INTERNATIONAL STANDARD

ISO/IEC 11889-1

Second edition
2015-12-15

Information technology — Trusted Platform Module Library —

Part 1: Architecture

Technologies de l'information — Bibliothèque de module de plate-forme de confiance —
Partie 1: Architecture

Reference number
ISO/IEC 11889-1:2015(E)

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

CONTENTS

Tables

Figures

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT), see the following URL: Foreword – Supplementary information.

ISO/IEC 11889-1 was prepared by the Trusted Computing Group (TCG) and was adopted, under the PAS procedure, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

This second edition cancels and replaces the first edition (ISO/IEC 11889-1:2009), which has been technically revised.

ISO/IEC 11889 consists of the following parts, under the general title *Information technology — Trusted Platform Module Library*:

– *Part 1: Architecture*

– *Part 2: Structures*

– *Part 3: Commands*

– *Part 4: Supporting routines*

Introduction

Collectively the four parts of ISO/IEC 11889 specify the architecture, data structures, command interface and behavior of a Trusted Platform Module (TPM). A properly constructed platform that incorporates a TPM meeting the requirements may enable establishing trust in platform scenarios involving security and privacy.

TPMs require hardware protections to provide three roots of trust: storage, measurement, and reporting. Basing TPM roots of trust in hardware is an improvement over software based solutions whose protections are vulnerable to malicious software. The architecture defines a TPM that is a passive component that receives commands and returns responses. The commands defined have meticulous descriptions and perform primitive actions on data confidential to the TPM. Typical implementations integrate a TPM in the context of a platform like a laptop or a mobile device. By sending commands to a TPM and processing the responses, security benefits accrue for the platform as a whole. Properly constructed platforms with a TPM can provide hardware based roots of trust for storage, measurement and/or reporting.

The root of trust for storage consists primarily of creating, managing and protecting cryptographic keys and other data values. Artifacts protected by or associated with encryption keys, like passwords, certificates or other credentials, can be used for authentication and many other security scenarios. Cryptographic keys can be created with restrictions on their use or management.

EXAMPLE 1 Cryptographic keys can be created that require a password to be used, have a single purpose (e.g. signing), or cannot be exported elsewhere.

Separate from cryptographic key management, the root of trust for storage also allows data values to be stored in a TPM so they are protected from unauthorized modification or can only be changed in defined ways.

EXAMPLE 2 A data value protected by the TPM might be defined so it can be incremented but not decremented.

Typical use cases can include storing security policy information that can only be updated by an authorized entity or incrementing a value to keep track of failed password entry attempts.

The root of trust for measurement is intended to reflect what software is running on a platform in a trusted way. This root consists of the TPM and other components of a platform that start a chain of measurements. Before software runs, its measurement is calculated and stored in the TPM by sending a command. Later software can add more measurements, but cannot erase its own measurement that was recorded before it started running. If the measurement process unconditionally starts when a platform is powered on, the TPM ends up holding an accurate measurement of all the software running on the system from each power-on. Across power cycles, the same software running on a platform results in the same measurements being present in the TPM. Combining the root of trust for measurement with the root of trust for storage, it is possible to create keys or store data using the TPM that only software with specific measurements can use. The result is specific software on a platform can maintain its own keys, confidential data or security policy information that is not accessible and not able to be tampered with by other software that may run on the platform.

The root of trust for reporting helps entities external to a platform establish trust in platform software measurements or cryptographic keys by proving the values exist in a TPM. TPMs have Endorsement Keys that are essentially unique identities for a TPM. Through commands the TPM provides, it is possible to prove keys exist in a TPM with a specific Endorsement Key. Establishing that a key exists in a TPM permits services external to the TPM to trust a key is protected by a TPM's root of trust for storage. Also, if a key is known to be in a TPM, and the key is used to sign measurements of software, evidence of what software is running on a platform can be shared with an external entity in a trusted way. This enhances privacy protections as the need to share the TPM's Endorsement Key identity directly with a remote verifier can be avoided, resulting in a platform that can anonymously prove to a remote verifier what software is running.

Not all privacy and security use cases enabled by this International Standard are relevant for all platforms. For this reason, this International Standard is defined as a generic library of commands, cryptographic algorithms and capabilities for which a subset can be used to meet the needs of a specific

platform or implementation, with the flexibility to meet diverse and even contradictory global requirement sets. To address the needs of platform types, market segments, regulations, assurance criteria, certification programs, etc., the structure allows complimentary materials to be developed by interested parties constraining the generic library to address specific requirement sets. To promote product interoperability, implementers are encouraged to consult existing specifications augmenting this International Standard with domain specific considerations for specific applications and platform types. A platform specific context is able to articulate scenarios, functionality, relevant security and privacy goals, implementation considerations and methods of assurance. Because of the generic nature of this International Standard, certification programs are likely to be based on additional specifications that further define platform specific implementations and security characteristics. Having a single TPM library that implementations across different platforms types can use to add security and privacy benefits lowers complexity and permits reuse of software and security analysis across the trusted computing ecosystem.

EXAMPLE 3 Opt-in versus secure by default are examples of contradictory requirements for TPM provisioning that are both supported by this International Standard.

This International Standard is useful for a variety of audiences. Two key audiences are TPM implementers and adopters. Clauses 6 and 7 in this part of ISO/IEC 11889 are useful to orientate audiences to the organization of the four parts and how they are used to build a compliant implementation. Implementers will need to use all four parts to build a compliant implementation. Adopters can benefit from the architectural concepts in this part of ISO/IEC 11889 when developing scenarios and incorporating a TPM into a platform design. ISO/IEC 11889-2 and ISO/IEC 11889-3 are beneficial for adopters to understand the syntax and semantics of using individual TPM commands.

Implementers and adopters of this International Standard need to carefully assess the appropriateness of controllability, security and privacy capabilities and algorithms implemented for satisfying their goals. In assessing algorithms, implementers and adopters should diligently evaluate available information. Solutions involving cryptography are dependent on the solution architecture and on the properties of cryptographic algorithms supported. Over time, cryptographic algorithms can develop deficiencies for reasons like advances in cryptographic techniques or increased computing power. Solutions that support a diversity of algorithms can remain durable when subsets of supported algorithms wane in usefulness. Therefore, implementers intent on providing robust solutions are responsible for evaluating both algorithm appropriateness and diversity.

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this document may involve the use of a patent.

ISO and IEC take no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured the ISO and IEC that he/she is willing to negotiate licences either free of charge or under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with ISO and IEC. Information may be obtained from:

Fujitsu Limited **1-1, Kamikodanaka 4-chrome, Nakahara-ku, Kawasaki-shi, Kanagawa, 211-8588 Japan**
Microsoft Corporation **One Microsoft Way, Redmond, WA 98052**
Enterasys Networks, Inc **50 Minuteman Road, US-Andover, MA 01810**

Lenovo **1009 Think Place, US-Morrisville, NC 27560-8496**
Advanced Micro devices, Inc. - AMD **7171 Southwest Parkway, Mailstop B100.3, US-Austin, Texas 78735**
Hewlett-Packard Company **P.O. Box 10490, US-Palo Alto, CA 94303-0969**
Infineon Technologies AG - Neubiberg **Am Campeon 1-12, DE-85579 Neubiberg**
Sun Microsystems Inc. - Menlo Park, CA **10 Network Circle, UMPK10-146, US-Menlo Park, CA 94025**
IBM Corporation **North Castle Drive, US-Armonk, N.Y. 10504**
Intel Corporation **5200 Elam Young Parkway, US-Hillsboro, OR 97123**

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those identified above. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO (www.iso.org/patents) and IEC (http://patents.iec.ch) maintain on-line databases of patents relevant to their standards. Users are encouraged to consult the databases for the most up to date information concerning patents.

Information technology — Trusted Platform Module Library — Part 1: Architecture

1 Scope

This part of ISO/IEC 11889 defines the architectural elements of the Trusted Platform Module (TPM), a device which enables trust in computing platforms in general. Some TPM concepts are explained adequately in the context of the TPM itself. Other TPM concepts are explained in the context of how a TPM helps establish trust in a computing platform. When describing how a TPM helps establish trust in a computing platform, this part of ISO/IEC 11889 provides some guidance for platform requirements. However, the scope of ISO/IEC 11889 is limited to TPM requirements.

This part of ISO/IEC 11889 illustrates TPM security and privacy techniques in the context of a platform through the use of cryptography. It includes definitions of how different cryptographic techniques are implemented by a TPM. The scope of ISO/IEC 11889 does not include cryptographic analysis or guidance about the applicability of different algorithms for specific uses cases.

TPM requirements in this part of ISO/IEC 11889 are general, covering concepts like integrity protection, isolation and confidentially. Defining a specific strength of function or assurance level is out of scope for ISO/IEC 11889. This approach limits the guarantees provided by ISO/IEC 11889 itself, but it does allow the TPM architectural elements defined to be adapted to meet diverse implementation and platform specific needs.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO/IEC 9797-2, *Information technology -- Security techniques -- Message Authentication Codes (MACs) -- Part 2: Mechanisms using a dedicated hash-function*
- ISO/IEC 10116:2006, *Information technology — Security techniques — Modes of operation for an n-bit block cipher*
- ISO/IEC 11889-2, *Information technology — Trusted Platform Module Library — Part 2: Structures*
- ISO/IEC 11889-3, *Information technology — Trusted Platform Module Library — Part 3: Commands*
- ISO/IEC 11889-4, *Information technology — Trusted Platform Module Library — Part 4: Supporting routines*
- TCG Algorithm Registry, available at <http://www.trustedcomputinggroup.org/resources/tcg_algorithm_registry>

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1
ancestor

<object loaded in a Trusted Platform Module> Storage Key that needs to have been loaded prior to loading an object

3.2
authValue

octet string containing a value that is used for access authorization

Note 1 to entry: The value is used as a password or to derive a key for an Hash Message Authentication Code calculation.

3.3
authPolicy

digest value produced by an execution of policy commands and used for access authorization

3.4
bound

authValue of the Object is not included in the Hash Message Authentication Code authorization for the authorization session

3.5
canonical form

data structure in the format used for transport to and from the Trusted Platform Module

Note 1 to entry: See clause 3.25.

3.6
CLEAR

bit with a value of zero (0), or the action of causing a bit to have a value of zero (0)

3.7
command

discrete Trusted Platform Module function that is exposed externally and recognizable by a Trusted Platform Module's command processor as well as the values sent to the Trusted Platform Module to indicate the operation to be performed

3.8
commandCode

numeric identifier of the operation to be performed by a Trusted Platform Module

3.9
context

collection of data that provides qualifying information about a data object to differentiate it from others of the same type or to differentiate one version of a data object from another

3.10
cpHash

hash of the command code, Object names, and parameters of a command

3.11
descendant

<Storage Key> Object whose loading is conditional on the Storage Key having been previously loaded

3.12
digest

result of a hash operation

3.13
duplicate

allowing a Protected Object created by a Trusted Platform Module to be used on a different Trusted Platform Module

3.14
ECDH

Diffie-Hellman secure secret sharing process using elliptic curve operations

3.15
Ephemeral Key

key created as part of a protocol that is not used again after the protocol is complete

3.16
EmptyAuth

Empty Buffer used as an authorization value

3.17
Empty Buffer

sized array with no data, indicated by a size field of zero followed by an array containing no elements

3.18
Empty Point

Elliptic curve cryptography point with Empty Buffers for both the x and y coordinates

3.19
Empty Policy

Empty Buffer used when a policy value is required

Note 1 to entry: As a *policyValue*, an Empty Buffer will satisfy no policy

Note 2 to entry: No policy can be satisfied by an Empty Policy because an Empty Policy has zero length but a *policyDigest* is the size of a hash digest and a digest is never zero length.

3.20
Endorsement Authorization

Authorization using either *endorsementAuth* or *endorsementPolicy*

3.21
Extend
Extended

operation that replaces the current value of a digest with the hash of a buffer constructed by concatenating new data to the current value of the digest

Note 1 to entry: See clause 11.4.7

Note 2 to entry: The new data concatenated is normally a digest.

3.22
External Object

Object that can be loaded into a Trusted Platform Module without being a member of a specific hierarchy

3.23
Failure mode

mode in which the Trusted Platform Module returns TPM_RC_FAILURE in response to all commands except TPM2_GetTestResult() or TPM2_GetCapability()

3.24
import

operation that allows a Protected Object not created by a Trusted Platform Module to be incorporated into a hierarchy of the Trusted Platform Module

3.25
internal form

data structure using a layout that is specific to an implementation that might or might not be the same as the canonical form

3.26
Lockout Authorization

authorization using either *lockoutAuth* or *lockoutPolicy*

3.27
non-volatile

data that is retained even when power is removed

3.28
NULL

<pointer> context-sensitive value that is a system-defined value indicating that the pointer does not reference data

3.29
NULL

<structure identified by an algorithm identifier> TPM_ALG_NULL value indicating that no additional data is present

3.30
NULL Password
NULL Auth

authorization where the authorization value is the Empty Buffer resulting in an authorization that is a fixed sequence of 9 octets of 40 00 00 09 00 00 00 00 00_{16}

3.31
NULL Signature

signature with the TPM_ALG_NULL signature scheme that contains no data

3.32
NULL-terminated

sequence of non-zero values followed by a value containing zero

3.33
NULL Ticket

ticket structure with *tag* set to a value that is correct for the context, *hierarchy* is TPM_RH_NULL, and *digest* is an Empty Buffer

3.34
Object

key or data that has a public portion and, optionally, a sensitive portion and which is a member of a hierarchy

3.35
octet

a group of eight adjacent bits

Note 1 to entry: On most modern computers, this is the smallest addressable unit of data.

[SOURCE: ISO/IEEE 11073-30300:2004(en), 3.1.26]

3.36
orderly shutdown

when the Trusted Platform Module has completed TPM2_Shutdown() before power to the Trusted Platform Module is removed or _TPM_Init is asserted

3.37
ordinary key

key produced with a seed taken from the Trusted Platform Module random number generator

3.38
Owner Authorization

authorization using either *ownerAuth* or *ownerPolicy*

3.39
PCR

one or more platform configuration registers each containing a digest

3.40
PCR.alg

hash algorithm associated with a specific platform configuration register

3.41
PCR bank

collection of platform configuration registers identified by a hash algorithm, with each platform configuration register in the bank containing a digest computed using the bank identifier's hash algorithm

3.42
PCR.digest

digest value associated with a specific platform configuration register

3.43
Permanent Entity

Trusted Platform Module resource with an architecturally defined handle that does not change

Note 1 to entry: The value of a Permanent Entity can change

3.44
Persistent Object

Trusted Platform Module resource created by a Protected Capability that persists in Trusted Platform Module memory across power cycles and Trusted Platform Module resets

3.45
Platform Authorization

authorization using either *platformAuth* or *platformPolicy*

3.46
policyDigest

digest uniquely representing an ordered set of policy commands and operands used to determine if a policy authorizing an action has been satisfied

3.47
policySession→cpHash

policy session context value that, if not the Empty Buffer, is the *cpHash* value that the authorized command needs to have for the authorization to be valid

3.48
platform firmware

code added to the platform by its manufacturer that is needed for booting and proper platform operation

Note 1 to entry: Commonly, but not exclusively, referred to as BIOS or UEFI or SMM code

3.49
Primary Key

key derived from a Primary Seed that is associated with the hierarchy of the Primary Seed

3.50
Primary Object

Primary Key or a data blob with a sensitive area that is encrypted using a symmetric key derived from the public area of the object and a Primary Seed

3.51
private area

encrypted and integrity protected blob that contains the sensitive area of an object

3.52
Primary Seed

large random value contained within a Trusted Platform Module from which Primary Keys and Primary Objects are derived

3.53
Protected Capability

operation performed by the Trusted Platform Module on data in a Shielded Location in response to a command sent to the Trusted Platform Module

3.54
Protected Object

Object with an encrypted sensitive portion, the sensitive portion of which the Trusted Platform Module will only decrypt when it is in a Shielded Location

3.55
Random Access Memory
RAM

memory that can be accessed in any order and which has no endurance limitations

3.56
reset interval

period between two successive Trusted Platform Module Resets and the interval during which the *resetCount* is not changed

3.57
response

values returned by the Trusted Platform Module when it completes processing of a command

3.58
Resume PCR

platform configuration register with a value that is preserved over a Trusted Platform Module Resume sequence

3.59
Root of Trust

component that needs to always behave in the expected manner because its misbehavior cannot be detected

Note 1 to entry: The complete set of Roots of Trust has at least the minimum set of functions to enable a description of the platform characteristics that affect the trustworthiness of the platform.

3.60
rpHash

hash of the response code and the parameters of a response

3.61
Sealed Data Object

encrypted, user-defined, data blob that is associated with a hierarchy and loaded using TPM2_Load() or TPM2_CreatePrimary()

3.62
sensitive area

contains the confidential or secret parts of an object that need to be encrypted and integrity protected when not in a Shielded Location on a Trusted Platform Module

3.63
sequence object

transient data structure used to hold hash state that has a handle and can be context swapped

Note 1 to entry: See clause 30

3.64
session

transient Trusted Platform Module structure that maintains the state associated with a sequence of authorizations or an audit digest

3.65
SET

bit with a value of one (1), or the action of causing a bit to have a value of one (1)

3.66
Shielded Location

location on a Trusted Platform Module that contains data that is shielded from access by any entity other than the Trusted Platform Module and which can be operated on only by a Protected Capability

3.67
Shutdown(CLEAR)

abbreviated form of the command TPM2_Shutdown() with the *startupType* parameter set to TPM_SU_CLEAR

3.68
Shutdown(STATE)

abbreviated form of the command TPM2_Shutdown() with the *startupType* parameter set to TPM_SU_STATE

3.69
sizeof(x)

operator that returns the number of octets in the operand 'x'

3.70
Startup(CLEAR)

abbreviated form of the command TPM2_Startup() with the *startupType* parameter set to TPM_SU_CLEAR

3.71
Startup(STATE)

abbreviated form of the command TPM2_Startup with the *startupType* parameter set to TPM_SU_STATE

3.72
Storage Key

key that can have descendant keys

3.73
Temporary Object

object that become unusable after a Trusted Platform Module Reset and that cannot be converted into a Persistent Object

3.74
temporary resource

data object created during the execution of a command that does not persist in TPM memory after the command completes

3.75
TPM_GENERATED_VALUE

32-bit number FF 54 43 47_{16} used to tag structures that are generated by a Trusted Platform Module

3.76
TPM Reset

resetting of all Trusted Platform Module internal state to default values due to Startup(CLEAR)

3.77
TPM Resource Manager
TRM

software executing on a system with a Trusted Platform Module that ensures that the resources necessary to execute Trusted Platform Module commands are present in the Trusted Platform Module

3.78
TPM Restart

Startup(CLEAR) that initializes all platform configuration registers but preserves most other Trusted Platform Module state from the previous Shutdown(STATE)

3.79
TPM Resume

Startup(STATE) that initializes some platform configuration registers but preserves most Trusted Platform Module state from the previous Shutdown(STATE)

3.80
transient resource

object or session that can be explicitly loaded and unloaded from Trusted Platform Module memory by the Trusted Platform Module Resource Manager and is cleared from Trusted Platform Module memory when the Trusted Platform Module is initialized (TPM2_Startup())

3.81
Trusted Platform Module
TPM

implementation of ISO/IEC 11889

3.82
user-installable software

any software that can be installed on a platform other than platform firmware

3.83
volatile data

data that is lost when power is removed

3.84
Zero Digest

non-zero-length digest with all octets set to zero

4 Symbols and Abbreviated Terms

4.1 Symbols

For the purposes of this document, the following symbol definitions apply unless the text is in the `Courier` font.

A || B	concatenation of B to A
|x|	the length of x in bits
⌈x⌉	the smallest integer not less than x
⌊x⌋	the largest integer not greater than x
A ≔ B	assignment of the results of the expression on the right (B) to the parameter on the left
A = B	equivalence (A is the same as B)
{ A }	an optional element
A ⊕ B	bitwise exclusive OR of elements
A & B	logical AND of elements
A | B	the logical OR of elements
{A | B}	selection of elements
{A : B}	an inclusive range of elements between A and B
<A, B, … >	an ordered list of elements (a tuple)
0…0	a context-sensitive number of octets of zero
F()	denotes a function **F**
F(*p* == *x*)	denotes a function or TPM command **F** with parameter *p* set to value *x*
H()	denotes the hash function
[*n*]*P*	multiplication of point *P* by the integer value *n*
A • B	multiplication of two integer values A and B
A→B	denotes a reference to element B within structure A
A mod B	A modulus B

Text in the **`Courier`** font indicates code written according to the C language standard.

4.2 Abbreviations

For the purposes of ISO/IEC 11889, the following abbreviations apply.

Abbreviation	Description
TPM	Prefix for an indication passed from the system interface of the TPM to a Protected Capability defined in ISO/IEC 11889
AK	Attestation Key
BIOS	Basic Input/Output System
CA	Certificate Authority
CFB	Cipher Feedback mode
CPU	Central Processing Unit
CRTM	Core Root of Trust for Measurement
CTR	Counter mode
D-RTM	dynamic RTM
DA	dictionary attack
DoS	Denial of Service
DRBG	Deterministic Random Bit Generator
DSA	Digital Signature Algorithm
EA	Enhanced Authorization
EAL	evaluated assurance level
ECC	Elliptic Curve Cryptography
ECDAA	ECC-based Direct Anonymous Attestation
ECDH	Elliptic Curve Diffie-Hellman
EK	Endorsement Key
EPS	Endorsement Primary Seed
FIPS	Federal Information Processing Standard
FUM	Field Upgrade mode
GPIO	General Purpose I/O
HMAC	Hash Message Authentication Code
I/O	Input/Output
IV	Initialization Vector
KDF	key derivation function
LPC	Low Pin Count
LSb	Least Significant bit
LSO	Least Significant Octet
MSb	Most Significant bit
MSO	Most Significant Octet
NIST	National Institute of Standards and Technology
NP	new parent

Abbreviation	Description
NV	non-volatile
NVRAM	Non-Volatile Random Access Memory
OAEP	Optimal Asymmetric Encryption Padding
OEM	Original Equipment Manufacturer
OIAP	Object-Independent Authorization Protocol
OID	Object Identifier in ASN.1 format
OSAP	Object-Specific Authorization Protocol
PCR	platform configuration register(s)
POST	Power on Self-Test
PP	Physical Presence
PPS	Platform Primary Seed
PRF	pseudo-random function
PRNG	pseudo-random number generator
PSS	Probabilistic Signature Scheme
QN	Qualified Name
RNG	random number generator
RSA	Rivest, Shamir and Adleman
RTM	Root of Trust for Measurement
RTR	Root of Trust for Reporting
RTS	Root of Trust for Storage
S-RTM	static RTM
SHA	Secure Hash Algorithm
SMM	System Management mode
SPS	Storage Primary Seed
SRK	Storage Root Key
TBB	trusted building block
TCB	trusted computing base
TCG	Trusted Computing Group
TPM	Trusted Platform Module
TPM2_	Prefix for a command defined in ISO/IEC 11889
TSS	TCG Software Stack
UEFI	Unified Extensible Firmware Interface

5 Conventions

5.1 Bit and Octet Numbering and Order

An integer value is considered to be an array of one or more octets. The octet at offset zero within the array is the most significant octet (MSO) of the integer. Bit number 0 of that integer is its least significant bit and the is the least significant bit in the last octet in the array.

EXAMPLE A 32-bit integer is an array of four octets; the MSO is at offset [0], and the most significant bit is bit number 31. Bit zero of this 32-bit integer is the least significant bit in the octet at offset [3] in the array.

NOTE 1 Array indexing is zero-based.

NOTE 2 This definition does not match the “network bit order” used in many IETF documents, such as RFC 4034. In those documents, the most significant bit of a datum has the lowest bit number. It is conventional practice to send that bit first when using a serial network protocol, and the bits are numbered in the order in which they are sent. ISO/IEC 11889 numbers bits according to the power of two to which they correspond within a datum. This numbering corresponds to the normal convention for bit numbering in hardware registers that hold integer values rather than fixed-point numbers.

The first listed member of a structure is at the lowest offset within the structure and the last listed member is at the highest offset within the structure.

For a character string (letters delimited by “”), the first character of the string contains the MSO.

5.2 Sized Buffer References

ISO/IEC 11889 makes extensive use of a data structure called a *sized buffer*. A sized buffer has a size field followed by an array of octets equal in number to the value in the size field.

The structure will have an identifying name. When ISO/IEC 11889 references the size field of the structure, the structure name is followed by “.size” (a period followed by the word “size”). When ISO/IEC 11889 references the octet array of the structure, the structure name is followed by “.buffer” (a period followed by the word “buffer”).

5.3 Numbers

Numbers are decimal unless a different radix is indicated.

Unless the number appears in a table intended to be machine readable, the radix is a subscript following the digits of the number. Only radix values of 2 and 16 are used in ISO/IEC 11889.

Radix 16 (hexadecimal) numbers have a space separator between groups of two hexadecimal digits.

EXAMPLE 1 $40\ FF\ 12\ 34_{16}$

Radix 2 (binary) numbers use a space separator between groups of four binary digits.

EXAMPLE 2 $0100\ 1110\ 0001_{2}$

For numbers using a binary radix, the number of digits indicates the number of bits in the representation.

EXAMPLE 3 20_{16} is a hexadecimal number that contains exactly 8 bits and has a decimal value of 32.

EXAMPLE 4 $10\ 0000_{2}$ is a binary number that contains exactly 6 bits and has a decimal value of 32.

EXAMPLE 5 $0\ 20_{16}$ is a hexadecimal number that contains exactly 12 bits and has a decimal value of 32.

A number in a machine-readable table may use the “0x” prefix to denote a base 16 number. In this format, the number of digits is not always indicative of the number of bits in the representation.

EXAMPLE 6 0x20 is a hexadecimal number with a value of 32, and the number of bits is determined by the context.

5.4 KDF Label Parameters

Specific strings representing octets of data are used normatively in this International Standard as the label parameter for key derivation functions. The octets are represented as quoted ASCII characters and exhaustively listed in Table 1 with their length and values to disambiguate them from regular words used in quotes.

Table 1 — KDF Label Parameters

Label	*Sequence size*	*Octet data*
“ATH”	4	$41\ 54\ 48\ 00_{16}$
“CFB”	4	$43\ 46\ 42\ 00_{16}$
“DUPLICATE”	10	$44\ 55\ 50\ 4C\ 49\ 43\ 41\ 54\ 45\ 00_{16}$
“IDENTITY”	9	$49\ 44\ 45\ 4E\ 54\ 49\ 54\ 59\ 00_{16}$
“INTEGRITY”	10	$49\ 4E\ 54\ 45\ 47\ 52\ 49\ 54\ 59\ 00_{16}$
“OBFUSCATE”	10	$4F\ 42\ 46\ 55\ 53\ 43\ 41\ 54\ 45\ 00_{16}$
“SECRET”	7	$53\ 45\ 43\ 52\ 45\ 54\ 00_{16}$
“STORAGE”	8	$53\ 54\ 4F\ 52\ 41\ 47\ 45\ 00_{16}$
“XOR”	4	$58\ 4F\ 52\ 00_{16}$
NOTE	See clause 11.4.9.3 for justification for the terminating octet of 00_{16}.	

6 ISO/IEC 11889 Organization

ISO/IEC 11889 defines the Trusted Platform Module (TPM), a device that enables trust in computing platforms in general. It is broken into parts to make the role of each part clear. All parts are required in order to constitute a complete standard.

For a complete definition of all requirements necessary to build a TPM, the designer will need to use the appropriate platform-specific specification to understand all of the requirements for a TPM in a specific application or make appropriate choices as an implementer.

Those wishing to create a TPM need to be aware that ISO/IEC 11889 does not provide a complete picture of the options and commands necessary to implement a TPM. To implement a TPM the designer needs to refer to the relevant platform-specific specification to understand the options and settings required for a TPM in a specific type of platform or make appropriate choices as an implementer.

EXAMPLE The number of platform configuration registers and their attributes are not defined in ISO/IEC 11889. Those values would be specified by a platform specific specification or alternatively determined by an implementer.

ISO/IEC 11889 contains four parts, as follows:

Part 1: Architecture

This part of ISO/IEC 11889 contains a narrative description of the properties, functions, and methods of a TPM. Unless otherwise noted, this narrative description is *informative*. This part of ISO/IEC 11889 contains descriptions of some of the data manipulation routines that are used by ISO/IEC 11889. The normative behavior for these routines is in C code in ISO/IEC 11889-3 and ISO/IEC 11889-4. Algorithms and processes specified in this part of ISO/IEC 11889 may be made normative by reference from ISO/IEC 11889-2, ISO/IEC 11889-3 or ISO/IEC 11889-4.

Part 2: Structures

ISO/IEC 11889-2 contains a *normative* description of the constants, data types, structures, and unions for the TPM interface. Unless otherwise noted: (1) all tables and C code in ISO/IEC 11889-2 are normative, and (2) normative content in ISO/IEC 11889-2 takes precedence over any other part of ISO/IEC 11889.

Part 3: Commands

ISO/IEC 11889-3 contains: (1) a *normative* description of commands, (2) tables describing the command and response formats, and (3) C code that illustrates the actions performed by a TPM. Within ISO/IEC 11889-3, command and response tables have the highest precedence, followed by the C code, followed by the narrative description of the command. ISO/IEC 11889-3 is subordinate to ISO/IEC 11889-2.

A TPM need not be implemented using the C code in ISO/IEC 11889-3. However, any implementation should provide equivalent or, in most cases, identical results as observed at the TPM interface or demonstrated through evaluation.

Part 4: Supporting routines

ISO/IEC 11889-4 presents C code that describes the algorithms and methods used by the command code in ISO/IEC 11889-3. The code in ISO/IEC 11889-4 augments ISO/IEC 11889-2 and ISO/IEC 11889-3 to provide a complete description of a TPM, including the supporting framework for the code that performs the command actions.

Any ISO/IEC 11889-4 code may be replaced by code that provides similar results when interfacing to the action code in ISO/IEC 11889-3. The behavior of ISO/IEC 11889-4 code not included in an annex is

normative, as observed at the interfaces with ISO/IEC 11889-3 code. Code in an annex is provided for completeness, that is, to allow a full implementation of ISO/IEC 11889.

NOTE ISO/IEC 11889 does not provide code for lower-level cryptographic algorithms and use of external libraries is required for a complete implementation.

Extensive modification of the code provided in ISO/IEC 11889-4 annexes is expected for any TPM implementation. Modifications are required in order to interface the TPM code with actual TPM hardware rather than the simulation framework provided. In addition, modifications of the code in ISO/IEC 11889-4 annexes would be necessary in order to meet the needs of applicable evaluation regimes.

7 Compliance

Unless the ISO/IEC 11889-3 general description of a command indicates that the command is mandatory, a compliant TPM need not implement the command. However, if implemented, the command is required to have the behavior defined in ISO/IEC 11889- 3. A platform-specific specification will indicate the commands from ISO/IEC 11889 that are required to be implemented in order to be compliant with that platform-specific specification.

The code in ISO/IEC 11889 is the reference implementation that describes required TPM behavior as observed from the TPM interface. The C-code may be reorganized or rewritten in any desired implementation language and remain compatible with ISO/IEC 11889 as long as the observable behavior is equivalent.

Even though the code in the reference implementation has undergone extensive testing, it is likely that some errors exist and one or more of those errors could lead to a TPM failure or exploit. Regardless of any other statement about normative behavior, one should not assume that a TPM exploit or failure is an intended behavior. It is not necessary to reproduce such a behavior in order to be compliant with ISO/IEC 11889.

NOTE 1 Please report bugs in the reference code to the TCG (admin@trustedcomputinggroup.org) so that the reference code may be brought into compliance with ISO/IEC 11889.

The response codes in ISO/IEC 11889 are normative. An implementation performing a check prescribed by ISO/IEC 11889 is required to return the indicated error if the check fails. The order in which checks are performed is not normative. This means that a command with multiple errors could return different response codes on different TPMs. However, the response code returned is required to be the normative response code used to indicate the specific failure.

Capacities and algorithms of a TPM implementation may vary from the reference implementation; in this case, the same error would not occur in the same situation. However, these differences should not cause a different response code to be returned when the nature of the error is the same as in the reference implementation. ISO/IEC 11889-4 contains major subsystems that may change for each instance of a TPM. When the subsystem is rewritten, an equivalent interface should be provided, and the errors returned are required to match those of the reference implementation.

EXAMPLE 1 An example where a TPM implementation might not return the same error as the reference implementation is one with more memory which might be able to satisfy a request where the reference implementation would have returned an error.

EXAMPLE 2 Examples of TPM capacities are the amount of nonvolatile storage space, the number of keys that maybe simultaneously loaded, the number of PCR, the number of PCR banks supported, etc.

EXAMPLE 3 The NV subsystem of the reference implementation is not representative of the actual implementation of most physical NV implementations but is a crude analog.

NOTE 2 A constraint on the design of the TPM was the process of compliance-testing of different TPM implementations. If a TPM implementation has modularity similar to the reference implementation, then TPM tests that assume a modular design will be able to produce reliable test results on each TPM implementation.

The reference implementation uses static and stack-based allocation of resources and does not do allocations on a heap. However, a TPM implementation may use heap-based memory management in which case some error conditions and codes will differ. These differences are limited, and the allowed response codes and error conditions are defined in 39.3.

8 Changes from Previous Versions

This version of ISO/IEC 11889 introduces these additional features to the TPM family:

- Definition of an interface that allows variability of underlying cryptographic algorithms – ISO/IEC 11889 (first edition) is constrained by its data structures to using RSA and SHA1. The ISO/IEC 11889 structure and interface defines support for a wide range of hash and asymmetric algorithms along with limited support for use of various block, symmetric ciphers. Of particular note is the addition of support for the elliptic curve (ECC) family of asymmetric algorithms.
- Unification of authorization methods – ISO/IEC 11889 (first edition) has different schemes to authorize the use, delegated use, and migration of objects. ISO/IEC 11889 provides a uniform framework for using authorization capabilities so they may be combined in unique ways to provide more flexibility.
- Expansion of authorization methods – ISO/IEC 11889 allows authorization with clear-text passwords and Hash Message Authentication Code (HMAC). It also allows construction of an arbitrarily complex authorization policy for an object using multiple authorization qualifiers.
- Dedicated BIOS support – ISO/IEC 11889 adds a Storage hierarchy controlled by platform firmware, letting the OEM benefit from the cryptographic capabilities of the TPM regardless of the support provided to the OS.
- Simplified control model – ISO/IEC 11889 needs no special provisioning process to be useful to applications. Although objects on which the TPM operates may have limitations, all commands are available all the time. This lets application developers rely on TPM capabilities being available whenever a TPM is present.

A TPM compatible with ISO/IEC 11889 need not be compatible with ISO/IEC 11889 (first edition).

ISO/IEC 11889 defines the operations a TPM performs and the structures used for communication between the TPM and the host system. It does not define an electrical interface to the TPM, nor does it specify which subset of ISO/IEC 11889 commands and resources are required for a specific platform. Please refer to platform-specific specifications for this information.

9 Trusted Platforms

9.1 Trust

In the context of Trusted Computing Group (TCG) specifications, “trust” is meant to convey an expectation of behavior. However, predictable behavior does not necessarily constitute behavior that is worthy of trust.

EXAMPLE We expect that a bank will behave like a bank, and we expect that a thief will behave like a thief.

In order to determine the expected behavior of a platform, it is necessary to determine its identity as it relates to the platform behavior. Physically different platforms may have identical behavior. If they are constructed of components (hardware and software) that have identical behavior, then their trust properties should be the same.

The TCG defines schemes for establishing trust in a platform that are based on identifying its hardware and software components. The Trusted Platform Module (TPM) provides methods for collecting and reporting these identities. A TPM used in a computer system reports on the hardware and software in a way that allows determination of expected behavior and, from that expectation, establishment of trust.

9.2 Trust Concepts

9.2.1 Trusted Building Block

A trusted building block (TBB) is a component or collection of components required to instantiate a Root of Trust. Typically platform-specific, a TBB is part of a Root of Trust that does not have Shielded Locations.

EXAMPLE One example of a TBB is the combination of the CRTM, the connection between CRTM storage and a motherboard, the path between CRTM storage and the CPU, the connection between the TPM and a motherboard, and the path between the CPU and the TPM. This combination comprises the Root of Trust for Reporting (RTR).

A TBB is a component that is expected to behave in a way that does not compromise the goals of trusted platforms.

9.2.2 Trusted Computing Base

A trusted computing base (TCB) is the collection of system resources (hardware and software) that is responsible for maintaining the security policy of the system. An important attribute of a TCB is that it be able to prevent itself from being compromised by any hardware or software that is not part of the TCB.

The TPM is not the trusted computing base of a system. Rather, a TPM is a component that allows an independent entity to determine if the TCB has been compromised. In some uses, the TPM can help prevent the system from starting if the TCB cannot be properly instantiated.

9.2.3 Trust Boundaries

The combination of TBB and Roots of Trust form a trust boundary, within which measurement, storage, and reporting may be accomplished for a minimal configuration. In systems that are more complex, it may be necessary for the CRTM to establish trust in other code, by making measurements of that other code and recording the measurement in a PCR. If the CRTM transfers control to that other code regardless of the measurement, then the trust boundary is expanded. If the CRTM will not run that code unless its

measurement is the expected value, the trust boundary remains the same because the measured code is an expected extension of the CRTM.

9.2.4 Transitive Trust

Transitive trust is a process whereby the Roots of Trust establish the trustworthiness of an executable function, and trust in that function is then used to establish the trustworthiness of the next executable function.

Transitive trust may be accomplished either by: (1) knowing that a function enforces a trust policy before it allows a subsequent function to take control of the TCB, or (2) using measurements of subsequent functions so that an independent evaluation may establish the trust. The TPM may support either of these methods.

9.2.5 Trust Authority

When the RTM begins to execute the CRTM, the entity that may vouch for the correctness of the TBB is the entity that created the TBB. For typical systems, this is the platform manufacturer. In other words, the manufacturer is the authority on what constitutes a valid TBB, and its reputation is what allows someone to trust a given TBB.

As the system transitions to code outside the CRTM, the transitive trust chain is maintained by measurement of that code. If execution of that code is conditional on its measurement, then the authority for that code remains unchanged. That is, if the platform manufacturer's CRTM does not run code outside the CRTM unless that code has a specific measured value, then the platform manufacturer remains the trust authority regardless of who provided that code.

In modern architectures, where firmware and software components come from many different suppliers, it is often not feasible for platform manufacturers to know the signers of all code that runs on a platform. Therefore, they may not remain the authority on platform state for very long. The measurements recorded in the RTS then determine the chain of authority for the current system state.

Two different methods allow evaluation of the trust authority for a platform.

1) Code is measured (hashed) and its value is recorded in the RTS. If the code is run regardless of its measurement, then the authority for the trust is the digest of the code reported by the RTR. That is, the measurements speak for themselves, and the verifier needs either to have knowledge of the measurements that constitute trustworthy code or knowledge of the measurements that indicates malicious or vulnerable code.
2) Code is signed so that the identity of the authority for the code is known. If this identity is recorded in the RTS, the evaluation can be changed. Instead of being based on knowing the digest of the code, it can be based on identities of the signers of the code.

Because trusted sources of code may sometimes produce code with security vulnerabilities, support for revocation is often required. To allow revocation of specific code modules, it is often necessary to use a hybrid solution where both authorities and details are recorded. This simplifies the process of determining whether a module from a specific vendor has been revoked.

NOTE If the code is measured (hashed) and not signed, it is harder to know if a specific measurement is valid unless there is a centralized database of all known digests of revoked code. When the identity of the authority is known, one can contact the vendor to determine if it has revoked code with a given hash.

9.3 Trusted Platform Module

A TPM is a system component that has state that is separate from the system on which it reports (the host system). The only interaction between the TPM and the host system is through the interface defined in ISO/IEC 11889.

TPMs are implemented on physical resources, either directly or indirectly. A TPM may be constructed using physical resources that are permanently and exclusively dedicated to the TPM, and/or using physical resources that are temporarily assigned to the TPM. All of a TPM's physical resources may be located within the same physical boundary, or different physical resources may be within different physical boundaries.

Some TPMs are implemented as single-chip components that are attached to systems (typically, a PC) using a low-performance interface. The TPM component has a processor, RAM, ROM, and Flash memory. The only interaction with the TPM is through the LPC. The host system cannot directly change the values in TPM memory other than through the I/O buffer that is part of the interface.

EXAMPLE An example of low performance interface is the Low Pin Count or LPC.

Another reasonable implementation of a TPM is to have the code run on the host processor while the processor is in a special execution mode. For these TPMs, parts of system memory are partitioned by hardware so that the memory used by the TPM is not accessible by the host processor unless it is in this special mode. Further, when the host processor switches modes, it always begins execution at specific entry points. This version of a TPM would have many of the same attributes as the stand-alone component in that the only way for the host to cause the TPM to modify its internal state is with well-defined interfaces. There are several different schemes for achieving this mode switching including System Management Mode, Trust Zone™, and processor virtualization.

