IEEE STANDARDS ASSOCIATION ◈IEEE

IEEE Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP)

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

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

IEEE Std 1735™-2014
(Incorporates
IEEE Std 1735-2014/Cor 1-2015)

IEEE Std 1735™-2014
(Incorporates
IEEE Std 1735-2014/Cor 1-2015)

IEEE Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP)

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 10 December 2014

IEEE SA-Standards Board

Acknowledgments

Grateful acknowledgment is made to the following for the permission to use the following source material:

Accellera/VSIA—*IP_Encrypt_VSIAtoIEEE.doc*: VSIA contribution from the IP-Encrypt Working Group.

Cadence—Material in clause titled "Rights Management" is derived from the document titled "Rights Management Specification for Verilog® Protected Envelopes" © 2007, 2011, Cadence Design Systems Inc. Used, modified, and reprinted by permission of Cadence Design Systems Inc.

Synopsys—*IP_Licensing_Recommendations_for_P1735.pptx*: Synopsys IP licensing overview.

Abstract: Guidance on technical protection measures to those who produce, use, process, or standardize the specifications of electronic design intellectual property (IP) are provided in this recommended practice. Distribution of IP creates a risk of unsanctioned use and dilution of the investment in its creation. The measures presented here include protection through encryption, specification, and management of use rights that have been granted by the producers of electronic designs, and methods for integrating license verification for granted rights.

Keywords: digital envelope, encrypted IP, IEEE 1735™, keys, rights management, trust model

---

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

Copyright © 2015 by The Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 23 September 2015. Printed in the United States of America.

IEEE, POSIX, and 802 are registered trademarks in the U.S. Patent & Trademark Office, owned by The Institute of Electrical and Electronics Engineers, Incorporated.

Verilog is a registered trademark of Cadence Design Systems, Inc.

Print: ISBN 978-0-7381-9492-9 STD20084
PDF: ISBN 978-0-7381-9493-6 STDPD20084

IEEE prohibits discrimination, harassment, and bullying.
For more information, visit http://www.ieee.org/web/aboutus/whatis/policies/p9-26.html.
No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

Important Notices and Disclaimers Concerning IEEE Standards Documents

IEEE documents are made available for use subject to important notices and legal disclaimers. These notices and disclaimers, or a reference to this page, appear in all standards and may be found under the heading "Important Notice" or "Important Notices and Disclaimers Concerning IEEE Standards Documents."

Notice and Disclaimer of Liability Concerning the Use of IEEE Standards Documents

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

IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims all warranties (express, implied and statutory) not included in this or any other document relating to the standard, including, but not limited to, the warranties of: merchantability; fitness for a particular purpose; non-infringement; and quality, accuracy, effectiveness, currency, or completeness of material. In addition, IEEE disclaims any and all conditions relating to: results; and workmanlike effort. IEEE standards documents are supplied "AS IS" and "WITH ALL FAULTS."

Use of an IEEE standard is wholly voluntary. The existence of an IEEE standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document, should rely upon his or her own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

IN NO EVENT SHALL IEEE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO: PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE PUBLICATION, USE OF, OR RELIANCE UPON ANY STANDARD, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND REGARDLESS OF WHETHER SUCH DAMAGE WAS FORESEEABLE.

Copyright © 2015 IEEE. All rights reserved.

Translations

The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE should be considered the approved IEEE standard.

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, or be relied upon as, a formal position of IEEE. 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 of IEEE.

Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE. However, IEEE does not provide consulting information or advice pertaining to IEEE Standards documents. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive 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 comments or questions except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in revisions to an IEEE standard is welcome to join the relevant IEEE working group.

Comments on standards should be submitted to the following address:

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

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

IEEE draft and approved standards are copyrighted by IEEE under U.S. and international copyright laws. They are made available by IEEE and are adopted for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making these documents available for use and adoption by public authorities and private users, IEEE does not waive any rights in copyright to the documents.

Copyright © 2015 IEEE. All rights reserved.

Photocopies

Subject to payment of the appropriate fee, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, non-commercial use only. To arrange for payment of licensing fees, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Updating of IEEE Standards documents

Users of IEEE Standards documents should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect.

Every IEEE standard is subjected to review at least every ten years. When a document is more than ten years old and has not undergone a revision process, 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 order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE-SA Website at http://ieeexplore.ieee.org/expel/standards.jsp or contact IEEE at the address listed previously. For more information about the IEEE-SA or IOWA's standards development process, visit the IEEE-SA Website at http://standards.ieee.org.

Errata

Errata, if any, for all IEEE standards can be accessed on the IEEE-SA Website at the following URL: http://standards.ieee.org/findstds/errata/index.html. Users are encouraged to check this URL for errata periodically.

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 by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE-SA Website at http://standards.ieee.org/about/sasb/patcom/patents.html. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Copyright © 2015 IEEE. All rights reserved.

Participants

IEEE Std 1735-2014

The Electronic Design Intellectual Property (IP) Working Group is entity based. At the time this standard was completed, the Electronic Design Intellectual Property (IP) Working Group had the following membership:

Dave Graubart, *Chair*
Joe Daniels, *Technical Editor*

Luis Humberto Rezende Barbosa
Dave Clemans
Steven Dovich
Jeff Fox
Parminder Gill
Satyam Jani
Jarek Kaczynski
Ray Martin
Gael Paul
Rod Price
Adam Sherer
John Shields
Michael Smith
Sourabh Tandon
Ruchi Tyagi

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

Accellera Organization, Inc.
ALDEC, Inc.
Altera Corporation
Atrenta Inc.
Cadence Design Systems, Inc.
Mentor Graphics
Synopsys, Inc.
Xilinx Inc.

When the IEEE-SA Standards Board approved this standard on 10 December 2014, it had the following membership:

John Kulick, *Chair*
Jon Walter Rosdahl, *Vice Chair*
Richard H. Hulett, *Past Chair*
Konstantinos Karachalios, *Secretary*

Peter Balma
Farooq Bari
Ted Burse
Clint Chaplain
Stephen Dukes
Jean-Phillippe Faure
Gary Hoffman
Michael Janezic
Jeffrey Katz
Joseph L. Koepfinger*
David J. Law
Hung Ling
Oleg Logvinov
T. W. Olsen
Glenn Parsons
Ron Peterson
Adrian Stephens
Peter Sutherland
Yatin Trivedi
Phil Winston
Don Wright
Yu Yuan

*Member Emeritus

Copyright © 2015 IEEE. All rights reserved.

At the time this standard was completed, the Electronic Design Intellectual Property (IP) Working Group had the following membership:

Dave Graubart, *Chair*
Joe Daniels, *Technical Editor*

Luis Humberto Rezende Barbosa
Dave Clemans
Steven Dovich
Jeff Fox
Parminder Gill
Satyam Jani
Jarek Kaczynski
Ray Martin
Gael Paul
Rod Price
Adam Sherer
John Shields
Michael Smith
Sourabh Tandon
Ruchi Tyagi

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

Accellera Organization, Inc.
ALDEC, Inc.
Altera Corporation
Atrenta Inc.
Cadence Design Systems, Inc.
Marvell Semiconductor, Inc.
Mentor Graphics
Micron Technology, Inc.
Synopsys, Inc.
Xilinx Inc.

When the IEEE-SA Standards Board approved this standard on 3 September 2015, it had the following membership:

John D. Kulick, *Chair*
Jon Walter Rosdahl, *Vice Chair*
Richard H. Hulett, *Past Chair*
Konstantinos Karachalios, *Secretary*

Masayuki Ariyoshi
Ted Burse
Stephen Dukes
Jean-Philippe Faure
J. Travis Griffith
Gary Hoffman
Michael Janezic
Joseph L. Koepfinger*
David J. Law
Hung Ling
Andrew Myles
T. W. Olsen
Glenn Parsons
Ronald C. Petersen
Annette D. Reilly
Stephen J. Shellhammer
Adrian P. Stephens
Yatin Trivedi
Phillip Winston
Don Wright
Yu Yuan
Daidi Zhong

*Member Emeritus

Copyright © 2015 IEEE. All rights reserved.

Introduction

This introduction is not part of IEEE Std 1735™-2014, IEEE Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP).

The purpose of this recommended practice is to provide guidance on protection of *electronic design intellectual property* (IP). The audience for this standard includes IP producers, IP consumers, vendors of tools that process protected IP, and standards development groups for IP specification formats.

When the electronic design automation (EDA) industry began creating standards for use in specifying, simulating, and implementing electronic circuits, there was no active market for the exchange of electronic designs. As interest in software reuse developed, the EDA industry began to share design collateral as a means of controlling the cost of development and managing the timeline for completing their design projects. Because that shared IP had a measurable development cost, business leaders began to insist on recovering those costs through licensing to other potential uses. Technical measures for IP protection were developed to augment the legal contracts that governed such shared use.

As the means for technical protection of IP proliferated, there was increased pressure to incorporate that technology in the existing standards for representation of IP. Expertise in protection technologies (such as encryption) was scarce in those standards development organizations (SDOs), which resulted in a slow pace of development within a given standard and technical divergence as independent organizations attempted to solve similar problems. As the marketplace for IP exchange continued to mature, concerns about technical protection of IP extended to include an interest in defining and managing the scope of use for protected IP.

This standard has been created to consolidate EDA industry efforts to unify the technical protections for IP and to guide SDOs in aligning their work for interoperability and compatibility. It is also intended to share best practices in IP protection for those who use standards that incorporate such technology, such as VHDL (IEC 61691-1-1, IEEE Std 1076™) and SystemVerilog (IEEE Std 1800™).[a]

Corrections were made to 7.4.3 as required by IEEE Std 1735-2014/Cor 1-2015.

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

Copyright © 2015 IEEE. All rights reserved.

Contents

1. Overview 1

1.1 Scope 1
1.2 Purpose, value, and approach 2
1.3 Key characteristics of this standard 2
1.4 Conventions used in this standard 3
1.5 Use of color in this standard 3
1.6 Contents of this standard 4

2. Normative references 5

3. Definitions, acronyms, and abbreviations 5

3.1 Definitions 5
3.2 Acronyms and abbreviations 7

4. Trust model 8

4.1 Stakeholders 8
4.2 Role of IP protection 8
4.3 Protection via encryption 9
4.4 Components of trust 10

5. Interoperability 12

5.1 Background 12
5.2 version pragma 12
5.3 Basic interoperability 13
5.4 Use cases 16
5.5 Secure keyring 19

6. Key management 21

6.1 Overview 21
6.2 Key considerations 21
6.3 Basic key exchange 22
6.4 Standard key exchange 23
6.5 Key scope recommendations 25

7. Rights management [V2] 27

7.1 Introduction 27
7.2 Rights scope 27
7.3 Tool types and rights 27
7.4 Syntax and markup 28
7.5 Tamper-proof requirements 34
7.6 Complete tool block and rights example 34

Copyright © 2015 IEEE. All rights reserved.

8. License management [V2] 36

8.1 Introduction 36
8.2 License system 36
8.3 License proxy 36
8.4 License specification 36
8.5 License proxy parameters 37
8.6 License use 38
8.7 Multiple envelopes 39
8.8 Proxy communication 40
8.9 License proxy transactions 40
8.10 License proxy commands 44
8.11 Deprecated licensing pragmas 45

9. Visibility management 46

9.1 Introduction 46
9.2 Background [V1] 46
9.3 Visibility in tool phases [V1] 47
9.4 Visibility and encryption envelopes [V1] 50
9.5 **viewport** pragmas 52
9.6 Programming language interfaces 58
9.7 Controlling visibility with rights [V2] 60
9.8 Visibility of dynamic objects [V2] 61
9.9 Unresolved visibility issues 61

10. Common rights [V2] 62

10.1 Defining common rights 62
10.2 Overriding common rights 62
10.3 Defaults and the delegated value 62
10.4 Common conditions for rights 63
10.5 Common right for error handling 64
10.6 Common right for visibility 65
10.7 Common right for child visibility 66
10.8 Common right for decryption 67

Annex A (informative) Bibliography 68

Annex B (informative) Other known issues with IP protection 69

Annex C (normative) Protection pragmas 75

Copyright © 2015 IEEE. All rights reserved.

IEEE Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP)

IMPORTANT NOTICE: IEEE Standards documents are not intended to ensure safety, health, or environmental protection, or ensure against interference with or from other devices or networks. Implementers of IEEE Standards documents are responsible for determining and complying with all appropriate safety, security, environmental, health, and interference protection practices and all applicable laws and regulations.

This IEEE document is made available for use subject to important notices and legal disclaimers. These notices and disclaimers appear in all publications containing this document and may be found under the heading "Important Notice" or "Important Notices and Disclaimers Concerning IEEE Documents." They can also be obtained on request from IEEE or viewed at http://standards.ieee.org/IPR/disclaimers.html.

1. Overview

1.1 Scope

This standard specifies embeddable and encapsulating markup syntaxes for design intellectual property encryption and rights management, together with recommendations for integration with design specification formats described in other standards. It also recommends use models for interoperable tool and hardware flows, which will include selecting encryption and encoding algorithms and encryption key management. The recommendation includes a description of the trust model assumed in the recommended use models. This standard does not specifically include any consideration of digitally encoded entertainment media. In the context of this document, the term *IP* will be used to mean *electronic design intellectual property*.

Electronic design intellectual property is a term used in the electronic design community. It refers to a reusable collection of design specifications that represent the behavior, properties, and/or representation of the design in various media. Examples of these collections include, but are not limited to, the following:

- A unit of electronic system design
- A design verification and analysis scheme (e.g., test bench)
- A netlist indicating elements and the interconnection thereof to implement a function
- A set of fabrication instructions
- A physical layout design or chip layout
- A design intent specification

Copyright © 2015 IEEE. All rights reserved.

The term is partially derived from the common practice for the collection to be considered the intellectual property of one party. Hardware and software descriptions are encompassed by this term.

1.2 Purpose, value, and approach

The intent of this document is to enable design flows that provide interoperability among IP authors, tool providers, integrators, and users of the IP. The resulting flow identified aids IP authors in providing IP that can be processed by tools without sharing protected information with IP users. Furthermore, this flow can support an integrated licensing scheme, enabling the IP authors to specify compile-time licenses. An integrated rights management scheme is also an element of the flow, which helps IP authors to control tool behavior including, but not limited to, IP visibility, allowed tool versions, and output file encryption.

There is currently no defined, independent standard for describing IP encryption markup for design information formats. Each design format that incorporates IP encryption describes their markup differently, leading to confusing interpretation. Users of those standards also lack a recommended practice for interoperable use of IP encryption.

For the IP author, the digital envelope containing the rights, key, and data blocks is the main focus. This envelope contains the source, encrypted with a symmetric cipher using a one-time session key. The session key is encrypted with a tool vendor public key for each supported vendor arranged in a series of key blocks. The license constraints and rights constraints are similarly encrypted with a tool vendor public key for each supported vendor arranged in a series of rights blocks.

The tool vendor supporting the standard can then parse and process the envelope and its contents. When the tool vendor finds the blocks with their public keys, each block is decrypted with their private key, thus extracting the data key, licensing requirements, and rights constraints. If required, the tool then checks for the presence of a valid license and, if successful, decrypts the source using the data key and obeying the rights constraints. Output files from the tool may be encrypted for use in downstream tools, provided such action is granted in the IP rights.

A standard defined with all these IP author and tool vendor features would make the overall flow transparent for the IP user. Therefore, this document provides guidelines and recommended practice for use of IP protection markup syntax and key management to enable interoperable tool flows with IP and tools from a wide array of suppliers. It includes algorithm selection for encryption and encoding.

This document also specifies a subset of markup syntax for hardware description language (HDL) formats that could adapted to other file formats. These files represent potential inputs and outputs of tools that would otherwise expose IP. The generic syntax of these directives may be suitably modified for a particular file format if there are syntactic conflicts and variations that may be described in recommended practices.

1.3 Key characteristics of this standard

The marketplace for electronic design data depends on trust between the parties to the transactions. Exchanges require a confidence between the parties that they represent legitimate interests, that the product of exchange is accurately described, and that the terms of the transaction will be fairly observed.

As a form of *intellectual property*, electronic design data is a virtual object and is susceptible to the evolving social conventions of respect for ownership rights. Despite uneven acceptance of their rights to control the use of IP, those who produce such property frequently want to monetize their investment and receive compensation for their investment in its creation. Risk premiums to compensate for loss are not commonly itemized, but can be assumed as a factor in compensation demands. The inverse of a risk premium is the trust discount, which is most commonly reflected in a greater willingness to market the IP as the strength of the trust model improves. This concept of the trust model is explained in more detail in Clause 4.

Copyright © 2015 IEEE. All rights reserved.

Trade secret protection is usually impractical as the security and reliability of the protection decreases with each new transaction. It also requires a degree of confidence in the integrity of the receiving party that is an unwise assumption if the objective is to preserve the financial value of the IP.

Since party integrity is a concern, the other forms of legal protection need to be accompanied by technical security to help enforce the terms of the transaction. The recommended practices described by this standard assist in the construction and maintenance of technical security consistent with the intent of the contracted terms of the transaction.

1.4 Conventions used in this standard

Each command description (syntax) consists of a command keyword followed by one or more parameters. The meta-syntax for the description of the syntax rules uses the conventions shown in Table 1.

Table 1—Document conventions

Visual cue	Represents
`courier`	The courier font indicates source code from one of the relevant HDL languages, e.g., `module test(a,b,c,d);`
bold	The bold font is used to indicate key terms, text that shall be typed exactly as it appears. For example, in the following markup directive, the keywords "`pragma" shall be typed as it appears: **`pragma** *pragma_keyword*
italic	The italic font represents variables. For example, an encryption algorithm identifier needs to be specified in the following line (after the "key_method" key term): **key_method** *algorithm_identifier*
::=	A production consists of a left-hand side, the symbol "::=" (which is read as "can be replaced by"), and a right-hand side. The left-hand side is always a syntactic category; the right-hand side is a replacement rule. The meaning of a production is a textual-replacement rule: any occurrence of the left-hand side may be replaced by an instance of the right-hand side.
[ ] square brackets	Square brackets indicate optional items on the right-hand side of a production. For example, the following productions are equivalent: author_spec ::= author_directive [author_info_directive] author_spec ::= author_directive | author_directive author_info_directive
| separator bar	The separator bar (|) character separates alternative choices unless it occurs immediately after an opening brace, in which case it stands for itself. For example, the following line shows the "delegated" or "true" or "false" key terms are possible values for the "Boolean_rights_value" production: **boolean_rights_value** :== **delegated** | **true** | **false**

1.5 Use of color in this standard

This standard uses a minimal amount of color to enhance readability. The coloring is not essential and does not affect the accuracy of this standard when viewed in pure black and white. The places where color is used are the following:

— Cross references that are hyperlinked to other portions of this standard are shown in underlined-blue text (hyperlinking works when this standard is viewed interactively as a PDF file).

Copyright © 2015 IEEE. All rights reserved.

— Syntactic keywords and tokens in the formal language definitions are shown in **boldface**.

1.6 Contents of this standard

The organization of the remainder of this standard is as follows:

— Clause 2 provides references to other applicable standards that are presumed or required for this standard.

— Clause 3 defines terms, acronyms, and abbreviations used throughout the different specifications contained in this standard.

— Clause 4 describes a trust model for protecting and sharing IP.

— The next several clauses provide the technical background and make general recommendations for encrypting (and sharing) IP in specific areas: **interoperability**, **key management**, **rights management** (and rights control structures), **license management**, **visibility management**, and **common rights**.

— Annexes. Following Clause 10 are a series of informative and normative annexes.

NOTE—Clause 4 through Clause 10 define the technical subparts needed to ensure the trust model and share IP.[1]

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

Copyright © 2015 IEEE. All rights reserved.

2. Normative references

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEC 61691-1-1, IEEE Std 1076™, Behavioural languages—Part 1-1: VHDL language reference manual (Adoption of IEEE Std 1076-2002).[2, 3]

IEEE Std 802®, IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture.[4]

IEEE Std 1003.1™, IEEE Standard for Information Technology—Portable Operating System Interface (POSIX®).

IEEE Std 1800™, IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language.

IETF RFC 3447, Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1.[5]

IETF RFC 5280, Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.

ISO/IEC 9594-8, Information technology—Open Systems Interconnection—The Directory—Part 8: Public-key and attribute certificate frameworks.[6]

NIST Special Publication 800-131A. Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths.[7]

3. Definitions, acronyms, and abbreviations

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary Online* should be consulted for terms not defined in this clause.[8]

3.1 Definitions

asymmetric encryption: An encryption/decryption scheme where the key to decrypt the data (private key) is different from the key to encrypt the data (public key).

decryption envelope: A region of electronic design intellectual property (IP) with markup instructions to direct the process of decryption.

---

[2]IEC publications are available from the International Electrotechnical Commission (http://www.iec.ch/). IEC publications are also available in the United States from the American National Standards Institute (http://www.ansi.org/).

[3]IEEE publications are available from The Institute of Electrical and Electronics Engineers (http://standards.ieee.org/).

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

[5]IETF documents (i.e., RFCs) are available for download at http://www.rfc-archive.org/.

[6]ISO/IEC publications are available from the ISO Central Secretariat (http://www.iso.org/). ISO publications are also available in the United States from the American National Standards Institute (http://www.ansi.org/).

[7]NIST publications are available from the National Institute of Standards and Technology (http://www.nist.gov/)

[8]*IEEE Standards Dictionary Online* subscription is available at:
http://www.ieee.org/portal/innovate/products/standard/standards_dictionary.html.

Copyright © 2015 IEEE. All rights reserved.

ciphertext: Encrypted data. The data may be encrypted by either a symmetric cipher or an asymmetric cipher.

common rights: An optional section describing rights common to all tools.

data block: The encrypted electronic design intellectual property (IP).

DER format: Distinguished Encoding Rules (X.690, ASN.1). A method of canonically representing certain data structures, here used to encode RSA Public Keys.

digest: A scheme where a validation string for a message is generated. This is used to check the integrity of the message and can be used to authenticate it—to ensure the message came from the claimed source.

digital envelope: A bounded data collection that uses encryption technology for protection and/or identification. *Digital envelope* is the name for a primary IEEE 1735 use case, in which there is just such a digital envelope constructed for each tool. That envelope is the **key block,** and it "contains" a **session key** encrypted with a public key.

NOTE—See 5.4.1.

encryption envelope: A region of electronic design intellectual property (IP) with markup instructions to direct the process of encryption.

Hash-based Message Authentication Code (HMAC): A scheme that uses a message digest algorithm and a secret key to generate a string for a message that can be used to authenticate the message. (Adapted from RFC 2104)

intellectual property: A work or invention that is the result of creativity, which may be covered by legal property rights such as patents, trademarks, copyrights, etc.

key block: The **session key** encrypted with a particular tool vendor's public key and encoded with Base64 Encoding.

key owner/key name: This pair uniquely describes the public/private key pair used to encrypt the **session key**.

license proxy: A program provided by the electronic design intellectual property (IP) author to control access by the tools to the IP. A **license proxy** may be an actual license server or a interface to a third-party license server.

plaintext: Unencrypted data.

proxy session key: A random key used to encrypt data for communication between the **license proxy** and the tool.

pseudo-random number: A number whose bits may appear random but that may be correlated with other bits or past values.

random number: A completely random number, meaning it is unpredictable with no bit sharing any relationship with any other bit, each bit equally likely to be a `1` or a `0`, and the values of any bit is completely unrelated to any previous random numbers.

rights digest: A Hash-based Message Authentication Code (HMAC) generated from the **session key**, **key owner**, **key name**, (optional) **common rights**, and (optional) **tool-specific rights**.

