STA's Supplicant also deletes the PTKSA when it disassociates/deauthenticates from all basic service set identifiers (BSSIDs) in the ESS.

— A STA (AP) can retain PMKs for APs (STAs) in the ESS to which it has previously performed a full IEEE 802.1X authentication. If a STA wishes to roam to an AP for which it has cached one or more PMKSAs, it can include one or more PMKIDs in the RSN information element of its (Re)Association Request frame. An AP whose Authenticator has retained the PMK for one or more of the PMKIDs can skip the 802.1X authentication and proceed with the 4-Way Handshake. The AP shall include the PMKID of the selected PMK in Message 1 of the 4-Way Handshake. If none of the PMKIDs of the cached PMKSAs matches any of the supplied PMKIDs, then the Authenticator shall perform another IEEE 802.1X authentication. Similarly, if the STA fails to send a PMKID, the STA and AP must perform a full IEEE 802.1X authentication.

— A STA already associated with the ESS can request its IEEE 802.1X Supplicant to authenticate with a new AP before associating to that new AP. The normal operation of the DS via the old AP provides the communication between the STA and the new AP. The STA's IEEE 802.11 management entity delays reassociation with the new AP until IEEE 802.1X authentication completes via the DS. If IEEE 802.1X authentication completes successfully, then PMKSAs shared between the new AP and the STA will be cached, thereby enabling the possible usage of reassociation without requiring a subsequent full IEEE 802.1X authentication procedure.

The MLME-DELETEKEYS.request primitive destroys the temporal keys established for the security association so that they cannot be used to protect subsequent IEEE 802.11 traffic. A STA's SME uses this primitive when it deletes a PTKSA or GTKSA.

8.4.1.2.2 Security association in an IBSS

In an IBSS, when a STA's SME establishes a security association with a peer STA, it creates both an IEEE 802.1X Supplicant and Authenticator for the peer.

A STA can receive IEEE 802.1X messages from a previously unknown MAC address.

Any STA within an IBSS may decline to form a security association with a STA joining the IBSS. An attempt to form a security association may also fail because, for example, the peer uses a different PSK from what the STA expects.

In an IBSS each STA defines its own group key, i.e., GTK, to secure its broadcast/multicast transmissions. Each STA shall use either the 4-Way Handshake or the Group Key Handshake to distribute its transmit GTK to its new peer STA. When the STA generates a new GTK, it also uses the Group Key Handshake to distribute the new GTK to each established peer.

8.4.2 RSNA selection

A STA prepared to establish RSNAs shall advertise its capabilities by including the RSN information element in Beacon and Probe Response messages. The included RSN information element shall specify all the authentication and cipher suites enabled by the STA's policy. A STA shall not advertise any authentication or cipher suite that is not enabled.

The STA's IEEE 802.11 management entity shall utilize the MLME-SCAN.request primitive to identify neighboring STAs that assert robust security and advertise an SSID identifying an authorized ESS or IBSS. A STA may decline to communicate with STAs that fail to advertise an RSN information element in their Beacon and Probe Response frames or that do not advertise an authorized SSID. A STA may also decline to communicate with other STAs that do not advertise authorized authentication and cipher suites within their RSN information elements.

A STA shall advertise the same RSN information element in both its Beacon and Probe Response frames.

NOTES

1—Whether a STA with robust security enabled may attempt to communicate with a STA that does not include the RSN information element is a matter of policy.

2—As a practical matter, if maximal interoperability is a goal, an AP will support TKIP as well as CCMP.

A STA shall observe the following rules when processing an RSN information element:

— A STA shall advertise the highest version it supports.

— A STA shall request the highest Version field value it supports that is less than or equal to the version advertised by the peer STA.

— Two peer STAs without overlapping supported Version field values shall not use RSNA methods to secure their communication.

— A STA shall ignore suite selectors that it does not recognize.

8.4.3 RSNA policy selection in an ESS

RSNA policy selection in an ESS utilizes the normal IEEE 802.11 association procedure. RSNA policy selection is performed by the associating STA. The STA does this by including an RSN information element in its (Re)Association Requests.

In an RSN, an AP shall not associate with pre-RSNA STAs, i.e., with STAs that fail to include the RSN information element in the Association or Reassociation Request frame.

The STA's SME initiating an association shall insert an RSN information element into its (Re)Association Request; via the MLME-ASSOCIATE.request primitive, when the targeted AP indicates RSNA support. The initiating STA's RSN information element shall include one authentication and pairwise cipher suite from among those advertised by the targeted AP in its Beacon and Probe Response frames. It shall also specify the group cipher suite specified by the targeted AP. If at least one RSN information element field from the AP's RSN information element fails to overlap with any value the STA supports, the STA shall decline to associate with that AP.

If an RSNA-capable AP receives a (Re)Association Request including an RSN information element and if it chooses to accept the association as a secure association, then it shall use the authentication and pairwise cipher suites in the (Re)Association Request, unless the AP includes an optional second RSN information element in Message 3 of the 4-Way Handshake. If the second RSN information element is supplied in Message 3, then the pairwise cipher suite used by the security association, if established, shall be the pairwise cipher from the second RSN information element.

In order to accommodate local security policy, a STA may choose not to associate with an AP that does not support any pairwise cipher suites. An AP indicates that it does not support any pairwise keys by advertising "Use group key" as the pairwise cipher suite selector.

NOTE—When an ESS uses PSKs, STAs negotiate a pairwise cipher. However, any STA in the ESS can derive the pairwise keys of any other that uses the same PSK by capturing the first two messages of the 4-Way Handshake. This provides malicious insiders with the ability to eavesdrop as well as the ability to establish a man-in-the-middle attack.

8.4.3.1 TSN policy selection in an ESS

In a TSN, an RSN STA shall include the RSN information element in its (Re)Association Requests.

An RSNA-capable AP configured to operate in a TSN shall include the RSN information element and may associate with both RSNA and pre-RSNA STAs. In other words, an RSNA-capable AP shall respond to an associating STA that includes the RSN information element just as in an RSN.

If an AP operating within a TSN receives a (Re)Association Request without an RSN information element, its IEEE 802.1X Controlled Port shall initially be blocked. The SME shall unblock the IEEE 802.1X Controlled Port when WEP has been enabled.

8.4.4 RSNA policy selection in an IBSS

In an IBSS, all STAs must use a single group cipher suite, and all STAs must support a common subset of pairwise cipher suites. However, the SMEs of any pair of STAs may negotiate to use any common pairwise cipher suite they both support. Each STA shall include the group cipher suite and its list of pairwise cipher suites in its Beacon and Probe Response messages. Two STAs may only establish a PMKSA if they have advertised the same group cipher suite. Similarly, the two STAs shall not establish a PMKSA if the STAs have advertised disjoint sets of pairwise cipher suites.

When an IBSS STA's SME wants to set up a security association with a peer STA, but does not know the peer's policy, it must first obtain the peer's security policy using a Probe Request frame. The SME entities of the two STAs select the pairwise cipher suites using one of the 4-Way Handshakes. The SMEs of each pair of STAs within an IBSS may use the EAPOL-Key 4-Way Handshake to select a pairwise cipher suite. As specified in 8.5.2, Message 2 and Message 3 of the 4-Way Handshake convey an RSN information element. The Message 2 RSN information element includes the selected pairwise cipher suite, and Message 3 includes the RSN information element that the STA would send in a Probe Response frame.

The pair of STAs shall use the pairwise cipher suite specified in Message 3 of the 4-Way Handshake sent by the Authenticator STA with the higher MAC address (see 8.5.1).

The SME shall check that the group cipher suite and AKMP match those in the Beacon and Probe Response frames for the IBSS.

NOTES

1—The RSN information elements in Message 2 and Message 3 are not the same as in the Beacon frame. The group cipher and AKMP are the same, but the pairwise ciphers may differ because Beacon frames from different STAs may advertise different pairwise ciphers. Thus, STAs in an IBSS use the same AKM suite and group cipher, while different pairwise ciphers can be used between STA pairs.

2—When an IBSS network uses PSKs, STAs can negotiate a pairwise cipher. However, any STA in the IBSS can derive the PTKs of any other that uses the same PSK by capturing the first two messages of the 4-Way Handshake. This provides malicious insiders with the ability to eavesdrop as well as the ability to establish a man-in-the-middle attack.

8.4.4.1 TSN policy selection in an IBSS

Pre-RSNA STAs generate Beacon and Probe Response frames without an RSN information element and will ignore the RSN information element because it is unknown to them. This allows an RSNA STA to identify the pre-RSNA STAs from which it has received Beacon and Probe Response frames.

If an RSNA STA's SME instead identifies a possible IBSS member on the basis of a received broadcast/multicast message, via MLME-PROTECTEDFRAMEDROPPED.indication primitive, it cannot identify the peer's security policy directly. The SME can attempt to obtain the peer STA's security policy via a Probe Request frame.

8.4.5 RSN management of the IEEE 802.1X Controlled Port

When the policy selection process chooses IEEE 802.1X authentication, this amendment assumes that IEEE 802.1X Supplicants and Authenticators exchange protocol information via the IEEE 802.1X Uncontrolled port. The IEEE 802.1X Controlled Port is blocked from passing general data traffic between the STAs until an IEEE 802.1X authentication procedure completes successfully over the IEEE 802.1X Uncontrolled Port. The security of an RSNA depends on this assumption being true.

In an ESS, the STA indicates the IEEE 802.11 link is available by invoking the MLME-ASSOCIATE.confirm or MLME-REASSOCIATE.confirm primitive. This signals the Supplicant that the MAC has transitioned from the disabled to enabled state. At this point, the Supplicant's Controlled Port is blocked, and communication of all non-IEEE 802.1X MSDUs sent or received via the port is not authorized.

In an ESS, the AP indicates that the IEEE 802.11 link is available by invoking the MLME-ASSOCIATE.indication or MLME-REASSOCIATE.indication primitive. At this point the Authenticator's Controlled Port corresponding to the STA's association is blocked, and communication of all non-IEEE 802.1X MSDUs sent or received via the Controlled Port is not authorized.

In an IBSS, the STA shall block all IEEE 802.1X ports at initialization. Communication of all non-IEEE 802.1X MSDUs sent or received via the Controlled Port is not authorized.

This amendment assumes each Controlled Port remains blocked until the IEEE 802.1X state variables portValid and keyDone both become true. This assumption means that the IEEE 802.1X Controlled Port discards MSDUs sent across the IEEE 802.11 channel prior to the installation of cryptographic keys into the MAC. This protects the STA's host from forged MSDUs written to the channel while it is still being initialized.

The MAC does not distinguish between MSDUs for the Controlled Port, and MSDUs for the Uncontrolled Port. In other words, IEEE 802.1X EAPOL frames will only be encrypted after invocation of the MLME-SETPROTECTION.request primitive.

This amendment assumes that IEEE 802.1X does not block the Controlled Port when authentication is triggered through reauthentication. During IEEE 802.1X reauthentication, an existing RSNA can protect all MSDUs exchanged between the STAs. Blocking MSDUs is not required during reauthentication over an RSNA.

8.4.6 RSNA authentication in an ESS

When establishing an RSNA, a STA shall use IEEE 802.11 Open System authentication prior to (re)association.

IEEE 802.1X authentication is initiated by any one of the following mechanisms:

— If a STA negotiates to use IEEE 802.1X authentication during (re)association, the STA's management entity can respond to the MLME-ASSOCIATE.confirm (or indication) primitive by requesting the STA's Supplicant (or AP's Authenticator) to initiate IEEE 802.1X authentication. Thus, in this case, authentication is driven by the STA's decision to associate and the AP's decision to accept the association.

— If a STA's MLME-SCAN.confirm primitive finds another AP within the current ESS, a STA may signal its Supplicant to use IEEE 802.1X to preauthenticate with that AP.

NOTE—A roaming STA's IEEE 802.1X Supplicant may initiate preauthentication by sending an EAPOL-Start message via its old AP, through the DS, to a new AP.

— If a STA receives an IEEE 802.1X message, it delivers this to its Supplicant or Authenticator, which may initiate a new IEEE 802.1X authentication.

8.4.6.1 Preauthentication and RSNA key management

A STA shall not use preauthentication except when pairwise keys are employed. Preauthentication shall not be used unless the new AP advertises the preauthentication capability in the RSN information element.

When preauthentication is used, then

a) Authentication is independent of roaming.

b) The STA's Supplicant may authenticate with multiple APs at a time.

NOTE—Preauthentication can be useful as a performance enhancement, as reassociation will not include the protocol overhead of a full reauthentication when it is used.

Preauthentication uses the IEEE 802.1X protocol and state machines with EtherType 88-C7, rather than the EtherType 88-8E. Only IEEE 802.1X frame types EAP-Packet and EAPOL-Start are valid for preauthentication.

NOTE—Some IEEE 802.1X Authenticators may not bridge IEEE 802.1X frames, as suggested in C.1.1 of IEEE P802.1X-REV. Preauthentication uses a distinct EtherType to enable such devices to bridge preauthentication frames.

A STA's Supplicant can initiate preauthentication when it has completed the 4-Way Handshake and configured the required temporal keys. To effect preauthentication, the STA's Supplicant sends an IEEE 802.1X EAPOL-Start message with the DA being the BSSID of a targeted AP and the RA being the BSSID of the AP with which it is associated. The target AP shall use a BSSID equal to the MAC address of its Authenticator. As preauthentication frames do not use the IEEE 802.1X EAPOL EtherType field, the AP with which the STA is currently associated need not apply any special handling. The AP and the MAC in the STA shall handle these frames in the same way as other frames with arbitrary EtherType field values that require distribution via the DS.

An AP's Authenticator that receives an EAPOL-Start message via the DS may initiate IEEE 802.1X authentication to the STA via the DS. The DS will forward this message to the AP with which the STA is associated.

The result of preauthentication may be a PMKSA, if the IEEE 802.1X authentication completes successfully. If preauthentication produces a PMKSA, then, when the Supplicant's STA associates with the preauthenticated AP, the Supplicant can use the PMKSA with the 4-Way Handshake.

Successful completion of EAP authentication over IEEE 802.1X establishes a PMKSA at the Supplicant. The Authenticator has the PMKSA when the AS completes the authentication, passes the keying information (the authentication, authorization, and accounting [AAA] key, a portion of which is the PMK) to the Authenticator, and the Authenticator creates a PMKSA using the PMK. The PMKSA is inserted into the PMKSA cache. Therefore, if the Supplicant and Authenticator lose synchronization with respect to the

PMKSA, the 4-Way Handshake will fail. In such circumstances, the MIB variable `dot11RSNAStats-4WayHandshakeFailures` shall be incremented.

A STA's Supplicant may initiate preauthentication with any AP within its present ESS with preauthentication enabled regardless of whether the targeted AP is within radio range.

Even if a STA has preauthenticated, it is still possible that it may have to undergo a full IEEE 802.1X authentication, as the AP's Authenticator may have purged its PMKSA due to, for example, unavailability of resources, delay in the STA associating, etc.

8.4.6.2 Cached PMKSAs and RSNA key management

A STA can retain PMKSAs it establishes as a result of previous authentication. The PMKSA cannot be changed while cached. The PMK in the PMKSA is used with the 4-Way Handshake to establish fresh PTKs.

If a non-AP STA in an ESS has determined it has a valid PMKSA with an AP to which it is about to (re)associate, it includes the PMKID for the PMKSA in the RSN information element in the (Re)Association Request. Upon receipt of a (Re)Association Request with one or more PMKIDs, an AP checks whether its Authenticator has retained a PMK for the PMKIDs and whether the PMK is still valid. If so, it asserts possession of that PMK by beginning the 4-Way Handshake after association has completed; otherwise it begins a full IEEE 802.1X authentication after association has completed.

If both sides assert possession of a cached PMKSA, but the 4-Way Handshake fails, both sides may delete the cached PMKSA for the selected PMKID.

If a STA roams to an AP with which it is preauthenticating and the STA does not have a PMKSA for that AP, the STA must initiate a full IEEE 802.1X EAP authentication.

8.4.7 RSNA authentication in an IBSS

When authentication is used in an IBSS, it is driven by each STA wishing to establish communications. The management entity of this STA chooses a set of STAs with which it may want to authenticate and then may cause the MAC to send an IEEE 802.11 Open System authentication message to each targeted STA. Candidate STAs can be identified from Beacon frames, Probe Response frames, and data frames from the same BSSID. Before communicating with STAs identified from data frames, the security policy of the STAs may be obtained, e.g., by sending a Probe Request frame to the STA and obtaining a Probe Response frame. Targeted STAs that wish to respond may return an IEEE 802.11 Open System authentication message to the initiating STA.

When IEEE 802.1X authentication is used, the STA management entity will then request its local IEEE 802.1X entity to create a Supplicant PAE for the peer STA. The Supplicant PAE will initiate the authentication to the peer STA by sending an EAPOL-Start message to the peer. The STA management entity will also request its local IEEE 802.1X entity to create an Authenticator PAE for the peer STA on receipt of the EAPOL-Start message. The Authenticator will initiate authentication to the peer STA by sending an EAP-Request message or, if PSK mode is in effect, Message 1 of the 4-Way Handshake.

Upon initial authentication between any pair of STAs, data frames, other than IEEE 802.1X messages, are not allowed to flow between the pair of STAs until both STAs in each pair of STAs have successfully completed AKM and have provided the supplied encryption keys.

Upon the initiation of an IEEE 802.1X reauthentication by any STA of a pair of STAs, data frames will continue to flow between the STAs while authentication completes. Upon a timeout or failure in the authentication process, the Authenticator of the STA initiating the reauthentication shall cause a Deauthentication message to be sent to the Supplicant of the STA targeted for reauthentication. The Deauthentication message

will cause both STAs in the pair of STAs to follow the deauthentication procedure defined in 11.3.3 and 11.3.4.

The IEEE 802.1X reauthentication timers in each STA are independent. If the reauthentication timer of the STA with the higher MAC address (see 8.5.1 for MAC comparison) triggers the reauthentication via its Authenticator, its Supplicant must send an EAPOL-Start message to the authenticator of the STA with the lower MAC address to trigger reauthentication on the other STA. This process keeps the pair of STAs in a consistent state with respect to derivation of fresh temporal keys upon an IEEE 802.1X reauthentication.

When it receives an MLME-AUTHENTICATE.indicate primitive due to an Open System authentication request, the IEEE 802.11 management entity on a targeted STA shall, if it wants to set up a security association with the peer STA, request its Authenticator to begin IEEE 802.1X authentication, i.e., to send an EAP-Request/Identity message or Message 1 of the 4-Way Handshake to the Supplicant.

The EAPOL-Key frame is used to exchange information between the Supplicant and the Authenticator to negotiate a fresh PTK. The 4-Way Handshake produces a single PTK from the PMK. The 4-Way Handshake and Group Key Handshake use the PTK to protect the GTK as it is transferred to the receiving STA.

PSK authentication may also be used in an IBSS. When a single PSK is shared among the IBSS STAs, the STA wishing to establish communication sends 4-Way Handshake Message 1 to the target STA(s). The targeted STA responds to Message 1 with Message 2 of the 4-Way Handshake and begins its 4-Way Handshake by sending Message 1 to the initiating STA. The two 4-Way Handshakes establish PTKSAs and GTKSAs to be used between the initiating STA and the targeted STA. PSK PMKIDs may also be used, enabling support for pairwise PSKs.

The model for security in an IBSS is not general. In particular, it assumes the following:

a) The sets of use cases for which the authentication procedures described in this subclause are valid are as follows:
   1) PSK-based authentication, typically managed by the pass-phrase hash method as described in H.4
   2) EAP-based authentication, using credentials that have been issued and preinstalled on the STAs within a common administrative domain, such as a single organization
b) All of the STAs are in direct radio communication. In particular, there is no routing, bridging, or forwarding of traffic by a third STA to effect communication. This assumption is made, because the model makes no provision to protect IBSS topology information from tampering by one of the members.

8.4.8 RSNA key management in an ESS

When the IEEE 802.1X authentication completes successfully, this amendment assumes that the STA's IEEE 802.1X Supplicant and the IEEE 802.1X AS will share a secret, called a PMK. The AS transfers the PMK, within the AAA key, to the AP, using a technique that is outside the scope of this amendment; the derivation of the PMK from the MSK is EAP-method-specific. With the PMK in place, the AP initiates a key confirmation handshake with the STA. The key confirmation handshake sets the IEEE 802.1X state variable portValid (as described in IEEE P802.1X-REV) to TRUE.

The key confirmation handshake is implemented by the 4-Way Handshake. The purposes of the 4-Way Handshake are as follows:

a) Confirm the existence of the PMK at the peer.
b) Ensure that the security association keys are fresh.
c) Synchronize the installation of temporal keys into the MAC.
d) Transfer the GTK from the Authenticator to the Supplicant.

e) Confirm the selection of cipher suites.

NOTES

1—Message 1 of the 4-Way Handshake can be forged. However, the forgery attempt will be detected in the failure of the 4-Way Handshake.

2—Neither the AP nor the STA can use the PMK for any purpose but the one specified herein without compromising the key. If the AP uses it for another purpose, then the STA can masquerade as the AP; similarly if the STA reuses the PMK in another context, then the AP can masquerade as the STA.

The Supplicant and Authenticator signal the completion of key management by utilizing the MLME-SETKEYS.request primitive to configure the agreed-upon temporal pairwise key into the IEEE 802.11 MAC and by calling the MLME-SETPROTECTION.request primitive to enable its use.

A second key exchange, the Group Key Handshake, is also defined. It distributes a subsequent GTK. The AP's Authenticator can use the Group Key Handshake to update the GTK at the STA's Supplicant. The Group Key Handshake uses the EAPOL-Key frames for this exchange. When it completes, the STA's Supplicant can use the MLME-SETKEYS.request primitive to configure the GTK into the IEEE 802.11 MAC.

8.4.9 RSNA key management in an IBSS

To establish a security association between two STAs in an IBSS, each STA's SME must have an accompanying IEEE 802.1X Authenticator and Supplicant. Each STA's SME initiates the 4-Way Handshake from the Authenticator to the peer STA's Supplicant (see 8.4.7). Two separate 4-Way Handshakes are conducted.

The 4-Way Handshake is used to negotiate the pairwise cipher suites, as described in 8.4.4. The IEEE 802.11 SME configures the temporal key portion of the PTK into the IEEE 802.11 MAC. Each Authenticator uses the KCK and KEK portions of the PTK negotiated by the exchange it initiates to distribute its own GTK. Each Authenticator generates its own GTK and uses either the 4-Way Handshake or the Group Key Handshake to transfer the GTK to other STAs with whom it has completed a 4-Way Handshake. The pairwise key used between any two STAs shall be the pairwise key from the 4-Way Handshake initiated by the STA with the highest MAC address.

A STA joining an IBSS is required to adopt the security configuration of the IBSS, which includes the group cipher suite, pairwise cipher suite, and AKMP (see 8.4.4). The STA shall not set up a security association with any STA having a different security configuration. The Beacon and Probe Response frames of the various STAs within an IBSS must reflect a consistent security policy, as the beacon initiation rotates among the STAs.

A STA joining an IBSS shall support and advertise in the Beacon frame the security configuration of the IBSS, which includes the group cipher suite, advertised pairwise cipher suite, and AKMP (see 8.4.4). The STA may use the Probe Request frame to discover the security policy of a STA, including additional unicast cipher suites the STA supports. A STA shall ignore Beacon frames that advertise a different security policy.

8.4.10 RSNA security association termination

When an SME receives or invokes of any of the MLME association, reassociation, disassociation, authentication, or deauthentication request or indication primitives, or if it believes that it has drifted out of radio range of another STA, it will delete some security associations. In the case of an ESS, the non-AP STA's SME shall delete the PTKSA and the GTKSA, and the AP's SME shall delete the PTKSA. In the case of an IBSS, the STA's SME shall delete the PTKSA and the receive GTKSA. Once the security associations have been deleted, the SME then invokes MLME-DELETEKEYS.request primitive to delete all temporal keys associated with the deleted security associations. The IEEE 802.1X Controlled Port returns to being blocked. As a result, all data frames are unauthorized before invocation of an MLME-DELETEKEYS.request primitive.

If a STA loses key state synchronization, it can apply the following rules to recover:

a) Any protected frame(s) received shall be discarded, and MLME-PROTECTEDFRAMED-ROPPED.indication primitive is invoked.

b) If the STA is RSNA-enabled and has joined an IBSS, the SME shall execute the authentication procedure as described in 11.3.1.

c) If the STA is RSNA-enabled and has joined an ESS, the SME shall execute the deauthentication procedures as described in 11.3.3. However, if the STA has initiated the RSN security association, but has not yet invoked the MLME-SETPROTECTION.request primitive, then no additional action is required.

NOTES

1—There is a race condition between when MLME-SETPROTECTION.request primitive is invoked on the Supplicant and when it is invoked on the Authenticator. During this time, an encrypted MPDU may be received that cannot be decrypted; and the MPDU will be discarded without a deauthentication occurring.

2—Because the IEEE 802.11 null data MPDU does not derive from an MA-UNITDATA.request, it is not protected.

If the selected AKMP fails between a STA and an AP that are associated, then both the STA and the AP shall invoke the MAC deauthentication procedure described in 11.3.3.

8.5 Keys and key distribution

8.5.1 Key hierarchy

RSNA defines two key hierarchies:

a) Pairwise key hierarchy, to protect unicast traffic

b) GTK, a hierarchy consisting of a single key to protect multicast and broadcast traffic

NOTE—Pairwise key support with TKIP or CCMP allows a receiving STA to detect MAC address spoofing and data forgery. The RSNA architecture binds the transmit and receive addresses to the pairwise key. If an attacker creates an MPDU with the spoofed TA, then the decapsulation procedure at the receiver will generate an error. GTKs do not have this property.

The description of the key hierarchies uses the following two functions:

— L(*Str*, *F*, *L*) From *Str* starting from the left, extract bits *F* through *F*+*L*–1, using the IEEE 802.11 bit conventions from 7.1.1.

— PRF-*n* Pseudo-random function producing *n* bits of output, defined in 8.5.1.1.

In an ESS, the IEEE 802.1X Authenticator MAC address (AA) and the AP's BSSID are the same, and the Supplicant's MAC address (SPA) and the STA's MAC address are equal. For the purposes of comparison, the MAC address is encoded as 6 octets, taken to represent an unsigned binary number. The first octet of the MAC address shall be used as the most significant octet. The bit numbering conventions in 7.1.1 shall be used within each octet.

An RSNA STA using CCMP shall support at least one pairwise key for any <TA,RA> pair. The <TA,RA> identifies the pairwise key, which does not correspond to any WEP key identifer.

In a a mixed environment, an AP may simultaneously communicate with some STAs using WEP with shared WEP keys and to STAs using CCMP or TKIP with pairwise keys. The STAs running WEP use default keys 0–3 for shared WEP keys; the important point here is that WEP can still use WEP default key 0. The AP can be configured to use the WEP key in WEP default key 0 for WEP; if the AP is configured in this way, STAs that cannot support WEP default key 0 simultaneously with a TKIP pairwise key shall specify the No Pairwise subfield in the RSN Capabilities field. If an AP is configured to use WEP default key 0 as a

WEP key and a "No Pairwise" STA associates, the AP shall not set the Install bit in the 4-Way Handshake. In other words, the STA will not install a pairwise temporal key and instead will use WEP default key 0 for all traffic.

NOTE—The behavior of "No Pairwise" STAs is only intended to support the migration of WEP to RSNA.

TKIP STAs in a mixed environment are expected to support a single pairwise key either by using a key mapping key or by mapping to default key 0. The AP will use a pairwise key for unicast traffic between the AP and the STA. If a key mapping key is available, the <RA,TA> pair identifies the key; if there is no key mapping key, then the default key 0 is used because the key index in the message will be 0.

A STA that cannot support TKIP keys and WEP default key 0 simultaneously advertises this deficiency by setting the No Pairwise subfield in the RSN information element it sends in the (Re)Association Request to the AP. In response, the AP will, in Message 3 of the 4-Way Handshake, clear the Install bit to notify the STA not to install the pairwise key. The AP will instead send the WEP shared key to the station to be plumbed as the WEP default key 0; this key will then be used with WEP to send and receive unicast traffic between the AP and the STA.

The TKIP STA that has this limitation may not know that it will be forced to use WEP for all transmissions until it has associated with the AP and been given the keys to use. (The STA cannot know that the AP has been configured to use WEP default key 0 for WEP communication.) If this does not satisfy the security policy configured at the STA, the STA's only recourse is to disassociate and try a different AP.

CCMP STAs in a TSN shall support pairwise keys and WEP default key 0 simultaneously. It is invalid for the STA to negotiate the No Pairwise subfield when CCMP is one of the configured ciphers.

8.5.1.1 PRF

A PRF is used in a number of places in this amendment. Depending on its use, it may need to output 128 bits, 192 bits, 256 bits, 384 bits, or 512 bits. This subclause defines five functions:

— PRF-128, which outputs 128 bits
— PRF-192, which outputs 192 bits
— PRF-256, which outputs 256 bits
— PRF-384, which outputs 384 bits
— PRF-512, which outputs 512 bits

In the following, *A* is a unique label for each different purpose of the PRF; *Y* is a single octet containing 0; *X* is a single octet containing the parameter; and || denotes concatenation:

H-SHA-1(*K*, *A*, *B*, *X*) ← HMAC-SHA-1(*K*, *A* || *Y* || *B* || *X*)

PRF(*K*, *A*, *B*, *Len*)
    for *i* ← 0 **to** (*Len*+159)/160 **do**
        R ← *R* || H-SHA-1(*K*, *A*, *B*, *i)*
    return L(*R*, 0, Len)

PRF-128(*K*, *A*, *B*) = PRF(*K*, *A*, *B*, 128)
PRF-192(*K*, *A*, *B*) = PRF(*K*, *A*, *B*, 192)
PRF-256(*K*, *A*, *B*) = PRF(*K*, *A*, *B*, 256)
PRF-384(*K*, *A*, *B*) = PRF(*K*, *A*, *B*, 384)
PRF-512(*K*, *A*, *B*) = PRF(*K*, *A*, *B*, 512)

8.5.1.2 Pairwise key hierarchy

The pairwise key hierarchy utilizes PRF-384 or PRF-512 to derive session-specific keys from a PMK, as depicted in Figure 43s. The PMK shall be 256 bits. The pairwise key hierarchy takes a PMK and generates a PTK. The PTK is partitioned into KCK and KEK, and temporal keys used by the MAC to protect unicast communication between the Authenticator's and Supplicant's respective STAs. PTKs are used between a single Supplicant and a single Authenticator.

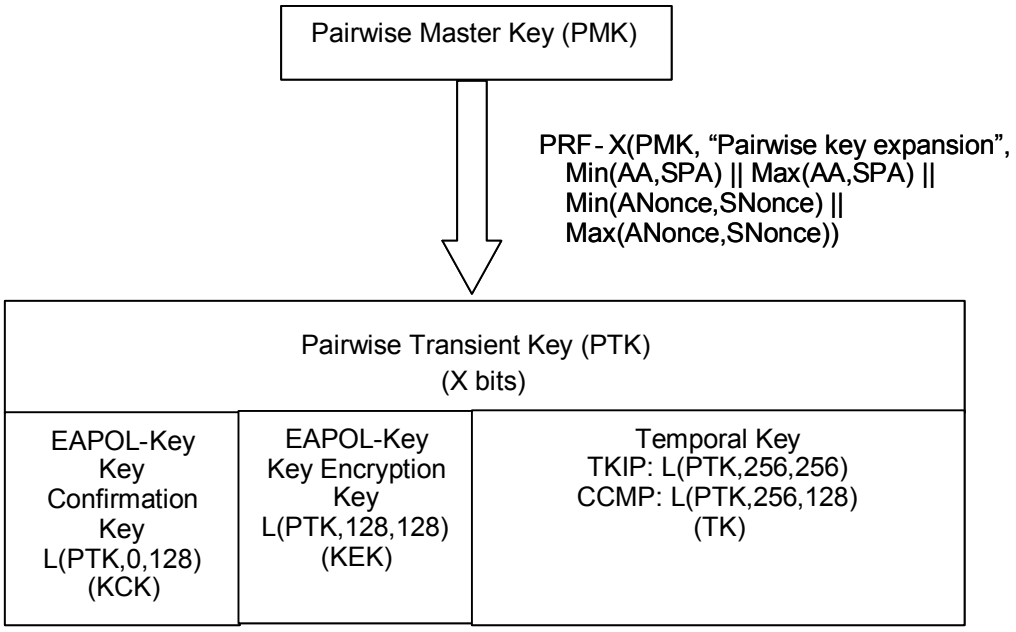


Figure 43s—Pairwise key hierarchy

When not using a PSK, the PMK is derived from the AAA key. The PMK shall be computed as the first 256 bits (bits 0–255) of the AAA key: PMK ← L(PTK, 0, 256). When this derivation is used, the AAA key must consist of at least 256 bits.

The PTK shall not be used longer than the PMK lifetime as determined by the minimum of the PMK lifetime indicated by the AS, e.g., Session-Timeout + `dot1xAuthTxPeriod` or from the `dot11RSNAConfig-PMKLifetime` MIB variable. When RADIUS is used and the Session-Timeout attribute is not in the RADIUS Accept message, and if the key lifetime is not otherwise specified, then the PMK lifetime is infinite.