Definition of the interaction between the host and the TPM is the primary objective of ISO/IEC 11889. Prescribed commands instruct the TPM to perform prescribed actions on data held with the TPM. A primary purpose of these commands is to allow determination of the trust state of a platform. The ability of a TPM to accomplish its objective depends on the proper implementation of Roots of Trust.

9.4 Roots of Trust

9.4.1 Introduction

TCG-defined methods rely on Roots of Trust. These are system elements that must be trusted because misbehavior is not detectable. The set of roots required by the TCG provides the minimum functionality necessary to describe characteristics that affect a platform's trustworthiness.

While it is not possible to determine if a Root of Trust is behaving properly, it is possible to know how roots are implemented. Manufacturers can provide assurances that the root has been implemented in a way that renders it trustworthy.

EXAMPLE A certificate may identify the manufacturer and evaluated assurance level (EAL) of a TPM.

A certification can provide confidence in the Roots of Trust implemented in the TPM. In addition, a certificate from a platform manufacturer may provide assurance that the TPM was properly installed on a machine that is compliant with TCG specifications so that the Root of Trust provided by the platform may be trusted. See 9.5.2 for more information on certification.

The TCG requires three Roots of Trust in a trusted platform:

- Root of Trust for Measurement (RTM),

- Root of Trust for Storage (RTS), and
- Root of Trust for Reporting (RTR).

Trust in the Roots of Trust can be achieved through a variety of means including technical evaluation by competent experts.

9.4.2 Root of Trust for Measurement (RTM)

The RTM sends integrity-relevant information (measurements) to the RTS. Typically, the RTM is the CPU controlled by the Core Root of Trust for Measurement (CRTM). The CRTM is the first set of instructions executed when a new chain of trust is established. When a system is reset, the CPU begins executing the CRTM. The CRTM then sends values that indicate its identity to the RTS. This establishes the starting point for a chain of trust. See 9.5.5 for a more detailed description of integrity measurement.

9.4.3 Root of Trust for Storage (RTS)

The TPM memory is shielded from access by any entity other than the TPM. Because the TPM can be trusted to prevent inappropriate access to its memory, the TPM can act as an RTS.

Some of the information in TPM memory locations is not sensitive and the TPM does not protect it from disclosure. Other information is sensitive and the TPM does not allow access to the information without proper authority.

EXAMPLE 1 An example of non-sensitive data is the current contents of a platform configuration register (PCR) containing a digest.

EXAMPLE 2 An example of sensitive data in a Shielded Location is the private part of an asymmetric key.

Sometimes, the TPM uses the contents of one Shielded Location to gate access to another Shielded Location.

EXAMPLE 3 Access to (use of) a private key for signing might be conditioned on PCR having specific values.

9.4.4 Root of Trust for Reporting (RTR)

9.4.4.1 Description

The RTR reports on the contents of the RTS. An RTR report is typically a digitally signed digest of the contents of selected values within a TPM.

NOTE Not all Shielded Locations are directly accessible.

EXAMPLE 1 The values of the private part of keys and authorizations are in Shielded Locations on which the TPM will not report.The values on which the RTR reports typically are

- evidence of a platform configuration in PCR,
- audit logs, and
- key properties.

EXAMPLE 2 An example of a command providing evidence of platform configuration is TPM2_Quote().

EXAMPLE 3 An example of a command providing audit logs is TPM2_GetCommandAuditDIgest ().

EXAMPLE 4 An example of a command providing key properties TPM2_Certify().

The interaction between the RTR and RTS is critical. The design and implementation of this interaction should mitigate tampering that would prevent accurate reporting by the RTR. An instantiation of the RTS and RTR will

- be resistant to all forms of software attack and to the forms of physical attack implied by the TPM's Protection Profile, and
- supply an accurate digest of all sequences of presented integrity metrics.

9.4.4.2 Identity of the RTR

The TPM contains cryptographically verifiable identities for the RTR. The identity is in the form of asymmetric aliases (Endorsement Keys or EKs) derived from a common seed. Each seed value and its aliases should be statistically unique to a TPM. That is, the probability of two TPMs having the same EK should be insignificant. (See clause 14 for more information about primary seeds.)

The seed may be used to generate multiple asymmetric keys, all of which would represent the same TPM and RTR.

9.4.4.3 RTR Binding to a Platform

The TPM reports on the state of the platform by quoting the PCR values. For assurance that these PCR values accurately reflect that state, it is necessary to establish the binding between the RTR and the platform. A Platform Certificate can provide proof of this binding. The Platform Certificate is assurance from the certifying authority of the physical binding between the platform (the RTM) and the RTR.

9.4.4.4 Platform Identity and Privacy Considerations

The uniqueness of an EK and its cryptographic verifiability raises the issue of whether direct use of that identity could result in aggregation of activity logs. Analysis of the aggregated activity could reveal personal information that a user of a platform would not otherwise approve for distribution to the aggregators.

To counter undesired aggregation, TCG encourages the use of domain-specific signing keys and restrictions on the use of an EK. The Privacy Administrator controls use of an EK, including the process of binding another key to the EK.

NOTE Privacy Administrator's control of the EK differs from Owner control of the RTS providing separation of the security and identity uses of the TPM.

Unless the EK is certified by a trusted entity, its trust and privacy properties are no different from any other asymmetric key that can be generated by pure software methods. Therefore, by itself, the public portion of the EK is not privacy sensitive.

9.5 Basic Trusted Platform Features

9.5.1 Introduction

At a minimum, a trusted platform provides the three Roots of Trust described previously. All three roots use certification and attestation to provide evidence of the accuracy of information. A trusted platform will also offer Protected Locations (see 9.5.4) for the keys and data objects entrusted to it. Finally, a trusted platform may provide integrity measurement to ensure the trustworthiness of a platform by logging changes to platform state; this is done by recording logged entries in PCR for later validation as being correct and unaltered. These basic TPM concepts are now described in detail.

9.5.2 Certification

The nominal method of establishing trust in a key is with a certificate indicating that the processes used for of creating and protecting the key meets necessary security criteria. A certificate may be provided by shipping the TPM with an embedded key (that is, an Endorsement Key) along with a Certificate of Authenticity for the EK. The EK and its certificate may be used to associate credentials (certificates) with other TPM keys; this process is specified in 9.5.3.3. When a certified key has attributes that let it sign TPM-created data, it may attest to the TPM-resident record of platform characteristics that affect the integrity (trustworthiness) of a platform.

NOTE The EK does not have to be installed when the TPM is shipped. At the factory, an EK could be generated from the Endorsement Seed and a Certificate of Authenticity created for that EK. The EK does not have to be permanently installed in the TPM. When the TPM is in possession of a customer, the customer could, at their discretion, have the TPM use the Endorsement Seed and recreate the EK for which they have a Certificate of Authenticity.

9.5.3 Attestation and Authentication

9.5.3.1 Types of Attestation

Trusted platforms employ a hierarchy of attestations:

1) An external entity attests to a TPM in order to vouch that the TPM is genuine and complies with ISO/IEC 11889. This attestation takes the form of an asymmetric key embedded in a genuine TPM, plus a credential that vouches for the public key of that pair.

   NOTE 1 A credential that is used to vouch for the embedded asymmetric key is commonly called an "Endorsement Certificate."

2) An external entity attests to a platform in order to vouch that the platform contains a Root-of-Trust-for-Measurement, a genuine TPM, plus a trusted path between the RTM and the TPM. This attestation takes the form of a credential that vouches for information including the public key of the asymmetric key pair in the TPM.

   NOTE 2 A credential used to vouch for the platform is commonly called a "Platform Certificate."

3) An external entity called an “Attestation CA” attests to an asymmetric key pair in a TPM in order to vouch that a key is protected by an unidentified but genuine TPM, and has particular properties. This attestation takes the form of a credential that vouches for information including the public key of the key pair. An Attestation CA typically relies upon attestations of type 1 and 2 in order to produce attestation of type 3.

   NOTE 3 The credential created by the CA is commonly called an "Attestation Key Certificate."

4) A trusted platform attests to an asymmetric key pair in order to vouch that a key pair is protected by a genuine but unidentified TPM and has particular properties. This attestation takes the form of a signature signed by the platform’s TPM over information that describes the key pair, using an attestation-key protected by the TPM, plus attestation of type 3 that vouches for that attestation-key.

   NOTE 4 This type of attestation is done using TPM2_Certify().

5) A trusted platform attests to a measurement in order to vouch that a particular software/firmware state exists in a platform. This attestation takes the form of a signature over a software/firmware measurement in a PCR using an attestation-key protected by the TPM, plus attestation of type 3 or 4 for that attestation-key.

   NOTE 5 This is type of attestation is commonly called a "quote" and is done with TPM2_Quote().

6) An external entity attests to a software/firmware measurement in order to vouch for particular software/firmware. This attestation takes the form of a credential that vouches for information including the value of a measurement and the state it represents.

NOTE 6 This is commonly called "third-party certification."

Attestation of types 3 and 4 entail the use of a key to sign the contents of Shielded Locations. An Attestation Key (AK) is a particular type of signing key that has a restriction on its use, in order to prevent forgery (the signing of external data that has the same format as genuine attestation data). The restriction is that an AK may be used only to sign a digest that the TPM has created. If an AK is known to be protected by a TPM (by virtue of attestation of type 3 or 4), it may be relied on to report accurately on Shielded Location content, and not sign externally provided data that appears to be valid and TPM-produced but is not.

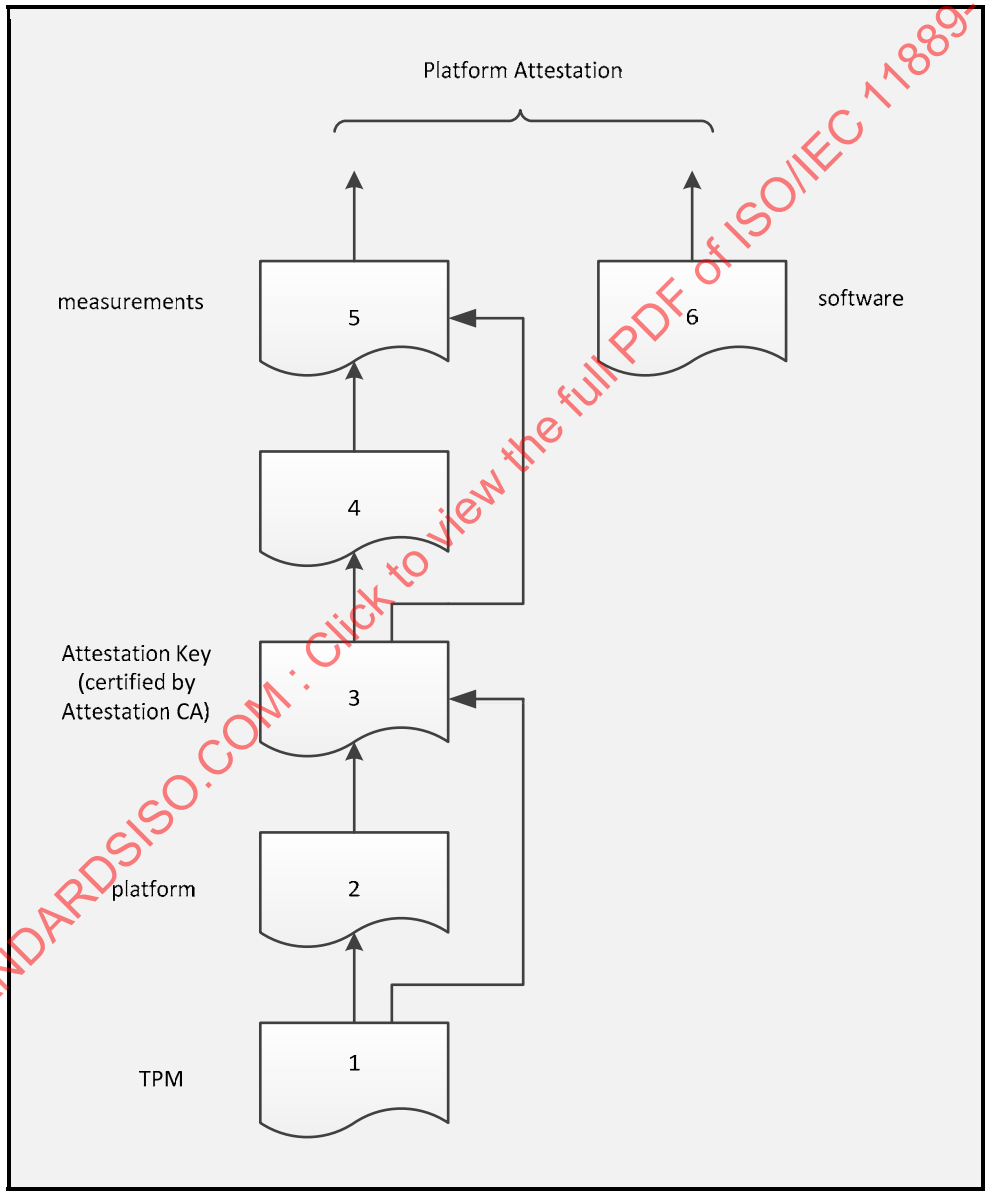


Figure 1 — Attestation Hierarchy

9.5.3.2 Attestation Keys

When the TPM creates a message to sign from internal TPM state, a special value (TPM_GENERATED_VALUE) is used as the message header. A TPM-generated message always begins with this value.

EXAMPLE 1 TPM2_Quote() is an example of a command where TPM creates a message to sign from internal TPM state.

When the TPM digests an externally provided message, it checks the first few octets of the message to ensure that they do not have the same value as TPM_GENERATED_VALUE. When the digest is complete, the TPM produces a ticket that indicates the message did not start with TPM_GENERATED_VALUE. When an AK is used to sign the digest, the caller provides the ticket so that the TPM can determine that the message used to create the digest was not a possible forgery of TPM attestation data.

NOTE The digest in the ticket must match the digest being presented to the AK for signing.

EXAMPLE 2 If an attacker produced a message block that was identical to a TPM-generated quote, that message block would start with TPM_GENERATED_VALUE to indicate that it is a proper TPM quote. When the TPM performs a digest of this block, it notes that the first octets are the same as TPM_GENERATED_VALUE. It will not generate the ticket indicating that the message is safe to sign, so an AK cannot be used to sign this digest. Similarly, an entity checking an attestation made by an AK can check that the message signed begins with TPM_GENERATED_VALUE in order to verify the message is indeed a TPM-generated quote.

Values signed by an AK may be assured to reflect TPM state, but AKs may also be used for general signing purposes.

An AK does not have much value to a remote challenger if the AK cannot be associated with the platform that it represents. This association is made using the identity certification process.

9.5.3.3 Attestation Key Identity Certification

Any TPM user that can create a key on a TPM can create a restricted-use signing key. The key creator may then ask a third party to provide a certificate for it. The third party may request that the caller provide some evidence that the key being certified is a TPM-resident key.

EXAMPLE 1 An example of an entity that might provide a certificate for a key is an attestation Certificate Authority (CA).

Evidence of TPM residency may be provided using a previously generated certificate for another key on the same TPM. An EK or Platform Certificate may provide this evidence.

NOTE 1 There is no requirement that certificates come only from a attestation CA. The method described above is an example of a scheme that could be used when privacy is required.

If a certified key may sign, it may be used to certify that some other object is resident on the same TPM. This allows the new AK to be linked to a certified key. A CA may use the certification from the TPM to produce a traditional certificate for the new key.

If the certified key is a decryption key and may not sign, then an alternative method is used to allow the new key or data object to be reliably certified. For this alternative certification, the identity of the Object to be certified and a certificate for the decryption key are provided to the CA. From the certificate, the CA determines the public key for the decryption key. The CA then produces a conditional certificate for the Object to be certified.

EXAMPLE 2 An EK is an example of a decryption key that may not sign.

The certificate is made conditional by performing some operation on the credential to produce a value that is required to be known before the credential can be used. This process produces a credential qualifier that is given to the TPM containing both the certified decryption key and the key to be certified.

EXAMPLE 3 Symmetrically encrypting a credential is an example of an operation to make its use conditional.

NOTE 2 A common credential qualifier would be a symmetric key that was used to encrypt the credential. Another option for the credential qualifier would be for it to be all or a portion of the signature of the certificate. Other options are possible.

The credential qualifier is protected using methods that are dependent on the type of the certified decryption key. The general method is specified in clause 24. Additional methods appropriate to RSA keys are specified in A.9.4 and additional methods appropriate to ECC key are in B.5.4. The protection process produces an encrypted blob, an HMAC over the blob, and a secret value that can only be recovered by the certified decryption key.

TPM2_ActivateCredential() is used to access the credential qualifier. The TPM recovers the secret value and uses it to generate the keys necessary to decrypt and validate the HMAC and encrypted blob. If the credential qualifier is recovered successfully, and the key being certified by the credential is loaded on the TPM, then the contents of the credential qualifier are returned to the caller. They may then use that value to complete the key certification.

NOTE 3 The protection process used for credential qualifiers is almost identical to the process used for key import. In order to make sure that there is no misuse of the encrypted structures, an application-specific value is used in the key recovery process. In the case of a credential qualifier, the label “IDENTITY” is used in the KDF that generates the keys (symmetric and HMAC) from the seed value. (See clause 5.4 for the definition of the label.)

TPM2_ActivateCredential() associates a credential with any object. The choice of attributes for an Object to be certified is at the discretion of the CA. Because a unique identifier for the Object is included in the integrity hash, the TPM enforces the credential’s accessibility only if the Object matches the criteria set by the CA as expressed in the object identifier.

9.5.4 Protected Location

When the sensitive portion of an object is not held in a Shielded Location on the TPM, it is encrypted. When encrypted, but not on the TPM, it is not protected from deletion, but it is protected from disclosure of its sensitive portions. Wherever it is stored, it is in a Protected Location.

Objects in long-term protected storage need to be loaded into the TPM for use. The application that created the objects manages their movement from long-term storage to the TPM.

Since a TPM has limited memory, it may be unable to hold all objects required by all applications simultaneously. The TPM supports swapping of object contexts by a TPM Resource Manager (TRM) so that the TPM can service these multiple applications. The object contexts are encrypted before being returned to the TRM by the TPM. If the object is needed later, the TRM can reload the context into the TPM providing a cache-like behavior.

Encryption of Protected Locations uses multiple seeds and keys that never leave the TPM. One of these is the Context Key. It is a symmetric key used to encrypt data when it is temporarily swapped out of the TPM so that a different working set of objects may be loaded. Other sensitive values that never leave the TPM are the Primary Seeds. These seeds are the root of the storage hierarchies that protect objects that are retained by applications. A Primary Seed is a random number used to generate protection keys for other objects; these objects may be Storage Keys that contain protection keys that are then used to protect still more objects.

Primary Seeds may be changed, and when they are changed, the objects they protected will no longer be usable.

EXAMPLE The Storage Primary Seed (SPS) creates the Storage hierarchy for owner-related data, and that seed changes when the owner changes.

9.5.5 Integrity Measurement and Reporting

The Core Root of Trust for Measurement (CRTM) is the starting point of measurement. This process makes the initial measurements of the platform that are Extended into PCR in the TPM. For measurements to be meaningful, the executing code needs to control the environment in which it is running, so that the values recorded in the TPM are representative of the initial trust state of the platform.

A power-on reset creates an environment in which the platform is in a known initial state, with the main CPU running code from some well-defined initial location. Since that code has exclusive control of the platform at that time, it may make measurements of the platform from firmware. From these initial measurements, a chain of trust may be established. Because this chain of trust is created once when the platform is reset, no change of the initial trust state is possible, so it is called a static RTM (S-RTM).

An alternative method of initializing the platform is available on some processor architectures. It lets the CPU act as the CRTM and apply protections to portions of memory it measures. This process lets a new chain of trust start without rebooting the platform. Because the RTM may be re-established dynamically, this method is called dynamic RTM (D-RTM). Both S-RTM and D-RTM may take a system in an unknown state and return it to a known state. The D-RTM has the advantage of not requiring the system to be rebooted.

An integrity measurement is a value that represents a possible change in the trust state of the platform. The measured object may be anything of meaning but is often

- a data value,
- the hash of code or data, or
- an indication of the signer of some code or data.

The RTM (usually, code running on the CPU) makes these measurements and records them in RTS using Extend. The Extend process (see 17.2) allows the TPM to accumulate an indefinite number of measurements in a relatively small amount of memory.

The digest of an arbitrary set of integrity measurements is statistically unique, and an evaluator might know the values representing particular sequences of measurements. To handle cases where PCR values are not well known, the RTM keeps a log of individual measurements. The PCR values may be used to determine the accuracy of the log, and log entries may be evaluated individually to determine if the change in system state indicated by the event is acceptable.

Implementers play a role in determining how event data is partitioned. TCG's platform-specific specifications provide additional insight into specifying platform configuration and representation as well as anticipated consumers of measurement data.

Integrity reporting is the process of attesting to integrity measurements recorded in a PCR. The philosophy behind integrity measurement, logging, and reporting is that a platform may enter any state possible — including undesirable or insecure states — but is required to accurately report those states. An independent process may evaluate the integrity states and determine an appropriate response.

10 TPM Protections

10.1 Introduction

This part of ISO/IEC 11889 describes the protections provided by the Trusted Platform Module. This clause describes the properties of selected capabilities and selected data locations for a TPM that has been evaluated according to a Protection Profile and a TPM that has not been modified by physical means.

TPM protections are based on the concepts of Protected Capabilities and Protected Objects. A Protected Capability is an operation that must be performed correctly for a TPM to be trusted. A Protected Object is data (including keys) that must be protected for a TPM operation to be trusted. Protected Objects in the TPM reside in Shielded Locations; the TPM may manipulate the contents of Shielded Locations only by using Protected Capabilities. Protected Objects outside Shielded Locations have their integrity and confidentiality protected cryptographically.

Since a Protected Object may reside outside of Shielded Location protections, the definition of “access” to a Protected Object denotes disclosure of its contents, not modification. Such objects are not protected against loss or tampering. However, before loading a Shielded Location with an outside object, the TPM will use a secure hash function to validate that the object was properly protected and not altered. If the integrity check fails, the TPM returns an error and does not load the object.

The only operations on Shielded Locations of a TPM are the Protected Capabilities defined in ISO/IEC 11889 and the vendor-specific operations that meet the requirements of 10.4.

10.2 Protection of Protected Capabilities

A Protected Capability may be modified only by other Protected Capabilities in the same TPM. Thus, the process of updating TPM firmware is required to be a Protected Capability.

10.3 Protection of Shielded Locations

As noted, access to any data on a TPM requires use of a Protected Capability. Therefore, all information on a TPM is in a Shielded Location. The contents of a Shielded Location are not disclosed unless the disclosure is intended by the definition of the Protected Capability. A TPM is not allowed to export data from a Shielded Location other than by using a Protected Capability.

NOTE Data in an I/O buffer that can be modified by the host is not “on” the TPM, even though the I/O buffer may be shielded from access while the TPM is processing a command or generating a response.

10.4 Exceptions and Clarifications

Vendor-specific operations may access and modify Shielded Locations on a TPM under the following circumstances.

- A vendor-specific operation may use the standard TPM authorization mechanism.
- A vendor-specific capability may read any TPM-resident structure that is not required to be in a Shielded Location at all times if the usage of that structure is authorized per the structure’s authorization mechanism.

  EXAMPLE A vendor-specific command could use the public portion of a key. If the key is a user key, no authorization would be required.

NOTE Among other things, the exception above enables access to a Shielded Location so the structure's access authorization can be checked.

- Vendor-specific operations may use a sequence of Protected Capabilities.
- Vendor-specific operations may use the standard TPM command interface or use a vendor-defined interface.

These clarifications serve to approve specific legitimate interpretations of the requirements.

- A vendor-specific operation that takes advantage of exceptions and clarifications to the "protection" requirements should be defined as part of the security target of the TPM. Such a vendor-specific command or capability should be evaluated to determine whether it meets Platform-specific TPM and System Security Targets.
- If a TPM stores vendor-specific cipher-text that is protected against subversion to the same or greater extent as internal TPM-resources stored outside the TPM with TCG-defined methods, then that cipher-text does not require protection from physical attack. If the TPM stores only vendor-specific cipher-text that does not require protection from physical attack, that location may be excluded from analysis when determining whether the TPM complies with the "physical protection" requirements specified by TCG.
- If a TPM uses external memory for non-volatile storage of TPM state (including seeds and proof values), movement of the TPM state to and from the NV memory constitutes a vendor-defined operation that is allowed by ISO/IEC 11889. The protection profile of that TPM should include a description of the protections of that data to insure confidentiality and integrity of the data and to mitigate against rollback attacks.

11 TPM Architecture

11.1 Introduction

This clause describes the overall operation of the TPM and the functional units required for its operation. The major elements of the architecture are shown in

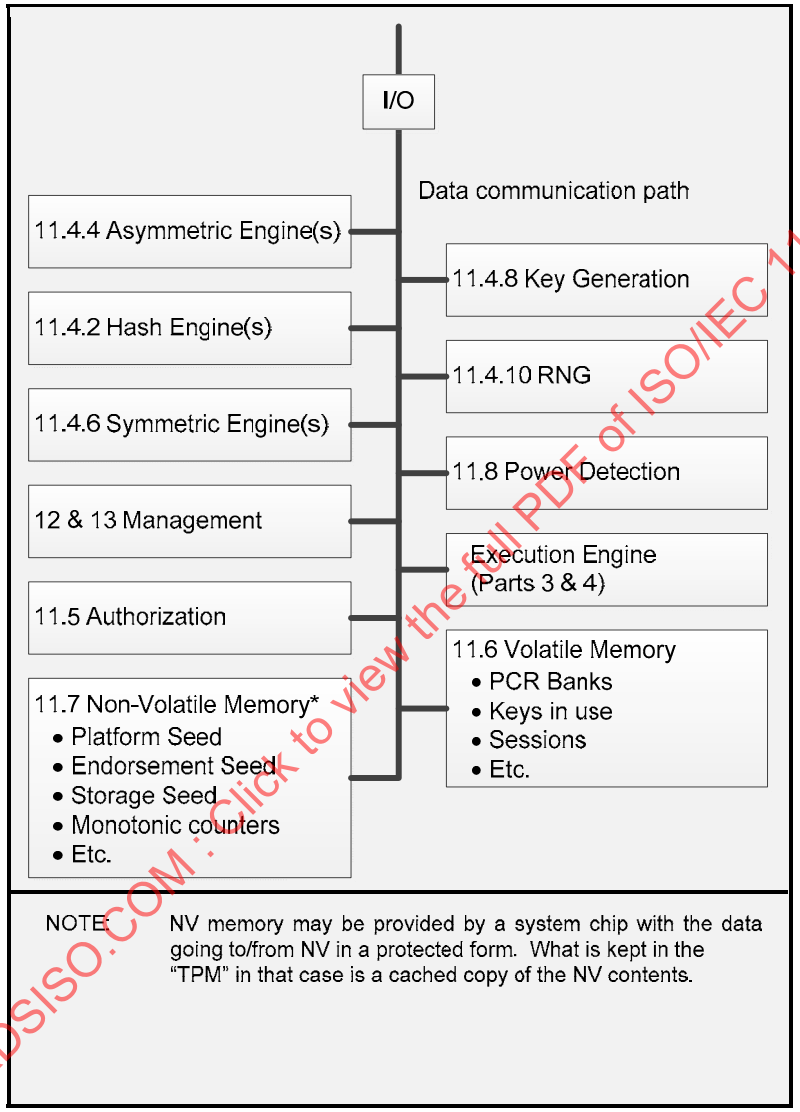


Figure 2 — Architectural Overview

11.2 TPM Command Processing Overview

Figure 3 is a high-level flow diagram for a TPM command. The figure shows only the normal flow for a command that executes successfully. The tabs on a box indicate the name of the module performing the operation. Additional details for each of the modules shown in Figure 3 are in clauses 11.4.2, 11.4.4, 11.4.6, 11.4.8, 11.4.10, 11.5, 11.6, 11.8, 12, 13 and ISO/IEC 11889-3 and ISO/IEC 11889-4.

The partitioning of functions in Figure 3 is illustrative and not normative.

The flow assumes that the command has been placed in an input buffer that is accessible to the Execute Command module (this name is used because of its similarity to the ExecCommand() function in the reference code that performs the functions illustrated here).

NOTE 1 The mechanism for getting the command into the TPM buffer and providing the command-available indication is specific to each physical interface and is defined in interface-specific documents.

The command structure includes a standard header (see clause 17.10) that Execute Command validates. It then determines if the command requires access to any Shielded Location that is identified by a handle. If so, it calls the Handle module to verify that the handle references the right type of resource for the command and that the resource is currently loaded on the TPM.

When control returns to Execute Command, it checks the *tag* parameter in the command header to determine if authorization values are provided. If so, Authorizations is called to validate that each of the authorizations is correct. The authorizations are associated with a handle value so the authorization is specific to a particular entity.

After validating the authorizations, Execute Command calls Command Dispatch to unmarshal the remaining command parameters and validate that the required parameters of the required type are present. All parameters are validated to meet the requirements of its data type as defined in ISO/IEC 11889-2 even if the parameter will subsequently be discarded because of optional behavior of a command.

After unmarshaling the parameters, Command Dispatch calls the command-specific library function to execute the specific command. Additional parameter checking may be required in the command-specific actions.

The command processing is structured so that changes to the TPM state do not occur until the TPM can validate that the command parameters are correct and that the resources necessary to complete the command are available. Only then will it make irreversible changes to the TPM state. This structuring ensures that when the TPM returns an error, the TPM will be in the same state as before command actions modified the data in any Shielded Location.

NOTE 2 Requiring that the TPM retain its state minimizes the interference between applications and helps prevent system instability due to careless use of the TPM by applications.

There are several classes of operations that return an error but may change TPM state.

- An authorization failure may update the dictionary attack mechanism.
- The self test mechanism has state that is considered to be different from the command execution state.

  EXAMPLE 1 Which algorithms have been tested might be part of the self test mechanism's state.

  Changes to this state may occur regardless of the command return code.

  EXAMPLE 2 An implicit self test invoked to test an algorithm required by the command might mark the algorithm as tested.

- If a self test fails, the TPM will go into Failure Mode.

When the command actions are complete, the Command Dispatch marshals response parameters into the output buffer. If the command had authorizations, Acknowledge is called to construct acknowledge session values for the response.

If the command encounters an error, the response packet will contain a code that is characteristic of the error and, when possible, an indication of whether the error was associated with a handle, an authorization session, or a command parameter. No additional qualifying data is present. In most cases, the error code and parameter location value suffice to isolate the problem.

NOTE 3 In the case of a self-test failure, the TPM response code is not sufficient to diagnose the problem. Therefore, a reporting scheme is provided so that the failure cause can be read. However, error report contents vary by vendor and are not standardized. There is thus no need to standardize self-test response codes because no standard remediation is possible for most self-test failures.

After constructing the response, including acknowledge sessions, the TPM indicates to the interface that the response is ready to be returned.

The TPM command/response structure is specified in Clause 17.10. See clause 19 for a description of the methods for creating the values that authorize use of a TPM Shielded Location and Clause 39 for response code formatting information.

During the processing of these commands, the TPM uses other modules that the following parts of clause 11 will describe.

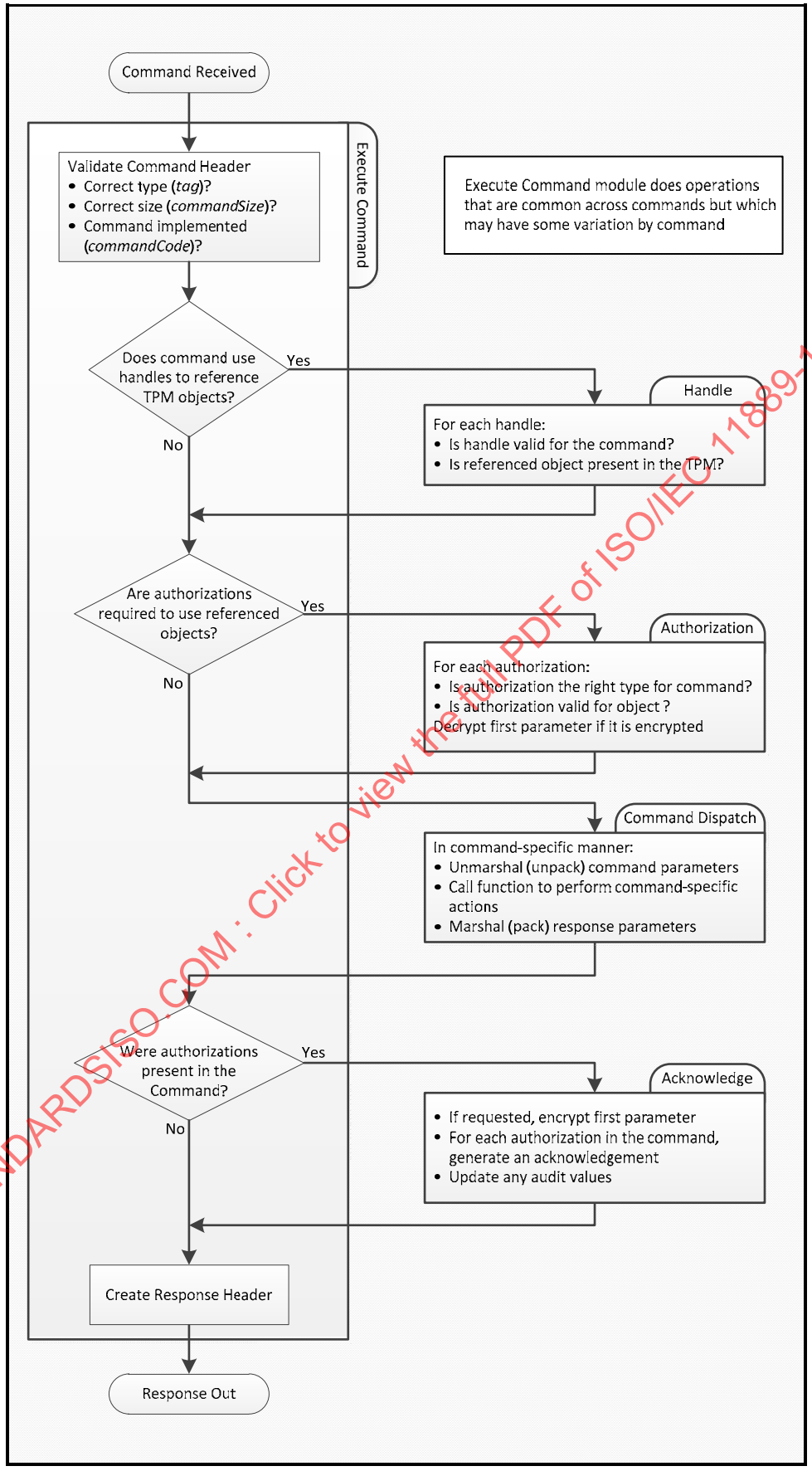


Figure 3 — Command Execution Flow

11.3 I/O Buffer

The I/O buffer is the communications area between a TPM and the host system. The system places command data in the I/O buffer and retrieves response data from the buffer.

A description of the physical processes used to move I/O buffer data to/from the system is beyond the scope of ISO/IEC 11889. Platform-specific working groups within the TCG produce the specifications for the physical interfaces to the TPM on their platforms. Those specifications detail the interactions between system software and the TPM I/O buffer.

There is no requirement that the I/O buffer be physically isolated from other parts of the system. It can be a shared memory. However, when processing of a command begins, the implementation must ensure that the TPM is using the correct values.

EXAMPLE If the TPM performs a hash of the command data as part of the authorization processing, the TPM needs to protect the validated command data from modification. That is, before the data is validated, it needs to be protected from modification. Before the data is modified, it needs to be in a Shielded Location.

11.4 Cryptography Subsystem

11.4.1 Introduction

The Cryptography subsystem implements the TPM's cryptographic functions. It may be called by the Command Parsing module, the Authorization Subsystem, or the Command Execution module. The TPM employs conventional cryptographic operations in conventional ways. These operations include

- hash functions,
- asymmetric encryption and decryption,
- asymmetric signing and signature verification,
- symmetric encryption and decryption,
- symmetric signing (HMAC) and signature verification, and
- key generation.

The remainder of clause 11 describes some algorithms usually found in a TPM to show how they are handled. These descriptions illustrate, but do not limit, the choice of available algorithms.

11.4.2 Hash Functions

Hash functions may be used directly by external software or as the side effect of many TPM operations. The TPM uses hashing to provide integrity checking and authentication as well as one-way functions, as needed (such as, KDF).

A TPM should implement an approved hash algorithm that has approximately the same security strength as its strongest asymmetric algorithm.

EXAMPLE 1 An ECC with a 384-bit key has a security strength of 192 bits. SHA384, with 192 bits of security, would meet the preceding requirement above.

NOTE The TCG could create sets of algorithms that do not have the same security strength for the hash and asymmetric algorithms.

A hash function will be denoted by $\mathbf{H}_{algorithm}()$ with the algorithm subscript indicating the hash algorithm or the parameter that contains the hash algorithm identifier. In some cases, the algorithm subscript is missing, in which case the algorithm will be determined by context.

The Command Dispatch module will use the hash function when validating certain types of authorizations. Hash functions are also used in support of other operations in the TPM.

EXAMPLE 2 PCR Extend is a TPM command that relies on a hash function.

11.4.3 HMAC Algorithm

The TPM implements the Hash Message Authentication Code (HMAC) algorithm specified in ISO/IEC 9797-2. Per clause 11.5, support for HMAC is mandatory, making ISO/IEC 9797-2 indispensable for an implementation of this International Standard.

An HMAC is a form of symmetric signature over some data. It provides assurance that protected data was not modified and that it came from an entity with access to a key value. To have usefulness in protecting data, the key value needs to be a secret or a shared secret.

ISO/IEC 9797-2 defines the HMAC operation as:

$$\mathbf{HMAC}(K, text) = \mathbf{H}((\bar{K} \oplus OPAD) \,||\, \mathbf{H}((\bar{K} \oplus IPAD) \,||\, text)) \quad (1)$$

NOTE See ISO/IEC 9797-2 for a description of parameters.

Performing the HMAC computation requires selection of a hash algorithm. ISO/IEC 11889 modifies the notation from ISO/IEC 9797-2 to be:

$$\mathbf{HMAC}_{hashAlg}\,(K, text) \quad (2)$$

If the algorithm subscript is not present, the hash algorithm is implied by the context.

The Command Dispatch module may use the HMAC function to validate an authorization. The HMAC function may be used by the Command Execution module in support of its operations.

11.4.4 Asymmetric Operations

A TPM uses asymmetric algorithms for attestation, identification, and secret sharing. A TPM may support any asymmetric algorithm to which the TCG has assigned an identifier. Because the algorithms supported need to be assigned an identifier by the TCG in the TCG Algorithm Registry, the TCG Algorithm Registry is indispensable for an implementation of this International Standard. An asymmetric algorithm identifier will indicate a family of algorithms and methods that are used with that algorithm.

The methods for using an asymmetric algorithm are found in algorithm-specific annexes to this part of ISO/IEC 11889. Currently, the only supported asymmetric algorithms are RSA (specified in Annex A) and ECC using prime curves (specified in Annex A).

A TPM is required to implement at least one asymmetric algorithm.

11.4.5 Signature Operations

11.4.5.1 Signing

The TPM may sign using either an asymmetric or a symmetric algorithm. The method of signing depends on the type of the key. For an asymmetric algorithm, the methods of signing are dependent on the algorithm (RSA or ECC). For symmetric signatures, only the HMAC signing scheme is currently defined. If a key may be used for signing, then it will have the *sign* attribute.

NOTE The signing schemes for RSA are specified in A.6 (RSASSA_PKCS1v1_5) and A.7 (RSASSA_PSS). The signing schemes for ECC are specified in B.4 (EC Signing).

A key with a sign attribute may also have a restriction on the contents of the message that can be signed with the key. When a key has this restriction, the TPM will not use the key to sign message digests that the TPM did not compute.

Any attestation message produced by a TPM will have a header (TPM_GENERATED_VALUE) to identify the data as being produced within a TPM. If a restricted key is used to sign this data, then a relying party can have assurance that the message data came from a TPM.

To allow a restricted key to sign an externally generated message, the TPM is used to produce the message digest. When the TPM computes the digest, it will validate that the message does not begin with TPM_GENERATED_VALUE. If it does, then the TPM will not produce the special certification (a ticket) that indicates that the digest was produced by the TPM and is safe to sign with a restricted key.

A key designated as a signing key may be used in any command that uses a signing key. For some commands, the signing scheme may be specified in the command. Not all schemes are valid for all keys, and the TPM generates an error if the scheme is not allowed with the indicated key type.

EXAMPLE 1 The RSASSA-PKCS1-v1_5 signing scheme is not valid with an ECC key.

EXAMPLE 2 A key that has the "restricted" attribute can only be used with one signing scheme. If it is limited to be used with RSASSA-PSS, it cannot be used with RSASSA-PKCS1-v1_5.

A restricted signing key is required to have a signing scheme specified in the key definition and that is the only signing scheme that is allowed to be used with the key. For an unrestricted key, the key definition may contain a signing scheme selection or the signing scheme may be determined when the key is used. To defer the signing scheme selection, the key would be created with TPM_ALG_NULL as the signing scheme selection.