Copyright © 2015 IEEE. All rights reserved.

session key: A random key used to encrypt data in an encryption block.

symmetric encryption: An encryption/decryption scheme where the key to encrypt the data is the same as the key to decrypt the data.

tool block: The section containing the **key block** and optional **tool-specific rights**.

tool-specific rights: An optional section describing rights specific to one tool.

trust model: The defined trust relationships and orientation of the communities participating with the organization in the continued operation of the business. This term also describes the common elements that need to be in place for a clear understanding of trust in systems related to information sharing, authentication, reliability, and many other endeavors. *Trust models* are often considered in information security contexts when there is a need to decide whether a specific security methodology (e.g., digital certificates) is appropriate for achieving a specific security objective (e.g., user authentication).

3.2 Acronyms and abbreviations

API application programmers interface

CA certificate authority

CLI command line interface

DER Distinguished Encoding Rules

DPI direct programming interface

EDA electronic design automation

FIPS Federal Information Processing Standards

HDL hardware description language

HMAC Hash-based Message Authentication Code

IP electronic design intellectual property

IV initialization vector

LRM language reference manual

NIST National Institute of Standards and Technology

PLI programming language interface

SDO standards development organization

VHDL VHSIC hardware description language

VHPI VHDL programming interface

VHSIC Very High Speed Integrated Circuit

VPI SystemVerilog programming interface

Copyright © 2015 IEEE. All rights reserved.

4. Trust model

It is necessary for the stakeholders in IP protection to trust each other in order to do business. If one is to understand how encryption plays a role and therefore to trust it sufficiently, one needs to have a concept of the trust model it supports. In general, understanding the trust model helps one decide that a given security mechanism is appropriate to achieve the security objective.

This clause identifies the stakeholders and their interests. It explains the role of IP protection, how it can be achieved via encryption, and why it is valuable. Finally, the components of the trust model are examined to understand the foundation they provide for trusting this form of IP protection.

4.1 Stakeholders

In any transaction there are parties of interest, those who hold a stake in the outcome of the transaction. Within this trust model, there are three primary stakeholders who represent the spectrum of concerns: the *IP author*, the *IP user*, and the *tool vendor*.

The *IP author* stakeholder is the legal owner of the IP and is the grantor of any rights that are conveyed in the course of transactions involving that IP. The nature of the rights offered and the compensation demanded for those rights are specified by the IP author. Their interest includes providing a valuable product and preventing loss through disclosure of their IP to either competition or IP users. Their ultimate interest is maximizing their return on investment in the IP.

The *IP user* stakeholder is the receiving party who seeks possession and rights to the IP to further their own interests. After providing satisfactory compensation to the IP author, the IP user receives the transacted rights to use the IP. Their interest includes choosing the right IP of the appropriate usefulness and quality and minimizing the risk and cost of mistakes in doing so. Their ultimate interest is in minimizing the total cost to use the right IP in their designs.

The *tool vendor* stakeholder provides a tool to the IP user that is enabled to work with the IP. Indeed, they provide a tool to the IP author as well. Here, the tool provides an IP protection mechanism based on encryption that can be used by the IP author to deliver protected IP to the IP user. This mechanism allows the IP user to work with the tool for its intended purpose with that protected IP, while providing a high barrier to disclosure of the IP. The tool vendor interests are similar to those described for the IP author; they include providing a valuable product, their tool, and preventing loss through disclosure of tool features and internal behavior to their competition. Their ultimate interest is maximizing their return on investment in the tool, which includes the investment in the IP protection mechanism.

4.2 Role of IP protection

The pricing of an IP is based on a complex set of factors. Accounting for the risk of loss due to disclosure is one of them. Though not commonly itemized, it is fair to assume this adds a premium to the compensation demands of an IP author. It is referred to here as the *risk premium*.

The goal of an IP protection mechanism is to mitigate the risk of loss due to disclosure. IP protection through encryption is one mechanism, but there are many others, including altering the representation of the IP through obfuscation or abstraction, limiting access to secure, fixed, “brick and mortar” locations, and a variety of legal mechanisms. None of these are perfect. An IP protection mechanism provides a degree of trust that a loss does not happen. An IP protection mechanism itself may have considerable cost to employ, which is a part of the risk premium.

Copyright © 2015 IEEE. All rights reserved.

When IP authors are uncertain about how to price the risk premium, they are usually more comfortable discounting “the loss aspect” when they trust that the IP protection mechanisms bring the risk down to an acceptable level. This may be reflected in lower prices or a greater willingness to engage in commerce with a wider audience of potential IP users. It is referred to here as the *trust discount*.

There is usually nothing specific an IP user can do to substantially improve the trust discount for a right to access the IP. Establishing a long-term trusted relationship with an IP author should have a positive effect, of course. While loss due to disclosure is an IP author interest, it is not a shared interest of the IP user. It is fair to say this is instead a conflict of interest.

The tool vendor can provide an IP protection mechanism that, if done well, should provide a trust discount for its use. IP protection via encryption is just such a mechanism, which is valuable to IP authors and IP users, both of whom seek a trust discount for their own self-interest. The tool vendor is an objective party that does not have an interest in the IP itself. Acting as a proxy to protect the IP and make it useful to the IP user presents no conflict of interest for the tool vendor.

Subclause 4.3 gives an overview of IP protection via encryption, described at a level to support further discussion of its trust model.

4.3 Protection via encryption

Encryption is a lossless transformation, not obfuscation or abstraction. The protected IP has 100% fidelity to its original representation. This IP can also be delivered to the IP user premises. In its encrypted form, and in the absence of the decryption key, the data is secure both in transmission and at rest in a file. This encrypted data is based on cryptographically strong encryption algorithms. There are no independent means to decrypt and access it at the IP user premises.

The IP protection mechanism is designed to allow IP use only through vendor tools designated by the IP author. A tool conforms to the expectations of the applicable standards with respect to rights granted by the IP author and disclosure of information during its execution. There is no intentional disclosure of protected information in an IP, except as permitted. The original description of the IP is processed to produce its encrypted form. This single encrypted form can be delivered and used by all authorized tools, i.e., the IP protection mechanism is interoperable.

This standard builds on previous work to define a common scheme of markup for encryption and decryption of IP. Following are the high level ideas for how it works.

a) The IP author marks up the IP lightly to describe the encryption scheme, what aspects are to be protected, what remains visible, what rights the IP user has, and so on.

b) In terms of that encryption scheme, the primary use case supported by this standard is the *Digital Envelope* model.
   1) In this model, the IP author chooses a set of vendor tools to enable access to their IP.
   2) Each tool has its own public/private key pair that is used to encrypt the common session key used to protect the IP.

c) In the resulting protected IP, there is a key block for each vendor tool that contains the encrypted, common session key.
   1) That key block can only be decrypted by the specific tool vendor whose key was used to protect it.
   2) With that session key, the tool can decrypt the protected IP.

See Clause 5 for a more complete understanding of interoperability and use cases supported by this IP protection mechanism.

Copyright © 2015 IEEE. All rights reserved.

4.4 Components of trust

It is good to know what needs to be trusted. For that, it is important to take a systematic look at the components of the trust model for this IP protection mechanism. This subclause provides a high level identification of those components.

4.4.1 Encryption strength

Guarding the IP against unauthorized disclosure requires use of a high quality encryption algorithm that uses a sufficiently long key. Secure encryption relies on the computational difficulty of reconstructing the key, even when the details of how the algorithm applies the key are known. Longer keys make that reconstruction difficult and eventually impractical. There comes a point where the weakness in the overall cryptosystem moves away from the choice of encryption algorithm and key, towards other aspects.

This applies to the protected IP when it is "at rest" in a file and when it is "in transit" over a network as part of delivery to IP users. See 5.3 for details on the required algorithms.

4.4.2 Digital identity verification

In the primary digital envelope use case, the IP author designates a particular tool as authorized to decrypt their IP by using the public key for that tool. In general, trust relies on the ability to establish that such keys, and any other digital identity tokens, that are offered correspond to the implied stakeholder. There are many strategies for introducing fraudulent identities, and there are corresponding defenses against this risk. This standard accounts for this trust model component in its key management recommendations (see Clause 6).

4.4.3 Tool vendor integrity

The structure of this IP protection mechanism requires the IP author to delegate access to their IP to the IP user by proxy. The tool is enabled to decrypt the IP and entrusted with enforcing the rights to use it during execution. The IP author is presumed to trust the integrity of the tool vendor, i.e., to trust that the tool vendor will not intentionally disclose the author's IP.

Unauthorized disclosure by the tool vendor is more difficult to guard against with technical measures. The principal tool for managing this risk is the integrity and reputation of the tool vendor. Functional restrictions on disclosure are expected before IP authors entrust their IP to a particular tool. There is also an opportunity for the IP author to consider: An agreement with the tool vendor for restricted access to decrypted source code can aid both parties with IP user support issues.

4.4.4 Security of design flow

The weakest link in a chain is where it is most vulnerable. It is important to consider all the tools in the design flow that the IP author enables to use their IP. It is expected that each tool, taken individually, implements the IP protection mechanism correctly with best practices for security and that tool vendor integrity is not in question. Care shall also be taken with how a particular tool fits together with others in the flow. If the interface between tools not only uses the protected IP representation, but also depends on other data flow, no protected information should be exposed in that other data flow either.

Other well-known data formats that are unprotected, for example, Standard Delay Format (SDF) and Unified Power Format (UPF), present a challenge in this respect. It does not matter whether it is a standard or proprietary data representation, allowing interoperability with exposed data is a gap that risks disclosure. Tools that are designed to have a common architecture for data protection are stronger in this respect than those that have no framework for collaborative protection over the whole flow.

Copyright © 2015 IEEE. All rights reserved.

The choices of what tools to use in a design flow are commonly based on factors like functionality, performance, cost, and quality. IP users want the best tools they can afford for their purposes. The IP author has additional considerations to apply to the choices of what tools to enable to work with their protected IP.

4.4.5 Tool quality

In this context, the quality of the IP protection mechanism is both with respect to correctness and secure coding practices. If a tool discloses protected information during its execution because of a tool defect or incorrect interpretation of standards, that is a correctness issue. If a disclosure happens because a tool is hacked during execution by an untrusted IP user, using any combination of binary tampering and debugging techniques, that is an implementation issue. Each tool vendor needs to decide what, if any, anti-debug, anti-tamper, and anti-key-discovery technologies they will use here.

Tool vendors instill confidence in their tools through quality of implementation. The effort invested in robust and secure programming practices results in fewer IP user disclosure events. There is no absolute protection, and constant reinvestment is required to stay ahead of evolving public expertise in hacking methodologies.

Understanding that there is no absolute protection, the following are some relevant considerations for agreements between stakeholders.

- Between tool vendors and their users (which are IP users and IP authors), an agreement for use of the tool should forbid tampering and reverse engineering without being granted explicit permission. With such permission, the results of any reverse engineering should also be disclosed to the tool vendor.
- Between an IP author and IP user, an agreement for use of protected IP should forbid an attempt to tamper or reverse engineer any protection mechanisms in any tool in order to obtain access to IP source code or encryption keys.

4.4.6 Summary

The components of the trust model give stakeholders an ability to consider the risks and rewards of this IP protection mechanism. All the stakeholders, acting in their own self-interests, should find considerable value in it and recognize its strengths in safeguarding IP and its potential for reducing the costs of doing business.

Copyright © 2015 IEEE. All rights reserved.

5. Interoperability

This recommended practice uses the terms *required* and *requirement* to indicate content that is necessary to be compliant with IEEE Std 1735, including material that is an extension of current usage in IEEE Std 1800 (SystemVerilog) or IEC 61691-1-1, IEEE Std 1076 (VHDL).[9] This standard also uses the terms *recommended* and *recommendation* for best practices and use cases.

The recommendations in this clause are intended to be incorporated by reference in other standards, e.g., VHDL, where they shall be read as mandatory requirements on tools conforming to those standards. The intended audience for these recommendations are any working groups considering incorporating this protection framework into other standards.

5.1 Background

The historical promise of IEEE IP protection for HDL representations is the ability to deliver a faithful representation of the IP and to be able to protect it economically for delivery to a wide audience of IP users. Support of a markup language, a set of pragmas that could be used to augment the original source code, should be all the IP author needs. A model of using a markup that is minimally intrusive to the code is the expectation. The ideal model, in fact, would require no edits to the source code and might be best described as encapsulation of HDL code by these pragmas. If this was truly standardized, it would be possible for an IP author to protect their IP in an interoperable way that could be delivered in a single form that is usable across a set of tools from different suppliers.

However, this did not occur. The state of SystemVerilog, VHDL, and IP protection prior to IEEE Std 1735 was IP protection pragmas could have some ambiguities in definition in the first two standards. It was also not clear how to use these pragmas effectively to achieve the basic goal. Furthermore, some of the pragmas had to change and new ones were needed. Plus, the definition ambiguities needed to be resolved. Therefore, a key first step is to define a mechanism for versioning the set of pragmas to manage change and backward compatibility. IEEE Std 1735 defines a **version** pragma and three distinct versions for the set of protection pragmas covered by this standard.

5.2 version pragma

A new pragma is introduced here to provide versioning for the encryption pragma definitions and use models. It allows the flexibility to make changes and preserve backward compatibility for IP encrypted with a previous version. The syntax [as expected to be expressed in the respective language reference manuals (LRMs)]:

For SystemVerilog:

> **`pragma protect version =** <number>

For VHDL:

> **`protect_version_directive** :: = **` protect version =** integer

The legal values are `0`, `1`, `2` and defined as follows:

> **0**—conforms to the SystemVerilog and VHDL LRMs with tool-specific interpretation of the use model and pragma syntax. There is no expectation of interoperability. This version only exists to support backward compatibility to older implementations.

[9]Information on references can be found in Clause 2

Copyright © 2015 IEEE. All rights reserved.

1—conforms to IEEE 1735 basic interoperability specifications, which are important both because they allow basic interoperability of protected IP across a set of tools and because there is already defacto conformance to them at this time. Clauses and subclauses marked [V1] in this standard contain level 1 conformance requirements.

2—conforms to all IEEE 1735 recommendations for use model and pragmas, including all *Version 1* requirements. Clauses and subclauses marked [V2] in this standard contain level 2 conformance requirements.

Clauses or subclauses not marked [V1] or [V2] are applicable to all versions.

If the **version** pragma is not present in an encryption or decryption envelope, the version is presumed to be level `0` (no expectation of interoperability).

5.3 Basic interoperability

5.3.1 Recommendations required for encryption interoperability [V1]

The required interoperable use case is the digital envelope. The digital envelope uses both symmetric and asymmetric encryption algorithms as well as encoding algorithms. As a consequence, the recommendations on these algorithms are also required and are considered fundamental to *Version 1* interoperability.

In general, algorithms should be freely available with open reference implementations and be complete enough to ensure interoperability. However, in recommending algorithms, there are some concerns. First, other algorithms are already specified in the existing LRMs. Second, these algorithms need to be considered, along with newer algorithms, in light of their adequacy to protect against the latest security threats. To do so, IEEE Std 1735 uses the National Institute of Standards and Technology (NIST) Special Publication 800-131A as the relevant source and standard.

The following terms are used in the remainder of this subclause.

Definitions of NIST recommendations

a) *Approved* means the algorithm is specified in a Federal Information Processing Standards (FIPS) or NIST Recommendation (published as a NIST Special Publication).

b) *Acceptable* means the algorithm and key length is safe to use; no security risk is currently known.

c) *Disallowed* means the specific algorithm and key length have been withdrawn from FIPS or NIST Recommendations.

d) *Legacy use* means the algorithm or key length may be used to process already protected information (e.g., to decrypt ciphertext data), but there may be risk in doing so. Methods for mitigating this risk should be considered.

Definitions of IEEE 1735 recommendations

e) *Recommended* means these algorithms should be used to achieve security and interoperability. Tools are expected to support recommended algorithms in creating or reading *Version 1* or *Version 2* digital envelopes.

f) *Deprecated* means the algorithms should not be used. Tools are only expected to support these algorithms for non-interoperable *Version 0* digital envelopes. Tools are not expected to support deprecated algorithms in creating or reading *Version 1* or *Version 2* digital envelopes.

g) *Legacy use* means the algorithm is no longer recommended, but is still in frequent practical use by either tools or IP authors. Tools are expected to support Legacy use algorithms.

Copyright © 2015 IEEE. All rights reserved.

5.3.1.1 Recommendation for session key encryption

For symmetric session key encryption, the IEEE 1735 recommendations are given in Table 2.

Table 2—Symmetric session key encryption

Algorithm (key length)	Security	NIST recommendation	IEEE 1735 recommendation
DES	Compromised	Disallowed	***Deprecated***
3DES(TDEA) 3 keys version	Secure	Acceptable	***Deprecated***
AES (128 and 256)	Secure	Acceptable	***Recommended***

Although the existing LRMs require the `DES` algorithm, it is no longer allowed by the NIST as its security has been compromised. Therefore, IEEE Std 1735 recommends that the `des-cbc` algorithm be considered Deprecated for use in symmetric session key encryption. Although secure at the moment, `3des_cbc` is not generally believed to be a long-term option, so it too carries the recommendation of Deprecated. `AES` encryption, with key lengths of either `128` or `256`, is required (`aes128-cbc` and `aes256-cbc`, respectively).

5.3.1.2 Recommendation for asymmetric encryption

For asymmetric encryption, the IEEE 1735 recommendations are given in Table 3.

Table 3—Asymmetric encryption

Algorithm (key length)	Operation	Security	NIST recommendation	IEEE 1735 recommendation
Diffie-Hellman	All operations	Varies	Most disallowed	***Deprecated***
RSA (<2048)	Encryption/signing	Questionable	Disallowed	***Legacy use***
RSA (<2048)	Decryption/ verification	Questionable	Legacy use	***Legacy use***
RSA (>=2048)	All operations	Secure	Acceptable	***Recommended***

While mentioned in the existing LRMs, most versions of `Diffie-Hellman` are disallowed by NIST now. Some tools are using `RSA` with key lengths less than `2048`. IEEE Std 1735 recommends that tool vendors stop using these key lengths; but for now, they should be supported by tools consistent with the Legacy use model.

Instead, `RSA` encryption with a key length greater than or equal to `2048` is the recommended asymmetric encryption methodology. For interoperability, it is also necessary to specify encoding and padding. Therefore, `RSAES-PKCS1-V1_5` should be used (see IETF RFC 3447).

Copyright © 2015 IEEE. All rights reserved.

5.3.1.3 Recommendation for encoding

While raw encoding may be useful for tool vendors during internal debugging, `base64` is required for interoperable encoding. Any other methodology may be considered Deprecated.

5.3.1.4 Recommendation for digest algorithms

For computation of digests (see 7.4.3), the IEEE 1735 recommendations are given in Table 4.

Table 4—Calculating digests

Algorithm (key length)	Security	NIST recommendation	IEEE 1735 recommendation
MD4	Compromised	Disallowed	***Deprecated***
MD5	Compromised	Disallowed	***Deprecated***
SHA-1	Questionable	Legacy use	***Deprecated***
SHA-2 (SHA-256,512)	Secure	Acceptable	***Recommended***

The recommended algorithm for calculating digests is `SHA-2`, either `256` or `512`.

5.3.2 Salting the session key for version tampering [V2]

To support tamper-proofing of the **version** pragma, the session key needs to be salted for `version` pragma values of `2` or greater, with a formula based on the version value at encryption time. This ensures an IP author's rights cannot be ignored simply by changing the plain text value of the **version** pragma.

During *Version 2* IP creation, the IP can be encrypted with a **session** key, then salt it by adding (*version pragma value*`-1`), then encrypt the result with public keys to create the key blocks. Since *Version 1* does not allow for salting of the **session** key, the preceding formula effectively adds `0` to a *Version 1* pragma, thus allowing encryption tools to use a single algorithm for both *Version 1* and *Version 2* session keys. When a tool decrypts a *Version 2* IP, decrypt the key block, subtract (`version-1`) to recreate the original session key and then proceed to decrypt the IP data. Addition and subtraction is done modulo $2^{<key\ length>}$ to avoid overflow.

For example, for a version pragma value of 2, to encrypt an IP with a 128-bit AES key of `0123456789ABCDEF`, the IP author adds `1` (*Version 2* minus `1`) for a salted key of `0123456789ABCDF0`. This is the value encrypted and encoded into the key blocks. When processing the decryption envelope, the tool retrieves `0123456789ABCDF0` from its key block, subtracts `1` (*Version 2* minus `1`) to get the unsalted original key `0123456789ABCDEF` that is used to decrypt the data block.

5.3.3 Recommended changes to VHDL and SystemVerilog for basic IP exchange [V1]

There are four major issues to resolve and achieve interoperability here.

The first issue clarifies the `RSA` algorithm conventions. Although both the SystemVerilog and VHDL LRMs just specify "rsa," the conventions that shall be used are `RSAES-PKCS1-V1_5` (see IETF RFC 3447). This essentially specifies the `DER` encoding/padding as the representation and standardizing this makes interoperable solutions possible. This is listed as a requirement in 5.3.1.2.

Copyright © 2015 IEEE. All rights reserved.

The second issue is ambiguity in the **key_block** directive in SystemVerilog, which was resolved in VHDL. The requirement is that the SystemVerilog specification be replaced with the VHDL specification. This also needs to be restated in a consistent way with the style of SystemVerilog in a future revision. The **key_block** specification says the presence of a `key_block` on encryption input implies the digital envelope and the session key should be created and encrypted with the associated public key. In decryption input, the key block is immediately followed by that encrypted session key. There is nothing else in the key block.

The third issue is an omission of the **key_public_key** pragma in VHDL. Its purpose was not understood and it was eliminated in the contributed proposal that led to the final VHDL specification. This matters because the **key_public_key** pragma is the only portable mechanism in *Version 1* to convey a public key to encryption tools. Regardless that IEEE Std 1735 has recommended an additional model for authenticated public key exchange, this update is required.

The fourth issue is SystemVerilog does not have a clear description of how to describe the input for a digital envelope in which multiple tools are allowed to use the protected IP. It is important to know when the values of **key_keyowner**, **key_keyname**, and **key_method** are considered to be finalized and used to specify a particular key block. Directive syntax enables specifying a directive and then respecifying it multiple times where it is the last directive's value that matters. Anyway, this was solved in VHDL, where the **key_block** directive indicates the current values are to be used to specify a key block. Then the IP user can revise directives for the next tool. SystemVerilog needs to have its semantic rules clarified. In particular, it declares the **key_block** directive is illegal on encryption input and not only should it be legal, it is actually vital to the digital envelope model. The description in VHDL can be used as a guide until better LRM text for SystemVerilog can be written.

5.4 Use cases

5.4.1 Digital envelope [V1]

5.4.1.1 Assumptions

All tools are compliant to the applicable LRM standards with respect to encryption/decryption pragmas.