NOTES

1—If the protocol between the Authenticator (or AP) and AS is RADIUS, then the MS-MPPE-Recv-Key attribute (vendor-id = 17; see Section 2.4.3 in IETF RFC 2548) may be used to transport the PMK to the AP.

2—When reauthenticating and changing the pairwise key, a race condition may occur. If a frame is received while MLME-SETKEYS.request primitive is being processed, the received frame may be decrypted with one key and the MIC checked with a different key. Two possible options to avoid this race condition are as follows: the frame can be checked against the old MIC key, and the received frames may be queued while the keys are changed.

3—If the AKMP is RSNA-PSK, then a 256-bit PSK may be configured into the STA and AP or a pass-phrase may be configured into the Supplicant or Authenticator. The method used to configure the PSK is outside this amendment, but one method is via user interaction. If a pass-phrase is configured, then a 256-bit key is derived and used as the PSK. In any RSNA-PSK method, the PSK is used directly as the PMK. Implementations may support different PSKs for each pair of communicating STAs.

Here, the following assumptions apply:

— SNonce is a random or pseudo-random value contributed by the Supplicant; its value is taken when a PTK is instantiated and is sent to the PTK Authenticator.

— ANonce is a random or pseudo-random value contributed by the Authenticator.

— The PTK shall be derived from the PMK by

PTK ← PRF-X(PMK, "Pairwise key expansion", Min(AA,SPA) || Max(AA,SPA) || Min(ANonce,SNonce) || Max(ANonce,SNonce))

TKIP uses X = 512 and CCMP uses X = 384. The Min and Max operations for IEEE 802 addresses are with the address converted to a positive integer treating the first transmitted octet as the most significant octet of the integer. The Min and Max operations for nonces are with the nonces treated as positive integers converted as specified in 7.1.1.

NOTE—The Authenticator and Supplicant normally derive a PTK only once per association. A Supplicant or an Authenticator may use the 4-Way Handshake to derive a new PTK. Both the Authenticator and Supplicant create a new nonce value for each 4-Way Handshake instance.

— The KCK shall be computed as the first 128 bits (bits 0–127) of the PTK:

KCK ← L(PTK, 0, 128)

The KCK is used by IEEE 802.1X to provided data origin authenticity in the 4-Way Handshake and Group Key Handshake messages.

— The KEK shall be computed as bits 128–255 of the PTK:

KEK ← L(PTK, 128, 128)

The KEK is used by the EAPOL-Key frames to provide confidentiality in the 4-Way Handshake and Group Key Handshake messages.

— The temporal key (TK) shall be computed as bits 256–383 (for CCMP) or bits 256–511 (for TKIP) of the PTK:

TK ← L(PTK, 256, 128) or

TK ← L(PTK, 256, 256)

The EAPOL-Key state machines (see 8.5.6 and 8.5.7) use the MLME-SETKEYS.request primitive to configure the temporal key into the STA. The STA uses the temporal key with the pairwise cipher suite; interpretation of this value is cipher-suite-specific.

A PMK identifier is defined as

PMKID = HMAC-SHA1-128(PMK, "PMK Name" || AA || SPA)

Here, HMAC-SHA1-128 is the first 128 bits of the HMAC-SHA1 of its argument list.

8.5.1.3 Group key hierarchy

The GTK shall be a random number.

Any group master key (GMK) may be reinitialized at a time interval configured into the AP to reduce the exposure of data if the GMK is ever compromised.

NOTES

1—The Authenticator may update the GTK for a number of reasons:

a) The Authenticator may change the GTK on disassociation or deauthentication of a STA.

b) An event within the STA's SME can trigger a Group Key Handshake.

2—The group key hierarchy may use PRF-128 (for CCMP) or PRF-256 (for TKIP) to derive a GTK. Figure 43t depicts one possible relationship among the keys of the group key hierarchy. In this model, the group key hierarchy takes a GMK and generates a GTK. The GTK is partitioned into temporal keys used by the MAC to protect broadcast/multicast

communication. GTKs are used between a single Authenticator and all Supplicants authenticated to that Authenticator. The Authenticator may derive new GTKs when it wants to update the GTKs.

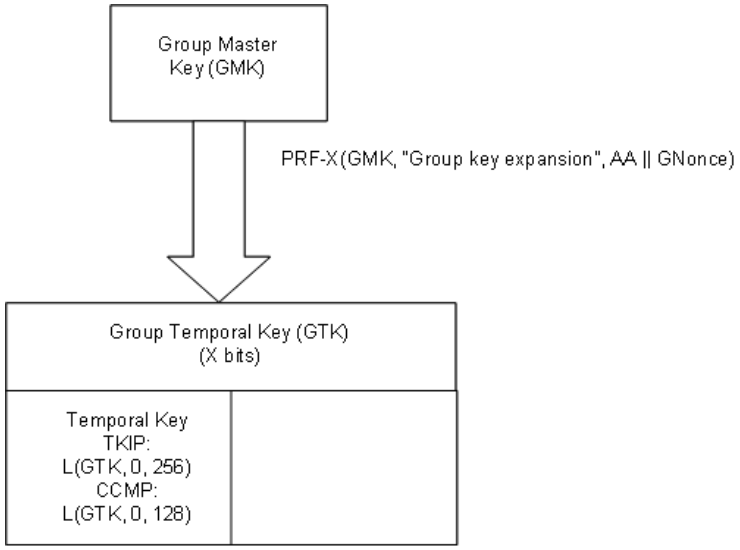


Figure 43t—Group key hierarchy (informative)

Here, the following assumptions apply:

— Group nonce (GNonce) shall be a random or pseudo-random value contributed by the IEEE 802.1X Authenticator.

— The GTK shall be derived from the GMK by

$$GTK \leftarrow PRF\text{-}X(GMK, \text{“Group key expansion”} \| AA \| GNonce)$$

TKIP uses X = 256, CCMP uses X = 128 and WEP use X = 40 or X = 104. AA is represented as an IEEE 802 address and GNonce as a bit string as defined in 7.1.1.

— The temporal key (TK) shall be bit 0–39, bits 0–103, bits 0–127, or bits 0–255 of the GTK:

$TK \leftarrow L(GTK, 0, 40)$ or

$TK \leftarrow L(GTK, 0, 104)$ or

$TK \leftarrow L(GTK, 0, 128)$ or

$TK \leftarrow L(GTK, 0, 256)$

The EAPOL-Key state machines (see 8.5.6 and 8.5.7) configure the temporal key into IEEE 802.11 via the MLME-SET-KEYS.request primitive, and IEEE 802.11 uses this key. Its interpretation is cipher-suite-specific.

8.5.2 EAPOL-Key frames

IEEE 802.11 uses EAPOL-Key frames to exchange information between STAs' Supplicants and Authenticators. These exchanges result in cryptographic keys and synchronization of security association state. EAPOL-Key frames are used to implement three different exchanges:

— 4-Way Handshake, to confirm that the PMK between associated STAs is the same and live and to transfer the GTK to the STA.

— Group Key Handshake, to update the GTK at the STA.

— STAKey Handshake, to deliver the STAKey to the initiating and peer STAs.

The RSNA key descriptor used by IEEE 802.11 does not use the IEEE 802.1X key descriptor. Instead, it uses the key descriptor described in this subclause.

The bit and octet convention for fields in the EAPOL-Key frame are defined in 7.1 of IEEE P802.1X-REV. EAPOL-Key frames containing invalid field values shall be silently discarded. Figure 43u depicts the format of an EAPOL-Key frame.

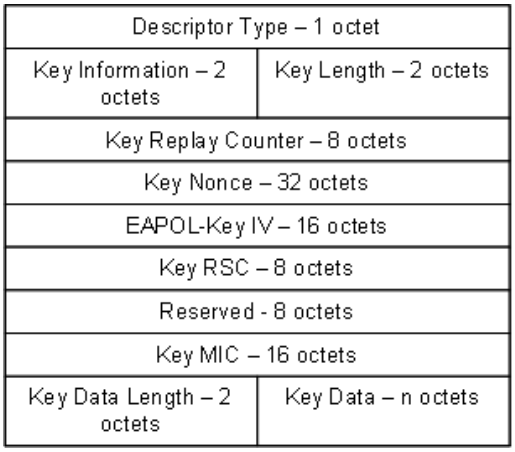


Figure 43u—EAPOL-Key frame

The fields of a EAPOL-Key frame are as follows:

a) **Descriptor Type.** This field is one octet and has a value defined by IEEE P802.1X-REV, identifying the IEEE 802.11 key descriptor.

b) **Key Information**. This field is 2 octets and specifies characteristics of the key. See Figure 43v.

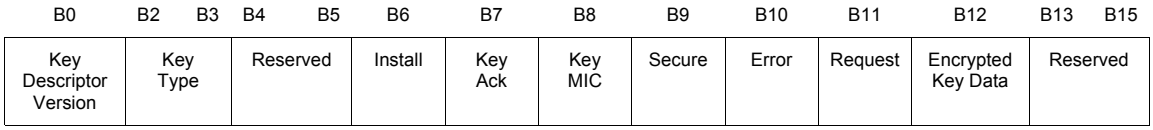


Figure 43v—Key Information bit layout

The bit convention used is as in 7.1 of IEEE P802.1X-REV. The subfields of the Key Information field are as follows:

1) Key Descriptor Version (bits 0–2) specifies the key descriptor version type.
   i) The value 1 shall be used for all EAPOL-Key frames to and from a STA when neither the group nor pairwise ciphers are CCMP for Key Descriptor 1. This value indicates the following:
      — HMAC-MD5 is the EAPOL-Key MIC.
      — RC4 is the EAPOL-Key encryption algorithm used to protect the Key Data field.
   ii) The value 2 shall be used for all EAPOL-Key frames to and from a STA when either the pairwise or the group cipher is AES-CCMP for Key Descriptor 2. This value indicates the following:
      — HMAC-SHA1-128 is the EAPOL-Key MIC. HMAC is defined in IETF RFC 2104; and SHA1, by FIPS PUB 180-1. The output of the HMAC-SHA1 shall be truncated to its 128 MSBs (octets 0–15 of the digest output by HMAC-SHA1), i.e., the last four octets generated shall be discarded.

— The NIST AES key wrap is the EAPOL-Key encryption algorithm used to protect the Key Data field. IETF RFC 3394 defines the NIST AES key wrap algorithm.

2) Key Type (bit 3) specifies whether this EAPOL-Key frame is part of a 4-Way Handshake deriving a PTK.
   i) The value 0 (Group/STAKey) indicates the message is not part of a PTK derivation.
   ii) The value 1 (Pairwise) indicates the message is part of a PTK derivation.
3) Reserved (bits 4–5). The sender shall set them to 0, and the receiver shall ignore the value of these bits.
4) Install (bit 6).
   i) If the value of Key Type (bit 3) is 1, then for the Install bit,
      — The value 1 means the IEEE 802.1X component shall configure the temporal key derived from this message into its IEEE 802.11 STA.
      — The value 0 means the IEEE 802.1X component shall not configure the temporal key into the IEEE 802.11 STA.
   ii) If the value of Key Type (bit 3) is 0, then this bit shall be 0 on transmit and ignored on receive.
5) Key Ack (bit 7) is set in messages from the Authenticator if an EAPOL-Key frame is required in response to this message and is clear otherwise. The Supplicant's response to this message shall use the same replay counter as this message.
6) Key MIC (bit 8) is set if a MIC is in this EAPOL-Key frame and is clear if this message contains no MIC.
7) Secure (bit 9) is set once the initial key exchange is complete.

   The Authenticator shall set the Secure bit to 0 in all EAPOL-Key frames sent before the Supplicant has the PTK and the GTK. The Authenticator shall set the Secure bit to 1 in all EAPOL-Key frames it sends to the Supplicant containing the last key needed to complete the Supplicant's initialization.

   The Supplicant shall set the Secure bit to 0 in all EAPOL-Key frames it sends before it has the PTK and the GTK and before it has received an EAPOL-Key frame from the Authenticator with the Secure bit set to 1 (this should be before receiving Message 3 of the 4-Way Handshake). The Supplicant shall set the Secure bit to 1 in all EAPOL-Key Frames sent after this until it loses the security association it shares with the Authenticator.
8) Error (bit 10) is set by a Supplicant to report that a MIC failure occurred in a TKIP MSDU. A Supplicant shall set this bit only when the Request (bit 11) is set.
9) Request (bit 11) is set by a Supplicant to request that the Authenticator initiate either a 4-Way Handshake or Group Key Handshake and is set by a Supplicant in a Michael MIC Failure Report. The Supplicant shall not set this bit in on-going 4-Way Handshakes, i.e., the Key Ack bit (bit 7) shall not be set in any message with the Request bit set. The Authenticator shall never set this bit.

   In a Michael MIC Failure Report, setting the bit is not a request to initiate a new handshake. However the recipient may initiate a new handshake on receiving such a message.

   If the EAPOL-Key frame with Request bit set has a key type of Pairwise, the Authenticator shall initiate a 4-Way Handshake. If the EAPOL-Key frame with Request bit set has a key type of Group/STAKey, the Authenticator shall change the GTK, initiate a 4-Way Handshake with the Supplicant, and then execute the Group Key Handshake to all Supplicants.
10) Encrypted Key Data (bit 12) is set if the Key Data field is encrypted and is clear if the Key Data field is not encrypted. This subfield shall be set, and the Key Data field shall be encrypted, if any key material (e.g., GTK or STAKey) is included in the frame.
11) Reserved (bits 13–15). The sender shall set them to 0, and the receiver shall ignore the value of these bits.

c) **Key Length**. This field is 2 octets in length, represented as an unsigned binary number. The value defines the length in octets of the pairwise temporal key to configure into IEEE 802.11. See Table 20f.

Table 20f—Cipher suite key lengths

Cipher suite	CCMP	TKIP	WEP-40	WEP-104
Key length (octets)	16	32	5	13

d) **Key Replay Counter**. This field is 8 octets, represented as an unsigned binary number, and is initialized to 0 when the PMK is established. The Supplicant shall use the key replay counter in the received EAPOL-Key frame when responding to an EAPOL-Key frame. It carries a sequence number that the protocol uses to detect replayed EAPOL-Key frames.

The Supplicant and Authenticator shall track the key replay counter per security association. The key replay counter shall be initialized to 0 on (re)association. The Authenticator shall increment the key replay counter on each successive EAPOL-Key frame.

When replying to a message from the Authenticator, the Supplicant shall use the Key Replay Counter field value from the last valid EAPOL-Key frames received from the Authenticator. The Authenticator should use the key replay counter to identify invalid messages to silently discard. The Supplicant should also use the key replay counter and ignore EAPOL-Key frames with a Key Replay Counter field value smaller than or equal to any received in a valid message. The local Key Replay Counter field should not be updated until the after EAPOL-Key MIC is checked and is valid. In other words, the Supplicant never updates the Key Replay Counter field for Message 1 in the 4-Way Handshake, as it includes no MIC. This implies the Supplicant must allow for retransmission of Message 1 when checking for the key replay counter of Message 3.

The Supplicant shall maintain a separate key replay counter for sending EAPOL-Key request frames to the Authenticator; the Authenticator also shall enforce monotonicity of a separate replay counter to filter received EAPOL-Key Request frames.

NOTE—The key replay counter does not play any role beyond a performance optimization in the 4-Way Handshake. In particular, replay protection is provided by selecting a never-before-used nonce value to incorporate into the PTK. It does, however, play a useful role in the Group Key Handshake.

e) **Key Nonce**. This field is 32 octets. It conveys the ANonce from the Authenticator and the SNonce from the Supplicant. It may contain 0 if a nonce is not required to be sent.

f) **EAPOL-Key IV**. This field is 16 octets. It contains the IV used with the KEK. It shall contain 0 when an IV is not required. It should be initialized by taking the current value of the global key counter (see 8.5.7) and then incrementing the counter. Note that only the lower 16 octets of the counter value will be used.

g) **Key RSC**. This field is 8 octets in length. It contains the receive sequence counter (RSC) for the GTK being installed in IEEE 802.11. It is used in Message 3 of the 4-Way Handshake and Message 1 of the Group Key Handshake, where it is used to synchronize the IEEE 802.11 replay state. It may also be used in the Michael MIC Failure Report frame, to report the TSC field value of the frame experiencing a MIC failure. It shall contain 0 in other messages. The Key RSC field gives the current message number for the GTK, to allow a STA to identify replayed MPDUs. If the Key RSC field value is less than 8 octets in length, the remaining octets shall be set to 0. The least significant octet of the TSC or PN should be in the first octet of the Key RSC field.

NOTE—The Key RSC field value for TKIP is the TSC in the first 6 octets and for CCMP is the PN in the first 6 octets. See Table 20g.

For WEP, the Key RSC value shall be set to 0 on transmit and shall not be used at the receiver.

Table 20g—Key RSC field

KeyRSC 0	KeyRSC 1	KeyRSC 2	KeyRSC 3	KeyRSC 4	KeyRSC 5	KeyRSC 6	KeyRSC 7
TSC0	TSC1	TSC2	TSC3	TSC4	TSC5	0	0
PN0	PN1	PN2	PN3	PN4	PN5	0	0

h) **Key MIC**. This field is 16 octets in length when the Key Descriptor Version subfield is 1 or 2. The EAPOL-Key MIC is a MIC of the EAPOL-Key frames, from and including the Key Descriptor Version field (of the Key Information field), to and including the Key Data field, calculated with the Key MIC field set to 0. If the Encrypted Key Data subfield (of the Key Information field) is set, the Key Data field is encrypted prior to computing the MIC.

   1) **Key Descriptor Version 1**: HMAC-MD5; IETF RFC 2104 and IETF RFC 1321 together define this function.

   2) **Key Descriptor Version 2**: HMAC-SHA1-128.

i) **Key Data Length**. This field is 2 octets in length, taken to represent an unsigned binary number. This represents the length of the Key Data field in octets. If the Encrypted Key Data subfield (of the Key Information field) is set, the length is the length of the Key Data field after encryption, including any padding.

j) **Key Data**. This field is a variable-length field that is used to include any additional data required for the key exchange that is not included in the fixed fields of the EAPOL-Key frame. The additional data may be zero or more information element(s) (such as the RSN information element) and zero or more key data encapsulation(s) (KDEs) (such as GTK(s), STAKey(s), or PMKID(s)). Information elements sent in the Key Data field include the Element ID and Length subfields. KDEs shall be encapsulated using the format in Figure 43w.

Type (0xdd)	Length	OUI	Data Type	Data
1 octet	1 octet	3 octets	1 octet	(Length – 4) octets

Figure 43w—KDE format

The Type field shall be set to 0xdd. The Length field specifies the number of octets in the OUI, Data Type, and Data fields. The order of the OUI field shall follow the ordering convention for MAC addresses from 7.1.1.

Table 20h lists the KDE selectors defined by this amendment.

Table 20h—KDE

OUI	Data type	Meaning
00-0F-AC	0	Reserved
00-0F-AC	1	GTK KDE
00-0F-AC	2	STAKey KDE
00-0F-AC	3	MAC address KDE

Table 20h—KDE *(continued)*

OUI	Data type	Meaning
00-0F-AC	4	PMKID KDE
00-0F-AC	5–255	Reserved
Vendor OUI	Any	Vendor specific
Other	Any	Reserved

STAs shall ignore any information elements and KDEs they do not understand.

If the Encrypted Key Data subfield (of the Key Information field) is set, the entire Key Data field shall be encrypted. If the Key Data field uses the NIST AES key wrap, then the Key Data field shall be padded before encrypting if the key data length is less than 16 octets or if it is not a multiple of 8. The padding consists of appending a single octet 0xdd followed by zero or more 0x00 octets. When processing a received EAPOL-Key message, the receiver shall ignore this trailing padding. Key Data fields that are encrypted, but do not contain the GroupKey or STAKey KDE, shall be accepted.

If the GroupKey or STAKey KDE is included in the Key Data field, but the Key Data field is not encrypted, the EAPOL-Key frames shall be ignored.

The format of the GTK KDE is shown in Figure 43x.

KeyID (0,1,2, or 3)	Tx	Reserved (0)	Reserved (0)	GTK
bits 0–1	bit 2	bit 3–7	1 octet	(Length – 6) octets

Figure 43x—GTK KDE format

If the value of the Tx field is 1, then the IEEE 802.1X component shall configure the temporal key derived from this KDE into its IEEE 802.11 STA for both transmission and reception.

If the value of the Tx field is 0, then the IEEE 802.1X component shall configure the temporal key derived from this KDE into its IEEE 802.11 STA for reception only.

The format of the STAKey and peer MAC address KDE is shown in Figure 43y.

Reserved (0)	STAKey MAC Address	STAKey
2 octets	6 octets	(Length – 12) octets

Figure 43y—STAKey KDE format

The format of the MAC address KDE is shown in Figure 43z.

MAC Address

Figure 43z—MAC address KDE format

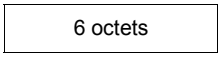


Figure 43z—MAC address KDE format

The format of the PMKID KDE is shown in Figure 43aa.

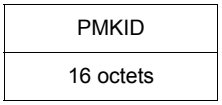


Figure 43aa—PMKID KDE format

The following EAPOL-Key frames are used to implement the three different exchanges:

— **4-Way Handshake Message 1** is an EAPOL-Key frame with the Key Type subfield set to 1. The Key Data field shall contain an encapsulated PMKID for the PMK that is being used in this key derivation and need not be encrypted.

— **4-Way Handshake Message 2** is an EAPOL-Key frame with the Key Type subfield set to 1. The Key Data field shall contain an RSN information element and need not be encrypted.

  An ESS Supplicant's SME shall insert the RSN information element it sent in its (Re)Association Request frame. The RSN information element is included as transmitted in the management frame. On receipt of Message 2, the Authenticator's SME shall validate the selected security configuration against the RSN information element received in the IEEE 802.11 (Re)Association Request.

  An IBSS Supplicant's SME shall insert an RSN information element containing the pairwise cipher suite select it wants to negotiate. The Authenticator's SME shall validate that the pairwise cipher suite selected is one of its configured cipher suites and that the group cipher suite and AKM are consistent.

— **4-Way Handshake Message 3** is an EAPOL-Key frame with the Key Type subfield set to 1. The Key Data field shall contain one or two RSN information elements. If a group cipher has been negotiated, this field shall also include an encapsulated GTK. This field shall be encrypted if a GTK is included.

  An Authenticator's SME shall insert the RSN information element it sent in its Beacon or Probe Response frame. The Supplicant's SME shall validate the selected security configuration against the RSN information element received in Message 3. If the second optional RSN information element is present, the STA shall either use that cipher suite with its pairwise key or deauthenticate. In either case, if the values do not match, then the receiver shall consider the RSN information element modified and shall use the MLME-DEAUTHENTICATE.request primitive to break the association. A security error should be logged at this time.

  It may happen, for example, that a STA's Supplicant selects a pairwise cipher suite which is advertised by an AP, but which policy disallows for this particular STA. An Authenticator may, therefore, insert a second RSN information element to overrule the STA's selection. An Authenticator's SME shall insert the second RSN information element, after the first RSN information element, only for this purpose. The pairwise cipher suite in the second RSN information element included shall be one of the ciphers advertised by the Authenticator. All other fields in the second RSN information element shall be identical to the first RSN information element.

  An encapsulated GTK shall be included and the unencrypted length of the GTK is six less than the length of the GTK KDE in octets. The entire Key Data field shall be encrypted as specified by the key descriptor version.

— **4-Way Handshake Message 4** is an EAPOL-Key frame with the Key Type subfield set to 1. The Key Data field can be empty.

— **Group Key Handshake Message 1** is an EAPOL-Key frame with the Key Type subfield set to 0. The Key Data field shall contain a GTK KDE and shall be encrypted.

— **Group Key Handshake Message 2** is an EAPOL-Key frame with the Key Type subfield set to 0. The Key Data field can be empty.

— **STAKey Handshake Message 1** is an EAPOL-Key frame with the Key Type subfield set to 0. The Key Data field shall contain a STAKey KDE and shall be encrypted. A STAKey is used to protect unicast traffic sent directly between two STAs that are associated with the same AP. The STAKey shall be cryptographically separated from the GTK.

— **STAKey Handshake Message 2** is an EAPOL-Key frame with the Key Type subfield set to 0. The Key Data field shall contain a MAC address KDE.

The key wrap algorithm selected depends on the key descriptor version:

— **Key Descriptor Version 1**: RC4 is used to encrypt the Key Data field using the KEK field from the derived PTK. No padding shall be used. The encryption key is generated by concatenating the EAPOL-Key IV field and the KEK. The first 256 octets of the RC4 key stream shall be discarded following RC4 stream cipher initialization with the KEK, and encryption begins using the 257^{th} key stream octet.

— **Key Descriptor Version 2**: AES key wrap, defined in IETF RFC 3394, shall be used to encrypt the Key Data field using the KEK field from the derived PTK. The key wrap default initial value shall be used.

NOTE—The cipher text output of the AES key wrap algorithm is 8 octets longer than the plaintext input.

8.5.2.1 STAKey Handshake for STA-to-STA link security

STA-to-STA keys are used to secure data frames directly to another STA, while associated with an AP. The AP must establish an RSNA with each STA. After the STAs establish the STA-to-STA connection, the AP sends the STAKey Handshake Message 1 to each STA, providing the key to use for securing the connection. This STAKey is used to create a STAKeySA between the two STAs.

The originating STA requests the STAKey by sending an EAPOL-Key frame to the AP, with the KeyType set to 0, Request bit set to 1, and with a MAC address KDE in the Key Data field. The cipher used with the STAKey shall be the cipher indicated in the Key Descriptor Version subfield in the EAPOL-Key frame: Version 1 indicates TKIP and Version 2 indicates CCMP.

The STAKey EAPOL-Key exchange provides a mechanism for obtaining the keys to be used for direct STA-to-STA communication in an infrastructure BSS. A STA should use this exchange prior to transferring any direct STA-to-STA data frames. The STAKeys should be deleted when the STA to STA connection is terminated. Figure 43ab depicts the sequence of events required to configure a STAKey.

8.5.2.2 EAPOL-Key frame notation

The following notation is used throughout the remainder of 8.5 to represent EAPOL-Key frames:

EAPOL-Key(S, M, A, I, K, KeyRSC, ANonce/SNonce, MIC, RSNIE, GTK[N])

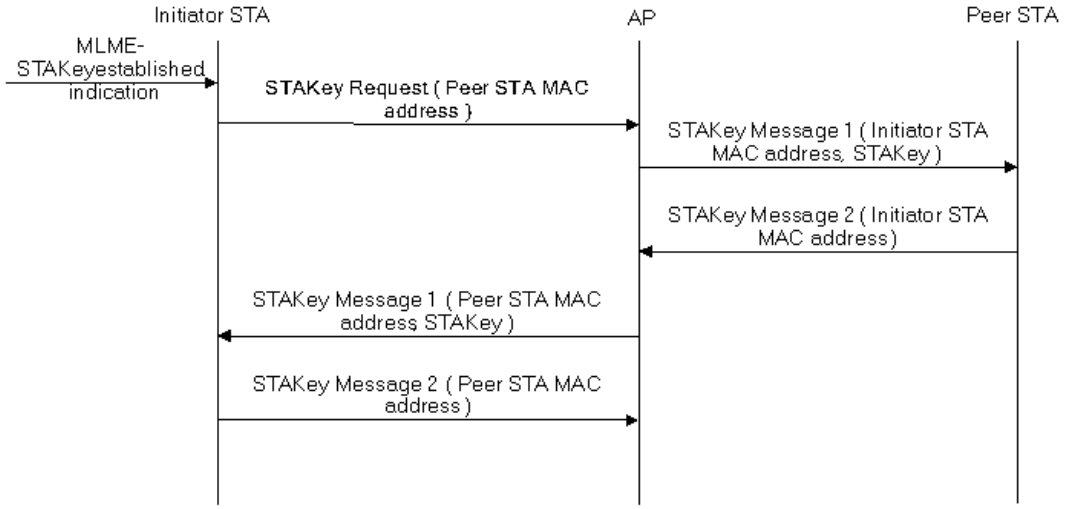


Figure 43ab—STAKey message exchange

where

S	means the initial key exchange is complete. This is the Secure bit of the Key Information field.
M	means the MIC is available in message. This should be set in all messages except Message 1 of a 4-Way Handshake. This is the Key MIC bit of the Key Information field.
A	means a response is required to this message. This is used when the receiver should respond to this message. This is the Key Ack bit of the Key Information field.
I	is the Install bit: Install/Not install for the pairwise key. This is the Install bit of the Key Information field.
K	is the key type: P (Pairwise), G (Group/STAKey). This is the Key Type bit of the Key Information field.
KeyRSC	is the key RSC. This is the Key RSC field.
ANonce/SNonce	is the Authenticator/Supplicant nonce. This is the Key Nonce field.
MIC	is the integrity check, which is generated using the KCK. This is the Key MIC field.
RSNIE	is the RSN information element. This is in the Key Data field.
GTK	is the encapsulated GTK. This is in the Key Data field.
N	is the key identifier, which specifies which index should be used for this GTK. Index 0 shall not be used for GTKs, except in mixed environments, as described in 8.5.1.

8.5.3 4-Way Handshake

RSNA defines a protocol using IEEE 802.1X EAPOL-Key frames called the 4-Way Handshake. The handshake completes the IEEE 802.1X authentication process. The information flow of the 4-Way Handshake is as follows:

Message 1. Authenticator → Supplicant: EAPOL-Key(0,0,1,0,P,0,ANonce,0,0,0)

Message 2. Supplicant → Authenticator: EAPOL-Key(0,1,0,0,P,0,SNonce,MIC,RSNIE,0)

Message 3. Authenticator → Supplicant:
EAPOL-Key(1,1,1,1,P,KeyRSC,ANonce,MIC,RSNIE,GTK[N])

Message 4. Supplicant → Authenticator: EAPOL-Key(1,1,0,0,P,0,0,MIC,0,0)

Here, the following assumptions apply:

— EAPOL-Key(·) denotes an EAPOL-Key frame conveying the specified argument list, using the notation introduced in 8.5.2.2.

— ANonce is a nonce the Authenticator contributes. ANonce has the same value in Message 1 and Message 3.

— SNonce is a nonce from the Supplicant.

— P means the pairwise bit is set.

— The MIC is computed over the body of the EAPOL-Key frame (with the Key MIC field first zeroed before the computation) using the KCK defined in 8.5.1.2.

— RSNIE represents the appropriate RSN information elements.

— GTK[N] represents the encapsulated GTK with its key identifier.

NOTE—While the MIC calculation is the same in each direction, the Key Ack bit is different in each direction. It is set in EAPOL-Key frames from the Authenticator and clear in EAPOL-Key frames from the Supplicant. 4-Way Handshake requests from the Supplicant have the Request bit set. The Authenticator and Supplicant must check these bits to stop reflection attacks. Message 1 contents must not update state, in particular the keys in use, until the data are validated with Message 3.

8.5.3.1 4-Way Handshake Message 1

Message 1 uses the following values for each of the EAPOL-Key frame fields:

Descriptor Type = N – see 8.5.2

Key Information:

Key Descriptor Version = 1 (RC4 encryption with HMAC-MD5) or 2 (NIST AES key wrap with HMAC-SHA1-128)

Key Type = 1 (Pairwise)

Install = 0

Key Ack = 1

Key MIC = 0

Secure = 0

Error = 0

Request = 0

Encrypted Key Data = 0

Reserved = 0 – unused by this protocol version

Key Length = Cipher-suite-specific; see Table 20f

Key Replay Counter = *n* – to allow Authenticator to match the right Message 2 from Supplicant

Key Nonce = ANonce

EAPOL-Key IV = 0

Key RSC = 0

Key MIC = 0

Key Data Length = 22

Key Data = PMKID for the PMK being used during this exchange

The Authenticator sends Message 1 to the Supplicant at the end of a successful IEEE 802.1X authentication, after PSK authentication is negotiated, when a cached PMKSA is used, or after a STA requests a new key. On reception of Message 1, the Supplicant determines whether the Key Replay Counter field value has been used before with the current PMKSA. If the Key Replay Counter field value is less than or equal to the current local value, the Supplicant discards the message. Otherwise, the Supplicant

a) Generates a new nonce SNonce.

b) Derives PTK.

c) Constructs Message 2.

8.5.3.2 4-Way Handshake Message 2

Message 2 uses the following values for each of the EAPOL-Key frame fields:

Descriptor Type = N – see 8.5.2

Key Information:

Key Descriptor Version = 1 (RC4 encryption with HMAC-MD5) or 2 (NIST AES key wrap with HMAC-SHA1-128) – same as Message 1
Key Type = 1 (Pairwise) – same as Message 1
Install = 0
Key Ack = 0
Key MIC = 1
Secure = 0 – same as Message 1
Error = 0 – same as Message 1
Request = 0 – same as Message 1
Encrypted Key Data = 0
Reserved = 0 – unused by this protocol version

Key Length = 0

Key Replay Counter = *n* – to let the Authenticator know to which Message 1 this corresponds

Key Nonce = SNonce

EAPOL-Key IV = 0

Key RSC = 0

Key MIC = MIC(KCK, EAPOL) – MIC computed over the body of this EAPOL-Key frame with the Key MIC field first initialized to 0

Key Data Length = length in octets of included RSN information element

Key Data = included RSN information element – the sending STA's RSN information element

The Supplicant sends Message 2 to the Authenticator.