11.4.5.2 Signature Verification

TPM2_VerifySignature() validates a signature. The command takes a handle of a public key, a digest, and a block that contains the signature over the digest.

The TPM validates that the signature scheme is compatible with the selected key. Any combination of hashes and non-anonymous signature schemes that a TPM supports for signing is also supported for signature verification.

If the signature is valid, the TPM will produce a ticket.

11.4.5.3 Tickets

A ticket is an HMAC signature that uses a proof value as the HMAC key.

NOTE Hierarchy proof values are specified in detail in 14.4.

The TPM uses tickets for two purposes:

- re-signing data. After checking an asymmetric signature, the TPM re-signs the digest using a TPM symmetric key. The TPM can later re-verify a signature without having to load the asymmetric key; and
- expanding state memory. When hashing an external message, the TPM has some state that indicates the message did not start with TPM_GENERATED_VALUE. This state information cannot be retained indefinitely in the TPM. A ticket allows this state to be stored off of the TPM in a way that is easy for the TPM to validate. When a digest is later presented to the TPM to be signed, the ticket is provided allowing the TPM to validate that the digest to be signed is safe to sign.

The proof value used for a ticket will minimally have a number of bits equal to the size of the digest produced by the hash algorithm.

EXAMPLE 1 A proof value of 256 bits is needed for a SHA256 ticket.

There are five different ticket types:

1) TPMT_TK_CREATION – this ticket type is produced when an object is created (TPM2_Create() or TPM2_CreatePrimary()). The ticket is used in TPM2_CertifyCreation() so that the TPM can certify that it created a specific object and the environmental parameters (PCR) that were extant when the object was created. This avoids having the digest of the creation data be a permanent part of an object's data structure.
2) TPMT_TK_VERIFIED – this ticket type is produced by TPM2_VerifySignature() and used by TPM2_PolicyAuthorize(). If a signature is signed by an asymmetric key, the signature verification might be time consuming. If the same authorization is going to be used many times, there is a performance advantage to having the asymmetric authorization converted so that it uses symmetric cryptography which is usually faster. This ticket is the symmetric equivalent authorization.

   EXAMPLE 2 An authorization for TPM2_PolicyAuthorize() is an example of an authorization that might be used many times.

3) TPMT_TK_AUTH – this ticket is produced by TPM2_PolicySigned() or TPM2_PolicySecret() and used in TPM2_PolicyTicket(). A policy authorization can be tied to a specific policy session or allowed to be used in any policy. When it can be used in any policy, it has a time at which it expires (which can be some arbitrary time in the future). The long lived authorization may be given in TPM2_PolicySigned()/TPM2_PolicySecret() and a ticket is produced that is used to verify the authorization parameters (what was authorized) and the time in the future when the authorization expires. This ticket is then processed by TPM2_PolicyTicket() and, until the ticket expires, will have the same effect on the *policyDigest* computation as the original authorization.

   NOTE If produced by TPM2_PolicySigned(), the ticket will use the TPM_ST_AUTH_SIGNED structure tag and if produced by TPM2_PolicySecret(), the ticket will use the TPM_ST_AUTH_SECRET structure tag. TPM2_PolicyTicket() will use this tag to indicate which command code to use (TPM_CC_PolicySigned/TPM_CC_PolicySecret) when extending *policyDigest*.

4) TPMT_TK_HASHCHECK – This ticket is used to indicate that a digest of external data is safe to sign using a restricted signing key. A restricted signing key may only sign a digest that was produced by the TPM. If the digest was produced from externally provided data, there needs to be an indication that the data did not start with the same first octets as are used for data that is generated within the TPM. This prevents "forgeries" of attestation data. This ticket is used to provide the evidence that the data used in the digest was checked by the TPM and is safe to sign. Assuming that the external data is "safe", this type of ticket is produced by TPM2_Hash() or TPM2_SequenceComplete() and used by TPM2_Sign().

5) NULL Ticket – A NULL Ticket is produced when a response has a ticket, but no ticket is produced. A NULL Ticket may also be used as an input parameter when the command requires a ticket but no ticket data is available.

EXAMPLE 3 A NULL Ticket can returned by TPM2_PolicySecret() with an expiration time of zero. It does not produce a ticket because, since the expiration time was zero, the authorization expires immediately.

11.4.6 Symmetric Encryption

11.4.6.1 Introduction

The TPM uses symmetric encryption to encrypt some command parameters (typically, authentication information) and to encrypt Protected Objects stored outside it. Cipher Feedback mode (CFB) is the only block cipher mode required by ISO/IEC 11889. CFB is defined ISO/IEC 10116:2006, making ISO/IEC 10116:2006 indispensable for an implementation of this International Standard.

Any symmetric block cipher supported by a TPM may be used for parameter encryption. Additionally, a TPM should support XOR obfuscation, which is a hash-based stream cipher. XOR obfuscation may be used only for confidential parameter passing.

When paired with an asymmetric key — as in an ECC decrypting key — a symmetric key is required to have as many bits of security strength as the asymmetric key with which it is paired.

EXAMPLE 1 SP800-57 classifies 2048-bit RSA as providing 112 bits of security. AES with 128- or 256-bit keys provides adequate symmetric security for pairing with a 2048-bit RSA key.

EXAMPLE 2 A prime-modulus ECC key has a security strength that is half the size of the prime modulus. AES with 128- or 256-bit keys is suitable for pairing with a 256-bit ECC key, but AES with 128-bit keys is not recommended for pairing with a 384-bit ECC key.

When a symmetric key is used for data encryption, the encrypted data has an HMAC. This HMAC is checked before the data is decrypted. Verification that the decrypted data is properly associated with the symmetric key is intended to make it more difficult to perform power analysis. To defeat the protections, it would be necessary to defeat two different families of protection rather than one as would exist if the integrity protection were applied to the clear text rather than the cipher text.

11.4.6.2 Block Cipher Modes

The block cipher modes referenced in ISO/IEC 11889 are defined in ISO/IEC 10116:2006. That standard allows parameterization of most of the modes. In a TPM implementation, the parameters are fixed, as defined in Table 2.

Table 2 — Block Cipher Parameters

Mode	Common Name	Parameter	Comments
CTR	Counter	j = n	size of the plaintext variable
OFB	Output Feedback	j = n	size of the plaintext variable
CBC	Cipher-block Chaining	m = 1	interleave factor
CFB	Cipher-feedback	r = n	size of feedback buffer
		k = n	size of feedback variable
		j = n	size of plaintext variable

ECB	Electronic Code Book	none	
NOTE	*n* is the input block size of the cipher.		

11.4.6.3 Cipher Feedback (CFB) Mode

CFB is used when a symmetric block cipher is chosen as the encryption algorithm associated with a session. When used for parameter encryption, the key and Initialization Vector (IV) are derived from a per-session key so that reuse of the same key and IV is statistically unlikely.

NOTE ISO/IEC 10116:2006 uses the term Start Variable instead of Initialization Vector (IV).

CFB is also used for symmetric encryption of the sensitive area of an object when the object is not stored in a Shielded Location. When used in this way, the key and IV are derived from a secret. In some cases, the IV is set to zero.

11.4.6.4 XOR Obfuscation

XOR obfuscation resembles Counter mode (CTR) block encryption, but it uses a KDF as the pseudo-random function instead of a symmetric block cipher.

XOR obfuscation reduces to one (a hash) the number of algorithms that a caller needs in common with the TPM in order to use the TPM with some level of confidentiality and authentication.

The XOR scheme in ISO/IEC 11889 differs from that used in ISO/IEC 11889 (first edition): it uses a different formulation for input into the hash function.

When ISO/IEC 11889 calls for use of XOR obfuscation, it uses a function reference. The function prototype is:

$$\mathbf{XOR}(data, hashAlg, key, contextU, contextV) \tag{3}$$

where

data	a variable-sized buffer containing the data to be obfuscated
hashAlg	the hash algorithm to be used in the KDF
key	a variable-sized value containing a secret key
contextU	a variable-sized value used to qualify one of the parties to the operation (often a nonce value)
contextV	a variable-sized value used to qualify one of the parties to the operation (often a nonce value)

The **XOR**() function uses the *hash*, *key*, *contextU*, and *context*V parameters in a call to **KDFa**() to produce a *mask* value:

$$mask := \mathbf{KDFa}\ (hashAlg, key, \text{“XOR”}, contextU, contextV, data.size \bullet 8) \tag{4}$$

See clause 5.4 for the definition of the third parameter in (4).

The octets of *mask* are then XOR'd with the octets of *data.buffer*.

11.4.7 Extend

The Extend operation is used to make incremental updates to a digest value. It is useful for updating PCR, auditing, and constructing policy. Extend uses a hash function to combine new data with an existing digest. Its notation is:

$$digest_{new} := \mathbf{H}_{hashAlg}\,(digest_{old} \,||\, data_{new}) \tag{5}$$

where

$digest_{new}$ the value of the digest (such as, a PCR) after the Extend operation

EXAMPLE 1 A PCR is an example of a digest.

$\mathbf{H}_{hashAlg}$ the hash function using a context-specific algorithm

EXAMPLE 2 The hash algorithm associated with a specific bank of PCR.

$digest_{old}$ the value of the digest before the Extend operation

$data_{new}$ a variable number of octets of data that are to be hashed with the initial value of $digest_{old}$ to produce Extend results

The Extend operation may also apply to an NV Index that has the TPMA_NV_EXTEND attribute.

11.4.8 Key Generation

Key generation produces two different types of keys. The first, an ordinary key, is produced using the random number generator (RNG) to seed the computation. The result of the computation is a secret key value kept in a Shielded Location.

The second type, a Primary Key, is derived from a seed value, not the RNG directly. The RNG usually generates the seed that is persistently stored on the TPM. Generation of a Primary Key from a seed is based on use of an approved key derivation function (KDF). The KDF from SP800-108 is widely used in ISO/IEC 11889.

ISO/IEC 11889 places no upper limit on the time allowed to generate a key. Platform-specific specifications may limit the time for generating various key types.

Depending on the application, the TPM may generate a key by

- using bits from the RNG, or
- deriving the key from another secret value.

There are many ways to generate keys; these methods are specified in detail in each clause where generation of a key is required.

11.4.9 Key Derivation Function

11.4.9.1 Introduction

The TPM uses a hash-based function to generate keys for multiple purposes. ISO/IEC 11889 uses two different schemes: one for ECDH and one for all other uses of a KDF.

The ECDH KDF is from SP800-56A. The Counter mode KDF, from SP800-108, uses HMAC as the pseudo-random function (PRF). It is referred to in ISO/IEC 11889 as **KDFa**().

11.4.9.2 KDFa()

With the exception of ECDH, **KDFa**() is used in all cases where a KDF is required. **KDFa**() uses Counter mode from SP800-108, with HMAC as the PRF.

As defined in SP800-108, the inner loop for building the key stream is:

$$K(i) := \mathbf{HMAC}\ (K_I, [i]_2 \,||\, Label \,||\, 00_{16} \,||\, Context \,||\, [L]_2) \quad (6)$$

where

$K(i)$	the i[th] iteration of the KDF inner loop
HMAC()	the HMAC algorithm using an approved hash algorithm
K_I	the secret key material
$[i]_2$	a 32-bit counter that starts at 1 and increments on each iteration
Label	a string indicating the use of the key produced by this KDF
Context	a binary string containing information relating to the derived keying material
$[L]_2$	a 32-bit value indicating the number of bits to be returned from the KDF

NOTE Equation (6) is not **KDFa**(). **KDFa**() is the function call defined below.

After each iteration, the HMAC digest data is concatenated to the previously produced value until the size of the concatenated string is at least as large as the requested value. The string is then truncated to the desired size (which causes the loss of some of the most recently added bits), and the value is returned.

When ISO/IEC 11889 calls for use of this KDF, it uses a function reference to **KDFa**(). The function prototype is:

$$\mathbf{KDFa}\ (hashAlg, key, label, contextU, contextV, bits) \quad (7)$$

where

hashAlg	a TPM_ALG_ID to be used in the HMAC in the KDF
key	a variable-sized value used as K_I
label	a variable-sized, null-terminated string
contextU	a variable-sized value concatenated with *contextV* to create the *Context* parameter used in equation (6) above
contextV	a variable-sized value concatenated to *contextU* to create the *Context* parameter used in equation (6) above
bits	a 32-bit value used as $[L]_2$; and is the number of bits returned by the function

The values of *contextU* and *contextV* are passed as sized buffers and only the buffer data is used to construct the *Context* parameter used in equation (6) above. The size fields of *contextU* and *contextV* are not included in the computation. That is:

$$Context := contextU.buffer \,||\, contextV.buffer \tag{8}$$

The 32-bit value of *bits* is in TPM canonical form, with the least significant bits of the value in the highest numbered octet.

Using the parameters from the call to **KDFa**() in equation (6) results in the inner loop being:

$$K(i) := \mathbf{HMAC}(key, [i]_2 \,||\, label \,||\, ContextU.buffer \,||\, ContextV.buffer \,||\, bits) \tag{9}$$

The implied return from this function is a sequence of octets with a length equal to (*bits* + 7) / 8. If *bits* is not an even multiple of 8, then the returned value occupies the least significant bits of the returned octet array, and the additional, high-order bits in the 0th octet are CLEAR. **The unused bits of the most significant octet (MSO) are masked off and not shifted.**

EXAMPLE If **KDFa**() were used to produce a 521-bit ECC private key, the returned value would occupy 66 octets, with the upper 7 bits of the octet at offset zero set to 0.

11.4.9.3 Note on Labels

As shown in equation (6), there is an octet of zero that separates *Label* from *Context*. In SP800-108, *Label* is a sequence of octets that may or may not have a final octet that is zero. ISO/IEC 11889 uses a NULL-terminated string for *Label* so that an additional octet of zero is not required.

11.4.9.4 KDFe for ECDH

Producing a symmetric encryption key for an ECC-protected object uses “One-Pass Diffie-Hellman, C(1, 1, ECC CDH)” from SP800-56A, 6.2.2.2. The KDF used is the “Concatenation Key Derivation Function (Approved Alternative 1)”. The inner loop of that KDF uses:

$$digest_i := \mathbf{H}(counter \,||\, Z \,||\, OtherInfo) \tag{10}$$

where

$digest_i$	the digest generated on the i[th] iteration of the loop (i starts at 1)
H()	an approved hash function
$counter$	a 32-bit counter that is initialized to 1 and incremented on each iteration
Z	the X coordinate of the product of a public ECC key and a different private ECC key
$OtherInfo$	a collection of qualifying data for the KDF defined below

The 32-bit *counter* value is included in TPM canonical form, with the least-significant bit of the counter in the highest numbered octet.

After each iteration, $digest_i$ is concatenated to the previously produced digests (MSO of $digest_i$ follows the LSO of $digest_{i-1}$). The number of iterations is determined by the number of bits to be produced and the size of the digest produced by the hash function. In the returned octet string, the MSO of the returned value is the MSO of $digest_1$.

In SP800-56A, *OtherInfo* is specified as:

$$OtherInfo \coloneqq AlgorithmID \,||\, PartyUInfo \,||\, PartyVInfo \,\{||\, SuppPubInfo\}\,\{||\, SuppPrivInfo\} \quad (11)$$

where

AlgorithmID	a bit string that indicates how the derived keying material will be parsed and for which algorithm(s)
PartyUInfo	public information contributed by party U (the initiator)
PartyVInfo	public information contributed by party V (the responder)
SuppPubInfo	public information known to both U and V (optional)
SuppPrivInfo	private (secret) information known to both U and V (optional)

ISO/IEC 11889 requires that *OtherInfo* be constructed as:

$$OtherInfo \coloneqq Use \,||\, PartyUInfo.buffer \,||\, PartyVInfo.buffer \quad (12)$$

where

Use	a null-terminated string indicating the use of the key. (See clause 5.4 for the definition of these values.) This field satisfies the requirements of SP800-56A since the parsing of keying material is determined by the use.

EXAMPLE 1 Example values of null-terminated strings indicating the use of the key are “DUPLICATE”, “IDENTITY”, “SECRET”, etc.

PartyUInfo.buffer	the x-coordinate of the public point of an ephemeral key
PartyVInfo.buffer	the x-coordinate of the public point of a static TPM key

The x-coordinates of the public points are *sized buffers* (that is, integers indicating the size in octets of the buffer that follows). The buffer data is used in the KDF but the size field is not.

When ISO/IEC 11889 calls for use of this KDF, it uses a function reference to **KDFe**(). The function prototype is:

$$\mathbf{KDFe}(hashAlg, Z, Use, PartyUInfo, PartyVInfo, bits) \quad (13)$$

where

hashAlg	the hash algorithm to be used as **H**() in equation (10) above
Z	the product of a public point and a private x-coordinate
Use	a null-terminated string indicating the use of the key. (See clause 5.4 for the definition of these values.)

EXAMPLE 2 Example values of null-terminated strings indicating the use of the key are “DUPLICATE”, “IDENTITY”, “SECRET”, etc.

PartyUInfo	the x-coordinate of the public point of an ephemeral key
PartyVInfo	the x-coordinate of the public point of a static TPM key

bits a 32-bit value indicating the number of bits to be returned

The implied return from this function is an octet string containing *bits*/8 octets. If *bits* is not an even multiple of 8, the return value is the least-significant bits of the return value, and the additional high-order bits in the 0th octet are CLEAR. The unused bits of the MSO are masked off and not shifted.

NOTE The function prototype in (13) is not a C-language prototype but, rather, a prototype to illustrate the parameters of the KDF for specific uses. The C-language prototype will include an extra parameter that will be the buffer to receive the key material generated by the KDF.

11.4.10 Random Number Generator (RNG) Module

11.4.10.1 Source of Randomness

The RNG is the source of randomness in the TPM. The TPM uses random values for nonces, in key generation, and for randomness in signatures.

The RNG is a Protected Capability with no access control. It nominally consists of

- an entropy source and collector,
- a state register, and
- a mixing function (typically, an approved hash function).

The entropy collector collects entropy from entropy sources and removes bias. The collected entropy is then used to update the state register providing input to the mixing function to produce the random numbers.

The mixing function may be implemented with a pseudo-random number generator (a PRNG). A PRNG may produce numbers that are apparently random from a non-random input. Combining an approved PRNG with an input that has considerably more entropy than a counter yields an RNG with properties no worse than the underlying PRNG and possibly much better.

EXAMPLE An example of non-random input is a counter.

The RNG should meet the certification requirements of the intended market.

The TPM should provide sufficient randomness for each use by an internal function. When accessed by an external call, it should be able to provide 32 octets of randomness. Larger requests may fail if insufficient randomness is available.

Each RNG access produces a new value regardless of the data's use. There is no distinction between accesses for internal versus external purposes.

11.4.10.2 Entropy Source and Collector

A TPM should have at least one internal source of entropy, and possibly more. These sources could include noise, clock variations, air movement, and other types of events.

As noted, the entropy collector is the process that collects the entropy from various sources and removes bias.

EXAMPLE If the entropy source has a bias of creating 60 percent 1s and only 40 percent 0s, then the collector design corrects the bias before sending the information to the state register.

The entropy source and collector should provide entropy to the state register in a manner that is not visible to an outside process or other TPM capability.

The entropy collector should regularly update the state register with additional, unbiased entropy.

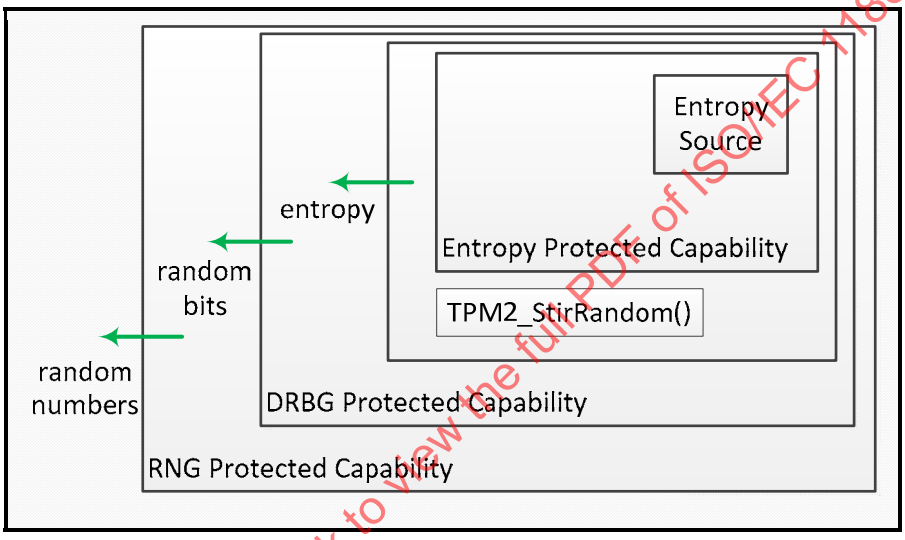


Figure 4 — Random Number Generation

Any Protected Capability that requires an unpredictable number obtains it from a Random Number Generator (RNG) Protected Capability in the same TPM. The RNG Protected Capability assembles random bits from a Deterministic Random Bit Generator (DRBG) Protected Capability in the same TPM. The DRBG Protected Capability obtains entropy from the entropy Protected Capability in the same TPM and the TPM2_StirRandom() Protected Capability can be used to add additional information. The entropy Protected Capability obtains entropy from an entropy source in the same TPM.

NOTE 1 The "additional information" added by TPM2_StirRandom() could be entropy gathered from other sources but the TPM has no way of determining if the value has any entropy or not. As a consequence, it is just deemed to be "additional information."

NOTE 2 The DRBG Protected Capability of a non-FIPS TPM could consist of a DRBG mechanism that complies with NIST Recommendation SP800-90 A; except it does not comply with its Clause 11.

NOTE 3 The DRBG Protected Capability of a FIPS TPM could consist of a DRBG mechanism that complies with NIST Recommendation SP800-90 A.

The DRBG mechanism security should be at least as strong as the security strength of the strongest cryptographic algorithm implemented in the TPM.

The DRBG Protected Capability should be reseeded using entropy from the entropy Protected Capability when:

- a flag is SET indicating that reseeding is required;
- TPM2_StirRandom() is executed;
- after the TPM has failed a self-test; or
- before the SPS is replaced.

It may be reseeded at other times, as well.

NOTE 4 Each TPM could be seeded during TPM manufacture, via a manufacturer-specific method, using a personalization string for the DRBG that is specific to the manufacturer and the type of TPM, plus a manufacturer-provided nonce that is specific to the individual TPM.

11.4.10.3 Nonce Creation

The RNG module provides the bits used in any TPM-generated nonce.

11.4.11 Algorithms

11.4.11.1 Algorithm Identifiers

The structures and commands in ISO/IEC 11889 are constructed with minimal reliance on algorithm defaults.

In most cases, an algorithm identifier identifies a family of algorithms followed by qualifiers. This differs from ISO/IEC 11889 (first edition), which often included the key size in the algorithm identifier (TPM_ALG_AES128). ISO/IEC 11889 only uses the ISO/IEC 11889 (first edition) form of algorithm identifiers for hash algorithms.

Since ISO/IEC 11889 depends on being able to discern the hash output size from the algorithm ID, its hash algorithm identifiers imply a digest size.

EXAMPLE 1 Some of the hash algorithm identifiers are TPM_ALG_SHA256, TPM_ALG_SHA384, and TPM_ALG_SM3_256.

Algorithm identifiers for symmetric and asymmetric encryption identify the family. For these algorithms, supplementary information is required to define parameters.

EXAMPLE 2 Examples of algorithm families are RSA, ECC, AES, etc.

EXAMPLE 3 Some family algorithm identifiers are TPM_ALG_ECC, TPM_ALG_RSA, TPM_ALG_SM4, and TPM_ALG_AES.

11.4.11.2 Algorithm Support

ISO/IEC 11889 does not require implementation of any specific set of algorithms. When determining algorithms or algorithm sets supported, implementers should carefully consider factors such as use cases, strength of function, interoperability, backward compatibility, algorithm diversity, etc. TCG recommends using TCG platform-specific specifications that reflect industry best practices.

NOTE 1 It is anticipated that support for ISO/IEC 11889 (first edition) compatible algorithms will be retained unless support for the ISO/IEC 11889 (first edition) algorithms (RSA 2048 and SHA1) would prevent that TPM from being sold.

TCG will specify sets of algorithms to be incorporated by various platform-specific specifications. Each set includes a minimum of one hash algorithm, one symmetric encryption algorithm with approved

parameters, and one asymmetric encryption/signing algorithm with approved parameters. Without a complete set of algorithms, the TPM would be unable to support all necessary functions.

A TPM may support algorithms in addition to the required sets. These do not need to be part of any set.

EXAMPLE The TPM might include an additional hash algorithm without including an additional asymmetric or symmetric algorithm.

It is possible, and very likely given the multitude of algorithms supported by the TPM, that key-size support will differ between TPM implementations. In addition, keys created by outside software may greatly increase the number of key sizes that are possible to load.

A TPM will not create or load an object that uses an algorithm that is not supported by the TPM. When creating an object, the TPM checks the template for the object being created and when loading an object the TPM checks the public area of the object. In both cases, the TPM validates that it supports all of the indicated algorithms, parameters, and key sizes.

The strength of at least one algorithm set supported by a TPM should be at least 112 bits. Other algorithms and algorithm sets may be supported in any combination.

NOTE 2 A set's strength is normally determined by the number of bits in a key of the symmetric algorithm. An exception is Suite B, Top Secret, where the strength is considered to be 192 bits even though the symmetric algorithm has 256-bit keys.

If a TPM supports RSA, it should support a key size of 2048 bits or larger. Support for smaller key sizes is allowed but discouraged.

NOTE 3 Support for smaller keys is allowed so that legacy keys can continue to be supported. Use of key-sizes less than 1024 bits is strongly discouraged.

A platform-specific specification may mandate support for algorithms or algorithm sets. It may select only those algorithms for which the TCG has assigned algorithm identifiers.

A TPM may only implement algorithms that have a TCG-assigned algorithm ID.

11.5 Authorization Subsystem

The Authorization Subsystem is called by the Command Dispatch module at the beginning and end of command execution. Before the command may be executed, the Authorization Subsystem checks that proper authorization for use of each of the Shielded Locations has been provided.

Some commands access Shielded Locations that require no authorizations; access to some locations may require a single-factor authorization; and access to other Shielded Locations may require use of an authorization policy of arbitrary complexity.

The only cryptographic functions required by the Authorization Subsystem are hash and HMAC. An asymmetric algorithm may be required if TPM2_PolicySigned() is implemented.

The details of the different methods of authorization are provided in Clause 19.

11.6 Random Access Memory

11.6.1 Introduction

Random access memory (RAM) holds TPM transient data. Data in TPM RAM is allowed, but not required, to be lost when TPM power is removed. Because the values in TPM RAM may be lost, in ISO/IEC 11889 they are referred to as being volatile, even if the data loss is implementation-dependent.

When ISO/IEC 11889 refers to a value that has both volatile and non-volatile copies, they may be kept in a single location as long as that location has the properties of allowing random access and having unlimited endurance.

Not all values in TPM RAM are in Shielded Locations. A portion of TPM RAM contains the I/O buffer with properties that are specified in 11.3.

11.6.2 Platform Configuration Registers (PCR)

PCR are Shielded Locations used to validate the contents of a log of measurements. The nominal behavior of a trusted platform is to maintain, in a log, a record of the events that affect the security state of the platform, at least through the boot process while it is establishing the TCB. When additions are made to the log, the TPM receives a copy of the log entry or the digest of data described by the log. The data sent to the TPM is included in an accumulative hash in a PCR. The TPM may then provide an attestation of the value in the PCR, which, in turn, verifies the contents of the log.

It is possible for a single PCR to record all log entries. However, this would make it difficult to evaluate the different stages of platform evolution as it boots into the operating system. Normally, multiple PCR are provided in a TPM to allow simplification of the evaluation.

EXAMPLE A TPM intended for a PC could have a PCR dedicated to recording measurements of the BIOS, a PCR dedicated to the boot ROM on add-in cards, and a PCR dedicated to the OS loader. The platform-specific specifications determine the number of PCR and their attributes in a TPM.

PCR may also be used to gate access to an object. If selected PCR do not have the required values, the TPM will not allow use of the object.

A TPM may maintain multiple banks of PCR. A PCR bank is a collection of PCR that are Extended with the same hash algorithm. PCR banks are identified by the hash algorithm used to Extend the PCR in that bank.

Multiple banks may handle situations where one hash algorithm is required for legacy or compatibility with one set of applications, while a different hash algorithm is required to meet the security needs of another application. Within a bank, all PCR updates use the same hash algorithm. Not all banks need to have the same number of PCR, but the attributes of all PCR with the same index, other than hash algorithm, are the same in all banks.

EXAMPLE If PCR[0] has an attribute that allows it to be reset by TPM2_PCR_Reset(), then that attribute applies to PCR[0] in all banks.

NOTE 1 Since banks may have different numbers of PCR, a PCR index value might not be valid for all banks. The allocation of PCR could also be changed by TPM2_AllocatePCR() using Platform Authorization. Changing the PCR allocation does not change the attributes of the PCR.

The contents of a PCR may be modified or reported. The two ways to modify a PCR are to reset it or Extend it. Reporting on a PCR may be accomplished through simple reading, inclusion in an attestation, or inclusion in a policy.

Although listed in clause 11.6.2, PCR need not be maintained in RAM. They may be kept in non-volatile memory. If kept in non-volatile memory, consideration must be made for the possible impact on TPM performance during the critical boot phase, when many measurements are recorded.

A TPM is required to implement a PCR bank for each supported algorithm. However, a PCR bank may be defined such that it contains no PCR.

NOTE 2 The requirement that a PCR bank be implemented for each hash algorithm allows the unmarshaling to be based on the implemented algorithms rather than the implemented PCR.

The TPM may support Resume PCR that retain their state across a TPM Resume sequence but are set to their default initial value on TPM Reset or TPM Restart.

11.6.3 Object Store

TPM RAM holds keys and data that are loaded into the TPM from external memory. In most cases, an object may not be used or modified unless it was first loaded into TPM RAM with one of the object load commands: (TPM2_Load(), TPM2_CreatePrimary(), TPM2_LoadExternal(), or TPM2_ContextLoad()).

NOTE TPM2_Create() does not automatically load the object. After creation, the object needs to be explicitly loaded with TPM2_Load(), to load both the public and private portions, or with TPM2_LoadExternal() to load just the public portion.

The structure used for keys may be generalized for use on data objects if the access properties used for keys are suitable for access to these objects.

EXAMPLE A data blob can be defined so that access requires that some set of PCR has defined values, or an authorization value is needed for access. Such a data blob, called a Sealed Data Object, is managed in the same way that a key is managed. That is, the Sealed Data Object is loaded before being accessed, and the loaded blob could be context saved.

The TPM operates on other structures that are passed as parameters in specific commands. These structures are transient and are not stored in the TPM as identifiable entities after the command has completed.

Items loaded in the TPM are given handles to let them be referenced in subsequent commands.

11.6.4 Session Store

The TPM uses sessions to control a sequence of operations. A session may audit actions, provide authorizations for actions, or encrypt parameters passed in commands.

A session may be created as needed using one of the session creation commands. The session is assigned a handle at that time.

A TPM may be designed so that the RAM used for sessions is from a memory pool shared with the object store. It may also be designed so that the session store and object store are separated and dedicated.

11.6.5 Size Requirements

Random access memory (RAM) should be large enough to handle the transient state, sessions, and objects needed for completion of any implemented command. The minimums for the worst-case command in ISO/IEC 11889 are:

- two loaded entities (two keys, a key and a Sealed Data Object, or a hash/HMAC sequence and a key);

- three authorization sessions;
- an input buffer able to accommodate the largest command or an output buffer required for the largest possible response;

NOTE The largest command or response depends on the algorithms supported by the implementation.

- any vendor-defined state required for operation; and
- all defined PCR.

11.7 Non-Volatile (NV) Memory

The NV memory module stores persistent state associated with the TPM. Some NV memory is available for allocation and use by the platform and entities authorized by the TPM Owner.

TPM NV memory contains Shielded Locations and Shielded Location can only be accessed with Protected Capabilities.

If ISO/IEC 11889 is not explicit about storage of a parameter, that parameter may be in either RAM or NV, according to vendor preference.

If the NV memory of the TPM is subject to wear, then the TPM should detect whether the data being written to an NV memory location is the same as that currently stored and not perform the NV write if they are the same.

The OS or the platform may define a special NV data structure (an NV Index) in order to store persistent data values. NV memory may also be used persistently to store a loaded object. When a persistent object is referenced in a TPM command, the TPM may move that object into an object slot so it may be accessed more efficiently. The TRM needs to ensure that sufficient object memory RAM is available to allow this movement.

NOTE The movement occurs transparently.

A TPM capability indicates if the TPM is using Transient Object resources when a command references a persistent object. If so, the TRM needs to ensure that a Transient Object slot is available for each persistent object so referenced.

11.8 Power Detection Module

This module manages TPM power states in conjunction with platform power states.

All platform-specific TCG specifications that define the binding of the TPM to the platform should include a requirement that the TPM be notified of all power state changes.

The TPM supports only the ON and OFF power states. Any system power transition requiring the RTM to be reset also causes the TPM to be reset (_TPM_Init). Any system power transition that causes the TPM to be reset will also cause the RTM to be reset.

NOTE In most cases, the RTM will be a host CPU.

12 TPM Operational States

12.1 Introduction

This clause describes TPM operational states and state transitions.

12.2 Basic TPM Operational States

12.2.1 Power-off State

A hardware TPM is in the Power-off state when reset is being asserted or when no power is applied to the TPM. The TPM may internally generate reset by detecting low power, or reset may be provided by an external source.

It is possible to transition to the Power-off state from any other state because power can be lost at any time.

NOTE Uncontrolled transitions to this state are not shown in diagrams/descriptions because they would add unnecessary clutter and provide no additional understanding.

12.2.2 Initialization State

The TPM is placed in its initialization state when it receives the _TPM_Init indication. _TPM_Init is provided in a platform-specific manner. For a hardware TPM, the _TPM_Init is normally signaled by the de-assertion of the TPM's reset signal. It may also be signaled by an interface protocol or setting. For a software implementation, _TPM_Init may be a dedicated procedure call.

Regardless of how it is generated, _TPM_Init should coincide with a reset of the Roots of Trust for Measurement for which the TPM is the Root of Trust for Reporting. After _TPM_Init is indicated, the RTM should begin executing the Core Root of Trust for Measurement. It should not be possible to reset the TPM without resetting the RTM. It should not be possible to reset the RTM without resetting the TPM.

EXAMPLE If the TPM is a component on the PC's motherboard, _TPM_Init should coincide with a reset of the processor and chipset.

While in the Initialization state, the TPM performs basic initialization functions in preparation for accepting commands on the TPM interface. These functions are implementation dependent but, minimally, the TPM should perform validation of the TPM firmware necessary to execute the expected command. If the TPM is in Field Upgrade mode (FUM), the expected command is TPM2_FieldUpgradeData(). If the TPM is not in FUM, the expected command is TPM2_Startup().

After completing the initializations, the TPM waits for the next command and, if the command is not the expected first command, the TPM will return an error indicative of the mode. If the TPM returns an error, it will continue to wait for the expected first command.

NOTE 1 If the TPM is not in FUM, it returns TPM_RC_INITIALIZE. If the TPM is in FUM, it returns TPM_RC_UPGRADE.

NOTE 2 If TPM2_Startup()/TPM2_FieldUpgrateData() is not the first command to the TPM, it indicates failure of the system to properly enter the CRTM, and the reliability of TPM measurements are not assured. While it is possible to define a special failure mode that prohibits just PCR-related operations, it is expected to be infrequent enough not to warrant such a mode and, as shown in Figure 5, the TPM does not enter Failure Mode, if the first command is not TPM2_Startup().

When the TPM receives TPM2_Startup(), it becomes operational and is able to process other commands.

NOTE 3 For compliance with other standards, such as FIPS 140-2, it could be necessary for the TPM to validate the firmware associated with a command's execution before that command is executed. This includes the code associated with TPM2_Startup() and TPM2_FieldUpgadeData(). This validation could require use of a digital signature or message authentication code.

Occasionally, some TPM state may need to be retained over a power transition. This might occur if the platform is entering the Suspend state, where the preponderance of system state is retained. To allow the TPM to reflect this condition, system software may issue TPM2_Shutdown(TPM_SU_STATE) to the TPM.

TPM2_Shutdown() initiates an orderly shutdown of the TPM. The command's *startupType* parameter indicates the type of startup that is anticipated to follow and the type of data to be saved. For TPM2_Shutdown(TPM_SU_CLEAR), the amount of data saved to NV memory is relatively small, with considerably more information retained when TPM_SU_STATE is indicated.

12.2.3 Startup State

12.2.3.1 TPM2_Startup()

TPM2_Startup() transitions the TPM from the Initialization state to an Operational state. The command includes information from the platform to inform the TPM of the platform's operating state. TPM2_Startup() has two options: TPM_SU_CLEAR and TPM_SU_STATE. The operating state of a TPM after TPM2_Startup() is dependent on how the TPM was shut down and the selected startup option.

12.2.3.2 Startup Types

The following terms are used to refer to the different startup and shutdown operations:

- Startup(CLEAR) means TPM2_Startup(*startupType* == TPM_SU_CLEAR);
- Startup(STATE) means TPM2_Startup(*startupType* == TPM_SU_STATE);
- Shutdown(STATE) means TPM2_Shutdown(*startupType* == TPM_SU_STATE); and
- Shutdown(CLEAR) means TPM2_Shutdown(*startupType* == TPM_SU_CLEAR).

The combinations of Shutdown() and Startup() provide three unique methods of preparing the TPM for operation:

1) **TPM Reset** is a Startup(CLEAR) that follows a Shutdown(CLEAR), or a Startup(CLEAR) for which there was no preceding Shutdown() (that is, a disorderly shutdown). A TPM Reset is roughly analogous to a reboot of a platform. As with a reboot, most values are placed in a default initial state, but persistent values are retained. Any value that is not required by ISO/IEC 11889 to be kept in NV memory is reinitialized. In some cases, this means that values are cleared, in others it means that new random values are selected.
2) **TPM Restart** is a Startup(CLEAR) that follows a Shutdown(STATE). This indicates a system that is restoring the OS from non-volatile storage, sometimes called "hibernation". For a TPM Restart, the TPM restores values saved by the preceding Shutdown(STATE) except that all the PCR are set to their default initial state. This allows the TPM to record the boot sequence to ensure that the TCB is properly instantiated while allowing continued function of the restored OS.
3) **TPM Resume** is a Startup(STATE) that follows a Shutdown(STATE). This indicates a system that is restarting the OS from RAM memory, sometimes called "sleep." For sleep, the expectation is that the CRTM will perform the minimal actions required to make the system functional and then "return" to the running OS rather than rebooting it. TPM Resume restores all of the state that was saved by Shutdown(STATE), including those PCR that are designated as being preserved by Startup(STATE). PCR not designated as being preserved, are reset to their default initial state.

NOTE 1 The PCR are designated in a platform-specific specification.

If the TPM receives Startup(STATE) that was not preceded by Shutdown(STATE), then there is no state to restore and the TPM will return TPM_RC_VALUE. The CRTM is expected to take corrective action to prevent malicious software from manipulating the PCR values such that they would misrepresent the state of the platform. The CRTM would abort the Startup(State) and restart with Startup(CLEAR).

NOTE 2 The startup behavior defined by ISO/IEC 11889 is different than ISO/IEC 11889 (first edition) with respect to Startup(STATE). An ISO/IEC 11889 (first edition) device will enter Failure Mode if no state is available when the TPM receives Startup(STATE). This is not the case in ISO/IEC 11889. It is up to the CRTM to take corrective action if it the TPM returns TPM_RC_VALUE in response to Startup(STATE).

The TPM is required to validate the integrity of any NV values before those values are used before that state is used. This includes the state saved by TPM2_Shutdown(STATE) (see 12.2.4). When the TPM determines that some NV value required for proper TPM operation is not valid, the TPM will enter Failure Mode.

It is not specified when the validation of state specific to TPM Resume is to be checked. This gives implementation options that may be specified by a platform-specific specification or determined by the vendor.

The startup sequences are illustrated in Figure 5.