5.4.1.2 Description

An IP author protects their IP with an encryption tool and enables a set of decryption tools to process that IP. The presumption is the set of tools supports one or more complete design flows for that IP author's prospective IP users. The IP author obtains or is supplied a public key for each tool.

The encryption tool is required to support one-time session key generation using any of the IEEE 1735 required symmetric ciphers. IP data is protected with the session key. For each public key provided, the session key shall be encrypted using that public key. This creates a set of digital envelopes that may only be "opened" by the owners of those public keys. These digital envelopes are part of the protected IP delivered to IP users.

The decryption tool needs to support the IEEE 1735 required symmetric ciphers, in particular to use the session key to decrypt the IP. In addition, it has secure access to the private key associated with at least one of the digital envelopes. This flow supports delivery of protected IP directly to IP users from the IP author or via any third party at the IP author's discretion. Any tool in the set of enabled tools may be used with that protected IP.

This use case specifically uses public/private keys to avoid requiring the IP author to distribute keys to IP users or tool vendors. IP users have no keys to receive or manage. Tool vendors need only make public keys available to IP authors for use during encryption. The session key and recommended symmetric ciphers

Copyright © 2015 IEEE. All rights reserved.

ensure strong encryption and fast enough decryption to be suitable for the large amount of data involved in typical IP.

This use case does not limit access to specific IP users. On the contrary, it limits use to specific tools for their intended purpose only. Other mechanisms may be used along with the digital envelope model to limit use, such as licensing. Placing licensing into a digital envelope is detailed in 5.4.2. Licensing is further detailed in Clause 8 and rights management is detailed in Clause 7.

5.4.1.3 Restrictions

While the concept of digital envelopes may be broadly applied to other encryption/decryption scenarios, the use case description here is meant to be specific enough to be deployed and achieve the intended interoperability across tools. The use of a symmetric cipher and its secret key to protect the session key in an envelope is not acceptable in this use case. The use of a supplied secret key by either the IP author or the encryption tool itself to replace the one-time generated session key is also not acceptable in this use case.

Rationale

It does not matter that these restricted variations are allowed by the standard pragmas or may have some value. Here, they are considered inferior, if not bad, practice in terms of key management. If they are justified in terms of both value and acceptable key management practice, a new use case (referenced as a *variation* of this basic use case) is required.

5.4.1.4 Refinements for IP transforming tools

An IP author protects their IP by encrypting it with a supported symmetric cipher. The key for this cipher is itself encrypted in one or more small digital envelopes. Each uses a supported asymmetric cipher and a public key that was published or otherwise made available by the vendor of the tool that reads it. Each digital envelope is private to the consuming tool and cannot be read by other tools.

Tools fall into two classes. First, there are those that read IP, but do not write out anything or write only HDL, which is not intended for use further along in the flow. Second are those that both read and write IP HDL. A simulator is usually a good example of the former and a synthesis tool of the latter.

For tools that both read and write IP, the output is expected to include the original IP or IP transformed according to methods specified by the IP author. The mechanism for specifying these methods is part of rights management, see Clause 7. Assuming the method is to preserve protection with encryption, the IP output is expected to be encrypted with the same session key and cipher as was used to read it, and all digital envelopes are expected to be copied verbatim from input to output. This allows other tools to read the IP. Digital envelopes may not be reconstructed as the content is private and no assumption can be made about the availability of other tool public keys, which are required to create an envelope.

5.4.2 Digital envelope with licensing [V2]

5.4.2.1 Assumptions

This use is a variation on the primary digital envelope use case in 5.4.1. It incorporates the assumptions and description of that use by reference.

5.4.2.2 Description

The digital envelope use case is a primary use case and the only use case that supports interoperability across tool flows. It allows an IP author to authorize a set of tools to work with their encrypted IP. There is nothing in that use case that limits use to particular IP users. Licensing is the appropriate mechanism to further

Copyright © 2015 IEEE. All rights reserved.

restrict access to IP for a particular IP user. In addition, licensing can also be used to control access to various IP features when combined with common rights. All of the issues for the IP author and IP user are common to those found in the conventional licensing of software.

The digital envelope with licensing use case allows IP authors to control access to their protected IP to both a set of tools and IP users. The tools are authorized by proxy to work with the IP. The tools need to further limit access to those IP users who present valid licenses provided to them by the IP author.

The security and flexibility of the license check has clear trade-offs for the IP author. The required support for licensing that a tool vendor needs to provide is described in Clause 8. The IP author needs to deliver a license service as part of their IP if they wish to use this licensing mechanism.

The protocol for expressing what license to check, when it is checked, and how the tool vendor accesses the license service is defined in 8.2. A common use case is where the IP author wishes to disallow decryption based on a failure to find a specific valid license. Combining a license check from Clause 8 with the **decryption** right (see 10.8) can achieve this.

5.4.3 Digital envelope with rights [V2]

5.4.3.1 Assumptions

This use is a variation on the primary digital envelope use case in 5.4.1. It incorporates the assumptions and description of that use by reference.

5.4.3.2 Description

The digital envelope use case allows for interoperability, and the digital envelope with licensing use case limits IP access to authorized IP users only. However, nothing in these two use cases directs a tool as to what visibility, actions, or other access it should allow to an authorized IP user. Currently, an individual tool decides the access it provides typically through default tool behavior when encountering encrypted HDL. Other tool vendors have devised proprietary methodologies allowing IP authors to set rights, but there is currently no expectation that these rights would be honored by other tools.

For rights management that is both interoperable and under IP author control, the digital envelope concept can be extended to include rights. For rights that the IP author intends all tools to honor, this can be done by incorporating a block of common rights into the digital envelope declaration.

In a competitive marketplace, tools are expected to develop unique processes or features where a common right may not apply. A tool vendor may create a specific right allowing the IP author to control the behavior of such a feature. Tool vendor specific rights can be specified within that vendor's key definition block in the digital envelope declaration.

Rights and their specification in the digital envelope declaration are described in Clause 7.

5.4.4 Tool-based secret key

5.4.4.1 Description

In this scenario, the tool vendor has both encryption and decryption tools that have a proprietary secret key for a symmetric cipher built into the tools. The IP author has no key management responsibilities. The IP author protects his IP using an encryption tool, which may subsequently be used by any of that tool vendor's decryption tools that have access to the same secret key.

Copyright © 2015 IEEE. All rights reserved.

This use case is roughly equivalent to the historical **`protect** mechanism in SystemVerilog. It is a proprietary solution and does not allow the protected IP to be used in a general tool flow unless all tools are provided by the same tool vendor.

5.4.4.2 Related use cases

The digital envelope use case can serve a similar role when the IP author uses only the public key of a single tool vendor. This is unlikely to be a key management burden for that vendor's encryption tool.

5.4.4.3 Proposed value judgment for IEEE Std 1735

The digital envelope use case is therefore preferable to the tool-based secret key use case. A single-vendor key in a digital envelope use case covers the primary tool-based secret key use case, with little, if any, usability toll for the IP author. Furthermore, the digital envelope use case is extensible to an interoperable model as previously described.

5.4.5 IP author secret key

Secret keys created by an IP author were originally used as the only option for tool independent secure data exchange. For some IP authors, the control of their own secret key continues to provide a sense of security. IP author secret key exchanges can still be done by encrypting and transmitting files and keys independently, but this falls outside the scope of any interoperable flow described here.

When an IP author's secret key is individually provided to the IP user, and the IP author and IP user are cooperating, that additional layer of protection is real. However, in the case of IP protection, where the IP author and IP user are not necessarily cooperating and ease-of-use is desirable, the use of an IP author created secret key provides no extra protection. Any methods that allow an IP author to transmit a key to a IP user without having direct access to their tools also allow the IP user to retransmit the key, unless steps are taken to restrict who can use the key. Such steps would essentially involve license management, which requires separating its enforcement from decryption. Therefore, the use of an IP author secret key adds no additional security and is counter to the interoperability model described previously.

The IP author secret key is outside the scope of the interoperable model described here and is considered deprecated in this context.

5.5 Secure keyring

5.5.1 Background

Encryption keys for modern ciphers are typically long binary numbers, from 100s to 1000s of bits in length. In the directives for IP protection in VHDL, SystemVerilog, and IEEE Std 1735, a key may be a named item. In general, the unique name for a key is a tuple of the keyowner and keyname.

Depending on the use case, an IP author may have a number of keys to refer to in directives that describe the encryption scenario for their IP. For example, in the digital envelope basic use case, the IP author needs to refer to the public key for each tool across the suite of tools being enabled. Keeping track of key references is one thing; managing the actual keys is another.

One possible and portable approach is to provide the actual key in a pragma. This is an encoded binary string and the **key_public_key** is a pragma for this purpose. It is the only interoperable way of providing such keys for the digital envelope use case in the *Version 1* recommendations. However, managing a set of text snippets, one for each key, that is composed with IP as input to an encryption tool might become tedious and error prone.

Copyright © 2015 IEEE. All rights reserved.

5.5.2 Description

The metaphor of a keyring as a holder for a number of keys is a useful concept for keeping track of encryption keys. The idea of a secure keyring is to provide protection for the keyring such that only an authorized user/application may access the keys. A keyring is a persistent repository for keys. Utility software for a keyring should allow a user to create a keyring, add to or delete keys from it, query what keys are on the keyring, etc. An encryption program that supports a keyring is able to access keys referenced in encryption directives directly from the keyring. An IP author can create a keyring and add keys appropriate to the desired use case for encryption, construct the encryption directives as appropriate, and encrypt the IP.

A secure keyring is a keyring that is protected from unauthorized use by some means. One mechanism, password or passphrase protection, requires the user of the keyring to provide a password to access keys. A utility or encryption program using a secure keyring would require a proper password before access to a key from its store is granted. A decryption tool accessing such a secure keyring would also behave in this manner, expecting its user to know the passphrase. The granularity of protection may be at the individual key or any key on the keyring.

A related, but conceptually alternative, mechanism for protecting a keyring is to encrypt it using the digital envelope concept. Here, a keyring is created and encrypted with a public key for secure delivery of the keyring to the owner of the public key. This can be done with any general encryption tool and then be decrypted upon delivery before using it. Hence, the idea is the tool has been designed to directly access the encrypted keyring.

5.5.3 Role of secure keyrings in IEEE 1735 use cases

How might secure keyrings fit into the standard IP encryption mechanisms and IEEE 1735 use cases for them?

There is nothing explicitly stated about requiring support for the use of keyring mechanisms in the existing VHDL or SystemVerilog for IP protection. Nevertheless, their conceptual use is implied when keys are referenced by name, but not required to be provided directly in the encryption input. An encryption tool may optionally provide such a mechanism as a convenience to its users as a competitive advantage. A decryption tool shall manage its secret or secrets in a private, secure manner. It may harden such secrets directly into its software or use an external persistent storage scheme. Thus, a decryption tool may be conceived of as privately using its own secure keyring.

Copyright © 2015 IEEE. All rights reserved.

6. Key management

6.1 Overview

This clause specifies the representation and data management procedures recommended for keys used by the encryption algorithms in the recommended use cases and processes covered by this standard. Tools implementing this standard are required to supply the public keys identified by the IP to the encryption algorithms; otherwise, successful use of the IP as intended is not possible.

6.2 Key considerations

6.2.1 Naming [V1]

Keys referenced in this standard are identified using two attributes: name and owner. These attributes are strings that are used to retrieve keys from the list of known keys. In addition, keys possess a method attribute that specifies the encryption algorithm for which that key is used. The *owner attribute* identifies the entity or party that created or is responsible for the use of that key. The *name attribute* shall uniquely identify a single key within the scope of the owner attribute. The list of potential keys is narrowed first by the owner attribute, then by the name attribute. If the owner attribute uniquely identifies a single key, then the name attribute may be omitted.

Distributing keys that share the same name and owner attributes is ambiguous, and tools should report an error if a given key shares these attributes with a previously registered key. Previous standards have implied the method attribute could be treated as an identification attribute. However, this standard recommends key owners use the name attribute to distinguish keys that are designated for a common purpose, but differ in the encryption algorithm for which they are intended.

This standard recommends key owners provide both attributes for every key they publish, even if they only publish a single key.

6.2.2 Representation and storage

The list of keys known to a particular tool may be exchanged or represented in one of two standard formats: first, the *pragma exchange format* (the **key_public_key** as specified in 5.5.1) and second, the *certificate exchange format* (X.509 as specified in IETF RFC 5280).

The pragma exchange format is included inside the IP itself as it is compatible with the markup that defines the protected envelopes. There is, however, limited provision for validating identity or establishing trust in the parties who provide access to the key. The certificate exchange format may be used to represent the key pairs required for the digital envelope use model. See 6.4.

This standard does not recommend any use cases where a basic key exchange is specified, see 6.3.

6.2.3 Exchange

Preparation of new protected envelopes requires public keys to be made accessible to the encryption tool. Conveying these keys requires agreement on the representation and method of exchange, so that the integrity of the key exchange is preserved and the resulting protected envelope is usable by the conforming tools for which it is intended.

Copyright © 2015 IEEE. All rights reserved.

6.2.4 Identity and trust

IP authors should verify the keys they use for creating protected envelopes containing their IP. Using unverified keys creates a security risk whereby the content of a protected envelope is open to a tool operating in bad faith. This standard includes recommended practices for applying common key-signing techniques to establish identity and trust.

Signed keys may either be signed by the key owner or by a certification authority that provides a verification service to the owner of an encryption key. The signing party vouches for the identity of the owner, using a signature that can be tested by the tool or by the IP user. Keys that are signed by the key owner do not raise the confidence level for establishing identity, so non-cryptographic approaches are necessary to verify identity. A self-signed key does establish confidence that the key was not corrupted during transmission from the key owner. An example of a non-cryptographic approach is using the tool associated with the key to validate a digital envelope covered by the key.

A public key is embedded in a digital certificate with additional information describing the owner of the public key, such as a name, street address, e-mail address, etc. Although the certificate includes information about the entity, that information may not be accurate. Digital signatures are used to authenticate the certificate. When a certificate is signed by encrypting it with the private key of an asymmetric encryption system, the linkage between that private key and the associated public key serves to validate the holder of the private key has endorsed the certificate. Signing certificate increases confidence in the authenticity of the ownership claim, but only in proportion to the confidence in the authenticity of the signer.

Establishing an identity, using asymmetric encryption, is accomplished by raising the authenticity confidence level past a proof threshold. Some approaches to signing certificates are more satisfying to recipients of that certificate. Unsigned certificates provide no inherent proof of identity. Strong proofs of identity can be obtained by a certificate authority (CA) scheme or a web-of-trust scheme.

A CA attests the public key contained in the certificate belongs to the person, organization, server, or other entity noted in the certificate. The CA's obligation in such schemes is to verify an applicant's credentials, so IP users and relying parties can trust the information in the CA's certificates.

The data embedded in a digital certificate is verified by a CA and digitally signed with the certificate authority's digital certificate. Well-known CAs provide a digital signature hierarchy for validating certificates they have verified and for validating delegations of signing authority.

A *web-of-trust scheme* for validating identity is based on publishing signatures from a diverse group of individuals or entities who can personally vouch for the identity of a key owner. Trust is established when the recipient can construct a linkage to one of these signatures starting from one of their own published signatures. The published signatures are an assertion by the signers that they have verified the identity of the key owner. Web-of-trust schemes are less common than certificate authorities, and require effort on the part of both the key owner and the recipient to establish a path of trust to validate identity.

This standard recommends certificates are signed by one of the established and reputable CAs. If securing such a signature is impractical, the fallback recommended is to self-sign a certificate (for tamper-resistance) and use the tool associated with the certificate to establish the necessary trust threshold for practical use.

6.3 Basic key exchange

Parties exchanging out-of-band keys for cross-encryption need to define the encoding scheme (`DER`, `X.509`, etc), and specify that the keys are provided under basic key exchange rules.

Copyright © 2015 IEEE. All rights reserved.

For interoperability purposes, this standard recommends that the encoding scheme for any out-of-band key exchange should be `DER` encoding.

This standard does not recommend any use cases where a basic key exchange is specified. It is only described here as a recommended practice for working with legacy implementations.

6.4 Standard key exchange

Tools providing standard key exchange consistent with this recommendation shall support at least both the pragma exchange and certificate exchange formats. The tools may also document the format(s) that can be used for decryption envelope processing and for encryption envelope processing.

6.4.1 Pragma exchange format [V1]

The pragma exchange format shall supply the public key using the **key_public_key** pragma, encoded with `DER`, along with any other pragmas needed to identify the key. With this, the IP author has to gather the public keys from the various tool vendor whose tools will process the encrypted IP. The following example shows the use of the **pragma** format:

```
`protect key_keyowner = "ABC"
`protect key_method = "rsa"
`protect key_keyname = "KEY1"
`protect key_public_key
<base64 DER encoded public key>

`protect key_keyowner = "XYZ"
`protect key_method = "rsa"
`protect key_keyname = "KEY2"
`protect key_public_key
<base64 DER encoded public key>
```

6.4.2 Certificate exchange format [V2]

The certificate exchange format shall supply the public key using X.509 certificates as specified in IETF RFC 5280 and ISO/IEC 9594-8. The **key_keyowner** pragma value in an encryption envelope shall match the `CN` component of the *X.509 Subject* for the certificate that contains the desired key. The **key_keyname** pragma value in an encryption envelope shall match the `OU` component of the *X.509 Subject* for the certificate that contains the desired key. The **key_method** pragma value in an encryption envelope shall identify the algorithm indicated in the *Public Key Algorithm* portion of the *X.509 Subject Public Key Info* component for the certificate that contains the desired key. This standard recommends that X.509 Certificate files use the (Privacy Enhanced Mail) Base64 `DER` certificate format, commonly stored in filenames with a `.PEM` extension, and enclosed between

```
-----BEGIN CERTIFICATE-----
```

and

```
-----END CERTIFICATE-----
```

6.4.2.1 Encapsulating keys in digital certificates

Keys referenced from any of the markup pragmas can be encapsulated in the form of a digital certificate. Certificates are a well known and frequently used mechanism in other application domains, such as web

Copyright © 2015 IEEE. All rights reserved.

browsers. In cryptography, a *public key certificate* (or digital certificate) is an electronic document that incorporates a digital signature to bind together a public key with an identity—information such as the name of a person or an organization, their address, and so forth. The certificate can be used to verify that a public key belongs to a specific individual or entity.

Validating a certificate in common practice is accomplished via one of the following methods. Unsigned and self-signed certificates can be validated operationally through the tool as described in 6.2.4. In a web-of-trust scheme, a path of signatures links a known and trusted signer with the certificate owner. The alternative is to get the certificate signed by a globally trusted third party, known as a *certificate authority*.

The digital certificate scheme involves the following steps with respect to the encryption and decryption:

a) Steps during encryption:
   1) Key generation: Generate a pair of public and private keys. IP will eventually be encrypted with this public key.
   2) Generate a digital certificate to embed the public key. The format of the digital certificate is discussed in B.1.
   3) Sign the digital certificate: A signing algorithm is used to sign the digital certificate. Given a message and a private key, the algorithm generates a signature. The message is a digest message computed from the complete certificate information.
   4) Store the digital certificate in the secure keyring.
b) Steps during decryption:
   1) Extract the certificate from a secure keyring.
   2) Verify the certificate using a signature verifying algorithm. Given a message, public key, and a signature, this algorithm either accepts or rejects the certificate.
   3) Retrieve the key from the certificate.

6.4.2.2 Certificate APIs and storage systems

There are open source libraries available, such as OpenSSL, that provide application programmers interfaces (APIs) to generate a certificate, test a new certificate, retrieve a certificate, extract information from a certificate, recognize/verify a certificate, sign a digest, verify a digest, generate a RSA key/public RSA key/ DSA key, etc.

Storing private keys in a secure mechanism is an intrinsic part of any public key infrastructure (PKI)-enabled system. While a certificate can be a public structure, possession of its private key is proof of certificate ownership.

A certificate needs to be stored in a secure keyring that is left implementation-specified. Most libraries that provide certificate services have options for managing, storing, and retrieving certificates. For example, OpenSSL has certain expectations about where certificates are stored and how they would be retrieved. The query parameters for a certificate search are the **key_keyowner** and **key_keyname** pragma values. So given a **key_keyowner** and **key_keyname**, a tool needs to be able search the secure keyring and retrieve a certificate.

6.4.2.3 Validity period

The validity period is commonly chosen based on a number of factors, such as the strength of the private key used to sign the certificate or the amount one is willing to pay for a certificate. This period is the expected interval that entities can rely on the certificate if the associated private key has not been compromised.

Copyright © 2015 IEEE. All rights reserved.

The strength of the key is one of the determining factors in deciding the validity period. The greater the expected cost in time for breaking the cryptography system and recovering the key, the longer the validity period can be. While this standard recommends the use of strong encryption that could be used to influence the choice of a validity period, recommendations on the use of this certificate parameter are not included. The behavior of tools with respect to expired keys is unspecified and tool vendors are encouraged to document their expected behavior with respect to expired certificates.

6.4.2.4 Key publication and validation procedures

This standard recommends certificates be constructed using X.509 certificates as specified in IETF RFC 5280 and ISO/IEC 9594-8, where the following also apply:

- The tools may use any of the standard algorithms (see 5.3.1) to sign the certificate. This standard recommends limiting the choice to one of the algorithms listed as required by this standard. The X.509 specified identifier for that algorithm is specified in the signature algorithm identifier field.
- The issuer name is the name of the signing authority, in the case of a self-signed certificate, it is the company's name to which the tool belongs. However, if a certificate authority is used to sign the certificate, it is the name of the CA.
- The validity period could be a combination of any legal start and end date and time.
- The subject name in all cases is the name of the company to which the tool belongs.

The encryptor tool should be able to read and verify the certificates and extract information from the same.

This standard recommends the **key_keyowner** and **key_keyname** pragma values be used to compose the Subject Name field of the certificate, using the following convention:

CN=<*key_keyowner* value>; OU=<*key_keyname* value>

No mapping considerations are recommended for any other certificate fields.

The tool should be able to generate key blocks for any tool whose certificate is available. Each encryptor tool vendor is encouraged to make these digital certificates containing the public key available with their tools. Distributing these digital certificates with open unrestricted access encourages IP authors to make their IP available for that tool. This standard recommends that such certificates be signed by a CA to reduce the risk of cryptosystem attacks.

6.5 Key scope recommendations

One key should be used by a single product or a set of closely related products from one company that shares a common code base. A single key should not be used by products that perform very different tasks nor should a key be used by a new product that did not exist when the key was first published.

Rationale

The interest in expanding the use of a key is typically in the name of convenience and usability for the IP author. One key for a suite of tools may indeed be convenient. It may also be driven by a desire to create the perception of tight integration of tools and completeness of a tool suite for a given design flow, e.g., a merger and acquisition may have brought missing and technically complementary tools to a tool suite.