On reception of Message 2, the Authenticator checks that the key replay counter corresponds to the outstanding Message 1. If not, it silently discards the message. Otherwise, the Authenticator

a) Derives PTK.
b) Verifies the Message 2 MIC.
    1) If the calculated MIC does not match the MIC that the Supplicant included in the EAPOL-Key frame, the Authenticator silently discards Message 2.
    2) If the MIC is valid, the Authenticator checks that the RSN information element bit-wise matches that from the (Re)Association Request message.
        i) If these are not exactly the same, the Authenticator uses MLME-DEAUTHENTICATE.request primitive to terminate the association.
        ii) If they do match bit-wise, the Authenticator constructs Message 3.

8.5.3.3 4-Way Handshake Message 3

Message 3 uses the following values for each of the EAPOL-Key frame fields:

Descriptor Type = N – see 8.5.2

Key Information:

Key Descriptor Version = 1 (RC4 encryption with HMAC-MD5) or 2 (NIST AES key wrap with HMAC-SHA1-128) – same as Message 1
Key Type = 1 (Pairwise) – same as Message 1

Install = 0/1 – 0 only if the AP does not support key mapping keys, or if the STA has the No Pairwise bit (in the RSN Capabilities field) set and only the group key will be used

Key Ack = 1

Key MIC = 1

Secure = 1 (keys installed)

Error = 0 – same as Message 1

Request = 0 – same as Message 1

Encrypted Key Data = 1

Reserved = 0 – unused by this protocol version

Key Length = Cipher-suite-specific; see Table 20f

Key Replay Counter = *n*+*1*

Key Nonce = ANonce – same as Message 1

EAPOL-Key IV = 0 (Version 2) or random (Version 1)

Key RSC = starting sequence number that the Authenticator's STA will use in MPDUs protected by GTK

Key MIC = MIC(KCK, EAPOL) – MIC computed over the body of this EAPOL-Key frame with the Key MIC field first initialized to 0

Key Data Length = length in octets of included RSN information elements and GTK

Key Data = the AP's Beacon/Probe Response frame's RSN information element, and, optionally, a second RSN information element that is the Authenticator's pairwise cipher suite assignment, and, if a group cipher has been negotiated, the encapsulated GTK and the GTK's key identifier (see 8.5.2)

The Authenticator sends Message 3 to the Supplicant.

On reception of Message 3, the Supplicant silently discards the message if the Key Replay Counter field value has already been used or if the ANonce value in Message 3 differs from the ANonce value in Message 1. The Supplicant also

a) Verifies the RSN information element.
   1) If it is not identical to that the STA received in the Beacon or Probe Response frame, the STA shall disassociate. If a second RSN information element is provided in the message, the Supplicant shall use the pairwise cipher suite specified in the second RSN information element or deauthenticate.
   2) If the RSN information element is correct, the Supplicant proceeds to Step b.
b) Verifies the Message 3 MIC.
   1) If the calculated MIC does not match the MIC that the Authenticator included in the EAPOL-Key frame, the Supplicant silently discards Message 3.
   2) Otherwise the Supplicant
      i) Updates the last-seen value of the Key Replay Counter field.
      ii) Constructs Message 4.
      iii) Sends Message 4 to the Authenticator.
      iv) Uses the MLME-SETKEYS.request primitive to configure the IEEE 802.11 MAC to send and receive Class 3 unicast MPDUs protected by the PTK. The GTK is also configured by MLME-SETKEYS primitive.

8.5.3.4 4-Way Handshake Message 4

Message 4 uses the following values for each of the EAPOL-Key frame fields:

Descriptor Type = N – see 8.5.2

Key Information:

- Key Descriptor Version = 1 (RC4 encryption with HMAC-MD5) or 2 (NIST AES key wrap with HMAC-SHA1-128) – same as Message 1
- Key Type = 1 (Pairwise) – same as Message 1
- Install = 0
- Key Ack = 0 – this is the last message
- Key MIC = 1
- Secure = 1
- Error = 0
- Request = 0
- Encrypted Key Data = 0
- Reserved = 0 – unused by this protocol version

Key Length = 0

Key Replay Counter = *n*+*1*

Key Nonce = 0

EAPOL-Key IV = 0

Key RSC = 0

Key MIC = MIC(KCK, EAPOL) – MIC computed over the body of this EAPOL-Key frame with the Key MIC field first initialized to 0

Key Data Length = 0

Key Data = none required

The Supplicant sends Message 4 to the Authenticator. Note that when the 4-Way Handshake is first used, Message 4 is sent in the clear.

On reception of Message 4, the Authenticator verifies that the Key Replay Counter field value is one that it used on this 4-Way Handshake; if it is not, it silently discards the message. Otherwise, the Authenticator

a) Checks the MIC.
   1) If the calculated MIC does not match the MIC that the Supplicant included in the EAPOL-Key frame, the Authenticator silently discards Message 4.
   2) If the MIC is valid, the Authenticator uses the MLME-SETKEYS.request primitive to configure the PTK into the IEEE 802.11 MAC.
b) Updates the Key Replay Counter field, so that it will use a fresh value if a rekey becomes necessary.

8.5.3.5 4-Way Handshake implementation considerations

If the Authenticator does not receive a reply to its messages, it shall attempt `dottRSNAConfigPairwiseUpdateCount` transmits of the message, plus a final timeout. The retransmit timeout value shall be 100 ms for the first timeout, half the listen interval for the second timeout, and the listen interval for subsequent timeouts. If there is no listen interval, then 100 ms shall be used for all timeout values. If it still has not received a response after these retries, then the Authenticator should deauthenticate the STA.

If the STA does not receive Message 1 within the expected time interval (prior to IEEE 802.1X timeout), it should disassociate, deauthenticate, and try another AP/STA.

The Authenticator should ignore EAPOL-Key frames it is not expecting in reply to messages it has sent or EAPOL-Key frames with the Ack bit set. This stops an attacker from sending the first message to the Supplicant who responds to the Authenticator.

An implementation should save the KCK and KEK beyond the 4-Way Handshake, as they are needed by the Group Key Handshake and to recover from TKIP MIC failures.

The Supplicant uses the MLME-SETKEYS.request primitive to configure the temporal key from 8.5.1 into its STA after sending Message 4 to the Authenticator.

NOTES

1—If the RSN information element check for Message 2 or Message 3 fails, IEEE 802.1X should log an error and deauthenticate the peer.

2—The Supplicant should check that if the RSN information element specified a pairwise cipher suite, then the 4-Way Handshake did specify to configure the temporal key portion of the PTK into the IEEE 802.11 STA.

8.5.3.6 Sample 4-Way Handshake (informative)

After IEEE 802.1X authentication completes by the AP sending an EAP-Success, the AP initiates the 4-Way Handshake. See Figure 43ac.

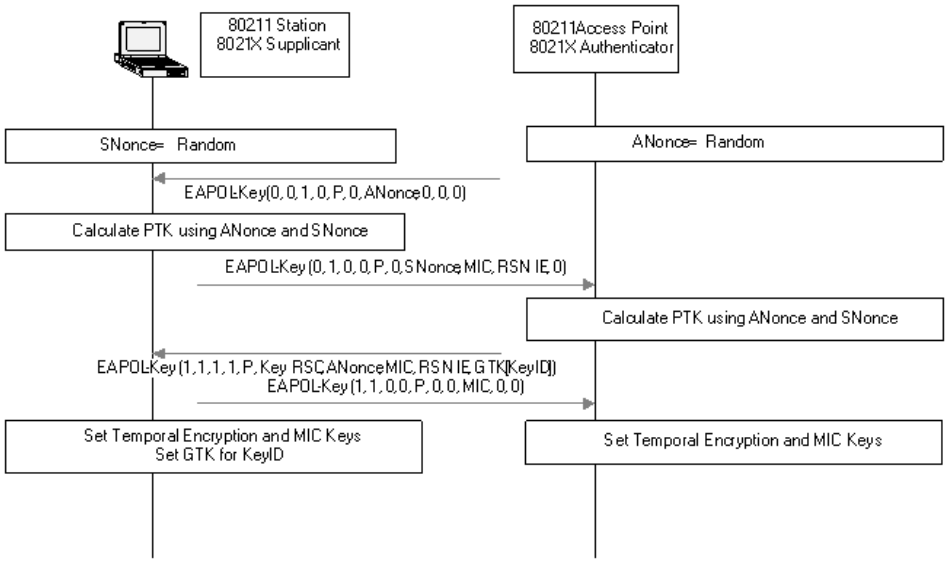


Figure 43ac—Sample 4-Way Handshake

The 4-Way Handshake consists of the following steps:

a) The Authenticator sends an EAPOL-Key frame containing an ANonce.

b) The Supplicant derives a PTK from ANonce and SNonce.

c) The Supplicant sends an EAPOL-Key frame containing SNonce, the RSN information element from the (Re)Association Request frame, and a MIC.

d) The Authenticator derives PTK from ANonce and SNonce and validates the MIC in the EAPOL-Key frame.

e) The Authenticator sends an EAPOL-Key frame containing ANonce, the RSN information element from its Beacon or Probe Response messages, MIC, whether to install the temporal keys, and the encapsulated GTK.

f) The Supplicant sends an EAPOL-Key frame to confirm that the temporal keys are installed.

8.5.3.7 4-Way Handshake analysis (informative)

This subclause makes the trust assumptions used in this protocol explicit. The protocol assumes the following:

- The PMK is known only by the Supplicant's STA and the Authenticator's STA.
- The Supplicant's STA uses IEEE 802 address SPA.
- The Authenticator's STA uses IEEE 802 address AA.

In many instantiations the RSNA architecture immediately breaks the first assumption because the IEEE 802.1X AS also knows the PMK. Therefore, additional assumptions are required:

- The AS does not expose the PMK to other parties.
- The AS does not masquerade as the Supplicant to the Authenticator.
- The AS does not masquerade as the Authenticator to the Supplicant.
- The AS does not masquerade as the Supplicant's STA.
- The AS does not masquerade as the Authenticator's STA.

The protocol also assumes this particular Supplicant-Authenticator pair is authorized to know this PMK and to use it in the 4-Way Handshake. If any of these assumptions are broken, then the protocol fails to provide any security guarantees.

The protocol also assumes that the AS delivers the correct PMK to the AP with IEEE 802 address AA and that the non-AP STA with IEEE 802 address SPA hosts the Supplicant that negotiated the PMK with the AS. None of the protocols defined by IEEE Std 802.11, 1999 Edition, and IEEE P802.1X-REV permit the AS, the Authenticator, the Supplicant, or either STA to verify these assumptions.

The PTK derivation step

PTK ← PRF-X(PMK, "Pairwise key expansion" || Min(AA,SPA) || Max(AA,SPA) ||
Min(ANonce,SNonce) || Max(ANonce,SNonce))

performs a number of functions:

- Including the AA and SPA in the computation
  - Binds the PTK to the communicating STAs and
  - Prevents undetected man-in-the-middle attacks against 4-Way Handshake messages between the STAs with these two IEEE 802 addresses.
- If ANonce is randomly selected, including ANonce
  - Guarantees the STA at IEEE 802 address AA that PTK is fresh,
  - Guarantees that Message 2 and Message 4 are live, and
  - Uniquely identifies PTK as <AA, ANonce>.
- If SNonce is randomly selected, including SNonce
  - Guarantees the STA at IEEE 802 address SPA that PTK is fresh,
  - Guarantees that Message 3 is live, and
  - Uniquely identifies PTK as <SPA, SNonce>.

Choosing the nonces randomly helps prevent precomputation attacks. With unpredictable nonces, a man-in-the-middle attack that uses the Supplicant to precompute messages to attack the Authenticator cannot progress beyond Message 2, and a similar attack against the Supplicant cannot progress beyond Message 3. The protocol can be executed further before an error if predictable nonces are used.

Message 1 delivers ANonce to the Supplicant and initiates negotiation for a new PTK. It identifies AA as the peer STA to the Supplicant's STA. If an adversary modifies either of the addresses or ANonce, the Authenticator will detect the result when validating the MIC in Message 2. Message 1 does not carry a MIC, as it is impossible for the Supplicant to distinguish this message from a replay without maintaining state of all security associations through all time (PMK might be a static key).

Message 2 delivers SNonce to the Authenticator so it can derive the PTK. If the Authenticator selected ANonce randomly, Message 2 also demonstrates to the Authenticator that the Supplicant is live, that the PTK is fresh, and that there is no man-in-the-middle attack, as the PTK includes the IEEE 802 MAC addresses of both. Inclusion of ANonce in the PTK derivation also protects against replay. The MIC prevents undetected modification of Message 2 contents.

Message 3 confirms to the Supplicant that there is no man-in-the-middle attack. If the Supplicant selected SNonce randomly, it also demonstrates that the PTK is fresh and that the Authenticator is live. The MIC again prevents undetected modification of Message 3.

While Message 4 serves no cryptographic purpose, it serves as an acknowledgment to Message 3. It is required to ensure reliability and to inform the Authenticator that the Supplicant has installed the PTK and GTK and hence can receive encrypted frames.

The PTK and GTK are installed by using MLME.SETKEYS.request primitive after Message 4 is sent. The PTK is installed before the GTK.

Then the 4-Way Handshake uses a correct, but unusual, mechanism to guard against replay. As noted earlier in this subclause, ANonce provides replay protection to the Authenticator, and SNonce to the Supplicant. In most session initiation protocols, replay protection is accomplished explicitly by selecting a nonce randomly and requiring the peer to reflect the received nonce in a response message. The 4-Way Handshake instead mixes ANonce and SNonce into the PTK, and replays are detected implicitly by MIC failures. In particular, the Key Replay Counter field serves no cryptographic purpose in the 4-Way Handshake. Its presence is not detrimental, however, and it plays a useful role as a minor performance optimization for processing stale instances of Message 2. This replay mechanism is correct, but its implicit nature makes the protocol harder to understand than an explicit approach.

It is critical to the correctness of the 4-Way Handshake that at least one bit differs in each message. Within the 4-Way Handshake, Message 1 can be recognized as the only one with the Key MIC bit clear, meaning Message 1 does not include the MIC, while Message 2 through Message 4 do. Message 3 differs from Message 2 by not asserting the Ack bit and from Message 4 by asserting the Ack Bit. Message 2 differs from Message 4 by including the RSN information element.

Request messages cannot be confused with 4-Way Handshake messages because the former asserts the Request bit and 4-Way Handshake messages do not. Group Key Handshake messages cannot be mistaken for 4-Way Handshake messages because they assert a different key type.

8.5.4 Group Key Handshake

The Authenticator uses the Group Key Handshake to send a new GTK to the Supplicant.

The Authenticator may initiate the exchange when a Supplicant is disassociated or deauthenticated.

Message 1: Authenticator → Supplicant: EAPOL-Key(1,1,1,0,G,Key RSC,0, MIC, 0,GTK[N])

Message 2: Supplicant → Authenticator: EAPOL-Key(1,1,0,0,G,0,0,MIC,0,0)

Here, the following assumptions apply:

— Key RSC denotes the last frame sequence number sent using the GTK.

— GTK[N] denotes the GTK encapsulated with its key identifier as defined in 8.5.2 using the KEK defined in 8.5.1.2 and associated IV.

— The MIC is computed over the body of the EAPOL-Key frame (with the MIC field zeroed for the computation) using the KCK defined in 8.5.1.2.

The Supplicant may trigger a Group Key Handshake by sending an EAPOL-Key frame with the Request bit set to 1 and the type of the Group Key bit.

An Authenticator shall do a 4-Way Handshake before a Group Key Handshake if both are required to be done.

NOTE—The Authenticator cannot initiate the Group Key Handshake until the 4-Way Handshake completes successfully.

8.5.4.1 Group Key Handshake Message 1

Message 1 uses the following values for each of the EAPOL-Key frame fields:

Descriptor Type = N – see 8.5.2

Key Information:

Key Descriptor Version = 1 (RC4 encryption with HMAC-MD5) or 2 (NIST AES key wrap with HMAC-SHA1-128)
Key Type = 0 (Group/STAKey)
Install = 0
Key Ack = 1
Key MIC = 1
Secure = 1
Error = 0
Request = 0
Encrypted Key Data = 1
Reserved = 0

Key Length = 0

Key Replay Counter = *n*+2

Key Nonce = 0

EAPOL-Key IV = 0 (Version 2) or random (Version 1)

Key RSC = last transmit sequence number for the GTK

Key MIC = MIC(KCK, EAPOL)

Key Data Length = Cipher-suite-specific; see Table 20f

Key Data = encrypted, encapsulated GTK and the GTK's key identifier (see 8.5.2)

The Authenticator sends Message 1 to the Supplicant.

On reception of Message 1, the Supplicant

a) Verifies that the Key Replay Counter field value has not yet been seen before, i.e., its value is strictly larger than that in any other EAPOL-Key frame received thus far during this session.

b) Verifies that the MIC is valid, i.e., it uses the KCK that is part of the PTK to verify that there is no data integrity error.

c) Uses the MLME-SETKEYS.request primitive to configure the temporal GTK into its IEEE 802.11 MAC.

d) Responds by creating and sending Message 2 of the Group Key Handshake to the Authenticator and incrementing the replay counter.

NOTE—The Authenticator must increment and use a new Key Replay Counter field value on every Message 1 instance, even retries, because the Message 2 responding to an earlier Message 1 may have been lost. If the Authenticator did not increment the replay counter, the Supplicant will discard the retry, and no responding Message 2 will ever arrive.

8.5.4.2 Group Key Handshake Message 2

Message 2 uses the following values for each of the EAPOL-Key frame fields:

Descriptor Type = N – see 8.5.2

Key Information:

Key Descriptor Version = 1 (RC4 encryption with HMAC-MD5) or 2 (NIST AES key wrap with HMAC-SHA1-128) – same as Message 1

Key Type = 0 (Group/STAKey) – same as Message 1

Install = 0

Key Ack = 0

Key MIC = 1

Secure = 1

Error = 0

Request = 0

Encrypted Key Data = 0

Reserved = 0

Key Length = 0

Key Replay Counter = *n*+2 – same as Message 1

Key Nonce = 0

EAPOL-Key IV = 0

Key RSC = 0

Key MIC = MIC(KCK, EAPOL)

Key Data Length = 0

Key Data = none required

On reception of Message 2, the Authenticator

a) Verifies that the Key Replay Counter field value matches one it has used in the Group Key Handshake.

b) Verifies that the MIC is valid, i.e., it uses the KCK that is part of the PTK to verify that there is no data integrity error.

8.5.4.3 Group Key Handshake implementation considerations

If the Authenticator does not receive a reply to its messages, its shall attempt `dot11RSNAConfigGroupUpdateCount` transmits of the message, plus a final timeout. The retransmit timeout value shall be 100 ms for the first timeout, half the listen interval for the second timeout, and the listen interval for subsequent timeouts. If there is no listen interval, then 100 ms shall be used for all timeout values. If it still has not received a response after this, then the Authenticator's STA should use the MLME-DEAUTHENTICATE.request primitive to deauthenticate the STA.

8.5.4.4 Sample Group Key Handshake (informative)

The state machines in 8.5.6 and 8.5.7 change the GTK in use by the network. See Figure 43ad.

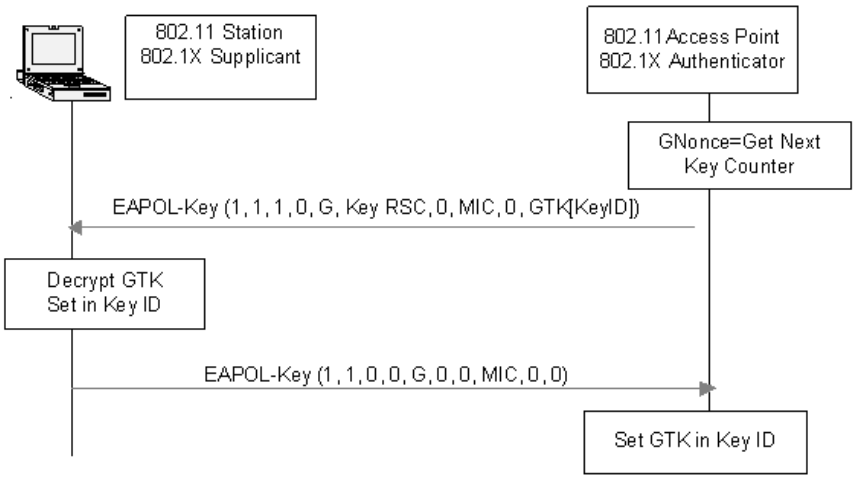


Figure 43ad—Sample Group Key Handshake

The following steps occur:

a) The Authenticator generates a new GTK. It encapsulates the GTK and sends an EAPOL-Key frame containing the GTK (Message 1), along with the last sequence number used with the GTK (RSC).

b) On receiving the EAPOL-Key frame, the Supplicant validates the MIC, decapsulates the GTK, and uses the MLME-SETKEYS.request primitive to configure the GTK and the RSC in its STA.

c) The Supplicant then constructs and sends an EAPOL-Key frame in acknowledgment to the Authenticator.

d) On receiving the EAPOL-Key frame, the Authenticator validates the MIC. If the GTK is not already configured into IEEE 802.11 MAC, after the Authenticator has delivered the GTK to all associated STAs, it uses the MLME-SETKEYS.request primitive to configure the GTK into the IEEE 802.11 STA.

8.5.5 STAKey Handshake

A STA may request the AP to establish a STAKeySA between itself and another STA associated with the same AP. Unlike the 4-Way Handshake and Group Key Handshake, the STAKey Handshake is initiated by the STA. Thus, the STAKey Request message is protected by the initiating STA's EAPOL-Key request replay counter. This is a monotonically increasing number. The AP maintains a separate request replay counter per STA to enforce replay protection.

STAKey Request: Initiating STA → Authenticator:

EAPOL-Key(1,1,0,0,G/0,0,0, MIC, 0,Peer MAC KDE)

Message 1: Authenticator → Peer STA:

EAPOL-Key(1,1,1,0,G/0,0,0, MIC, 0,Initiator MAC KDE, STAKey)

Message 2: Peer STA→ Authenticator:

EAPOL-Key(1,1,0,0,G/0,0,0,MIC,0,Initiator MAC KDE, STAKey)

Message 1: Authenticator → Initiating STA:

EAPOL-Key(1,1,1,0,G/0,0,0, MIC, 0,Peer MAC KDE, STAKey)

Message 2: Initiating STA→ Authenticator:

EAPOL-Key(1,1,0,0,G/0,0,0,MIC,0,Peer MAC KDE, STAKey)

8.5.5.1 STAKey Request message

The STAKey Request message uses the following values for each of the EAPOL-Key frame fields:

Descriptor Type = N – see 8.5.2

Key Information:

- Key Descriptor Version = 1 (RC4 encryption with HMAC-MD5) or 2 (NIST AES key wrap with HMAC-SHA1-128)
- Key Type = 0 (Group/STAKey)
- Install = 0
- Key Ack = 0
- Key MIC = 1
- Secure = 1
- Error = 0
- Request = 1
- Encrypted Key Data = 0
- Reserved = 0

Key Length = 0

Key Replay Counter = request replay counter of initiating STA

Key Nonce = 0

EAPOL-Key IV = 0

Key RSC = 0

Key MIC = MIC(initiating STA's KCK, EAPOL)

Key Data Length = Length of Key Data field in octets

Key Data = peer MAC address KDE (see Figure 43z)

A STA sends a protected STAKey Request message to the AP with the MAC address of the peer STA. On reception of a STAKey Request message, the AP verifies that the received Key Replay Counter field value is equal to or larger than its local copy of the counter. It then verifies that the MIC is valid and sets the local Key Replay Counter field value for the initiating STA to the received value.

8.5.5.2 STAKey Message 1

The STAKey Message 1 uses the following values for each of the EAPOL-Key frame fields:

Descriptor Type = N – see 8.5.2

Key Information:

- Key Descriptor Version = 1 (RC4 encryption with HMAC-MD5) or 2 (NIST AES key wrap with HMAC-SHA1-128)
- Key Type = 0 (Group/STAKey)
- Install = 1
- Key Ack = 1
- Key MIC = 1
- Secure = 1
- Error = 0
- Request = 0
- Encrypted Key Data = 1
- Reserved = 0

Key Length = Cipher-suite-specific; see Table 20f

Key Replay Counter = *n*+3 (assuming this follows the Group Key Handshake between the peer STA and the AP)

Key Nonce = 0

EAPOL-Key IV = 0 (Version 2) or random (Version 1)

Key RSC = 0

Key MIC = MIC(peer STA's KCK, EAPOL)

Key Data Length = length of Key Data field in octets

Key Data = encrypted initiator MAC address KDE and STAKey (see Figure 43y)

In response to a STAKey Request message, the AP sends STAKey Message 1 to the peer STA identified by the Request message. On reception of a STAKey Message 1, the STA verifies that the received Key Replay Counter field value was never seen before in the context of the current PMKSA. It then validates the MIC and sets the local Key Replay Counter field value from the received value in the STAKey Message 1. Next, the peer STA configures the STAKey for direct communication with the initiating STA. Finally, it sends STAKey Message 2 to the AP.

8.5.5.3 STAKey Message 2

The STAKey Message 2 uses the following values for each of the EAPOL-Key frame fields:

Descriptor Type = N – see 8.5.2

Key Information:

- Key Descriptor Version = 1 (RC4 encryption with HMAC-MD5) or 2 (NIST AES key wrap with HMAC-SHA1-128)
- Key Type = 0 (Group/STAKey)
- Install = 0
- Key Ack = 0
- Key MIC = 1
- Secure = 1
- Error = 0
- Request = 0
- Encrypted Key Data = 0
- Reserved = 0

Key Length = 0

Key Replay Counter = *n*+3 (as in STAKey Message 1)

Key Nonce = 0

EAPOL-Key IV = 0

Key RSC = 0

Key MIC = MIC(peer STA's KCK, EAPOL)

Key Data Length = length of Key Data field in octets

Key Data = initiator MAC Address KDE (see Figure 43z)

STAKey Message 2 is an integrity and replay protected acknowledgement to STAKey Message 1. Upon reception of a STAKey Message 2, the AP verifies that the received Key Replay Counter field value is the same value it sent in STAKey Message 1. It then validates the MIC and increments its Key Replay Counter field value for the peer STA. The AP finally sends STAKey Message 1 to the initiating STA.

8.5.5.4 STAKey Message 1 and Message 2 to the initiating STA

After the STAKey message exchange with the peer STA, the AP initiates the same STAKey Message 1 and Message 2 exchange with the initiating STA. This distributes the same STAKey to the initiating STA and uses identical messages to that in the STAKey exchange with the peer STA, except for the following differences:

1) The Key Replay Counter field value, KCK, and KEK correspond to the PMKSA between the AP and the initiating STA instead of those for the PMKSA between the AP and the peer STA.
2) The initiating STA's MAC and peer STA's MAC KDEs are reversed.
3) If the exchange fails, then the AP notifies the peer STA to delete the STAKey SA.

8.5.6 RSNA Supplicant key management state machine

The Supplicant shall reinitialize the Supplicant state machine whenever its system initializes. A Supplicant enters the AUTHENTICATION state on an event from the MAC that requests another STA to be authenticated. A Supplicant enters the STAKEYSTART state on receiving an EAPOL-Key frame from the Authenticator. If the MIC or any of the EAPOL-Key frames fails, the Supplicant silently discards the frame. Figure 43ae depicts the Supplicant state machine.

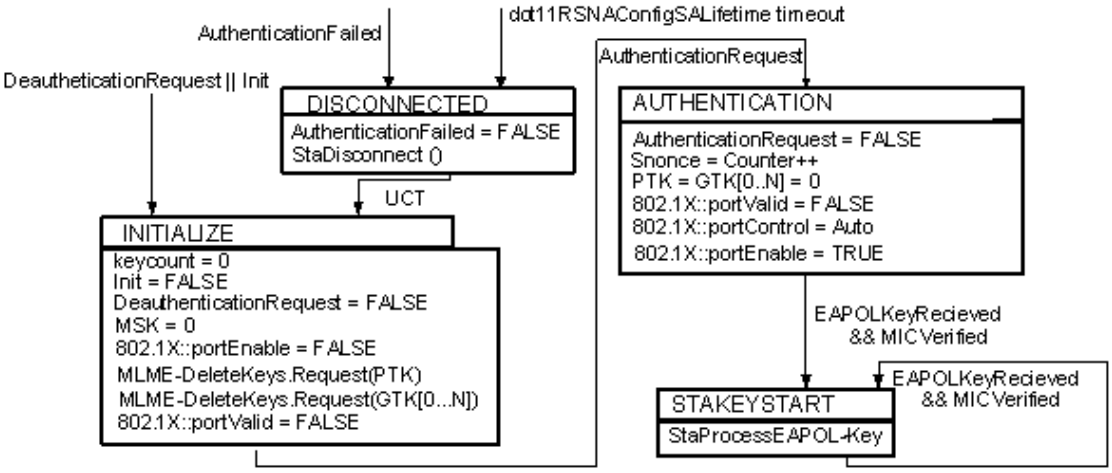


Figure 43ae—RSNA Supplicant key management state machine

Unconditional transfer (UCT) means the event triggers an immediate transition.

This state machine does not use timeouts or retries. The IEEE 802.1X state machine has timeouts that recover from authentication failures, etc.

The management entity will send an authentication request event when it wants an Authenticator authenticated. This can be before or after the STA associates to the AP. In an IBSS environment, the event will be generated when a Probe Response frame is received.

8.5.6.1 Supplicant state machine states

The following list summarizes the states of the Supplicant state machine:

— **AUTHENTICATION**: A STA's Supplicant enters this state when it sends an IEEE 802.1X AuthenticationRequest to authenticate to a SSID.

— **DISCONNECTED**: A STA's Supplicant enters this state when IEEE 802.1X authentication fails. The Supplicant executes StaDisconnect and enters the INITIALIZE state.

— **INITIALIZE**: A STA's Supplicant enters this state from the DISCONNECTED state, when it receives Disassociation or Deauthentication messages or when the STA initializes, causing the STA's Supplicant to initialize the key state variables.

— **STAKEYSTART**: A STA's Supplicant enters this state when it receives an EAPOL-Key frame. All the information to process the EAPOL-Key frame is in the message and is described in the StaProcessEAPOL-Key procedure.

8.5.6.2 Supplicant state machine variables

The following list summarizes the variables used by the Supplicant state machine:

— *DeauthenticationRequest* – The Supplicant sets this variable to TRUE if the Supplicant's STA reports it has received Disassociation or Deauthentication messages.

— *AuthenticationRequest* – The Supplicant sets this variable to TRUE if its STA's IEEE 802.11 management entity reports it wants an SSID authenticated. This can be on association or at other times.

— *AuthenticationFailed* – The Supplicant sets this variable to TRUE if the IEEE 802.1X authentication failed. The Supplicant uses the MLME-DISASSOCIATE.request primitive to cause its STA to disassociate from the Authenticator's STA.

— *EAPOLKeyReceived* – The Supplicant sets this variable to TRUE when it receives an EAPOL-Key frame.

— *IntegrityFailed* – The Supplicant sets this variable to TRUE when its STA reports that a fatal data integrity error (e.g., Michael failure) has occurred.

NOTE—A Michael failure is not the same as MICVerified because IntegrityFailed is generated if the Michael integrity check fails; MICVerified is generated from validating the EAPOL-Key integrity check. Note also the STA does not generate this event for CCMP because countermeasures are not required.

— *MICVerified* – The Supplicant sets this variable to TRUE if the MIC on the received EAPOL-Key frame verifies as correct. The Supplicant silently discards any EAPOL-Key frame received with an invalid MIC.

— *Counter* – The Supplicant uses this variable as a global counter used for generating nonces.

— *SNonce* – This variable represents the Supplicant's nonce.

— *PTK* – This variable represents the current PTK.

— *TPTK* – This variable represents the current PTK until Message 3 of the 4-Way Handshake arrives and is verified.

— *GTK[]* – This variable represents the current GTKs for each group key index.

— *PMK* – This variable represents the current PMK.

— *keycount* – This variable is used in IBSS mode to decide when all the keys have been delivered and an IBSS link is secure.

— *802.1X::XXX* – This variable denotes another IEEE 802.1X state variable *XXX* not specified in this amendment.

8.5.6.3 Supplicant state machine procedures

The following list summarizes the procedures used by the Supplicant state machine:

— **STADisconnect** – The Supplicant invokes this procedure to disassociate and deauthenticate its STA from the AP.

— **MIC**(*x*) – The Supplicant invokes this procedure to compute a MIC of the data *x*.

— **CheckMIC**() – The Supplicant invokes this procedure to verify a MIC computed by the MIC() function.