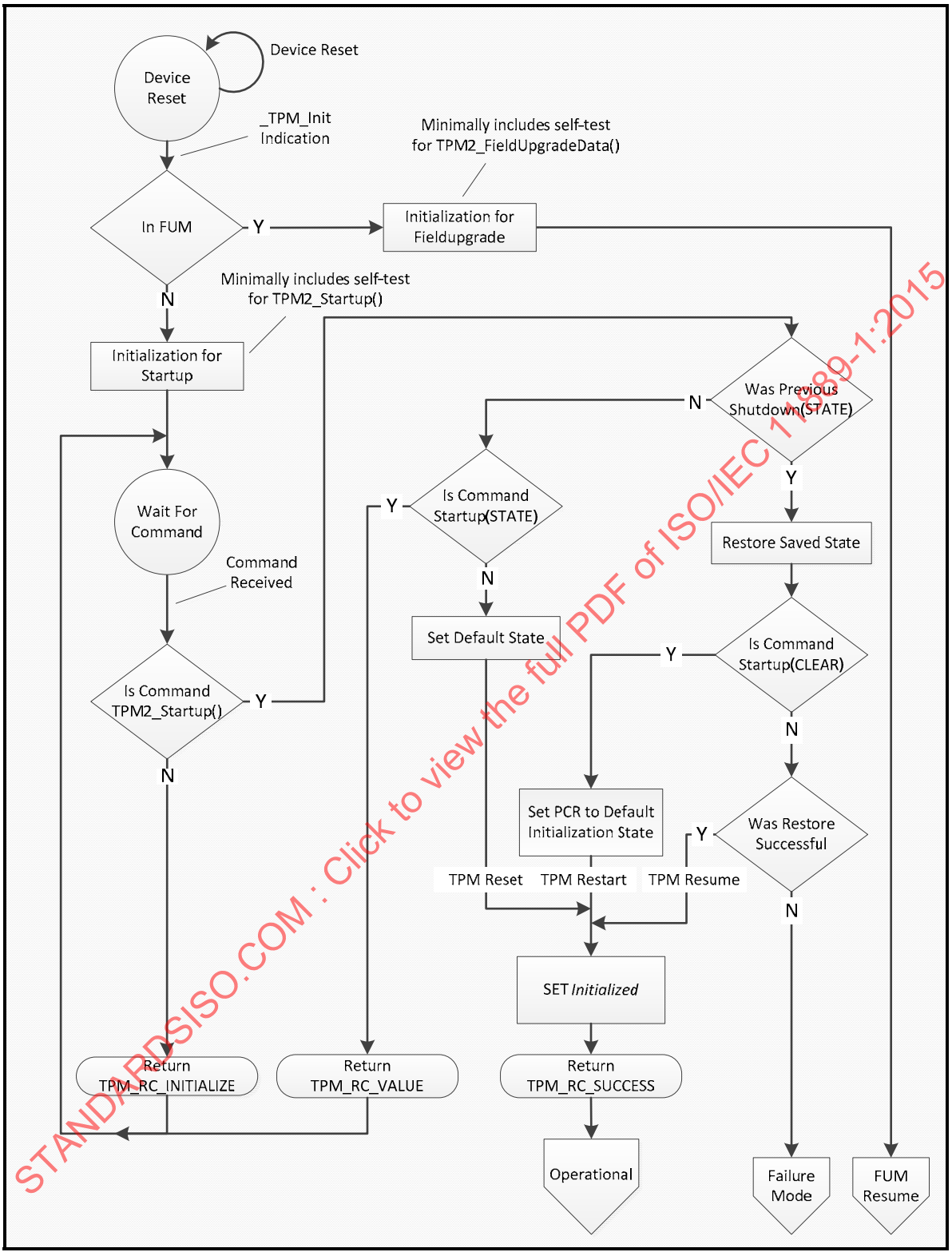


Figure 5 — TPM Startup Sequences

12.2.4 Shutdown State

TPM2_Shutdown() is used to prepare the TPM for loss of power. As with TPM2_Startup(), TPM2_Shutdown() has two options: TPM_SU_CLEAR and TPM_SU_STATE.

TPM2_Shutdown(TPM_SU_STATE) preserves the majority of the TPM operational state so that it may be restored on a subsequent TPM2_Startup(). TPM2_Shutdown(TPM_SU_CLEAR) preserves a minimal amount of state, mostly to ensure continuity of the TPM timing functions.

NOTE 1 The timing functions are specified in Clause 36.

The TPM preserves state data in NV memory. Data is copied from RAM into NV memory so that it is not lost when power is removed from the TPM. The amount of data copied to NV memory is largely implementation-dependent, but ISO/IEC 11889 indicates the state data that is required to be preserved. This state data is recovered in a subsequent TPM2_Startup(). The type of startup determines what parts of the saved state data is restored and what is discarded.

A shutdown is “orderly” if the TPM receives TPM2_Shutdown() before power is lost and if the state is not subsequently modified by a TPM command before the next _TPM_Init.

These commands will invalidate saved TPM state:

NOTE 2 This is not an inclusive list:

- TPM2_Clear(), TPM2_ChangeEPS(), TPM2_ChangePPS() – these commands invalidate saved contexts in the hierarchy. TPM2_Clear() invalidates preserved contexts in both the storage and endorsement hierarchies.
- TPM2_ContextSave() – context variables are modified by context save. Saving a session context changes the session context ID and its tracking state (saved or in memory). Saving an object context changes the object context ID.
- TPM2_ContextLoad() for a session – the context ID and tracking state (in TPM or context saved) for each active session should be retained across a TPM Restart or TPM Resume sequence. Saving or loading a session context changes the context ID or its tracking state. Saving or loading an object context need not invalidate a preserved context.
- Any command that modifies a PCR – regardless of the implementation, any change to a Resume PCR will invalidate the saved state. If the TPM implements TPM2_PolicyPCR() and uses a PCR generation counter, any PCR modification will change this counter value.

  EXAMPLE If a Shutdown(STATE) occurs but, prior to Startup(STATE), a TPM2_PCR_Event() is executed selecting a Resume PCR, then the preserved state is no longer valid, and Startup(STATE) is not valid until another Shutdown(STATE) occurs.

- Any command that modifies *Clock* or returns the value of *Clock*.

A TPM implementation may invalidate a preserved context on any command except TPM2_GetCapability().

12.2.5 Startup Alternatives

The description of the startup process above was given in terms of a command interface. In some systems, the TPM code is run in a special processor mode that provides the required isolation between the TPM state and any other program state. For these implementations, TPM2_Startup() may not be a command that is actually implemented. That is, the platform initialization may boot, validate the TPM code, and place the TPM in a state that is functionally equivalent to having run TPM2_Startup() on a discrete TPM component.

12.3 Self-Test Modes

If a command requires use of an untested algorithm or functional module, the TPM performs the test and then completes the command actions. When performing a self-test on demand, the TPM should test only those algorithms needed to complete the command. See Figure 6.

NOTE 1 It is preferable for the TPM to perform self-tests on untested algorithms and functional blocks as a background task to increase the likelihood that algorithms are tested before they are needed.

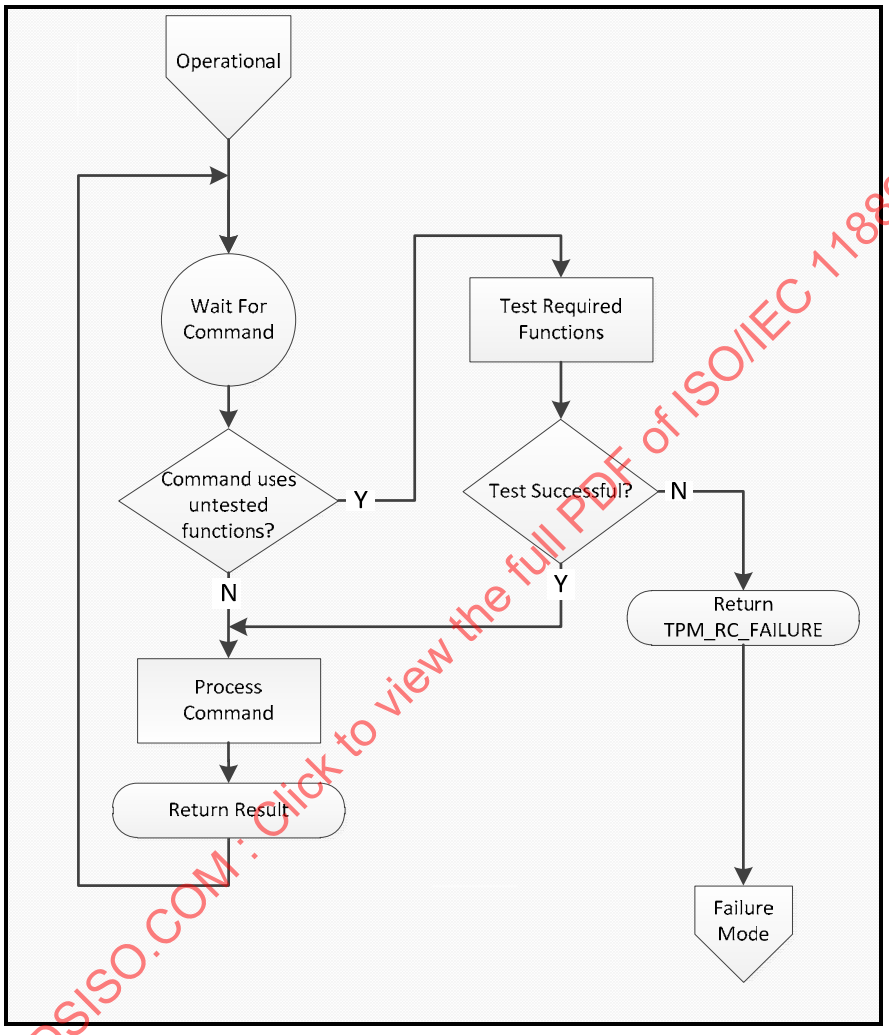


Figure 6 — On-Demand Self-Test

After sending TPM2_Startup(), the system may use either TPM2_SelfTest() or TPM2_IncrementalSelfTest() to cause the TPM to perform tests of untested algorithms. TPM2_SelfTest() may optionally cause the TPM to perform a full self-test of all algorithms and functional blocks. Once these commands are issued, the TPM returns TPM_RC_TESTING for any command that requires use of any testable function until all requested tests are completed.

NOTE 2 FIPS 140-2 requires that all power-on self-tests be complete before the TPM returns any value that depends on the results of a testable function. If compliance with FIPS 140-2 is required, any command that requires use of an untested function causes the TPM to operate as if TPM2_SelfTest(fullTest = NO) was received. The TPM returns TPM_RC_TESTING and continues to return TPM_RC_TESTING until all tests are complete. Alternatively, it could complete all tests and then complete the command. It could also return TPM_RC_NEEDS_TEST.

NOTE 3 Authenticated tests could be generated by attaching an audit session to TPM2_GetTestResult() and then using TPM2_GetSessionAuditDigest() to obtain the signature.

If any self-tests fail, the TPM goes into Failure mode and does not allow execution of any Protected Capabilities except TPM2_GetTestResult() and TPM2_GetCapability(). The TPM exits Failure mode when it receives _TPM_Init.

12.4 Failure Mode

If the TPM fails an internal test, it enters Failure mode. While in Failure mode, the TPM returns TPM_RC_FAILURE in response to any command except TPM2_GetTestResult() or TPM2_GetCapability() (See Figure 7). While in Failure mode, the TPM is only required to provide a limited number of property values. They are all in the set of TPM properties (TPM_CAP_TPM_PROPERTIES):

- TPM_PT_MANUFACTURER
- TPM_PT_VENDOR_STRING_1
- TPM_PT_VENDOR_STRING_2
- TPM_PT_VENDOR_STRING_3
- TPM_PT_VENDOR_STRING_4
- TPM_PT_VENDOR_TPM_TYPE
- TPM_PT_FIRMWARE_VERSION_1
- TPM_PT_FIRMWARE_VERSION_2

NOTE An implementation could return other property values.

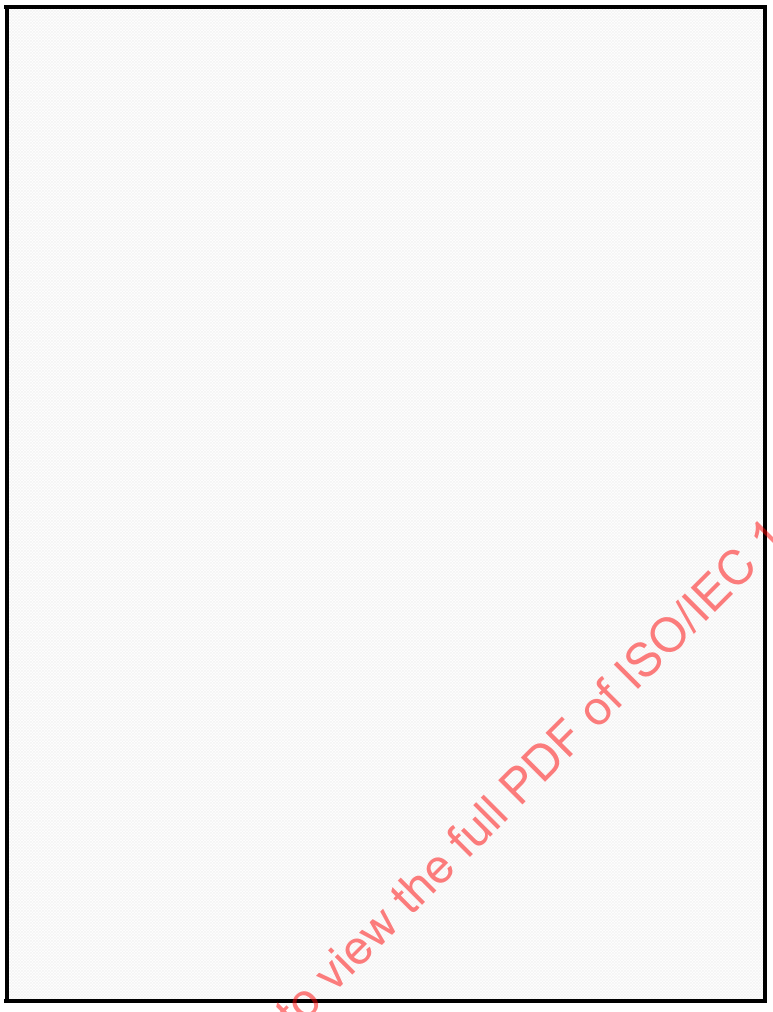


Figure 7 — Failure Mode Behavior

12.5 Field Upgrade

12.5.1 Introduction

ISO/IEC 11889 describes optional Protected Capabilities for upgrading the TPM firmware. The methods specified in ISO/IEC 11889 would allow the upgrade process to be handled in a standard way on TPMs from multiple vendors. The methods described here should not be viewed as limiting the vendor's options for implementation of their own, vendor-specific methods for upgrading the TPM firmware. However, the field upgrade methods chosen by the vendor should not be less robust than the methods specified in ISO/IEC 11889. In particular, the authorizations for the upgrade should be the same as the field upgrade commands in ISO/IEC 11889.

12.5.2 Field Upgrade Mode

ISO/IEC 11889 describes two optional upgrade methods: full and incremental. These terms do not refer to how much of the firmware in the TPM changes, but to how the upgrade is applied.

- For a full upgrade, the TPM stores in Shielded Locations all blocks of the firmware update. It makes no change to the executing firmware unless all the blocks are confirmed to be correct. The upgrade process may be interrupted or abandoned without affecting TPM functionality.

- For an incremental upgrade, firmware updates may be applied as each block is received. The TPM may not be fully functional if the upgrade process is abandoned.

The field upgrade process starts when the TPM receives a properly authorized TPM2_FieldUpgradeStart(). See Figure 6. That command contains the digest of a first block of the upgrade. If the next command is TPM2_FieldUpgradeData() and the digest of the data parameter (*fuData*) of the command matches the signed digest in TPM2_FieldUgradeStart(), the TPM accepts *fuData* as containing the upgrade data.

The TPM may buffer firmware update blocks and not change the firmware until its buffer is full. When a consequential change to the running firmware is made, the TPM enters Field Upgrade mode (FUM) and does not accept any command but TPM2_FieldUpgradeData() until the update is complete. See Figure 7. Before the TPM enters FUM

- it may accept other commands, and
- the update sequence may be abandoned by sending a zero-length upgrade data buffer. The TPM acknowledges that it has abandoned the field upgrade by returning TPM_ALG_NULL for *nextDigest*.

When the field upgrade process is complete, the TPM may either return to normal operation or enter a mode that requires _TPM_Init before normal TPM operations resume. The TPM vendor should determine if a reboot is required after the firmware update and cause the TPM to set the mode appropriately.

If the TPM is reset (_TPM_Init) while in FUM and the TPM is not able to revert to normal operation, three possibilities exist for recovery. The choice is determined by the digest of the first upgrade block provided to the TPM after _TPM_Init. The TPM may retain up to three digest values that it uses for comparison:

1) the digest of the first upgrade block of the current sequence to be used when the intent is to restart the current upgrade sequence from the start (called Digest C in Figure 8);
2) the digest of the first block of the firmware that was being replaced (called Digest P in Figure 8) to be used when the intent is to abort the upgrade and restore the previous firmware; and
3) the digest of the first upgrade block of the factory installed firmware (called Digest F in Figure 8) to restore the TPM to its factory state.

To enable option 2) above, the TPM may support TPM2_FirmwareRead() so that the software performing the upgrade can save a copy of the current TPM firmware in case the upgrade fails.

NOTE TPM2_FirmwareRead() might not be supported on a TPM even if the TPM can perform a field upgrade.

If _TPM_Init is received while the TPM is in FUM, then TPM Reset is required after the field upgrade completes, regardless of the nature of the firmware changes. This reset is required because the TPM does not accept TPM2_Startup() while in FUM, and the TPM will not reflect the state of the platform.

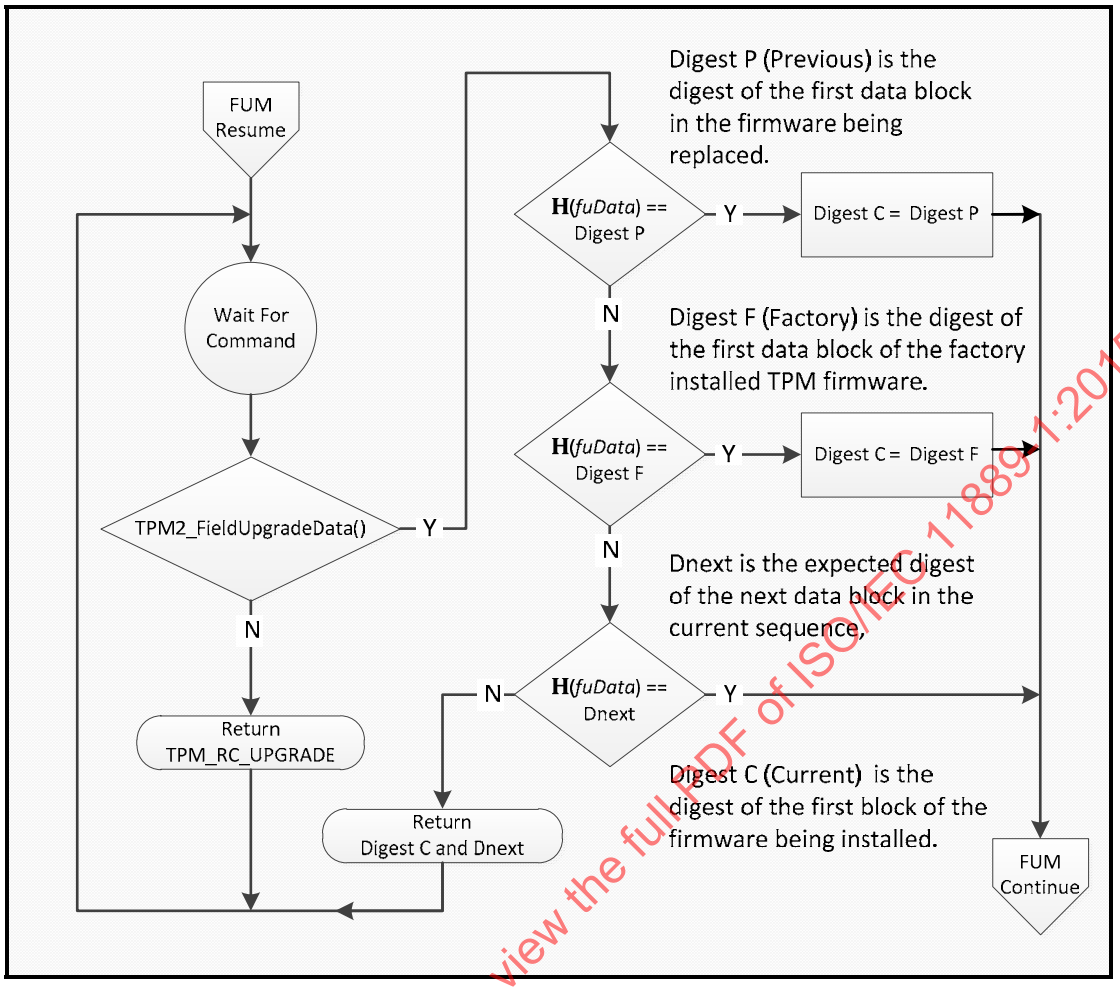


Figure 8 — Resuming FUM after _TPM_Init

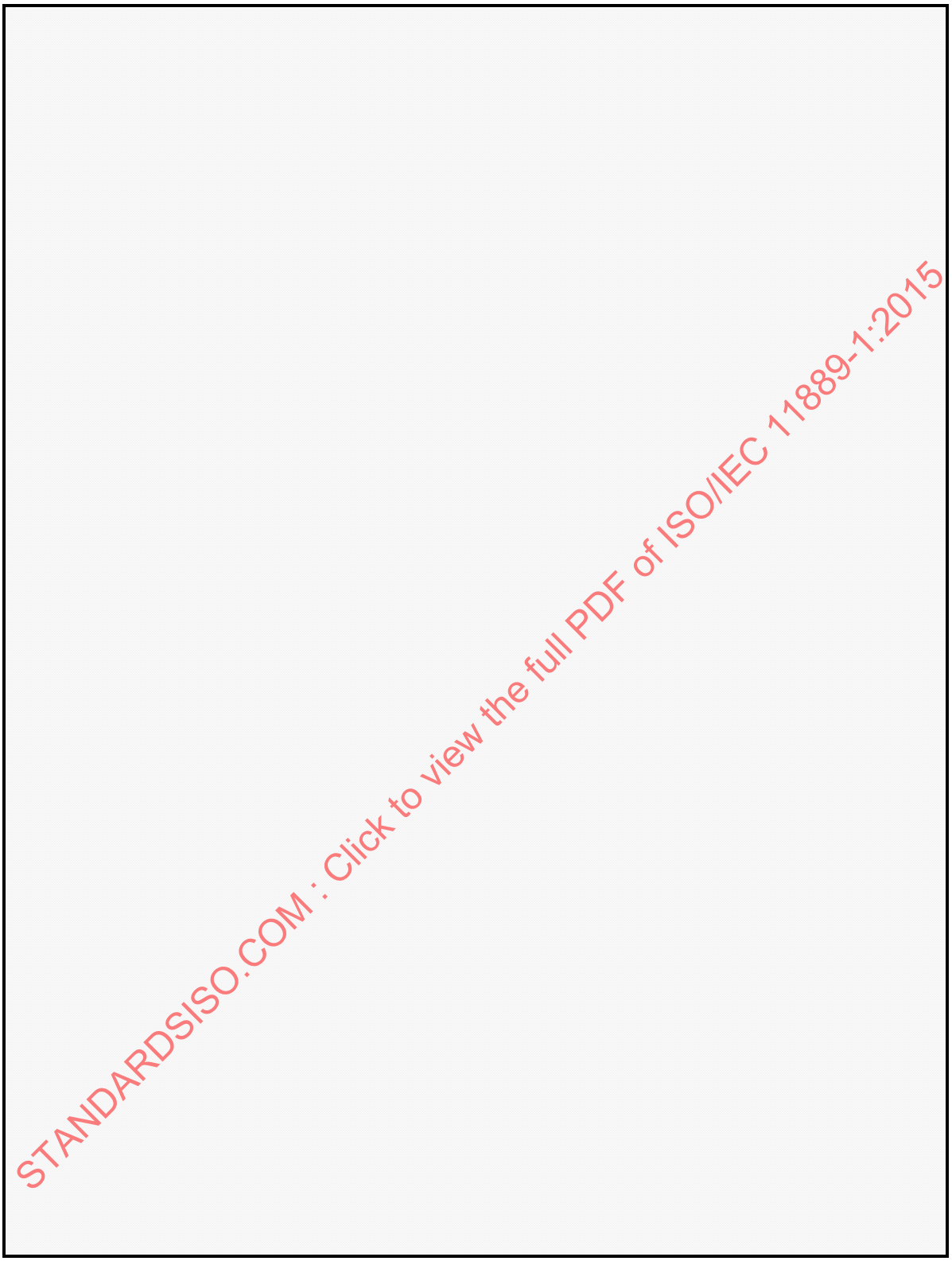


Figure 9 — Field Upgrade Mode

12.5.3 Preserved TPM State

A field upgrade may not cause exposure of any data that is specific to a TPM instance. This includes:

- Primary Seeds;
- Hierarchy *authValue*, *authPolicy*, and *proof* values;
- *lockoutAuth* and authorization failure count values;
- PCR authValue and authPolicy values;
- NV Index allocations and contents;
- Persistent object allocations and contents; and
- Clock.

In particular, if the TPM supports TPM2_FirmwareRead(), the returned data is not allowed to contain any data that is unique to the TPM instance.

A field upgrade should not cause the loss of any data that is specific to a TPM instance.

12.5.4 Field Upgrade Implementation Options

The method described above for management of a TPM field upgrade is intended for use in a TPM that is implemented as stand-alone component (that is, when the TPM is manufactured and sold as a component that is added to a platform). When the TPM is not a stand-alone component, other methods of field upgrade are possible and are not precluded by ISO/IEC 11889.

If other methods are used, the security of that method is the responsibility of the platform manufacturer.

13 TPM Control Domains

13.1 Introduction

Three entities control the TPM: the platform firmware, the platform Owner, and the Privacy Administrator. The Owner and Privacy Administrator are often the same entity. This control does not give these entities the ability to access user keys or data, but it does give them the ability to control selected TPM resources.

Each of the three entities has its own domain of control. Within that domain are TPM resources reserved to that entity. Each entity exercises its control over its domain by use of domain-specific authorization values.

The NV space defined by the platform firmware has an additional control, *phEnableNV*. When SET, NV space defined by the platform firmware is accessible. When CLEAR, it is inaccessible. This permits independent control of the platform firmware hierarchy and its NV space.

EXAMPLE The platform hierarchy can be disabled while still permitting access to platform firmware NV space.

13.2 Controls

The platform firmware, platform Owner, and Privacy Administrator each have an authorization value and an authorization policy to control some portion of the TPM, including a specific Primary Seed (see clause 14). The authorizations, policies, and Primary Seed for each domain are:

- *platformAuth/platformPolicy*/PPS for platform firmware;
- *ownerAuth/ownerPolicy*/SPS for the Owner; and
- *endorsementAuth/endorsementPolicy*/EPS for the Privacy Administrator.

Associated with each hierarchy is a logical switch (that is, an “enable”) that determines whether the hierarchy is enabled. These enables are *phEnable*, *shEnable*, and *ehEnable*.

When the enable for a hierarchy is SET (1) and ISO/IEC 11889 indicates that an action may be authorized with an authorization value, the corresponding policy is also allowed.

EXAMPLE When *phEnable* is SET and *platformAuth* is allowed, *platformPolicy* can also be used.

When the enable for a hierarchy is CLEAR, neither the corresponding *authValue* nor *authPolicy* may authorize operations.

The interaction of the two authorization types (value and policy) and the associated hierarchy enable are intended to provide a flexible set of controls. Table 2 shows the control combinations.

Table 3 shows the *authValue* as either being "Known" or "Unknown". These correspond to the enabled and disabled states for an *authValue*. When the authValue is known, it can be used for authorization but it cannot be used when the *authValue* is unknown. Since a zero-length string (Empty Buffer) is a valid, knowable *authValue*; the way to make the *authValue* unknown, and disable its use, is to set it to a large random number and then discard that number. Table 3 shows the *authPolicy* as either being "Set" or "Empty". These also correspond to the enabled and disabled states for an *authPolicy*. An *authPolicy* will have to match the value of a digest (*policyDigest*) in order for it to be a valid authorization. Since no digest has a zero length, setting the *authPolicy* to an empty buffer will disable use of the *authPolicy*. It is also possible to disable use of the *authPolicy* by setting it to any value that does not represent a known policy but the conventional way to disable use of *authPolicy* is to set it to an empty buffer (see 19.7 for the description of *policyDigest* generation and use).

.

Table 3 — Hierarchy Control Setting Combinations

hierarchy *enable*	***authValue***	***authPolicy***	**Description**
SET	Known	Set	The hierarchy is enabled, and objects in it may be loaded. Either *authValue* or *authPolicy* may manage resources related to the hierarchy.
SET	Unknown	Set	The *authValue* may be made unknown by setting it to a random value and then discarding the value. This prevents the *authValue* from being used. This combination is useful for keeping the hierarchy enabled but using a policy-based delegation scheme for managing hierarchy-related resources.
SET	Known	Empty	When the *authPolicy* is empty, it cannot match any *policyDigest* value so the use of *authPolicy* is disabled. This combination is most analogous to the control scheme of ISO/IEC 11889 (first edition), where an *authValue* (*ownerAuth*) is used to manage the resources of the single hierarchy supported by an implementation of ISO/IEC 11889 (first edition).
CLEAR	N/A	N/A	When an *enable* is FALSE, the corresponding *authValue* and *authPolicy* may not be used to authorize any TPM action.
EXAMPLE	An example of using hierarchy enable SET, an unknown authValue, and authPolicy SET is delegating control of creating Primary Objects in a hierarchy to one entity while delegating control of related NV resources to a different entity.		

TPM2_HierarchyChangeAuth() may change the *authValue* associated with a hierarchy but only if the hierarchy is enabled. Either the *authPolicy* or the *authValue* of a hierarchy may be used to authorize a change to the *authValue*.

13.3 Platform Controls

The platform firmware has overall control of the TPM and the availability of the TPM to the platform Owner or Privacy Administrator. The platform firmware is assumed to be provided by the platform manufacturer and performs the management of the hardware in preparation for use by an operating system (the operating system may be provided by a different vendor). In some systems, platform firmware runs after the OS is loaded. Often this firmware is required to ensure the safety of the system.

EXAMPLE Some systems have thermal properties that, if not managed properly, could lead to destruction of the system, and could even lead to the system becoming a fire hazard.

If the firmware is crucial to the safety of the system, the platform manufacture may design in a firmware update process that ensures that only firmware approved by the manufacturer for a specific machine is allowed to be loaded on the system. This firmware may use cryptography to validate the firmware update before it is loaded. The TPM has cryptographic functions that are similar or identical to the functions needed by the platform firmware for its management of the system. Rather than replicate those cryptographic capabilities, the platform firmware is given its own set of TPM resources for its use. Reuse of the TPM cryptographic capabilities by the platform is intended primarily as a cost savings.

The platform manufacturer decides if it is possible to disable use of the TPM by the platform. The method for disabling use of the TPM by the platform is platform-manufacturer specific.

The properties of the TPM required by the platform manufacturer need not match those of the Owner. The platform manufacturer decides what cryptographic algorithms are required to safeguard the platform. These algorithms may differ from the algorithms use by the Owner or the Privacy Administrator.

Platform controls allow the following operations not available to an ordinary TPM user:

- allocation of TPM NV memory;
- PCR configuration;
- control of the availability of any key hierarchies; and
- change of the PPS, SPS, and EPS and reset of associated authorization values and policy.

NOTE 1 This is not a comprehensive list. The uses of the platform controls are specified in ISO/IEC 11889-3. In ISO/IEC 11889-3, an authorization of a command that allows the use of the platform handle (TPM_RH_PLATFORM) indicates that the command accepts *platformAuth* or *platformPolicy*.

phEnable gates use of both *platformAuth/platformPolicy* and the PPS hierarchy, as specified in the previous clause. When *phEnable* is CLEAR, a _TPM_Init is required to SET it.

On any _TPM_Init, *phEnable* is SET to ensure that the platform may use the TPM during its initialization.

On TPM Reset or TPM Restart, *platformAuth* is set to an EmptyAuth, and *platformPolicy* is set to an Empty Policy.

NOTE 2 Platform controls are reset on TPM Restart because the BIOS goes through a full initialization and has no memory of any previous authorization values.

A *platformAuth/platformPolicy* may be used in TPM2_HierarchyControl() to SET or CLEAR *shEnable* or *ehEnable*.

13.4 Owner Controls

The TPM controls available to the Owner are a subset of those available to the platform. These include the

- allocation of TPM NV memory, and
- control of the availability of any storage hierarchies.

The s*hEnable* gates use of both *ownerAuth/ownerPolicy* and the SPS hierarchy, as specified in 13.2.

The s*hEnable* is SET on each TPM Reset, TPM Restart, or when the SPS is changed (TPM2_Clear()). The s*hEnable* may be CLEAR (TPM2_HierarchyControl()) using either Lockout Authorization or Platform Authorization. When *shEnable* is CLEAR, it may only be SET (TPM2_HierarchyControl()) if Platform Authorization is provided.

The o*wnerAuth* and *ownerPolicy* values are persistent. They are set to standard initialization values when the SPS is changed (TPM2_Clear()): *ownerAuth* is set to an EmptyAuth, and *ownerPolicy* is set to an Empty Policy. They may be explicitly changed by designated commands.

13.5 Privacy Administrator Controls

The Privacy administrator has control over the Endorsement Hierarchy and reporting of privacy-sensitive data.

The Privacy Administrator uses *endorsementAuth* and *endorsementPolicy* to exercise its control. The Privacy Administrator has a more limited domain of control than those of the platform firmware and the Owner. The cases when *endorsementAuth* or *endorsementPolicy* are required are:

- when creating Primary Objects in the Endorsement hierarchy, and

- when controlling the availability of the Endorsement hierarchy.

Other actions that may be considered to be privacy-sensitive require use of objects in the Endorsement hierarchy. The privacy administrator of the TPM is expected to manage the creation of objects in the Endorsement hierarchy to ensure that the use of those objects is in accordance with their privacy policy.

EXAMPLE Certification of a TPM object by the TPM produces a data structure that has data that could allow cross-correlation of the objects. This data is obfuscated unless the certifying key is in the Endorsement hierarchy.

The e*hEnable* gates use of *endorsementAuth/endorsementPolicy* and the EPS hierarchy, as specified in 13.1. It also gates use of the vendor-specific handles TPM_RH_AUTH_00 to TPM_RH_AUTH_FF. Additionally, when the SPS changes, the objects in the EPS hierarchy are flushed from the TPM, and new EPS objects (that is, Primary Objects) must be created.

NOTE 1 Clearing the hierarchy is necessary to ensure that the new Owner cannot abuse objects created by a previous one and so that objects belonging to the previous Owner cannot compromise the new one.

The e*hEnable* is SET on each TPM2_Startup(TPM_SU_CLEAR) (that is, TPM Reset or TPM Restart) or when the SPS is changed (TPM2_Clear()). The e*hEnable* may be CLEAR using either Endorsement Authorization or Platform Authorization. When *ehEnable* is CLEAR, it may be SET using Platform Authorization

NOTE 2 TPM2_HierarchyControl() will SET or CLEAR *ehEnable* if the proper authorization is provided.

The e*ndorsementAuth* and *endorsementPolicy* values are persistent. They are set to standard initialization values when the SPS (TPM2_Clear()) or EPS (TPM2_ChangeEPS()) are changed: *endorsementAuth* is set to an EmptyAuth, and *endorsementPolicy* is set to an Empty Policy. They may be explicitly changed by designated commands.

13.6 Primary Seed Authorizations

Use of a Primary Seed to create a Primary Object requires use of the authorization associated with that Primary Seed: Platform Authorization for the PPS, Owner Authorization for the SPS, and Endorsement Authorization for the EPS.

13.7 Lockout Control

A TPM is required to implement a lockout mechanism to protect against so-called “dictionary attacks,” where an attacker tries numerous authorization values until one succeeds. Dictionary attack protection is common for security devices that use human input for authorization. A human source of authorization likely has too little entropy to protect against an automated attack, so logic that prevents high-speed guessing of the values is required.

EXAMPLE Smartcards are an example of a security device that uses human input for authorization.

When the dictionary attack lockout is engaged, preventing use of some resources, it is helpful to have a secret value that resets lockout. The TPM stores the secret value as *lockoutAuth.* Alternatively, a policy *(lockoutPolicy)* can be used to reset lockout.

NOTE 1 The primary attack model for the dictionary attack begins when a system falls into the hands of a thief. The thief tries to recover data on the system by guessing the password used to protect a disk’s encryption keys. The dictionary attack logic defeats this attack by preventing the thief from making many guesses before the TPM locks out further attempts. When/if the system is returned to its rightful owner, that owner can enter the *lockoutAuth* value or satisfy *lockoutPolicy*, access the disk encryption keys, and return to normal operation.

NOTE 2 Unfortunately, dictionary attack logic is not forgiving of poor typing or a short memory. If someone types his or her password incorrectly due to clumsiness or poor memory, the dictionary attack logic might not differentiate this from an attack, so it locks the TPM. Lockout Authorization allows recovery from this situation.

The *lockoutAuth* value is reset to EmptyAuth and *lockoutPolicy* to the Empty Buffer when TPM2_Clear() is executed.

NOTE 3 TPM2_Clear() changes the SPS rendering all previously-created user objects inaccessible. There are, therefore, no keys for the dictionary attack logic to protect.

The l*ockoutAuth* value may be changed (TPM2_HierarchyChangeAuth()) only when its current value is provided. *LockoutPolicy* may be changed using TPM2_SetPrimaryPolicy().

Generally, dictionary attack protection is not applied to objects associated with the PPS or to NV Indexes defined using Platform Authorization. The platform firmware is expected to select a high-entropy value when setting the *platformAuth* after a TPM reset. Additionally, since Platform Authorization does not provide access to user data protected by the TPM, disclosure of *platformAuth* does not expose user secrets.

See 19.11 for full details on setting of parameters associated with dictionary attack logic and other aspects of the dictionary attack protection.

13.8 TPM Ownership

13.8.1 Taking Ownership

Taking ownership of a TPM is the process of inserting authorization values for the *ownerAuth*, *endorsementAuth*, and *lockoutAuth*.

A TPM that has been cleared (TPM2_Clear()) has its *ownerAuth*, *endorsementAuth*, and *lockoutAuth* values set to EmptyAuth and its *ownerPolicy*, *endorsementPolicy,* and *lockoutPolicy* values set to Empty Buffers. Different options exist for setting the values.

EXAMPLE 1 After the TPM is cleared, an OS could change the authorization and policy values and manage them on behalf of the platform Owner..

EXAMPLE 2 Platform firmware could provide a utility for the platform owner to directly set the authorization and policy values after the TPM is cleared.

The platform may prevent access to the hierarchies associated with Owner Authorization and Endorsement Authorization and prevent use of the TPM's persistent storage by the operating system and user applications. TPM cryptographic capabilities would still be available, and these could be used as if the TPM were a software cryptographic library.

13.8.2 Releasing Ownership

TPM2_Clear() clears the current Owner from the TPM. A persistent TPM control (TPMA_PERMANENT.*disableClear*) controls whether TPM2_Clear() is functional. If *disableClear* is CLEAR, then TPM2_Clear() may be authorized using either Platform Authorization or Lockout Authorization. If the control is SET, then TPM2_Clear() is not functional.

NOTE TPMA_PERMANENT.*disableClear* could be SET or CLEAR using platformAuth/platformPolicy, giving the platform the ability to enable execution of TPM2_Clear() when needed.

TPM2_Clear() instructs the TPM to:

- flush any transient or persistent objects associated with the SPS or EPS hierarchies (PPS objects are not affected);
- release any NV Index locations that do not have their TPMA_NV_PLATFORMCREATE attribute SET;
- set *shEnable* and *ehEnable* to TRUE;
- set *ownerAuth*, *endorsementAuth*, and *lockoutAuth* to an EmptyAuth;
- set *ownerPolicy, endorsementPolicy,* and *lockoutPolicy* to an Empty Policy;
- replace the existing SPS with a new value from the RNG; and
- recompute *shProof*, and *ehProof*.

14 Primary Seeds

14.1 Introduction

A Primary Seed is a large, random value that is persistently stored in a TPM; it is never stored off the TPM in any form. Primary Seeds are used in the generation of symmetric keys, asymmetric keys, other seeds, and proof values.

A Primary Seed generates Primary Objects using the methods specified in Clause 27.5. In brief, the caller provides the parameters of an object to be created, and the TPM uses these parameters and the Primary Seed in a key derivation function (KDF) to produce an object of the desired type. After the TPM generates a Primary Object, it uses the parameters of that object and the Primary Seed to generate a symmetric key to encrypt the sensitive portion of the object (that is, the private data and authorizations). It then returns the public portion and name of the object to the caller. The Primary Object may then be context saved and loaded like any other object. It may be stored persistently in the TPM's NV memory (TPM2_EvictControl()).

Primary Seeds generate only Primary Objects. All other objects use the random number generator of the TPM as the source of entropy for generating secrets in the object.

14.2 Rationale

The algorithm flexibility provided by ISO/IEC 11889 makes it possible for the TPM to support many different asymmetric key types. ISO/IEC 11889 (first edition) supported only the RSA algorithm with a limited number of commonly used parameters. The addition of ECC support significantly increases the number of parameters because curve parameters may vary based on application.

While this flexibility is a major benefit of ISO/IEC 11889, it creates new challenges for managing TPM Endorsement Keys (EKs) and EK certificates. As mentioned in 9.4.4.2, an EK is an identity for the Root of Trust for Reporting (RTR), and algorithm agility creates the possibility of having many identities for the same RTR, with each identity based on a different set of cryptographic algorithms.