Despite such seemingly straightforward considerations, it is not that simple. This recommendation is made in the best interests of security in order to minimize the impact of a key being compromised and in the best interests of IP author decisions for what tools are authorized to work their IP. The following points elaborate on this thinking:

Copyright © 2015 IEEE. All rights reserved.

a) An IP author should have the option to test that a tool protects its IP to its satisfaction prior to publishing the IP with the key block for that tool. It is not acceptable for that key block to allow a future product to process the IP without the IP author's consent. That is best achieved by the future tool using a different key. A possible exception is when a future tool is very closely related in functionality and shares a common code base for supporting IP protection.

b) An IP author may choose to authorize one type of tool from a tool vendor to process its IP and to not authorize another type of tool from the same vendor. This could be for a variety of business, technical, or test resource reasons. The IP author should not be forced to allow consumption by a wide range of tools because it wants to grant access to one type of tool.

c) The scope of a key should be limited to tools that share a common code base. The nature of IP compromise by a tool that would require a new product revision and new associated key is serious. That exposure is not something that should add insult to injury, where forcing revision of other unaffected tools or the re-release of other unaffected IP is the insult.

d) A more flexible approach is to enable easy management of multiple keys with encryption tool features. This can provide the desired convenience without the kind of compromises shared keys bring.

Copyright © 2015 IEEE. All rights reserved.

7. Rights management [V2]

7.1 Introduction

The IP protection pragmas defined in SystemVerilog and VHDL do not provide the IP author with enough control over how the IP may be used. At a high level, the default expectation for protected IP requires the tool vendor to protect all aspects of the IP at all times while allowing the tool to work for its intended purpose. This was a good starting point and early adopters began very conservatively, adjusting their tool behavior to allow protected IP to be usable. Still, such delegation of responsibility to the tool vendor has historically led to some problems. Some tools have been too permissive about information and inadvertently exposed a name or path within the IP in ordinary tool output. At the other end of the spectrum, the IP user encounters results that are unexplainable in their use of the IP, and the information required to provide support by either the tool vendor or the IP author has been suppressed. Experience has shown that these problems are subjective enough that all IP authors will not agree with a single choice. They need more control.

Rights management pragmas are a principal way for that control to be provided. The rights pragmas are a mechanism that can be used flexibly. This should promote innovation in business models and usability of protected IP over time. The rights management capability is present beginning in *Version 2* envelopes. This is not available in *Version 1* envelopes.

7.2 Rights scope

A right can be expressed as a common right that all tools are required to enforce or as a tool-specific right. Tool-specific rights are defined by the tool vendor to tailor that tool's behavior and can be uniquely suited to that tool's purpose and features. This can be powerful. There is no "standard" set of tool-specific rights and the IP author needs to obtain a definition of what rights may be expressed for each tool they wish to support from the appropriate tool vendor. This is also a compromise.

An IP author needs to learn what tool-specific rights are available and apply those to their needs for each tool they enable. Similar tools from different tool vendors will not necessarily provide the same rights. Similar rights and their conditions may be named differently. Rights named the same may have different semantics. Despite these compromises, this is an appropriate way for the IP community (IP users, IP authors, and tool vendors) to develop the most useful set of controls.

Any update to common right names, values, and conditions can only be done with a revision to this recommended practice. By their nature, tool-specific right names, values, and conditions can be updated independently. When updated, the tool vendor has the responsibility to communicate this to IP authors and IP users.

Clause 10 describes the specific common rights that shall be supported. Clause 8 describes the licensing mechanism whereby a common- or tool-specific right can be granted to specific users of a tool rather than to all users of the tool.

7.3 Tool types and rights

Tools can be divided into two categories for purposes of this discussion. *Analysis tools* read protected IP, but do not produce a derivative work of the IP. *Transformative tools* can read protected IP and can produce a derivative work.

Copyright © 2015 IEEE. All rights reserved.

7.3.1 Rights for analysis tools

Analysis tools include simulators, timing analyzers, and formal verification systems. The types of rights most needed for these tool types relate to visibility of objects. With a simulator, for example, the IP author can choose to what extent the names and structure of objects within the IP block are included in output reports, waveforms, or are otherwise accessible to the IP user. A common right often supports the desired control. When a common right is not available, a tool-specific right may be used.

7.3.2 Rights for transformative tools

Transformative tools include logic synthesis, clock-tree insertion, and placers. These tools require the same rights as analysis tools plus additional controls. Rights specific to transformative tools include allowed synthesized target technologies, the amount of protection to be placed on derivative work, and types of modifications allowed on the IP. These rights are generally tool-specific.

7.3.3 Rights for all tools

Some rights may be desirable independent of the tool type. The following examples, however, are not appropriate as common rights. While the concepts are common across many tools, the available values differ and the controls therefore need to be implemented as tool-specific rights.

a) The tool vendor may choose to define a right related to the version of their tool. This might be a minimum version number, a range of version numbers, or a single supported version number. This allows the IP author to restrict use of an IP to tested versions, to versions containing a bug fix, or to versions that support a new right.
b) The tool vendor may also choose to define a right related to a "trim" level of his product. If the tool is available for example in `Basic`, `Professional`, and `Ultra` versions, a right can be expressed to restrict IP use to specific trim levels. This allows the IP author to permit use of an IP only for desired markets or in tested trim levels.
c) The tool vendor can define other rights to allow the IP author to enable or prohibit use of an IP based on conditions of their choosing, e.g., compute platform, time-zone, specific domain.

7.4 Syntax and markup

Both common and tool-specific rights appear in cleartext. The rationale is transparency of the deliverables. A digest prevents the rights from being modified. Markup is language-specific, differing between SystemVerilog and VHDL.

— Common rights, if present, shall be placed in a common block. If no common rights are specified, the defaults described in Clause 10 shall apply. The syntax for this is defined in 7.4.2.
— Tool-specific rights, if present, shall be placed in a tool block. A tool block also contains the key block and a digest. The digest verifies the content of the common block and any tool-specific rights. The digest is required even if there are no tool-specific rights. The syntax for this is defined in 7.4.2.

Tool-specific rights are optional. If empty, common rights plus tool-defaults define the rights. The tool-specific rights digest is mandatory, even for an envelope with no rights specified, and it confirms content has not been removed or changed.

7.4.1 Tool and common blocks

In place of a *Version 1* key block, *Version 2* and later decryption envelopes have a tool block that contains the encrypted key and the tool-specific rights. The structure for VHDL is as follows. (The SystemVerilog syntax is identical, except **`pragma protect** is used instead of **`protect**.)

Copyright © 2015 IEEE. All rights reserved.

```
`protect begin_toolblock
`protect key_keyowner="abc"
`protect key_keyname="def"
`protect key_method="rsa"
`protect rights_digest_method="sha256"
`protect key_block
Base64EncodedEncryptedSaltedSessionKey
`protect control right1="val1"
`protect control right2="val2"
`protect end_toolblock="base64EncodedHMAC256"
```

A tool-rights block without a **key_keyowner**, **key_keyname**, **key_block**, and **rights_digest_method** is invalid. The tool may report an error and shall otherwise ignore that tool block.

A new feature in *Version 2* envelopes is a common block that contains the common rights. The structure for VHDL is as follows.

```
`protect begin_commonblock
`protect control right1="val1"
`protect license_proxyname="xyzzy"
`protect license_symmetric_key_method="aes128-cbc"
`protect license_keyowner="IPCo"
`protect license_keyname="rsa1"
`protect license_public_key
Base64EncodedLicensePublicKey==
`protect control decrypt=license("blurbi") ? "TRUE" : "FALSE"
`protect end_commonblock
```

Salting the key is described in 5.3.2. The digest is described in 7.4.3. License pragmas are described in Clause 8.

7.4.2 Basic syntax

VHDL markup and syntax for a common or tool-specific set of rights is as follows. Additional syntax for conditional rights is described in 7.4.5.

`protect control *right* **=** **"***value***"**

Multiple comma-separated rights may appear on a single line:

`protect control *right1* **=** **"***value1***",** *right2* **=** **"***value2***"**

SystemVerilog syntax is identical except **`pragma protect** is used instead of **`protect**.

White space is allowed but not required on either side of the = (equal sign).

The identifier for a right shall be unique and legal in all languages as transformative tools may read one language and write another. To achieve this, the name of each right shall be composed of lower-case letters, digits, and underscores (_).

The quoted *value* may contain any base64 printable characters other than the quote itself (").

Copyright © 2015 IEEE. All rights reserved.

7.4.3 Rights digest

A digest is associated with each tool-specific set of rights in a decryption envelope. This insures rights were not modified. There is no digest following common rights. There are no digests in encryption envelopes.

The rights are authenticated by a message authentication code, which is created from the pragmas in the common and tool blocks by the keyed-hash message authentication code (HMAC) protocol.

The digest is calculated by following these steps:

a) Concatenate the common block and tool block, strip out trailing white space, convert end-of-lines to UNIX-style, strip out leading **`protect** (or **`pragma protect**) pragmas, strip out leading white space, remove empty lines, and remove the following lines: `begin_commonblock`, `end_commonblock`, `begin_toolblock`, and `end_toolblock`, This canonicalization is necessary so the digest remains valid if the file is moved between platforms with different `newline` characters or is transformed to a different language with a different markup. It is highly recommended transformative tools copy the digest from input file to output file rather than regenerate the digest. If a tool generates a digest, it needs to take the responsibility to validate the original digest of rights—even ones unused by the tool—to prevent tampering of any rights in the input files.

b) Create a message authentication code of the pragmas from the previous step using the keyed-hash message authentication code (HMAC) protocol. The salted session key of the data block is used as the secret key. The hash algorithm used is specified by the rights_digest_method pragma.

c) Encode the hash with `base64` and place it in the **end_toolblock** pragma, as shown in 7.4.3.1.

The canonicalization preserves the validity of the digest through language changes and platform-specific newline differences. Other changes to the rights, such as by mailers or editors, are indistinguishable from tampering and will invalidate the digest. Care needs to be taken not to introduce such changes

7.4.3.1 Example

An example of the block on which to create a digest follows. Notice any language-specific markup is removed. While not visible, UNIX `newline`s are expected here.

Using the data from the examples in 7.4.1, a HMAC of the following would be calculated:

```
control right1="val1"
license_proxyname="xyzzy"
license_symmetric_key_method="aes128-cbc"
license_keyowner="IPCo"
license_keyname="rsa1"
license_public_key
Base64EncodedLicensePublicKey
control decrypt=license("blurbi") ? "TRUE" : "FALSE"
key_keyowner="abc"
key_keyname="def"
key_method="rsa"
rights_digest_method="sha256"
key_block
Base64EncodedEncryptedSaltedSessionKey==
control right1="val1"
control right2="val2"
```


Copyright © 2015 IEEE. All rights reserved.

7.4.3.2 Syntax

The tool needs to validate the digest by computing the digest of the rights and confirming it matches the one in the envelope. A mismatch shall be considered an error by the tool where no further processing of the decryption envelope is allowed. A tool shall validate its own rights. It does not have responsibility for validating rights in other tool blocks intended for other tools. Use the following syntax to specify the digest algorithm.

`protect rights_digest_method="*digest_method***"**

The digest itself is part of the **end_toolblock** pragma:

`protect end_toolblock="*HMAC hash***"**

7.4.3.3 Supported digest algorithms

The following digest algorithms shall be supported as documented in 5.3.1.4.

```
sha256
sha512
```

The digest algorithm shall be specified in each rights digest, e.g.,

```
`protect rights_digest_method="sha256"
```

7.4.4 Tool block and common block rules

— It shall be an error for a tool block to be missing any of the **key_keyowner**, **key_keyname**, **rights_digest_method**, or **key_block** pragmas. The **rights** pragmas and the **key_method** pragma are optional (**rights** default to `none`, **key_method** defaults to `rsa`), as are any encoding pragmas (which default to base64, no line length or character count). It shall be an error for any other pragma to be present in the tool block.

— Common blocks are optional; **license** or **control** directives in the common block are optional. It shall be an error for any other pragma to be present in the common block.

— It shall be an error for the following pragmas to occur outside of common blocks or tool blocks: **control**, **license**, **key_keyowner**, **key_keyname**, **rights_digest_method**, **key_block**, **key_method**, or **license_public_key**.

— It shall be an error for a tool or common block to be nested. It shall be an error for a decryption envelope to open a tool or common block and not to close it.

— There can be at most one common block in a decryption envelope.

— There can be at most one tool block in a decryption envelope with each **key_keyowner**/ **key_keyname** pair.

— Multiple pragmas may appear in a comma-separated list, except **begin** and **end** pragmas, which shall be the only pragma in a statement, e.g.,

```
`protect begin_toolblock
`protect key_keyowner="abc",
`protect key_keyname="def",
`protect key_method="rsa"
```

is equivalent to

```
`protect begin_toolblock
`protect key_keyowner="abc", key_keyname="def", key_method="rsa"
```


Copyright © 2015 IEEE. All rights reserved.

7.4.5 Conditional rights

A right may be made conditional by using the following syntax.

`protect control *right* = *condition* **?** *true_value* **:** *false_value*

Conditions supported by this standard are license checkout and comparison of condition names to constant string values. The condition names and values are either defined in 10.3 for common rights or tool-specific condition names and values. The condition expression can use any of the following standard C language operators. They are shown in order of decreasing precedence.

`()` for defining order of evaluation

`!` indicating NOT

`==` indicating equivalence and `!=` indicating non-equivalence

`&&` indicating AND

`||` indicating OR

`?` and **:** as the conditional operator

White space is allowed, but not required, before and after the preceding characters.

The condition shall be a Boolean. This can be a license call, an equivalence check, or a composite of more than one of these, e.g.,

```
license("abc")
name1 == "value1"
(name1 != "value1" && name2 == "value2") || license ("def")
name1 == "value1" || name1 == "value2"
license("aaa") || license("bbb")
```

Nesting is allowed where the *true_value* and/or the *false_value* is another condition. There is no limit to the nesting depth. Parentheses are not needed but are allowed to improve readability.

The following are equivalent; they show nesting of the *false_value*.

`protect control *right* = *condition1* **?** *true-value1* **:** *condition2* **?** *true-value2* **:** *false_value2*

`protect control *right* = *condition1* **?** *true-value1* **: (***condition2* **?** *true-value2* **:** *false_value2***)**

The following are equivalent; they show nesting of the *true_value*.

`protect control *right* = *condition1* **?** *condition2* **?** *true-value2* **:** *false_value2* **:** *false_value1*

`protect control *right* = *condition1* **? (***condition2* **?** *true-value2* **:** *false_value2***) :** *false_value1*

Nesting can be useful for *false_value* where failure to get one license may be reason to attempt to check out a different license. Here is an example that sets a right to one of three values depending on the license available.

```
`protect control views = license("abc") ? "all" : license("def") ? "some" :
   "none"
```

Short-circuit evaluation is mandatory. If the first clause of an **&&** operator evaluates to `FALSE`, the second clause shall not be evaluated; if the conditional of the **?:** evaluates to `TRUE`, the false leg shall not be evaluated. Evaluation within a precedence level shall be left-to-right. This is particularly important for licenses (see Clause 8) so that licenses are not checked out when it is unnecessary. For example,

Copyright © 2015 IEEE. All rights reserved.

```
`protect control visibility = (license("a") || license("b") || license("c"))
   ? "high" : "low"
```

Here, license "a" is checked and, if granted, "b" and "c" are not attempted. If "a" fails, then "b" is checked and, if granted, "c" is not attempted. If both "a" and "b" are denied, only then is "c" attempted.

7.4.6 Conditional rights examples

Table 5 shows a deliberately complex set of conditions for the right **error_handling** where the value depends on the value of the “activity” condition, the “toolphase” condition, and the ability to check out a license.

Table 5—Example of complex condition combinations

error_handling	synthesis	simulation	analysis
compile	srcrefs	lic(abc) ? plaintext: delegated[a]	delegated[a]
elaborate	lic(abc) ? plaintext : srcrefs	srcrefs	srcrefs
run	delegated[a]	srcrefs	lic(def) ? srcrefs : delegated[a]

[a]The delegated value is defined in 10.3.

This could be implemented using one very long expression with nested conditional statements on the right-hand side of an assignment, but this may be error prone and difficult to read and understand. Instead, the best practice is to break the expression up, such as shown as follows. Notice the first assignment sets a default value and the rest of the assignments set the value differently under some conditions. When those conditions are not met, the previous value is reassigned by having the condition name in the *false_value* of the conditional expression.

```
error_handling = "srcrefs"

error_handling = (activity=="synthesis" && toolphase=="run") ||
    (activity=="analysis" && toolphase =="compile") ? "delgated" :
    error_handling

error_handling = (activity == "synthesis" && toolphase == "elaborate" &&
    license("abc")) ? "plaintext" : error_handling

error_handling = (activity == "simulation" && toolphase == "compile") ?
    (license("abc") ? "plaintext" : "delgated") : error_handling

error_handling = (activity == "analysis" && toolphase == "run" && !
    license("def") ? "delgated") : error_handling
```

A tool-specific right could add its own condition and/or value to be assigned to the right, e.g.,

```
error_handling = (activity == "emulation" && toolphase == "run") ?
    "board_location" : error_handling
```


Copyright © 2015 IEEE. All rights reserved.

7.4.7 Semantic rules for evaluating rights

The structure and order of appearance of rights blocks are discussed in 7.4.4. The syntax of the control pragma for expressing basic and conditional rights is also explained including examples (see 7.4.5 and 7.4.6). The following rules define the semantics for determining the final value of a right:

a) Rights shall have an initial predefined default value.
b) Rights are evaluated in order of appearance as sequential assignments, first in the common block, if it exists, then in the tool-specific block.
c) A right may be assigned a value more than once and the final value is the only value used for that right.
d) A right may be used in the value expression of itself or another right.
e) A right assignment whose value is unused and its contributing value expression(s) do not need to be evaluated.

NOTE 1—The last rule (e) provides an opportunity for efficiency and to avoid the side effects of value expression evaluation, but does not require it. An example of a side effect is a **license()** call (see 8.6) that causes a license to be obtained.

NOTE 2—A tool may determine that an assignment is unused after evaluation of any value expressions and is allowed to return a license that was unnecessarily checked out, but that is also not required behavior.

7.5 Tamper-proof requirements

For a set of rights to be effective, it is imperative that no IP user have the ability to modify them. Rights are expressed in cleartext for transparency, but are protected by a digest as described in 7.4.3.

Threats to the integrity of rights, and the means by which tampering is prevented, are as follows:

a) Removing or changing a rights specification

   This is prevented with the rights digest described in 7.4.3. The digest is a function of common rights appended with tool-specific rights. Any addition, modification, or removal of a right shall invalidate the digest. The absence of a digest is also an error. Since the IEEE 1735 *Version 1* has no rights, there could be a temptation to remove rights and change the **version** pragma to `1`. However, the session key is salted with the **version** pragma value such that the IP cannot be decrypted in this manner.

b) Moving rights from one IP block to another IP block

   This is prevented with the digest since the digest value is a function of the session key and every IP is expected to have a different random session key (see 7.4.3). Failure to have a different session key per IP would leave this vulnerability open. Also, an IP composed of multiple decryption envelopes can share the same session key if the different envelopes are granted identical rights. Therefore, if different envelopes need different rights, they also need to use different session keys to avoid this vulnerability.

c) Moving tool-specific rights intended to control one tool to control a different tool

   This too is prevented by the digest since the digest value is a function of the key name and value associated with one tool.

7.6 Complete tool block and rights example

The following is a full decryption envelope example with real data, except the license syntax is simplified:

```
`protect begin_protected
`protect version=2
`protect author="ACME IP Company", author_info="Silicon Valley Branch, © 2012"
```


Copyright © 2015 IEEE. All rights reserved.

```
`protect encrypt_agent="ACME encyptor", encrypt_agent_info="version 6.7”

`protect begin_commonblock
`protect license_proxyname="xyzzy"
`protect control decrypt = license("dec_acme")? "true" : "false"
`protect control error_handling = "srcrefs"
`protect end_commonblock

`protect begin_toolblock
`protect key_keyowner="VendorA", key_keyname="EDA_A_2012_123",
   key_method="rsa", rights_digest_method="sha256"

`protect key_block
s0N5eK4m1NlNzQr76767GgC9qS4IbDQZbNPA8zhLvyDU7BWDTqI0IdYgmATSWNWxN+nYDohoyR
CoaULFnb+x/Dd2Oj5DwknyutNTKSLW21tt+NTk52Q4T9sRIbAj6z13SW65gZzSQqErfS3YW4I3C+
QtWNV3KcO5O5t+X2xQCqJNOJAFFZu/s7r3nOeQ==
`protect control allowed_view = license("view_lic") ? "schematic" : "none"
`protect control allowed_versions = "2011-2012"
`protect end_toolblock="bDQZbNPA8zhLvyDU7BWDTqI0IdYgmATSWNW"

`protect begin_toolblock
`protect key_keyowner="ToolVendorB", key_keyname="KEY_ABCDE",
   key_method="rsa", rights_digest_method="sha256"
`protect key_block
nyutNTKSLW2CoaULFnb+x/Dd2Oj5Dwk1tt+NTk52Q4T9sRIbAj6z13SW65gZzSQqErfS3YW4I3C+
8zhLvyDU7BWDTqI0IdYg s0N5eK4m1+nYDohoyRNlNzQr76767GgC9qS4IbDQZbNPAmATSWNWxN
AFFZu/s7Q3nOeQ tWNV3KcO5O5t+X2xQCqJNOJr==
`protect control output_type = license ("xx") ? "clear" : license("yy") ?
   "encrypt" : "blackbox"
`protect control allowed_tier = "premium"
`protect end_toolblock="BWDTqI0IdYgmATSWGgC9qS4IbDQZbNPA8zhL"
`protect data_method="aes128-cbc"
`protect data_block
N4jXAVS3vSnPKrgE2sWBn28sB+fQQRxQ5B5oOKAZZdh9Xy96o3Aa6Ncoj4uQR4L8vWI/zZPFgRXR
rgE2sWBn28sN4jXAVS3vSnPKB+fQQRxQ5B5oOKAZI/zZPFgRXRZdh9Xy96o3Aa6Ncoj4uQR4L8vW
AVS3vSnPKrgE2sWBn28sB+fQPFgRXRQRxN4jXQ5B5oOKAZZdh9Xy96o3Aa6Ncoj4uQR4L8vWI/zZ
NAkgxp6IauNFSpdYuAQ2BL6MN03VNQ8=

`protect end_protected
```


Copyright © 2015 IEEE. All rights reserved.

8. License management [V2]

8.1 Introduction

This clause defines the rationale and mechanism for IP licensing. Rights management (see Clause 7) describes how rights can be granted to all users of a tool. License management adds the ability to have a right conditional on the availability of a license. The use case, fully described in 5.4.2, is for the IP author to distribute licenses to some, but not all users of a tool. The license can control which IP users have any access to the IP as described in 10.7. A license can also grant different rights to different IP users. Each common and tool-specific right may or may not support values being conditional on a license, as part of the definition of that specific right.

8.2 License system

The licensing system contains the following components:

— *License proxy* is an executable provided by the IP author. It securely communicates with the tool through a socket.

— *License specification* is several lines in a common block, identifying the proxy, attributes, and a public key where the matching private key is accessible by the license proxy.

— *License use* appears in a rights assignment as described in Clause 7.

— *Proxy communication* is the details of secure message construction and communication between the license proxy and the tool.

8.3 License proxy

This executable is created and distributed by the IP author. It performs two functions. One is to securely communicate with a tool per this standard; the other is to perform an arbitrary function as defined by the IP author to decide whether or not to grant a license. Examples of this arbitrary function are being a proxy for a commercial license management system and checking that a password or serial number is valid.