— **StaProcessEAPOL-Key** – The Supplicant invokes this procedure to process a received EAPOL-Key frame. The pseudo-code for this procedure is as follows:

StaProcessEAPOL-Key (*S*, *M*, *A*, *I*, *K*, *RSC*, *ANonce*, RSC, *MIC*, RSNIE, *GTK[N]*)

TPTK ← PTK
TSNonce ← 0

```
PRSC ← 0
UpdatePTK ← 0
State ← UNKNOWN
if M = 1 then
    if Check MIC(PTK, EAPOL-Key frame) fails then
        State ← FAILED
    else
        State ← MICOK
    endif
endif
if K = P then
    if State ≠ FAILED then
        if PSK exists then – PSK is a preshared key
            PMK ← PSK
        else
            PMK ← L(AAA Key, 0, 256)
        endif
        TSNonce ← SNonce
        if ANonce ≠ PreANonce then
            TPTK ← Calc PTK(PMK, ANonce, TSNonce)
            PreANonce ← ANonce
        endif
        if State = MICOK then
            PTK ← TPTK
            UpdatePTK ← I
            if UpdatePTK = 1 then
                if no GTK then
                    PRSC ← RSC
                endif
                if MLME-SETKEYS.request(0, TRUE, PRSC, PTK) fails then
                    invoke MLME-
                    DEAUTHENTICATE.request
                endif
                MLME.SETPROTECTION.request(TA, Rx)
            endif
            if GTK then
                if (GTK[N] ← Decrypt GTK) succeeds then
                    if MLME-SETKEYS.request(N, 0, RSC, GTK[N]) fails then
                        invoke MLME-DEAUTHENTICATE.request
                    endif
                else
                    State ← FAILED
                endif
            endif
        endif
    endif
else if KeyData = GTK then
    if State = MICOK then
        if (GTK[N] ← Decrypt GTK) succeeds then
            if MLME-SETKEYS.request(N, T, RSC, GTK[N]) fails then
                invoke MLME-
                DEAUTHENTICATE.request
            endif
        else
```

```
                    State ← FAILED
                endif
            else
                State ← FAILED
            endif
        else if KeyData = STAKey then // STAKey
            if State = MICOK then
                if (SK ← Decrypt SK) succeeds then
                    if MLME-SETKEYS.request(0, 1, RSC, SK) fails then
                        invoke MLME-
                        DEAUTHENTICATE.request
                    endif
                else
                    State ← FAILED
                endif
            else
                State ← FAILED
            endif
        endif
        if A = 1 && State ≠ Failed then
            Send EAPOL-Key(0,1,0,0,K,0,TSNonce,0,0,MIC(TPTK),RSNIE,0)
        endif
        if UpdatePTK = 1 then
            MLME.SETPROTECTION.request(TA, Tx_Rx)
        endif
        if State = MICOK && S = 1 then
            MLME.SETPROTECTION.request(TA, Tx_Rx)
            if IBSS then
                keycount++
                if keycount = 2 then
                    802.1X::portValid = TRUE
                endif
            else
                802.1X::portValid = TRUE
            endif
        endif
```

Here UNKNOWN, MICOK, and FAILED are values of the variable State used in the Supplicant pseudo-code. State is used to decide how to do the key processing. MICOK is set when the MIC of the EAPOL-Key has been checked and is valid. FAILED is used when a failure has occurred in processing the EAPOL-Key frame. UNKNOWN is the initial value of the variable State.

When processing 4-Way Handshake Message 3, the GTK is decrypted from the EAPOL-Key frame and installed. The PTK shall be installed before the GTK.

The Key Replay Counter field used by the Supplicant for EAPOL-Key frames that are sent in response to a received EAPOL-Key frame shall be the received Key Replay Counter field. Invalid EAPOL-Key frames such as invalid MIC, GTK without a MIC, etc., shall be ignored.

NOTES

1—TPTK is used to stop attackers changing the PTK on the Supplicant by sending the first message of the 4-Way Handshake. An attacker can still affect the 4-Way Handshake while the 4-Way Handshake is being carried out.

2—The PMK will be supplied by the authentication method used with IEEE 802.1X if preshared mode is not used.

3—A PTK is configured into the encryption/integrity engine depending on the Tx/Rx bit, but if configured, is always a transmit key. A GTK is configured into the encryption/integrity engine independent of the state of the Tx/Rx bit, but whether the GTK is used as a transmit key is dependent on the state of the Tx/Rx bit.

— **CalcGTK**(x) – Generates the GTK.

— **DecryptGTK(x)** – Decrypt the GTK from the EAPOL-Key frame.

8.5.7 RSNA Authenticator key management state machine

There is one state diagram for the Authenticator. In an ESS, the Authenticator will always be on the AP; and in an IBSS environment, the Authenticator will be on every STA.

The state diagram shown in parts in Figure 43af through Figure 43ai consists of the following states:

a) The AUTHENTICATION, AUTHENTICATION2, INITPMK, INITPSK, PTKSTART, PTKCALCNEGOTIATING, PTKCALCNEGOTIATING2, PTKINITNEGOTIATING, PTKINITDONE, DISCONNECT, DISCONNECTED, and INITIALIZE states. These states handle the initialization, 4-Way Handshake, tear-down, and general clean-up. These states are per associated STA.

b) The IDLE, REKEYNEGOTIATING, KEYERROR, and REKEYESTABLISHED states. These states handle the transfer of the GTK to the associated client. These states are per associated STA.

c) The GTK_INIT, SETKEYS, and SETKEYSDONE states. These states change the GTK when required, trigger all the PTK group key state machines, and update the IEEE 802.11 MAC in the Authenticator's AP when all STAs have the updated GTK. These states are global to the Authenticator.

Because there are two GTKs, responsibility for updating these keys is given to the group key state machine (see Figure 43ah). In other words, this state machine determines which GTK is in use at any time.

When a second STA associates, the group key state machine is already initialized, and a GTK is already available and in use.

When the GTK is to be updated the variable GTKReKey is set. The SETKEYS state updates the GTK and triggers all the PTK group key state machines that currently exist—one per associated STA. Each PTK group key state machine sends the GTK to its STA. When all the STAs have received the GTK (or failed to receive the key), the SETKEYSDONE state is executed which updates the APs encryption/integrity engine with the new key.

Both the PTK state machine and the PTK group key state machine use received EAPOL-Key frames as an event to change states. The PTK state machine only uses EAPOL-Key frames with the Key Type field set to Pairwise, and the PTK group key state machine only uses EAPOL-Key frames with the Key Type field set to Group/STAKey.

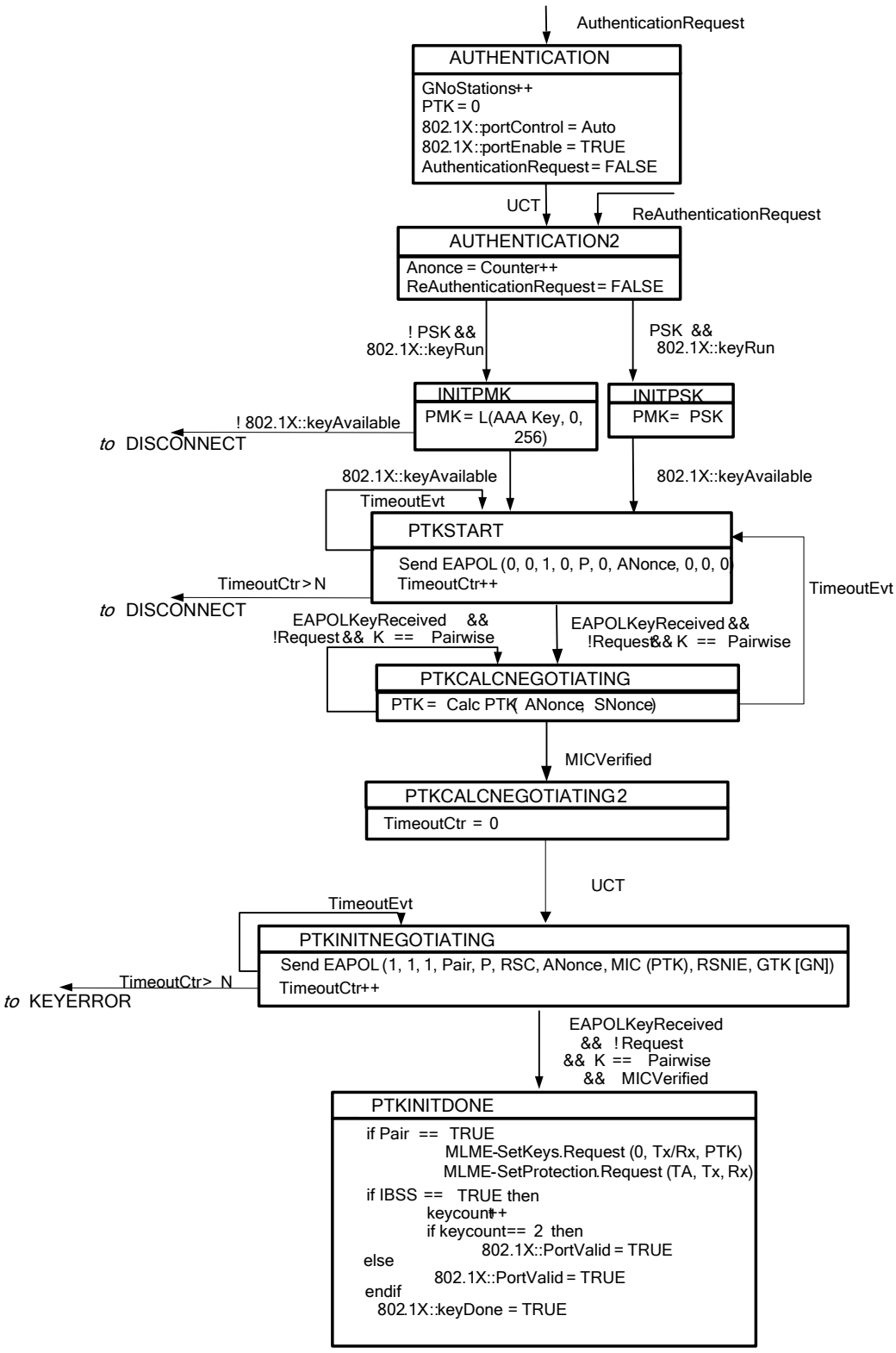


Figure 43af—Authenticator state machines, part 1

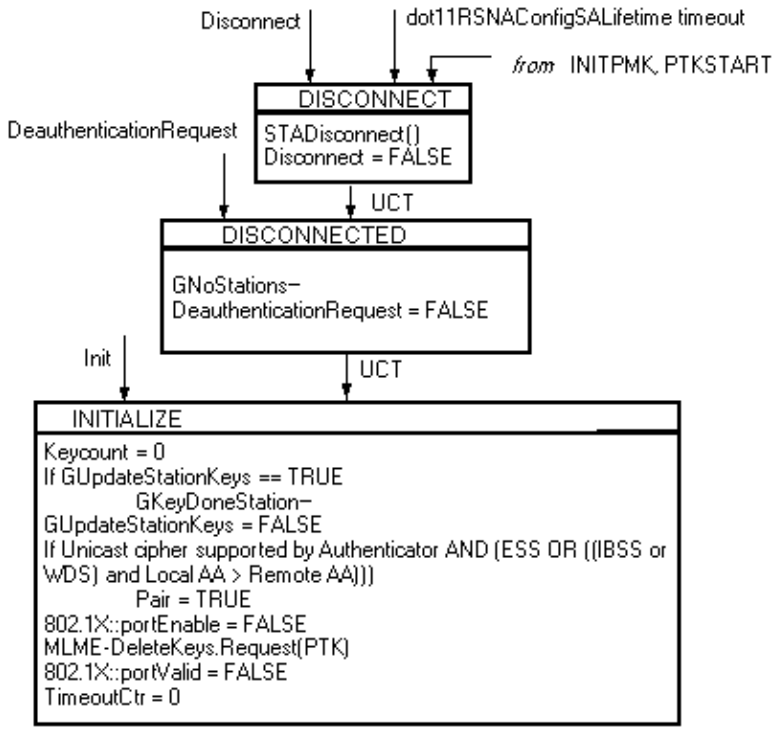


Figure 43ag—Authenticator state machines, part 2

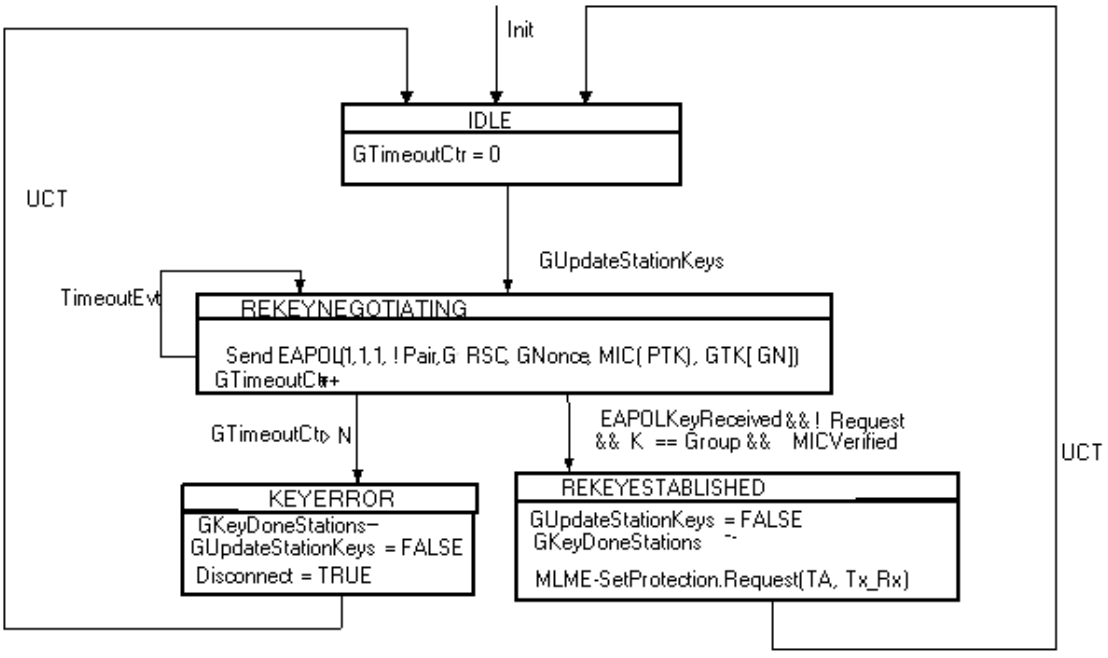


Figure 43ah—Authenticator state machines, part 3

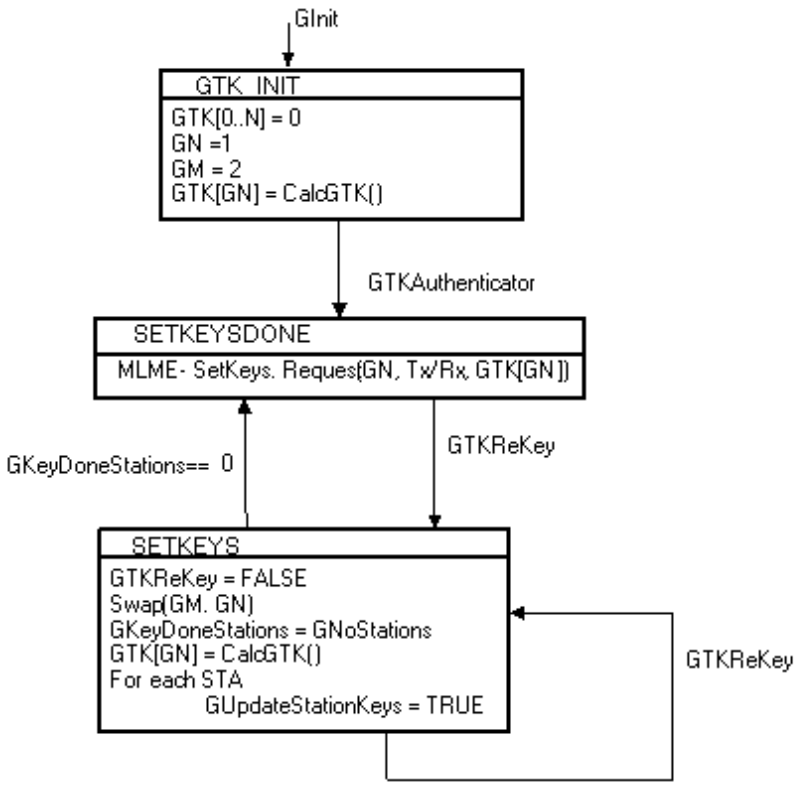


Figure 43ai—Authenticator state machines, part 4

8.5.7.1 Authenticator state machine states

8.5.7.1.1 Authenticator state machine: 4-Way Handshake (per STA)

The following list summarizes the states the Authenticator state machine uses to support the 4-Way Handshake:

— **AUTHENTICATION**: This state is entered when an AuthenticationRequest is sent from the management entity to authenticate a BSSID.

— **AUTHENTICATION2**: This state is entered from the AUTHENTICATION state or from the PTKINITDONE state.

— **DISCONNECT**: This state is entered if an EAPOL-Key frame is received and fails its MIC check. It sends a Deauthentication message to the STA and enters the INITIALIZE state.

— **DISCONNECTED**: This state is entered when Disassociation or Deauthentication messages are received.

— **INITIALIZE**: This state is entered from the DISCONNECTED state, when a deauthentication request event occurs, or when the station initializes. The state initializes the key state variables.

— **INITPMK**: This state is entered when the IEEE 802.1X backend AS completes successfully. If a PMK is supplied, it goes to the PTKSTART state; otherwise, it goes to the DISCONNECTED state.

— **INITPSK:** This state is entered when a PSK is configured.

— **PTKCALCNEGOTIATING:** This state is entered when the second EAPOL-Key frame for the 4-Way Handshake is received with the key type of Pairwise.

— **PTKCALNEGOTIATING2:** This state is entered when the MIC for the second EAPOL-Key frame of the 4-Way Handshake is verified.

— **PTKINITNEGOTIATING:** This state is entered when the MIC for the second EAPOL-Key frame for the 4-Way Handshake is verified. When Message 3 of the 4-Way Handshake is sent in state PTKINITNEGOTIATING, the encrypted GTK shall be sent at the end of the data field, and the GTK length is put in the GTK Length field.

— **PTKINITDONE:** This state is entered when the last EAPOL-Key frame for the 4-Way Handshake is received with the key type of Pairwise. This state may call SetPTK; if this call fails, the AP should detect and recover from the situation, for example, by doing a disconnect event for this association.

— **PTKSTART:** This state is entered from INITPMK or INITPSK to start the 4-Way Handshake or if no response to the 4-Way Handshake occurs.

8.5.7.1.2 Authenticator state machine: Group Key Handshake (per STA)

The following list summarizes the states the Authenticator state machine uses to support the Group Key Handshake:

— **IDLE:** This state is entered when no Group Key Handshake is occurring.

— **KEYERROR:** This state is entered if the EAPOL-Key acknowledgment for the Group Key Handshake is not received.

— **REKEYESTABLISHED:** This state is entered when an EAPOL-Key frame is received from the Supplicant with the Key Type subfield set to Group/STAKey.

— **REKEYNEGOTIATING:** This state is entered when the GTK is to be sent to the Supplicant.

NOTE—The TxRx flag for sending a GTK is always the opposite of whether the pairwise key is used for data encryption/integrity or not. If a pairwise key is used for encryption/integrity, then the STA never transmits with the GTK; otherwise, the STA uses the GTK for transmit.

8.5.7.1.3 Authenticator state machine: Group Key Handshake (global)

The following list summarizes the states the Authenticator state machine uses to coordinate a group key update of all STAs:

— **GTK_INIT:** This state is entered on system initialization.

— **SETKEYS:** This state is entered if the GTK is to be updated on all Supplicants.

— **SETKEYSDONE:** This state is entered if the GTK has been updated on all Supplicants.

NOTE—SETKEYSDONE calls SetGTK to set the GTK for all associated STA. If this fails, all communication via this key will fail, and the AP needs to detect and recover from this situation.

8.5.7.2 Authenticator state machine variables

The following list summarizes the variables used by the Authenticator state machine:

— *AuthenticationRequest* – This variable is set to TRUE if the STA's IEEE 802.11 management entity wants an association to be authenticated. This can be set when the STA associates or at other times.

— *ReAuthenticationRequest* – This variable is set to TRUE if the IEEE 802.1X Authenticator received an eapStart or 802.1X::reAuthenticate is set.

— *DeauthenticationRequest* – This variable is set to TRUE if a Disassociation or Deauthentication message is received.

— *Disconnect* – This variable is set to TRUE when the STA should initiate a deauthentication.

— *EAPOLKeyReceived* – This variable is set to TRUE when an EAPOL-Key frame is received. EAPOL-Key frames that are received in response to an EAPOL-Key frame sent by the Authenticator must contain the same Key Replay Counter field value as the Key Replay Counter field in the

transmitted message. EAPOL-Key frames that contain different Key Replay Counter field values should be discarded. An EAPOL-Key frame that is sent by the Supplicant in response to an EAPOL-Key frame from the Authenticator must not have the Ack bit set. EAPOL-Key frames sent by the Supplicant not in response to an EAPOL-Key frame from the Authenticator must have the Request bit set.

NOTES

1—EAPOL-Key frames with a key type of Pairwise and a nonzero key index should be ignored.

2—EAPOL-Key frames with a key type of Group/STAKey and an invalid key index should be ignored.

3—When an EAPOL-Key frame with the Ack bit cleared is received, then it is expected as a reply to a message that the Authenticator sent, and the replay counter is checked against the replay counter used in the sent EAPOL-Key frame. When an EAPOL-Key frame with the Request bit set is received, then a replay counter for these messages is used that is a different replay counter than the replay counter used for sending messages to the Supplicant.

— *GTimeoutCtr* – This variable maintains the count of EAPOL-Key receive timeouts for the Group Key Handshake. It is incremented each time a timeout occurs on EAPIOL-Key receive event and is initialized to 0. Annex D details the timeout values. The Key Replay Counter field value for the EAPOL-Key frame shall be incremented on each transmission of the EAPOL-Key frame.

— *GInit* – This variable is used to initialize the group key state machine. This is a group variable.

— *Init* – This variable is used to initialize per-STA state machine

— *TimeoutEvt* – This variable is set to TRUE if the EAPOL-Key frame sent out fails to obtain a response from the Supplicant. The variable may be set by management action or set by the operation of a timeout while in the PTKSTART and REKEYNEGOTIATING states.

— *TimeoutCtr* **–** This variable maintains the count of EAPOL-Key receive timeouts. It is incremented each time a timeout occurs on EAPOL-Key receive event and is initialized to 0. Annex D contains details of the timeout values. The Key Replay Counter field value for the EAPOL-Key frame shall be incremented on each transmission of the EAPOL-Key frame.

— *MICVerified* – This variable is set to TRUE if the MIC on the received EAPOL-Key frame is verified and is correct. Any EAPOL-Key frames with an invalid MIC will be dropped and ignored.

— *GTKAuthenticator* – This variable is set to TRUE if the Authenticator is on an AP or it is the designated Authenticator for an IBSS.

— *GKeyDoneStations* – Count of number of STAs left to have their GTK updated. This is a global variable.

— *GTKRekey* – This variable is set to TRUE when a Group Key Handshake is required. This is a global variable.

— *GUpdateStationKeys* – This variable is set to TRUE when a new GTK is available to be sent to Supplicants.

— *GNoStations* – This variable counts the number of Authenticators so it is known how many Supplicants need to be sent the GTK. This is a global variable.

— *Counter* – This variable is the global STA key counter.

— *ANonce* – This variable holds the current nonce to be used if the STA is an Authenticator.

— *GN*, *GM* – These are the current key indices for GTKs. Swap(GM, GN) means that the global key index in GN is swapped with the global key index in GM, so now GM and GN are reversed.

— *PTK* – This variable is the current PTK.

— *GTK[]* – This variable is the current GTKs for each GTK index.

— *PMK* – This variable is the buffer holding the current PMK.

— *802.1X::XXX* – This variable is the IEEE 802.1X state variable *XXX*.

— *keycount* – This variable is used in IBSS mode to decide when all the keys have been delivered and an IBSS link is secure.

8.5.7.3 Authenticator state machine procedures

The following list summarizes the procedures used by the Authenticator state machine:

— **STADisconnect**() – Execution of this procedure deauthenticates the STA.

— **CalcGTK**(x) – Generates the GTK.

— **MIC**(x) – Computes a MIC over the plaintext data.

8.5.8 Nonce generation (informative)

All STAs contain a global key counter, which is 256 bits in size. It should be initialized at system boot-up time to a fresh cryptographic-quality random number. Refer to H.6 on random number generation. It is recommended that the counter value is initialized to the following:

PRF-256(Random number, “Init Counter”, Local MAC Address || Time)

The local MAC address should be AA on the Authenticator and SPA on the Supplicant.

The random number is 256 bits in size. Time should be the current time [from Network Time Protocol (NTP) or another time in NTP format] whenever possible. This initialization is to ensure that different initial key counter values occur across system restarts regardless of whether a real-time clock is available. The key counter must be incremented (all 256 bits) each time a value is used as an IV. The key counter must not be allowed to wrap to the initialization value.

8.6 Mapping EAPOL keys to IEEE 802.11 keys

8.6.1 Mapping PTK to TKIP keys

See 8.5.1.2 for the definition of the EAPOL temporal key derived from PTK.

A STA shall use bits 0–127 of the temporal key as its input to the TKIP Phase 1 and Phase 2 mixing functions.

A STA shall use bits 128–191 of the temporal key as the Michael key for MSDUs from the Authenticator’s STA to the Supplicant’s STA or from the initiating STA to the peer STA for STAKeys.

A STA shall use bits 192–255 of the temporal key as the Michael key for MSDUs from the Supplicant’s STA to the Authenticator’s STA or from the peer STA to the initiating STA for STAKeys.

8.6.2 Mapping GTK to TKIP keys

See 8.5.1.3 for the definition of the EAPOL temporal key derived from GTK.

A STA shall use bits 0–127 of the temporal key as the input to the TKIP Phase 1 and Phase 2 mixing functions.

A STA shall use bits 128–191 of the temporal key as the Michael key for MSDUs from the Authenticator’s STA to the Supplicant’s STA or from the initiating STA to the peer STA for STAKeys.

A STA shall use bits 192-255 of the temporal key as the Michael key for MSDUs from the Supplicant’s STA to the Authenticator’s STA or from the peer STA to the initiating STA for STAKeys.

8.6.3 Mapping PTK to CCMP keys

See 8.5.1.2 for the definition of the EAPOL temporal key derived from PTK.

A STA shall use the temporal key as the CCMP key for MSDUs between the two communicating STAs.

8.6.4 Mapping GTK to CCMP keys

See 8.5.1.3 for the definition of the EAPOL temporal key derived from GTK.

A STA shall use the temporal key as the CCMP key.

8.6.5 Mapping GTK to WEP-40 keys

See 8.5.1.3 for the definition of the EAPOL temporal key derived from GTK.

A STA shall use bits 0–39 of the temporal key as the WEP-40 key.

8.6.6 Mapping GTK to WEP-104 keys

See 8.5.1.3 for the definition of the EAPOL temporal key derived from GTK.

A STA shall use bits 0–103 of the temporal key as the WEP-104 key.

8.7 Per-frame pseudo-code

8.7.1 WEP frame pseudo-code

An MPDU of type Data with the Protected Frame subfield of the Frame Control field equal to 1 is called a WEP MPDU. Other MPDUs of type Data are called non-WEP MPDUs.

A STA shall not transmit WEP-encapsulated MPDUs when value of the MIB variable `dot11PrivacyInvoked` is set to FALSE. This MIB variable does not affect MPDU or MAC management protocol data unit (MMPDU) reception.

if `dot11PrivacyInvoked` is "false" **then**
    the MPDU is transmitted without WEP encapsulation
else
    if (the MPDU has an individual RA and there is an entry in `dot11WEPKeyMappings` for that RA) **then**
        if that entry has WEPOn set to "false" **then**
            the MPDU is transmitted without WEP encapsulation
        else
            if that entry contains a key that is null **then**
                discard the MPDU's entire MSDU and generate an MA-UNITDATA-STATUS.indication primitive to notify LLC that the MSDU was undeliverable due to a null WEP key
            else
                encrypt the MPDU using that entry's key, setting the Key ID subfield of the IV field to zero
            endif
        endif
    else

if (the MPDU has a group RA and the Privacy subfield of the Capability Information field in this BSS is set to 0) **then**
the MPDU is transmitted without WEP encapsulation
else
if dot11WEPDefaultKeys[dot11WEPDefaultKeyID] is null **then**
discard the MPDU's entire MSDU and generate an MA-UNITDATA-STATUS.indication primitive to notify LLC that the MSDU was undeliverable due to a null WEP key
else
WEP-encapsulate the MPDU using the key dot11WEPDefaultKeys[dot11WEPDefaultKeyID], setting the Key ID subfield of the IV field to dot11WEPDefaultKeyID
endif
endif
endif
endif

When the boolean attribute aExcludeUnencrypted is set to TRUE, non-WEP MPDUs shall not be indicated at the MAC service interface, and only MSDUs successfully reassembled from successfully decrypted MPDUs shall be indicated at the MAC service interface. When receiving a frame of type Data, the values of dot11PrivacyOptionImplemented, dot11WEPKeyMappings, dot11WEPDefaultKeys, dot11WEPDefaultKeyID, and aExcludeUnencrypted in effect at the time the PHY-RXSTART.indication primitive is received by the MAC shall be used according to the following decision tree:

if the Protected Frame subfield of the Frame Control Field is zero **then**
if aExcludeUnencrypted is "true" **then**
discard the frame body without indication to LLC and increment dot11WEPExcludedCount
else
receive the frame without WEP decapsulation
endif
else
if dot11PrivacyOptionImplemented is "true" **then**
if (the MPDU has individual RA and there is an entry in dot11WEPKeyMappings matching the MPDU's TA) **then**
if that entry has WEPOn set to "false" **then**
discard the frame body and increment dot11WEPUndecryptableCount
else
if that entry contains a key that is null **then**
discard the frame body and increment dot11WEPUndecryptableCount
else
WEP-decapsulate with that key, incrementing dot11WEPICVErrorCount if the ICV check fails
endif
endif
else
if dot11WEPDefaultKeys[Key ID] is null **then**
discard the frame body and increment dot11WEPUndecryptableCount
else
WEP-decapsulate with dot11WEPDefaultKeys[Key ID], incrementing dot11WEPICVErrorCount if the ICV check fails
endif
endif

```
        else
            discard the frame body and increment dot11WEPUndecryptableCount
        endif
    endif
```

8.7.2 RSNA frame pseudo-code

STAs transmit protected MSDUs to a RA when temporal keys are configured and an MLME.SETPROTECTION.request primitive has been invoked for transmit to that RA. STAs expect to receive protected MSDUs from a TA when temporal keys are configured and an MLME.SETPROTECTION.request primitive has been invoked for receive from that TA. MSDUs that do not match these conditions are sent in the clear and are received in the clear.