One possible approach for handling many EKs and their associated certificates would be for the TPM manufacturer to have the TPM create EKs for many key parameters and store them on the TPM; in this way, a key with the correct parameters would be available in most situations. The TPM vendor could then create one or more certificates for those keys. However, this approach would require a prohibitive amount of NV memory to store all the key pairs and associated parameters. The approach taken in ISO/IEC 11889 allows certification of a large number of EKs with different parameters without requiring that any of them be stored in persistent TPM memory.

The mechanism of ISO/IEC 11889 uses a persistent, randomly generated seed value from which EKs are derived. The derivation process lets the TPM generate a different EK for each set of key parameters. As long as the seed value does not change, the same key parameters generate the same EKs.

The typical use of this EK generation approach is as follows: The TPM manufacturer or the platform manufacturer has the TPM create a new Endorsement Primary Seed (EPS) and then generate key pairs based on sets of input parameters and that EPS. The TPM retains the generated keys. Combinations of key parameters should be chosen to ensure that likely TPM users would find a combination to suit their needs. The manufacturer then generates one or more certificates for the generated public keys and then ships the TPM/system with no EK pair stored on it. The system owner decides which key types are needed, and the parameters for those types are entered into the TPM. If the parameters are the same as those used by the manufacturer, the TPM generates the same key pair. The system owner then has an EK with its certificate. A Primary Key does not leave the TPM so the owner has a choice to make. They may either re-create the EK whenever it is needed or tell the TPM to save the EK in persistent memory.

The seed key concept may be applied to two other TPM key hierarchies: one used by platform firmware, and one used for the owner's Storage hierarchy. The Endorsement Keys (EK) are generated from the Endorsement Primary Seed (EPS), platform keys from the Platform Primary Seed (PPS), and Storage Root Keys (SRK) from the Storage Primary Seed (SPS). Each seed value has a different life cycle, but the way it seeds the associated hierarchies is approximately the same.

This approach allows multiple storage hierarchies with differing security properties, as needed by various applications, without requiring that all of the SRKs occupy persistent TPM memory. An SRK may be made persistent in TPM NV memory if required by the application.

This scheme is also used in support of the Platform hierarchy for implementation simplicity.

14.3 Primary Seed Properties

14.3.1 Introduction

A Primary Seed is required to have at least twice the number of bits as the security strength of any symmetric or asymmetric algorithm implemented on the TPM.

EXAMPLE 1 RSA2048 is considered to have a security strength of 112 bits. If it were the strongest algorithm on the TPM, then the size of an associated Primary Seed would be at least 224 bits.

EXAMPLE 2 If AES256 were implemented, the Primary Seed would be 512 bits even if: (1) the desired security strength is 196 bits, and (2) AES256 is used only for convenience, as is the case with Suite B.

A different authority controls each Primary Seed. In normal use, Primary Seeds are expected to have different lifetimes.

14.3.2 Endorsement Primary Seed (EPS)

The EPS is used to generate EKs and is the basis of the RTR's identity.

The TPM creates an EPS whenever it is powered on and no EPS is present. TPM2_ChangeEPS() may change the EPS (replace it with a new EPS), but this requires authorization by Platform Authorization.

The TPM manufacturer may inject an EPS and, under controlled conditions, compute the asymmetric EKs that the TPM would generate given specific input parameters. Only the TPM vendor may inject an EPS.

When an EPS is replaced, all objects in the Endorsement Hierarchy are invalidated, and certificates associated with the EKs generated from that EPS are no longer useful. This means that certificates for new EPS-based EKs may be needed. The environment in which this process occurs may not provide assurance that the EKs are generated from a genuine TPM. To support recertification in such an environment, the TPM allows cross certification of keys between the Platform hierarchy and the Endorsement hierarchy under control of the platform firmware. Cross certification allows a chain of trust to be maintained as the seeds are changed.

When a platform enters the distribution channel, it is expected to have a certificate for at least one EK for the TPM on that platform.

Either e*ndorsementAuth* or *endorsementPolicy* is required to use the EPS for creation of a Primary Object in the Endorsement hierarchy.

14.3.3 Platform Primary Seed (PPS)

The PPS is used to generate the hierarchy controlled by platform firmware. The hierarchies derived from this seed are for exclusive use by platform firmware and should not be made available to user-installable software.

EXAMPLE An example of user-installable software is an OS and applications.

NOTE 1 The platform firmware may be changed because of actions by a person with possession of the platform, but that is not included in the definition of user-installable software.

The TPM creates a PPS whenever it is powered on and no PPS is present. TPM2_ChangePPS() may change the PPS (replace it with a new PPS), but this requires authorization by Platform Authorization.

A PPS may be injected but only by the TPM manufacturer.

Platform Authorization is required to use the PPS to create a Primary Object in the Platform hierarchy.

The authorization for use of objects in the PPS hierarchy should use a policy containing a reference to *platformAuth* and not be based on a key-specific authorization value.

NOTE 2 The TPM does not enforce this imperative.

NOTE 3 A simple way to achieve this control is to create a policy that references *platformAuth* in a TPM2_PolicySecret(). If the only component of the policy is TPM2_PolicySecret() referencing TPM_RH_PLATFORM, the policy could be the same for all objects in the Platform hierarchy and for all platforms that implement the chosen policy hash.

14.3.4 Storage Primary Seed (SPS)

The SPS is used to generate hierarchies controlled by the platform owner. This seed generates the keys that serve as Storage Root Keys for normal OS and application use.

The TPM creates the SPS whenever it is powered on and no SPS is present. TPM2_Clear() may be used to change the SPS if the TPM owner wants to ensure that no previously generated keys in the Storage hierarchy may be used in the future.

Changing the SPS invalidates all objects in the Storage Hierarchy and they cannot be recreated. Changing the SPS also invalidates all objects in the Endorsement Hierarchy and only the Primary Objects in the Endorsement Hierarchy may be recreated.

14.3.5 The Null Seed

The Null Seed is set to a random value on every TPM reset. The Null Seed can be used to generate hierarchies (primary objects and children of primary keys) that are only usable until the next TPM reset.

Objects in the null-hierarchy cannot be made into persistent objects. However, in other respects objects in this hierarchy behave like objects in the other hierarchy.

14.4 Hierarchy Proofs

The TPM uses a proof value to prove that it created or checked an externally provided value. A proof value is associated with a hierarchy and is statistically unique. The proof values are used in tickets. The tickets use the hierarchy-specific proof values. A ticket may not be used when its associated hierarchy is disabled.

EXAMPLE 1 The TPM can validate asymmetrically signed data. After doing so, it produces a ticket that is an HMAC over the signed data, with the HMAC key being a proof value. This proves to the TPM that it has already checked the asymmetric signature so it does not have to do so again. Subsequently, when the TPM needs to check that the data was properly signed, it can use symmetric cryptography (a hash) rather than asymmetric cryptography to validate the signature.

EXAMPLE 2 When the TPM performs TPM2_ContextSave() on an object in the Storage hierarchy, it can include the Storage hierarchy proof (*shProof*) in the object's integrity value. When the SPS is changed, *shProof* will change so that the saved contexts cannot be reloaded.

A Platform hierarchy proof (*phProof)*, used for objects associated with the Platform hierarchy. *phProof* changes when the PPS changes. An *shProof*, used for the Storage and Endorsement hierarchies, changes when the SPS changes.

NOTE 1 It is possible to create objects in the Endorsement Hierarchy that are not Primary Objects. Those Ordinary Objects are considered to belong to a specific TPM Owner. A change of the SPS indicates a change of Owner for the TPM. Inclusion of *shProof* in the protection of Ordinary Objects in the Endorsement Hierarchy insures that those Objects will be deleted when the Owner changes.

A proof is a value that may be kept in permanent storage on the TPM or it may be regenerated from the PPS or SPS on each boot or as needed. A proof value is never stored off the TPM in any form. Hierarchy proof values are only used as an HMAC key if the result of the computation is stored off the TPM. A hierarchy proof value may be used in other computations as long as the result of the computation does not leave the TPM.

EXAMPLE 3 Saved contexts and tickets are examples of hierarchy proof values used as HMAC keys when the result of the computation is stored off the TPM.

The TPM should produce proof values that are the larger of either

- the size of the largest digest produced by any hash algorithm implemented on the TPM, or
- twice the size of the largest symmetric key supported by the TPM.

EXAMPLE 4 If the TPM implements SHA384 and AES256, the proof value will have a size of 512 bits.

NOTE 2 According to SP800-57, the security strength of SHA256 in an HMAC function equals 256 bits. Since security strength is not improved when the key size is larger than the digest size, the recommendation for proof size provides the appropriate strength when the TPM is implementing balanced algorithm sets. A TPM using SHA256, ECC256, and AES128 is balanced, and the proof value is 256 bits.

15 TPM Handles

15.1 Introduction

TPM resources are referenced by handles that uniquely identify a resource that occupies TPM memory — either RAM or NV. A handle is a 32-bit value. Its most significant octet identifies the type of referenced resource. At any given instant, its low-order 24 bits identify a unique resource of that type. The actual resource identified by the low-order 24 bits may change with time.

A specific handle value may refer to only one TPM-resident resource at a time.

15.2 PCR Handles (MSO=00_{16})

To reduce confusion, PCR are assigned handles that have the same values as in ISO/IEC 11889 (first edition). A PCR handle is an index into an array of PCR. A PCR's index and handle value are the same.

15.3 NV Index Handles (MSO=01_{16})

An NV Index is associated with a persistent TPM resource created by TPM2_NV_DefineSpace().

15.4 Session Handles (MSO=02_{16} and 03_{16})

The TPM assigns session handles when an authorization session is started (TPM2_StartAuthSession()). An HMAC session is assigned a handle with an MSO of 02_{16} and a policy session is assigned a handle with an MSO of 03_{16}. Each authorization session handle is associated with a unique context that may exist in only one place at a time: either on the TPM in a Shielded Location, or in a saved context as a Protected Object. The handle remains associated with the session as long as the session exists and does not change when the session is context-saved and reloaded.

The low order 3 octets of each session handle are unique. They are assigned interchangeably to HMAC or policy sessions but to only one at a time.

EXAMPLE 1 If a policy session has a value of 03 00 00 01_{16}, then an HMAC session with a value of 02 00 00 01_{16} will not be assigned at the same time.

NOTE 1 The policy and session handles are assigned from a common pool of handle values.

When TPM2_GetCapability() is used to obtain a list of sessions that are currently loaded on the TPM, the caller would use a handle with an MSO of 02_{16}. While this would normally be an HMAC handle reference, the TPM will respond with a list that includes both HMAC and policy sessions. The handles will be returned in ascending order of the low-order three octets.

EXAMPLE 2 A list of loaded handles returned by the TPM in response to a TPM2_GetCapability(*capability* = TPM_CAP_HANDLES, *property* = 02 00 00 00_{16}), the TPM might return the list: 02 00 00 02_{16}, 03 00 00 04_{16}, and 02 00 00 05_{16}.

When TPM2_GetCapability() is used to obtain a list of sessions that are active but not on the TPM, the caller would use a handle with an MSO of 03_{16} which normally would reference a policy session. The TPM will respond with a list of session handles that are in use, but not on the TPM. Since the TPM does not keep a record of whether the saved session context was an HMAC or policy session, all of the handles in the list will have an MSO of 02_{16}.

The TPM is required to maintain a list of all, currently assigned session handles as well as the correct "version number" for any saved session contexts.

NOTE 2 the "version number" is how the TPM prevents replay of an authorization.

When an authorization session is no longer needed, TPM2_FlushContext() may be used to delete all context associated with the session from TPM memory (see 30.6). The session handle for this command may use an upper octet of either 02_{16} or 03_{16}.

NOTE 3 Flushing a session context deletes any data in the TPM relating to the context and frees the handle associated with that context and invalidates the version number of any saved context.

NOTE 4 An alternative method of flushing a session context exists that is not available for other entities. On the last use of the session, the caller may indicate (in one of the session attributes) that the session is no longer needed. If the command completes successfully, the TPM will complete the response computations for the session and delete the session context from TPM memory (see 18.6.4).

All session contexts in TPM memory are flushed on any TPM2_Startup(). The saved session contexts remain valid until a TPM Reset.

15.5 Permanent Resource Handles (MSO=40_{16})

Fixed resource handles refer to Shielded Locations that are always associated with the same handle.

EXAMPLE Examples of these resources are Owner, Platform, and Endorsement hierarchy controls. These resources have handles with an MSO of 40_{16}.

15.6 Transient Object Handles (MSO=80_{16})

The TPM assigns Object handles when an Object is loaded or when the Object's persistence is changed (TPM2_EvictControl()). Transient objects in TPM RAM have handles with an MSO of 80_{16}; they may have a different value for the three LSOs each time the Object is used. This is because the Object's context may have been swapped out and the TPM assigned a new handle when the object was swapped back in. The TRM ensures that the handle references the correct object.

All Transient Objects are flushed from TPM memory on any TPM2_Startup(). A loaded Transient Object context may be flushed from TPM memory using TPM2_FlushContext() and indicating the handle of the loaded context to be flushed.

15.7 Persistent Object Handles (MSO=81_{16})

TPM2_EvictControl() may make a Transient Object into a Persistent Object. A Persistent Object, placed in the TPM's NV memory, is not cleared by a TPM2_Startup().

Making an Object persistent requires either Platform Authorization or Owner Authorization.

When the TPM changes a Transient Object to a Persistent Object, the caller indicates the handle to be assigned to the Persistent Object. The MSO of the handle is required to be 81_{16}. The next most significant bit is required to be CLEAR if the authorization is provided using Owner Authorization and SET if the authorization is provided using Platform Authorization. If the handle is not already in use, and space is available, a persistent copy of the Object is created and assigned the handle provided by the caller. This handle always references the same Persistent Object as long as it remains persistent. The handle assigned to a Persistent Object may be assigned to a new Persistent Object if the first Object is deleted from persistent storage.

16 Names

The Name of an entity is its unique identifier. The handle associated with an object may change due to context management (TPM2_ContextSave() / TPM2_ContextLoad()), but the Name of an object remains constant. The Name associated with an NV Index will change based on changes to the attributes of the Index.

EXAMPLE 1 When an NV Index is initially defined, it will have a Name for an Index with TPMA_NV_WRITTEN CLEAR. After the Index is written, the Name will change to reflect that TPMA_NV_WRITTEN is SET for the Index.

When an NV Index becomes locked (TPMA_NV_WRITELOCKED or TPMA_NV_READLOCKED is SET), the Name of the NV Index changes. This has two implications:

1. The caller should read the NV public area and calculate the Name before using it in an HMAC authorization calculation. Otherwise, an invalid authorization may trigger the dictionary attack protection depending on TPMA_NV_NO_DA

2. The TPM must check access control before checking authorization.

   EXAMPLE 2 The TPM ought to reject a read to a read locked NV Index before doing an authorization check that might trigger the dictionary attack protection.

The method of computing the Name for an entity varies according to the entity type that is the MSO of the handle. Table 3 shows the method and the handle's MSO for different entity types.

When the computation of a Name uses a hash algorithm, the algorithm identifier is included in the Name structure. If the Name is a handle, the Name is only the handle value.

Table 4 — Equations for Computing Entity Names

MSO of Handle	Entity Type	Equation for Computing the Name
00_{16}	PCR	$Name := handle$ No hash is performed on the handle to produce the name and the name is only the size of the handle.
02_{16}	HMAC Session	
03_{16}	Policy Session	
40_{16}	Permanent Values	
01_{16}	NV Index	$Name := nameAlg \mid\mid \mathbf{H}_{nameAlg}(handle \rightarrow nvPublicArea)$ where $nameAlg$ — algorithm used to compute $Name$ $\mathrm{H}_{nameAlg}$ — hash using the $nameAlg$ parameter in the NV Index location associated with $handle$ $nvPublicArea$ — contents of the TPMS_NV_PUBLIC associated with $handle$

MSO of Handle	Entity Type	Equation for Computing the Name
80_{16}	Transient Objects[1]	$Name := nameAlg \mid\mid \mathbf{H}_{nameAlg}(handle \rightarrow publicArea)$ where $nameAlg$ algorithm used to compute $Name$ $\mathrm{H}_{nameAlg}$ hash using the $nameAlg$ parameter in the object associated with $handle$ $publicArea$ contents of the TPMT_PUBLIC associated with $handle$
81_{16}	Persistent Objects	
NOTE The Name of a sequence object is an Empty Buffer (see 32.4.5).		

When an object is created, a "template" for the public area is used to define the properties for the new object. That template has the structure of an object's public area. The Name of a public area template is computed in the same way as the Name of a Transient Object.

17 PCR Operations

17.1 Initializing PCR

All platform configuration registers (PCR) are reset to their default initial condition on TPM Reset and TPM Restart. Some PCR may be designated as being preserved by TPM Resume. Those that are preserved are restored to the state that they had at the last TPM2_Shutdown(STATE) operation. When TPM2_Startup() completes successfully, PCR that are not designated as being preserved by TPM Resume will be in their default initial condition.

If allowed by its attributes, a PCR may also be reset by TPM2_PCR_Reset() or by a Dynamic Root of Trust (D-RTM) sequence (see 34.2). PCR attributes are defined in a platform-specific specification. They determine the reset value of a PCR as well as the localities required to perform the reset.

A PCR's default initial condition may be either all bits CLEAR, all bits SET, or an indicator the first measurement came from an H-CRTM.. Other platform types may use other means of identifying the locality of the access. A platform-specific specification should indicate, for each defined PCR, the option that applies.

EXAMPLE 1 A platform-specific specification might designate that the default initial condition for PCR[0-16] is all zeros, and for PCR[17-20], it is all ones.

EXAMPLE 2 A platform-specific specification might designate that the default initial condition for PCR[0] is the locality indicator and that PCR[1-16] have an initial condition of all zeros.

NOTE The locality indicator is an integer value between 0 and the maximum locality implemented on a TPM. Currently, the maximum hardware locality is 4. In a TPMA_LOCALITY, a locality of four would be represented by the octet $0001\ 0000_2$. When encoded for a PCR initial value, locality 4 would be represented by the octet $0000\ 0100_2$.

EXAMPLE 3 A virtual TPM might use a unique identifier for each of the software entities that might access it. If specific software is associated with a specific PCR, then the reset value of that PCR could be the unique identifier of the software that is allowed to change it.

TPM2_PCR_Reset() requires that the proper authorization be provided for the operation (see 17.7).

17.2 Extend of a PCR

Other than reset, described above, the only way to change a PCR value is to Extend it. The Extend operation on a PCR is defined as

$$PCR_{new} := \mathbf{H}_{alg}\,(PCR_{old} \,||\, digest) \tag{14}$$

After each Extend, the PCR value is unique for the specific order and combination of digest values that were Extended.

Except for D-RTM, authorization is required to extend a PCR (see 17.7).

17.3 Using Extend with PCR Banks

TPM2_PCR_Extend() has a handle to indicate the PCR to Extend and the data to be Extended. Extended data is a structure that contains one or more digests along with the algorithm identifier for the digest(s). Each digest is Extended to the PCR bank that has the same algorithm. If no digest data is provided for one of the PCR banks, no change is made to the PCR in that bank.

The TPM should perform the following operation for each algorithm in which *pcrNum* is defined:

$$PCR.digest[pcrNum][alg]_{new} := \mathbf{H}_{alg}\,(PCR.digest[pcrNum][alg]_{old} \,||\, digest) \quad (15)$$

where

$\mathbf{H}_{alg}$	hash function using the algorithm associated with the PCR instance
$PCR.digest$	digest value in a PCR
$pcrNum$	PCR numeric selector
alg	PCR algorithmic selector
$digest$	digest part of the list entry that has the same algorithm identifier as the PCR bank

EXAMPLE If a TPM supports three PCR banks (such as, SHA-1, SHA256, and SHA512), then an Extend to PCR[2] with a SHA-1 digest and SHA256 digest would be Extended to PCR[2] in the SHA-1 bank, and the SHA256 digest would be Extended to PCR[2] in the SHA256 bank. There would be no change to any PCR in the SHA512 bank.

17.4 Recording Events

An alternative way to record log entries is to input the full log entry to the TPM rather than performing the digests outside the TPM. This performs a hash on the log entry for each of the hash algorithms associated with a PCR bank. Events no larger than 1024 octets may use TPM2_PCR_Event(). Events exceeding 1024 octets may use the sequence commands: TPM2_HashSequenceStart(), TPM2_SequenceUpdate(), and TPM2_EventSequenceComplete()).

TPM2_PCR_Event() and TPM2_EventSequenceComplete() return a list of tagged digests. The digests are the digests of the event data using the hash algorithm of each bank.

EXAMPLE For a TPM implementing two banks (such as, SHA256 and SM3), the event commands return a list of two tagged digests.

TPM2_EventSequenceComplete() requires that proper authorization be provided (see 17.7).

Recording of an event may also occur as the result of a _TPM_Hash_Start/_TPM_Hash_Data/_TPM_Hash_End sequence (an *H-CRTM Event Sequence*). The indications for the H-CRTM sequence come from the TPM interface and not through the command buffer. On receipt of _TPM_Hash_Start, the TPM will created an Event Sequence context. If no object context space is available when the TPM receives _TPM_Hash_Start, the TPM will flush a context (vendor's choice) in order to create the Event Sequence context. _TPM_Hash_Data is used to update the H-CRTM Event Sequence context and _TPM_Hash_End completes the sequence. The digest or digests computed during the H-CRTM Event Sequence will be extended into the PCR designated by the relevant platform-specific specification. A platform-specific specification may allow an H-CRTM Event Sequence before or after TPM2_Startup(). An H-CRTM Event prior to TPM2_Startup() affects PCR[0]. After TPM2_Startup(), an H-CRTM Event affects PCR[17].

17.5 Selecting Multiple PCR

TPM2_PCR_Event() implicitly selects all PCR with the same index. Some commands allow the selection of multiple PCR in different banks. When a command allows multiple PCR to be selected, a list of selectors is used. Each entry in the list consists of an algorithm ID followed by a bit array. Each bit in the bit array corresponds to one PCR. If a bit is SET, then the indicated PCR in the bank corresponding to the algorithm ID is selected.

EXAMPLE 1 TPM2_PCR_Read(), TPM2_Quote(), and TPM2_PolicyPCR() are commands allow the caller to make arbitrary selections of PCR in multiple banks.

The bit correspondence to PCR is that the bit corresponding to PCR[*n*] is the (*n* **mod** 8) bit in the ⌊*n*/8⌋ octet of the array.

EXAMPLE 2 An array to select PCR[0] and PCR[13] in a TPM with 16 PCR would be 01 20_{16}. The bit for PCR[0] is the 0 **mod** 8 = 0^{th} bit in the ⌊0/8⌋ = 0^{th} octet (the octet with the 01_{16} value) and the bit for PCR[13] is the 13 **mod** 8 = 5^{th} bit in the ⌊13/8⌋ = 1^{st} octet (the octet with the 20_{16} value).

The list of selectors is processed in order. The selected PCR are concatenated, with the lowest numbered PCR in the first selector being the first in the list and the highest numbered PCR in the last selector being the last.

TPM2_PCR_Read() returns a list of PCR values that correspond to the PCR selected in the selector list. TPM2_Quote() and TPM2_PolicyPCR() digest the concatenation of PCR.

It is not an error for the PCR selection to indicate a PCR that is not implemented in a bank. No value is included in the concatenation of PCR for an unimplemented PCR. It is an error if the algorithm ID selects a hash algorithm that is not implemented.

17.6 Reporting on PCR

17.6.1 Reading PCR

TPM2_PCR_Read() reads the current values of a selection of PCR. For this command, the caller indicates a list of PCR to be read using a PCR selection structure. This structure is an array of lists. Each array entry has a hash identifier and a bit field. The hash identifier indicates the bank of PCR, and the bit field indicates the PCR being selected in the bank.

In the response, the TPM provides a PCR selection structure and a list of PCR values. The PCR selection structure indicates the PCR that are present in the return structure. The size of the requested return data structure may not fit in the available TPM output buffer. In that case, the list of PCR values is truncated, and the response PCR selection structure indicates the PCR that were returned. If the returned structure does not contain all of the PCR, the caller may modify the selection structure and make another read request to get additional PCR values.

Since the PCR may change between the calls to collect the full set of PCR of interest, the TPM returns a counter that increments on most invocations of TPM2_PCR_Extend(), TPM2_PCR_Event(), TPM2_EventSequenceComplete(), or TPM2_PCR_Reset() (see 17.9 for exemptions). If this counter value changes between calls, the sequence may need to be repeated until the desired PCR are all returned with no change to the counter value.

17.6.2 Attesting to PCR

In some cases, it is necessary for selected PCR to be in a specific state. When indicating that state, it is not desirable to have to list the contents of each PCR. Instead, a digest of a concatenation of PCR (a composite PCR digest) will indicate the current contents of all of the PCR of interest.

The PCR to be included in the composite digest are selected by the same type of structure used for TPM2_PCR_Read(). The selection structure is first filtered so that unimplemented PCR are not in the selection structure. Then, a composite digest of all of the selected PCR is created. Finally, the filtered selection structure and the composite digest are hashed to create the final digest value. That digest may be compared to a required digest (TPM2_PolicyPCR()) or returned in an attestation (TPM2_Quote()).

17.7 PCR Authorizations

17.7.1 Introduction

TPM2_PCR_Reset(), TPM2_PCR_Extend(), TPM2_PCR_Event(), and TPM2_EventSequenceComplete() require authorization for the PCR being modified. The type of the authorization may differ based on the PCR being modified. A PCR may be defined as having a fixed, EmptyAuth; a variable *authValue;* or a variable *authPolicy*.

The authorization (*authValue* or *authPolicy*) for a PCR may apply to a set of PCR. That is, several PCR may be designated as using the same authorization value so that changing the authorization value (*authValue* or *authPolicy*) of any PCR in the set will change the value for all PCR in the set. A set of PCR that are authorized by an authValue are in an *authorization set*. A set of PCR that are authorized by an *authPolicy* are in a *policy set*.

The type of authorization associated with each PCR is fixed by a platform-specific specification. For each set, the platform-specific specification defines the PCRs that are in the set. A PCR should not be in more than one policy set or one authorization set.

A PCR may be in both a policy set and an authorization set. If it is in both, the only way to use the *authValue* of the authorization set is with a policy that contains TPM2_PolicyAuthValue() or TPM2_PolicyPassword().

An indication of the PCR in an authorization set may be read using TPM2_GetCapability(*capability* == TPM_CAP_PCR_PROPERTIES, *property* == TPM_PT_PCR_AUTH) and the PCR in a policy set may be read using TPM2_GetCapability(*capability* == TPM_CAP_PCR_PROPERTIES, *property* == TPM_PT_PCR_POLICY).

NOTE 1 The reference implementation only provides support for one set of each type. If additional sets are needed, the property types for TPM_CAP_PCR_PROPERTIES could be extended.

NOTE 2 If a PCR is in multiple policy or authorization sets, the TPM will use the policy or authorization of the lowest numbered set. That is, the set with the lowest TPM_PT_PCR_POLICY or TPM_PT_PCR_AUTH property.

To authorize a PCR, the correct authorization type is required, which will depend on the authorization set of a PCR. In all cases, The EmptyAuth value may be provided in either an HMAC session using a zero-length *authValue* in the HMAC calculation or as a zero length password.

17.7.2 PCR Not in a Set

If the PCR is in no set, then the authorization may only be with an EmptyAuth value.

17.7.3 Authorization Set

If the PCR is in an *authorization set,* then the authValue of the PCR is provided either with an HMAC session or in a password. When a PCR has a fixed, EmptyAuth value, an authorization session is still required.

When a PCR has a variable authValue, that *authValue* is reset to an EmptyAuth on each STARTUP(CLEAR). It is preserved across STARTUP(STATE). A variable *authValue* may be changed using TPM2_PCR_SetAuthValue() by an entity with knowledge of the *authValue*.

17.7.4 Policy Set

An *authPolicy* for a policy set has both a hash algorithm and a digest value.

If the hash algorithm for the *authPolicy* is TPM_ALG_NULL, the policy has not been set. This *uninitialized policy set* will use an EmptyAuth.

If the digest algorithm for the policy is not TPM_ALG_NULL, then the policy set is an *initialized policy set*. If the PCR is in an initialized policy set, then the authorization may only be given with a policy session.

The hash algorithm for all policy sets is set to TPM_ALG_NULL by TPM2_ChangePPS(). The algorithm and *authPolicy* associated with a PCR may only be changed using TPM2_SetAuthPolicy() by an entity with knowledge of the Platform Authorization.

If an HMAC session or a password is used for a PCR in an initialized policy set, then the TPM will return an error (TPM_RC_AUTH_TYPE). If a policy session is used for a PCR that is not in an initialized policy set, then the TPM will return an error (TPM_RC_POLICY_FAIL). Neither of these two failures would cause an update of the dictionary attack protection.

17.7.5 Order of Checking

When determining the correct type of authorization for a PCR, the TPM will use the authorization type. If the authorization is a password or HMAC session, The TPM will check to see if the PCR is in an authorization set.

17.8 PCR Allocation

A TPM may support reallocation of the PCR by the platform. To change the allocation of PCR, the platform would use TPM2_PCR_Allocate(). The allocation structure has a PCR selection for each implemented hash algorithm. To allocate a PCR in a bank, the corresponding bit would be SET in the selection for that bank.

The TPM2_PCR_Allocate() changes to PCR allocation take effect upon the next TPM2_Startup(TPM_SU_CLEAR) and persist until the next TPM2_PCR_Allocate().

NOTE 1 Because of RAM limitations, an implementation might not allow arbitrary allocation of PCR within a bank. This does not create a deployment issue as the platform is expected to be able to manage the TPMs that would be attached to that platform.

An allocation may not be made for PCR if the attributes for the PCR are not defined by the platform-specific specification of that TPM.

NOTE 2 The attributes for a PCR include the Startup() initialization value, the locality for reset, and the locality for extend.

There is a requirement that a bank exists for each hash algorithm but there is no requirement that the bank have any PCR (that is, all selection PCR selection bits for the bank may be CLEAR).

It is a valid implementation for the TPM to ship with a specific PCR allocation that is not changeable. If the TPM does not allow the changing of the allocation, it would not implement TPM2_PCR_Allocate().

17.9 PCR Change Tracking

To support the use of PCR in policy the TPM maintains a *pcrUpdateCounter.* In general, this counter is incremented each time a PCR is modified (either extended or reset). This counter is used when a policy requires that PCR have a specific value (see 19.7.6.6).

A platform-specific specification may designate that updates of selected PCR will not cause a change to *pcrUpdateCounter*.

A bitmap of the PCR that can be updated without changing *pcrUpdateCounter* can be read with TPM2_GetCapability(*capability* == TMP_CAP_PCR_PROPERTY, property == TPM_PT_PCR_NO_INCREMENT).

17.10 Other Uses for PCR

The PCR-related commands defined in this library cover common use cases. Platform-specific specifications define PCR attributes that control this behavior and describe how PCR should be used by external software.

EXAMPLE 1 Example use cases supported by PCR-related commands in this library are logging of components during boot or a runtime-switch in the TCB.

However, PCR are designed for more generalized representation of platform state, and platform-specific specifications may define additional PCR behaviors that capture this. Generally, a platform specification may define a PCR to represent any value that is authoritatively known by the TPM or has been securely communicated to the TPM. ISO/IEC 11889 demands no particular behavior or value-semantics for such PCR.

EXAMPLE 2 A TPM for a “trusted lock” might define a PCR that has value of zero to indicate that a door is closed, and one to indicate that a door is open.

EXAMPLE 3 A virtual-TPM specification might define a PCR that has a value that represents some characteristic of the virtual machine that is issuing the TPM command..

NOTE A PCR can "represent" a value either by having the PCR set to that value or by having the PCR extended with the value. In the case of the "trusted lock," it is more likely that the PCR would contain either a zero or one to represent the state of the lock than that each change to the lock be extended to a PCR.

This does not mean that the platform-specific working groups are allowed to define new commands to operate on PCR.

18 TPM Command/Response Structure

18.1 Introduction

A command is a TPM Protected Capability that indicates an operation to be performed by the TPM. It contains from one to five components, in the following order:

1) a command header that indicates the overall size of the command, the command code, and a tag indicating whether the Authorization Area is present;
2) a command-dependent number (zero to three) of handles identifying the Shielded Locations with/on which the command (Protected Capability) operates;
3) a 32-bit value indicating the size of the Authorization Area;
4) an Authorization Area containing one to three session structures; and

NOTE 1 Components 3 and 4 always occur together. The authorization size parameter is not present if there are no sessions in the Authorization Area.

5) a command-dependent parameter area containing qualifying information for the command.

A response contains

1) a response header that indicates the overall size of the response, the response code, and a tag indicating whether the Authorization Area is present;
2) a command-dependent number (zero or one) of handles identifying the Shielded Locations with/on which the command (Protected Capability) operates;
3) a 32-bit value indicating the size of the parameter area;
4) a command-dependent parameter area containing the values produced by the TPM; and
5) an Authorization Area containing one to three session structures.

NOTE 2 Components 3 and 5 always occur together. That is, if the Authorization Area is empty, the 32-bit value for the parameter size will not be present.

As with the command, the formats for the remaining areas of the response are dependent on the value of the associated command code. The session and parameter area order are reversed in a response.

The ordering of authorization structures and command-dependent parameters is intended to minimize TPM complexity. In a command, the authorization structures are first in order that the TPM can generate its authorization digests from the command-dependent parameters as they arrive. In a response, command-dependent parameters are first in order that the TPM can use the output buffer to assemble the command-dependent parameters prior to generating its authorization digests.

NOTE 3 In traditional implementations, all of the octets of a command are available at the same time so skipping around in the data structure was not an issue. In some anticipated implementations, this will not be the case and the processing of a command or response will need to be more linear.

For tables in ISO/IEC 11889, the separators indicating the demarcations between the header, handle, authorization, and parameter components are shown in Table 5.

Table 5 — Separators

Separator	Meaning
	This type of separator is followed by one or more handles.
	In a command, this type of separator is followed by a 32-bit value indicating the number of octets in the *Authorization Area*. In a response, it is followed by a 32-bit value indicating the number of *parameter* octets (present only if tag for command/response is TPM_ST_SESSIONS).
	This type of separator is followed by one or more session structures (present only if tag for command/response is TPM_ST_SESSIONS).
	This type of separator is followed by one or more parameters

.

Figure 10 and Figure 11 show the basic layout of a TPM command and response (see 18.9 for a detailed example command and 18.10 for a detailed example response).

NOTE Not all sessions in the Authorization Area are required to be used for authorization. Sessions may also be used for audit or parameter encryption.

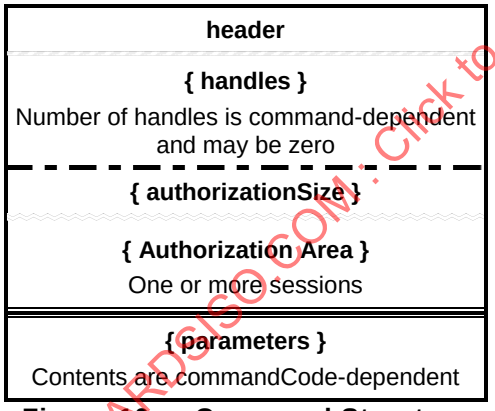


Figure 10 — Command Structure

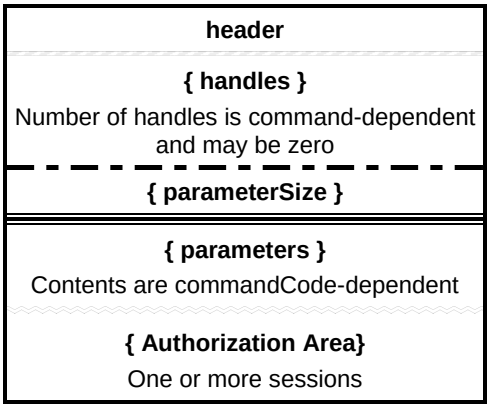


Figure 11 — Response Structure

18.2 Command/Response Header Fields

18.2.1 Introduction

A command or response header always contains three values, displayed in Figure 12.

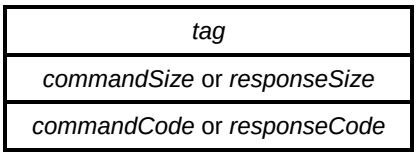


Figure 12 — Command/Response Header Structure

18.2.2 *tag*

A *tag* is present in all commands sent to the TPM and in responses received from the TPM. The *tag* indicates whether a command is formatted according to ISO/IEC 11889 (first edition) or ISO/IEC 11889. If the latter, the *tag* indicates if any session data is present.

Table 6 lists the *tag* values used for commands and response defined in ISO/IEC 11889.

NOTE The tags for commands defined in ISO/IEC 11889 indicate only whether the command uses one or more sessions, and do not indicate the number of sessions present in the Authorization Area. Each session structure that uses a variable session handle follows the same format, which could be parsed to find the start of the next session.

Table 6 — *Tag* Values

Value	Description
TPM_ST_NO_SESSIONS	This value indicates that the command or response is formatted according to ISO/IEC 11889 and that the Authorization Area is empty. It is used in a response if the command used this tag or if the command did not complete successfully.
TPM_ST_SESSIONS	This value indicates that the command or response is formatted according to ISO/IEC 11889 and that the Authorization Area contains one or more authorizations. It indicates that the *authorizationSize* value is present; in a response, it indicates that the *parameterSize* value is present.

18.2.3 commandSize/responseSize

The *commandSize*/*responseSize* value indicates the total number of octets of this command/response, starting with the first octet of *tag.*

18.2.4 commandCode

The *commandCode* appears only in the command to the TPM. It indicates the operation that the TPM should perform and the formats of the handle and parameter areas for the command and response. The *commandCode* parameter is included in the command parameter hash (*cpHash*) and the response parameter hash (*rpHash*).

18.2.5 responseCode

The *responseCode* appears only in the response from the TPM. A *responseCode* of TPM_RC_SUCCESS (zero) indicates that the TPM has successfully completed the command and,

depending on the command format, that the handle, parameter, and authorization components are present.

A non-zero *responseCode* indicates an error or fault. In this case, *tag* will be TPM_ST_NO_SESSIONS, and *responseSize* is 10, indicating that no octets follow the *responseCode*. No handle, parameter, or session response components are present.

18.3 Handles

Handles are TPM-assigned values that let the caller indicate the TPM-resident structure that a command is to manipulate. That is, the handle identifies the Shielded Location with/on which a Protected Capability is to operate. Some TPM commands require no handles.

EXAMPLE 1 TPM2_Startup() is a command that requires no handles.

The number of handles in the command and in the response is implied by the *commandCode*. It also indicates the command handles that have an associated authorization session. Handles that require authorization in an associated authorization session are listed ahead of handles that do not have an associated authorization session.

EXAMPLE 2 TPM2_ObjectChangeAuth() has two handles, one (*objectHandle*) that uses an authorization session, and one (*parentHandle*) that does not. The standard command syntax dictates that *objectHandle* occur first.

A response may have handles only if the *responseCode* is TPM_RC_SUCCESS.

The architectural limit for the number of handles in the handle area is seven. This limit is determined by the error-reporting scheme.

NOTE No currently defined command uses more than three handles.

18.4 Parameters

The *commandCode* indicates the structure of the optional handle and parameter areas. The contents of these parameter areas differ for commands and responses. Some TPM commands require no parameters.

EXAMPLE 1 TPM2_Clear() is an example of a TPM command that requires no parameters.

All parameter values and the *commandCode* are included in the *cpHash* or *rpHash*. *authorizationSize* is not included in the *cpHash*, and *parameterSize* is not included in the *rpHash*.

NOTE 1 If a parameter is encrypted, it is included in the *cpHash/rpHash* after encryption.

A response may have parameters only if the *responseCode* is TPM_RC_SUCCESS.

The architectural limit for the number of parameters in the handle area is 15. This limit is determined by the error-reporting scheme.

NOTE 2 This is the limit of parameters in the parameter list, not the number of values that may be in the parameter area. If a command needs more than 15 parameters, a new structure could be defined that encapsulates two or more of those parameters into a single structure, which may then be unmarshaled as a unit. The only loss is that error reporting might not provide as much detail when a compound parameter has an error.

As specified in clause 21, for a command or response parameter to be encrypted, it must be the first parameter and it must be a TPM2B type.

NOTE 3 In order to encrypt more than one parameter, they are encapsulated in a TPM2B making them a single parameter.

EXAMPLE 2 The TPM2B_SENSITIVE_CREATE is the first parameter to TPM2_CreatePrimary(). The data member, TPMS_SENSITIVE_CREATE, has two members, a TPM2B_AUTH and a TPM2B_SENSITIVE_DATA. The encapsulation of them in the TPM2B_SENSITIVE_CREATE permits both to be encrypted.

18.5 *authorizationSize*/*parameterSize*

These values are only present if the tag of the command/response is TPM_ST_SESSIONS.

In a command, the *authorizationSize* indicates the number of octets in all of the authorization structures in the Authorization Area of the command. *authorizationSize* does not include the four octets of the *authorizationSize* value. The minimum value for *authorizationSize* is 9.

NOTE 1 The maximum value depends on the size of the largest digest produced by any hash implemented on the TPM.