The proxy-tool communication is through a socket so that the two do not have to be on the same computer or compute platform. This is helpful when the tools are on a lesser-used platform that is not otherwise supported by the IP author or when one proxy is used to service a number of IP users on multiple machines. Communication details are covered in 8.5.1.

8.4 License specification

Within a common block and prior to using a license in a right, a license specification consisting of a set of required and optional license attributes shall be defined. Required attributes include the proxy name and the public key to encrypt communication to the proxy. Optional attributes are any machine or user attributes. For example:

```
`protect license_proxyname="acme_proxy"
`protect license_attributes="USER,MAC,PROXYINFO=1.2"
`protect license_symmetric_key_method="aes128-cbc"
`protect license_public_key_method="rsa"
`protect license_keyowner="ACME", license_keyname="ACME2014"
`protect license_public_key
AV6f6JyGUxBpr49EwxN7jfdUQcRqRFDN7Mto2ltk+emrRCQS+bW/Yvu8U3w9kx7g
rCXFueY/S8lyKUExUP2Yi5C3K9WIFqF7Ov3Hm9/fEumxDzvYkGOGuGu/xv/OYIyb
whEkaiPmVI8+7S/+8NJeOm3BiVaIH5XLcZvz7EW9bvc=
```


Copyright © 2015 IEEE. All rights reserved.

The license specification defines how license requests appearing in the decryption envelope are processed. This implies all licenses in a digital envelope share the same proxy and attributes.

8.5 License proxy parameters

Several statements are required to specify the proxy and the information to be communicated as shown in the previous example. Each is explained in this subclause, which refers to a decryption envelope. An encryption envelope, in contrast, may place any of this information elsewhere, such as referencing a public key by **license_keyowner** and **license_keyname**.

8.5.1 license_proxyname

This mandatory attribute names the proxy. The name determines the environment variable to inspect and find the `port@host` for communicating with the license proxy, e.g., `license_proxyname="myproxy"` causes the tool to look up the value of the environment variable `MYPROXY_LICENSE_PROXY` (see 8.9.1). This presumes the proxy is already running and the environment variable is defined. If a license needs to be checked and this variable is not defined or communication cannot be established with the `port@host` value, the tool shall proceed with the behavior defined by the absence of the license, e.g., the *False* clause of the conditional right, along with a warning that the license checkout could not be attempted.

8.5.2 Attributes

This optional attribute names what machine or user attributes need to be provided to the license proxy, allowing for a license grant conditional on these values. The IP author decides which, if any, attributes shall be used to satisfy their license requirements. The tool shall determine the values of the specified attributes and include *`name`*`=`*`value`* in communication to the proxy as described in 8.9.3. The proxy may use this information to determine if a license will be granted or denied. All tools shall support the following attributes.

The following attributes specify requirements of the user or compute platform where the tool is running:

a) **HOSTNAME** is the standard host name as specified in IEEE Std 1003.1 (POSIX), and its string value is acquired by the tool using the `gethostname()` API defined by that standard. For programming environments not supporting the POSIX standard, an implementation-defined equivalent shall be used.

b) **MAC** is a 48-bit universal LAN MAC address as specified in IEEE Std 802 and shall be represented as a base 16 (hex) string value. For hosts that have more than one network interface with a MAC address, an implementation-defined sort order shall be applied to the list of such interfaces and only the first interface from that list shall be used to retrieve the MAC address. Many of the programming environments that support POSIX provide an implementation-defined API to acquire the MAC address from a specified network interface. For example, in many POSIX-compliant systems, the user-level command `ifconfig` returns an implementation-defined result that includes the MAC address in a printable format. A sequence of system services API calls should provide the equivalent result. Readers of this standard should familiarize themselves with how their programming environment provides the 48-bit universal LAN MAC address for the selected network interface.

c) **USER** is the user name associated with the real user ID of the process as specified in POSIX, and its string value is acquired by the tool using the `getpwuid(getuid())` API defined by that standard. For programming environments not supporting the POSIX standard, an implementation-defined equivalent shall be used.

Copyright © 2015 IEEE. All rights reserved.

The following attribute specifies requirements of the proxy:

d) **PROXYINFO=***value*—this string shall not be validated or modified by the tool and shall be passed to the proxy. The value is alphanumeric, with _, -, and . allowed. Commas or double quotes are not allowed. The proxy can interpret this in arbitrary ways. A typical use would be to treat this as the minimum allowed version of the proxy.

Additional license validation may be performed by the license proxy, such as verifying that the MAC address of the compute platform running the proxy matches a certain value. Proxy-side validation does not affect the contents of the decryption envelope.

8.5.3 Symmetric key method

This mandatory attribute names the cipher to be used with a key for encrypting messages between the tool and the license proxy. The possible values for this are specified in 5.3.

The symmetric key shall be cryptographically random, generated by the handshake protocol between the tool and the proxy. When different proxies, or the same proxy with different public keys, are used at the same time, a different key shall be used for each.

8.5.4 Public key method

This mandatory attribute names the cipher to be used for the public key that follows. The value shall be the same as the **license_public_key_method** pragma.

8.5.5 Key owner and key name

These mandatory attributes name the public key. The key owner is the same as the IP author who provides the license proxy. The key owner and key name contribute to the secure messages, as shown in 8.9.2. This allows the IP author to support more than one key in a single proxy. The use of different public keys with the same proxy causes the tool to open multiple sockets to that proxy, one for each distinct public key.

Each **license_keyowner** / **license_keyname** pair for a proxy is associated with one public key and one symmetric method. The converse is not true—the same public key/symmetric method pair may be associated with multiple **license_keyowner**s or **license_keyname**s. The **license_keyowner**/ **license_keyname** pair shares the same namespace with all other public keys referenced in the encryption and decryption envelopes and shall, therefore, be unique.

8.5.6 Public key

This mandatory attribute is the public key to be used to encrypt the symmetric key. It is `base64`-encoded `DER` format. The matching private key shall be accessible to the license proxy executable.

8.6 License use

Following a license specification in the common block, a license condition may be used on any right that supports it. The basic syntax is

`protect control right = license("*feature***") ? "***true_value***" : "***false_value***"**

Copyright © 2015 IEEE. All rights reserved.

Example

```
`protect control decrypt = license("IP123") ? "true" : "false"
`protect control error_handling = license("IP123_full_visibility") ?
   "plaintext" : "nonames"
```

The syntax and semantics that describe conditional rights are defined in Clause 7.

The **license()** call shall establish communications if necessary with the proxy and issue a license request as described in 8.7. If the proxy grants the license, the **license()** call shall return the Boolean value `TRUE`; if the proxy denies the license, the **license()** call shall return the Boolean value `FALSE`.

A second license call is available when a string rather than a Boolean is required. Its basic syntax is

`protect control right = license_string("*feature*")

Example

```
`protect control myrights = license_string("IP789")
```

The **license_string** function shall return an empty string if the license request is denied by the proxy.

One decryption envelope can contain any number of **license()** calls and any number of **license_string()** calls.

There is no license counting within a tool to proxy session. Making one or several license requests shall behave identically.

8.7 Multiple envelopes

A tool may encounter multiple decryption envelopes with license specifications. Each use case is described here. All content applies to both **license()** and **license_string()** functions.

a) Different **license_proxyname**

   A different proxy is typically used. The **license_proxyname** is used to look up the socket address of the proxy. If more than one **license_proxyname** point to the same proxy, they share the proxy, each with its own socket connection.

b) Same **license_proxyname** and different **license_keyowner** or **license_keyname**

   The same proxy is used for both, but a different socket connection and symmetric key are used.

c) Same **license_proxyname**, **license_keyowner**, and **license_keyname**, but different attributes

   The proxy name, public key, and proxy symmetric cipher match, so the same socket connection shall be used. The tool generates a new license request for every unique license feature combination; i.e., the string passed to the proxy that is computed by concatenating the argument of the license call with the *`name=value`* expansion of the license attributes.

d) Same **license_proxyname**, **license_keyowner**, **license_keyname**, and attributes

   The proxy name, proxy public key, and proxy symmetric cipher match, so the same socket shall be used. The tool shall check if the feature attributes match a license already checked out on that proxy connection and reuse the license returned for any subsequent calls, subject to revocation due to heartbeat failures. This license identity check is a short-cut designed to reduce the amount of traffic over the proxy socket.

A **license_keyowner** / **license_keyname** pair shall be unique as detailed in 6.2.1 and 8.5.5.

Copyright © 2015 IEEE. All rights reserved.

8.8 Proxy communication

The socket communication consists of two different levels. The lowest level is the message, consisting of a length and contents. The length allows the reader to easily detect when the complete message has been delivered. Messages typically are a few dozen bytes (a few hundred for the public key handshake), so having two bytes for the length should be sufficient. The length is specified as most significant byte first.

All messages passed by this protocol have a type indicator as their first character. The two supported message types are symmetric encryption, with a type indicator of `'S'`, and public key encryption, with a type of `'P'`.

The complete public key transmission consists of the two bytes of length, the public key type indicator (`'P'`), two bytes indicating the length of the tag, the message tag, and the encrypted message. The message length is the length of the encrypted text, plus the length of the tag plus two (for the tag length), plus one for the type indicator, or three more than the sum of the number of bytes in the encrypted text and the number of bytes in the tag (see Figure).

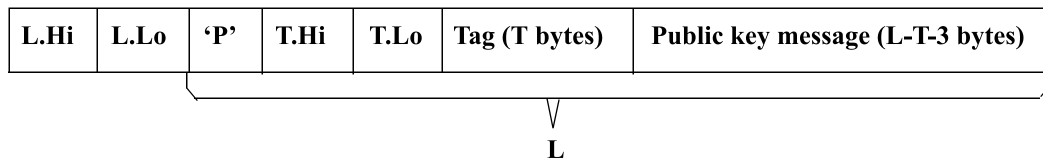

Figure 1—Public key encrypted message

A symmetrically-encrypted transmission consists of the two bytes of length, the symmetric type indicator (`'S'`), the initialization vector, and the encrypted message. The length of the initialization vector (IV) depends on the symmetric key algorithm—AES256 uses a 16-byte initialization vector plus the message, so the message length would be one (the length of the type specifier) plus 16 (the length of the IV) plus the length of the encrypted message (see Figure).

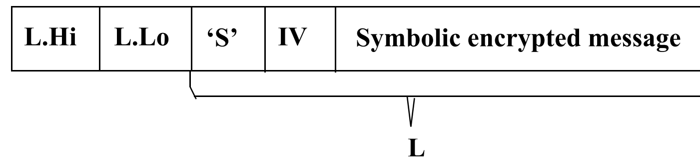

Figure 2—Symmetric key encrypted message

The license interface-setup—key interchange, heartbeats, requests, grants, denials, releases—is built on top of this. Most messages at this level consist of the license id (which doubles as the initialization vector for the encryption), followed by a symmetrically encrypted action and arguments.

8.9 License proxy transactions

The interaction between the tool and proxy can be divided into the following steps.

Copyright © 2015 IEEE. All rights reserved.

8.9.1 Contact the proxy

The tool shall extract the proxy name, public key owner, public key name, and public key from the license attributes of the decryption envelope to determine whether it is necessary to open a new socket. Both the proxy name and public key (the comparison needs to be done with the entire public key, not the public key name) shall match those associated with an existing socket for that socket to be used. If no matching socket can be found, a new one shall be opened.

In addition, the tool needs to be able to determine the host and the port for the proxy; this information is installation-specific and cannot be contained in a file intended for general distribution, such as an IEEE 1735-compliant IP file. Instead, the tool shall construct an environment variable from the upshifted value of the **license_proxyname** attribute, concatenated with `"_LICENSE_PROXY"`, and check its value. The value is expected to be set to the port number the license proxy is listening on and the host it is running on, separated by an `'@'`. So, if **license_proxyname** is set to `"IPPROXY"`, the associated environment variable would be `"IPPROXY_LICENSE_PROXY"` and its value would be of the form `portno@hostname`.

Once the tool has determined the correct port and host, the tool shall attempt to open a socket to the proxy. Once this socket is established, all communications to the proxy with that public key (even for several different licenses) shall go through this socket.

When the tool attempts to contact the proxy, the proxy detects activity on its socket and splits off a new socket for communications to and from the tool. This is all done automatically as part of the socket package.

Up to this point, there has been no actual message passed between the tool and the proxy—the only activity has been opening a socket between the tool and the proxy.

8.9.2 Verify the initial handshake

The tool now verifies the proxy is indeed the proxy for which it was looking.

a) The IP specifies the proxy's public key owner and key name, the actual public key, and the proxy's public and symmetric encryption methods; the tool creates a cryptographically secure random key suitable for that symmetric cipher, encrypts it using the proxy's public key, and sends that encrypted package using the public key owner and key name, separated by a comma, as a tag to the proxy [if the proxy is using a 4096 bit key and the key owner was "owner" and the key name was "key," the encrypted message would be 512 bytes, so 526 bytes would be sent through the socket: two bytes of length information (`0x02` and `0x0C`), one byte of type information (`type=public key`), two bytes of tag length information (`0x0` and `0x9`), nine bytes of tag ("owner key"), and 512 bytes of encrypted message].

b) The proxy receives the message, determines the message type (`public key`), extracts the key owner and key name, attempts to decipher the encrypted part using the private key associated with that key owner and key name, verifies the decrypted message has the same size as its expected symmetric key, creates a new cryptographically secure random permanent key and sets the symmetric key for that socket to that value, creates a pseudo-random IV, encrypts the symmetric key using the tool-supplied temporary key as a key, and sends that back to the tool using the "symmetric key handshake" protocol command [for AES256, the bytes sent are two bytes of length, one byte of message type (`'S'`), 16 bytes of IV, and 48 bytes of message (one byte for the protocol command concatenated with 32 bytes of key, rounded up to the next block size)].

Copyright © 2015 IEEE. All rights reserved.

The length sent would thus be 65 (1 + 16 + 48); the total number of bytes actually sent would be 67 (the two bytes of length plus the 65 bytes of message). See Figure c.

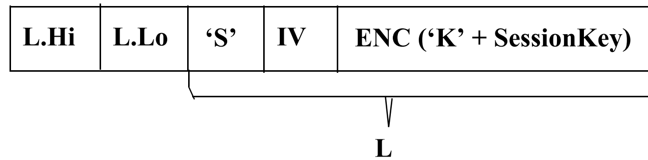

Figure 3—Format of symmetric key handshake response

c) The tool receives and decrypts the reply with the temporary key, verifies that the protocol command is "symmetric key handshake," verifies that the message decrypts properly and is consistent with a key for the specified symmetric algorithm, and sets that as the permanent shared secret key for the socket.

d) If the proxy does not recognize the symmetric cipher tag, or if it fails to decrypt the initial temporary key sent by the tool, or if the key is incompatible with the specified encryption algorithm, the proxy shall close the socket without further communications. The tool shall detect the premature socket closure and treat any license calls through that proxy name/public key setting as if they returned a license denied (and the **license_return_string** is set to the empty string). This failure is likely due to an incorrect setting of the proxy environment variable, the proxy not running on the target machine, or an incompatible proxy on the target machine.

   The tool may choose to report these possibilities to the user.

e) If the tool is unable to decrypt the permanent secret key returned from the proxy or if the decrypted key is incompatible with the specified encryption algorithm, the tool shall shut down the socket and cause any license calls for that proxy to return license denied (and set the **license_return_string** to the empty string).

If the handshake with the proxy has been successfully completed, the tool shall associate the public key used for the proxy and the proxy name with the open socket. Subsequent license requests with that public key and proxy name shall be transacted over that socket.

8.9.3 Request a license

At this point, the tool has examined the proxy name and public key and set up a communications channel and shared secret key with the proxy. The contact and handshake steps (see 8.9.1 and 8.9.2) need only be run once for each unique proxy and proxy public key. Once a secure channel has been established, the actual task of requesting and processing licenses can begin.

a) For each license requested, the tool generates a unique license id whose length is that of the IV of the symmetric encryption method the proxy is using. This license id is used by both the tool and the proxy to distinguish between different licenses (a tool may have multiple licenses open at one time with a given proxy). This license id should be unique for the duration of a given socket—it should not be reused if the tool releases a license and then requests a new license (to avoid attacks where a fake proxy is somehow substituted that just replays exchanges from an earlier run of a real proxy).

b) The tool parses the license string information from the IP rights block (see 8.10) and sends that using the "new license request" protocol, which prepends the "new license request" command byte and encrypts this with the shared symmetric key and the license id as an initialization vector, prepends the license id, and sends the message to the proxy.

c) The proxy receives the message, determines that the message is symmetrically encrypted, decrypts it, determines that it is a new license request, parses the request, and determines whether or not to grant the license (this may involve a call to another license manager).

Copyright © 2015 IEEE. All rights reserved.

Notice the use of a fixed IV is nonstandard. The protocol has been designed so each exchange is different either in the IV or the encrypted text; this should prevent counterfeit proxies from simply replaying earlier transactions with the real proxy.

The license request string is a comma-separated list with *name=value* pairs, including the license feature name and any attributes listed in the rights block for the proxy, e.g.,

```
keyname=acme2014,license=abc,MAC=123456789012
```

8.9.4 Grant a license

The proxy shall analyze the license request and determine (possibly after calling an external license manager) whether the license request should be granted or denied, and send back either a grant or deny response. This is sent via the symmetric-encryption protocol (the license grant is sent encrypted so it can be authenticated, the license denial could reasonably be sent in the clear, but the protocol is more regular if it is encrypted as well).

The argument to the license grant request is the license string the tool used to request the license, optionally followed by a colon and an application-dependent string (this string may be used to convey information outside the scope of this standard); the tool needs to verify the license string portion matches exactly the string that it sent. The license denial argument is the license string optionally followed by a colon (`:`) and a string indicating the reason for the denial.

a) The argument for a grant of the previous example may look like
```
keyname=acme2014,license=abc,MAC=123456789012
```
b) The argument for a denial of the previous example may look like
```
keyname=acme2014,license=abc,MAC=123456789012:All licenses are in use
```

The **license()** function returns Boolean `TRUE` when the license is granted and `FALSE` when denied.

When **license_string()** is used instead of **license()**, the proxy shall return an argument to the license grant request that contains the string. It is the substring following the colon (`:`) separator.

c) The argument for a grant may look like
```
keyname=acme2014,license=def,MAC=123456789012:110001010
```
d) The argument for a denial is identical to when **license()** is used.
```
keyname=acme2014,license=def,MAC=123456789012:All licenses are in use
```

The **license_string()** function in this example returns `"110001010"` when the license is granted and `""` when denied.

Notice that in the first case (a), the grant message is identical to the request message (from 8.9.3). The proxy protocol (see 8.10) ensures that, after encryption, the two do not resemble each other at all unless decrypted with the proxy session key.

license() and **license_string()** both generate the same call to the proxy; the only difference is how the return string from the proxy is interpreted.

8.9.5 Check for heartbeats

The basic socket mechanism allows both the tool and proxy to detect if the socket has been disconnected; the proxy can immediately reclaim any open licenses on that socket. The tool shall immediately cease any processing of licensed IP upon detecting a socket disconnect. The tool may attempt to reestablish

Copyright © 2015 IEEE. All rights reserved.

communication to the proxy, reacquire the license, and resume processing the licensed IP. In addition, the protocol supports heartbeats, both from the tool to the proxy and from the proxy to the tool. The proxy shall require periodic heartbeats from the tool to know the tool is still alive and using the license; the tool shall require periodic heartbeats from the proxy to know the proxy is still alive. Heartbeats may be omitted when tool execution time is very small.

Heartbeats are also encrypted using one of two heartbeat protocols; the heartbeat from the proxy to the tool is sent using the "proxy heartbeat" command, the heartbeat from the tool to the proxy is sent using the "tool heartbeat" command. The heartbeat argument is a 4-byte integer (the UNIX time—the number of seconds since the UNIX epoch). Both the tool and proxy should ensure heartbeat timestamps are reasonable—within a few hundred seconds of their current time, in ascending sequence for a particular license.

Tool and proxy heartbeats use different identifiers to prevent a proxy being replaced with something that just echoes the heartbeats sent from the tool to the proxy back into the tool.

8.9.6 Release a license

When the tool is done with a license, it should send a command to release the particular license. Disconnecting the socket would eventually release the licenses, but the proxy might choose to keep a particular license checked out for a while if the socket is disconnected. The tool may also be using the socket for additional license requests or wish to keep it open in the event of future requests and not have to go through the handshake to establish a new connection.

8.10 License proxy commands

All traffic between the proxy and tool, with the exception of the initial public key encrypted message from the tool to the proxy to establish the shared key, is encrypted with the proxy's symmetric key algorithm. All encrypted commands, with the exception of the "symmetric key handshake" command, use the id of the license being operated on as an initialization vector; the "symmetric key handshake" command has no associated license and uses a pseudo-random number as an initialization vector.

There are several encrypted commands (see Table 6). Each command is one byte in length.

Table 6—License proxy commands

Command	Symbol	Direction	Arguments
Symmetric key handshake	`'K'`	Proxy->tool	Shared symmetric key
New license request	`'N'`	Tool->proxy	License arguments
Grant license request	`'G'`	Proxy->tool	License arguments + optional return
Deny license request	`'D'`	Proxy->tool	License arguments + reject reason
Tool heartbeat	`'H'`	Tool->proxy	Heart beat timestamp
Proxy heartbeat	`'P'`	Proxy->tool	Heart beat timestamp
License release	`'R'`	Tool->proxy	N/A

License grant or license deny messages are typically only sent as replies to the initial license request. It may prove useful if the proxy can send a license deny at any time after the initial license has been granted, to

Copyright © 2015 IEEE. All rights reserved.

allow the license proxy to expire a license after it had been issued for a certain period. The tool would then need to reacquire the license before operations on the IP could continue.

8.11 Deprecated licensing pragmas

The **decrypt_license** and **runtime_license** pragmas defined in SystemVerilog and VHDL respectively are superseded by licensing as a condition in a right. Both pragmas are not interoperable and inherently are very weak as protection mechanisms. It is recommended to remove each of them from SystemVerilog and VHDL, respectively.

Copyright © 2015 IEEE. All rights reserved.

9. Visibility management

9.1 Introduction

The ability to control what is visible and when it is visible in a protected IP is a basic capability that IP authors need. SystemVerilog and VHDL currently do not adequately define the semantics of protection. The existing mechanisms and their limitations are described for clarification. Improved semantics are defined to remove ambiguity in interpretation and provide more usability.

In addition, new mechanisms are defined that provide more flexibility for IP authors to control visibility.

9.2 Background [V1]

The default expectation for a protected IP requires the tool vendor to protect all aspects of the IP at all times while allowing the tool to work for its intended purpose. This responsibility is delegated to the tool vendor by the IP author when the tool is enabled to work with a protected IP. With respect to visibility, this has meant no names inside the IP are visible; hierarchical paths into or through protected IP are opaque; tool and design data inputs from the IP user are not recognized as names inside the IP; and tool outputs including errors and warnings make no references to names or anything else recognizable (such as the original source filenames and line numbers) from the internals of the IP. In other words, this has been interpreted in a conservative manner. In some cases, it has been a draconian interpretation that does more harm than good. However, when the tool has been proven to be unusable for its intended purpose without certain visibility, that has been corrected. There are three areas where default expectation needs more clarity—error messages, model fidelity, and visibility requirements—in the various phases of tool execution.