8.7.2.1 Per-MSDU Tx pseudo-code

```
if dot11RSNAEnabled = true then
    if MSDU has an individual RA and Protection for RA is off for Tx then
        transmit the MSDU without protections
    else if (MPDU has individual RA and Pairwise key exists for the MPDU's RA) or (MPDU has
        a multicast or broadcast RA and network type is IBSS and IBSS GTK exists for MPDU's
        TA) then
        // If we find a suitable Pairwise or GTK for the mode we are in…
        if key is a null key then
            discard the entire MSDU and generate an MA-UNITDATA-STATUS.indication
              primitive to notify LLC that the MSDU was undeliverable due to a null key
        else
            // Note that it is assumed that no entry will be in the key
            // mapping table of a cipher type that is unsupported.
            Set the Key ID subfield of the IV field to zero.
            if cipher type of entry is AES-CCM then
                Transmit the MSDU, to be protected after fragmentation using AES-CCM
            else if cipher type of entry is TKIP then
                Compute MIC using Michael algorithm and entry's Tx MIC key.
                Append MIC to MSDU
                Transmit the MSDU, to be protected with TKIP
            else if cipher type of entry is WEP then
                Transmit the MSDU, to be protected with WEP
            endif
        endif
    else // Else we didn't find a key but we are protected, so handle the default key case or discard
        if GTK entry for Key ID contains null then
            discard the MSDU and generate an MA-UNITDATA-STATUS.indication primitive
              to notify LLC that the entire MSDU was undeliverable due to a null GTK
        else if GTK entry for Key ID is not null then
            Set the Key ID subfield of the IV field to the Key ID.
            if MPDU has an individual RA and cipher type of entry is not TKIP then
                discard the entire MSDU and generate an MA-UNITDATA-STATUS.indica-
                  tion primitive to notify LLC that the MSDU was undeliverable due to a null
                  key
            else if cipher type of entry is AES-CCM then
                Transmit the MSDU, to be protected after fragmentation using AES-CCM
            else if cipher type of entry is TKIP then
                Compute MIC using Michael algorithm and entry's Tx MIC key.
                Append MIC to MSDU
                Transmit the MSDU, to be protected with TKIP
```

```
                    else if cipher type of entry is WEP then
                        Transmit the MSDU, to be protected with WEP
                    endif
                endif
            endif
        endif
```

8.7.2.2 Per-MPDU Tx pseudo-code

```
if dot11RSNAEnabled = TRUE then
    if MPDU is member of an MSDU that is to be transmitted without protections
        transmit the MPDU without protections
    else if MSDU that MPDU is a member of is to be protected using AES-CCM
        Protect the MPDU using entry's key and AES-CCM
        Transmit the MPDU
    else if MSDU that MPDU is a member of is to be protected using TKIP
        Protect the MPDU using TKIP encryption
        Transmit the MPDU
    else if MSDU that MPDU is a member of is to be protected using WEP
        Encrypt the MPDU using entry's key and WEP
        Transmit the MPDU
    else
        // should not arrive here
    endif
endif
```

8.7.2.3 Per-MPDU Rx pseudo-code

```
if dot11RSNAEnabled = TRUE then
    if the Protected Frame subfield of the Frame Control Field is zero then
        if Protection for TA is off for Rx then
            Receive the unencrypted MPDU without protections
        else
            discard the frame body without indication to LLC and increment
              dot11WEPExcludedCount
        endif
    else if Protection is true for TA then
        if ((MPDU has individual RA and Pairwise key exists for the MPDU's TA) or (MPDU
          has a broadcast/multicast RA and network type is IBSS and IBSS GTK exists for
          MPDU's RA)) then
            if key is null then
                discard the frame body and increment dot11WEPUndecryptableCount
            else if entry has an AES-CCM key then
                decrypt frame using AES-CCM key
                discard the frame if the integrity check fails and increment dot11RSNA-
                  StatsCCMPDecryptErrors
            else if entry has a TKIP key then
                prepare a temporal key from the TA, TKIP key and PN
                decrypt the frame using RC4
                discard the frame if the ICV fails and increment dot11RSNAStatsTKIP-
                  LocalMicFailures
            else if entry has a WEP key then
                decrypt the frame using WEP decryption
                discard the frame if the ICV fails and increment dot11WEPICVErrorCount
            else
```

discard the frame body **and** increment `dot11WEPUndecryptableCount`
endif
else if GTK for the Key ID does not exist **then**
discard the frame body **and** increment `dot11WEPUndecryptableCount`
else if GTK for the Key ID is null **then**
discard the frame body **and** increment `dot11WEPUndecryptableCount`
else if the GTK for the Key ID is a CCM key **then**
decrypt frame using AES-CCM key
discard the frame if the integrity check fails and increment `dot11RSNAStats-CCMPDecryptErrors`
else if the GTK for the Key ID is a TKIP key **then**
prepare a temporal key from the TA, TKIP key and PN
decrypt the frame using RC4
discard the frame if the ICV fails and increment `dot11RSNAStatsTKIPICV-Errors`
else if the GTK for the Key ID is a WEP key **then**
decrypt the frame using WEP decryption
discard the frame if the ICV fails and increment `dot11WEPICVErrorCount`
endif
else
MLME-PROTECTEDFRAMEDROPPED.indication
discard the frame body **and** increment `dot11WEPUndecryptableCount`
endif
endif

8.7.2.4 Per-MSDU Rx pseudo-code

if *dot11RSNAEnabled* = TRUE **then**
if the frame was not protected **then**
Receive the MSDU unprotected
Make MSDU available to higher layers
else// Have a protected MSDU
if `Pairwise key is` an AES-CCM key **then**
Accept the MSDU if its MPDUs had sequential PNs (or if it consists of only one MPDU), otherwise discard the MSDU as a replay attack and increment dot11RSNAStatsCCMPReplays
Make MSDU available to higher layers
else if `Pairwise key is` a TKIP key **then**
Compute the MIC using the Michael algorithm
Compare the received MIC against the computed MIC
discard the frame if the MIC fails increment dot11RSNAStatsTKIPLocalMIC-Failures and invoke countermeasures if appropriate
compare TSC against replay counter, if replay check fails increment dot11RSNA-StatsTKIPReplays
otherwise accept the MSDU
Make MSDU available to higher layers
else if `dot11WEPKeyMappings` has a WEP key **then**
Accept the MSDU since the decryption took place at the MPDU
Make MSDU available to higher layers
endif
endif
endif

End of changes to Clause 8.

10. Layer management

10.3 MLME SAP interface

10.3.2 Scan

10.3.2.2 MLME-SCAN.confirm

10.3.2.2.2 Semantics of the service primitive

Insert the following elements at the end of the untitled table listing the elements of BSSDescription in 10.3.2.2.2:

Name	Type	Valid range	Description
RSN	RSN information element	As defined in frame format	A description of the cipher suites and AKM suites supported in the BSS.

10.3.6 Associate

10.3.6.1 MLME-ASSOCIATE.request

10.3.6.1.2 Semantics of the service primitive

Change the following primitive parameter list in 10.3.6.1.2:

MLME-ASSOCIATE.request(
PeerSTAAddress,
AssociateFailureTimeout,
CapabilityInformation,
ListenInterval,
Supported Channels
RSN
)

Insert the following row at the end of the untitled table defining the primitive parameters in 10.3.6.1.2:

Name	Type	Valid range	Description
RSN	RSN information element	As defined in frame format	A description of the cipher suites and AKM suites supported in the BSS.

10.3.6.3 MLME-ASSOCIATE.indication

10.3.6.3.2 Semantics of the service primitive

Change the following primitive parameter list in 10.3.6.3.2:

MLME-ASSOCIATE.indication(
PeerSTAAddress,
RSN
)

Insert the following row at the end of the untitled table defining the primitive parameters in 10.3.6.3.2:

Name	Type	Valid range	Description
RSN	RSN information element	As defined in frame format	A description of the cipher suites and AKM suites supported in the BSS. Only one pair-wise cipher suite and only one authenticated key suite are allowed in the RSN information element.

10.3.7 Reassociate

10.3.7.1 MLME-REASSOCIATE.request

10.3.7.1.2 Semantics of the service primitive

Change the following primitive parameter list in 10.3.7.1.2:

MLME-REASSOCIATE.request(
NewAPAddress,
ReassociateFailureTimeout,
CapabilityInformation,
ListenInterval,
Supported Channels,
RSN
)

Insert the following row at the end of the untitled table defining the primitive parameters in 10.3.7.1.2:

Name	Type	Valid range	Description
RSN	RSN information element	As defined in frame format	A description of the cipher suites and AKM suites supported in the BSS.

10.3.7.3 MLME-REASSOCIATE.indication

10.3.7.3.2 Semantics of the service primitive

Change the following primitive parameter list in 10.3.7.3.2:

MLME-REASSOCIATE.indication(
PeerSTAAddress,
RSN
)

Insert the following row at the end of the untitled table defining the primitive parameters in 10.3.7.3.2:

Name	Type	Valid range	Description
RSN	RSN information element	As defined in frame format	A description of the cipher suites and AKM suites supported in the BSS.

After 10.3.16.2.4, insert 10.3.17 through 10.3.23.1.4:

10.3.17 SetKeys

10.3.17.1 MLME-SETKEYS.request

10.3.17.1.1 Function

This primitive causes the keys identified in the parameters of the primitive to be set in the MAC and enabled for use.

10.3.17.1.2 Semantics of the service primitive

The primitive parameters are as follows:

MLME-SETKEYS.request(
Keylist
)

Name	Type	Valid range	Description
Keylist	A set of SetKeyDescriptors	N/A	The list of keys to be used by the MAC.

Each SetKeyDescriptor consists of the following elements:

Name	Type	Valid range	Description
Key	Bit string	N/A	The temporal key value
Length	Integer	N/A	The number of bits in the Key to be used.
Key ID	Integer	0–3	Key identifier
Key Type	Integer	Group, Pairwise, STAKey	Defines whether this key is a group key, pairwise key, or STAKey.
Address	MACAddress	Any valid individual MAC address	This parameter is valid only when the Key Type value is Pairwise, when the Key Type value is Group and the STA is in IBSS, or when the Key Type value is STAKey.
Receive Sequence Count	8 octets	N/A	Value the receive sequence counter(s) should be initialized to
Authenticator/Supplicant or Initiator/Peer	Boolean	True, false	Whether the key is configured by the Authenticator or Supplicant; true indicates Authenticator or Initiator.
Cipher Suite Selector	4 octets	As defined in the RSN information element format	The cipher suite required for this association.

10.3.17.1.3 When generated

This primitive is generated by the SME at any time when one or more keys are to be set in the MAC.

10.3.17.1.4 Effect of receipt

Receipt of this primitive causes the MAC to set the appropriate keys and to begin using them for future MA-UNITDATA.request and MA-UNITDATA.indication primitives provided the MLME-SETPROTECTION.request primitive has been issued.

10.3.17.2 MLME-SETKEYS.confirm

10.3.17.2.1 Function

This primitive confirms that the action of the associated MLME-SETKEYS.request primitive has been completed.

10.3.17.2.2 Semantics of the service primitive

This primitive has no parameters.

10.3.17.2.3 When generated

This primitive is generated by the MAC in response to receipt of a MLME-SETKEYS.request primitive. This primitive is issued when the action requested has been completed.

10.3.17.2.4 Effect of receipt

The SME is notified that the requested action of the MLME-SETKEYS.request primitive is completed.

10.3.18 DeleteKeys

10.3.18.1 MLME-DELETEKEYS.request

10.3.18.1.1 Function

This primitive causes the keys identified in the parameters of the primitive to be deleted from the MAC and thus disabled for use.

10.3.18.1.2 Semantics of the service primitive

The primitive parameters are as follows:

MLME-DELETEKEYS.request(
Keylist
)

Name	Type	Valid range	Description
Keylist	A set of DeleteKeyDescriptors	N/A	The list of keys to be deleted from the MAC.

Each DeleteKeyDescriptor consists of the following elements:

Name	Type	Valid range	Description
Key ID	Integer	N/A	Key identifier.
Key Type	Integer	Group, Pairwise, STAKey	Defines whether this key is a group key, pairwise key, or STAKey.
Address	MAC Address	Any valid individual MAC address	This parameter is valid only when the Key Type value is Pairwise, or when the Key Type value is Group and is from an IBSS STA, or when the Key Type value is STAKey.

10.3.18.1.3 When generated

This primitive is generated by the SME at any time when keys for a security association are to be deleted in the MAC.

10.3.18.1.4 Effect of receipt

Receipt of this primitive causes the MAC to delete the temporal keys identified by the Keylist Address, including Group, Pairwise and STAKey, and to cease using them.

10.3.18.2 MLME-DELETEKEYS.confirm

10.3.18.2.1 Function

This primitive confirms that the action of the associated MLME-DELETEKEYS.request primitive has been completed.

10.3.18.2.2 Semantics of the service primitive

This primitive has no parameters.

10.3.18.2.3 When generated

This primitive is generated by the MAC in response to receipt of a MLME-DELETEKEYS.request primitive. This primitive is issued when the action requested has been completed.

10.3.18.2.4 Effect of receipt

The SME is notified that the requested action of the MLME-DELETEKEYS.request primitive is completed.

10.3.19 MIC (Michael) failure event

10.3.19.1 MLME-MICHAELMICFAILURE.indication

10.3.19.1.1 Function

This primitive reports that a MIC failure event was detected.

10.3.19.1.2 Semantics of the service primitive

The primitive parameters are as follows:

MLME-MICHAELMICFAILURE.indication (
Count,
Address,
Key Type,
Key ID,
TSC
)

Name	Type	Valid range	Description
Count	Integer	1 or 2	The current number of MIC failure events.
Address	MACAddress	Any valid individual MAC address	The source MAC address of the frame.
Key Type	Integer	Group, Pairwise, STAKey	The key type that the receive frame used.
Key ID	Integer	0–3	Key identifier.
TSC	6 octets	N/A	The TSC value of the frame that generated the MIC failure.

10.3.19.1.3 When generated

This primitive is generated by the MAC when it has detected a MIC failure.

10.3.19.1.4 Effect of receipt

The SME is notified that the MAC has detected a MIC failure.

10.3.20 EAPOL

10.3.20.1 MLME-EAPOL.request

10.3.20.1.1 Function

This primitive is used to transfer a Michael MIC Failure Report frame.

10.3.20.1.2 Semantics of the service primitive

The primitive parameters are as follows:

MLME-EAPOL.request (
Source Address,
Destination Address,
Data
)

Name	Type	Valid range	Description
Source Address	MACAddress	N/A	The MAC sublayer address from which the EAPOL-Key frame is being sent.
Destination Address	MACAddress	N/A	The MAC sublayer entity address to which the EAPOL-Key frame is being sent.
Data	IEEE 802.1X EAPOL-Key frame	N/A	The EAPOL-Key frame to be transmitted.

10.3.20.1.3 When generated

This primitive is generated by the SME when the SME has a Michael MIC Failure Report to send.

10.3.20.1.4 Effect of receipt

The MAC sends this EAPOL-Key frame.

10.3.20.2 MLME-EAPOL.confirm

10.3.20.2.1 Function

This primitive indicates that this EAPOL-Key frame has been acknowledged by the IEEE 802.11 MAC.

10.3.20.2.2 Semantics of the service primitive

The primitive parameters are as follows:

MLME-EAPOL.confirm (
ResultCode
)

Name	Type	Valid range	Description
ResultCode	Enumeration	SUCCESS, TIMEOUT	Indicates whether the EAPOL-Key frame has been acknowledged by the target STA.

10.3.20.2.3 When generated

This primitive is generated by the MAC as a result of an MLME-EAPOL.request being generated to send an EAPOL-Key frame.

10.3.20.2.4 Effect of receipt

The SME is always notified whether this EAPOL-Key frame has been acknowledged by the IEEE 802.11 MAC.

10.3.21 MLME-STAKEYESTABLISHED

10.3.21.1 MLME-STAKEYESTABLISHED.indication

10.3.21.1.1 Function

This primitive notifies the SME that a STAKey is required.

10.3.21.1.2 Semantics of the service primitive

This primitive has two parameters, the MAC addresses of the two STAs.

The primitive parameters are as follows:

```
MLME-STAKEYESTABLISHED.indication (
                                    Address1,
                                    Address2
                                    )
```

Name	Type	Valid range	Description
Address1	MACAddress	Any valid individual MAC address	MAC address of initiating STA.
Address2	MACAddress	Any valid individual MAC address	MAC address of peer STA.

10.3.21.1.3 When generated

This primitive is generated by the MAC when a STAKey is required.

10.3.21.1.4 Effect of receipt

The SME is notified that a STAKey is required, is made aware of the two STA MAC addresses involved, and can then send STAKey Handshake messages to each STA.

10.3.22 SetProtection

10.3.22.1 MLME-SETPROTECTION.request

10.3.22.1.1 Function

This primitive indicates whether protection is required for frames sent to and received from the indicated MAC address.

10.3.22.1.2 Semantics of the service primitive

The primitive parameters are as follows:

```
MLME-SETPROTECTION.request(
                                    Protectlist
                                    )
```

Name	Type	Valid range	Description
Protectlist	A set of protection elements	N/A	The list of how each key is being used currently.

Each Protectlist consists of the following elements:

Name	Type	Valid range	Description
Address	MACAddress	Any valid individual MAC address	This parameter is valid only when the Key Type value is Pairwise or STAKey or when the Key Type value is Group and is from an IBSS STA.
ProtectType	Enumeration	None, Rx, Tx, Rx_Tx	The protection value for this MAC.
Key Type	Integer	Group, Pairwise, or STAKey	Defines whether this key is a group key, pairwise key, or STAkey.

10.3.22.1.3 When generated

This primitive is generated by the SME when protection is required for frames sent to and received from the indicated MAC address.

10.3.22.1.4 Effect of receipt

Receipt of this primitive causes the MAC to set the protection and to protect data frames as indicated in the ProtectType element of the Protectlist parameter:

— None: Specifies that data frames neither from the MAC address nor to the MAC address shall be protected.
— Rx: Specifies that data frames from MAC address shall be protected.
— Tx: Specifies that data frames to MAC address shall be protected.
— Rx_Tx: Specifies that data frames to and from MAC address shall be protected.

Once it is specified that a data frame is protected to or from a MAC address, this shall be reset by the MLME-SETPROTECTION.request primitive. The MLME-SETPROTECTION.request primitive deletes the state by specifying None.

10.3.22.2 MLME-SETPROTECTION.confirm

10.3.22.2.1 Function

This primitive indicates that the frame protection request is completed.

10.3.22.2.2 Semantics of the service primitive

There are no parameters for this primitive.

10.3.22.2.3 When generated

This primitive is generated by the MAC when the protection request is complete.

10.3.22.2.4 Effect of receipt

The SME is notified that the protection request is complete.

10.3.23 MLME-PROTECTEDFRAMEDROPPED

10.3.23.1 MLME- PROTECTEDFRAMEDROPPED.indication

10.3.23.1.1 Function

This primitive notifies the SME that a frame has been dropped because a temporal key was unavailable.

10.3.23.1.2 Semantics of the service primitive

This primitive has two parameters, the MAC addresses of the two STAs.

The primitive parameters are as follows:

MLME- PROTECTEDFRAMEDROPPED.indication (
Address1,
Address2
)

Name	Type	Valid range	Description
Address1	MACAddress	Any valid individual MAC address	MAC address of SA.
Address2	MACAddress	Any valid individual MAC address	MAC address of RA.

10.3.23.1.3 When generated

This primitive is generated by the MAC when a frame is dropped because no temporal key is available for the frame.

10.3.23.1.4 Effect of receipt

The SME is notified that a frame was dropped. The SME can use this information in an IBSS to initiate a security association to the peer STA.

End of changes to Clause 10.

11. MAC sublayer management entity

Replace 11.3 and 11.4 in their entirety with the following text:

11.3 Association and reassociation

This subclause describes the procedures used for IEEE 802.11 authentication and deauthentication. The states used in this description are those defined in 5.5.

11.3.1 Authentication—originating STA

Upon receipt of an MLME-AUTHENTICATE.request primitive, the originating STA shall authenticate with the indicated STA using the following procedure:

a) In an ESS, or optionally in an IBSS, the STA shall execute the authentication mechanism described in 8.2.2.2.

b) If the authentication was successful, the state variable for the indicated STA shall be set to State 2.

c) The STA shall issue an MLME-AUTHENTICATE.confirm primitive to inform the SME of the result of the authentication.

The STA's SME shall delete any PTKSA and temporal keys held for communication with the indicated STA by using MLME-DELETEKEYS.request primitive (see 8.4.10) before invoking MLME-AUTHENTICATE.request primitive.

11.3.2 Authentication—destination STA

Upon receipt of an Authentication frame with authentication transaction sequence number equal to 1, the destination STA shall authenticate with the indicated STA using the following procedure:

a) The STA shall execute the authentication mechanism described in 8.2.2.2.

b) The STA shall issue an MLME-AUTHENTICATE.indication primitive to inform the SME of the authentication.

The STA's SME shall delete any PTKSA and temporal keys held for communication with the indicated STA by using the MLME-DELETEKEYS.request primitive (see 8.4.10) upon receiving a MLME-AUTHENTICATE.indication primitive.

If the STA is in an IBSS, if the SME decides to initiate an RSNA, and if the SME does not know the security policy of the peer, it may issue a unicast Probe Request frame to the peer by invoking an MLME-SCAN.request to discover the peer's security policy.

11.3.3 Deauthentication—originating STA

Upon receipt of an MLME-DEAUTHENTICATE.request primitive, the originating STA shall deauthenticate with the indicated STA using the following procedure:

a) If the state variable for the indicated STA is in State 2 or State 3, the STA shall send a Deauthentication frame to the indicated STA.

b) The state variable for the indicated STA shall be set to State 1.

c) The STA shall issue an MLME-DEAUTHENTICATE.confirm primitive to inform the SME of the completion of the deauthentication.

The STA's SME shall delete any PTKSA and temporal keys held for communication with the indicated STA by using the MLME-DELETEKEYS.request primitive (see 8.4.10) and by invoking MLME-SETPROTECTION.request(None) before invoking the MLME-DEAUTHENTICATE.request primitive.

11.3.4 Deauthentication—destination STA

Upon receipt of a Deauthentication frame, the destination STA shall deauthenticate with the indicated STA using the following procedure:

a) The state variable for the indicated STA shall be set to State 1.

b) The STA shall issue an MLME-DEAUTHENTICATE.indication primitive to inform the SME of the deauthentication.

The STA's SME shall delete any PTKSA and temporal keys held for communication with the indicated STA by using the MLME-DELETEKEYS.request primitive (see 8.4.10) and by invoking MLME-SETPROTECTION.request(None) upon receiving an MLME-DEAUTHENTICATE.indication primitive.

11.4 Association, reassociation, and disassociation

This subclause defines how a STA associates and reassociates with an AP and how it disassociates from it.

The states used in this description are those defined in 5.5.

11.4.1 STA association procedures

Upon receipt of an MLME-ASSOCIATE.request primitive, a STA shall associate with an AP via the following procedure:

a) The STA shall transmit an Association Request frame to an AP with which that STA is authenticated. If the MLME-ASSOCIATE.request primitive contained an RSN information element with only one pairwise cipher suite and only one authenticated key suite, this RSN information element shall be included in the Association Request frame.

b) If an Association Response frame is received with a status value of "successful," the STA is now associated with the AP. The state variable shall be set to State 3, and the MLME shall issue an MLME-ASSOCIATE.confirm primitive indicating the successful completion of the operation.

c) If an Association Response frame is received with a status value other than "successful" or the AssociateFailureTimeout expires, the STA is not associated with the AP. The MLME shall issue an MLME-ASSOCIATE.confirm primitive indicating the failure of the operation.

d) The SME shall establish an RSNA, or it shall enable WEP by calling MLME.SETPROTECTION.request primitive with ProtectType set to "Rx_Tx," or it shall do nothing if it does not wish to secure communication.

The STA's SME shall delete any PTKSA and temporal keys held for communication with the indicated STA by using MLME-DELETEKEYS.request primitive (see 8.4.10) before invoking MLME-ASSOCIATE.request primitive.

11.4.2 AP association procedures

When an Association Request frame is received from a STA, the AP shall associate with the STA using the following procedure:

a) If the STA is not authenticated, the AP shall transmit a Deauthentication frame to the STA and terminate the association procedure.

b) In an RSNA, the AP shall check the values received in the RSN information element, to see if the values received match the AP's security policy. If not, the association shall not be accepted.

c) The AP shall transmit an Association Response with a status code as defined in 7.3.1.9. If the status value is "successful," the association identifier assigned to the STA shall be included in the response.

d) When the Association Response with a status value of "successful" is acknowledged by the STA, the STA is considered to be associated with this AP. The state variable for the STA shall be set to State 3.

e) The MLME shall issue a MLME-ASSOCIATE.indication primitive to inform the SME of the association.

f) The SME shall establish an RSNA, or it shall enable WEP by calling MLME.SETPROTECTION.request primitive with ProtectType set to "Rx_Tx," or it shall do nothing if it does not wish to secure communication.

g) The SME will inform the DS of the new association.

The STA's SME shall delete any PTKSA and temporal keys held for communication with the indicated STA by using MLME-DELETEKEYS.request primitive (see 8.4.10) upon receiving a MLME-ASSOCIATE.indication primitive.

11.4.3 STA reassociation procedures

Upon receipt of an MLME-REASSOCIATE.request primitive, a STA shall reassociate with an AP via the following procedure:

a) If the state variable is in State 1, the STA shall inform the SME of the failure of the reassociation by issuing an MLME-REASSOCIATE.confirm primitive.

b) The STA shall transmit a Reassociation Request frame to the new AP. If the MLME-REASSOCIATE.request primitive contained an RSN information element with only one pairwise cipher suite and only one authenticated key suite, this RSN information element shall be included in the Reassociation Request frame.

c) If a Reassociation Response frame is received with a status value of "successful," the STA is now associated with the new AP. The state variable shall be set to State 3, and the MLME shall issue an MLME-REASSOCIATE.confirm primitive indicating the successful completion of the operation.

d) If a Reassociation Response frame is received with a status value other than "successful" or the AssociateFailureTimeout expires, the STA is not associated with the AP. The MLME shall issue an MLME-REASSOCIATE.confirm primitive indicating the failure of the operation.

e) The SME shall establish an RSNA, or it shall enable WEP by calling MLME.SETPROTECTION.request primitive with ProtectType set to "Rx_Tx," or it shall do nothing if it does not wish to secure communication.

The STA's SME shall delete any PTKSA and temporal keys held for communication with the indicated STA by using MLME-DELETEKEYS.request primitive (see 8.4.10) before invoking MLME-REASSOCIATE.request primitive.

11.4.4 AP reassociation procedures

Whenever a Reassociation Request frame is received from a STA, the AP uses the following procedure to support reassociation:

a) If the STA is not authenticated, the AP shall transmit a Deauthentication frame to the STA and terminate the reassociation procedure.

b) In an RSNA, the AP shall check the values received in the RSN information element, to see whether the values received match the AP's security policy. If not, the association shall not be accepted.

c) The AP shall transmit a Reassociation Response frame with a status code as defined in 7.3.1.9. If the status value is "successful," the association identifier assigned to the STA shall be included in the response.

d) When the Reassociation Response frame with a status value of "successful" is acknowledged by the STA, the STA is considered to be associated with this AP. The state variable for the STA shall be set to State 3.

e) The MLME shall issue an MLME-REASSOCIATE.indication primitive to inform the SME of the association.

f) The SME shall establish an RSNA, or it shall enable WEP by calling MLME.SETPROTECTION.request primitive with ProtectType set to "Rx_Tx," or it shall do nothing if it does not wish to secure communication.

g) The SME will inform the DS of the new association.

The STA's SME shall delete any PTKSA and temporal keys held for communication with the indicated STA by using MLME-DELETEKEYS.request primitive (see 8.4.10) upon receiving a MLME-REASSOCIATE.indication primitive.

11.4.5 STA disassociation procedures

Upon receipt of an MLME-DISASSOCIATE.request primitive, an associated STA shall disassociate from an AP using the following procedure:

a) The STA shall transmit a Disassociation frame to the AP with which that STA is associated.
b) The state variable for the AP shall be set to State 2 if and only if it was not State 1.
c) The MLME shall issue an MLME-DISASSOCIATE.confirm primitive indicating the successful completion of the operation.

The STA's SME shall delete any PTKSA and temporal keys held for communication with the indicated STA by using the MLME-DELETEKEYS.request primitive (see 8.4.10) and by invoking MLME-SETPROTECTION.request(None) before invoking the MLME-DISASSOCIATE.request primitive.

11.4.6 AP disassociation procedures

Upon receipt of a Disassociation frame from an associated STA, the AP shall disassociate the STA via the following procedure:

a) The state variable for the STA shall be set to State 2.
b) The MLME shall issue an MLME-DISASSOCIATE.indication primitive to inform the SME of the disassociation.
c) The SME will update the DS.

The STA's SME shall delete any PTKSA and temporal keys held for communication with the indicated STA by using the MLME-DELETEKEYS.request primitive (see 8.4.10) and by invoking MLME-SETPROTECTION.request(None) upon receiving a MLME-DISASSOCIATE.indication primitive.

End of changes to Clause 11.

Annex A

(normative)

Protocol Implementation Conformance Statements (PICS)

A.4 PICS proforma—IEEE Std 802.11, 1999 Edition

A.4.4 MAC protocol

A.4.4.1 MAC protocol capabilities

Insert the following at the end of the table in A.4.4.1:

Item	Protocol capability	References	Status	Support
	Are the following MAC protocol capabilities supported?			
PC34	Robust security network association (RSNA)	7.2.2, 7.3.1.4, 5.4.3.3, 8.7.2, 11.3, 11.4, 8.3.3	O	Yes ❑ No ❑
PC34.1	RSN Information Element (IE)	7.3.2.25	PC34:M	Yes ❑ No ❑
PC34.1.1	Group cipher suite	7.3.2.25	PC34.1:M	Yes ❑ No ❑
PC34.1.2	Pairwise cipher suite list	7.3.2.25	PC34.1:M	Yes ❑ No ❑
PC34.1.2.1	CTR [counter mode] with CBC-MAC [cipher-block chaining (CBC) with message authentication code (MAC)] Protocol (CCMP) data confidentiality protocol	8.3.3	PC34:M	Yes ❑ No ❑
PC34.1.2.1.1	CCMP encapsulation procedure	8.3.3.3	PC34.1.2.1:M	Yes ❑ No ❑
PC34.1.2.1.2	CCMP decapsulation procedure	8.3.3.4	PC34.1.2.1:M	Yes ❑ No ❑
PC34.1.2.2	Temporal Key Integrity Protocol (TKIP) data confidentiality protocol	8.3.2	PC34:O	Yes ❑ No ❑
PC34.1.2.2.1	TKIP encapsulation procedure	8.3.2.1.1	PC34.1.2.2:M	Yes ❑ No ❑
PC34.1.2.2.2	TKIP decapsulation procedure	8.3.2.1.2	PC34.1.2.2:M	Yes ❑ No ❑
PC34.1.2.2.3	TKIP countermeasures	8.3.2.4	PC34.1.2.2:M	Yes ❑ No ❑
PC34.1.2.2.4	TKIP security services management	8.3.2.3	PC34.1.2.2:M	Yes ❑ No ❑
PC34.1.3	Authentication key management (AKM) suite list	7.3.2.25, 8.1.3	PC34.1:M	Yes ❑ No ❑

Item	Protocol capability	References	Status	Support
PC34.1.3.1	IEEE 802.1X-defined/ RSNA key management	7.3.2.25	PC34.1:M	Yes ❑ No ❑
PC34.1.3.2	Preshared key (PSK)/ RSNA key management	7.3.2.25	PC34.1:M	Yes ❑ No ❑
PC34.1.3.3	RSNA key management	8.5	PC34.1:M	Yes ❑ No ❑
PC34.1.3.3.1	Key hierarchy	8.5, 8.6	PC34.1:M	Yes ❑ No ❑
PC34.1.3.3.1.1	Pairwise key hierarchy	8.5.1.2	PC34.1:M	Yes ❑ No ❑
PC34.1.3.3.1.2	Group key hierarchy	8.5.1.3	PC34.1:M	Yes ❑ No ❑
PC34.1.3.3.2	4-Way Handshake	8.5.3	PC34.1:M	Yes ❑ No ❑
PC34.1.3.3.3	Group Key Handshake	8.5.4	PC34.1:M	Yes ❑ No ❑
PC34.1.4	RSN capabilities	7.3.2.25, 8.1.2	PC34.1:M	Yes ❑ No ❑
PC34.1.5	RSNA preauthentication	8.4.6.1	PC34.1:O	Yes ❑ No ❑
PC34.1.6	RSNA security association management	8.4	PC34.1:M	Yes ❑ No ❑
PC34.1.7	RSNA pairwise master key security association (PMKSA) caching	8.4.1, 8.4.6.2	PC34.1:M	Yes ❑ No ❑
PC34.1.8	RSNA extended service set (ESS)	8.4.6, 8.4.8	(PC34.1 and CF1):M	Yes ❑ No ❑
PC34.1.8.1	RSNA STAKey	8.5.2.1	PC34.1.8:O	Yes ❑ No ❑
PC34.1.9	RSNA independent basic service set (IBSS)	8.4.4, 8.4.7, 8.4.9	(PC34.1 and CF2):O	Yes ❑ No ❑

Annex C

(normative)

Formal description of MAC operation

C.3 State machines for MAC stations

Insert the following text as the final paragraph before the first diagram of C.3:

This subclause describes the security behavior of only 8.2.1 and 8.2.2.

C.4 State machines for MAC AP

Insert the following text as the final paragraph before the first diagram of C.4:

This subclause describes the security behavior of only 8.2.1 and 8.2.2.