NOTE 2 The driver and the TPM use the *authorizationSize* field to determine the number of authorizations After *authorizationSize* bytes have been processed, there are no more authorizations.

In a response, *parameterSize* indicates the number of octets in the parameter area of the response and does not include the four octets of the *parameterSize* value. *parameterSize* may have a value of zero.

authorizationSize is not included in *cpHash*, and *parameterSize* is not included in the *rpHash*.

18.6 Authorization Area

18.6.1 Introduction

The Authorization Area is present in a command only if *tag* for the command is TPM_ST_SESSIONS. If present, the Authorization Area will contain:

- zero, one, or two authorizations (session or password);
- an optional session used for decrypting data sent to the TPM;
- an optional session used for encrypting data sent by the TPM; or
- an optional session used for auditing.

If *tag* is TPM_ST_SESSIONS, then the Authorization Area will have at least one but no more than three authorization/session blocks. If tag is TPM_ST_NO_SESSIONS, then there is no Authorization Area.

The number of authorization sessions that a command will have is indicated in the command schematic in ISO/IEC 11889-3. If a handle in the handle area has the "@" decoration, then an authorization session is required be present (an authorization session being either a password, a policy session, or an HMAC session).

The authorization sessions occur in the order of the associated entity handles. That is, the first handle with an "@" decoration will be associated with the first session in the Authorization Area.

Other sessions may be added to the Authorization Area. Those sessions may be designated as being for encryption, decryption, or audit; in any combination, in any order. However, in a single command, only one session is allow to have the *encrypt* attribute, one session is allowed to have the *decrypt* attribute and one session is allowed to have the *audit* attribute.

A single session may be used for authorization, encryption, decryption, and audit at the same time. That is, if a session has one handle with the "@" decoration, the associated authorization session may have the *encrypt*, *decrypt*, and *audit* attributes all set. A password authorization may not be used for anything but authorization and the TPM will return an error (TPM_RC_ATTRIBUTES) if *encrypt*, *decrypt*, or *audit* is SET in a password authorization.

NOTE 1 If an authorization session has *encrypt*, *decrypt*, and *audit* all SET, then the command can only have one authorization session.

The combinations of attributes allowed for each session are summarized in Table 7

Table 7 — Use of Authorization/Session Blocks

Position	password authorization(1)(6)	authorization session(2)(6)	encryption session(3)	decryption session(4)	audit session(5)
1	✓	✓	✓	✓	✓
2	✓	✓	✓	✓	✓
3			✓	✓	✓

NOTE 1 A password authorization can not be used for encryption, decryption, or audit.

NOTE 2 An HMAC authorization session can be also be used for encryption, decryption, and audit and a policy authorization session can also be used for encryption and decryption.

NOTE 3 Only one session can be designated as being used for encryption.

NOTE 4 Only one session can be designated as being used for decryption.

NOTE 5 Password authorization sessions and policy sessions cannot be used for audit.

NOTE 6 Authorization sessions come before sessions used only for encryption, decryption, or audit

In ISO/IEC 11889-3, the schematic for each command will indicate if it has handles and if use of those handles requires authorizations. If there is an *at* symbol ("@") character in front of the handle name, then use of the TPM resource associated with the handle requires authorization and an authorization (session or password) will be present. An authorization will be present for each TPM resource that requires authorization (each handle with an "@"). An additional indication that a handle requires authorization is that, in the "Description" column of the command schematic, each handle has an "Auth Index:" entry. If that entry says "None", then no authorization is required. If that entry is followed by a number, then the number indicates the order of the associated authorization in the list of authorizations.

NOTE 2 Currently, no command requires more than two authorizations.

If a command requires authorizations, then those authorizations will be first in the list of authorizations/sessions. They may then be followed by other sessions used for encryption, decryption, or audit.

If the *responseCode* is TPM_RC_SUCCESS, the response has the same number of sessions in the same order as the request. Otherwise, no authorization or audit sessions are present.

18.6.2 Authorization Structure

18.6.2.1 Command

In a command, each authorization structure has the format shown in Figure 13.

session handle	A four-octet value indicating the session number associated with this data block (will be TPM_RS_PW for a password authorization)
size field	A two-octet value indicating the number of octets in *nonce*
nonce	If present, an octet array that contains a number chosen by the caller
session attributes	A single octet with bit fields that indicate session usage
size field	A two-octet value indicating the number of octets in *authorization*
authorization	If present, an octet array that contains either an HMAC or a password, depending on the session type

Figure 13 — Authorization Layout for Command

18.6.2.2 Response

In a response, each session structure has the format shown in Figure 14.

size field	A two-octet value indicating the number of octets in *nonce* (will be zero for a password authorization)
nonce	If present, an octet array that contains a number chosen by the TPM
session attributes	A single octet with bit fields that indicate session usage
size field	A two-octet value indicating the number of octets in *acknowledgment*
acknowledgment	If present, an octet array that contains an HMAC

Figure 14 — Authorization Layout for Response

Clause 19.6.7 describes the methods for creating an authorization session.

18.6.3 Session Handles

Session handles are specified in 15.4. They identify the session being referenced by a specific session structure.

For a given command, the handle associated with a specific HMAC or policy session may occur only once in the Authorization Area. The handle representing a password authorization (TPM_RS_PW) can occur multiple times.

18.6.4 Session Attributes (*sessionAttributes*)

Each session has a *sessionAttributes* octet to indicate how the session is to be applied. Table 8 explains the meaning of the fields in this octet.

Table 8 — Description of *sessionAttributes*

Attribute	Meaning
continueSession	This attribute is used to indicate to the TPM if the session is to remain 'active' when the command completes. If this attribute is CLEAR in the command and the command completes successfully (TPM_RC_SUCCESS), then the session will be flushed from TPM memory and the associated session handle will be available to be assigned to new sessions. When the TPM responds, it will echo this attribute to indicate that the session remains open. The primary purpose of this attribute is to eliminate having to do explicit flushes (TPM2_FlushContext()) of a session when it is no longer used. Having this bit CLEAR on the last use of the session will end it and reclaim the TPM resources assigned to this session. For a password authorization, this attribute has no effect, as there are no TPM resources associated with a password authorization. This attribute will always be SET in a response associated with a password authorization. If the audit attribute is SET, then this attribute should also be SET since the audit data will be lost if the session is flushed.
decrypt	This attribute is used to indicate to the TPM that the secrets associated with the session are to be used to decrypt the first parameter of the command (the session-based encryption scheme is defined in clause 21). The parameter will be decrypted after the HMAC computations are successfully completed. This attribute may only be SET in a command that has a sized buffer as its first parameter. This attribute is required to be CLEAR in a password session. If SET in a password session, then the TPM will return an error because there is no session key for the decrypt operation This attribute is echoed by the TPM in the corresponding session in the response This attribute may only be SET in one session per command. A session with this attribute does not need to be associated with an entity identified in the handle area. That is, the session may be added just for using the session's secret for parameter decryption. This attribute can be SET in combination with any other session attribute and any session type, including TPM_SE_TRIAL.

Attribute	Meaning
encrypt	This attribute is used to indicate to the TPM that the secrets associated with the session are to be used to encrypt the first parameter of the response (the session-based encryption scheme is defined in clause 21). The parameter will be encrypted before the TPM performs the HMAC computations for any of the sessions. This attribute may only be SET in a response that has a sized buffer as its first parameter. This attribute is required to be CLEAR in a password session. If SET in a password session, then the TPM will return an error because there is no session key for the encrypt operation. This attribute is echoed by the TPM in the corresponding session in the response. This attribute may only be SET in one session per command. A session with this attribute does not need to be associated with an entity identified in the handle area. That is, the session may be added just for using the session's secret for parameter decryption. This attribute can be SET in combination with any other session attribute and any session type, including TPM_SE_TRIAL.
audit	This attribute indicates that the session is being used for audit. A digest is maintained in the session context and is updated each time the session is used with a command and *audit* is SET. This attribute does not need to be SET in every use of the session but the TPM will only update the audit data when the session is used with this attribute SET. This attribute has no meaning for a password authorization and is required to be CLEAR. This attribute is not allowed to be SET in a policy or trial policy session. This is because the context of the policy session would have to increase in order to hold the additional audit digest. This is significant overhead and, rather than require the additional memory in policy sessions, use of audit is restricted to HMAC sessions. After an HMAC session is started (TPM2_StartAuthSession(*sessionType* = TPM_SE_HMAC), this attribute may be set in any subsequent use of the session. On the first use of the session with this attribute set, the TPM will initialize the audit digest to 0…0 and then extend the cpHash for the command and then extend the rpHash for the command. This attribute will be echoed by the TPM in the response. This attribute may be used in combination with any other session attributes but only one session in each command may have this attribute SET.
auditExclusive	This attribute is use to restrict use of an audit session. When this attribute is SET, the TPM will validate that the session has been used for all auditable commands since the audit sequence was started. If the session was used for all auditable commands, then it is said to be "exclusive"(see 20.2 for explanation of exclusive audit sessions). If this attribute is SET and the session is exclusive, then the command will execute. Otherwise, the TPM will fail this command to indicate to the caller that some TPM actions were not included in the audit sequence. Evaluation of the exclusive status is done at the start of the command. A session does not obtain the exclusive status until the end of the command (this prevents a session from becoming exclusive if the command fails). The implication of this processing is that, if this attribute is SET in the command that starts the audit sequence, the command will fail because the session has not yet become exclusive. In a response, this attribute will be SET if the session has exclusive status. When a session is first used as an audit session this attribute will be SET in the response as no command has executed without this session since the start of the sequence. This attribute may only be SET when the *audit* attribute is SET which excludes this attribute from being SET on a password authorization or a policy session.

Attribute	Meaning
auditReset	This attribute allows the caller to restart an audit sequence with a session that has previously been used for audit. If the associated command completes successfully, the TPM will initialize the session audit hash with 0…0 before Extending the cpHash and the rpHash. The response will have the exclusive attribute SET. This attribute may only be SET if audit is SET. The TPM will echo this attribute in the response.

NOTE 1 For the continueSession attribute, "echo" means that the value of a session attribute will be the same in the response as it was in the command.

NOTE 2 For the auditExclusive attribute, an audit sequence is started when the audit digest is reset to 0…0. The audit digest is set to 0…0 when the session is first used as an audit session and when the audit digest is reset (see the description of the *auditReset* attribute above).

18.7 Command Parameter Hash (*cpHash*)

The command parameter hash (*cpHash*) is used in the computation of a command authorization HMAC and is included in the digests of session and command audits (depending on the policy, the *cpHash* may also be used in the authorization). The *cpHash* is computed from the parameters of the command as follows:

$$cpHash := \mathbf{H}_{sessionAlg}\,(commandCode\ \{||\ Name1\ \{||\ Name2\ \{||\ Name3\ \}\}\}\ \{||\ parameters\ \}) \quad (16)$$

where

$\mathbf{H}_{sessionAlg}$	hash function using the algorithm selected for the session when it was initialized
$commandCode$	command code for the command
$Name1$	unique identity of the entity associated with the first handle
$Name2$	unique identity of the entity associated with the second handle
$Name3$	unique identity of the entity associated with the third handle
$parameters$	remaining command parameters

18.8 Response Parameter Hash (*rpHash*)

The response parameter hash is used in the computation of a response acknowledgment HMAC and is included in the digest of session and command audits. The *rpHash* is computed from the parameters of the response as follows:

$$rpHash := \mathbf{H}_{sessionAlg}\,(responseCode\ ||\ commandCode\ \{||\ parameters\ \}) \quad (17)$$

where

$\mathbf{H}_{sessionAlg}$	hash function using the algorithm selected for the session when it was initialized
$responseCode$	command result code
$commandCode$	the commandCode from the command
$parameters$	response parameters

The contents of the *handles* area of the response are not included in the *rpHash*.

NOTE An *rpHash* needs to be computed only when the *responseCode* is TPM_SUCCESS, which means that it is redundant to include the response code. It is retained for legacy reasons.

18.9 Command Example

Table 9 shows an example of a command schematic used in ISO/IEC 11889. The command has two object handles (*handleA* and *handleB*). The "@" on the handleA name indicates that use of the entity associated with the handle requires authorization. The command has at least one session to authorize use of *handleA*. It will not have a session for use of *handleB*. The Authorization Area may have an additional audit session and a session used only for parameter encryption. Since one session is required, *tag* is TPM_ST_SESSIONS, and the *authorizationSize* field is present.

Although they are not shown in the command schematic, the *authorizationSize* value and the Authorization Area would be present in the command buffer, and be located between *handleB* and *dataSize*.

NOTE: The Authorization Area is not shown with the command schematic because no single representation is possible.

The command and response tables have three columns.

1) **Type** – This column indicates the data type of the parameter passed to the TPM in a command or received from the TPM in a response.
2) **Name** – This column indicates the name of the parameter. This name is referenced in the description of the command that precedes the command table and in the detailed actions of the command that follows the response table.
3) **Description** – This column provides a limited description of the parameter and indicates the possible options for the command.

EXAMPLE 1

Table 9 — Command Layout for Example Command

Type	Name	Description
TPM_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Example
TPM_HANDLE	@handleA	handle to use for one object of the command Auth Index: 1 Auth Role: USER
TPM_HANDLE	handleB	handle to use for the second object Auth Index: None
UINT32	dataSize	example data size
OCTET	data[dataSize]	example data

Table 10 illustrates all command octets for the command in Table 9. In this example, the nonce size is 20 octets and the authorization HMAC is computed using SHA256. The values in shaded cells are not shown in the ISO/IEC 11889-3 schematic of the command but are included in the command data sent to the TPM.

EXAMPLE 2

Table 10 — Example Command Showing *authorizationSize*

Offset	Size	Parameter	Value
0	2	tag	TPM_ST_SESSIONS
2	4	commandSize	209 < size in octets of the command >
6	4	commandCode	TPM_CC_Example
10	4	handleA	< a valid TPM resource handle>
14	4	handleB	< a valid TPM resource handle>
18	4	authorizationSize	61 < size of the authorization session >
22	4	authHandle	< a valid TPMI_SH_AUTH_SESSION >
26	2	nonceCallerSize	20 < size of *nonce* >
28	20	nonceCaller	< a 20-octet random value>
48	1	sessionAttributes	(continueSession=1)
49	2	hmacSize	32 <size of *HMAC*>
51	32	HMAC	< a 32-octet HMAC value based on SHA256>
83	2	dataSize	32 < size of the buffer >
85	124	data[dataSize]	< 124 octet buffer >
209			

18.10 Response Example

Table 11 shows an example schematic as it would appear in ISO/IEC 11889-3. The example is for a response sent from the TPM after successful completion of the example command in Table 9. The response has the same number of sessions in the same order as did the command.

EXAMPLE

Table 11 — Response Layout for Example Command

Type	Name	Description
TPM_TAG	tag	TPM_ST_SESSIONS
UINT32	responseSize	
TPM_RC	responseCode	response code of the operation
TPM_HANDLE	handle	not included in the rpHash
UINT32	dataSize	size in octets of the following data
OCTET	data[dataSize]	a returned block of information

Table 12 illustrates the full response for the command in Table 9. As in the command, the nonce size is 20 octets and the acknowledgment HMAC is computed using SHA256. The values in shaded cells are not shown in the ISO/IEC 11889-3 schematic of the response but are present in the response data from the TPM.

Table 12 — Example Response Showing *parameterSize*

Offset	Size	Parameter	Value
0	2	tag	TPM_ST_SESSIONS
2	4	responseSize	203 < size in octets of the response >
6	4	responseCode	0 < success >
10	4	handle	< a valid TPM_HANDLE >
14	4	parameterSize	128
18	4	dataSize	124
22	124	data[dataSize]	< 124 octet buffer >
146	2	nonceTpmSize	20
148	20	nonceTPM	< a 20-octet random value >
168	1	sessionAttributes	(continueSession=1)
169	2	hmacSize	32
171	32	HMAC	< a 32-octet HMAC value based on SHA256 >
203			

19 Authorizations and Acknowledgments

19.1 Introduction

Many commands to the TPM reference TPM-resident structures, and use of these structures may require authorization. This authorization is provided in structured data that follows the command data. When an authorization is provided to a TPM, the TPM will provide an acknowledgment.

To provide flexibility in how the authorizations are given to the TPM, ISO/IEC 11889 defines three authorization types:

1) password;
2) HMAC; and
3) policy.

Depending on the command, zero, one, or two authorizations may be required. In a command, the authorizations follow the handles and in a response, the authorization replies follow the response parameters. The command definition indicates how many authorizations are required.

19.2 Authorization Roles

For each object and NV Index, there is a set of operations that can be performed on or with that object or NV Index. The operations are divided into groups, based on the impact of the operation on the object. To perform an operation with or on an object in a group, the authorization specific to that group must be provided. When performing an operation in one of the groups, the caller is acting in a specific role with respect to that object.

The TPM supports three different authorization roles:

1) **USER** – this authorization role is used for the normal uses of a key. Methods are defined to allow USER role authorization to be provided either with an authorization value (*authValue*) or a policy. If *userWithAuth* is SET then USER role authorization may be provided with a password authorization or an HMAC session. If *userWithAuth* is CLEAR, then a password and HMAC authorizations may not be used to provide USER role authorizations. A policy session that satisfies the *authPolicy* of the entity may be used regardless of the setting of *userWithAuth*.

   EXAMPLE 1 Examples of normal use of a key are signing with a signing key or loading the child of a Storage Key.

   NOTE 1 For USER role, an *authPolicy* is satisfied when the *policyDigest* of a policy session matches the value of the *authPolicy* value of the object.

   NOTE 2 If use of an object is to be gated based on PCR values, a policy session is used (see 19.7). If the intent is that different Users have access to the object but only if the PCR are correct, then it is likely that authorization with the *authValue* will be disabled; otherwise, the caller could circumvent PCR protections simply by providing the *authValue*.

2) **ADMIN** – the object Administrator controls the certification of an object (TPM2_Certify() and TPM2_ActivateCredential()) and controls changing of the *authValue* of an object (TPM2_ObjectChangeAuth()). When an action requires ADMIN role authorization, that authorization may be provided using the authValue of the object if the *adminWithPolicy* attribute of the object is CLEAR. As with USER role authorization, ADMIN role may always be provided with a policy session as long as the policy session satisfies the *authPolicy* of the object.

NOTE 3 For ADMIN role, an *authPolicy* is satisfied when *policySession→policyDigest* matches the value of the *authPolicy* value of the object and *policySession→commandCode* matches *commandCode* for the authorized command.

EXAMPLE 2 If the *adminWithPolicy* attribute of an object is SET, and if no branch in the object's policy equation contains TPM2_PolicyCommandCode(TPM_CC_Certify), then certification of that key cannot occur.

3) **DUP** – this authorization role is only used for TPM2_Duplicate(). If duplication is allowed, authorization must always be provided by a policy session and the authPolicy equation of the object must contain a command that sets the policy command code to TPM_CC_Duplicate.

19.3 Physical Presence Authorization

Authorization for some commands requires that it be provided with Platform Authorization. Authorization for some other commands allows use of either Platform Authorization or Owner Authorization (Most of these commands cause persistent state change of the TPM). For these commands, it is possible to require that authorization be augmented with an out-of-band method.

For commands that require Platform Authorization and commands that require a hierarchy authorization, it is possible to require an out-of-band authorization. This may take any number of forms. Whatever the form, the out-of-band authorization is referred to in ISO/IEC 11889 as Physical Presence (PP). This does not mean that the signaling requires a human to be physically present in order for the indication to be provided. The term is used in ISO/IEC 11889 because it was used in ISO/IEC 11889 (first edition) to refer to a similar concept.

EXAMPLE Examples of forms of out of band authorization are a dedicated pin in the TPM, a special signaling method through the TPM interface, or any desired alternative.

The TPM maintains a table of the commands that require that PP be asserted to authorize command execution. Only certain commands may be included in this table. If, in ISO/IEC 11889-3, the schematic for a command has TPM_RH_PLATFORM in the "Description" column for one of the handles, then that command can be added to the list of commands that require PP. Otherwise, it may not.

NOTE 1 In the "Description" column, TPM_RH_PLATFORM will be followed by +PP if assertion of Physical Presence is required or "+{PP}" to indicate that assertion of Physical Presence could be required if indicated by the table.

NOTE 2 A platform-specific specification could require that the table be initialized in a specific way. It could even require that the table have certain commands defined to require PP confirmation even though a PP interface is not provided on the TPM. This would serve to disable the use of that command by the platform.

When the authorization handle is TPM_RH_PLATFORM, the TPM checks the table to see if the command requires confirmation with PP. If so, PP is checked before the TPM performs any other authorization checks.

TPM2_PP_Commands() is used to change the contents of the table of commands that require confirmation with PP authorization. Authorization of the command TPM2_PP_Commands() requires that PP be asserted and TPM2_PP_Commands() may not be removed from the list of commands that require PP.

NOTE 3 This constraint on TPM2_PP_Commands() prevents setting or modification of the table if no PP interface exists on the TPM.

The contents of the table may be read using TPM2_GetCapability(*capability* == TPM_CAP_PP_COMMANDS).

19.4 Password Authorizations

A plaintext password value may be used to authorize an action when use of an *authValue* is allowed. A plaintext password may be appropriate for cases in which the path between the caller and the TPM is trusted or when the authorization value is well known. For these instances, encryption of parameters or the hiding of authorization values in an HMAC is not required.

NOTE 1 While it may seem relatively easy for a caller to perform an HMAC, there are situations where the caller is resource-constrained and unable to do so. This is especially true when the calling software does not support the hash algorithms implemented in the TPM. Additionally, authentication using a cryptographic protocol makes it difficult to provide operating system abstractions.

A reserved authorization handle (TPM_RS_PW) indicates that the authorization is a password.

TPM_RS_PW is always available, and a separate action to create an authorization session is not required. A password authorization does not use nonces.

A password authorization lets the caller send more or fewer octets than are present in the object's authorization field. The TPM truncates any octets of zero on either of the two values before they are compared.

If present, a password authorization is always associated with a command handle that requires authorization as there is no session context associated with a password that would allow it to be used for encryption or command audit.

Unlike other handles for other session types, the TPM_RS_PW session handle may be used for more than one authorization.

Password authorization data sent to the TPM has the format shown in Table 13.

Table 13 — Password Authorization of Command

Type	Name	Description
TPMI_SH_AUTH_SESSION	authHandle	required to be the reserved authorization session handle TPM_RS_PW
TPM2B_NONCE	nonce	required to be an Empty Buffer
TPMA_SESSION	sessionAttributes	only *continueSession* may be SET
TPM2B_AUTH	password	authorization compared to the *authValue* of the TPM entity

Table 14 illustrates the format of a password authorization in a response. This structure is provided to ensure a one-to-one correspondence between the sessions in the command and in the response.

NOTE 2 This structure is used to provide symmetry between password and other response sessions.

Table 14 — Password Acknowledgment in Response

Type	Name	Description
TPM2B_NONCE	nonceTPM	zero-length for a password authorization
TPMA_SESSION	sessionAttributes	copy of the flags from the password authorization in the command
TPM2B_AUTH	hmac	zero-length buffer for a password authorization

19.5 Sessions

A session is a collection of TPM state that changes after each use of that session. When an object context is loaded into the TPM, multiple copies of the object context may exist both on the TPM and in saved contexts (see clause 30). When a session context is created, only one copy of that context may exist either on the TPM or as a saved context. The context of a session changes on each use.

A session has a handle that is assigned by the TPM when the session is created. That handle will always refer to the same session until the session is closed. If a handle is re-assigned to a subsequently created session, the session context data will contain a TPM-generated nonce that makes the new instance of the session unique, even though the handle may have been used previously. This nonce will change each time the session is used so that previous instances of the same session can be distinguished from each other (i.e., the nonce prevents reuse of stale session contexts).

There are three uses of a session:

1) **authorization** – A session associated with a handle is used to authorize use of an object associated with a handle. If it is not a password authorization, it may also be used to provide keys for encryption of command or response parameters. A policy session used to authorize may not also be used as an audit session. An HMAC session used to authorize may be used as an audit session.
2) **audit** – An audit session collects a digest of command/response parameters to provide proof that a certain sequence of events occurred. An audit session may also be used to provide secrets for encryption of command or response parameters and may be used for authorization of an HMAC session.
3) **encryption** – A session that is not used for authorization or audit may be present for the purpose of encrypting command or response parameters. If an encryption-only session exists, it will follow the authorization sessions and may come before or after a session used only for audit.

A command may have as many as three authorization blocks. Password blocks may only be used for authorization so the maximum number of password blocks is equal to the number of authorizations required by the command.

19.6 Session-Based Authorizations

19.6.1 Introduction

Session-based authorizations are used both for protocols that require confidentiality for the authorization value and for audit sessions that require tracking of a sequence of commands sent to the TPM. An authorization session also provides a means of linking the uses of the session.

There are two types of session-based authorization: HMAC and policy. Both types of session are initiated using TPM2_StartAuthSession(). That command establishes the parameters that will be used for the authorizations. The *sessionType* parameter determines if the session will be an HMAC or policy session. When the session is started, the hash algorithm and TPM nonce size used in the session are specified by the caller. The command may include an initial caller nonce and a *salt* value to generate the session key. The parameters of each session are independent from the parameters of any other session and are limited only by the capabilities of the TPM. When TPM2_StartAuthSession() completes successfully, the TPM returns a handle for the session as well as the initial *nonceTPM* value.

Once an authorization session is established, it may be used to authorize actions in multiple commands. The session is not ended until explicitly closed or flushed.

The secret values of a session are determined by the handles used when the session is started. The command for starting a session allows selection of up to two object handles. One handle indicates a TPM

object that is used to encrypt a salt value that is sent when the session is started. A second handle indicates an object containing a shared secret. The salt value and the shared secret are combined with a nonce provided by the caller to create the session secrets.

19.6.2 Authorization Session Formats

For a session-based authorization session, the authorization structure for a command is as shown in Table 15.

Table 15 — Session-Based Authorization of Command

Type	Name	Description
TPMI_SH_AUTH_SESSION	authHandle	the handle for the authorization session
TPM2B_NONCE	nonceCaller	the caller-provided session nonce; size may be zero
TPMA_SESSION	sessionAttributes	the flags associated with the session
TPM2B_AUTH	hmac	the session HMAC digest value

In a response, the format for the acknowledgement is as shown in Table 16.

Table 16 — Session-Based Acknowledgment in Response

Type	Name	Description
TPM2B_NONCE	nonceTPM	the TPM-provided session nonce. Size is as specified when the session was started.
TPMA_SESSION	sessionAttributes	the flags associated with the session. This attribute should be the same as the values in the command except *continueSession* may be CLEAR.
TPM2B_AUTH	hmac	the session HMAC digest value

19.6.3 Session Nonces

19.6.3.1 Overview

The primary use of a nonce in a session is to prevent an authorization from being reused. When the session is started by TPM2_StartAuthSession(), the caller indicates, among other things, the size of the nonces to be used in the authorization HMAC and an initial nonce value (*nonceCaller*). After establishing the session, the TPM returns a handle to identify the session and a TPM-generated random nonce (*nonceTPM*). The TPM stores this *nonceTPM* in the context of the session.

Each time the session is used for authorization, the caller performs an HMAC using, along with other parameters, the last *nonceTPM* for the session and a new *nonceCaller* for the session. The TPM then uses the received *nonceCaller* and the saved *nonceTPM* to validate the HMAC. For a response, the TPM uses the last *nonceCaller* and a newly generated *nonceTPM* in the HMAC. The caller then uses the received *nonceTPM* and the saved *nonceCaller* to validate the HMAC in the response.

A nonce has a size field indicating the number of octets in the nonce followed by the nonce data. The nonce size is not included in the HMAC computation.

19.6.3.2 Session Nonce Size

When an authorization session is created, the caller provides an initial nonce (*nonceCaller*). The size field of *nonceCaller* is retained by the TPM and used to determine the size of all nonces generated by the TPM (*nonceTPM*) in the subsequent uses of the session. The minimum size for *nonceCaller* in TPM2_StartAuthSession() is 16 octets.

After the initial session setup, the caller may use any size for a *nonceCaller* in each use of the session. The *nonceCaller* size may vary from zero (0) up to the size of *nonceTPM* (the initial *nonceCaller* size).

NOTE A TPM implementation could allow larger nonce sizes but the caller should not expect a TPM to accept a nonce size larger than the initial *nonceCaller* size.

The maximum size that may be requested for *nonceTPM* is the size of the digest produced by the authorization session hash.

EXAMPLE For SHA-1 the maximum size for *nonceTPM* is 20 octets and for SHA256 it is 32 octets.

When a session nonce is used in the authorization session HMAC, the *size* field of the nonce is not included in the authorization computation. If the nonce *size* field is zero (0), then the nonce does not affect the authorization HMAC value.

19.6.3.3 Guidance on Nonce Size Selection

The size of the nonce should be chosen to provide a reasonable guarantee that a TPM-generated nonce value will not be used twice with the same *sessionKey*. The choice of nonce size is not related to the number of uses of a specific authorization session but is related to the number of uses of the *sessionKey*.

An HMAC *sessionKey* is derived from the *authValue* kept in an object and that *authValue* may have a long lifetime. To prevent replay attacks on a long-lived *authValue,* use of large nonces is recommended.

NOTE 1 The combined *nonceCaller* plus *nonceTPM* are what determine the anti-replay protection provided by the nonces. Making the combined size larger than the block size of the session hash is not particularly useful. If the caller does not have a good source of entropy for an RNG, then making the *nonceTPM* the size of the digest of the session hash is recommended, so that a *nonceCaller* size of zero would be satisfactory.

NOTE 2 When using a session for encryption, if a parameter is encrypted in a response to one command and a parameter is encrypted in the request of the next command, and they both use the same session for encryption, then the caller would provide a *nonceCaller* in order to prevent the use of the same encryption key on the input and output. A nonce of length 1 with a value of zero would suffice.

19.6.3.4 Nonce Binding

A command may have sessions other than those required for authorization. One use of an extra session is to encrypt a command or response parameter. If an extra encrypting session were removed by an attacker, the TPM would not properly encrypt/decrypt the data and could, as a result, fail to encrypt a response parameter. To prevent removal of extra encrypting sessions, the *nonceTPM* of each of these sessions is included in the HMAC computation of the first authorization session of a command. If an extra session is removed by an attacker, the first authorization will fail and the command will not be executed.

To simplify the logic in the TPM, the *nonceTPM* of any session used for encryption of command or response data is included in the HMAC computation for the first session even if the encrypt or decrypt session is also an authorization session.

NOTE If the first session is a password authorization, then the path to the TPM is trusted and there is no need to guard against the extra session being removed, also there is probably no need for parameter encryption when a trusted path is present.

19.6.4 Authorization Values

19.6.4.1 Overview

An object may have a value used to authorize various actions on the object. An authorization session is the mechanism through which a caller proves knowledge of the authorization value (*authValue*) needed to allow an action.

An *authValue* may be sent as a password that does not provide confidentiality (see 19.4), or in an HMAC-based authorization session that can provide confidentiality of the *authValue*.

19.6.4.2 authValue Size

An *authValue* may be as small as zero octets but not larger than the digest size of the algorithm used to compute the Name of the object.

EXAMPLE If the Name algorithm for an object is SHA256, then the largest *authValue* for the object would be 32 octets.

19.6.4.3 Authorization Size Convention

When an *authValue* is based on a password or passphrase, then the *authValue* should be the password/phrase as long as the password/phrase is no larger than the digest produced by the *nameAlg* of the object.

EXAMPLE If the passphrase is “This is a sample passphrase”, and *nameAlg* is TPM_ALG_SHA256, then the *authValue* is 27 octets long containing the value “This is a sample passphrase”.

Trailing octets of zero are to be removed from any string before it is used as an *authValue*.

If the password/phrase, with trailing zeros removed, is longer than the digest produced by the *nameAlg* of the object, then the password/phrase – with trailing octets of zero removed – is hashed using *nameAlg* and the resulting hash given to the TPM as the *authValue* for the object.

NOTE Compliance with various security standards (such as, FIPS) could require that the object creator provide an *authValue* that has a size that is at least L/2 where L is the size of the hash algorithm digest. There is no TPM enforcement of this requirement.

19.6.5 HMAC Computation

The HMAC computation for all session types is the same. A *sessionKey* value is concatenated to an *authValue* to create the key that is used in the computation of the HMAC in a command or response. If *sesssionkey* and *authvalue* are both the Empty String, see 19.6.5.

$$authHMAC := \mathbf{HMAC}_{sessionAlg}((sessionKey \,||\, authValue), (\mathrm{pHash} \,||\, \mathrm{nonceNewer} \,||\, nonceOlder \{ \,||\, nonceTPM_{decrypt} \} \{ \,||\, nonceTPM_{encrypt} \} \,||\, sessionAttributes)) \quad (18)$$

where

HMAC$_{sessionAlg}$ — the HMAC function using the hash algorithm specified when the session was started

sessionKey — a value that is computed in a protocol-dependent way, using **KDFa**(). When used in an HMAC or KDF, the size field for this value is not included.

authValue — a value that is found in the sensitive area of an entity. This value is an EmptyAuth if the HMAC is being computed to authorize an action on the object to which the session is bound. The size field for this value is not included in any KDF or hash function.

NOTE 1 For policy sessions, the *authValue* is not included in the HMAC calculation unless the policy session included TPM2_PolicyAuthValue() and it was not superceded by TPM2_PolicyPassword().

NOTE 2 Trailing zeros are always removed from an *authValue* before it is used in an authorization computation.

pHash — digest of the command (cpHash) or response parameters (rpHash) using the session hash algorithm.

nonceNewer — a value that is generated by the entity using the session. A new nonce is generated on each use of the session. For a command, this will be nonceCaller and for a response, nonceTPM. The nonce size field is not included in the HMAC.

nonceOlder — a value that was received the previous time the session was used. For a command, this will be nonceTPM and for a response, nonceCaller. The nonce size field is not included in the HMAC.

nonceTPM$_{decrypt}$ — in the HMAC computation for the first authorization session of a command, if a different session is being used for parameter decryption, then the nonceTPM for that session is included in the HMAC of the first authorization session; but only in the command (see 19.6.3.4). The nonce size field is not included in the HMAC.

NOTE 3 The *decrypt* session is used by the TPM to decrypt a parameter in the command.

NOTE 4 The nonce of the *decrypt* session is included even if that session is also used for authorization.

nonceTPM$_{encrypt}$ — in the HMAC computation for the first authorization session of a command, if a different session is being used for parameter encryption, then the nonceTPM for that session is included in the HMAC of the first

authorization session; but only in the command (see 19.6.3.4). The nonce size field is not included in the HMAC.

NOTE 5 The *encrypt* session is used by the TPM to encrypt a parameter in the response.

NOTE 6 The nonce of the *decrypt* session is included even if that session is also used for authorization.

sessionAttributes an octet indicating the attributes associated with a particular use of the session

With the exception of *sessionAttributes*, all the values are large numbers, typically with sizes of 20 octets or more.

In the HMAC computation equations shown below, the possibility that the HMAC computation may include $nonceTPM_{decrypt}$ or $nonceTPM_{encrypt}$ is indicated by "*nonceOlder**" (asterisk added).

19.6.6 Note on Use of Nonces in HMAC Computations

In equation (18), and the HMAC computation equations that follow, all of the nonce values are in TPM2B_NONCE data structures. In the HMAC computations, the nonce entries should all be read as if they had the *.buffer* suffix indicating that only the data portion of a nonce is ever used in an HMAC computation.

19.6.7 Starting an Authorization Session

TPM2_StartAuthSession() is used to start an authorization session. The parameters of this command may be chosen to produce sessions with different properties.

Table 17 — Schematic of TPM2_StartAuthSession Command

Type	Name	Description
TPM_ST	tag	
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StartAuthSession
TPMI_DH_OBJECT+	tpmKey	handle of a loaded key used to encrypt *salt* may be TPM_RH_NULL Auth Index: None
TPMI_DH_ENTITY+	bind	entity providing the *authValue* may be TPM_RH_NULL Auth Index: None
TPM2B_NONCE	nonceCaller	initial *nonceCaller,* sets nonce size for the session
TPM_SE	sessionType	indicates the type of session (HMAC or policy)
TPM2B_ENCRYPTED_SECRET	encryptedSalt	*tpmKey* algorithm-dependent secret if *tpmKey* is TPM_RH_NULL, this shall be an Empty Buffer

TPMT_SYM_DEF+	symmetric	the algorithm and key size for parameter encryption may select TPM_ALG_NULL
TPMI_ALG_HASH	authHash	hash algorithm to use for the session; and shall be a hash algorithm implemented on the TPM and not TPM_ALG_NULL

The two values that determine the session protection values are *tpmKey* and *bind*. Both of these handles can reference TPM_RH_NULL or a TPM entity. The *tpmKey* parameter references the key that is used to encrypt a salt value that is used in the computation of the *sessionKey*. The *bind* parameter references a TPM entity that may provide an *authValue* to the computation for the *sessionKey*. The four variations for *tpmKey* and *bind* give sessions with different properties.

Table 18 — Handle Parameters for TPM2_StartAuthSession

tpmKey	bind	session properties
TPM_RH_NULL	TPM_RH_NULL	Unbound session
TPM_RH_NULL	TPM entity	Bound session
TPM key	TPM_RH_NULL	Salted session
TPM key	TPM entity	Salted and bound session

19.6.8 *sessionKey* Creation

A *sessionKey* value is used in the HMAC computation as shown in equation (18). If both *tpmKey* and bind are TPM_RH_NULL, then *sessionKey* is set to an Empty Buffer. Otherwise, the *sessionKey* is created as follows:

$$sessionKey := \mathbf{KDFa}(sessionAlg, (authValue \,||\, salt), \text{“ATH”}, nonceTPM, nonceCaller, bits) \quad (19)$$

where

sessionAlg	a TPM_ALG_ID for a hash that was chosen by the caller when the session was started
authValue	if *bind* is not TPM_RH_NULL, a TPM2B_AUTH.buffer that is found in the sensitive area of a TPM entity; otherwise, an Empty Buffer
salt	if *tpmKey* is not TPM_RH_NULL, then the salt value recovered from *encryptedSalt*; otherwise, an Empty Buffer
“ATH”	a four-octet label value (see clause 5.4 for the definition of this label)
nonceTPM	a TPM2B_NONCE that is generated by the TPM when the session was started
nonceCaller	a TPM2B_NONCE that is provided by the caller when the session was started.

bits the number of bits returned is the size of the digest produced by *sessionAlg*

NOTE When an authorization failure occurs, the TPM will check to see if the use of the object is exempt from dictionary attack protection. If it is exempt, the response code is changed from TPM_RC_AUTH_FAIL to TPM_RC_BAD_AUTH and no increment of the failed authorization counter occurs (see 19.11).

19.6.9 Unbound and Unsalted Session Key Generation

In this session key generation method used by TPM2_StartAuthSession(), *tpmKey* and *bind* are both TPM_RH_NULL. This results in the session having no *sessionKey* (it is an Empty Buffer). The session is not bound to any object.

NOTE This session type is similar to the OIAP session in ISO/IEC 11889 (first edition).

A session started using this format can be used for parameter encryption while executing TPM commands. However, during these commands, the key used to encrypt the parameter will only use the *authValue* of the object being accessed by the commands in the key generation, so the strength of the encryption will be no better than the entropy in the *authValue* of the object.

When computing the HMAC, the *authValue* of the referenced entity is used:

$$authHMAC := \mathbf{HMAC}_{sessionAlg}(authValue_{entity}.buffer, (pHash \,||\, nonceNewer.buffer \,||\, nonceOlder^{*}.buffer \,||\, sessionAttributes)) \quad (20)$$

If the size of authValue is zero, then the caller may omit the HMAC from the authorization (see 19.6.15).

Table 19 — Format to Start Unbounded, Unsalted Session

Type	Name	Description
TPM_ST	tag	
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StartAuthSession
TPMI_DH_OBJECT+	tpmKey	TPM_RH_NULL
TPMI_DH_ENTITY+	bind	TPM_RH_NULL
TPM2B_NONCE	nonceCaller	initial *nonceCaller,* sets *nonceTPM* size for the session
TPM2B_ENCRYPTED_SECRET	encryptedSalt	00 00_{16}
TPM_SE	sessionType	indicates the type of the session (HMAC, policy, or trial)
TPMT_SYM_DEF+	symmetric	will normally be TPM_ALG_NULL for an unbound and unsalted session
TPMI_ALG_HASH	authHash	hash algorithm to use for the session; required to be a hash algorithm implemented on the TPM and not TPM_ALG_NULL

NOTE When *sessionType* is TPM_SE_TRIAL, there is no benefit in using any other version of TPM2_StartAuthSession() as a trial session is not allowed to be used for authorization. This means that the *sessionKey* of the session will never be used so there is no point in having the TPM generate it.

19.6.10 Bound Session Key Generation

In this session key generation method used by TPM2_StartAuthSession(), *tpmKey* is TPM_RH_NULL indicating that no *salt* value is present but *bind* references some TPM entity with an *authValue*.

NOTE 1 This session type has properties that are similar to an OSAP session in ISO/IEC 11889 (first edition).

The *sessionKey* is computed using the *authValue* from *bind* and an Empty Buffer in place of the *salt* value.

$$sessionKey := \mathbf{KDFa}\ (sessionAlg, authValue_{bind}, \text{“ATH”}, nonceTPM, nonceCaller, bits) \quad (21)$$

NOTE 2 If handle references a TPM resource that has an EmptyAuth, the *sessionKey* is still computed.