9.2.1 Error handling

Applying the default expectation to error and warning messages produced by tools is challenging. It applies to any message that refers to information that is protected, regardless of the scope in which the message occurs. Message handling has more gray areas than black and white rules to follow. The fundamental idea is a tool should not reveal information that is an aspect of the IP, such as names of declared objects. That is a painful restriction (see 9.3), but it does not require that the message provide only useless information.

It is desirable for a tool to explain the nature of the error or warning without compromising protection. For example, in most cases, the text of the message can be displayed without substituting actual information for its associated parameters. This may mean substituting values that are not meaningful or obfuscating the protected information (see 10.5.3). It is also reasonable to reword the message when it occurs in the context of protected IP. However, while it is not incorrect to produce a message that gives no information, that scenario is the least usable way to comply with the default expectation.

Finally, a tool vendor may decide not to issue any error or warning message in a protected IP scenario where it otherwise would. The consideration here is the very fact that such a message was issued would reveal information about the protected IP. This last scenario would be a rare one.

9.2.2 Model fidelity

IP protection is a lossless transformation. Visibility of model outputs that have otherwise protected information is required to ensure the model behaves in the same manner when protected as it did in its unprotected state. This occurs when a model is written to produce output that is introspective and by its nature reveals information. This is true for an HDL where the language supports a mechanism and the IP author chose to use that mechanism. Here are a couple of examples:

Copyright © 2015 IEEE. All rights reserved.

`%p` in `$display` in SystemVerilog;

`` `path_name `` or `` `instance_name `` in VHDL.

It is invalid for a tool to modify the behavior of the model that the IP author wrote. The default expectation described in 9.2 only applies to tool behavior, not model behavior. The IP author requires no further mechanism to control visibility in these cases either. The choice to use an introspective feature already belongs to the IP author.

The same principle applies to input visibility if it is the IP author's model that reads inputs and requires such visibility to behave in the same manner when protected as it did in its unprotected state. It needs to be emphasized that this refers to behavior defined in the IP author's protected model using the features of the IP representation, i.e., HDL language constructs. With that definition, it can be considered part of model fidelity.

9.2.3 Behavior outside the model

Model fidelity does not apply to input or output produced outside the model. For example, model fidelity does not apply to tool commands that might expect to reference names in protected IP. The term *command line interface* (CLI) refers to this kind of tool behavior, specifically its interpretation of user input and the output it produces. CLI is a sufficiently general concept to encompass CLI-like operations, such as a SystemVerilog system task that allows the model to execute CLI commands.

It is obvious to say model fidelity refers to the behavior of the model, not the behavior of the tool. What is less obvious, perhaps, is foreign language mechanisms like the SystemVerilog programming interface (VPI), VHDL programming interface (VHPI), and direct programming interface (DPI) that are also considered outside the model from the perspective of protection. Because these mechanisms can be bound independently into the model at the point of tool execution and have no means to be both protected and associated with an encryption envelope, they shall be considered to be outside all encryption envelopes.

CLI, VPI, VHPI, and DPI are discussed further in 9.3.4, 9.6.2, and Annex B

9.2.4 Role of tool phases

Tool phases partition the execution of a tool into a common set of abstract phases. Some visibility is required in certain tool phases for tools to operate with protected IP in their intended manner. This is an important aspect of the default expectation that requires more clarity. Subclause 9.3 provides that clarity and motivates the need for additional controls.

9.3 Visibility in tool phases [V1]

9.3.1 Definitions

Visibility in protected IP refers to a number of things as the default expectation in 9.2 clarified. Names are an important aspect, and these terms are defined to help distinguish certain names and make visibility semantics more precise.

> *Visible name*—has complete visibility inside and outside the model for any purpose; it is not protected. It may originate inside some aspect of an otherwise protected IP, but it is considered outside that IP.
>
> *Protected name*—is part of a protected IP and has no other visibility, except as defined for protected names or as granted by the IP author through some mechanism (to be defined later) such as a viewport or right.

Copyright © 2015 IEEE. All rights reserved.

Disclosed name—is a protected name that has been revealed in some manner to the IP user. It is commonly revealed intentionally by the IP author in documentation or other external mechanism, such as IEEE Std 1685 (IP-XACT). However, there is no distinction made between intentional and unintentional disclosure, or significance to the means by which it happened.

The names of tool phases and their definition are introduced here and are used to support common conditions in conditional rights (see 10.4.1).

Compile—is the abstract phase where the IP is read into the tool and checked in any manner for self-consistency and correct semantics for its representation. It refers to the analysis of protected IP definitions. Conceptually, those definitions are independent of any use of it by the IP user.

Elaborate—is the abstract phase where the IP user's design is being constructed into some information model for use by the tool and where protected IP is being referenced for that purpose.

Run—is the abstract phase where the tool is functioning for its intended purpose and during which it may accept inputs or generate outputs that could contain references to information contained within protected IP.

9.3.2 Compile visibility

During *Compile*, IP may make self-references to protected names that need to be inherently visible. IP users also make references to information in protected IP, such as types and interfaces (parameters, ports, etc.), that need to be visible to support the compilation of their own code. The IP user's understanding of protected names and their purpose within the IP is provided by the IP author through documentation or independent means. In other words, they are disclosed names. Disclosed names are visible during the compile phase for the IP user. All protected names shall therefore be able to be referenced during the compile phase from any HDL code according to the rules of the language. There is no mechanism with IP protection to restrict this compile visibility.

While compile visibility is fundamental, the default expectation implies a tool should not make anything explicitly visible in its tool outputs during compile phase from protected IP. All protected names, disclosed or not, have no visibility during compile for CLI (see 9.2.3).

Nevertheless, there is a need to increase visibility in tool outputs during compile, provided it can be controlled by the IP author. To begin with, the IP author is expected to validate their IP compiles before being protected. However, the IP author may not have verified the protected IP with all the tools in the flow. Having access to all required tools can be expensive, especially for smaller IP author businesses.

Assuming the validation was done as previously described, surely it would then be expected to compile for the IP user after protection without the need to expose anything to the IP user. However, there are a few general ways where compilation may fail.

— The IP user may be allowed to define macros that are referenced within the IP and cause an error.
— The language used to describe the IP (SystemVerilog or VHDL) changes or tightens semantic restrictions after the IP has been protected. There are numerous examples in the history of SystemVerilog.
— The IP user may compile with incompatible language version or tool-specific switches that expose an incompatibility.
— A newer tool version has an unfortunate bug.

Besides unexpected errors in the protected IP itself, it is possible IP users may make errors in referencing types and interfaces from protected IP that cause errors in the compile phase.

Copyright © 2015 IEEE. All rights reserved.

Some of these failures experienced by the IP user can be handled without increasing visibility of the protected IP in tool outputs. However, if the IP author cannot use the identical tool version or cannot get access to the IP user's code, it can be very difficult to resolve issues. Support may be expensive and involve the collaboration of all stakeholders. There can be a catch-22 situation, where both IP user and IP author regard their HDL as proprietary and do not share it openly. An available mechanism for increasing visibility can help in these situations (see 10.5).

9.3.3 Elaborate visibility

Elaborate is the abstract phase where the IP user's design is being constructed and protected IP is being referenced for that purpose. These references in HDL source code include, for example, support for instantiation, hierarchical references, and binding of assertions. During elaborate, the IP may also make self-references to protected names that need to be visible. Many references are finalized at this time and the failure to resolve a reference can be a fatal error. Parameter values are often finalized, which may violate a constraint or expose an inconsistency. All protected names, disclosed or not, shall therefore be able to be referenced during the elaborate phase from any HDL code according to the rules of the language. There is no mechanism with IP protection to restrict this ability to reference protected names during the elaborate phase.

While this is fundamental, the default expectation still implies a tool should not make anything explicitly visible in its tool outputs during elaborate phase from protected IP or references to it from IP user code. All protected names, disclosed or not, have no visibility during elaborate for CLI (see 9.2.3).

A macro is a language construct whose end result after processing is source code. Regardless of where the macro is defined, it is the object names themselves that are protected, not the macro. If the macro is used to reference visible user objects, there is no conflict with errors that reference those names. If the macro references a protected name, however, the fact that the protected name occurs in user code does not matter—the tool shall not disclose it.

The same argument for the impact of an error in a compile reference applies to an elaborate reference. The difference between the compile and elaborate phases and the visibility requirements is only in the details of what can be referenced and the nature of the composition errors the IP user can make. The nature of elaborate errors is the IP user is more likely to cause them than compile errors, and they can be more difficult to resolve without access to both the IP user's and IP author's source code or better visibility in tool outputs.

There is, therefore, a similar case to be made for a mechanism, under the control of the IP author, to increase visibility in tool outputs during elaborate (see 10.5).

9.3.4 Run visibility

Run is the abstract phase where the tool is functioning for its intended purpose and during which it may accept inputs or generate outputs that could contain references to information contained within protected IP. A name that is visible at runtime is equivalent to being unprotected. It is discoverable, may be queried, operated upon, or shown in tools' outputs via CLI. Visible name is an important definition (see 9.3.1).

There is also no requirement for run visibility for protected names implied by the default expectation other than model fidelity (see 9.2.2). The default expectation is a tool should not make anything explicitly visible from protected IP in its tool outputs during the run phase. Tool inputs at runtime also should not recognize references to objects within protected IP or support operations/queries on those objects. Protected names and paths to elaborated objects corresponding to those names are not visible. The tool CLI shall not recognize disclosed names.

Errors during the run phase are the least predictable sort of problems the IP user can encounter. There can be considerable separation from the root cause of the error and where it manifests itself during tool execution.

Copyright © 2015 IEEE. All rights reserved.

Dereferencing a null SystemVerilog class handle or VHDL access type and assignment of values or reference to indexes out of range for a particular object type are examples of problems with this kind of behavior.

Beyond errors, usability of an IP can depend on visibility of essential interfaces to support logging or other runtime interaction for understanding the behavior and debugging of the IP user's use of the IP. There may be an internal register that needs to be visible to provide control, configuration, and/or status. Regardless, the trade-off between run visibility and protection is not a tool's prerogative, but belongs to the IP author. There is value in having a mechanism, under the control of the IP author, to provide run visibility. The rest of Clause 9 and Clause 10 are focused on those mechanisms.

There are some concerns about run visibility, protected IP, and transformative tools. In the IP user's design prior to synthesis, there are potentially some visible names and protected names arising from use of protected IP. The default expectation says the protected names should remain protected in the transformed output unless there are some rights granted to the contrary. Is there a requirement that visible names in the original be preserved as visible in the transformed design? The answer is: no.

Preserving visible names in a transformed design is not a proper IP protection requirement. It is a tool issue and an orthogonal concern. It may be impossible with transformations, such as those that flatten the design and optimize the results, to preserve mappings to original objects. If such mappings are unknown, then visibility cannot be preserved. This may be a usability issue and the IP author may expect to have some control over this. If so, tool-specific rights are likely to emerge in this area, but there are no required rights for transformative tools to manage this visibility issue. The default expectation is all that applies.

9.4 Visibility and encryption envelopes [V1]

Prior to this standard and for *Version 1* interoperability (see Clause 5), there were limited mechanisms to provide run visibility. To provide visibility to a protected name, the IP author needs to make it a visible name. The primary *Version 1* mechanism to control visibility is just the encryption envelope. An IP author could protect subsets of their IP, leaving the remainder unprotected, using multiple, disjoint, encryption envelopes. This was also the only mechanism to extend visibility to tool outputs during the compile and elaborate phases. It is an awkward mechanism, but it is necessary to understand it, including its limitations.

9.4.1 Encryption envelopes

Encryption envelopes define a protected region and are used to control visibility in a body of HDL code. The current definition in both SystemVerilog and VHDL uses **protect** pragmas to delimit these envelopes and can be placed between any tokens in the language. As such, an arbitrary subset of any HDL description can be presumed to be protected with a granularity down to an individual token. Judiciously applied, it suggests any desired partition between fully visible and protected is possible and supportable by using some number of disjoint encryption envelopes.

There are serious shortcomings with this definition. It is not at all clear what it means to partition certain subsets, such as part of a declared object or a type. Semantic rules that are language specific are needed to determine what is legal use in terms of where **protect** pragmas can be placed. This is an unresolved issue. A lack of interoperability stems from this shortcoming. In the face of this, this standard makes a conservative recommendation to achieve interoperability (see 9.4.3). Taken together with alternative mechanisms to control visibility, the recommendation is more easily composed and manageable for IP authors as well.

The relationship between a protected envelope and its surrounding HDL has not had an adequate semantic definition either. This relationship is clarified next.

Copyright © 2015 IEEE. All rights reserved.

9.4.2 Relationship between protected and unprotected regions

HDL code outside a protected envelope and objects or design units marked by a **viewport** pragma (see 9.5.3) are equivalent with respect to run visibility; they both have it. In examining the implications of run visibility, a number of use cases need to be considered. Some cause concerns for the IP author because they may imply unintended run visibility. This needs to be understood and requires additional mechanism to control it.

9.4.2.1 Use cases

For HDL written by the IP user that instantiates or otherwise references protected IP, the fact that the IP user code is unprotected and has full visibility is expected. This creates no issues that compromise a protected IP. When an IP author provides run visibility to IP users for objects, interfaces of design units, or entire design units, there are cases when that intent may cause a compromise. If the object is in the immediate declarative region of a protected design unit that the IP user instantiates, any path to the object is entirely in the IP user's code. This is not an issue. However, if the object is nested deeper in the defined hierarchy of the IP author's code, but the IP author deemed it important to make that internal object visible, it may be a compromise. If this compromise is to be avoided, the structure of the IP author's code needs to change so the path to the internal object is not nested. That is already in the IP author's control.

When an IP author's private implementation of some aspect of their IP uses design units that are also part of the public interface provided to an IP user, the basis for unintended compromise is stronger. Any object that has run visibility implies the path to that object is visible. The potential is that a path through private implementation is therefore exposed. Source code organization and good judgment about what should be made visible are necessary to manage this compromise.

The two use cases are outside the IP author's complete control. A complex IP (call it IP "A") may be composed of other, third-party IP (call it IP "B") from a different IP author. Any IP "B" provided run visibility when instantiated in IP "A" has a path through IP "A" that is visible, by the definition of run visibility. The value of that visibility to users of IP "B," including the IP "A" author, is not disputed. The compromise comes later when it exposes the structure of IP "A" to its users. Whether that is acceptable should be up to the IP author of "A," yet it was defined by IP author "B."

A somewhat similar situation happens with IP "A" when it instantiates unprotected IP user code. This sounds a bit esoteric, but a common example where it can happen is when IP "A" is synthesized to an unprotected technology library of components and primitives. There is a presumption the synthesized output is required to remain protected. Those unprotected library elements, however, become its leaf elements. Because unprotected IP user code has visibility, the paths through the synthesized version of IP "A" become exposed. A mechanism to control this is defined in 10.7.

9.4.2.2 Limitations of envelopes

A protected envelope has no relationship to any other protected envelope. They are disjoint and have no identity. There is no concept that two envelopes are part of the same logical IP and could therefore convey privileges to each other. This is fundamental. An IP author may provide a rights or use mechanism that is associated with a distinct envelope "A." Everything else is outside the envelope. It does not matter whether it is protected or not, it is not a part of "A." As a result, a mechanism to control run visibility cannot be based on any notion of identity. However, it may be based on the hierarchical relationships created by design unit instantiation.

9.4.3 Recommendations

It is recommended that encryption envelopes should enclose one or more complete design unit declarations only and not be used at any finer granularity by IP authors. Design unit granularity in VHDL means only

Copyright © 2015 IEEE. All rights reserved.

whole packages, package bodies, entities, architectures, and configurations can be protected. In SystemVerilog, design unit granularity means only whole packages, modules, programs, and interfaces can be protected. This model fits a common use case for protecting entire libraries of components. It is compatible with encryption tools that support an encapsulation style use model. In that use model, an entire source file or set of source files are protected with protection pragmas defined externally.

This is a recommendation to IP authors to ensure portability of their protected IP, given the limitations described in 9.4.1. It is premature to advise any other use or to recommend deprecating use at a finer granularity. The potential for elaboration issues and structural overhead with finer granularity should be sorted out in the language working groups first. The goal would be to ensure portability and scalable performance was achievable. Regardless, a basic limitation remains that using encryption envelopes to control visibility is costly. An IP author should first prefer to encapsulate their final description with protection without touching the validated source code of the IP. Protection becomes a matter of policy, independent of IP details. Even when the IP author's intent is for a specific subset of names and their definition to have run visibility, using disjoint encryption envelopes requires the most intrusive change to the source code. Using equivalent but more expressive mechanisms to control run visibility should be preferred. **viewport** pragmas and rights are such mechanisms.

9.5 viewport pragmas

There is a **viewport** pragma defined in VHDL and SystemVerilog. It is not considered interoperable. The history is explained here for context, but the intent of this subclause is to define new **viewport** pragmas for *Version 2* that are interoperable. These pragmas provide a detailed mechanism for providing run visibility that involves explicit designation of objects and design units.

9.5.1 Background

The current definitions in SystemVerilog and VHDL are inconsistent. The VHDL **viewport** pragma is portable. It has rigorous syntax because the object designation is tied to the formal information model defined by VHPI. The SystemVerilog **viewport** pragma does not have a rigorous definition. Neither has any concept of execution phases, but appear to be focused only on runtime objects that have values.

There is a concept in both these current **viewport** pragma definitions that goes beyond run visibility; it relates to limiting the kind of access or operations that may be performed on such objects. This qualification of access does not serve IP protection. The semantics of language govern what access is meaningful and the IP author documentation should provide guidance about the purpose of interface or implementation objects having visibility. It is not whether further access restrictions might inherently be useful, it is whether an IP protection mechanism is appropriate to define them. An access restriction mechanism should be orthogonal to IP protection and usable with the IP representation, whether it has been protected or not.

9.5.2 Current viewport pragma definition [V1]

Because of the issues in 9.5.1, this standard recommends deprecating the current definitions and replacing them with better definitions. The **version** pragma makes this possible. Prior to *Version 1*, any tool behavior with the **viewport** pragma shall have a de facto definition and is not interoperable. With *Version 1*, the **viewport** pragma is not supported. If present, tools may choose to ignore it, report an error that the pragma is unsupported, or continue their current behavior.

The following definitions are required for *Version 2* interoperability.

Copyright © 2015 IEEE. All rights reserved.

9.5.3 New viewport pragma [V2]

The **viewport** pragma makes the single name that it refers to and its complete definition have visibility. For HDL IP representations, the use of the **viewport** pragma is restricted to the declaration of objects or entire design units. It may not be used for types and subtypes, or other non-object declarations. It always refers to the entire declaration, not a subset or isolated reference to the name. The syntax is as follows:

For SystemVerilog:

`pragma protect viewport [<*name*> | <*lexically_scoped_name*>]

For VHDL:

`protect_viewport_directive** :: = **` protect viewport [<*name*> | <*lexically_scoped_name*>]

If used to refer to a design unit, the entire design unit would be made visible, both its interface and implementation objects. The following examples illustrate this concept. The parameters of the **viewport** pragma and rules for its use are explained in 9.5.5.

Example 1

The **viewport** pragma in the following SystemVerilog source code makes the next declaration, the `state_machine_mon` module, and all its declarations visible.

```
`pragma protect begin
`pragma protect viewport
module state_machine_mon(err, d, state, clk);
   Parameter DATA_WIDTH = 4;
   output                              err;
   input [DATA_WIDTH-1:0]d;
   input [1:0] state;
   input        clk;
   reg          err;
   reg [1:0] prev_state = `STATE_INITIAL;

   always @(state)
…

endmodule
`pragma protect end
```

Example 2

A **viewport** in a SystemVerilog module only makes the next object declaration, `prev_state`, visible.

```
`pragma protect begin
module state_machine_mon(err, d, state, clk);
   Parameter DATA_WIDTH = 4;
   output                              err;
   input [DATA_WIDTH-1:0]d;
   input [1:0] state;
   input        clk;
   reg          err;
`pragma protect viewport
   reg [1:0] prev_state = `STATE_INITIAL;
```


Copyright © 2015 IEEE. All rights reserved.

```
    always @(state)
…

endmodule
`pragma protect end
```

In Example 1, all declarations in the module are visible. Using `` `pragma protect viewport state_machine_mon `` is equivalent and this could be placed outside the module declaration, provided it remained inside the encryption envelope. In Example 2, only `prev_state` is made visible. Using `` `pragma protect viewport prev_state `` is equivalent and this could be placed immediately above the module declaration of `state_machine_order` or anywhere in the declarative region of the module above the declaration of `prev_state`.

Example 3

A **viewport** for both a VHDL entity and architecture that makes the entire interface and implementation of the design units visible.

```
`protect begin
`protect viewport
entity ff is
  port(
    clk  : in std_ulogic;
    data : in std_ulogic
    q    : out std_ulogic;
    );
end ff;

`protect viewport
architecture only of ff is
begin
…
 end only;
`protect end
```

In Example 3, a **viewport** for the entity makes all the declarations of the entity visible. Given the structure of VHDL, the entity represents the interface of an entity architecture pair and rarely has internal declarations. To make the implementation visible, a **viewport** for the entire architecture declaration is also used.

9.5.4 interface_viewport pragma [V2]

A common scenario is the need to protect the design unit, but allow its interface to be visible. The **interface_viewport** pragma is one way to do that.

The **interface_viewport** pragma makes the logical interface to the named object to which it refers have visibility. For HDL IP representations, the use of the **interface_viewport** pragma is restricted to the declaration of specific kinds of design units that have logical interfaces. The logical interface consists of its formal declarations, e.g., the parameters and ports of a module that is used to instantiate it into a design.

The legal design units allowed in an **interface_viewport** are an entity for VHDL and a module, program, or interface for SystemVerilog.

The **interface_viewport** can be effective for protected IP organized as a set of module or entity/architecture declarations, where a subset is considered public and the remainder is private to the implementation of the IP, but they have not been segregated in separate files. The same protection is possible by other means, i.e.,

Copyright © 2015 IEEE. All rights reserved.

the use of separate encryption envelopes or a right. Such protection would also imply source code organization that separates public and private design units. The syntax is as follows:

For SystemVerilog:

> **`pragma protect interface_viewport** [<*name*> | < *lexically_scoped_name* >]

For VHDL:

> **`protect_interface_viewport_directive** :: = **` protect interface_viewport** [<*name*> |
> < *lexically_scoped_name* >]

The following examples illustrate this concept. The parameters of the **interface_viewport** pragma and rules for its use are explained in 9.5.5.

Example 1

This is an **interface_viewport** for a VHDL entity `ff` that makes its ports and generics visible.

```
`protect begin
`protect interface_viewport ff
entity ff is
  port(
    clk  : in std_ulogic;
    data : in std_ulogic
    q    : out std_ulogic;
    );
end ff;

architecture only of ff is
begin

  p1 : process(clk)
  begin
    if (rising_edge(clk)) then
      q <= data after 1 ns;
    end if;
  end process;
end only;
`protect end
```

Example 2

This is an **interface_viewport** for a SystemVerilog module `ff` that makes its ports and parameters visible.

```
`pragma protect begin
`pragma protect interface_viewport ff
module ff (clk, data, q);
    input  clk, data;
    output q;
    wire q;

    always @(posedge clk)
           q <=  data;

endmodule
`pragma protect end
```


Copyright © 2015 IEEE. All rights reserved.