Annex D

(normative)

ASN.1 encoding of the MAC and PHY MIB

In "Major sections" of Annex D, change the dot11smt object identifier list as follows:

```
-- dot11smt GROUPS
-- dot11StationConfigTable                      ::= { dot11smt 1 }
-- dot11AuthenticationAlgorithmsTable           ::= { dot11smt 2 }
-- dot11WEPDefaultKeysTable                     ::= { dot11smt 3 }
-- dot11WEPKeyMappingsTable                     ::= { dot11smt 4 }
-- dot11PrivacyTable                            ::= { dot11smt 5 }
-- dot11SMTnotification                         ::= { dot11smt 6 }
-- dot11MultiDomainCapabilityTable              ::= { dot11smt 7 }
-- dot11SpectrumManagementTable                 ::= { dot11smt 8 }
-- dot11RSNAConfigTable                         ::= { dot11smt 9 }
-- dot11RSNAConfigPairwiseCiphersTable          ::= { dot11smt 10 }
-- dot11RSNAConfigAuthenticationSuitesTable     ::= { dot11smt 11 }
-- dot11RSNAStatsTable                          ::= { dot11smt 12 }
```

In "SMT Station Config Table" in Annex D, change Dot11StationConfigEntry as follows:

```
Dot11StationConfigEntry ::=
      SEQUENCE {
            dot11StationID                          MacAddress,
            dot11MediumOccupancyLimit               INTEGER,
            dot11CFPollable                         TruthValue,
            dot11CFPPeriod                          INTEGER,
            dot11CFPMaxDuration                     INTEGER,
            dot11AuthenticationResponseTimeOut      Unsigned32,
            dot11PrivacyOptionImplemented           TruthValue,
            dot11PowerManagementMode                INTEGER,
            dot11DesiredSSID                        OCTET STRING,
            dot11DesiredBSSType                     INTEGER,
            dot11OperationalRateSet                 OCTET STRING,
            dot11BeaconPeriod                       INTEGER,
            dot11DTIMPeriod                         INTEGER,
            dot11AssociationResponseTimeOut         Unsigned32,
            dot11DisassociateReason                 INTEGER,
            dot11DisassociateStation                MacAddress,
            dot11DeauthenticateReason               INTEGER,
            dot11DeauthenticateStation              MacAddress,
            dot11AuthenticateFailStatus             INTEGER,
            dot11AuthenticateFailStation            MacAddress,
            dot11MultiDomainCapabilityImplemented   TruthValue,
            dot11MultiDomainCapabilityEnabled       TruthValue,
            dot11CountryString                      OCTET STRING,
            dot11SpectrumManagementImplemented      TruthValue,
            dot11SpectrumManagementRequired         TruthValue,
            dot11RSNAOptionImplemented              TruthValue,
            dot11RSNAPreauthenticationImplemented   TruthValue }
```

In "SMT Station Config Table" in Annex D, insert the following attributes after dot11Spectrum-ManagementRequired { dot11StationConfigEntry 25 }:

```
dot11RSNAOptionImplemented OBJECT-TYPE
      SYNTAX TruthValue
      MAX-ACCESS read-only
```

```
        STATUS current
        DESCRIPTION
                "This variable indicates whether the entity is RSNA-capable."
    ::= { dot11StationConfigEntry 26 }

dot11RSNAPreauthenticationImplemented OBJECT-TYPE
        SYNTAX TruthValue
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "This variable indicates whether the entity supports RSNA
                preauthentication. This cannot be TRUE unless
                dot11RSNAOptionImplemented is TRUE."
::= { dot11StationConfigEntry 27 }
```

In "dot11PrivacyTable TABLE" in Annex D, change Dot11PrivacyEntry as follows:

```
Dot11PrivacyEntry ::=
        SEQUENCE {
                dot11PrivacyInvoked              TruthValue,
                dot11WEPDefaultKeyID             INTEGER,
                dot11WEPKeyMappingLength         Unsigned32,
                dot11ExcludeUnencrypted          TruthValue,
                dot11WEPICVErrorCount            Counter32,
                dot11WEPExcludedCount            Counter32,
                dot11RSNAEnabled                 TruthValue,
                dot11RSNAPreauthenticationEnabled TruthValue }
```

In the "dot11PrivacyTable TABLE" of Annex D, change dot11PrivacyInvoked as follows:

```
dot11PrivacyInvoked OBJECT-TYPE
        SYNTAX TruthValue
        MAX-ACCESS read-write
        STATUS current
        DESCRIPTION
                "When this attribute is ~~true~~TRUE, it shall indicate that ~~the IEEE
                802.11 WEP mechanism is used~~some level of security is invoked for
                transmitting frames of type Data. ~~The default value of this
                attribute shall be false.~~ For WEP-only clients, the security mecha-
                nism used is WEP.

                For RSNA-capable clients, an additional variable dot11RSNAEnabled
                indicates whether RSNA is enabled. If dot11RSNAEnabled is FALSE or
                the MIB variable does not exist, the security mechanism invoked is
                WEP; if dot11RSNAEnabled is TRUE, RSNA security mechanisms invoked
                are configured in the dot11RSNAConfigTable. The default value of
                this attribute shall be FALSE."
    ::= { dot11PrivacyEntry 1 }
```

In "dot11PrivacyTable TABLE" of Annex D, change dot11ExcludeUnencrypted as follows:

```
dot11ExcludeUnencrypted OBJECT-TYPE
        SYNTAX TruthValue
        MAX-ACCESS read-write
        STATUS current
        DESCRIPTION
                "When this attribute is true, the STA shall not indicate at the MAC
                service interface received MSDUs that have the ~~WEP~~ Protected Frame
                subfield of the Frame Control field equal to zero. When this
                attribute is false, the STA may accept MSDUs that have the WEP sub-
                field of the Frame Control field equal to zero. The default value
                of this attribute shall be false."
    ::= { dot11PrivacyEntry 4 }
```


In "dot11PrivacyTable TABLE" of Annex D, change dot11WEPICVErrorCount as follows:

```
dot11WEPICVErrorCount OBJECT-TYPE
       SYNTAX Counter32
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "This counter shall increment when a frame is received with the
              ~~WEP~~ Protected Frame subfield of the Frame Control field set to one
              and the value of the ICV as received in the frame does not match the
              ICV value that is calculated for the contents of the received
              frame. ICV errors for TKIP are not counted in this variable but
              in dot11RSNAStatsTKIPICVErrors."
   ::= { dot11PrivacyEntry 5 }
```

In "dot11PrivacyTable TABLE" of Annex D, change dot11WEPExcludedCount as follows:

```
dot11WEPExcludedCount OBJECT-TYPE
       SYNTAX Counter32
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
       "This counter shall increment when a frame is received with the ~~WEP~~
              Protected Frame subfield of the Frame Control field set to zero and
              the value of dot11ExcludeUnencrypted causes that frame to be dis-
              carded."
   ::= { dot11PrivacyEntry 6 }
```

In "dot11PrivacyTable TABLE" in Annex D, insert the following attributes after dot11WEPExcludedCount { dot11PrivacyEntry 6 }:

```
dot11RSNAEnabled OBJECT-TYPE
       SYNTAX TruthValue
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "When this object is set to TRUE, this shall indicate that RSNA is
              enabled on this entity. The entity will advertise the RSN Informa-
              tion Element in its Beacon and Probe Response frames. Configuration
              variables for RSNA operation are found in the dot11RSNAConfigTable.

              This object requires that dot11PrivacyInvoked also be set to TRUE."
   ::= { dot11PrivacyEntry 7 }

dot11RSNAPreauthenticationEnabled OBJECT-TYPE
       SYNTAX TruthValue
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "When this object is set to TRUE, this shall indicate that RSNA
              preauthentication is enabled on this entity.

              This object requires that dot11RSNAEnabled also be set to TRUE."
   ::= { dot11PrivacyEntry 8 }
```

In "dot11CountersEntry TABLE" of Annex D, change dot11WEPUndecryptableCount as follows:

```
dot11WEPUndecryptableCount OBJECT-TYPE
       SYNTAX Counter32
       MAX-ACCESS read-only
       STATUS current
```

```
       DESCRIPTION
              "This counter shall increment when a frame is received with the ~~WEP~~
              Protected Frame subfield of the Frame Control field set to one and
              the WEPOn value for the key mapped to the transmitter's MAC address
              indicates that the frame should not have been encrypted or that
              frame is discarded due to the receiving STA not implementing the
              privacy option."
   ::= { dot11CountersEntry 14 }
```

In "Compliance Statements" of Annex D, change the dot11Compliance as follows:

```
dot11Compliance MODULE-COMPLIANCE
       STATUS current
       DESCRIPTION
              "The compliance statement for SNMPv2 entities
              that implement the IEEE 802.11 MIB."
       MODULE  -- this module
       MANDATORY-GROUPS {
              dot11SMTbase~~2~~4,
              dot11MACbase, dot11CountersGroup,
              dot11SmtAuthenticationAlgorithms,
              dot11ResourceTypeID, dot11PhyOperationComplianceGroup }
```

In "Compliance Statements" of Annex D, change OPTIONAL-GROUPS as follows:

```
-- OPTIONAL-GROUPS { dot11SMTprivacy, dot11MACStatistics,
--     dot11PhyAntennaComplianceGroup, dot11PhyTxPowerComplianceGroup,
--     dot11PhyRegDomainsSupportGroup,
--     dot11PhyAntennasListGroup, dot11PhyRateGroup,
--     dot11SMTbase3, dot11MultiDomainCapabilityGroup,
--     dot11PhyFHSSComplianceGroup2, dot11RSNAadditions }
--

::= { dot11Compliances 1 }
```

In "Groups - units of conformance" of Annex D, change dot11SMTbase2 as follows:

```
dot11SMTbase2 OBJECT-GROUP
       OBJECTS { dot11MediumOccupancyLimit,
              dot11CFPollable,
              dot11CFPPeriod,
              dot11CFPMaxDuration,
              dot11AuthenticationResponseTimeOut,
              dot11PrivacyOptionImplemented,
              dot11PowerManagementMode,
              dot11DesiredSSID, dot11DesiredBSSType,
              dot11OperationalRateSet,
              dot11BeaconPeriod, dot11DTIMPeriod,
              dot11AssociationResponseTimeOut,
              dot11DisassociateReason,
              dot11DisassociateStation,
              dot11DeauthenticateReason,
              dot11DeauthenticateStation,
              dot11AuthenticateFailStatus,
              dot11AuthenticateFailStation }
       STATUS ~~current~~deprecated
       DESCRIPTION
              "The SMTbase2 object class provides the necessary support at the
              STA to manage the processes in the STA such that the STA may
              work cooperatively as a part of an IEEE 802.11 network."
   ::= { dot11Groups 18 }
```

In "Groups - units of conformance" in Annex D, insert the following objects after dot11PhyERP-ComplianceGroup { dot11Groups 24 }:

```
dot11RSNAadditions OBJECT-GROUP
       OBJECTS { dot11RSNAEnabled,
              dot11RSNAPreAuthenticationEnabled }
       STATUS current
       DESCRIPTION
              "This object class provides the objects from the IEEE 802.11 MIB
              required to manage RSNA functionality. Note that additional objects
              for managing this functionality are located in the IEEE 802.11 RSN
              MIB."
   ::= { dot11Groups 25 }

dot11SMTbase4 OBJECT-GROUP
       OBJECTS { dot11MediumOccupancyLimit,
              dot11CFPollable,
              dot11CFPPeriod,
              dot11CFPMaxDuration,
              dot11AuthenticationResponseTimeOut,
              dot11PrivacyOptionImplemented,
              dot11PowerManagementMode,
              dot11DesiredSSID, dot11DesiredBSSType,
              dot11OperationalRateSet,
              dot11BeaconPeriod, dot11DTIMPeriod,
              dot11AssociationResponseTimeOut,
              dot11DisassociateReason,
              dot11DisassociateStation,
              dot11DeauthenticateReason,
              dot11DeauthenticateStation,
              dot11AuthenticateFailStatus,
              dot11AuthenticateFailStation,
              dot11MultiDomainCapabilityImplemented,
              dot11MultiDomainCapabilityEnabled,
              dot11CountryString,
              dot11RSNAOptionImplemented }
       STATUS current
       DESCRIPTION
              "The SMTbase4 object class provides the necessary support at the
              IEEE STA to manage the processes in the STA so that the STA may work
              cooperatively as a part of an IEEE 802.11 network."
   ::= { dot11Groups 26 }
```

After the "Groups - units of conformance" section (and before the "End of 802.11 MIB" line) in Annex D, insert the new sections "dot11RSNAConfig Table (RSNA and TSN)," "dot11RSNAConfigPairwiseCiphers Table," "dot11RSNAConfigAuthenticationSuites Table," "dot11RSNAStats Table," "Conformance information - RSN," "Compliance statements - RSN," and "Groups - units of conformance - RSN" as follows:

```
-- ********************************************************************
-- * dot11RSNAConfig TABLE (RSNA and TSN)
-- ********************************************************************

dot11RSNAConfigTable OBJECT-TYPE
       SYNTAX SEQUENCE OF Dot11RSNAConfigEntry
       MAX-ACCESS not-accessible
       STATUS current
       DESCRIPTION
              "The table containing RSNA configuration objects."
   ::= { dot11smt 9 }

dot11RSNAConfigEntry OBJECT-TYPE
```

```
       SYNTAX Dot11RSNAConfigEntry
       MAX-ACCESS not-accessible
       STATUS current
       DESCRIPTION
              "An entry in the dot11RSNAConfigTable."
       INDEX { ifIndex }
   ::= { dot11RSNAConfigTable 1 }

Dot11RSNAConfigEntry ::=
       SEQUENCE {
              dot11RSNAConfigVersion                          Integer32,
              dot11RSNAConfigPairwiseKeysSupported            Unsigned32,
              dot11RSNAConfigGroupCipher                      OCTET STRING,
              dot11RSNAConfigGroupRekeyMethod                 INTEGER,
              dot11RSNAConfigGroupRekeyTime                   Unsigned32,
              dot11RSNAConfigGroupRekeyPackets                Unsigned32,
              dot11RSNAConfigGroupRekeyStrict                 TruthValue,
              dot11RSNAConfigPSKValue                         OCTET STRING,
              dot11RSNAConfigPSKPassPhrase                    DisplayString,
              dot11RSNAConfigGroupUpdateCount                 Unsigned32,
              dot11RSNAConfigPairwiseUpdateCount              Unsigned32,
              dot11RSNAConfigGroupCipherSize                  Unsigned32,
              dot11RSNAConfigPMKLifetime                      Unsigned32,
              dot11RSNAConfigPMKReauthThreshold               Unsigned32,
              dot11RSNAConfigNumberOfPTKSAReplayCounters      INTEGER,
              dot11RSNAConfigSATimeout                        Unsigned32,
              dot11RSNAAuthenticationSuiteSelected            OCTET STRING,
              dot11RSNAPairwiseCipherSelected                 OCTET STRING,
              dot11RSNAGroupCipherSelected                    OCTET STRING,
              dot11RSNAPMKIDUsed                              OCTET STRING,
              dot11RSNAAuthenticationSuiteRequested           OCTET STRING,
              dot11RSNAPairwiseCipherRequested                OCTET STRING,
              dot11RSNAGroupCipherRequested                   OCTET STRING,
              dot11RSNATKIPCounterMeasuresInvoked             Unsigned32,
              dot11RSNA4WayHandshakeFailures                  Unsigned32,
              dot11RSNAConfigNumberOfGTKSAReplayCounters      INTEGER }

-- dot11RSNAConfigEntry 1 has been deprecated.

dot11RSNAConfigVersion OBJECT-TYPE
       SYNTAX Integer32
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "The highest RSNA version this entity supports. See 7.3.2.9."
   ::= { dot11RSNAConfigEntry 2 }

dot11RSNAConfigPairwiseKeysSupported OBJECT-TYPE
       SYNTAX Unsigned32
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "This object indicates how many pairwise keys the entity supports
              for RSNA."
   ::= { dot11RSNAConfigEntry 3 }

dot11RSNAConfigGroupCipher OBJECT-TYPE
       SYNTAX OCTET STRING (SIZE(4))
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "This object indicates the group cipher suite selector the entity
              must use. The group cipher suite in the RSN Information Element
```



```
              shall take its value from this variable. It consists of an OUI (the
              first 3 octets) and a cipher suite identifier (the last octet)."
   ::= { dot11RSNAConfigEntry 4 }

dot11RSNAConfigGroupRekeyMethod OBJECT-TYPE
       SYNTAX INTEGER { disabled(1), timeBased(2), packetBased(3), timepacket-
              Based(4) }
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "This object selects a mechanism for rekeying the RSNA GTK. The
              default is time-based, once per day. Rekeying the GTK is only
              applicable to an entity acting in the Authenticator role (an AP in
              an ESS)."
       DEFVAL { timeBased }
   ::= { dot11RSNAConfigEntry 5 }

dot11RSNAConfigGroupRekeyTime OBJECT-TYPE
       SYNTAX Unsigned32 (1..4294967295)
       UNITS "seconds"
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "The time in seconds after which the RSNA GTK shall be refreshed.
              The timer shall start at the moment the GTK was set using the MLME-
              SETKEYS.request primitive."
       DEFVAL { 86400 } -- once per day
   ::= { dot11RSNAConfigEntry 6 }

dot11RSNAConfigGroupRekeyPackets OBJECT-TYPE
       SYNTAX Unsigned32 (1..4294967295)
       UNITS "1000 packets"
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "A packet count (in 1000s of packets) after which the RSNA GTK
              shall be refreshed. The packet counter shall start at the moment
              the GTK was set using the MLME-SETKEYS.request primitive and it
              shall count all packets encrypted using the current GTK."
   ::= { dot11RSNAConfigEntry 7 }

dot11RSNAConfigGroupRekeyStrict OBJECT-TYPE
       SYNTAX TruthValue
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "This object signals that the GTK shall be refreshed whenever a STA
              leaves the BSS that possesses the GTK."
   ::= { dot11RSNAConfigEntry 8 }

dot11RSNAConfigPSKValue OBJECT-TYPE
       SYNTAX OCTET STRING (SIZE(32))
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "The PSK for when RSNA in PSK mode is the selected AKM suite. In
              that case, the PMK will obtain its value from this object.

              This object is logically write-only. Reading this variable shall
              return unsuccessful status or null or zero."
   ::= { dot11RSNAConfigEntry 9 }

dot11RSNAConfigPSKPassPhrase OBJECT-TYPE
       SYNTAX DisplayString
```

```
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "The PSK, for when RSNA in PSK mode is the selected AKM suite, is
              configured by dot11RSNAConfigPSKValue.

              An alternative manner of setting the PSK uses the password-to-key
              algorithm defined in H.4. This variable provides a means to enter a
              pass-phrase. When this object is written, the RSNA entity shall use
              the password-to-key algorithm specified in H.4 to derive a pre-
              shared and populate dot11RSNAConfigPSKValue with this key.
              This object is logically write-only. Reading this variable shall
              return unsuccessful status or null or zero."
   ::= { dot11RSNAConfigEntry 10 }

-- dot11RSNAConfigEntry 11 and dot11RSNAConfigEntry 12 have been
-- deprecated.

dot11RSNAConfigGroupUpdateCount OBJECT-TYPE
       SYNTAX Unsigned32 (1..4294967295)
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "The number of times Message 1 in the RSNA Group Key Handshake will
              be retried per GTK Handshake attempt."
       DEFVAL { 3 } --
   ::= { dot11RSNAConfigEntry 13 }

dot11RSNAConfigPairwiseUpdateCount OBJECT-TYPE
       SYNTAX Unsigned32 (1..4294967295)
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "The number of times Message 1 and Message 3 in the RSNA 4-Way Hand-
              shake will be retried per 4-Way Handshake attempt."
       DEFVAL { 3 } --
   ::= { dot11RSNAConfigEntry 14 }

dot11RSNAConfigGroupCipherSize OBJECT-TYPE
       SYNTAX Unsigned32 (0..4294967295)
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "This object indicates the length in bits of the group cipher key."
   ::= { dot11RSNAConfigEntry 15 }

dot11RSNAConfigPMKLifetime OBJECT-TYPE
       SYNTAX Unsigned32 (1..4294967295)
       UNITS "seconds"
       MAX-ACCESSread-write
       STATUS current
       DESCRIPTION
              "The maximum lifetime of a PMK in the PMK cache."
       DEFVAL { 43200 } --
   ::= { dot11RSNAConfigEntry 16 }

dot11RSNAConfigPMKReauthThreshold OBJECT-TYPE
       SYNTAX Unsigned32 (1..100)
       UNITS "percentage"
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "The percentage of the PMK lifetime that should expire before an
              IEEE 802.1X reauthentication occurs."
```



```
        DEFVAL { 70 } --
    ::= { dot11RSNAConfigEntry 17 }

dot11RSNAConfigNumberOfPTKSAReplayCounters OBJECT-TYPE
        SYNTAX INTEGER
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "Specifies the number of PTKSA replay counters per association:
                        0 -> 1 replay counter,
                        1 -> 2 replay counters,
                        2 -> 4 replay counters,
                        3 -> 16 replay counters"
    ::= { dot11RSNAConfigEntry 18 }

dot11RSNAConfigSATimeout OBJECT-TYPE
        SYNTAX Unsigned32 (1..4294967295)
        UNITS "seconds"
        MAX-ACCESS read-write
        STATUS current
        DESCRIPTION
                "The maximum time a security association shall take to set up."
        DEFVAL { 60 } --
    ::= { dot11RSNAConfigEntry 19 }

dot11RSNAAuthenticationSuiteSelected OBJECT-TYPE
        SYNTAX OCTET STRING (SIZE(4))
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "The selector of the last AKM suite negotiated."
    ::= { dot11RSNAConfigEntry 20 }

dot11RSNAPairwiseCipherSelected OBJECT-TYPE
        SYNTAX OCTET STRING (SIZE(4))
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "The selector of the last pairwise cipher negotiated."
    ::= { dot11RSNAConfigEntry 21 }

dot11RSNAGroupCipherSelected OBJECT-TYPE
        SYNTAX OCTET STRING (SIZE(4))
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "The selector of the last group cipher negotiated."
    ::= { dot11RSNAConfigEntry 22 }

dot11RSNAPMKIDUsed OBJECT-TYPE
        SYNTAX OCTET STRING (SIZE(16))
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "The selector of the last PMKID used in the last 4-Way Handshake."
    ::= { dot11RSNAConfigEntry 23 }

dot11RSNAAuthenticationSuiteRequested OBJECT-TYPE
        SYNTAX OCTET STRING (SIZE(4))
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "The selector of the last AKM suite requested."
    ::= { dot11RSNAConfigEntry 24 }
```

```
dot11RSNAPairwiseCipherRequested OBJECT-TYPE
       SYNTAX OCTET STRING (SIZE(4))
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "The selector of the last pairwise cipher requested."
   ::= { dot11RSNAConfigEntry 25 }

dot11RSNAGroupCipherRequested OBJECT-TYPE
       SYNTAX OCTET STRING (SIZE(4))
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "The selector of the last group cipher requested."
   ::= { dot11RSNAConfigEntry 26 }

dot11RSNATKIPCounterMeasuresInvoked OBJECT-TYPE
       SYNTAX Unsigned32 (1..4294967295)
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "Counts the number of times that a TKIP MIC failure occurred two
              times within 60 s and TKIP countermeasures were invoked. This
              attribute counts both local and remote MIC failure events reported
              to this STA. It increments every time TKIP countermeasures are
              invoked"
   ::= { dot11RSNAConfigEntry 27 }

dot11RSNA4WayHandshakeFailures OBJECT-TYPE
       SYNTAX Unsigned32 (1..4294967295)
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "Counts the number of 4-Way Handshake failures."
   ::= { dot11RSNAConfigEntry 28 }

dot11RSNAConfigNumberOfGTKSAReplayCounters OBJECT-TYPE
       SYNTAX INTEGER
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "Specifies the number of GTKSA replay counters per association:
                     0 -> 1 replay counter,
                     1 -> 2 replay counters,
                     2 -> 4 replay counters,
                     3 -> 16 replay counters"
   ::= { dot11RSNAConfigEntry 29 }

-- ****************************************************************
-- * End of dot11RSNAConfig TABLE
-- ****************************************************************

-- ****************************************************************
-- * dot11RSNAConfigPairwiseCiphers TABLE
-- ****************************************************************

dot11RSNAConfigPairwiseCiphersTable OBJECT-TYPE
       SYNTAX SEQUENCE OF Dot11RSNAConfigPairwiseCiphersEntry
       MAX-ACCESS not-accessible
       STATUS current
       DESCRIPTION
              "This table lists the pairwise ciphers supported by this entity. It
              allows enabling and disabling of each pairwise cipher by network
```



```
              management. The pairwise cipher suite list in the RSN Information
              Element is formed using the information in this table."
   ::= { dot11smt 10 }

dot11RSNAConfigPairwiseCiphersEntry OBJECT-TYPE
       SYNTAX Dot11RSNAConfigPairwiseCiphersEntry
       MAX-ACCESS not-accessible
       STATUS current
       DESCRIPTION
              "The table entry, indexed by the interface index (or all inter-
              faces) and the pairwise cipher."
       INDEX { dot11RSNAConfigIndex, dot11RSNAConfigPairwiseCipherIndex }
   ::= { dot11RSNAConfigPairwiseCiphersTable 1 }

Dot11RSNAConfigPairwiseCiphersEntry ::=
       SEQUENCE {
              dot11RSNAConfigPairwiseCipherIndex      Unsigned32,
              dot11RSNAConfigPairwiseCipher           OCTET STRING,
              dot11RSNAConfigPairwiseCipherEnabled    TruthValue,
              dot11RSNAConfigPairwiseCipherSize       Unsigned32  }

dot11RSNAConfigPairwiseCipherIndex OBJECT-TYPE
       SYNTAX Unsigned32 (1..4294967295)
       MAX-ACCESS not-accessible
       STATUS current
       DESCRIPTION
              "The auxiliary index into the dot11RSNAConfigPairwiseCiphersTable."
   ::= { dot11RSNAConfigPairwiseCiphersEntry 1 }

dot11RSNAConfigPairwiseCipher OBJECT-TYPE
       SYNTAX OCTET STRING (SIZE(4))
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "The selector of a supported pairwise cipher. It consists of an OUI
              (the first 3 octets) and a cipher suite identifier (the last
              octet)."
   ::= { dot11RSNAConfigPairwiseCiphersEntry 2 }

dot11RSNAConfigPairwiseCipherEnabled OBJECT-TYPE
       SYNTAX TruthValue
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "This object enables or disables the pairwise cipher."
   ::= { dot11RSNAConfigPairwiseCiphersEntry 3 }

dot11RSNAConfigPairwiseCipherSize OBJECT-TYPE
       SYNTAX Unsigned32 (0..4294967295)
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "This object indicates the length in bits of the pairwise cipher
              key. This should be 256 for TKIP and 128 for CCMP."
   ::= { dot11RSNAConfigPairwiseCiphersEntry 4 }

-- ****************************************************************
-- * End of dot11RSNAConfigPairwiseCiphers TABLE
-- ****************************************************************

-- ****************************************************************
-- * dot11RSNAConfigAuthenticationSuites TABLE
-- ****************************************************************
```

```
dot11RSNAConfigAuthenticationSuitesTable OBJECT-TYPE
       SYNTAX SEQUENCE OF Dot11RSNAConfigAuthenticationSuitesEntry
       MAX-ACCESS not-accessible
       STATUS current
       DESCRIPTION
              "This table lists the AKM suites supported by this entity. Each AKM
              suite can be individually enabled and disabled. The AKM suite list
              in the RSN information element is formed using the information in
              this table."
   ::= { dot11smt 11 }

dot11RSNAConfigAuthenticationSuitesEntry OBJECT-TYPE
       SYNTAX Dot11RSNAConfigAuthenticationSuitesEntry
       MAX-ACCESS not-accessible
       STATUS current
       DESCRIPTION
              "An entry (row) in the dot11RSNAConfigAuthenticationSuitesTable."
       INDEX { dot11RSNAConfigAuthenticationSuiteIndex }
   ::= { dot11RSNAConfigAuthenticationSuitesTable 1 }

Dot11RSNAConfigAuthenticationSuitesEntry ::=
       SEQUENCE {
              dot11RSNAConfigAuthenticationSuiteIndex       Unsigned32,
              dot11RSNAConfigAuthenticationSuite            OCTET STRING,
              dot11RSNAConfigAuthenticationSuiteEnabled     TruthValue }

dot11RSNAConfigAuthenticationSuiteIndex OBJECT-TYPE
       SYNTAX Unsigned32 (1..4294967295)
       MAX-ACCESS not-accessible
       STATUS current
       DESCRIPTION
              "The auxiliary variable used as an index into the
              dot11RSNAConfigAuthenticationSuitesTable."
   ::= { dot11RSNAConfigAuthenticationSuitesEntry 1 }

dot11RSNAConfigAuthenticationSuite OBJECT-TYPE
       SYNTAX OCTET STRING (SIZE(4))
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "The selector of an AKM suite. It consists of an OUI (the first 3
              octets) and a cipher suite identifier (the last octet)."
   ::= { dot11RSNAConfigAuthenticationSuitesEntry 2 }

dot11RSNAConfigAuthenticationSuiteEnabled OBJECT-TYPE
       SYNTAX TruthValue
       MAX-ACCESS read-write
       STATUS current
       DESCRIPTION
              "This variable indicates whether the corresponding AKM suite is
              enabled/disabled."
   ::= { dot11RSNAConfigAuthenticationSuitesEntry 3 }

-- ****************************************************************
-- * End of dot11RSNAConfigAuthenticationSuites TABLE
-- ****************************************************************

-- ****************************************************************
-- * dot11RSNAStats TABLE
-- ****************************************************************

dot11RSNAStatsTable OBJECT-TYPE
       SYNTAX SEQUENCE OF Dot11RSNAStatsEntry
       MAX-ACCESS not-accessible
```

```
       STATUS current
       DESCRIPTION
              "This table maintains per-STA statistics in an RSN. The entry with
              dot11RSNAStatsSTAAddress set to FF-FF-FF-FF-FF-FF shall contain
              statistics for broadcast/multicast traffic."
   ::= { dot11smt 12 }

dot11RSNAStatsEntry OBJECT-TYPE
       SYNTAX Dot11RSNAStatsEntry
       MAX-ACCESS not-accessible
       STATUS current
       DESCRIPTION
              "An entry in the dot11RSNAStatsTable."
       INDEX { dot11RSNAConfigIndex, dot11RSNAStatsIndex }
   ::= { dot11RSNAStatsTable 1 }

Dot11RSNAStatsEntry ::=
       SEQUENCE {
              dot11RSNAStatsIndex                     Unsigned32,
              dot11RSNAStatsSTAAddress                MacAddress,
              dot11RSNAStatsVersion                   Unsigned32,
              dot11RSNAStatsSelectedPairwiseCipher    OCTET STRING,
              dot11RSNAStatsTKIPICVErrors             Counter32,
              dot11RSNAStatsTKIPLocalMICFailures      Counter32,
              dot11RSNAStatsTKIPRemoteMICFailures     Counter32,
              dot11RSNAStatsCCMPReplays               Counter32,
              dot11RSNAStatsCCMPDecryptErrors         Counter32,
              dot11RSNAStatsTKIPReplays               Counter32 }

dot11RSNAStatsIndex OBJECT-TYPE
       SYNTAX Unsigned32 (1..4294967295)
       MAX-ACCESS not-accessible
       STATUS current
       DESCRIPTION
              "An auxiliary index into the dot11RSNAStatsTable."
   ::= { dot11RSNAStatsEntry 1 }

dot11RSNAStatsSTAAddress OBJECT-TYPE
       SYNTAX MacAddress
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "The MAC address of the STA to which the statistics in this
              conceptual row belong."
   ::= { dot11RSNAStatsEntry 2 }

dot11RSNAStatsVersion OBJECT-TYPE
       SYNTAX Unsigned32 (1..4294967295)
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "The RSNA version with which the STA associated."
   ::= { dot11RSNAStatsEntry 3 }

dot11RSNAStatsSelectedPairwiseCipher OBJECT-TYPE
       SYNTAX OCTET STRING (SIZE(4))
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "The pairwise cipher suite Selector (as defined in 7.3.29.1) used
              during association, in transmission order."
   ::= { dot11RSNAStatsEntry 4 }

dot11RSNAStatsTKIPICVErrors OBJECT-TYPE
```

```
       SYNTAX Counter32
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "Counts the number of TKIP ICV errors encountered when decrypting
              packets for the STA."
   ::= { dot11RSNAStatsEntry 5 }

dot11RSNAStatsTKIPLocalMICFailures OBJECT-TYPE
       SYNTAX Counter32
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "Counts the number of MIC failures encountered when checking the
              integrity of packets received from the STA at this entity."
   ::= { dot11RSNAStatsEntry 6 }

dot11RSNAStatsTKIPRemoteMICFailures OBJECT-TYPE
       SYNTAX Counter32
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "Counts the number of MIC failures encountered by the STA identi-
              fied by dot11StatsSTAAddress and reported back to this entity."
   ::= { dot11RSNAStatsEntry 7 }

dot11RSNAStatsCCMPReplays OBJECT-TYPE
       SYNTAX Counter32
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "The number of received CCMP MPDUs discarded by the replay
              mechanism."
   ::= { dot11RSNAStatsEntry 8 }

dot11RSNAStatsCCMPDecryptErrors OBJECT-TYPE
       SYNTAX Counter32
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "The number of received MPDUs discarded by the CCMP decryption
              algorithm."
   ::= { dot11RSNAStatsEntry 9 }

dot11RSNAStatsTKIPReplays OBJECT-TYPE
       SYNTAX Counter32
       MAX-ACCESS read-only
       STATUS current
       DESCRIPTION
              "Counts the number of TKIP replay errors detected."
   ::= { dot11RSNAStatsEntry 10 }

-- ********************************************************************
-- * End of dot11RSNAStats TABLE
-- ********************************************************************

-- ********************************************************************
-- * Conformance information - RSN
-- ********************************************************************

-- ********************************************************************
-- * Compliance Statements - RSN
-- ********************************************************************
dot11RSNCompliance MODULE-COMPLIANCE
```

```
        STATUS current
        DESCRIPTION
                "The compliance statement for SNMPv2 entities that implement the
                IEEE 802.11 RSN MIB."
        MODULE -- this module
        MANDATORY-GROUPS {
                dot11RSNBase }

-- OPTIONAL-GROUPS {dot11RSNPMKcachingGroup }
::= { dot11Compliances 2 }

-- ********************************************************************
-- * Groups - units of conformance - RSN
-- ********************************************************************

dot11RSNBase OBJECT-GROUP
        OBJECTS {
                dot11RSNAConfigVersion,
                dot11RSNAConfigPairwiseKeysSupported,
                dot11RSNAConfigGroupCipher,
                dot11RSNAConfigGroupRekeyMethod,
                dot11RSNAConfigGroupRekeyTime,
                dot11RSNAConfigGroupRekeyPackets,
                dot11RSNAConfigGroupRekeyStrict,
                dot11RSNAConfigPSKValue,
                dot11RSNAConfigPSKPassPhrase,
                dot11RSNAConfigGroupUpdateCount,
                dot11RSNAConfigPairwiseUpdateCount,
                dot11RSNAConfigGroupCipherSize,
                dot11RSNAConfigPairwiseCipher,
                dot11RSNAConfigPairwiseCipherEnabled,
                dot11RSNAConfigPairwiseCipherSize,
                dot11RSNAConfigAuthenticationSuite,
                dot11RSNAConfigAuthenticationSuiteEnabled,
                dot11RSNAConfigNumberOfPTKSAReplayCounters,
                dot11RSNAConfigSATimeout,
                dot11RSNAConfigNumberOfGTKSAReplayCounters,
                dot11RSNAAuthenticationSuiteSelected,
                dot11RSNAPairwiseCipherSelected,
                dot11RSNAGroupCipherSelected,
                dot11RSNAPMKIDUsed,
                dot11RSNAAuthenticationSuiteRequested,
                dot11RSNAPairwiseCipherRequested,
                dot11RSNAGroupCipherRequested,
                dot11RSNAStatsSTAAddress,
                dot11RSNAStatsVersion,
                dot11RSNAStatsSelectedPairwiseCipher,
                dot11RSNAStatsTKIPICVErrors,
                dot11RSNAStatsTKIPLocalMICFailures,
                dot11RSNAStatsTKIPRemoteMICFailures,
                dot11RSNAStatsTKIPCounterMeasuresInvoked,
                dot11RSNAStatsCCMPReplays,
                dot11RSNAStatsCCMPDecryptErrors,
                dot11RSNAStatsTKIPReplays,
                dot11RSNAStats4WayHandshakeFailures }
        STATUS current
        DESCRIPTION
                "The dot11RSNBase object class provides the necessary support for
                managing RSNA functionality in the STA."
    ::= { dot11Groups 26 }

dot11RSNPMKcachingGroup OBJECT-GROUP
        OBJECTS {
                dot11RSNAConfigPMKLifetime,
```

```
            dot11RSNAConfigPMKReauthThreshold
            }
      STATUS current
      DESCRIPTION
            "The dot11RSNPMKcachingGroup object class provides the necessary
            support for managing PMK caching functionality in the STA"
   ::= { dot11Groups 27 }
```

Annex E

(informative)

Bibliography

E.1 General

Insert the following references into E.1 and renumber references as appropriate:

[B11] Arazi, E. G., *A Commonsense Approach to the Theory of Error Correcting Codes*, MIT Press, 1988.