When performing an HMAC for authorization, the HMAC key is normally the concatenation of the entity's *authValue* to the sessions *sessionKey* (created at TPM2_StartAuthSession). See (22). However, if the authorization is for the entity to which the session is bound, the authValue is not included in the HMAC key. See (23). When a policy requires that an HMAC be computed, it is always done according to (22).

$$authHMAC := \mathbf{HMAC}_{sessionAlg}\ ((sessionKey \,||\, authValue_{entity}),\ (pHash \,||\, nonceNewer \,||\, nonceOlder \,||\, sessionAttributes)) \quad (22)$$

$$authHMAC := \mathbf{HMAC}_{sessionAlg}\ (sessionKey,\ (pHash \,||\, nonceNewer \,||\, nonceOlder^{*} \,||\, sessionAttributes)) \quad (23)$$

The TPM is required to keep track of the entity to which the session is bound. This is nominally accomplished when the session is started by recording, in the session context, the Name of the *bind* entity. For an NV Index or persistent handle, the TPM is required to also record the authorization value associated with the entity.

NOTE 3 Recording of the NV Index authorization is needed to prevent an attacker from "squatting" on an Index. This would be accomplished by creating an NV Index that has properties that are identical to an NV Index that is expected to be created, but with an authorization value known to the attacker. The attacker would then start an authorization session bound to the NV Index and delete the NV Index. When the NV Index to be attacked is created, the attacker would have an authorization session bound to an index with the same Name and could access to the NV Index even though the actual authorization value is unknown.

On a command, the TPM will check to see if the authorization is being used for the entity to which it was bound. If so, then the *authValue* of the bound entity is not used in the HMAC computation. The TPM will record the fact that the *authValue* was not used in the HMAC computation of the authorization and not include it in the HMAC computation on the response.

NOTE 4 This allows the session to remain bound to an NV Index for the duration of the first command that writes to the Index even though the Name of the Index changes during the command processing. The session will not be bound to the Index when the command completes. The session can continue to be used, but it, in effect, is no longer bound because there is no longer a TPM entity with the correct Name.

For a persistent object, the authorization value is included so that authorization can be revoked. If the administrator for a persistent object changes the authorization, sessions bound to the old authorization should no longer be valid.

NOTE 5 To change the authorization of a persistent object, TPM2_ObjectChangeAuth() would be called. It would return a new sensitive area. The current persistent object would be deleted (TPM2_EvictControl()) and the object with the new authorization loaded (TPM2_Load()). Finally, the loaded object would be made persistent (TPM2_EvictControl()). It is only required that the old object be deleted if the new object is to have the same handle or if it is desired to revoke the old authorization.

The *noDA* attribute of the bind object is recorded in the session context. For a description of the rationale, see clause 19.11.7.

Table 20 — Format to Start Bound Session

Type	Name	Description
TPM_ST	tag	
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StartAuthSession
TPMI_DH_OBJECT+	tpmKey	TPM_RH_NULL
TPMI_DH_ENTITY	bind	entity providing the *authValue* to which the session is bound and not TPM_RH_NULL
TPM2B_NONCE	nonceCaller	initial *nonceCaller*, sets *nonceTPM* size for the session
TPM2B_ENCRYPTED_SECRET	encryptedSalt	00 00_{16}
TPM_SE	sessionType	indicates the type of the session (HMAC, policy, or trial)
TPMT_SYM_DEF+	symmetric	if the session is to be used for parameter encryption, set this to an algorithm and key size
TPMI_ALG_HASH	authHash	hash algorithm to use for the session; required to be a hash algorithm implemented on the TPM and not TPM_ALG_NULL

19.6.11 Salted Session Key Generation

In this session key generation method used by TPM2_StartAuthSession(), *bind* is TPM_RH_NULL, indicating that no entity is referenced to provide an *authValue,* but *tpmKey* is present and indicates a key used to encrypt the *salt* value. The *sessionKey* is computed with an Empty Buffer in place of the *authValue*.

$$sessionKey := \mathbf{KDFa}\ (sessionAlg,\ salt,\ \text{“ATH”},\ nonceTPM,\ nonceCaller,\ bits) \tag{24}$$

Because *bind* is TPM_RH_NULL, the session is not bound to any entity. When the session is used to access any entity, the HMAC will use the *sessionKey* and the *authValue* of that entity.

$$\begin{aligned} authHMAC := \mathbf{HMAC}_{sessionAlg}\ ((sessionKey\ ||\ authValue_{entity}),\\ (pHash\ ||\ nonceNewer\ ||\ nonceOlder^{*}\\ ||\ sessionAttributes)) \end{aligned} \tag{25}$$

Table 21 — Format to Start Salted Session

Type	Name	Description
TPM_ST	tag	
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StartAuthSession
TPMI_DH_OBJECT	tpmKey	handle of a loaded key used to encrypt *salt*
TPMI_DH_ENTITY+	bind	TPM_RH_NULL
TPM2B_NONCE	nonceCaller	initial *nonceCaller,* sets *nonceTPM* size for the session
TPM2B_ENCRYPTED_SECRET	encryptedSalt	conveys a secret value used to generate the *sessionKey* – method of conveying this value is dependent on the type of *tpmKey*
TPM_SE	sessionType	indicates the type of the session (HMAC, policy, or trial)
TPMT_SYM_DEF+	symmetric	if the session is to be used for parameter encryption, set this to an algorithm and key size
TPMI_ALG_HASH	authHash	hash algorithm to use for the session; required to be a hash algorithm implemented on the TPM and not TPM_ALG_NULL

19.6.12 Salted and Bound Session Key Generation

This version of TPM2_StartAuthSession() creates a session that has properties that are similar to the OSAP session type of ISO/IEC 11889 (first edition) but also allows salting. For this version of the command, *bind* is used to provide an *authValue*, *tpmKey* encrypts the *salt* value and the *sessionKey* is computed using both.

$$sessionKey := \mathbf{KDFa}\,(sessionAlg, (authValue_{bind} \,||\, salt), \text{“ATH”}, nonceTPM, nonceCaller, bits) \quad (26)$$

If the session is an HMAC session:

Because *bind* is present, the session is bound to that entity. That is, when the session is used to authorize use of the bound entity, the HMAC will use *sessionKey* but not the *authValue*.

$$authHMAC := \mathbf{HMAC}_{sessionAlg}\,(sessionKey, (pHash \,||\, nonceNewer \,||\, nonceOlder^{*} \,||\, sessionAttributes)) \quad (27)$$

If the session is a policy session:

The session is not bound to that entity. That is, when the session is used to authorize use of any entity, the HMAC (if required) will use the *sessionKey* and the *authValue*.

$$authHMAC := \mathbf{HMAC}_{sessionAlg}\,((sessionKey \,||\, authValue_{entity}), (pHash \,||\, nonceNewer \,||\, nonceOlder \,||\, sessionAttributes)) \quad (28)$$

The *noDA* attribute of the bind object is recorded in the session context. For a description of the rationale, see clause 19.11.7.

Table 22 — Format to Start Salted and Bound Session

Type	Name	Description
TPM_ST	tag	
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StartAuthSession
TPMI_DH_OBJECT+	tpmKey	handle of a loaded key used to encrypt *salt*
TPMI_DH_ENTITY	bind	entity providing the *authValue* and to which the session is bound
TPM2B_NONCE	nonceCaller	initial *nonceCaller*, sets *nonceTPM* size for the session
TPM2B_ENCRYPTED_SECRET	encryptedSalt	contains a secret value used to generate the *sessionKey* – method of encrypting this value is dependent on the type of *tpmKey*
TPM_SE	sessionType	indicates the type of the session (HMAC, policy, or trial)
TPMT_SYM_DEF+	symmetric	if the session is to be used for parameter encryption, set this to an algorithm and key size
TPMI_ALG_HASH	authHash	hash algorithm to use for the session; required to be a hash algorithm implemented on the TPM and not TPM_ALG_NULL

19.6.13 Encryption of *salt*

19.6.13.1 Overview

The *salt* parameter for TPM2_StartAuthSession() may be symmetrically or asymmetrically encrypted using the methods specified in clause 19.6.13.

The value produced by the secret exchange process using *salt* should be the size of the digest produced by the *authHash* of the session. For ECC, the size of the *seed* is limited because it is an ECC point; but for RSA, XOR, and AES, the size of *salt* may vary.

When the value of salt is determined, it is used in the computation of *sessionKey* as shown in equation (19).

19.6.13.2 Asymmetric Encryption of Salt

The methods of encrypting the salt and producing the session secret differ for each asymmetric algorithm. The methods are specified in the algorithm-specific annexes to this part of ISO/IEC 11889.

19.6.13.3 XOR obfuscation of Salt

When *tpmKey* is an XOR key, the operation used for removing the obfuscation from *salt* is:

$$\mathbf{XOR}(encryptedSalt,\ hashAlg,\ key,\ nonceCaller,\ nullNonce) \tag{29}$$

where

encryptedSalt	the parameter in TPM2_StartAuthSession()
hashAlg	the hash algorithm use to compute the Name of tpmKey
key	the symmetric secret HMAC value in the object referenced by tpmKey in TPM2_StartAuthSession()
nonceCaller	the parameter from TPM2_StartAuthSession()
nullNonce	the Empty Buffer

NOTE 1 XOR obfuscation is defined in 11.4.6.4.

NOTE 2 *NonceTPM* is not used in this call as it had not yet been generated when the caller had to compute the XOR mask.

All of the bits of *salt* are used as *sessionKey*.

19.6.13.4 Symmetric Block Cipher Encryption of Salt

If *tpmKey* is a symmetric block encryption key, CFB-mode encryption is used. The *seed* value is CFB-encrypted using *nonceCaller* as the Initialization Vector (IV). If *nonceCaller* is larger than the block size of the cipher, it is truncated (high-order octets retained). If it is smaller than the block size, it is padded to the right (least-significant end) with octets of zero. All of the decrypted octets are used as *sessionKey*.

19.6.14 Caution on use of Unsalted Authorization Sessions

If an authValue has low entropy, confidentiality of the value may not be preserved if the authValue is used in an unsalted authorization session. For an unbound, unsalted session, the HMAC computation for the response from the TPM is:

$$authHMAC := \mathbf{HMAC}_{sessionAlg}(authValue, (rpHash \,||\, nonceTPM \,||\, nonceCaller \,||\, sessionAttributes)) \quad (30)$$

If an attacker can read the response from the TPM, then the only values unknown to the attacker are *authValue* and *nonceCaller*. An attacker may be able to determine *nonceCaller* by reading the command as it is sent to the TPM. If the attacker has all the variables but *authValue*, they could perform an "off-line" attack on the *authValue* using trial versions of *authValue* until one is found that produces a matching *authHMAC*.

NOTE 1 In this context, an "off-line" attack means that the attacker can perform computations that do not involve the TPM meaning that the protections that the TPM provides against authValue attacks has no effect.

It is important to note that this vulnerability only occurs if an attacker has access to both the command and response of a successful command using the *authValue*. If a user has a password protecting a key and the system is lost or stolen, the key is protected because the attacker will not be able to observe the legitimate owner of the key perform a successful operation with the key.

For a bound session without salt, the attack is a bit more complicated. The HMAC computation for the response is:

$$authHMAC := \mathbf{HMAC}_{sessionAlg}((sessionKey \,||\, authValue_{entity}), (pHash \,||\, nonceNewer \,||\, nonceOlder \,||\, sessionAttributes)) \quad (31)$$

If the attacker observes a TPM2_StartAuthSession() command and response and the authValue for the bind entity has low entropy, then they would have all of the components of *sessionKey* except for the *authValue* of the *bind* entity. Then, by observing another successful transaction, an attacker could know everything but the two *authValues* and they could again perform an offline attack.

NOTE 2 If the successful operation is on the *bind* entity, then only one *authValue* is unknown.

As with the unbound and unsalted session, the vulnerability for a bound session only occurs if the attacker is able to observe successful command response sequences.

Salting provides a mechanism to allow use of low entropy *authValues* and still maintain confidentiality for the authValue. It is also possible to use a high entropy authValue to protect the confidentiality of a low-entropy value.

EXAMPLE If the *bind* entity *authValue* has high-entropy, then there would be greater computational complexity in guessing *sessionKey* || $authValue_{entity.}$ Depending on the *authValue* and *salt* sizes, a bound session could have a *sessionKey* that is as difficult to guess as does a salted session..

19.6.15 No HMAC Authorization

For a session-based authorization, both HMAC and policy, an *authHMAC* value is computed as shown in equation (18) and that value is used as *hmac* in an authorization or acknowledgement as shown in Table 15 and Table 16 respectively. If an authorization session is started with *bind* and *tpmKey* both set to TPM_RH_NULL, then *sessionKey* in equation (18) will be an Empty String. If the *authValue* in equation (18) is also an Empty String, then the HMAC key will be an Empty Buffer. When this situation exists, the caller has the option of either providing the results of the *authHMAC* computation, or not.

If *authHMAC* is provided, it will be computed as shown in equation (18) with an Empty Buffer as the HMAC key and the TPM will validate that the value in *hmac* matches the internally calculated value.

If *authHMAC* is not provided, the size of *hmac* (see Table 15) will be zero and the TPM will accept this value of *hmac* as providing valid authorization for the object.

For an HMAC session, *authValue* in equation (18) will only be an Empty String if the *authValue* of the authorized object is an EmptyAuth.

For a policy session, two situations will result in *authValue* being an Empty String:

1) the *authValue* of the authorized object is an EmptyAuth, or
2) the policy does not use the *authValue* of the object (that is, the evaluated policy does not contain TPM2_PolicyAuthValue())(see 19.7.6.6).

For these two cases, *hmac* is allowed to be either a valid *authHMAC* or an Empty String.

The TPM will use the same formulation in the response as was in the command. This is, if *hmac* was non-zero in the command, the TPM will compute *authHMAC* as shown in equation (18) and use the result as *hmac*. If *hmac* was an Empty Buffer in the command, it will be an Empty Buffer in the response.

19.6.16 Authorization Selection Logic for Objects

Each object has two attributes in its public structure to indicate how use of the object is authorized.

1) ***userWithAuth*** – If this attribute is SET, then USER role authorization for an object may be provided with an HMAC session or a password. If this attribute is CLEAR, then the *authValue* may not be used for USER role authorization, meaning that authorization may not be done using an HMAC session or a password. USER role authorizations with a policy are always allowed regardless of the setting of this attribute.
2) ***adminWithPolicy*** – If this attribute is SET, then ADMIN role authorization for an object may only be provided with a policy session. If this attribute is CLEAR, then authorization may be provided with a policy session, with an HMAC session, or with a password.

When authorization is with a policy session and ADMIN role authorization is being provided, the command code value of the policy session must match the command code for the command being authorized.

For TPM_RH_OWNER, TPM_RH_ENDORSEMENT, and TPM_RH_PLATFORM); *userWithAuth* and *adminWithPolicy* are always SET.

For an NV Index, NV index attributes (TPMA_NV) determine authorization selection.

NOTE For TPM_RH_OWNER, TPM_RH_ENDORSEMENT, and TPM_RH_PLATFORM); *userWithAuth* and *adminWithPolicy* do not have to be implemented as separate attributes. The code may simply assume that the attributes are SET and act accordingly.

19.6.17 Authorization Session Termination

The TPM will terminate a session (authorization or audit) and clear all associated context under the following circumstances:

- when TPM2_FlushContext() selects the session;

- if *sessionAttributes.continueSession* is CLEAR in the command, the TPM will CLEAR the continueSession flag in the response and perform TPM2_FlushContext() actions;

  NOTE When *sessionAttributes.continueSession* is CLEAR in the command but the command does not return success, then the session is not terminated.

- when the TPM executes TPM2_Startup(TPM_SU_CLEAR), all authorization sessions are terminated; and
- when the TPM executes TPM2_Startup(TPM_SU_STATE), authorization sessions in TPM memory will be terminated but sessions stored off the TPM will remain active.

19.7 Enhanced Authorization

19.7.1 Introduction

Enhanced authorization is a TPM capability that allows entity-creators or administrators to require specific tests or actions to be performed before an action can be completed. The specific policy is encapsulated in a value called an *authPolicy* that is associated with an entity

When an HMAC session is used for authorization, the *authValue* of the entity is used to determine if the authorization is valid. When a policy session is used for authorization, the *authPolicy* of the entity is used.

Many TPM entities have or may have an associated *authPolicy*. A policy defines the conditions for use of an entity.

EXAMPLE 1 A policy might limit the use of a key unless selected PCR have specific values.

EXAMPLE 2 A policy might not allow use of a key after a specific time.

EXAMPLE 3 A policy might require that authorization to change an NV Index be provided by two different entities.

EXAMPLE 4 A policy might limit a particular signing key to attest to PCR values but not to certify another TPM key.

A policy may be arbitrarily complex. However, the policy is expressed as one (statistically unique) digest called the *authPolicy*.

The digest representing a particular policy may be included in an Object or NV Index when the Object or NV Index is created (the digest representing a policy is created using the methods specified in subsequent parts of clause 19.7). In order to use the Object or Index, a policy session is created and then the TPM is given a sequence of policy commands that modify the digest in the policy session. After executing all of the commands of the policy, the TPM will have computed a digest value that is characteristic of the policy. The policy session is then used as an authorization session. If the digest accumulated in the policy session matches the *policyDigest* of the entity (and certain other optional conditions are true) then the command is authorized.

After a policy session is used for authorization, *policySession→nonceTPM* is changed to a new, random value; *policySession→startTime* is set to the current time; and the other values of the policy session context are initialized to the state they had when the session was first created by TPM2_StartAuthSession() (see 19.7.7).

The mechanisms of policy creation and evaluation are explained in the remainder of clause 19.7.

19.7.2 Policy Assertion

An assertion is a statement that something is true. In an authorization policy, an assertion is a statement of something that must be true before the policy is satisfied. The list of all policy assertions defined by ISO/IEC 11889 is in 19.7.6.6.

EXAMPLE An assertion might be that a set of PCR must have specific values to allow an object to be authorized for use in a specific command..

A combination of one or more assertions is used to construct an authorization policy.

19.7.3 Policy AND

A policy may be expressed in an equation as a set of assertions that must all be satisfied before the policy is valid.

EXAMPLE 1 A policy that requires that 4 assertions be true could be written as:

$$a \,\&\, b \,\&\, c \,\&\, d \tag{32}$$

A possible implementation of the policy logic in the example above would be to have all the assertions evaluated at the same time to determine if the policy is satisfied. This approach would require that the TPM resources scale with the number of assertions that would need to be evaluated for the policy.

The alternative use in the TPM is to evaluate the expression one assertion at a time with each assertion ANDed with the results of the previous evaluation.

EXAMPLE 2 A policy example requiring the TPM to evaluate the expression one assertion at a time with each assertion ANDed with the results of the previous evaluation could be written as:

$$(((1 \,\&\, a) \,\&\, b) \,\&\, c) \,\&\, d \tag{33}$$

In the example above, the (1 & *a*) term means that assertion *a* is ANDed with an initial TRUE. This allows each assertion to be just the AND of a new assertion with the results of the previous assertion evaluation. A pictorial representation of the policy evaluation is in Figure 15.

Figure 15 — A Policy Evaluation

Any number of assertions can be combined in this way using a fixed set of TPM resources.

The logic of a TPM policy cannot actually be expressed as a simple 1 or 0. For the policy to be valid, not only does it need to evaluate to "TRUE" but it also has to be the correct policy.

EXAMPLE 3 Two policies may both evaluate to the same logic value (TRUE), but they do not represent the same policies.

A pictorial representation of the example above is in Figure 16.

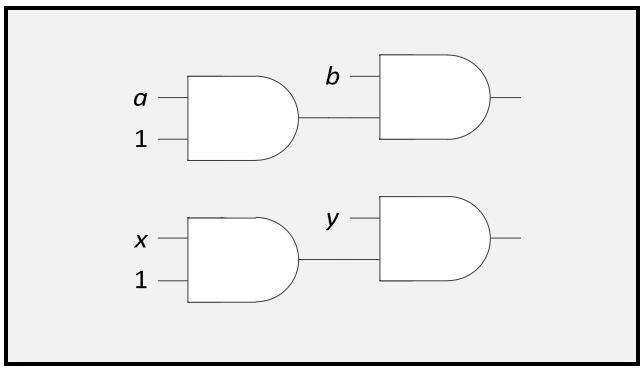


Figure 16 — Two Different Policy Expressions

So that it can differentiate (*a* & *b*) from (*x* & *y*) in Figure 16, the TPM will update a running digest value for each assertion that is added to the policy. The final digest value indicates the policy that was evaluated.

The running digest value is called the *policyDigest*. The *policyDigest* is initialized to a Zero Digest (0…0) when the policy session is started (TPM2_StartAuthSession()). Then, as each policy assertion is evaluated, the *policyDigest* is updated.

$$policyDigest_{new} := \mathbf{H}(policyDigest_{old} \,||\, PolicyAssertion) \qquad (34)$$

NOTE 1 This equates to the Extend operation.

The *policyDigest* will only be updated if a policy assertion is valid (TRUE) (see 19.7.9 for exception relating to trial policies). This gives an alternative possibility for interpreting the output of one of the policy AND gates. Instead of simply being a 1 (TRUE) or 0 (FALSE), the output of the gate is current value of the *policyDigest*. Using this perspective, the four-term policy is shown in Figure 17:

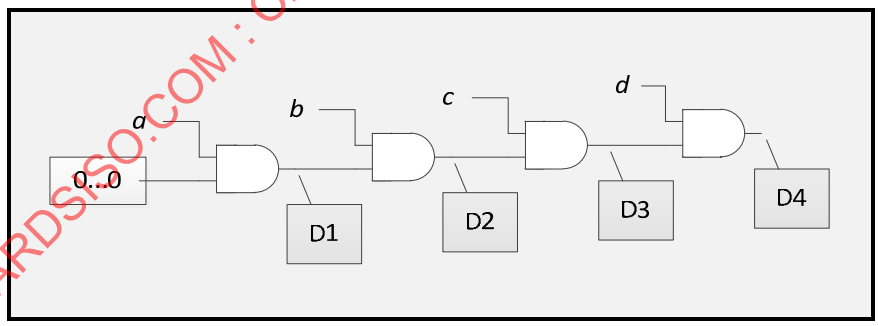


Figure 17 — A Four-Term Policy

where

0…0	the initial value of the policy digest
D1	**H**(0…0 || *a*)
D2	**H**(D1 || *b*)
D3	**H**(D2 || *c*)
D4	**H**(D3 || *d*)

NOTE 2 In these illustrations, the parameters for the Extend operations are simple parameters ("*a*", "b", etc.). The actual parameters for the Extend are more complex but including the details in the illustrations would add complexity without adding clarity.

19.7.4 Policy OR

If the only type of policy assertion was an AND, then the policies that could be evaluated by the TPM would be of limited value. To make the policies more flexible, an OR policy assertion is defined. As with a logic OR gate, the OR policy assertion will be valid if any of the inputs is valid.

A simple policy using an OR might be written as:

$$(\mathrm{a}\ \&\ \mathrm{b}) \mid (\mathrm{x}\ \&\ y) \tag{35}$$

or as

$$(((0...0)\ \&\ \mathrm{a})\ \&\ \mathrm{b}) \mid (((0...0)\ \&\ \mathrm{x})\ \&\ \mathrm{y}) \tag{36}$$

Evaluating the AND branches individually, the left side evaluates to:

$$\mathrm{D}_{\mathrm{left}} := \mathbf{H}(\mathbf{H}(0...0 \mid\mid a) \mid\mid b) \tag{37}$$

and the right side to:

$$\mathrm{D}_{\mathrm{right}} := \mathbf{H}(\mathbf{H}(0...0 \mid\mid x) \mid\mid y) \tag{38}$$

Then, the output from a 2-input policy OR operation will be defined to be

$$policyDigest_{new} := \mathbf{H}(\mathrm{D}_{\mathrm{left}} \mid\mid \mathrm{D}_{\mathrm{right}}) \tag{39}$$

Notice that the OR operation replaces the *policyDigest* with a new value instead of Extending it as is done in an AND operation.

Pictorially, a policy with an OR is shown in Figure 18:

Figure 18 — Policy with an OR

The TPM processes the OR by comparing the current value of *policyDigest* with a list of digest values provided by the caller. If *policyDigest* is on the list, then the TPM will digest the concatenation of all of the digests in the list.

EXAMPLE 1 To perform the OR operation above, assume that the TPM has processed (*a* & *b*) producing D_{left}. Then the TPM would be given a list of digests (D_{left}, D_{right}). Because the *policyDigest* is on the list, the TPM computes $D_{OR} := \mathbf{H}(D_{left} \mid D_{right})$ and replaces *policyDigest*. If the TPM had processed (*c* & *d*) to compute D_{right}, and was then given the same list of digests (D_{left}, D_{right}), the resulting *policyDigest* would be the same.

When processing a policy that has an OR, only one branch of the policy needs to be evaluated.

EXAMPLE 2 If C and D assertions were valid, then only the *right* branch would need to be evaluated.

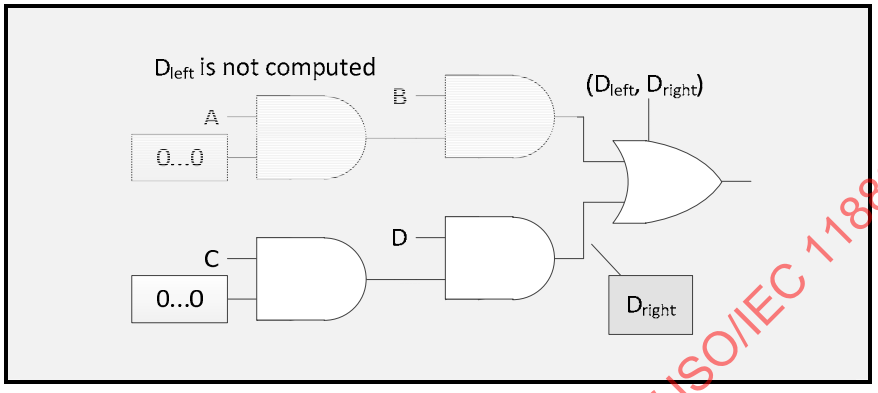


Figure 19 — Policy where only one OR Branch is Evaluated

The list given to the TPM for a TPM2_PolicyOR() is limited to 8 digests. However, the effective size of the list can be expanded indefinitely by using cascading OR.

EXAMPLE 3 Figure 20 illustrates one of the many ways to construct a 12 input OR.

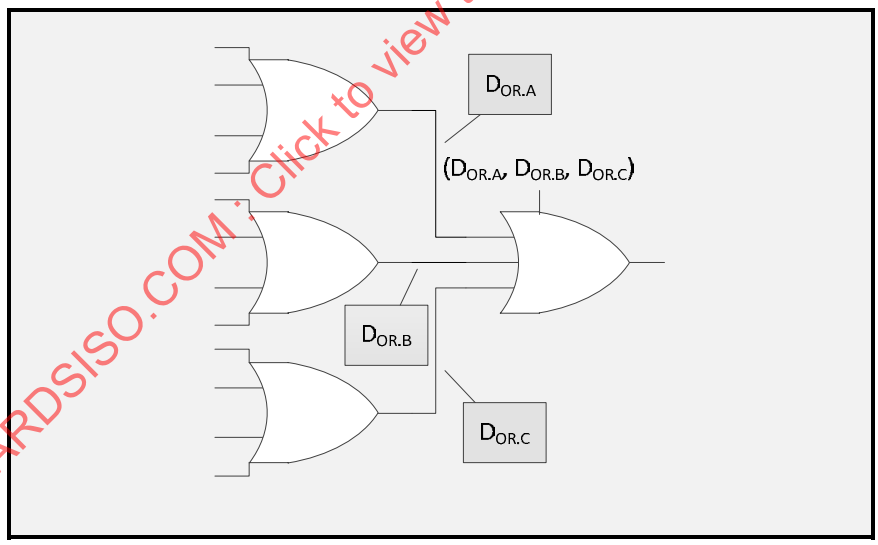


Figure 20 — A 12-input OR Policy

When the OR list can contain 8 digests, 64 different branches can be ORed in just two levels.

The result of an OR operation may be an input to an AND assertion allowing construction of arbitrarily complex policies.

19.7.5 Order of Evaluation

Because the TPM uses digests, the order of evaluations is important. For policy evaluation, (A & B) is not the same as (B & A). In addition, when performing an OR operation, the same list of digests (same number in the same order) must be given to the TPM each time. The list (D_{left}, D_{right}) will not give the same result as (D_{right}, D_{left}) or (D_{left}, D_{right}, D_{other}).

19.7.6 Policy Assertions (Policy Commands)

19.7.6.1 Introduction

In ISO/IEC 11889-3 the set of policy assertions are the commands with names of the form TPM2_Policyxxx() where "xxx" is an indicator of the type of policy assertion.

EXAMPLE 1 TPM2_PolicySigned() is a policy assertion that an authorization was signed by a specific entity

EXAMPLE 2 TPM2_PolicyPCR() is an assertion that a selected set of PCR have a specific value.

Normally, each policy command will cause the *policyDigest* to be changed in a different way which is why they are different commands. In some cases, the policy command will also cause other changes to the policy session context.

EXAMPLE 3 TPM2_PolicyLocality() modifies the policy state that indicates the locality that is allowed when the policy session is used for authorization.

EXAMPLE 4 TPM2_PolicyCommandCode() changes the policy state so that the policy may only be used to authorize a specific command.

The details of the *policyDigest* computation performed by each policy command are provided in the General Description clause of each command found in ISO/IEC 11889-3. The description also indicates the policy state that is modified.

The assertions fall into three different groups: immediate, deferred, and combined.

19.7.6.2 Immediate Assertions

For an immediate assertion, the input values are validated and the TPM will return a failure and not update the *policyDigest* if the assertion is not valid.

EXAMPLE An example of an immediate assertion is TPM2_PolicyNV(). For this assertion, the TPM validates the logical or arithmetic relationship between an input value and an NV Index. If the specified relationship is not valid, the TPM returns an error and the *policyDigest* is not modified. If the relationship is valid, then the *policyDigest* is updated with the Index Name and the relationship that was validated.

19.7.6.3 Deferred Assertions

For a deferred assertion, the TPM will update the *policyDigest* based on the input values and record some parameters in the policy session's context. These parameters are checked when the policy is used for authorization.

EXAMPLE An example of a deferred assertion is TPM2_PolicyCommandCode(). For this assertion, the input is a TPM command code. The *policyDigest* will be updated to record the fact that the TPM2_PolicyCommandCode() was executed and the *commandCode* value that was specified. The TPM also directly records the *commandCode* parameter in the policy session context. When the policy is used for authorization, the TPM will verify that the command being authorized is the same as the command in the policy and the authorization (and command) will fail if they are not the same.

19.7.6.4 Combined Assertions

For a combined assertion, the TPM will validate some condition of the input and record or modify some parameters in the policy session's context.

EXAMPLE An example of a combined assertion is TPM2_PolicySigned(). For this assertion type, the TPM validates that the parameters of the command have been signed by the indicated key. If so, it will update the policy session context based on the input parameters. One of the context values that may be updated is the *cpHash* of the session. If the *cpHash* of the authorized command is not the same as the authorized *cpHash* then the command will not be authorized.

19.7.6.5 Repetition of Assertions

In general, any policy assertion may occur multiple times within a policy as long as the assertion is compatible with previous assertions.

EXAMPLE 1 An example of an incompatible set of assertions is two occurrences of TPM2_PolicyCommandCode() that specify different command codes.

The TPM will return an error if an assertion is incompatible with a previous assertion. It is possible that the failed assertion is incompatible with an assertion of a different type.

EXAMPLE 2 For example, a TPM2_PolicyCpHash() may be incompatible with a TPM2_PolicySigned(). If they specify different values of *policySession→cpHash,* then the TPM will return an error.

NOTE When referring to an *element* of the policy context, the notation *policySession→element* is used to denote a particular member of the policy context.

19.7.6.6 List of Assertions

The assertions listed in clause 19.7.6.6 will all update the *policyDigest* of the policy session being operated on if the assertion condition is met. They may also cause a change to other policy session, context values (the list of policy session context values is in 19.7.7) as indicated in the brief description for each assertion.

- **TPM2_PolicyAuthorize() –** valid if *policySession→policyDigest* has the value authorized by the selected key. This is an immediate assertion and is specified in more detail in 19.7.10.
- **TPM2_PolicyAuthValue() –** valid if *authValue* of the authorized entity is provided when the policy session is used for authorization. This deferred assertion will SET *policySession→isAuthValueNeeded*. When the policy is used for authorization, the TPM will check *policySession→isAuthValueNeeded*. If it is SET, then the TPM performs an HMAC check on the session as if it were an HMAC session. This HMAC validation will only succeed if the caller is able to prove knowledge of the entity's authValue by computing the correct HMAC.
- **TPM2_PolicyCommandCode() –** valid when the authorized command has the specified command code. This deferred assertion sets *policySession→commandCode*.
- **TPM2_PolicyCounterTimer() –** valid when an portion of the TPM's TPMS_TIME_INFO structure has the desired numerical relationship with another value. This is an immediate assertion. If the selected subset of the TPM's TPMS_TIME_INFO structure does not have the specified relationship with the input data, then the TPM will return an error and not change the *policyDigest*.

  36.1 describes use cases.
- **TPM2_PolicyCpHash() –** valid if the cpHash of the authorized command has a specific value. This deferred assertion *modifies policySession→cpHash*.
- **TPM2_PolicyDuplicationSelect() –** valid if the handles of the authorized command reference specific objects and the command code is TPM2_Duplicate(). This deferred assertion modifies *policySession→cpHash* and *policySession→commandCode*.
- **TPM2_PolicyLocality() –** valid if the command being authorized is being executed at one of the allowed localities. This is a deferred assertion that modifies *policySession→locality*. For localities 0-4, the input locality parameter is a bit field that indicates the allowed localities. If an execution of this assertion would result in no locality being allowed, then the TPM will return an error. For extended localities, *policySession→locality* is set to the *locality* parameter of the command if the *policySession→locality* was not previously set. Otherwise, the *locality* parameter is required to be the same as the current value of *policySession→locality.*
- **TPM2_PolicyNameHash() –** valid if the handles of the authorized command reference specific objects. This deferred assertion modifies *policySession→cpHash.*
- **TPM2_PolicyNV() –** valid if the contents of NV have the desired relationship with another value. This is an immediate assertion. If the selected portion of the NV Index does not have the specified relationship with the input data, then the TPM will return an error and not change the *policyDigest*.
- **TPM2_PolicyOR() –** valid if *policySession→policyDigest* is on a list of digests. This is an immediate assertion. If *policySession→policyDigest* is not on the list of digests, then TPM returns an error. Otherwise, *policySession→policyDigest* is replaced with the digest of the list.
- **TPM2_PolicyPassword() –** valid if the authValue of the authorized entity is provided when the session is used for authorization. This deferred assertion will SET *policySession→isPasswordNeeded*. When the policy is used for authorization, the TPM will check *policySession→isPasswordNeeded*. If it is SET, then the TPM performs a password check on the session as if it were a password session. . This password validation will only succeed if the caller is able to prove knowledge of the entity's *authValue* by providing the correct value as the password.

NOTE 1 A session can use TPM2_PolicyAuthValue() and TPM2_PolicyPassword() interchangeably. If TPM2_PolicyAuthValue() and TPM2_Policy Password() are both used, then TPM will perform the check according to the last one used in the policy.

- **TPM2_PolicyPCR() –** valid if the selected PCR have the desired value. This assertion may be either an combined or a deferred assertion. If the caller provides a digest, the TPM validates that the current values of the PCR match the input value and return an error (TPM_RC_VALUE) if not. If this command completes successfully, the *policyDigest* will have been updated with the digest of the selected PCR. The TPM will also record that the PCR have been checked. If the PCR are changed after they are checked but before the policy is used for authorization, then the policy will fail.

  NOTE 2 The reference implementation provides this assurance by maintaining a PCR update counter that increments each time the PCR are modified. The update counter is saved in the policy session context. If the update counter does not change between the check of the PCR and the use of the policy session for authorization, then the PCR are the same.

- **TPM2_PolicyPhysicalPresence() –** valid if the physical presence is asserted when the authorized command is executed. This deferred assertion sets *policySession→isPPRequired.*
- **TPM2_PolicySecret() –** valid if the knowledge of a secret value is provided. This assertion is an immediate and possibly also a deferred assertion. Based on the input parameters, this command may modify *policySession→cpHash* and *policySession→timeout.*

  NOTE 3 The secret value will be the authValue of some TPM entity.

- **TPM2_PolicySigned() –** valid if the parameters are properly signed. This assertion is an immediate and possibly also a deferred assertion. Based on the input parameters, this command may modify *policySession→cpHash* and *policySession→timeout.*
- **TPM2_PolicyTicket() –** valid if the ticket is valid. This assertion is an immediate and possibly also a deferred assertion. Based on the input parameters, this command may modify policySession→cpHash and policySession→timeout.
- **TPM2_PolicyNvWritten() –** valid when the TPMA_NV_WRITTEN attribute of the specified NV index has the desired value. This deferred assertion sets *policySession→checkNvWritten* and the state of *policySession→nvWrittenState*.

19.7.7 Policy Session Context Values

A policy session context contains the state and tracking information for evaluation of a policy. The context values are set to their default values when the session is created and again each time the session is successfully used to authorize a command.

The values may be changed by a policy assertion. The policy assertions are listed in 19.7.6.6 with an indication of the policy session context values that they modify. The policy session context values are described further here.

- ***policyDigest*** – digest that is updated by each assertion. The default value for *policyDigest* is a Zero Digest (a buffer with a length equal to the digest size of the hash algorithm with all octets having a value of zero).
- ***nonceTPM*** – set from the RNG and is sized according to the size of *nonceCaller* in TPM2_StartAuthSession(). This value does not change during the policy evaluation. However, it does change when the policy session is used for authorization.
- ***cpHash*** – set by an assertion that limits the authorization to a specific set of command parameters. If an assertion would set *policySession→cpHash* and a previous assertion has set *policySession→cpHash* to a different value, then the assertion will fail. The default for *policySession→cpHash* is an Empty Buffer.

- ***nameHash*** – set by TPM2_PolicyNameHash() and indicates the combination of Name values for a command. This context parameter occupies the same location as *policySession→cpHash*. If an assertion would set *policySession→cpHash* and a previous assertion has set *cpHash* to a different value, then the assertion will fail. The default for *policySession→nameHash* is an Empty Buffer.
- ***startTime*** – set to TPMS_TIME_INFO.*clockInfo.clock* when *policySession→nonceTPM* changes. No assertion changes this value. It is updated to the current value of *clock* by TPM2_StartAuthSession() and when the session is used for authorization.
- ***timeout*** – the time when the policy session expires. Its default setting is an implementation-specific value corresponding to "never expires." This value is updated if an assertion has a non-zero expiration time that is sooner than the current setting of *policySession→timeOut*. An assertion may only decrease the value of *policySession→timeout.*
- ***commandCode*** – set by an assertion that limits the policy to a specific command but does not limit the command parameters (TPM2_PolicyCpHash() limits the command and its parameters). If an assertion sets *policySession→commandCode* and a previous assertion has set *policySession→commandCode* to a different value, then the TPM will return an error. The default for *policySession→commandCode* is an implementation-specific value that indicates that it has not been set.
- ***pcrUpdateCounter*** – set by TPM2_PolicyPCR(). The TPM maintains a *pcrUpdateCounter* that is incremented each time a PCR changes (with a few exceptions as specified in 17.9). When it executes TPM2_PolicyPCR(), the TPM will copy *pcrUpdateCounter* to *policySession→pcrUpdateCounter.* When the policy session is used for authorization, the TPM will verify that *policySession→pcrUpdateCounter* matches *pcrUpdateCounter*. A match provides assurance that the PCR values still match the values evaluated by TPM2_PolicyPCR().
- ***commandLocality*** – indicates the locality required for the command being authorized by the policy. The default for *policySession→commandLocality* is any locality. Each locality that is not enabled in TPM2_PolicyLocality(*locality)* is disabled in *policySession→commandLocality*. If the result of this operation would result in there being no locality at which the policy would be valid, the TPM will return an error and not change *policySession→commandLocality*. If *commandLocality* is set to an extended locality (greater than 31), then the locality cannot be change by subsequent TPM2_PolicyLocality().
- ***isPPRequired*** – SET by TPM2_PolicyPhysicalPresence() to indicate that presence is required to be asserted when authorized command is executed. The default value is CLEAR.
- ***isAuthValueNeeded*** – SET by TPM2_PolicyAuthValue()to indicate that the *authValue* of the authorized entity will need to be provided when the policy session is used for authorization. The *authValue* is required to be included in an HMAC. The default value is CLEAR. It will also be CLEAR by TPM2_PolicyPassword()
- ***isPasswordNeeded*** – SET by TPM2_PolicyPassword() to indicate that the *authValue* of the authorized entity will need to be provided when the policy session is used for authorization. The *authValue* is required to be provided as a password. The default value is CLEAR. It will also be CLEAR by TPM2_PolicyAuthValue().
- ***isTrialPolicy*** – SET to indicate that policySession→policyDigest is to be updated even if the assertion is not valid. The session may not be used for authorization.
- ***checkNvWritten*** **–** SET to indicate that the TPMA_NV_WRITTEN attribute of the authorized NV Index must be compared with *nvWrittenState.*
- ***nvWrittenState*** – SET when TPMA_NV_WRITTEN is required to be SET in the NV Index being authorized. This attribute has no meaning when ***checkNvWritten*** is not SET.