Example 1 and Example 2 show the use of the **interface_viewport** pragma for VHDL and SystemVerilog, respectively. If there were other non-interface declarations in the entity or module, they would remain protected. The use the **interface_viewport** pragma without specifying the design name is equivalent in these two examples.

9.5.5 General rules for using either viewport pragma [V2]

References to viewport pragma in this subclause refer to both the **viewport** and **interface_viewport** pragmas. A viewport pragma shall be contained within an encryption envelope. If it is used outside an encryption envelope, it has no effect. The declaration it refers to shall be a reference to a declaration contained within the same encryption envelope as the pragma. A warning must be provided for a reference outside the encryption envelope or a reference that is not valid. A viewport pragma with such a reference has no other effect. Note that a reference to an object definition in a conditionally generated structure may not exist in the elaborated design, but would still be considered a valid reference.

The *name* (or *lexically_scoped_name*) is an optional attribute of the pragma. If it is not supplied, the pragma refers to the appropriate declaration in source code that immediately follows the pragma. This positional association is convenient and makes it possible for the IP author to simply express this intention without name lookup. Because it is cohesive with the HDL source code, it enhances maintainability.

If the pragma appears in a declarative region and a name is supplied, it refers to an object of that name in the current declarative region. If it appears outside a declarative region, it refers to an object of that name in the next declarative region that is lexically after the pragma. It may be in the next design unit. A viewport pragma only applies to code that follows it lexically.

Both relative and absolute *lexically_scoped_name*s are supported. These *lexically_scoped_name*s shall refer to definitions in the designated structure of an IP. The syntax of the *lexically_scoped_name* should conform to definitions in the LRMs of the IP representation. In SystemVerilog, the name structure is defined for hierarchical names. It shall be used for lexically scoped names in the viewport pragma. In contrast to hierarchical names, it needs to be emphasized that the lexically scoped name in a viewport pragma is a reference to a definition within a design unit, not an instance specific pathname to an object in an elaborated design. VHDL has formally defined the VHPI `DefName` property and the absolute *lexically_scoped_name* should conform to that. IEEE Std 1735 defines additional rules in this subclause. In the absence of other specific rules for lexically scoped names used with IP protection pragmas in the SystemVerilog or VHDL LRMs, the IEEE 1735 rules shall be used.

Both languages support generated structures using a `for generate` construct. When one makes an object declaration in the declarative region of a `for generate` visible with a viewport pragma, the implication is that all instances of that object are made visible. The *lexically_scoped_name* of the `for generate` statement shall omit the reference to the generate index. An unlabeled lexical scope may not be referred to in a viewport pragma. There are some rules in the LRMs that define implicit naming conventions for unlabeled statements, but code maintenance including the very act of adding a viewport pragma can perturb that implicit naming. The IP author has the ability to explicitly label any scope where it is intended to be referenced in a viewport pragma. These IEEE 1735 rules provide clarity and portability for the viewport pragma with generates.

If the pragma appears within a declarative region and a relative *lexically_scoped_name* is supplied, it refers to a name in the nested declarative region below the current declarative region. If a relative *lexically_scoped_name* appears outside a declarative region, it refers to an object in the nested declarative region starting from a subsequent declarative region. This may be in the first declarative region of the next design unit. If an absolute *lexically_scoped_name* is supplied, it starts from its specified design unit and refers to an object of that name in a nested declarative region within that design unit.

Copyright © 2015 IEEE. All rights reserved.

Taken together, the name and *lexically_scoped_name* allow a viewport specification to refer to any name within a design unit. A *lexically_scoped_name* is never a path name through the design hierarchy of an elaborated design.

Both of the following Example 1 and Example 2 are contrived to show that a lexically nested name is necessary to refer to some object declarations. For these cases, it may be simpler to use a viewport pragma without a *lexically_scoped_name*, but placed just above the desired declaration.

Example 1

This is a viewport to a SystemVerilog nested name in a generate block.

```
`pragma protect begin
`pragma protect viewport foo.internal.a
module foo():
    parameter genblk2 = 0;
    wire a,b,c;

   generate

   if (genblk2)
     begin : internal
       logic a;   // foo.internal.a
     end : internal

   end generate

endmodule
`pragma protect end
```

Example 2

This is a viewport to VHDL control signal in a nested block of an architecture.

```
`protect begin
`protect viewport test:only:block1:control
entity test is
end;

architecture only of test is

begin
...


block1: block is
   port (in1 : in std_logic);
   port map('0');
begin
  control : std_logic;
  ...
end block block1;
end only;
`protect end
```


Copyright © 2015 IEEE. All rights reserved.

9.5.6 Error handling with viewport pragmas [V2]

References to viewport pragma in this subclause refer to both the **viewport** and **interface_viewport** pragmas. A viewport pragma is checked for correct pragma syntax by the encryption tool, but no name lookup or other semantic check is required during encryption. This is essential to the concept that an encryption utility is not required to parse and understand the semantics of an IP representation. That does not imply that it might not be valuable for the encryption utility to do so.

A warning should be given for viewport pragmas placed outside of any encryption envelope. By the end of the elaborate tool phase, a semantic check shall be performed to validate the existence of the name within the IP, as well as the general rules described in 9.5.5. For a locally resolvable viewport name, the semantic check may be performed earlier in the compile phase. In general, because of language features like generate blocks, semantic checks may not be possible in some tools until design elaboration is finalized. Errors and warnings provided to the IP user should be transparent and display any explicit *name* or *lexically_scoped_name* used in the pragma, as appropriate.

9.5.7 Limitations [V2]

References to viewport pragma in this subclause refer to both the **viewport** and **interface_viewport** pragmas.There are some complex issues with viewport usage that need to be clarified in the appropriate LRMs. For example, the use of a parameterized name is not considered interoperable. While the legal objects that a viewport pragma might refer to could be expanded by a particular tool beyond what is described here, that also would not be interoperable. A tool may consider such expansion of viewport features as an error or warning. Though viewport pragmas are a mechanism to allow visibility, tool behavior may optimize that visibility away for other considerations, such as performance. Mechanisms to guarantee visibility at runtime are tool-specific concerns that are orthogonal to IP protection.

9.6 Programming language interfaces

Programming language interfaces (PLIs) refer to VPI, DPI, and VHPI. As noted in 9.2.3, a PLI application, task, or function is considered outside of the model. There is no way to consider them part of a protected IP nor is there a mechanism to identify them with a protected IP for the purpose of conveying specific privileges. There are semantics defined for IP protection with both VPI and VHPI in the respective SystemVerilog and VHDL LRMs. This is discussed in 9.6.1. Finally, SystemVerilog DPI does not have any such semantics defined and it is different in nature from VPI. DPI recommended semantics with respect to IP protection are defined.

9.6.1 VPI and VHPI [V1]

SystemVerilog describes the formal information model for PLI and is the definitive explanation for what can or cannot be done with protected IP. It is useful to provide a general summary here. VPI prescribes that all objects from protected envelopes are marked with an `IsProtected` property. The same is true for instances of packages, modules, programs, and interfaces. In general, all access to relationships and properties of a protected object are not allowed. If a handle to a protected object is obtained, it is possible to determine that the object is protected and access the object's type. It is not possible to place callbacks on protected objects. A protected object may be encountered by navigation. `Handle_by_name()` may not be used to obtain a handle using the disclosed name of a protected object or instance. There is effectively nothing that can be done with a PLI application in a protected IP.

VHDL is more succinct is stating that nothing can be done using VHPI in a protected IP. There are no changes to its formal information model for IP protection defined. The rationale is use cases that describe what wants to be enabled for a protected IP shall be analyzed before information model changes are made.

Copyright © 2015 IEEE. All rights reserved.

This standard adds no additional visibility and provides no further guidance for VPI and VHPI. Annex B discusses some of the issues that need to be considered in providing any visibility to PLI applications.

9.6.2 DPI [V2]

The DPI is a valuable aspect of SystemVerilog. The need for DPI in protected IP is supported by important, widely understood use cases. With a use case in mind and a basic understanding of DPI, the rationale for allowing DPI use is explained. Interpreting the default expectation is not intuitive for DPI, but the details are straightforward. The parts of the DPI C layer that are affected by use with protected IP are noted with recommendations.

9.6.2.1 Use case

A key use case for DPI is a SystemVerilog testbench that uses verification IP and has System C or other behavioral C reference models with which it interacts. Verification IP is reusable technology that has high value and its authors require IP protection. Another use case applies to a virtual prototype for a System on a Chip (SOC), which has high performance models for various system components. Consider the common use of an instruction set simulator (ISS) for the software component, typically written in C/C++. The mechanism that links components that are written in C with SystemVerilog code is DPI. The general statement is that DPI is considered a necessary language feature. If DPI could not be used because of IP protection concerns, these IP would have to be left unprotected. Fortunately, that is not the case.

9.6.2.2 Features of DPI

DPI is used to call C routines from SystemVerilog code (import DPI functions) as well as to call SystemVerilog code from C code (export DPI functions). DPI code exchanges its data by parameter passing. It has no mechanism to navigate an information model and obtain access to other objects; only VPI can do that. It has the concept of a context import DPI function, which associates the DPI function with an instance in the design hierarchy, providing an execution context. The context is essential to enable two important features.

First, the function can save private data with the instance and retrieve it later. Second, context allows exported DPI functions to be called. It enables a third feature, the ability to use VPI within the DPI function, but there is no linkage between VPI and DPI.

The DPI context does not provide a VPI handle, nor does it give a mechanism to obtain one. As a result, the issue of what can be done with VPI from DPI in a protected region is orthogonal. As noted in 9.6.1, nothing can be done and no malicious discovery of information is enabled.

For a protected IP to call import DPI functions or export SystemVerilog functions to be called, its SystemVerilog code needs to be compiled with the appropriate import and export declarations. The essential concept is the IP author explicitly enables DPI to be used by writing these declarations in the protected IP and, for imports, writing the call to the imported foreign routine. There is no malicious path for an IP user to enable unintended import or export DPI with a protected IP.

The foreign code written for a DPI function by an IP user exchanges its data by parameter passing. The interface at the parameters of a DPI function do not allow pass by reference. There is no exposure to protected information available to the IP user from DPI except what can be passed by value from a call inside a protected IP made by the IP author. There is a large body of detail related to argument handling in the SystemVerilog LRM, but it has no impact on IP protection issues.

Of course, it is common that the DPI code itself is actually written by the IP author and necessary to the proper functioning of the protected IP. It is not an assumption that can be used as a basis for enabling DPI

Copyright © 2015 IEEE. All rights reserved.

with protected IP. DPI code has to be considered outside the model as was described earlier in 9.2.3. Fortunately, such an assumption is not needed to ensure protection.

9.6.2.3 Recommendation

The following recommendation enables the use of DPI with protected IP.

There is a set of utility routines provided for use by DPI context import functions. They establish and query the DPI execution context and provide a mechanism to put/get private, instance-specific data. There are seven functions described in the SystemVerilog LRM, Annex H, the (normative) DPI C layer, as follows:

a) `svScope svGetScope();`

b) `svScope svSetScope(const svScope scope);`

c) `const char* svGetNameFromScope(const svScope);`

d) `svScope svGetScopeFromName(const char* scopeName);`

e) `int svPutUserData(const svScope scope, void *userKey, void* userData);`

f) `void* svGetUserData(const svScope scope, void* userKey);`

g) `int svGetCallerInfo(const char **fileName, int *lineNumber);`

`svGetScopeByName` is the key function required to establish the context for an import DPI function; it enables the use of `svSetScope`, `svPutUserData`, and `svGetUserData`. It is effectively the elaboration linkage between DPI and SystemVerilog code. This provides the important features noted in 9.6.2.2 that are essential to the use cases described in 9.6.2.1. `svGetScopeByName` requires a scope name, which is a pathname to an instance. To use this for protected IP, `svScopeByName` shall be allowed to use disclosed names.

There are restrictions. `svGetNameFromScope` is not allowed to return protected names and is required to return meaningless data when called from a protected scope. `svGetCallerInfo` shall not reveal the source code location of the IP author's SystemVerilog code that called the DPI function. The returned values should be appropriately obfuscated. That is unfortunate, because they can be useful for error handling. While the DPI may actually be written by the IP author, that cannot be assumed, as was explained in 9.6.2.2. Of course, this is an aspect that a tool-specific right could enable.

9.7 Controlling visibility with rights [V2]

This subclause has discussed visibility concepts and some specific mechanisms to enable visibility. Using disjoint envelopes and viewport pragmas are effective, but strongly tied to a particular IP. They are embedded in its code. Using rights provides another mechanism for controlling visibility that is not coupled to specific IP source code.

The rights framework (see Clause 7) makes a distinction between tool-specific and common rights. Tool-specific rights are very flexible. They are not standardized across common classes of tools. Negotiation between tool vendors and IP authors shall determine what private visibility controls become available as tool-specific rights.

Now that visibility concepts have been defined, Clause 10 defines a set of common rights that includes the control of visibility for error handling and other common use cases that apply to an entire protected envelope.

Copyright © 2015 IEEE. All rights reserved.

9.8 Visibility of dynamic objects [V2]

The visibility of objects and controls to manage it have been with respect to objects that are defined in the source code of IP representations and give rise to elaborated design objects in an IP user's design. Both SystemVerilog and VHDL also support dynamic objects. In SystemVerilog, it is class instances, dynamic and associative arrays, and queues. In VHDL, it is the access type and any object of its designated type. These objects are referenced with pointers, e.g., class variables and access variables in SystemVerilog and VHDL, respectively. The objects themselves do not have names and are not rooted in the design hierarchy, but may be passed freely and referred to from anywhere in the elaborated design.

From the point of view of IP protection and this standard, only the "pointer" (e.g., class and access) variables named and rooted in the elaborated design hierarchy are subject to protection. The dynamic objects are always visible. A careful understanding shows these objects are only intended to be accessible through those pointer variables. Dynamic objects created and referenced only within protected IP remain sufficiently protected without additional mechanism or protected status.

The visibility of dynamic types recommended here is straightforward and interoperable. There is an alternative conceptual model that a dynamic object could be protected, but the semantics of what defines whether it is are not addressed. Fundamental concerns about protection of types and protection of a subset of a composite type need to be addressed. This is yet another unresolved visibility issue.

9.9 Unresolved visibility issues

It is not practical for this standard to resolve all the issues with interpretation of IP protection. Some are so specific to the IP representation that they require resolution in the governing standard. There are also gray areas where insufficient experience exists to offer more definitive recommendations and the default expectation remains as the sole guidance. See Annex B for some informative discussion of these issues.

Copyright © 2015 IEEE. All rights reserved.

10. Common rights [V2]

This clause defines common rights and conditions, utilizing rights management framework and building on the visibility management concepts of Clause 9. Common rights support interoperability in a tool chain.

10.1 Defining common rights

A right can be expressed as a common right that all tools are required to enforce or as a tool-specific right. Tool-specific rights are defined by the tool vendor to tailor that tool's behavior and can be uniquely suited to that tool's purpose and features. This can be powerful. There is no "standard" set of tool-specific rights, however, and the IP author needs to obtain from the appropriate tool vendor a definition of what rights may be expressed for each tool they wish to support.There is also a compromise as the IP author needs to learn what tool-specific rights are available and apply those to their needs for each tool they enable. Similar competitive tools from different tool vendors do not necessarily provide the same rights. Similar rights and their conditions may be named differently. Rights named the same way may have different semantics. Despite these kinds of compromises, this process is an appropriate way for the IP community (IP users, IP authors, and tool vendors) to develop the most useful set of controls.

A common right means the same thing to all tools. This standard recommends a small number of common rights. Each common right has a specific name, set of legal values, and set of legal conditions that may be expressed. An IP author has a lower burden of understanding the meaning of a common right and a stronger expectation of consistency across a tool chain. That is equally true for the IP user. It is mandatory for the tool vendor to support common rights. This means all legal values of the common right with all allowed conditions and their values shall be supported. If a tool is presented with a common right used in a manner that it does not support, it is required to terminate with an appropriate error.

Common rights are not devoid of compromise. It is a challenge to express a right so it is applicable and appropriate for all tools. The kind of data that IP protection pragmas have been applied to historically has been hardware description language (HDL) source code, but as the scope of what data is being protected widens, it may affect the interpretation of a common right. A well-defined common right with a well-defined set of conditions is expected here, and unanticipated problems with a particular tool or data in the future may be out of scope for the current set of recommendations.

10.2 Overriding common rights

As defined in 7.4.7, an IP author can override a specific common right in a tool-specific rights block. The meaning of a common right expressed in a tool-specific rights block cannot be arbitrarily changed by the tool vendor. Its value can be overridden by another legal value for that right. If that mechanism and the tool-specific rights available for that tool are not acceptable to the IP author, the only other recourse is to not allow that tool to work with his IP until it meets expectations.

10.3 Defaults and the delegated value

Every common right shall have a default value. This is the value that applies if the IP author does not specify the common right in a common rights block. It is not the prerogative of the tool vendor to define a tool-specific default for an unspecified common right.

Every common right shall also have a value, named `delegated`, that means for this common right, accept the default expectation (see 9.3). In effect, the IP author delegates the responsibility to the tool vendor to interpret this common right consistent with the default expectation. The downside is it is possible that similar tools will not have an identical interpretation. The purpose of this value is to make the transition

Copyright © 2015 IEEE. All rights reserved.

from *Version 1* to *Version 2* behavior easy for all stakeholders. Nevertheless, it is the IP author who decides what rights are granted for their IP and whether to use the delegated value.

A useful implication of the rules for rights resolution and the use of the `delegated` value is it shall always be possible to declare a common right in the common rights block and make an exception to it for a particular tool in its tool-specific rights block. Because rights are always in cleartext, this is a very transparent decision to all stakeholders.

10.4 Common conditions for rights

A common condition is a definition of a keyword and legal values that conforms to the rules of 7.4.5. A common condition may be applied to common or tool-specific rights, but does not automatically apply to any right. A proper definition of a right shall include the specification of all valid conditions that may be used to qualify it. Common conditions support consistency across common and tool-specific rights for common concepts. The definition of a common condition may not be redefined by a tool vendor for use in a tool-specific rights block. If there is a need to do so, the tool vendor shall use a different name for the (tool-specific) condition. The definition of a value for a common condition may not be redefined by a tool vendor either. On the other hand, a tool vendor may extend the legal values for a common condition for use within their supported tool-specific right.

10.4.1 toolphase

The *toolphase condition* partitions the execution of a tool into a common set of abstract phases. It provides control for IP authors to limit the applicability of certain rights. This condition is expressed as

toolphase = compile | **elaborate** | **run**

The definitions of the **toolphase** values can be found in 9.3.1.

10.4.2 activity

The *activity condition* classifies a tool. It is useful to factor the broad range of tools useful in design flows into a small common set of abstract activities in order to qualify a right. The tool vendor needs to make it clear to the IP author which activity values apply to their tool. This condition is expressed as

activity = simulation | **emulation** | **synthesis** | **analysis** | **layout**

where

simulation—is the abstract activity that applies to tools that provide execution semantics.

emulation—is the abstract activity that applies to tools that provide execution semantics in real hardware that duplicates the features and functions of a real system, so that it can behave like the actual system.

synthesis—is the abstract activity that applies to tools that transform the IP into another symbolic form.

analysis—is the abstract activity that applies to tools that evaluate the IP without execution semantics and reports on properties derived from the IP specification.

layout—is the abstract activity that applies to tools that transform the IP into rendering instructions for hardware realization.

Copyright © 2015 IEEE. All rights reserved.

It is possible to classify tools in many ways, of course. This current classification is broad enough to apply to almost all the tools in a tool chain where IP protection applies. The values can be extended for use with a tool-specific right, if these are not sufficient.

10.4.3 license

A *license condition* as described in Clause 8 provides a mechanism for granting a right to a specific IP user by the IP author. It is the basis for the digital envelope with licensing use case described in 5.4.2.

A license condition is expressed as a function that accepts a string argument describing the license feature and returns a Boolean value. It may be used in a conditional right, using the syntax of Clause 8.

10.5 Common right for error handling

10.5.1 Background

All tools do error checking and report errors and warnings to the user. The quality of those error messages reflects on the usability of those tools, and by extension, the quality of protected IP. With respect to protected IP, the information that is hidden may define the difference between a useful and a useless error message.

It is impossible to predict all the ways in which a protected IP may be reused and expose an unexpected flaw. An IP user composes a design using protected IP and, in the process, an error occurs. The root cause of that error may be in the IP user's design, in the IP itself, in the use of the IP in an unanticipated manner, or in the implementation of the tool being used. Regardless of the root cause, the first manifestation of the error may be in the context of the protected IP (see 9.3).

What are the reasons IP authors choose to protect information from being made transparent in tool outputs? The fundamental concern is that IP is valuable and needs to be protected. If the original and complete design can be stolen and reused freely, it will damage the business of the IP author. If tool outputs were free to report anything, IP users might reconstruct the IP from them.

So where do error messages fit? Tool outputs are generally user directed, but error messages are fundamentally not. They are responses to real or possible problems. For IP of more than trivial complexity, it is highly unlikely that information in error messages will fundamentally compromise the IP and allow essential information to be stolen. Therefore, there is a good argument that protected IP is more usable with error messages that are transparent and the risk of loss of value will be little to none.

Nevertheless, it may be possible to learn something about the structure of IP by examining the pathname to an internal object. One aspect may just be the identity of an internal supplier whose IP is being reused in the composition of the protected IP. Since all IP authors may not be willing to provide transparent error messages, another alternative that can substitute for transparent names is a reference to an object only understood by the IP author. One such mechanism that is commonly available is a source code reference, e.g., the file and line number associated with the original source code of the IP, provided it has been preserved in the tool.

It should be apparent that balancing concerns of protection with usability is an important decision for the IP author.

10.5.2 Definition

A common right to allow IP authors to control error handling is

Copyright © 2015 IEEE. All rights reserved.

error_handling = delegated | srcrefs | plaintext

where

> **delegated**—this tool complies with the default expectation and may report some information with certain errors to allow the tool to be usable for its intended purpose. This is the default value for error handling.
>
> **srcrefs**—where it is possible for a tool to provide it, a source reference, e.g., to an associated file and line number of an unencrypted source, is to be used wherever a name would otherwise appear. If it is not possible, treat any names as if this setting is `delegated`. **srcrefs** is a straightforward form of obfuscation (see 10.5.3). It may benefit from non-standard line control directives in the original source code, depending on tool behavior.
>
> **plaintext**—any references to protected names in error messages are presented as plain text in the same manner as if they were visible names.

It is possible for **srcrefs** to include other information, such as a column number or other tool-specific obfuscation. The intent is to allow some interpretation of where the name is located in the original source code so the IP author may identify the name without the tool disclosing it. It bears repeating that a tool is not required to support **srcrefs** or may only support it for a subset of references to names or only during a subset of **toolphase**s.

The common conditions that may be applied to error handling are **toolphase**, **activity**, and **license** (see 10.4).