[B12] IETF RFC 1305-1992, Network Time Protocol (Version 3) Specification, Implementation and Analysis.

[B13] IETF RFC 2548-1999, Microsoft Vendor-specific RADIUS Attributes.

[B14] IETF RFC 2865-2000, Remote Authentication Dial in User Service (RADIUS).

[B15] IETF RFC 3588-2003, Diameter Base Protocol.

[B16] PKCS #5 v2.0, "Password-Based Cryptography Standard," http://www.rsasecurity.com/rsalabs/node.asp?id=2127.

Insert the following text for Annex H following Annex G:

Annex H

(informative)

RSNA reference implementations and test vectors

H.1 TKIP temporal key mixing function reference implementation and test vector

This clause provides a C-language reference implementation of the temporal key mixing function.

```
/*********************************************************************

   Contents:    Generate IEEE 802.11 per-frame RC4 key hash test vectors
   Date:        April 19, 2002
   Notes:
   This code is written for pedagogical purposes, NOT for performance.

*********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <time.h>

typedef unsigned char  byte;   /*  8-bit byte (octet)  */
typedef unsigned short u16b;   /* 16-bit unsigned word */
typedef unsigned long  u32b;   /* 32-bit unsigned word */

/* macros for extraction/creation of byte/u16b values  */
#define RotR1(v16)    ((((v16) >> 1) & 0x7FFF) ^ (((v16) & 1) << 15))
#define   Lo8(v16)    ((byte)( (v16)       & 0x00FF))
#define   Hi8(v16)    ((byte)(((v16) >> 8) & 0x00FF))
#define  Lo16(v32)    ((u16b)( (v32)       & 0xFFFF))
#define  Hi16(v32)    ((u16b)(((v32) >>16) & 0xFFFF))
#define  Mk16(hi,lo) ((lo) ^ (((u16b)(hi)) << 8))

/* select the Nth 16-bit word of the Temporal Key byte array TK[]   */
#define  TK16(N)     Mk16(TK[2*(N)+1],TK[2*(N)])

/* S-box lookup: 16 bits --> 16 bits */
#define _S_(v16)     (Sbox[0][Lo8(v16)] ^ Sbox[1][Hi8(v16)])

/* fixed algorithm "parameters" */
#define PHASE1_LOOP_CNT   8    /* this needs to be "big enough"     */
#define TA_SIZE           6    /*  48-bit transmitter address       */
#define TK_SIZE          16    /* 128-bit Temporal Key              */
#define P1K_SIZE         10    /*  80-bit Phase1 key                */
#define RC4_KEY_SIZE     16    /* 128-bit RC4KEY (104 bits unknown) */

/* configuration settings */
#define DO_SANITY_CHECK   1    /* validate properties of S-box?     */
```

```
/* 2-byte by 2-byte subset of the full AES S-box table */
const u16b Sbox[2][256]=       /* Sbox for hash (can be in ROM)     */
{ {
   0xC6A5,0xF884,0xEE99,0xF68D,0xFF0D,0xD6BD,0xDEB1,0x9154,
   0x6050,0x0203,0xCEA9,0x567D,0xE719,0xB562,0x4DE6,0xEC9A,
   0x8F45,0x1F9D,0x8940,0xFA87,0xEF15,0xB2EB,0x8EC9,0xFB0B,
   0x41EC,0xB367,0x5FFD,0x45EA,0x23BF,0x53F7,0xE496,0x9B5B,
   0x75C2,0xE11C,0x3DAE,0x4C6A,0x6C5A,0x7E41,0xF502,0x834F,
   0x685C,0x51F4,0xD134,0xF908,0xE293,0xAB73,0x6253,0x2A3F,
   0x080C,0x9552,0x4665,0x9D5E,0x3028,0x37A1,0x0A0F,0x2FB5,
   0x0E09,0x2436,0x1B9B,0xDF3D,0xCD26,0x4E69,0x7FCD,0xEA9F,
   0x121B,0x1D9E,0x5874,0x342E,0x362D,0xDCB2,0xB4EE,0x5BFB,
   0xA4F6,0x764D,0xB761,0x7DCE,0x527B,0xDD3E,0x5E71,0x1397,
   0xA6F5,0xB968,0x0000,0xC12C,0x4060,0xE31F,0x79C8,0xB6ED,
   0xD4BE,0x8D46,0x67D9,0x724B,0x94DE,0x98D4,0xB0E8,0x854A,
   0xBB6B,0xC52A,0x4FE5,0xED16,0x86C5,0x9AD7,0x6655,0x1194,
   0x8ACF,0xE910,0x0406,0xFE81,0xA0F0,0x7844,0x25BA,0x4BE3,
   0xA2F3,0x5DFE,0x80C0,0x058A,0x3FAD,0x21BC,0x7048,0xF104,
   0x63DF,0x77C1,0xAF75,0x4263,0x2030,0xE51A,0xFD0E,0xBF6D,
   0x814C,0x1814,0x2635,0xC32F,0xBEE1,0x35A2,0x88CC,0x2E39,
   0x9357,0x55F2,0xFC82,0x7A47,0xC8AC,0xBAE7,0x322B,0xE695,
   0xC0A0,0x1998,0x9ED1,0xA37F,0x4466,0x547E,0x3BAB,0x0B83,
   0x8CCA,0xC729,0x6BD3,0x283C,0xA779,0xBCE2,0x161D,0xAD76,
   0xDB3B,0x6456,0x744E,0x141E,0x92DB,0x0C0A,0x486C,0xB8E4,
   0x9F5D,0xBD6E,0x43EF,0xC4A6,0x39A8,0x31A4,0xD337,0xF28B,
   0xD532,0x8B43,0x6E59,0xDAB7,0x018C,0xB164,0x9CD2,0x49E0,
   0xD8B4,0xACFA,0xF307,0xCF25,0xCAAF,0xF48E,0x47E9,0x1018,
   0x6FD5,0xF088,0x4A6F,0x5C72,0x3824,0x57F1,0x73C7,0x9751,
   0xCB23,0xA17C,0xE89C,0x3E21,0x96DD,0x61DC,0x0D86,0x0F85,
   0xE090,0x7C42,0x71C4,0xCCAA,0x90D8,0x0605,0xF701,0x1C12,
   0xC2A3,0x6A5F,0xAEF9,0x69D0,0x1791,0x9958,0x3A27,0x27B9,
   0xD938,0xEB13,0x2BB3,0x2233,0xD2BB,0xA970,0x0789,0x33A7,
   0x2DB6,0x3C22,0x1592,0xC920,0x8749,0xAAFF,0x5078,0xA57A,
   0x038F,0x59F8,0x0980,0x1A17,0x65DA,0xD731,0x84C6,0xD0B8,
   0x82C3,0x29B0,0x5A77,0x1E11,0x7BCB,0xA8FC,0x6DD6,0x2C3A,
  },

  {  /* second half of table is byte-reversed version of first! */
   0xA5C6,0x84F8,0x99EE,0x8DF6,0x0DFF,0xBDD6,0xB1DE,0x5491,
   0x5060,0x0302,0xA9CE,0x7D56,0x19E7,0x62B5,0xE64D,0x9AEC,
   0x458F,0x9D1F,0x4089,0x87FA,0x15EF,0xEBB2,0xC98E,0x0BFB,
   0xEC41,0x67B3,0xFD5F,0xEA45,0xBF23,0xF753,0x96E4,0x5B9B,
   0xC275,0x1CE1,0xAE3D,0x6A4C,0x5A6C,0x417E,0x02F5,0x4F83,
   0x5C68,0xF451,0x34D1,0x08F9,0x93E2,0x73AB,0x5362,0x3F2A,
   0x0C08,0x5295,0x6546,0x5E9D,0x2830,0xA137,0x0F0A,0xB52F,
   0x090E,0x3624,0x9B1B,0x3DDF,0x26CD,0x694E,0xCD7F,0x9FEA,
   0x1B12,0x9E1D,0x7458,0x2E34,0x2D36,0xB2DC,0xEEB4,0xFB5B,
   0xF6A4,0x4D76,0x61B7,0xCE7D,0x7B52,0x3EDD,0x715E,0x9713,
   0xF5A6,0x68B9,0x0000,0x2CC1,0x6040,0x1FE3,0xC879,0xEDB6,
   0xBED4,0x468D,0xD967,0x4B72,0xDE94,0xD498,0xE8B0,0x4A85,
   0x6BBB,0x2AC5,0xE54F,0x16ED,0xC586,0xD79A,0x5566,0x9411,
   0xCF8A,0x10E9,0x0604,0x81FE,0xF0A0,0x4478,0xBA25,0xE34B,
   0xF3A2,0xFE5D,0xC080,0x8A05,0xAD3F,0xBC21,0x4870,0x04F1,
   0xDF63,0xC177,0x75AF,0x6342,0x3020,0x1AE5,0x0EFD,0x6DBF,
   0x4C81,0x1418,0x3526,0x2FC3,0xE1BE,0xA235,0xCC88,0x392E,
   0x5793,0xF255,0x82FC,0x477A,0xACC8,0xE7BA,0x2B32,0x95E6,
   0xA0C0,0x9819,0xD19E,0x7FA3,0x6644,0x7E54,0xAB3B,0x830B,
   0xCA8C,0x29C7,0xD36B,0x3C28,0x79A7,0xE2BC,0x1D16,0x76AD,
```

```
   0x3BDB,0x5664,0x4E74,0x1E14,0xDB92,0x0A0C,0x6C48,0xE4B8,
   0x5D9F,0x6EBD,0xEF43,0xA6C4,0xA839,0xA431,0x37D3,0x8BF2,
   0x32D5,0x438B,0x596E,0xB7DA,0x8C01,0x64B1,0xD29C,0xE049,
   0xB4D8,0xFAAC,0x07F3,0x25CF,0xAFCA,0x8EF4,0xE947,0x1810,
   0xD56F,0x88F0,0x6F4A,0x725C,0x2438,0xF157,0xC773,0x5197,
   0x23CB,0x7CA1,0x9CE8,0x213E,0xDD96,0xDC61,0x860D,0x850F,
   0x90E0,0x427C,0xC471,0xAACC,0xD890,0x0506,0x01F7,0x121C,
   0xA3C2,0x5F6A,0xF9AE,0xD069,0x9117,0x5899,0x273A,0xB927,
   0x38D9,0x13EB,0xB32B,0x3322,0xBBD2,0x70A9,0x8907,0xA733,
   0xB62D,0x223C,0x9215,0x20C9,0x4987,0xFFAA,0x7850,0x7AA5,
   0x8F03,0xF859,0x8009,0x171A,0xDA65,0x31D7,0xC684,0xB8D0,
   0xC382,0xB029,0x775A,0x111E,0xCB7B,0xFCA8,0xD66D,0x3A2C,
  }
};

#if DO_SANITY_CHECK
/*
*********************************************************************
* Routine: SanityCheckTable -- verify Sbox properties
*
* Inputs:  Sbox
* Output:  None, but an assertion fails if the tables are wrong
* Notes:
*   The purpose of this routine is solely to illustrate and
*   verify the following properties of the Sbox table:
*      - the Sbox is a "2x2" subset of the AES table:
*            Sbox + affine transform + MDS.
*      - the Sbox table can be easily designed to fit in a
*            512-byte table, using a byte swap
*      - the Sbox table can be easily designed to fit in a
*            256-byte table, using some shifts and a byte swap
*********************************************************************
*/
void SanityCheckTable(void)
    {
    const static int  M_x = 0x11B;   /* AES irreducible polynomial */
    const static byte Sbox8[256] = { /* AES 8-bit Sbox */
        0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,
        0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
        0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,
        0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
        0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,
        0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
        0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,
        0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
        0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,
        0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
        0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,
        0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
        0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,
        0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
        0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,
        0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
        0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,
        0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
        0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,
        0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
        0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,
        0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
```

```
            0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,
            0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
            0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,
            0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
            0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,
            0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
            0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,
            0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
            0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,
            0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16 };

        int i,k,k2,k3;
        byte bitmap[0x2000];

        /* show that smaller tables can be used, if desired */
        for (i=0;i<256;i++)
            {
            k  = Sbox8[i];
            k2 = (k << 1) ^ ((k & 0x80) ? M_x : 0);
            k3 =  k ^ k2;
            assert(Sbox[0][i] == ((k2 << 8) ^ k3));
            assert(Sbox[1][i] == ((k3 << 8) ^ k2));
            }

        /* now make sure that it's a 16-bit permutation */
        memset(bitmap,0,sizeof(bitmap));
        for (i=0;i<0x10000;i++)
            {
            k = _S_(i); /* do an S-box lookup: 16 --> 16 bits */
            assert(k < (1 << 16));
            assert((bitmap[k >> 3] & (1 << (k & 7))) == 0);
            bitmap[k >> 3] |= 1 << (k & 7);
            }
        for (i=0;i<sizeof(bitmap);i++)
            assert(bitmap[i] == 0xFF);

        /* if we reach here, the 16-bit Sbox is ok */
        printf("Table sanity check successful\n");
        }
    #endif

    /*
    **********************************************************************
    * Routine: Phase 1 -- generate P1K, given TA, TK, IV32
    *
    * Inputs:
    *     TK[]      = Temporal Key                         [128 bits]
    *     TA[]      = transmitter's MAC address            [ 48 bits]
    *     IV32      = upper 32 bits of IV                  [ 32 bits]
    * Output:
    *     P1K[]     = Phase 1 key                          [ 80 bits]
    *
    * Note:
    *     This function only needs to be called every 2**16 frames,
    *     although in theory it could be called every frame.
    *
    **********************************************************************
    */
    void Phase1(u16b *P1K,const byte *TK,const byte *TA,u32b IV32)
```

```
    {
    int  i;

    /* Initialize the 80 bits of P1K[] from IV32 and TA[0..5]     */
    P1K[0]      = Lo16(IV32);
    P1K[1]      = Hi16(IV32);
    P1K[2]      = Mk16(TA[1],TA[0]); /* use TA[] as little-endian */
    P1K[3]      = Mk16(TA[3],TA[2]);
    P1K[4]      = Mk16(TA[5],TA[4]);

    /* Now compute an unbalanced Feistel cipher with 80-bit block */
    /* size on the 80-bit block P1K[], using the 128-bit key TK[] */
    for (i=0; i < PHASE1_LOOP_CNT ;i++)
        {                /* Each add operation here is mod 2**16 */
        P1K[0] += _S_(P1K[4] ^ TK16((i&1)+0));
        P1K[1] += _S_(P1K[0] ^ TK16((i&1)+2));
        P1K[2] += _S_(P1K[1] ^ TK16((i&1)+4));
        P1K[3] += _S_(P1K[2] ^ TK16((i&1)+6));
        P1K[4] += _S_(P1K[3] ^ TK16((i&1)+0));
        P1K[4] +=  i;                    /* avoid "slide attacks" */
        }
    }

/*
**********************************************************************
* Routine: Phase 2 -- generate RC4KEY, given TK, P1K, IV16
*
* Inputs:
*     TK[]      = Temporal Key                         [128 bits]
*     P1K[]     = Phase 1 output key                   [ 80 bits]
*     IV16      = low 16 bits of IV counter            [ 16 bits]
* Output:
*     RC4KEY[]  = the key used to encrypt the frame    [128 bits]
*
* Note:
*     The value {TA,IV32,IV16} for Phase1/Phase2 must be unique
*     across all frames using the same key TK value. Then, for a
*     given value of TK[], this TKIP48 construction guarantees that
*     the final RC4KEY value is unique across all frames.
*
* Suggested implementation optimization: if PPK[] is "overlaid"
*     appropriately on RC4KEY[], there is no need for the final
*     for loop below that copies the PPK[] result into RC4KEY[].
*
**********************************************************************
*/
void Phase2(byte *RC4KEY,const byte *TK,const u16b *P1K,u16b IV16)
    {
    int  i;
    u16b PPK[6];                        /* temporary key for mixing    */

    /* all adds in the PPK[] equations below are mod 2**16         */
    for (i=0;i<5;i++) PPK[i]=P1K[i];    /* first, copy P1K to PPK      */
    PPK[5]  =  P1K[4] + IV16;           /* next,  add in IV16          */

    /* Bijective non-linear mixing of the 96 bits of PPK[0..5]         */
    PPK[0] +=    _S_(PPK[5] ^ TK16(0)); /* Mix key in each "round"     */
    PPK[1] +=    _S_(PPK[0] ^ TK16(1));
    PPK[2] +=    _S_(PPK[1] ^ TK16(2));
```

```
    PPK[3] +=    _S_(PPK[2] ^ TK16(3));
    PPK[4] +=    _S_(PPK[3] ^ TK16(4));
    PPK[5] +=    _S_(PPK[4] ^ TK16(5)); /* Total # S-box lookups == 6  */

    /* Final sweep: bijective, linear. Rotates kill LSB correlations   */
    PPK[0] +=  RotR1(PPK[5] ^ TK16(6));
    PPK[1] +=  RotR1(PPK[0] ^ TK16(7)); /* Use all of TK[] in Phase2   */
    PPK[2] +=  RotR1(PPK[1]);
    PPK[3] +=  RotR1(PPK[2]);
    PPK[4] +=  RotR1(PPK[3]);
    PPK[5] +=  RotR1(PPK[4]);
    /* At this point, for a given key TK[0..15], the 96-bit output */
    /*     value PPK[0..5] is guaranteed to be unique, as a function   */
    /*     of the 96-bit "input" value   {TA,IV32,IV16}. That is, P1K  */
    /*     is now a keyed permutation of {TA,IV32,IV16}.               */

    /* Set RC4KEY[0..3], which includes cleartext portion of RC4 key   */
    RC4KEY[0] = Hi8(IV16);              /* RC4KEY[0..2] is the WEP IV  */
    RC4KEY[1] =(Hi8(IV16) | 0x20) & 0x7F; /* Help avoid FMS weak keys  */
    RC4KEY[2] = Lo8(IV16);
    RC4KEY[3] = Lo8((PPK[5] ^ TK16(0)) >> 1);

    /* Copy 96 bits of PPK[0..5] to RC4KEY[4..15]  (little-endian)      */
    for (i=0;i<6;i++)
        {
        RC4KEY[4+2*i] = Lo8(PPK[i]);
        RC4KEY[5+2*i] = Hi8(PPK[i]);
        }
    }

/*
**********************************************************************
* Routine: doTestCase -- execute a test case, and print results
**********************************************************************
*/
void DoTestCase(byte *RC4KEY,u32b IV32,u16b IV16,const byte *TA,const byte
*TK)
    {
    int  i;
    u16b P1K[P1K_SIZE/2];  /* "temp" copy of phase1 key */

    printf("\nTK    =");
    for (i=0;i<TK_SIZE;i++) printf(" %02X",TK[i]);
    printf("\nTA    =");
    for (i=0;i<TA_SIZE;i++) printf(" %02X",TA[i]);
    printf("\nIV32  = %08X   [transmitted as",IV32);  /* show byte order */
    for (i=0;i<4;i++) printf(" %02X",(IV32 >> (24-8*i)) & 0xFF);
    printf("]");
    printf("\nIV16  = %04X",IV16);

    Phase1(P1K,TK,TA,IV32);

    printf("\nP1K   =");
    for (i=0;i<P1K_SIZE/2;i++) printf(" %04X ",P1K[i] & 0xFFFF);

    Phase2(RC4KEY,TK,P1K,IV16);

    printf("\nRC4KEY= ");
    for (i=0;i<RC4_KEY_SIZE;i++) printf("%02X ",RC4KEY[i]);
```

```
    }

/*
********************************************************************
* Static (Repeatable) Test Cases
********************************************************************
*/
void DoStaticTestCases(int testCnt)
    {
    int  i,j;
    byte TA[TA_SIZE],TK[TK_SIZE],RC4KEY[RC4_KEY_SIZE];
    u16b IV16=0;
    u32b IV32=0;

    /* set a fixed starting point */
    for (i=0;i<TK_SIZE;i++) TK[i]=i;
    for (i=0;i<TA_SIZE;i++) TA[i]=(i+1)*17;
    TA[0] = TA[0] & 0xFC;             /* Clear I/G and U/L bits in OUI */

    /* now generate tests, feeding results back into new tests */
    for (i=0; i<testCnt/2; i++)
        {
        printf("\n\nTest vector #%d:",2*i+1);
        DoTestCase(RC4KEY,IV32,IV16,TA,TK);
        IV16++;                      /* emulate per-frame "increment" */
        if (IV16 == 0) IV32++;
        printf("\n\nTest vector #%d:",2*i+2);
        DoTestCase(RC4KEY,IV32,IV16,TA,TK);
        /* feed results back to seed the next test input values    */
        IV16 = (i) ? Mk16(RC4KEY[15],RC4KEY[4]) : 0xFFFF;/* force wrap */
        IV32 =       Mk16(RC4KEY[14],RC4KEY[5]);
        IV32 =       Mk16(RC4KEY[13],RC4KEY[7]) + (IV32 << 16);
        for (j=0;j<TA_SIZE;j++) TA[j]^=RC4KEY[12-j];
        for (j=0;j<TK_SIZE;j++) TK[j]^=RC4KEY[(j+i+1) % RC4_KEY_SIZE] ^
                                       RC4KEY[(j+i+7) % RC4_KEY_SIZE] ;
        TA[0] = TA[0] & 0xFC;         /* Clear I/G and U/L bits in OUI */
        }
    /* comparing the final output is a good check of correctness   */
    printf("\n");
    }

/*
********************************************************************
* Test Cases Generated at Random
********************************************************************
*/
void DoRandomTestCases(int testCnt)
    {
    int  i,j;
    u16b IV16;
    u32b IV32;
    byte TA[TA_SIZE],RC4KEY[RC4_KEY_SIZE],TK[TK_SIZE];

    printf("Random tests:\n");
    /* now generate tests "recursively" */
    for (i=0; i<testCnt; i++)
        {
        IV16 = rand() & 0xFFFF;
        IV32 = rand() + (rand() << 16);
```

```
        for (j=0;j<TK_SIZE;j++) TK[j]=rand() & 0xFF;
        for (j=0;j<TA_SIZE;j++) TA[j]=rand() & 0xFF;
        TA[0] = TA[0] & 0xFC;         /* Clear I/G and U/L bits in OUI */
        printf("\n\nRandom test vector #%d:",i+1);
        DoTestCase(RC4KEY,IV32,IV16,TA,TK);
        }
    printf("\n");
    }

/*
************************************************************************
* Usage text
************************************************************************
*/
#define NUM_TEST_CNT  8
void Usage(void)
    {
    printf(
        "Usage:   TKIP48 [options]\n"
        "Purpose: Generate test vectors for IEEE 802.11 TKIP48\n"
        "Options  -?   -- output this usage text\n"
        "         -r   -- generate test vectors at random\n"
        "         -sN  -- init random seed to N\n"
        "         -tN  -- generate N tests (default = %d)\n",
        NUM_TEST_CNT
        );
    exit(0);
    }

/*
************************************************************************
* Main
************************************************************************
*/
int main(int argc, char **argv)
    {
    char *parg;
    int   i,doRand = 0;
    int   testCnt  = NUM_TEST_CNT;
    u32b  seed     = (u32b) time(NULL);

#if DO_SANITY_CHECK
    SanityCheckTable();
#endif

    for (i=1; i<argc; i++)
        {
        parg = argv[i];
        switch (parg[0])
            {
            case '-':
                switch (parg[1])
                    {
                    case '?':
                    case 'H':
                    case 'h':
                        Usage();
                        return 0;
                    case 'R':
```

```
                        case 'r':  /* generate some random test vectors */
                            doRand  = 1;
                            break;
                        case 'S':
                        case 's':
                            seed    = atoi(parg+2);
                            break;
                        case 'T':
                        case 't':
                            testCnt = atoi(parg+2);
                            break;
                        default:
                            break;
                        }
                    break;
                case '?':
                    Usage();
                    return 0;
                default:
                    printf("Invalid argument: \"%s\"\n", parg);
                    return 1;
                }
            }
        srand(seed);
        if (doRand) printf("Seed = %u\n",seed);

        /* generate some test vectors */
        if (doRand) DoRandomTestCases(testCnt);
        else        DoStaticTestCases(testCnt);

        return 0;
        }
```

H.1.1 Test vectors

The following output is provided to test implementations of the temporal key hash algorithm. All input and output values are shown in hexadecimal.

```
Test vector #1:
TK    = 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F [LSB on left, MSB on right]
TA    = 10-22-33-44-55-66
PN    = 000000000000
IV32  = 00000000
IV16  = 0000
P1K   = 3DD2  016E  76F4  8697  B2E8
RC4KEY= 00 20 00 33 EA 8D 2F 60 CA 6D 13 74 23 4A 66 0B

Test vector #2:
TK    = 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F [LSB on left, MSB on right]
TA    = 10-22-33-44-55-66
PN    = 000000000001
IV32  = 00000000
IV16  = 0001
P1K   = 3DD2  016E  76F4  8697  B2E8
RC4KEY= 00 20 01 90 FF DC 31 43 89 A9 D9 D0 74 FD 20 AA

Test vector #3:
TK    = 63 89 3B 25 08 40 B8 AE 0B D0 FA 7E 61 D2 78 3E [LSB on left, MSB on right]
TA    = 64-F2-EA-ED-DC-25
PN    = 20DCFD43FFFF
```



```
IV32  = 20DCFD43
IV16  = FFFF
P1K   = 7C67  49D7  9724  B5E9  B4F1
RC4KEY= FF 7F FF 93 81 0F C6 E5 8F 5D D3 26 25 15 44 CE

Test vector #4:
TK    = 63 89 3B 25 08 40 B8 AE 0B D0 FA 7E 61 D2 78 3E [LSB on left, MSB on right]
TA    = 64-F2-EA-ED-DC-25
PN    = 20DCFD440000
IV32  = 20DCFD44
IV16  = 0000
P1K   = 5A5D  73A8  A859  2EC1  DC8B
RC4KEY= 00 20 00 49 8C A4 71 FC FB FA A1 6E 36 10 F0 05

Test vector #5:
TK    = 98 3A 16 EF 4F AC B3 51 AA 9E CC 27 1D 73 09 E2 [LSB on left, MSB on right]
TA    = 50-9C-4B-17-27-D9
PN    = F0A410FC058C
IV32  = F0A410FC
IV16  = 058C
P1K   = F2DF  EBB1  88D3  5923  A07C
RC4KEY= 05 25 8C F4 D8 51 52 F4 D9 AF 1A 64 F1 D0 70 21

Test vector #6:
TK    = 98 3A 16 EF 4F AC B3 51 AA 9E CC 27 1D 73 09 E2 [LSB on left, MSB on right]
TA    = 50-9C-4B-17-27-D9
PN    = F0A410FC058D
IV32  = F0A410FC
IV16  = 058D
P1K   = F2DF  EBB1  88D3  5923  A07C
RC4KEY= 05 25 8D 09 F8 15 43 B7 6A 59 6F C2 C6 73 8B 30

Test vector #7:
TK    = C8 AD C1 6A 8B 4D DA 3B 4D D5 B6 54 38 35 9B 05 [LSB on left, MSB on right]
TA    = 94-5E-24-4E-4D-6E
PN    = 8B1573B730F8
IV32  = 8B1573B7
IV16  = 30F8
P1K   = EFF1  3F38  A364  60A9  76F3
RC4KEY= 30 30 F8 65 0D A0 73 EA 61 4E A8 F4 74 EE 03 19

Test vector #8:
TK    = C8 AD C1 6A 8B 4D DA 3B 4D D5 B6 54 38 35 9B 05 [LSB on left, MSB on right]
TA    = 94-5E-24-4E-4D-6E
PN    = 8B1573B730F9
IV32  = 8B1573B7
IV16  = 30F9
P1K   = EFF1  3F38  A364  60A9  76F3
RC4KEY= 30 30 F9 31 55 CE 29 34 37 CC 76 71 27 16 AB 8F
```

H.2 Michael reference implementation and test vectors

H.2.1 Michael test vectors

To ensure correct implementation of Michael, here are some test vectors. These test vectors still have to be confirmed by an independent implementation.

H.2.1.1 Block function

Table H.1 gives some test vectors for the block function.

Table H.1—Test vectors for block function

Input	# times	Output
(00000000, 00000000)	1	(00000000, 00000000)
(00000000, 00000001)	1	(c00015a8, c0000b95)
(00000001, 00000000)	1	(6b519593, 572b8b8a)
(01234567, 83659326)	1	(441492c2, 1d8427ed)
(00000001, 00000000)	1000	(9f04c4ad, 2ec6c2bf)

The first four rows give test vectors for a single application of the block function *b*. The last row gives a test vector for 1000 repeated applications of the block function. Together these should provide adequate test coverage.

H.2.1.2 Michael

Table H.2 gives some test vectors for Michael.

Table H.2—Test vectors for Michael

Key	Message	Output
0000000000000000	""	82925c1ca1d130b8
82925c1ca1d130b8	"M”	434721ca40639b3f
434721ca40639b3f	"Mi “	e8f9becae97e5d29
E8f9becae97e5d29	"Mic"	90038fc6cf13c1db
90038fc6cf13c1db	"Mich"	D55e100510128986
D55e100510128986	"Michael"	0a942b124ecaa546

Note that each key is the result of the previous line, which makes it easy to construct a single test out of all of these test cases.