An additional condition is defined to qualify what messages are applicable to this right:

message_severity = fatal | error | warning | note

If no **message_severity** condition is specified, the default is that fatal, error, and warning messages are subject to this right. Notes are excluded by default because they are informative messages that are not classified as issues that might require a response from the user. The classification of messages is the responsibility of the tool vendor.

10.5.3 Obfuscation

Allowing error handling with **srcrefs** is one example of a basic technique to hide information in plain sight —*obfuscation*. Obfuscation can be a lossy transformation. The original source code of the IP can be preprocessed to obscure some internal names, its code structure, eliminate comments, etc.

This preprocessing is not a component of IEEE 1735 IP protection, but a technique identified for IP authors to employ independently. If an IP author takes steps to obfuscate their source code before encrypting it, using the common right for `error_handling = plaintext` can be an effective choice. The obfuscated aspects will then appear in messages and hide the original source code information.

Depending on the way obfuscation is done, the obfuscated aspects may be meaningfully interpreted by the IP author to provide support to the IP user. No one else can likely interpret the obfuscation. IP authors should consider this technique as complementary to IP protection.

10.6 Common right for visibility

There are multiple mechanisms to control visibility. A complete discussion appears in Clause 9. A common right provides a mechanism that applies to the entire IP without requiring identification of specific names. This subclause defines this right and how it is useful.

Copyright © 2015 IEEE. All rights reserved.

10.6.1 Use cases

There is a separation between interface and implementation in IP descriptions. For an IP user, the interfaces need to be open and documented to reference the IP in a design. It is impossible to instantiate a SystemVerilog module or VHDL entity without such information. An IP user cannot write formal assertions for the interface without it either. The implementation is internal and, normally, no aspect of it is needed for reasonable use of an IP.

It is common for an IP author to allow the visibility of interfaces for an IP user while keeping implementation protected. The source code organization of the IP description can support this goal. Keeping public and private source code separate allows distinct encryption envelopes with different visibility to be proscribed for each.

There is a use case in which all names within protected IP, the interface, and implementation are visible. This does not mean the entire source of the IP is visible and unprotected. Protected IP does not always come from a third-party supply chain. An IP user and IP author may, in fact, be members of the same organization, but with different roles. The IP user may be a verification engineer and the IP author a design engineer. What IP protection with this kind of visibility enforces is the discipline that no ability to make design changes or white box testing of the IP is granted to the verification engineer, but complete introspection of data and access to internal controls is. This kind of access is sufficient to allow design assertions to be written that reach beyond the interface. For verification in which coverage metrics are gathered, this would allow all internal coverage details to be exposed.

10.6.2 Definition

A common right to allow IP authors to grant runtime visibility within protected IP is

runtime_visibility = delegated | interface_names | all_names

The common conditions that may be applied to **runtime_visibility** are **activity** and **license** (see 10.4). Run visibility means visibility during the run toolphase only. Visibility has been discussed in 9.3 and the need for additional visibility during the compile and elaborate tool phases is minimal and considered addressed by the common right for error handling. As a result, using **toolphase** as a condition is not allowed for this right.

10.6.3 Finer visibility control

A rule-based mechanism like rights intentionally does not reference design units or other objects by name, which has good value for IP authors who wish to describe protection policy applicable to any IP. If finer control over visibility is required, the **viewport** pragma described in 9.5.3 is the better alternative.

10.7 Common right for child visibility

An IP author has control of run visibility for objects defined within the encryption envelopes they create for their IP. However, if that IP uses other, third-party IP or instantiates unprotected IP user code, the run visibility of those instantiated components, as defined by their IP authors, can compromise this protected IP (see 9.4.2).

The common right for child visibility enables the IP author to allow or deny the run visibility of objects defined outside their encryption envelope that have been instantiated within their encryption envelope. This extends appropriately to transformative tools, such as synthesis that, for example, transform RTL to gate level models and instantiate unprotected components or primitives.

Copyright © 2015 IEEE. All rights reserved.

Of course, **child_visibility** cannot provide any run visibility to any protected name outside its encryption envelope, disclosed or not. It only affects names that are already visible and determines whether they are allowed to remain visible when instantiated in this protected IP.

child_visibility = **delegated** | **allowed** | **denied**

where

allowed—means run visibility to visible child objects is granted, exposing paths through the IP as a consequence.

denied—means run visibility to visible child objects defined outside the encryption envelope is blocked. Those objects are not visible.

delegated—is the default value and visibility may be denied or restricted to meet the default expectation.

The common conditions that may be applied to **child_visibility** are **activity** and **license** (see 10.4). **child_visibility** only affects visibility during the run toolphase. The same rationale described in 10.6.2 applies here. For further discussion of this right and its practical limitations, see Annex B.

10.8 Common right for decryption

Protecting IP for a set of tools using the digital envelope use model grants some usage of that IP to any user by proxy, where the tool is the proxy. Licensing is required to limit the ability to use a protected IP for a particular user. This is described in 5.4.2 as the digital envelope with licensing use case. The common right for decryption effectively enables this use case. Requiring a successful license check before allowing any processing and decryption of protected IP may also improve the security of the tool.

There is a cost to provide a license service for the IP author and a latency cost to the IP user to process the license check. Given the speed of compilation and decryption today, this latency may have a real impact on compilation throughput with protected source files.

Definition

A common right to allow IP authors to control decryption is

decryption = delegated | **true** | **false**

where

delegated—is the default value and is equivalent to **true**.

true—means the IP can be decrypted as part of tool processing.

false—means the IP cannot be decrypted and shall cause a fatal tool error.

The common condition that may be applied to **decryption** is **license** (see 10.4). Indeed, the use of this right only makes sense in conjunction with a **license** condition, e.g.,

```
`protect control decryption = license("basic") ? "true" : "false"
```


Copyright © 2015 IEEE. All rights reserved.

Annex A

(informative)

Bibliography

Bibliographical references are resources that provide additional or helpful material but do not need to be understood or used to implement this standard. Reference to these resources is made for informational use only.

[B1] ISO/IEC 8859-1, Information technology—8-bit single-byte coded graphic character sets—Part 1: Latin Alphabet No. 1.[10]

[10]ISO/IEC publications are available from the ISO Central Secretariat (http://www.iso.org/). ISO publications are also available in the United States from the American National Standards Institute (http://www.ansi.org/).

Copyright © 2015 IEEE. All rights reserved.

Annex B

(informative)

Other known issues with IP protection

There are other considerations that relate to this specification that are important for stakeholders to understand. In some cases, they are unsolved problems. In others, they are opportunities for action outside this standard or where such outside action is anticipated. There may be some suggestions in this annex, but there are no formal recommendations here; it is strictly informative.

B.1 Key management

The supported use cases in IEEE Std 1735 are simple and well defined. They cover the core needs, but there are open issues. While it is easy to deliver interoperable IP, it is a more complicated matter to fix problems after delivery. For example, an IP author may want a delivered IP to stop working with a particular tool revision. The obvious mechanism for that is tool-specific rights and licensing. Is there a related use case for tool vendors?

A tool vendor may wish to limit the use of an older version of their tool with protected IP. Consider the case of a tool vulnerability that could be exploited to expose protected IP. It was unknown when the tool was released and the IP was protected, but then gets discovered. Resolving it requires a new tool revision, but that is only a partial solution; issuing a new public key for use with that new revision is also required. Now, IP protected with the new key cannot be decrypted in the older, vulnerable version. A logical further step is to ensure other IP is no longer protected with that old key. If that happens, it can also be used with that vulnerable tool version and perhaps be compromised. Though it may be painful for the stakeholders involved, all this is currently doable.

The issue that remains is whether something more can be done to prevent potential loss for already released IP. One idea is to proactively limit such losses by having keys that expire after a fixed time period. Another related idea is to develop a mechanism that supports key recall. That idea would require considerable infrastructure and need to overcome some technical problems.

This recommended practice supports the use of key certificates, which offers some convenience and the possibility of using existing public key infrastructure to improve trust. The provenance of the key can be established with keys signed by trusted CAs, for example. Web browsers and client applications that have secure communication mechanisms to servers can do a lot more.

Browser infrastructure that supports key expiration, recall, and update cannot just be reused. The conventional model of certificates having expiration dates and the ability to accept revocation of a certificate works in browsers because it is in the best interest of the browser user not to override or circumvent those mechanisms. The browser user can ultimately decide whether to use an expired certificate, and the risk of loss is only to the browser user.

With IP protection, it is the tool that should no longer allow a public key to be used. A tool can honor the expiration of its own key if the expiration was defined proactively, which can be built into to the tool. Any recall of a key or action that implies the tool needs to communicate with an external service is another matter. Tools are generally not clients of some server that could insist on key recall. Another bottom line is the IP user might not see this as an action that is in his self-interest to allow.

Copyright © 2015 IEEE. All rights reserved.

Even if recall were possible, it would be painful. The ideal action should only affect the parties that desire it, but all IP protected with that key would be affected. That kind of upset to design flows of work in progress would be unacceptable and tool vendors would have a difficult time imposing it on the all stakeholders.

Even expiration would be difficult to impose on all stakeholders. One IP author's view of the length of validity of a tool vendor key is not likely to match every other. More study is required before a use case for managing key lifetimes can be defined and mechanisms to support it are specified.

Going back to the premise that changing keys and managing that has a role, there is an alternative that works in some scenarios. Rather than issuing a new public key on discovery of a tool vulnerability is using a tool-specific right that limits the allowed tool versions. This creates a combination of the key and the version right to authorize use of the tool. Just issuing a new key is never a worse choice, but it is not always better, and it can be more expensive. If the vulnerability includes processing of the version right or if the vulnerability is that the key itself has been compromised, then a new key and the considerations that it implies is the only reasonable remedy.

B.2 Rights framework

There is an intersection between the rights framework and encryption envelopes worth noting. The recommendation for keeping the granularity of encryption envelopes to entire design units has been addressed in Clause 9 (and explored further in B.3). It also matters for handling rights efficiently.

It is reasonable to record that an individual object is protected. This is a very small amount of information, perhaps as minimal as a single flag. On the other hand, the information in a rights block is arbitrarily complex. It will not scale to have an arbitrarily complex set of information associated with individual objects. Rights blocks are applicable to an entire encryption envelope and all the objects in it.

The problem happens when an encryption envelope is crafted to protect an arbitrary subset of a design unit. How can the set of rights be associated with that subset? This is an implementation problem, but it also true that IP protection does not have a requirement to craft rights for individual objects. It is the implication of how rights and envelopes are defined that makes it possible: The lack of stronger semantic rules is an open issue, but the recommendation that any use of encryption envelopes at a finer granularity than design units is not portable also matters for rights blocks. This standard says nothing about the resulting behavior from creating multiple envelopes with different rights within a single design unit.

B.3 Common rights

Common rights and their related common conditions are defined conservatively. They are matched to common characteristics of HDL-based tools and refer to concepts that are broadly applicable to all tools. They allow a seamless transition from basic tool interoperability with the default expectation of protection to a first use of rights.

Many more common rights and conditions are conceivable. The path to common rights is building consensus, because all tools to which they apply are required to support them. Being conservative is necessary for that.

Exploration of other rights that are candidates to be handled in common is well supported by tool-specific rights. Willing IP stakeholders will accomplish this and then find consensus among stakeholders that supporting it has value. Because a common right is currently defined to be applicable to all tools, the possible set is still fairly self-limiting. Another likely outcome is that a common set of rights for a particular class of tools will emerge.

Copyright © 2015 IEEE. All rights reserved.

Finally, the existing definitions of common rights are conservative enough that ideas for feature richness were rejected in some cases. Child visibility offers a good example. A middle ground between allowing visibility of child objects and disallowing it is the idea of using obfuscated paths through protected IP to such objects. This would allow tool outputs to reference those objects, but disallow tool inputs. If such a right has high enough value for a particular tool, it will emerge as an extension to the child visibility common right; however, it will likely first appear as a tool-specific right.

B.4 Visibility

Visibility is a complex area. Though much has been clarified in this standard and new mechanisms created here, some issues remain.

B.4.1 Granularity of encryption envelopes

This has been well considered in 9.4.1 and the recommendation is conservative. It is possible to allow more flexibility in the use of envelopes than just whole design unit declarations and being interoperable. This is the domain of the standards for the supported representations.

That is not to say it is a good idea. The goal of IEEE Std 1735 is to avoid the need for that in all cases. What is required then is the use of other visibility mechanisms. Clearly both rights and the `viewport` pragma are strong steps in this direction.

Regardless, it is recommended the SystemVerilog and VHDL standards define the following:

a) A granularity that is supportable
b) Rules for granularity that, when violated, cause a semantic error
c) When semantic checking of granularity happens

It would be ideal if encryption tools were required to check granularity rules. That is somewhat contradicted in the HDL LRMs, where encryption tools are expected to ignore the HDL text, but that could be improved. It also is in conflict with the goal of encapsulation of any representation of IP with a common set of protection pragmas and with a single, common protection tool. These concerns have yet to be balanced.

B.4.2 Relationship between protected envelopes

The use of multiple protected envelopes interleaved with unprotected subsets of source code to control visibility may have been necessary in the past. Eliminating that need and replacing it with other mechanisms is a goal that rights and the `viewport` pragma substantially address. There will always be the possibility of multiple envelopes, however.

An IP may be protected in a single session of an encryption tool or multiple sessions at different times. An IP user or IP author may employ multiple protected IPs from different IP authors in the composition of their design. From a protection standpoint, what constitutes a single logical IP is neither defined nor enforceable. All of the protected envelopes from all sources have no relationship between each that can be used to define rights or enforce visibility rules.

It is conceivable that establishing an identification mechanism that links disjoint protected envelopes could support privilege within a logical IP and further isolate and segregate the external code of the IP user. There are no requirements to do so today, and this would be a major undertaking if there were.

Copyright © 2015 IEEE. All rights reserved.

B.4.3 Programming language interfaces

Visibility matters when it comes to C code that uses the HDL information model of a design. If everything can be “seen,” it is possible to write a de-compiler. That would be disastrous if it was allowed to operate with protected IP. Those programming language interfaces and the information models that drive them carry protection attributes inferred from protection envelopes. These attributes imply a default visibility in protected regions that prevents such bad things from happening. The default is draconian and does not allow good things to happen either. This standard only clarifies those restrictions at a high level; it does not propose any visibility management features that directly affect PLI.

Programming language interfaces have been used to create valuable tools and models to support analysis and debug. There is a gap in usability of tool flows with protected IP and PLI-based applications because there is no safe mechanism to allow their effective use. That may be OK for now.

If it wants to be solved, what is missing to begin doing so is meaningful, compelling use cases for PLI that require more visibility into protected IP. Consider, for example, a PLI-based tool that does connectivity analysis to compute power metrics or a debug application that looks at driver chains to establish causality of a state within the IP user’s code. Is it possible to write code that obtains obfuscated information for this? Can an enhanced information model correctly promote abstracted information associated with internals of an IP to its public IP interface?

Given use cases, the formal information models and new semantic rules for protected regions would have to be designed to enable access to PLI. The result would also have to be carefully analyzed for vulnerabilities. The scope of that effort is not currently understood. If successful, this would provide a foundation for other “safe” access to information in protected regions that could be used by the tool’s user interface.

B.4.4 Command line interface

There is interest in having disclosed names (see 9.3.1) be recognized in command input within tools and there are cases in which using a particular command will lead to something very desirable. In some cases, the inability to recognize disclosed names in CLI can be disruptive to IP author regression tests that are driven by command inputs. There is also potential for protected information to leak, if this were allowed. Disallowing any privileges with disclosed names is the current decision; the only thing that can be done is to make such names visible or provide a tool-specific right to enable some use of disclosed names and take the associated risks to grant it.

It may not be worth doing more than this. A simple approach is to associate a tool-specific right with a desirable command or commands and let the IP author just enable specific functionality.

If it is worth doing, the same analysis and extension of the formal information that would enable PLI should be considered. If the PLI information model provided a handle to a disclosed name and let meaningful things be done with it, that would provide a basis for tool commands. Another way to say this is that any command that accesses no more information than is allowed for PLI should clearly also be allowed.

B.4.5 Dynamic data

Statically elaborated information stays where it was put in a design, and its protection emanates from the protected envelope where it was defined. Dynamic data used in verification presents some new challenges.

It does not exist when the information model is constructed. One could define a basis under which dynamic objects are considered “protected,” but that has not been done. It would have to be somehow associated with the type of the object. This would be the first notion that a type being protected has a meaningful semantic.

Copyright © 2015 IEEE. All rights reserved.

In order to consider the value of protecting dynamic objects versus the need to keep them visible, various use cases were considered. For example, what should be the visibility of a protected class instance created inside protected IP, but passed outside protected IP to IP user code? What about construction of dynamic data by the IP user using a type whose class definition is inside protected IP? Or construction by protected IP author code using a type that is unprotected?

It is not common to see dynamic data used in design IP because it is not able to be synthesized, though it is possible for synthesis to have white box knowledge of a particular model and replace it with an implementation that has the correct behavior. A more likely place to find dynamic data is in verification IP, e.g., with Universal Verification Methodology (UVM), an Accellera standard, based models. On the other hand, a common use for verification IP is aspects that need to be protected are written in C/C++ code. There is insufficient analysis of dynamic data use cases to argue for what kind of IP protection is needed. When arguments are made, it is very likely SystemVerilog and VHDL expert knowledge will be required to balance out what is desired with what is feasible.

The current visibility model that has been clarified is that all dynamic objects are visible when they are being referenced in IP user code, regardless of where they were created. Keeping them protected, if desired, is a matter of not passing them outside of protected IP in the first place. There is no other mechanism here.

B.5 Obfuscation

Obfuscation was introduced in 10.5.3. There is a small internal use of it for improving error handling. It was also suggested as an additional, external mechanism that an IP author may use to preprocess source code prior to encryption. Used that way, it bears no relationship to IP protection pragmas; it is simply a lossy transformation of the original source code. That may help make it very hard for someone who discovers the obfuscated version of the source code (or snippets of it) to be able to understand it.

Further internal use of obfuscation may help tools both comply with the default expectation and provide a higher quality of result for the IP user. A collaborative approach where an encryption tool supports obfuscation techniques and produces appropriate mapping information may help IP authors provide better support to their customers in the field.

This is a rich area for exploration outside this standard.

B.6 Validating IP protection

An ideal expectation of an IP author is to develop their IP in HDL, protect it as a post-process manufacturing step quickly, and ship it. To insure a quality result, an ideal process would employ regression tests developed with the IP, but now repeated using the protected IP. This would expose latent issues with visibility required by those tests and problems in the tool chain the tests employ. It may even expose problems with the protection specification itself that is not diagnosed by encryption tools, which have a minimal ability and few requirements to do so.

The IP author decides what tools to trust with their IP and a legitimate question is whether to validate that trust with each tool and its production revisions that are allowed to process the protected IP. To do so, the IP author would test their IP with each tool for functional correctness in managing protected information. These are ideal process steps to ensure quality, but what is practical?

Whatever an IP author does to insure correctness of their IP should be made possible to apply to the protected IP. This is an additional requirement on their test construction process, but should be practical with tools that they use internally that support IEEE 1735 IP protection. The remaining tools to which they do not

Copyright © 2015 IEEE. All rights reserved.

have access, yet choose, for business reasons, to allow to process their IP, are an issue. IP authors should be aware of the risk they are taking here. Eliminating the risk is a good idea. This standard offers no further recommendations in this arena.

B.7 Validating tools

The first distinction to make is between functional correctness of the IP protection and resistance of the tool to attempts to defeat the IP protection mechanism. All tools should implement the specification correctly. In particular, there should be no leakage of protected information in accepting tool inputs and producing tool outputs. At the same time, what is visible should indeed be visible (or possible to make so with suitable use of the tool). This is a quality matter that can be objectively validated by tool vendors and independently by IP authors. The risk of loss is born more sharply by the IP author, however.

The resistance of the tool to attempts to defeat the IP protection mechanism is a complex issue. This is a component of the trust model well discussed in 4.4.5. IEEE Std 1735 offers little more than general guidance that tools should employ best practices. Detailed guidance would bring along many issues. The idea of identifying techniques for hacking in this domain and strategies to mitigate them is a double-edged sword. This is a very open-ended domain for software development practice today.

One way to look at it is tool vendors need to consider this issue and define due diligence for themselves to construct a solution that is resistant. IP authors also need to address the problem and not have unrealistic expectations. Therefore, IP authors have their own due diligence to consider and need to consider the role of IP protection carefully (see 4.2).

Both parties may choose to employ ethical hacking to evaluate the resistance of tools. An important reminder is that the tool vendor has the possibility of disallowing such attempts in their license agreements. In fact, there are strong arguments that they should disallow it. IP authors or anyone intending to employ ethical hacking should ensure they have proper permissions.

A hopeful note is that this form of IP protection really does keep honest people honest. It is possible to present a very high bar against compromise. In conjunction with the rule of law and remedies when laws are broken and IP is stolen, IEEE 1735 IP protection can enable better business models and support growth in the IP industry.

Copyright © 2015 IEEE. All rights reserved.

Annex C

(normative)

Protection pragmas

Table C.1 lists the protection pragmas defined or used in this standard. Additional pragmas may be specified by related language standards such as IEEE Std 1800 and IEEE Std 1076.

Table C.1—IEEE 1735 protection pragmas

Pragmas	V1 Encryption envelope	V1 Decryption envelope	V2 Encryption envelope	V2 Decryption envelope	Deprecated
author	X	X	X	X	
author_info	X	X	X	X	
begin	X		X		
begin_commonblock			X	X	
begin_protected		X		X	
begin_toolblock			X	X	
control			X	X	
data_block		X		X	
data_method	X	X	X	X	
decrypt_license					X
encrypt_agent		X		X	
encrypt_agent_info		X		X	
end	X		X		
end_commonblock			X	X	
end_protected		X		X	
end_toolblock			X	X	
interface_viewport			X	X[a]	
key_block	X	X	X	X	
key_keyname	X	X	X	X	
key_keyowner	X	X	X	X	
key_method	X	X	X	X	
key_public_key	X		X		
license_attributes			X	X	

Copyright © 2015 IEEE. All rights reserved.

Table C.1—IEEE 1735 protection pragmas *(continued)*

Pragmas	V1 Encryption envelope	V1 Decryption envelope	V2 Encryption envelope	V2 Decryption envelope	Deprecated
license_keyname			X	X	
license_keyowner			X	X	
license_proxyname			X	X	
license_public_key			X	X	
license_public_key_method			X	X	
license_symmetric_key_method			X	X	
rights_digest_method			X	X	
runtime_license					X
version	X	X	X	X	
viewport			X	X[a]	

[a]These pragmas are encrypted within the data block. All other pragmas are visible in cleartext.

Copyright © 2015 IEEE. All rights reserved.

Consensus

WE BUILD IT.

Connect with us on:

Facebook: https://www.facebook.com/ieeesa

Twitter: @ieeesa

LinkedIn: http://www.linkedin.com/groups/IEEESA-Official-IEEE-Standards-Association-1791118

IEEE-SA Standards Insight blog: http://standardsinsight.com

YouTube: IEEE-SA Channel

IEEE
standards.ieee.org
Phone: +1 732 981 0060 Fax: +1 732 562 1571
© IEEE