H.2.2 Sample code for Michael

```
//
// Michael.h    Reference implementation for Michael

//
// A Michael object implements the computation of the MIC.
//
// Conceptually, the object stores the message to be authenticated.
// At construction the message is empty.
// The append() method appends bytes to the message.
// The getMic() method computes the MIC over the message and returns the
// result.
```

```
// As a side-effect it also resets the stored message
// to the empty message so that the object can be re-used
// for another MIC computation.

class Michael
{

public:
    // Constructor requires a pointer to 8 bytes of key
    Michael( Byte * key );

    // Destructor
    ~Michael();

    // Clear the internal message,
    // resets the object to the state just after construction.
    void clear();

    // Set the key to a new value
    void setKey( Byte * key );

    // Append bytes to the message to be MICed
    void append( Byte * src, int nBytes );

    // Get the MIC result. Destination should accept 8 bytes of result.
    // This also resets the message to empty.
    void getMIC( Byte * dst );

    // Run the test plan to verify proper operations
    static void runTestPlan();

private:
    // Copy constructor declared but not defined,
    //avoids compiler-generated version.
    Michael( const Michael & );
    // Assignment operator declared but not defined,
    //avoids compiler-generated version.
    void operator=( const Michael & );

    // A bunch of internal functions

    // Get UInt32 from 4 bytes LSByte first
    static UInt32 getUInt32( Byte * p );

    // Put UInt32 into 4 bytes LSByte first
    static void putUInt32( Byte * p, UInt32 val );

    // Add a single byte to the internal message
    void appendByte( Byte b );

    // Conversion of hex string to binary string
    static void hexToBin( char *src, Byte * dst );

    // More conversion of hex string to binary string
    static void hexToBin( char *src, int nChars, Byte * dst );

    // Helper function for hex conversion
    static Byte hexToBinNibble( char c );
```

```
    // Run a single test case
    static void runSingleTest( char * cKey, char * cMsg, char * cResult );

    UInt32  K0, K1;         // Key
    UInt32  L, R;           // Current state
    UInt32  M;              // Message accumulator (single word)
    int     nBytesInM;      // # bytes in M
};

//
// Michael.cpp  Reference implementation for Michael
//

// Adapt these typedefs to your local platform
typedef unsigned long UInt32;
typedef unsigned char Byte;

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "Michael.h"

// Rotation functions on 32 bit values
#define ROL32( A, n ) \
 ( ((A) << (n)) | ( ((A)>>(32-(n)))  & ( (1UL << (n)) - 1 ) ) )
#define ROR32( A, n ) ROL32( (A), 32-(n) )

UInt32 Michael::getUInt32( Byte * p )
// Convert from Byte[] to UInt32 in a portable way
{
    UInt32 res = 0;
    for( int i=0; i<4; i++ )
    {
        res |= (*p++) << (8*i);
    }
    return res;
}

void Michael::putUInt32( Byte * p, UInt32 val )
// Convert from UInt32 to Byte[] in a portable way
{
    for( int i=0; i<4; i++ )
    {
        *p++ = (Byte) (val & 0xff);
        val >>= 8;
    }
}

void Michael::clear()
{
    // Reset the state to the empty message.
    L = K0;
    R = K1;
    nBytesInM = 0;
    M = 0;
}
```

```
void Michael::setKey( Byte * key )
{
    // Set the key
    K0 = getUInt32( key );
    K1 = getUInt32( key + 4 );
    // and reset the message
    clear();
}

Michael::Michael( Byte * key )
{
    setKey( key );
}

Michael::~Michael()
{
    // Wipe the key material
    K0 = 0;
    K1 = 0;

    // And the other fields as well.
    //Note that this sets (L,R) to (K0,K1) which is just fine.
    clear();
}

void Michael::appendByte( Byte b )
{
    // Append the byte to our word-sized buffer
    M |= b << (8*nBytesInM);
    nBytesInM++;
    // Process the word if it is full.
    if( nBytesInM >= 4 )
    {
        L ^= M;
        R ^= ROL32( L, 17 );
        L += R;
        R ^= ((L & 0xff00ff00) >> 8) | ((L & 0x00ff00ff) << 8);
        L += R;
        R ^= ROL32( L, 3 );
        L += R;
        R ^= ROR32( L, 2 );
        L += R;
        // Clear the buffer
        M = 0;
        nBytesInM = 0;
    }
}

void Michael::append( Byte * src, int nBytes )
{
    // This is simple
    while( nBytes > 0 )
    {
        appendByte( *src++ );
        nBytes--;
    }
}

void Michael::getMIC( Byte * dst )
```

```
{
    // Append the minimum padding
    appendByte( 0x5a );
    appendByte( 0 );
    appendByte( 0 );
    appendByte( 0 );
    appendByte( 0 );
    // and then zeroes until the length is a multiple of 4
    while( nBytesInM != 0 )
    {
        appendByte( 0 );
    }
    // The appendByte function has already computed the result.
    putUInt32( dst, L );
    putUInt32( dst+4, R );
    // Reset to the empty message.
    clear();
}

void Michael::hexToBin( char *src, Byte * dst )
{
    // Simple wrapper
    hexToBin( src, strlen( src ), dst );
}

void Michael::hexToBin( char *src, int nChars, Byte * dst )
{
    assert( (nChars & 1) == 0 );
    int nBytes = nChars/2;

    // Straightforward conversion
    for( int i=0; i<nBytes; i++ )
    {
        dst[i] = (Byte)((hexToBinNibble( src[0] ) << 4) |
            hexToBinNibble( src[1] ));
        src += 2;
    }
}

Byte Michael::hexToBinNibble( char c )
{
    if( '0' <= c && c <= '9' )
    {
        return (Byte)(c - '0');
    }
    // Make it upper case
    c &= ~('a'-'A');

    assert( 'A' <= c && c <= 'F' );
    return (Byte)(c - 'A' + 10);
}

void Michael::runSingleTest( char * cKey, char * cMsg, char * cResult )
{
    Byte key[ 8 ];
    Byte result[ 8 ];
    Byte res[ 8 ];

    // Convert key and result to binary form
```

```
    hexToBin( cKey, key );
    hexToBin( cResult, result );

    // Compute the MIC value
    Michael mic( key );
    mic.append( (Byte *)cMsg, strlen( cMsg) );
    mic.getMIC( res );

    // Check that it matches
    assert( memcmp( res, result, 8 ) == 0 );
}

void Michael::runTestPlan()
// As usual, test plans can be quite tedious but this should
// ensure that the implementation runs as expected.
{
    Byte key[8] ;
    Byte msg[12];
    int i;

    // First we test the test vectors for the block function

    // The case (0,0)
    putUInt32( key, 0 );
    putUInt32( key+4, 0 );
    putUInt32( msg, 0 );

    Michael mic( key );
    mic.append( msg, 4 );

    assert( mic.L == 0 && mic.R == 0 );

    // The case (0,1)
    putUInt32( key, 0 );
    putUInt32( key+4, 1 );
    mic.setKey( key );
    mic.append( msg, 4 );

    assert( mic.L == 0xc00015a8 && mic.R == 0xc0000b95 );

    // The case (1,0)
    putUInt32( key, 1 );
    putUInt32( key+4, 0 );
    mic.setKey( key );
    mic.append( msg, 4 );

    assert( mic.L == 0x6b519593 && mic.R == 0x572b8b8a );

    // The case (01234567, 83659326)
    putUInt32( key, 0x01234567 );
    putUInt32( key+4, 0x83659326 );
    mic.setKey( key );
    mic.append( msg, 4 );

    assert( mic.L == 0x441492c2 && mic.R == 0x1d8427ed );

    // The repeated case
    putUInt32( key, 1 );
    putUInt32( key+4,0 );
```

```
    mic.setKey( key );

    for( i=0; i<1000; i++ )
    {
        mic.append( msg, 4 );
    }

    assert( mic.L == 0x9f04c4ad && mic.R == 0x2ec6c2bf );

    // And now for the real test cases
    runSingleTest( "0000000000000000", ""         , "82925c1ca1d130b8" );
    runSingleTest( "82925c1ca1d130b8", "M"        , "434721ca40639b3f" );
    runSingleTest( "434721ca40639b3f", "Mi"       , "e8f9becae97e5d29" );
    runSingleTest( "e8f9becae97e5d29", "Mic"      , "90038fc6cf13c1db" );
    runSingleTest( "90038fc6cf13c1db", "Mich"     , "d55e100510128986" );
    runSingleTest( "d55e100510128986", "Michael"  , "0a942b124ecaa546" );
}
```

H.3 PRF reference implementation and test vectors

H.3.1 PRF reference code

```
/*
 * PRF -- Length of output is in octets rather than bits
 *     since length is always a multiple of 8 output array is
 *     organized so first N octets starting from 0 contains PRF output
 *
 *     supported inputs are 16, 24, 32, 48, 64
 *     output array must be 80 octets to allow for sha1 overflow
 */
void PRF(
    unsigned char *key, int key_len,
    unsigned char *prefix, int prefix_len,
    unsigned char *data, int data_len,
    unsigned char *output, int len)
{
    int i;
    unsigned char input[1024]; /* concatenated input */
    int currentindex = 0;
    int total_len;

    memcpy(input, prefix, prefix_len);
    input[prefix_len] = 0; /* single octet 0 */
    memcpy(&input[prefix_len+1], data, data_len);
    total_len = prefix_len + 1 + data_len;
    input[total_len] = 0; /* single octet count, starts at 0 */
    total_len++;
    for (i = 0; i < (len+19)/20; i++) {
        hmac_sha1(input, total_len, key, key_len,
            &output[currentindex]);
        currentindex += 20;/* next concatenation location */
        input[total_len-1]++; /* increment octet count */
    }
}
```

H.3.2 PRF test vectors

```
Test case 1
Key    0x0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b
Key length20
Prefix "prefix"
Prefix length6
Data   "Hi There"
Data length8
PRF-512 0xbcd4c650b30b9684951829e0d75f9d54
       0xb862175ed9f00606e17d8da35402ffee
       0x75df78c3d31e0f889f012120c0862beb
       0x67753e7439ae242edb8373698356cf5a

Test case 2
Key    "Jefe"
Key length4
Prefix "prefix"
Prefix length6
Data   "what do ya want for nothing?"
Data length28
PRF-512 0x51f4de5b33f249adf81aeb713a3c20f4
       0xfe631446fabdfa58244759ae58ef9009
       0xa99abf4eac2ca5fa87e692c440eb4002
       0x3e7babb206d61de7b92f41529092b8fc

Test case 3
Key    0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Key length20
Prefix "prefix"
Prefix length6
Data   0xdd repeated 50 times
Data length50
PRF-5120xe1ac546ec4cb636f9976487be5c86be1
       0x7a0252ca5d8d8df12cfb0473525249ce
       0x9dd8d177ead710bc9b590547239107ae
       0xf7b4abd43d87f0a68f1cbd9e2b6f7607
```

H.4 Suggested pass-phrase-to-PSK mapping

H.4.1 Introduction

The RSNA PSK consists of 256 bits, or 64 octets when represented in hex. It is difficult for a user to correctly enter 64 hex characters. Most users, however, are familiar with passwords and pass-phrases and feel more comfortable entering them than entering keys. A user is more likely to be able to enter an ASCII password or pass-phrase, even though doing so limits the set of possible keys. This suggests that the best that can be done is to introduce a pass-phrase to PSK mapping.

This clause defines a pass-phrase–to–PSK mapping that is the recommended practice for use with RSNAs. This pass-phrase mapping was introduced to encourage users unfamiliar with cryptographic concepts to enable the security features of their WLAN.

A pass-phrase typically has about 2.5 bits of security per character, so the pass-phrase mapping converts an n octet password into a key with about $2.5n + 12$ bits of security. Hence, it provides a relatively low level of security, with keys generated from short passwords subject to dictionary attack. Use of the key hash is recommended only where it is impractical to make use of a stronger form of user authentication. A key generated from a pass-phrase of less than about 20 characters is unlikely to deter attacks.

The pass-phrase mapping defined in this subclause uses the PBKDF2 method from PKCS [B16].

$$PSK = \text{PBKDF2}(PassPhrase, ssid, ssidLength, 4096, 256)$$

Here, the following assumptions apply:

— A pass-phrase is a sequence of between 8 and 63 ASCII-encoded characters. The limit of 63 comes from the desire to distinguish between a pass-phrase and a PSK displayed as 64 hexadecimal characters.

— Each character in the pass-phrase must have an encoding in the range of 32 to 126 (decimal), inclusive.

— *ssid* is the SSID of the ESS or IBSS where this pass-phrase is in use, encoded as an octet string used in the Beacon and Probe Response frames for the ESS or IBSS.

— *ssidLength* is the number of octets of the *ssid*.

— 4096 is the number of times the pass-phrase is hashed.

— 256 is the number of bits output by the pass-phrase mapping.

H.4.2 Reference implementation

```
/*
 * F(P, S, c, i) = U1 xor U2 xor ... Uc
 * U1 = PRF(P, S || Int(i))
 * U2 = PRF(P, U1)
 * Uc = PRF(P, Uc-1)
 */

void F(
    char *password,
    unsigned char *ssid,
    int ssidlength,
    int iterations,
    int count,
    unsigned char *output)
{
    unsigned char digest[36], digest1[A_SHA_DIGEST_LEN];
    int i, j;

    for (i = 0; i < strlen(password); i++) {
        assert((password[i] >= 32) && (password[i] <= 126));
    }

    /* U1 = PRF(P, S || int(i)) */
    memcpy(digest, ssid, ssidlength);
    digest[ssidlength] = (unsigned char)((count>>24) & 0xff);
    digest[ssidlength+1] = (unsigned char)((count>>16) & 0xff);
    digest[ssidlength+2] = (unsigned char)((count>>8) & 0xff);
    digest[ssidlength+3] = (unsigned char)(count & 0xff);
    hmac_sha1(digest, ssidlength+4, (unsigned char*) password,
        (int) strlen(password), digest, digest1);

    /* output = U1 */
    memcpy(output, digest1, A_SHA_DIGEST_LEN);

    for (i = 1; i < iterations; i++) {
        /* Un = PRF(P, Un-1) */
```



```
            hmac_sha1(digest1,  A_SHA_DIGEST_LEN,  (unsigned  char*)  password,
                (int) strlen(password), digest);
            memcpy(digest1, digest, A_SHA_DIGEST_LEN);

            /* output = output xor Un */
            for (j = 0; j < A_SHA_DIGEST_LEN; j++) {
                output[j] ^= digest[j];
            }
        }
    }

    /*
     * password - ascii string up to 63 characters in length
     * ssid - octet string up to 32 octets
     * ssidlength - length of ssid in octets
     * output must be 40 octets in length and outputs 256 bits of key
     */
    int PasswordHash (
        char *password,
        unsigned char *ssid,
        int ssidlength,
        unsigned char *output)
    {
        if ((strlen(password) > 63) || (ssidlength > 32))
            return 0;

        F(password, ssid, ssidlength, 4096, 1, output);
        F(password, ssid, ssidlength, 4096, 2,
            &output[A_SHA_DIGEST_LEN]);
        return 1;
    }
```

H.4.3 Test vectors

```
Test case 1
Passphrase = "password"
SSID = { 'I', 'E', 'E', 'E' }
SSIDLength = 4
PSK = f42c6fc52df0ebef9ebb4b90b38a5f90 2e83fe1b135a70e23aed762e9710a12e

Test case 2
Passphrase = "ThisIsAPassword"
SSID = { 'T', 'h', 'i', 's', 'I', 's', 'A', 'S', 'S', 'I', 'D' }
SSIDLength = 11
PSK = 0dc0d6eb90555ed6419756b9a15ec3e3 209b63df707dd508d14581f8982721af

Test case 3
Password = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
SSID = {'Z','Z','Z','Z', 'Z','Z','Z','Z', 'Z','Z','Z','Z', 'Z','Z','Z','Z',
            'Z','Z','Z','Z', 'Z','Z','Z','Z', 'Z','Z','Z','Z', 'Z','Z','Z','Z'}
SSIDLength = 32
PSK = becb93866bb8c3832cb777c2f559807c 8c59afcb6eae734885001300a981cc62
```

H.5 Suggestions for random number generation

In order to properly implement cryptographic protocols, every platform needs the ability to generate cryptographic-quality random numbers. IETF RFC 1750 explains the notion of cryptographic-quality random numbers and provides advice on ways to harvest suitable randomness. It recommends sampling

multiple sources, each of which contains some randomness, and by passing the complete set of samples through a PRF. By following this advice, an implementation can usually collect enough randomness to distill into a seed for a PRNG whose output will be unpredictable.

This clause suggests two sample techniques that can be combined with the other recommendations of IETF RFC 1750 to harvest randomness. The first method is a software solution that can be implemented on most hardware; the second is a hardware-assisted solution. These solutions are expository only, to demonstrate that it is feasible to harvest randomness on any IEEE 802.11 platform. They are not mutually exclusive, and they do not preclude the use of other sources of randomness when available, such as a noisy diode in a power circuit; in this case, the more the merrier. As many sources of randomness as possible should be gathered into a buffer, and then hashed, to obtain a seed for the PRNG.

H.5.1 Software sampling

Due to the nature of clock circuits in modern electronics, there will be some lack of correlation between two clocks in two different pieces of equipment, even when high-quality crystals are used—crystal clocks are subject to jitter, noise, drift, and frequency mismatch. This randomness may be as little as the placement of the clock waveform edges. Even if one entity were to attempt to synchronize itself to another entity's clock, the correlation cannot be perfect due to noise and uncertainties of the synchronization.

Two clock circuits in the same piece of equipment may synchronize in frequency, but again the correlation will not be perfect due to the noise and jitter of the circuits.

The randomness between the two clocks may not be much per sample—a tenth of a bit or less—but enough samples may be collected to gather enough randomness to form a seed.

A device can use software methods to take advantage of this lack of synchronization, to collect randomness from different sources. As an example, an AP might measure the frame arrival times on Ethernet wireless ports. There is always some amount of traffic on modern Ethernets: ARPs, DHCP requests, NetBIOS advertisements, etc. The sample algorithm in this subclause uses this traffic. In the example, an AP obtains randomness from the available traffic. If Ethernet traffic is available, the AP utilizes that for a source of randomness. Otherwise, it waits for the first association and creates traffic from which it can obtain randomness.

The clocks used to time the frames should be the highest resolution available, preferable 1 ms resolution or better. The clock used to time frame arrival should not be related to the clock used for frame serialization.

```
Initialize result to empty array
LoopCounter = 0
Wait until Ethernet traffic or association
Repeat until global key counter "random enough" or 32 times {
    result = PRF-256(0, "Init Counter",
    Local Mac Address || Time || result || LoopCounter)
    LoopCounter++
    Repeat 32 times {
        If Ethernet traffic available then {
            Take lowest byte of time when Ethernet packet is seen
            Concatenate the seen time onto result
        } else {
            Start 4-Way Handshake; after receipt of Message 2, deauthenticate
            Take lowest byte of time of when Message 1 is sent
            Take lowest byte of time of when Message 2 is received
            Take lowest byte of RSSI from Message 2
```



```
                    Take SNonce from Message 2
                    Concatenate the sent time; receive time, RSSI and SNonce onto result
                }
            }
        }
        Global key counter = result = PRF-256(0, "Init Counter",
        Local Mac Address || Time || result || LoopCounter)
```

NOTE—The Time may be 0 if it is not available.

H.5.2 Hardware-assisted solution

The sample implementation in this subclause uses hardware ring oscillators to generate randomness, as depicted in Figure H.1.

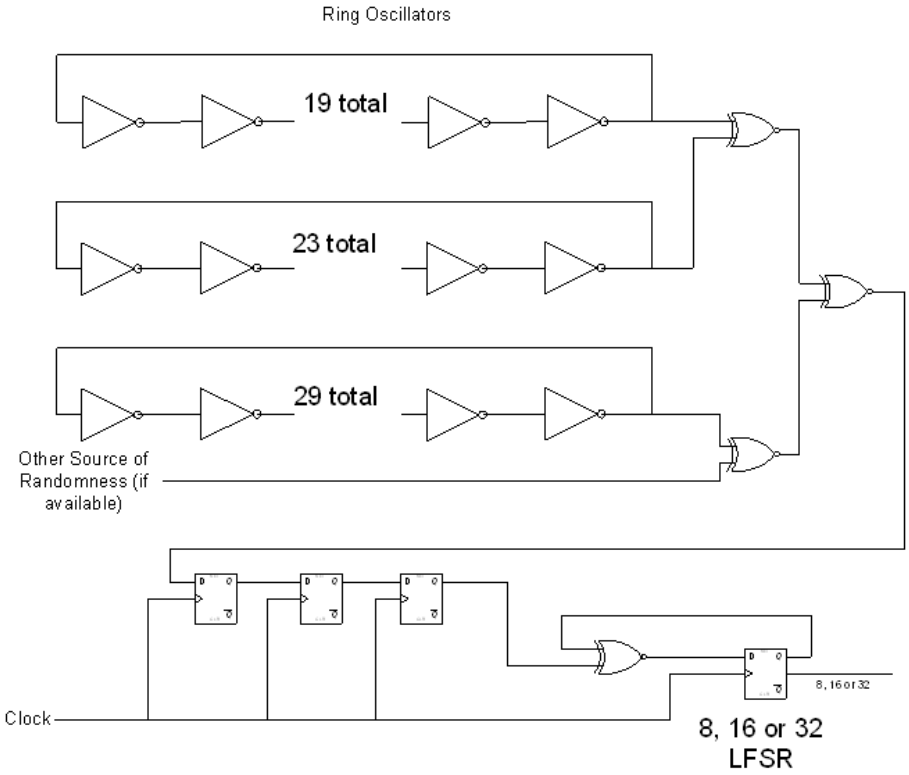


Figure H.1—Randomness generating circuit

The circuit in Figure H.1 generates randomness. The clock input should be about the same frequency as the ring oscillator's natural frequencies. The LFSR should be chosen to be one that is maximal length. Sample LFSRs can be found in Arazi [B11].

The three ring oscillators should be isolated from each other as much as possible to avoid harmonic locking between them. In addition, the three ring oscillators should not be near any other clock circuitry within the system to avoid these ring oscillators' locking to system clocks. The oscillators should be tested to ensure that their output is not correlated.

The output of the LFSR is read by software and concatenated until enough randomness is collected. As a rule of thumb, reading from the LFSR 8 to 16 times the number of bits as the desired number of random bits is sufficient.

```
Initialize result to empty array
Repeat 1024 times {
      Read LFSR
      result = result | LFSR
      Wait a time period
}
Global key counter = PRF-256(0, "Init Counter", result)
```

H.6 Additional test vectors

H.6.1 Notation

In the examples in H.6, frames are represented as a stream of octets, each octet in hex notation, sometimes with text annotation. The order of transmission for octets is left to right, top to bottom. For example, consider the following representation of a frame in Table H.3.

Table H.3—Notation example

Description #1	00 01 02 03
	04 05
Description #2	06 07 08

The frame consists of 9 octets, represented in hex notation as “00”, “01”, ..., “08”. The octet represented by “00” is transmitted first, and the octet represented by “08” is transmitted last. Similar tables are used for other purposes, such as describing a cryptographic operation.

In the text discussion outside of tables, integer values are represented in either hex notation using a “0x” prefix or in decimal notation using no prefix. For example, the hex notation 0x12345 and the decimal notation 74565 represent the same integer value.

H.6.2 WEP encapsulation

The discussion in this subclause represents an RC4 encryption using a table that shows the key, plaintext input, and cipher text output. The MPDU data, prior to WEP encapsulation, is shown in Table H.4.

Table H.4—Sample plaintext MPDU

MPDU data	aa aa 03 00 00 00 08 00 45 00 00 4e 66 1a 00 00 80 11 be 64 0a 00 01 22 0a ff ff ff 00 89 00 89 00 3a 00 00 80 a6 01 10 00 01 00 00 00 00 00 00 20 45 43 45 4a 45 48 45 43 46 43 45 50 46 45 45 49 45 46 46 43 43 41 43 41 43 41 43 41 43 41 41 41 00 00 20 00 01

RC4 encryption is performed as shown in Table H.5.

Table H.5—RC4 encryption

Key	fb 02 9e 30 31 32 33 34
Plaintext	aa aa 03 00 00 00 08 00 45 00 00 4e 66 1a 00 00 80 11 be 64 0a 00 01 22 0a ff ff ff 00 89 00 89 00 3a 00 00 80 a6 01 10 00 01 00 00 00 00 00 00 20 45 43 45 4a 45 48 45 43 46 43 45 50 46 45 45 49 45 46 46 43 43 41 43 41 43 41 43 41 43 41 41 41 00 00 20 00 01 1b d0 b6 04
Ciphertext	f6 9c 58 06 bd 6c e8 46 26 bc be fb 94 74 65 0a ad 1f 79 09 b0 f6 4d 5f 58 a5 03 a2 58 b7 ed 22 eb 0e a6 49 30 d3 a0 56 a5 57 42 fc ce 14 1d 48 5f 8a a8 36 de a1 8d f4 2c 53 80 80 5a d0 c6 1a 5d 6f 58 f4 10 40 b2 4b 7d 1a 69 38 56 ed 0d 43 98 e7 ae e3 bf 0e 2a 2c a8 f7

The plaintext consists of the MPDU data, followed by a 4-octet CRC-32 calculated over the MPDU data.

The expanded MPDU, after WEP encapsulation, is shown in Table H.6.

Table H.6—Expanded MPDU after WEP encapsulation

IV	fb 02 9e 80
MPDU data	f6 9c 58 06 bd 6c e8 46 26 bc be fb 94 74 65 0a ad 1f 79 09 b0 f6 4d 5f 58 a5 03 a2 58 b7 ed 22 eb 0e a6 49 30 d3 a0 56 a5 57 42 fc ce 14 1d 48 5f 8a a8 36 de a1 8d f4 2c 53 80 80 5a d0 c6 1a 5d 6f 58 f4 10 40 b2 4b 7d 1a 69 38 56 ed 0d 43 98 e7 ae e3 bf 0e
ICV	2a 2c a8 f7

The IV consists of the first 3 octets of the RC4 key, followed by an octet containing the Key ID value in the upper 2 bits. In this example, the Key ID value is 2. The MPDU data consists of the cipher text, excluding the last 4 octets. The ICV consists of the last 4 octets of the cipher text, which is the encrypted CRC-32 value.

H.6.3 TKIP test vector

An example of a TKIP MSDU is provided in Table H.7 and Table H.8. The key and PN are used to create the IV, Phase1, and Phase2 keys.

Table H.7—Sample TKIP parameters

Key	12 34 56 78 90 12 34 56 78 90 12 34 56 78 90 12 34 56 78 90 12 34 56 78 90 12 34
PN	0x000000000001

Table H.7—Sample TKIP parameters *(continued)*

IV	00 20 01 20 00 00 00 00
Phase1	bb 58 07 1f 9e 93 b4 38 25 4b
Phase2	00 20 01 4c fe 67 be d2 7c 86 7b 1b f8 02 8b 1c

Table H.8—Sample plaintext and ciphertext MPDUs, using parameter from Table H.7

Plaintext MPDU with TKIP MIC	08 42 2c 00 02 03 04 05 06 08 02 03 04 05 06 07 02 03 04 05 06 07 d0 02 00 20 01 20 00 00 00 00 aa aa 03 00 00 00 08 00 45 00 00 54 00 00 40 00 40 01 a5 55 c0 a8 0a 02 c0 a8 0a 01 08 00 3a b0 00 00 00 00 cd 4c 05 00 00 00 00 00 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35 36 37 68 81 a3 f3 d6 48 d0 3c
Encrypted MPDU with MIC and ICV	08 42 2c 00 02 03 04 05 06 08 02 03 04 05 06 07 02 03 04 05 06 07 d0 02 00 20 01 20 00 00 00 00 c0 0e 14 fc e7 cf ab c7 75 47 e6 66 e5 7c 0d ac 70 4a 1e 35 8a 88 c1 1c 8e 2e 28 2e 38 01 02 7a 46 56 05 5e e9 3e 9c 25 47 02 e9 73 58 05 dd b5 76 9b a7 3f 1e bb 56 e8 44 ef 91 22 85 d3 dd 6e 54 1e 82 38 73 55 8a db a0 79 06 8a bd 7f 7f 50 95 96 75 ac c4 b4 de 9a a9 9c 05 f2 89 a7 c5 2f ee 5b fc 14 f6 f8 e5 f8

H.6.4 CCMP test vector

```
==== CCMP test mpdu ====

-- MPDU Fields

Version  = 0
Type     = 2   SubType  = 0  Data
ToDS     = 0   FromDS   = 0
MoreFrag = 0   Retry    = 1
PwrMgt   = 0   moreData = 0
Encrypt  = 1
Order    = 0
Duration = 11459
A1 = 0f-d2-e1-28-a5-7c    DA
A2 = 50-30-f1-84-44-08    SA
A3 = ab-ae-a5-b8-fc-ba    BSSID
SC = 0x3380
seqNum = 824 (0x0338)  fraqNum = 0 (0x00)
Algorithm = AES_CCM
Key ID = 0

TK =   c9 7c 1f 67 ce 37 11 85  51 4a 8a 19 f2 bd d5 2f

PN =   199027030681356  (0xB5039776E70C)

802.11 Header =08 48 c3 2c 0f d2 e1 28 a5 7c 50 30 f1 84 44 08 ab ae a5 b8 fc ba
            80 33
```

```
Muted 802.11 Header =08 40 0f d2 e1 28 a5 7c 50 30 f1 84 44 08 ab ae a5 b8 fc ba
             00 00

CCMP Header =0c e7 00 20 76 97 03 b5

CCM Nonce =00 50 30 f1 84 44 08 b5  03 97 76 e7 0c

Plaintext Data =f8 ba 1a 55 d0 2f 85 ae 96 7b b6 2f b6 cd a8 eb 7e 78 a0 50

CCM MIC =78 45 ce 0b 16 f9 76 23

-- Encrypted MPDU with FCS

08 48 c3 2c 0f d2 e1 28 a5 7c 50 30 f1 84 44 08 ab ae a5 b8 fc ba 80 33 0c e7 00
             20 76 97 03 b5 f3 d0 a2 fe 9a 3d bf 23 42 a6 43 e4 32 46 e8 0c 3c 04
             d0 19 78 45 ce 0b 16 f9 76 23 1d 99 f0 66
```

H.6.5 PRF test vectors

A set of test vectors are provided for each size of PRF function used in this subclause. See Table H.9 through Table H.12. The inputs to the PRF function are strings for key, prefix, and data. The length can be any multiple of 8, but the values 192, 256, 384, and 512 are used in this subclause. The test vectors were taken from IETF RFC 2202 with additional vectors added to test larger key and data sizes.

Table H.9—RSN PRF Test Vector 1

Test_case	1
Key	0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b
Prefix	"prefix"
Data	"Hi There"
Length	192
PRF-192	bc d4 c6 50 b3 0b 96 84 95 18 29 e0 d7 5f 9d 54 b8 62 17 5e d9 f0 06 06

Table H.10—RSN PRF Test Vector 2

Test_case	2
Key	'Jefe'
Prefix	"prefix-2"
Data	"what do ya want for nothing?"
Length	256
PRF-256	47 c4 90 8e 30 c9 47 52 1a d2 0b e9 05 34 50 ec be a2 3d 3a a6 04 b7 73 26 d8 b3 82 5f f7 47 5c

Table H.11—RSN PRF Test Vector 3

Test_case	3
Key	aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
Prefix	"prefix-3"
Data	"Test Using Larger Than Block-Size Key - Hash Key First"
Length	384
PRF-384	0a b6 c3 3c cf 70 d0 d7 36 f4 b0 4c 8a 73 73 25 55 11 ab c5 07 37 13 16 3b d0 b8 c9 ee b7 e1 95 6f a0 66 82 0a 73 dd ee 3f 6d 3b d4 07 e0 68 2a

Table H.12—RSN PRF Test Vector 4

Test_case	4
Key	0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b
Prefix	"prefix-4"
Data	"Hi There Again"
Length	512
PRF-512	24 8c fb c5 32 ab 38 ff a4 83 c8 a2 e4 0b f1 70 eb 54 2a 2e 09 16 d7 bf 6d 97 da 2c 4c 5c a8 77 73 6c 53 a6 5b 03 fa 4b 37 45 ce 76 13 f6 ad 68 e0 e4 a7 98 b7 cf 69 1c 96 17 6f d6 34 a5 9a 49

H.7 Key hierarchy test vectors

The test vectors in H.7.1 provide an example of PTK derivation for both CCMP and TKIP.

H.7.1 Pairwise key derivation

Pairwise keys are derived from the PMK, AA, SPA, SNonce, and ANonce. The values in Table H.13 are used as input to the pairwise key derivation test vectors.

Table H.13—Sample values for pairwise key derivations

PMK	0d c0 d6 eb 90 55 5e d6 41 97 56 b9 a1 5e c3 e3 20 9b 63 df 70 7d d5 08 d1 45 81 f8 98 27 21 af
AA	a0 a1 a1 a3 a4 a5

Table H.13—Sample values for pairwise key derivations *(continued)*

SPA	b0 b1 b2 b3 b4 b5
SNonce	c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 d0 d1 d2 d3 d4 d5 d6 d7 d8 d9
ANonce	e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9

H.7.1.1 CCMP pairwise key derivation

Using the values from Table H.13 for PMK, AA, SPA, SNonce, and ANonce, the key derivation process for CCMP generates a temporal key as shown in Table H.14.

Table H.14—Sample derived TKIP temporal key (TK)

TK	8c b7 78 33 2e 94 ac a6 d3 0b 89 cb e8 2a 9c a9

H.7.1.2 TKIP pairwise key derivation

Using the values from Table H.13 for PMK, AA, SPA, SNonce, and ANonce, the key derivation process for TKIP generates the values shown in Table H.15.

Table H.15—Sample derived PTK

KCK	Aa 7c fc 85 60 25 1e 4b c6 87 e0 cb 8d 29 83 63
KEK	Ba 53 16 3d f3 2a 86 38 f4 79 ab e3 4b fd 2b c8
TK	8c b7 78 33 2e 94 ac a6 d3 0b 89 cb e8 2a 9c a9 36 4a ff bb ce 87 5f 5d f2 dd 58 41 c0 ed 2a 41
Authenticator Tx MIC_key	36 4a ff bb ce 87 5f 5d
Supplicant Tx MIC_key	f2 dd 58 41 c0 ed 2a 41