19.7.8 Policy Example

In ISO/IEC 11889 (first edition), the basic policy for use of a key was limited to a combination of an authorization value and PCR state. This policy was built in to each key. In ISO/IEC 11889 there is no built-in policy. An ISO/IEC 11889 policy that is the same as the ISO/IEC 11889 (first edition) policy is:

TPM2_PolicyPCR() & TPM2_PolicyAuthValue()

Note This policy could also be written as

TPM2_PolicyAuthValue() & TPM2_PolicyPCR()

This policy would have a different *policyDigest* because the order of evaluation affects the digest.

To associate this policy with a key, evaluate the policy to determine the *policyDigest* that it would generate. Then create the key with this digest as the *authPolicy* and CLEAR the *userWithAuth* attribute. When *userWithAuth* is CLEAR, USER mode actions for the key will require use of the key's *authPolicy*.

19.7.9 Trial Policy

The policy evaluation to determine the value for the *authPolicy* may be done in software that does the same *policyDigest* computation as the TPM. Alternatively, a trial policy session may be used. A trial policy session is created and used in a sequence of policy commands just like a normal policy session. The difference is, in a trial policy, a policy assertion is always assumed to be TRUE and the *policyDigest* updated accordingly. The *policyDigest* value computed in the trial policy can be read from the TPM and used as an object's *authPolicy*. Since the assertions in the trial policy do not need to be valid, the trial session may not be used for authorization.

19.7.10 Modification of Policies

Some policies may be altered directly by changing the *authPolicy* value. Policies associated with Objects and NV Indices may not be directly altered. The reason that these policies may not be altered is that the policy can affect the trust that someone places in the use of that entity.

EXAMPLE 1 Examples of policies that can be altered directly by changing the *authPolicy* value are those associated with the hierarchies.

EXAMPLE 2 A key may only be trusted if it may only be used when the PCR have a specific set of values. If the policy could be changed, then the PCR check could be removed and the key would no longer be trusted. There would be no way for the trusting entity to know if a version of the key exists where the PCR are not checked.

Even though there is no way to directly change a policy, it can be indirectly changed. The command that allows this is TPM2_PolicyAuthorize(). When this command is included in a policy, it allows a designated entity (an "authority") to authorize a *policyDigest* to be included in the policy. This is best described with an example.

EXAMPLE 3 It is common to seal a data value to PCR values so that the data value can only be recovered if the platform has booted in a known way. A problem with this is that if there is a BIOS update, the PCR will change and the sealed data value can no longer be retrieved and some kind of recovery process is necessary.

In the example above, the inability of a policy to accommodate changes to PCR values is called "brittleness". That term suggests that the policy is easy to break (make unusable). This brittleness could be a problem with ISO/IEC 11889 if the policy was completely fixed.

EXAMPLE 4 Figure 21 illustrates the use of TPM2_PolicyAuthorize() to implement a flexible policy. This assertion evaluation checks to see if the current *policyDigest* is authorized by a signing key – that is, did an authorizing entity sign a digest indicating that a specific value of *policyDigest* represents a known set of PCR values. If the *policyDigest* value was signed, then *policyDigest* is replaced by a digest of the Name of the key that was used for authorization and *policyRef* (see 19.7.11). An example of how of this assertion type may be used to avoid PCR brittleness is shown in Figure 21. This shows the example policy in 19.7.8 but with the ability to satisfy the policy with different PCR values.

As shown, a PolicyPCR assertion is followed by PolicyAuthorize(). If there is an authorization signed by KEY for the current *policyDIgest* (in this case, $D_{PCR.A}$), then the result of the PolicyAuthorize() will be D_{KEY}. This is the same output that would be produced if the input to the PolicyAuthorize() were $D_{PCR.B}$ and there was an authorization sighed by KEY for $D_{PCR.B}$. That is, in TPM2_PolicyAuthorize(), if the key authorized the current *policyDigest*, *policyDigest* is replaced by (not extended with) the Name of the key. The *policyDigest* value D_{KEY} no longer reflects the previous value ($D_{PCR.A}$ or $D_{PCR.B}$).

In the case of a BIOS update that changes PCR, the platform OEM could provide a signature for the PCR values created by the new BIOS. Now, if the policy of the sealed data includes a TPM2_PolicyAuthorize() from the OEM, then the BIOS can be updated and no recovery process would be needed to deal with the new PCR values. That is, with either authorized set of PCR, D_{KEY} and D_{final} will be the same, even though $D_{PCR.A}$ and $D_{PCR.B}$ are different.

NOTE 1 Other information is included with the Name of the key when the new *policyDigest* is computed in order to indicate that the Name was included as the result of a TPM2_PolicyAuthorized() operation.

NOTE 2 This example purposefully avoids using terms that would indicate that the signing entity does anything other than indicate that the PCR values are the expected values. In particular, the signing entity does not have to certify that the PCR values are safe. The signing entity could provide other assurances but, in the case of PCR, it is not necessary to warrant anything other than that the PCR values are expected.

NOTE 3 The actual *authPolicy* in the authorized entity would contain (PolicyAuthorize & PolicyAuthValue).

Figure 21 — Use of TPM2_PolicyAuthorize() to Avoid PCR Brittleness

19.7.11 TPM2_PolicySigned(), TPM2_PolicySecret(), and TPM2_PolicyTicket()

The set of assertions discussed in clause 19.7.11 have properties that enable a number of authorization scenarios. Among these are:

- ability to give an authorization that can persist for a specific amount of time (in many protocols, access to a resource (such as, a network) is granted for some time interval), and
- ability to associate an authorization with a policy of the authorizing entity (in many instances, the authorizing entity may use the same key or secret for different purposes).

TPM2_PolicySigned() and TPM2_PolicySecret() convey an authorization by signing a set of parameters that indicate the nature of the authorization. With TPM2_PolicySigned() the signature is with a key value

(symmetric or asymmetric) and with TPM2_PolicySecret() the signature is with an HMAC using an *authValue* in the HMAC key

These commands use a common set of parameters.

- **nonceTPM** – if the caller chooses to limit the authorization to a single policy session, they would include this value in the signed data structure. If this is not part of the signed authorization, then this parameter should be set to the Empty Buffer.

- **cpHashA** – if the caller chooses to limit the authorization to a specific command and command parameters, they would include this value in the signed data structure. Use of this parameter allows the caller to provide an authorization that is similar to the HMAC authorization. That type of authorization is only valid for a specific command and set of command parameters. If this parameter is not part of the signed authorization, then this parameter should be set to the Empty Buffer.

- **policyRef** – in some circumstances, it is desirable to have an authorization convey some information relating to the authorizing entity. The TPM includes this value in the *policyDigest*. If this parameter is not part of the signed authorization, then this parameter should be an Empty Buffer.

  EXAMPLE 1 A fingerprint reader might have a signing key that it uses to verify when it has recognized a fingerprint regardless of whose fingerprint it might be. This type of authorization would be difficult to use if it were not possible to indicate whose fingerprint was scanned. The *policyRef* parameter would allow the fingerprint reader to provide this indication. Because the TPM includes the *policyRef* value in the *policyDigest*, this means that the *policyDigest* would only have the correct value if the fingerprint reader scanned a finger from the correct person.

- **expiration** – this parameter is used to place a time limit on an authorization. If this value is not zero, then *expiration* is the number of seconds before the authorization expires. The expiration time is measured from the time that *nonceTPM* was created for the policy session.

  If *expiration* is a positive number, then the authorization may only be used as long as there is a policy session with the indicated *nonceTPM*. If *expiration* is a negative number, it indicates that the authorization is not specific to a *policySession->nonceTPM* and may be used with other policies, until the authorization expires.

  NOTE 1 A policy session is not flushed when the session is used to authorize a command. However, the *nonceTPM* for the session is changed in order to prevent replay. The policy session is still valid but, since the *nonceTPM* has changed, any TPM2_PolicySigned() and TPM2_PolicySecret() authorizations that were tied to the *nonceTPM* will no longer be valid.

  When a session is started and a specific *nonceTPM* generated, the TPM will record the current TPM time in *policySession→startTime*. When an assertion includes a non-zero *expiration* value, its absolute value is added to *policySession→startTime*. This produces a time that is relative to the TPM Clock. If this computed time is less than the current TPM Clock time, then the authorization has already expired. Otherwise, the authorization is valid. If the computed time is less than the current value of *policySession->timeout*, then *policySession->timeout* is set to the computed value.

  To allow a time-limited authorization to be used in other policies, the TPM generates a ticket. This ticket contains a digest of the command parameters of the authorization but with a *timeout* instead of an *expiration* (the difference being that *expiration* is relative to the creation of a *nonceTPM* and a *timeout* is relative to TPM Clock). TPM2_PolicyTicket() is used to include the authorization in a different policy. As long as the ticket has not expired, its effect on a *policySession→policDigest* will be the same as the command that generated the ticket (TPM2_PolicySigned() or TPM2_PolicySecret()).

The TPM will only produce a ticket if the expiration parameter in TPM2_PolicySigned() or TPM2_PolicySecret() is a negative number. The *timeout* in the ticket will be *policySession→startTime* plus the absolute value of *expiration.*

In order to prevent clock discontinuities from preventing tickets from ever expiring, the timeout value used in the calculation of the ticket structure (an implementation-specific value indicating when the authorization expires) must include enough information that the TPM can distinguish (and discard) tickets when the clock used in the ticket has undergone a discontinuity in its measurement of time. This can be accomplished in several ways.

EXAMPLE 1 Each time the clock is powered on, the TPM might create a random number nonce and that "clock nonce" can be included in the timeout value.

EXAMPLE 2 A monotonic counter can be included with the timeout value which is incremented every time the clock is powered on.

EXAMPLE 3 A system implementation can be created that prevents a ticket from ever being used if the clock in the TPM has not been synchronized with an external trusted clock.

NOTE 2 A "clock nonce" needs to be large enough that a replay is infeasible. That is, a ticket issued with a given nonce cannot be useable after a future power cycle because the nonce values happen to match. In the context of a specific ticket, a nonce collision is not a “birthday problem” as the nonce has to match exactly rather than being one of a group of values that are equivalent.

- The ticket may be used in a policy in place of a TPM2_PolicySecret() or TPM2_PolicySigned() that has the same parameters.

EXAMPLE 4 When authorizing the use of a socket encryption key for an hour, the authorization would contain an expiration of -3600 (an hour of seconds) and the first use of the authorization would be in TPM2_PolicySigned(). That command will return a ticket. For the next hour, TPM2_PolicyTicket() could be used in place of an equivalent TPM2_PolicySigned() assertion.

In summary:

If *nonceTPM* is empty, *expiration* must be zero. Use is not limited to a policy session or time limited.

If *nonceTPM* is included:

If *expiration* is zero, use is not time limited in this policy session

If *expiration* is negative, use is time limited, but a ticket is created, so it can be used in another policy session.

If *expiration* is positive, use is time limited in this session.

19.8 Policy Session Creation

TPM2_StartAuthSession(*sessionType* = TPM_SE_POLICY) is used to start an authorization session. The authorization session may use any of the four options for *tpmKey* and *bind.*

NOTE 1 A policy session does not maintain a binding with a specific object. The bind parameter is used only for session key creation. This allows the context space of the session that is used for the binding value to be dedicated to other policy parameters.

The most typical use of a policy session will be with *tpmKey* and *bind* both set to TPM_RH_NULL. When this option is selected, an HMAC computation might not be performed when the policy session is used and the session *nonce* and *auth* values may be Empty Buffers. See ISO/IEC 11889-3, clause “TPM2_PolicyAuthValue”.

NOTE 2 When the session is created, *nonceCaller* still needs to be provided and its size is required to meet the minimum requirements of the command.

When the authorization session is to be used to authorize a command that has an encrypted command or response parameter, then either *tpmKey* or *bind* should be used in the TPM2_CreateAuthSession() that starts the session so that a secure *sessionKey* is created.

19.9 Use of TPM for *authPolicy* Computation

To use a policy for authorization for an object or NV Index, the creator of an object or NV Index is required to know, at the time of creation of the Object or NV Index, the digest of the policy. The computation of this policy requires duplication of the steps that would be performed by the TPM when it evaluates the policy and updates the accumulated *policyDigest* of the session.

This computation can be done by software but would require that the policy update process for each command be replicated by software. As an alternative, the TPM can be used to perform the computation.

To use the TPM, a policy session is created and various policy commands are sent to the TPM as if the policy were being evaluated in order to authorize an action. TPM2_PolicyGetDigest() may then be used to read the final *policyDigest* from the TPM. That *policyDigest* value may then be used as the *authPolicy* parameter in a new Object/NV Index.

NOTE There is no requirement that the *authPolicy* for each Object or NV Index be unique.

If the policy is complex and uses TPM2_PolicyOR(), it will be necessary to compute multiple *policyDigest* values. The same policy session can be used for all of the computations by using TPM2_PolicyRestart() after the *policyDigest* for a branch is computed. When the last branch is computed, it may be used in a TPM2_PolicyOR that is constructed from the previously computed values.

TPM2_PolicyGetDigest() could also be used to help validate the software that is implementing the digest computation. The value computed by the TPM can be compared to the value computed by the software library to insure that they are the same. If desired, TPM2_PolicyGetDigest() can be called after each policy command.

19.10 Trial Policy Session

If a policy requires a signed (symmetric or asymmetric) authorization for an action, that authorization may not be available at the time that the Object/NV Index is created and, in fact, the authorizing entity might not be willing or able to provide the necessary authorization at the time of creation.

EXAMPLE 1 If the Object is to have a duplication authorization, the duplication authority might not provide the authorization for the duplication when the Object is created. If they did, then the migration policy could be computed; the *policyDigest* of the session read and placed in a new Object, and immediately used for duplication of the Object. The duplication authority might not want to allow the duplication at that time.

The TPM provides a special type of policy session to be used for the purpose of computing the policy without enabling the use of the policy. When a session is created by TPM2_StartAuthSession(*policyType* = TPM_SE_TRIAL), a policy session is created that cannot be used for authorization. Since it cannot be used for authorization, authorizations are not needed in the computation of the policy.

EXAMPLE 2 If TPM2_PolicySigned() is called to update the digest of a trial policy session, the signature is not validated but the *policyDigest* is updated as if a correct signature was provided.

19.11 Dictionary Attack Protection

19.11.1 Introduction

The TPM incorporates mechanisms that provide protection against guessing or exhaustive searches of authorization values stored within the TPM.

The dictionary attack (DA) protection logic is triggered when the rate of authorization failures is too high. If this occurs, the TPM enters Lockout mode in which the TPM will return TPM_RC_LOCKOUT for an operation that requires use of a DA protected *authValue*. Depending on the settings of the configurable parameters described below, the TPM can "self-heal" after a specified amount of time or be programmatically reset using proof of knowledge of an authorization value or satisfaction of a policy.

The *authValue* for an object receives DA protection unless the object's *noDA* attribute is SET. The authValue for an NV Index receives DA protection unless the TPMA_NV_NO_DA attribute of the Index is SET. The *authValue* associated with a permanent entity, other than TPM_RH_LOCKOUT, does not receive DA protection. Sequence objects created by TPM2_HMAC_Start() and TPM2_HashSequenceStart() do not receive DA protection.

NOTE Authorization values associated with permanent entities, other than TPM_RH_LOCKOUT, are expected to be high-entropy values that are managed by a computer or will be well-known values, In either case, they will not need DA protection. While it is safer when *lockoutAuth* is a high-entropy value, it is possible that *lockoutAuth* will be a value chosen to be remembered by a human which will likely have less entropy than other permanent entities. As a consequence, *lockoutAuth* is DA protected even though it is a permanent entity.

The reason for being able to exclude entities from DA protection is that lockout of all TPM use could make the system unstable. The OS may have uses for the TPM that should not be blocked due to authorization problems with keys associated with user-mode applications. The OS is expected to use a well-known or high-entropy *authValue* for any entities that it manages and an *authValue* of neither type needs DA-protection.

An *authValue* may be used for authorization in three ways:

1) a password;
2) the *authValue* parameter in the HMAC computation of equation (18); or
3) the *authValue* parameter in the computation of *sessionKey* for a bound session as shown in equation (19).

All uses of a DA protected *authValue* receive DA protection.

19.11.2 Lockout Mode Configuration Parameters

The TPM uses four, 32-bit, non-volatile state variables to control the initiation and recovery from the DA-lockout mode.

NOTE 1 The "NV" notation indicates that these values need to be held in persistent memory and be updated in NV when they change

a) ***failedTries*** (NV) – This counter is incremented when the TPM returns TPM_RC_AUTH_FAIL. TPM2_Clear() will reset this counter to zero. This counter is also set to zero on a successful invocation of TPM2_DictionaryAttackLockReset(). This counter is decremented by one after *recoveryTime* seconds if:
   1) the TPM does not record an authorization failure of a DA-protected entity,
   2) there is no power interruption, and

3) *failedTries* is not zero.

NOTE 2 If the TPM has a trusted source of time that runs when TPM power is lost, then *failedTries* can be reduced when power is restored. The amount that *failedTries* is decremented would be dependent on the duration of the power loss and the value of *recoveryTime*.

b) ***maxTries*** (NV) – The TPM is in Lockout mode as long as *failedTries* equals this value. When a new owner is installed, *maxTries* is set to its default value as specified in the relevant platform-specific specification.

c) ***recoveryTime*** (NV) – This value indicates, in seconds, the rate at which *failedTries* is decremented. This can be set to a large value (2^{32} - 1) which essentially inhibits automatic exit from Lockout mode. When a new owner is installed, this value is set to its default value as specified in the relevant platform-specific specification.

d) ***lockoutRecovery*** (NV) – This value indicates the delay in seconds between attempts to use *lockoutAuth*. The time delay only applies after an authorization failure using *lockoutAuth*. A value of zero indicates that a system reboot (TPM2_Startup(TPM_SU_CLEAR)) is required between lockout attempts.

The parameters *maxTries*, *recoveryTime*, and *lockoutRecovery* are set with TPM2_DictionaryAttackParameters(). This command requires Lockout Authorization.

19.11.3 Lockout Mode

The TPM is in Lockout mode while failedTries is equal to *maxTries*. While in Lockout mode, any use of a DA-protected authValue will return TPM_RC_LOCKOUT.

NOTE 1 An exception is that TPM2_DictionaryAttackLockReset() can execute even though *lockoutAuth* is DA protected.

NOTE 2 If there is an authorization failure that does not increment *failedTries*, the TPM returns TPM_RC_BAD_AUTH

An authorization failure may occur with a password or an HMAC. For a policy authorization, the policy is validated before the HMAC is computed. If the policy fails, the TPM returns TPM_RC_POLICY to indicate that dictionary attack protection was not involved.

NOTE 3 A policy authorization does not always have an associated HMAC.

19.11.4 Recovering from Lockout Mode

The TPM can recover from Lockout mode in three ways.

1) TPM2_DictionaryAttackLockReset() sets *failedTries* to zero. This command requires Lockout Authorization. The TPM does not have to be in Lockout mode in order to use this command.

2) The TPM decrements *failedTries* by one if no TPM resets are recorded during *recoveryTime*.

NOTE 1 If the TPM is in Lockout mode, then the TPM will always leave Lockout mode when *failedTries* decrements because *failedTries* will no longer be equal to *maxTries*.

NOTE 2 The failure count is not decremented below zero.

3) *failedTries* is set to zero if the owner changes.

Configuration and programmatic recovery of the dictionary attack logic requires proof of knowledge of Lockout Authorization. When the TPM owner is changed by changing the SPS, *lockoutAuth* is set to the EmptyAuth and *lockoutPolicy* is set to the Empty Buffer

TPM2_DictionaryAttackLockReset() allows external software to reset the dictionary attack protection logic by providing Lockout Authorization. This command can be used when the TPM is in Lockout mode.

19.11.5 Authorization Failures Involving *lockoutAuth*

When *lockoutAuth* is used in an authorization and that authorization fails, the TPM enters a lockout state intended to provide special protection for the *lockoutAuth* value. An authorization failure associated with *lockoutAuth* causes the TPM to enter this special lockout state regardless of the setting of *failedTries* and *maxTries*.

When in this special lockout state, the TPM will not allow use of *lockoutAuth*. The TPM will exit this state when TPM2_DictionaryAttackLockReset() is used with a successful lockoutPolicy or when after the TPM is powered for a configurable time period (*lockoutRecovery*). If *lockoutRecovery* is set to zero, then the TPM will not exit this state until the next TPM Reset or until lockoutPolicy is used.

NOTE The design depends upon the trusted computing base to filter commands to the TPM such as TPM2_DictionaryAttackLockReset(). This prevents a rogue application from completing a denial of service attack on the TPM by intentionally sending the command with a bad *lockoutAuth*.

19.11.6 Non-orderly Shutdown

A TPM may be implemented such that the command execution unit does not always have access to NV memory (see 37.7.2). For such an implementation, it may not be possible to increment the NV copy of *failedTries* when the authorization failure occurs. When the failure occurs, the TPM will return TPM_RC_AUTH_FAIL and, until the NV version of *failedTries* is updated, the TPM will be in lockout.

It is possible that the TPM will be reset when a write to the NV version of *failedTries* is pending. If the TPM did not handle this special case, then an attacker could try an authorization for a DA protected object when NV writes are not possible. When the TPM restarted, the failed attempt would not be recorded and the attacker could try again.

To prevent this type of attack, at TPM2_Startup(), the TPM checks if it is starting after an orderly shutdown. If not, and *failedTries* is not already equal to *maxTries*, then the TPM will increment *failedTries* by one.

NOTE This check and increment of *failedTries* might not be necessary if it is impractical for an attacker to prevent update of the NV version of *failedTries*.

19.11.7 Justification for Lockout Due to Session Binding

When a bound session is created, the caller does not have to prove knowledge of the *authValue* of the bind object. The *authValue* is used in the creation of the *sessionKey* and if the caller does not know the *authValue*, they will not be able to compute the correct *sessionKey* and use the authorization session.

A bound authorization session may be used to authorize actions on another object. If that object does not have DA protection, then an attacker could use binding to circumvent DA protection on the bind object.

The attack is as follows:

a) An attacker creates an object (D) that has no DA protection and authValue known to the attacker.

b) An attacker guesses a possible *authValue* for a DA protected object (object A).

c) The attacker uses object A as the *bind* object in TPM2_StartAuthSession() to create a session (S).

d) The attacker uses session S to authorize an action on object D.

e) If the authorization fails, the attacker goes to step b) and tries a new value.

By retaining the DA state of object A in the session state, the attack is prevented. When the session is used for authorization, the authorization failure counter (*failedTries*) is incremented if either the entity being authorized is subject to DA protection or if the session is bound to an entity that has DA protection.

NOTE If a session is bound to a permanent entity other than TPM_RH_LOCKOUT, then the session is not bound to an entity that has DA protection.

19.11.8 Sample Configurations for Lockout Parameters

19.11.8.1 Introduction

Two common configurations are anticipated: one for enterprise-managed TPMs, and one for home users.

NOTE It is anticipated that the operating system will layer additional anti-hammering protection atop that provided by the TPM so that it is unlikely that one OS user will be able to interfere with the actions of another user or the trusted computing base (TCB).

19.11.8.2 Enterprise Use

In this use, it is expected that the TPM owner will set the *lockoutAuth* to a high-entropy value that is held in a database and set the *lockoutRecovery* to a small, non-zero value. The enterprise will use this value to recover the TPM when suitable non-automated validation procedures have been performed.

EXAMPLE 1 An example of a small, non-zero lockoutRecovery value is one.

The enterprise would likely set *maxTries* to a relatively low value.

EXAMPLE 2 An example of a relative low value for maxTries is 10.

For a server or data center, the *recoveryTime* would be set to a large value implying manual recovery. For laptops, a setting of a few hours would provide adequate protection for PINs.

EXAMPLE 3 An example large value for recoveryTime is 2^{32}.

19.11.8.3 Home or Unmanaged Use

In this application, the *lockoutAuth* may be set to a random, high-entropy value that is then erased so that programmatic lockout recovery is not possible. *maxTries* and *recoveryTime* can be set to balance security and convenience.

NOTE If this configuration is used, the only way to execute TPM2_Clear() to change the owner is to use Platform Authorization.

20 Audit Session

20.1 Introduction

An audit session allows for the auditing of a selected sequence of commands so that evidence may be provided that the commands were executed.

Any HMAC authorization session may be designated for auditing but only one session may be used for audit in each command. A session is designated for auditing by setting the *audit* attribute in the session.

When a session is first used as an audit session, the TPM will initialize the audit hash for the audit session. The initialization value is a Zero Digest with the number of octets determined by the hash algorithm of the session.

If the session was bound when created (see 19.6.10 and 19.6.12), the bind value is lost and any further use of the session for authorization will require that the *authValue* be used in the HMAC.

Since the first use of an audit session may cause the size of the session context to change, the command may fail due to insufficient memory. TPM-management software may save other session contexts and retry the command.

NOTE 1 The TPM needs to have sufficient memory to allow three sessions to be simultaneously loaded, one of which may be an audit session.

For all commands using a session tagged as audit (including the initial use), if the command completes successfully, the *cpHash* and the *rpHash* are Extended to the audit session digest. When a command fails, the audit session digest is not changed and, as is normal in the case of a command failure, the sessions are not included in the response and session nonces are not updated.

The equation for updating the audit session digest is:

$$auditDigest_{new} := \mathbf{H}_{auditAlg}\left(auditDigest_{old} \,||\, cpHash \,||\, rpHash\right) \tag{40}$$

The hash algorithm is the algorithm designated in TPM2_StartAuthSession().

Unless a command description indicates that no sessions are allowed, an audit session may be used with any command. A command may have only one audit session.

An audit session uses the same session format as other HMAC-based sessions. The method of computing the HMAC differs in that, if the audit session is not associated with any object handle, no *authValue* is available for use in the authorization HMAC. All HMAC computations for an audit session will set *authValue* to an Empty Buffer.

NOTE 2 If the *sessionKey* is also an Empty Buffer, then no HMAC computation is performed and the *hmac* parameter of the session structure will be an Empty Buffer.

If an unbound and unsalted session is used as the basis for the audit session, then there is no assurance from the audit session that the commands being audited are actually associated with a TPM. On the other hand, a bound session allows association with a known *authValue* in a TPM, which can provide assurance that the commands being audited are actually associated with a specific TPM. However, if others know the *authValue*, then the unsalted audit session may have the same association issue as the unbound session. A salted session can be associated with a key that is known to be TPM-resident so the audit based on a salted session can be reliably associated with a specific TPM.

20.2 Exclusive Audit Sessions

In a response, the *auditExclusive* attribute of an audit session will indicate if the TPM has executed any commands that were not audited by the session. If there was another user of the TPM, the *auditExclusive* attribute will be CLEAR, and if not the attribute will be SET.

The TPM keeps track of the current exclusive session. A session becomes the current exclusive audit session when it is first used as an audit session. It may also become the current exclusive audit session if the *auditReset* attribute of the session is set. The session is no longer the current exclusive audit session if it is flushed (TPM2_FlushContext()) or if an auditable command is executed that does not use the current exclusive audit session.

A command that is not allowed to have any sessions will not change the current exclusive audit session. Those commands include the context management commands (TPM2_ContextSave(), TPM2_ContextLoad(), and TPM2_Flush()), TPM2_Startup(), and TPM2_ReadClock().

NOTE 1 It is the responsibility of the TCG Software Stack (TSS) or other controlling software to preserve the integrity of the exclusive audit session. As the purpose of the exclusive audit session is to show that no other commands were executed during the session, the expectation is that the controlling software would limit access to the TPM to prevent any other uses of the TPM.

To indicate the start of an exclusive interval, the caller may SET *auditReset* in the first command of the exclusive sequence. In the response, the *auditExclusive* attribute of the session will be SET and the session is exclusive.

NOTE 2 *AuditReset* can only be SET if audit is also SET.

The first time that a session is used for audit, the session becomes exclusive, regardless of the setting of *auditReset*.

In a response, if an audit session is exclusive, the *auditExclusive* attribute will be SET

20.3 Command Gating Based on Exclusivity

If the *auditExclusive* attribute of an audit session is SET, then the TPM will return TPM_RC_EXCLUSIVE if the session is not exclusive.

NOTE As with other error returns, no change is made to the state of the session and it remains active.

20.4 Audit Session Reporting

The audit status of an audit session can be determined with TPM2_GetSessionAuditDigest(). This command returns a signed data structure that includes the audit session digest.

Because the audit digest is signed before the audit digest is updated, the *cpHash* and *rpHash* for a TPM2_GetSessionAuditDigest() is not included in the audit digest of the signed data structure. Possession of the audit digest is proof that the command executed. However, the *cpHash* and *rpHash* of TPM2_GetSessionAuditDigest() will be included in subsequent audits if the audit session remains active.

TPM2_GetSessionAuditDigest() requires that the indicated session be an audit session and will return TPM_RC_TYPE if it is not. The TPM does not change internal state unless the command actions complete successfully. This means that a session cannot become an audit session unless the command in which it is designated as an audit session completes successfully. From this we can conclude that a session cannot be designated as being an audit session in a TPM2_GetSessionAuditDigest() in which the same session is the audited session.

20.5 Audit Establishment Failures

If a command is the first use of a session as an audit session, and the command fails, then the state of a session as an audit sessions will not change. This means that, if a session was not an audit session before the command was executed, it will not be an audit session after the command fails. If a session was an audit session before the command was executed, it will be an audit session after the command fails.

If a command fails, then the exclusive status of sessions does not change. A session that was exclusive before the command failure is exclusive after the command failure.

21 Session-based encryption

21.1 Introduction

Several commands have parameters that may need to be encrypted going to or from the TPM. Session-based encryption may be used to ensure confidentiality of these parameters.

EXAMPLE An example is the authorization data that is passed to the TPM when an object is created or when the authorization value is changed.

Not all commands support parameter encryption. If session-based encryption is allowed, only the first parameter in the parameter area of a request or response may be encrypted. That parameter must have an explicit size field. Only the data portion of the parameter is encrypted. The two encryption methods (XOR and CFB) do not require that the data be padded for encryption so the encrypted data size and the plain-text data size is the same.

If the symmetric algorithm is TPM_ALG_NULL and encryption or decryption is specified, the TPM returns TPM_RC_SYMMETRIC.

Any first parameter may be encrypted as long as the parameter has a size field.

Session-based encryption uses the algorithm parameters established when the session is started and values that are derived from the session-specific *sessionKey*. The encryption values are created in a way that is dependent on both the session type and the session encryption parameters.

If *sessionAttributes.decrypt* is SET in a session in a command, and the first parameter of the command is a sized buffer, then that parameter is encrypted using the encryption parameters of the session. If *sessionAttributes.encrypt* is SET in a session of a command, and the first parameter of the response is a sized buffer, then the TPM will encrypt that parameter using the encryption parameters of the session. The *encrypt* attribute may only be SET in one session that is used in a command and the *decrypt* attribute may only be SET in one session per command. The attributes may be SET in different sessions or in the same session.

Parameters in commands are encrypted before any *cpHash* is computed. Parameters in responses are encrypted before any *rpHash* is computed.

The parameter data buffer is protected with either XOR obfuscation or CFB mode encryption. The size field of the parameter is not protected.

When a command/response with an encrypted parameter is received, the *cpHash*/*rpHash* is computed as required for the sessions before the parameter is decrypted.

NOTE The caller can obfuscate the true size of an authorization value by adding octets of zero to the end. The extra octets of zero will have no impact on the authorization computations and can be discarded by the TPM.

The two methods of session-based encryption used in ISO/IEC 11889 are, by themselves, malleable. That is, an attacker can make a controlled change (bit reversal) in the encrypted data that will result in an identical change in the decrypted data. This kind of attack is mitigated as the encryption uses a key that is associated with an HMAC authorization session.

21.2 XOR Parameter Obfuscation

For session-based obfuscation using **XOR**(), the operation is:

$$\mathbf{XOR}(parameter, hashAlg, sessionValue, nonceNewer, nonceOlder) \quad (41)$$

where

parameter	a variable sized buffer containing the parameter to be obfuscated
hashAlg	the hash algorithm associated with the session
sessionValue	the session-specific HMAC key
nonceNewer	for commands, this will be nonceCaller and for responses it will be nonceTPM
nonceOlder	for commands, this will be *nonceTPM* and for responses it will be *nonceCaller*

NOTE 1 Depending on the usage, *sessionValue* can be either the *sessionKey* or the *sessionKey* with a concatenated *authValue*.

NOTE 2 The **XOR**() function is defined in 11.4.6.4.

NOTE 3 The obfuscated size of parameter is the same as the size of the underlying parameter. That is, if a TPMB_CREATE is obfuscated, the size of the obfuscated data is the same as the size of the data.

21.3 CFB Mode Parameter Encryption

When session-based encryption uses a symmetric block cipher, an encryption key and IV will be generated from:

$$\mathbf{KDFa}\ (hashAlg, sessionKey, \text{“CFB”}, nonceNewer, nonceOlder, bits) \quad (42)$$

where

hashAlg	the hash algorithm associated with the session
sessionKey	the *sessionKey* value associated with the session
“CFB”	label to differentiate use of **KDFa**() (see clause 5.4 for the definition of this label.)
nonceNewer	*nonceCaller* for a command and *nonceTPM* for a response
nonceOlder	*nonceTPM* for a command and *nonceCaller* for a response
bits	the number of bits required for the symmetric key plus an IV

NOTE 1 The IV size is equal to the block size of the cipher.

The most significant octets of the returned value are used as the encryption key and the remaining octets are used as the IV. The number of octets used for the encryption key and for the IV is dependent on the algorithm parameters of the session.

EXAMPLE For AES, the block size is 16 octets regardless of the key size. If the key size were 256 bits (32 octets), then, in the call to **KDFa** (), *bits* would be set to 48*8. The most significant 32 octets of the returned value would be used as the key for the encryption and the next 16 octets would be used for the IV.

NOTE 2 If the key size is not an even multiple of 8 bits, the first N octets of the returned value will contain the key and the remaining octets the IV. N is the smallest integer such that (N * 8) ≥ the key size.

The data portion of the parameter is then encrypted using the symmetric key and the symmetric block cipher algorithm associated with the session.

22 Protected Storage

22.1 Introduction

When a Protected Object is in the TPM, it is in a Shielded Location because the only access to the context of the object is with a Protected Capability (a TPM command). The size of TPM memory may be limited and if the only storage for Protected Objects were the TPM Shielded Locations, the TPM's usefulness would be reduced. The effective memory of the TPM is expanded by using cryptographic methods for Protected Objects when they are not in Shielded Locations.

22.2 Object Protections

The cryptographic protections for a Protected Object include encryption to prevent disclosure of the confidential contents, and an integrity check to allow detection of modifications to the externally stored Protected Object. The integrity check detects modifications to either the confidential or the non-confidential portions of the Protected Object.

The integrity value is computed over the encrypted data. If the integrity check fails, then symmetric decryption will not occur. Since the integrity value contains the digest of the associated public area (its Name), the confidential contents of the Protected Object will not be decrypted if they are not properly paired with a public area.

22.3 Protection Values

The protection of a sensitive area uses two keys. These values are created from a secret value associated with the parent. One of the keys is used as an HMAC key and the second is a symmetric encryption key.

A seed value is used in the generation of the symmetric encryption key and the HMAC key. The source of the seed is dependent on the situation. If the protections are for an object in a hierarchy, the seed is the *seedValue* in the parent's sensitive area. If the protections are for a duplication blob, the seed is derived from a shared secret that is protected using asymmetric methods of the new parent. The algorithm-specific annexes to this part of 11889 contain the formulations for deriving the seed when asymmetric protections are used.

To produce the symmetric key, the seed value and object Name are used in **KDFa**() as shown in equation (43). This method is used when a symmetric key is generated for the protection of sensitive areas attached to a hierarchy or sensitive data in a duplication blob (see 23.3).

NOTE 1 This method is also used to generate the symmetric key used for the protection of credential values (see clause 24.4).

To produce the HMAC key, the seed is used in **KDFa**() as shown in equation (45). This method is used when an HMAC is used to protect the integrity of a sensitive area attached to a hierarchy or for sensitive data in a duplication blob.

NOTE 2 This method is also used to generate the HMAC key for credential values (see clause 24.4).

When performing symmetric encryption, an IV of zero is used unless the same symmetric key is used multiple times. The same symmetric key is used each time that the sensitive area of a child changes due to TPM2_ObjectChangeAuth(). For encryption of a child, a random IV is generated by the TPM each time it performs the encryption.

22.4 Symmetric Encryption

A symmetric key is used to encrypt the sensitive area of an object that was created by TPM2_Create() or imported by TPM2_Import(). It is also used when re-encrypting a sensitive area when the authorization value is changed (TPM2_ObjectChangeAuth()). The symmetric key is derived from a seed value contained in the parent's sensitive area and the Name of the protected object.

The block cipher used for encrypting the object's sensitive area is the symmetric cipher of the parent.

The symmetric key for the encryption is computed by:

$$symKey := \mathbf{KDFa}\ (pNameAlg, seedValue, \text{“STORAGE”}, name, \text{NULL}, bits) \quad (43)$$

where

pNameAlg	*nameAlg* of the object's parent
seedValue	symmetric seed value in the sensitive area of the object's parent (see 27.6.4)
“STORAGE”	a value used to differentiate the uses of the KDF (see clause 5.4 for the definition of this value)
name	Name of the object being encrypted / decrypted
bits	number of bits required for the symmetric key

When a *symKey* is being used to protect the sensitive area of a child object, the TPM will create a random IV value (*symIv*) that is the size of an encryption block of the symmetric algorithm. This *symIV* is included in the private area and in the HMAC computation of the sensitive area. A *symIV* of zero is used when encrypting the sensitive area for duplication or a credential to be used in TPM2_ActivateCredential().

The *symKey* and *symIv* are used to encrypt the sensitive data.

$$encSensitive := \mathbf{CFB}_{pSymAlg}\ (symKey, symIv, sensitive) \quad (44)$$

where

$\mathbf{CFB}_{pSymAlg}$	symmetric encryption in CFB mode using the symmetric algorithm of the parent
symKey	symmetric key from (43)
symIv	IV from RNG or 0
sensitive	a TPM2B_SENSITIVE containing the sensitive area structure being protected

NOTE The *size* and *buffer* fields of *sensitive* are encrypted.

After the data is encrypted, the TPM2B_IV containing the random *symIv* is placed in front of the encrypted data in preparation for the integrity computation. If the *symIV* was zero, then no value is added to the encrypted data.

22.5 Integrity

The HMAC key (*HMACkey*) for the integrity is computed by:

$$HMACkey := \mathbf{KDFa}\ (pNameAlg, seedValue, \text{“INTEGRITY”}, \text{NULL}, \text{NULL}, bits) \quad (45)$$

where

pNameAlg	the *nameAlg* of the object's parent
seedValue	the symmetric seed value in the sensitive area of the object's parent (see 27.6.4)
“INTEGRITY”	a value used to differentiate the uses of the KDF (see clause 5.4 for the definition of this value)
bits	the number of bits in the digest produced by *pNameAlg*

HMACkey is then used in the integrity computation.

An HMAC is performed over the *symIV* and the *encSensitive* produced in (44).

NOTE 1 This is called an *outerHMAC* because it is the same HMAC process that is used when an object is duplicated. The duplication can produce an inner and an outer HMAC.

$$outerHMAC := \mathbf{HMAC}_{pNameAlg}\ (HMACkey, symIv\ ||\ encSensitive\ ||\ name.buffer) \quad (46)$$

where

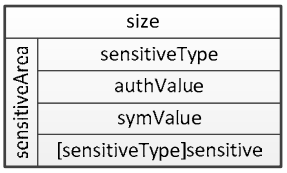
$\mathbf{HMAC}_{pNameAlg}$	the HMAC function using *nameAlg* of the object's parent
HMACkey	a value derived from the parent symmetric protection value according to equation (45)
symIv	a marshaled TPM2B_IV containing the symmetric IV value used in (44). Both the size and buffer fields are included in the HMAC
encSensitive	encrypted TPM2B_SENSITIVE produced in (44); after encryption, the size and buffer fields are not separable
name.buffer	the Name of the object being protected (does not include a size field)

The integrity value is placed before the symmetric IV.

NOTE 2 Placement of the integrity value at the beginning of the sensitive area in preparation simplifies the process of finding the integrity value when the protected data contains variable-sized elements.

NOTE 3 Inclusion of the Name ensures that the sensitive area is associated with the correct public area.

1) Marshal the sensitive area into a TPM2B_SENSITIVE



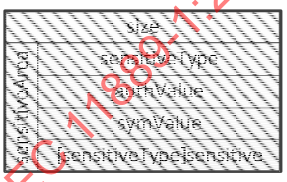
2) Create a symmetric key and IV for encryption:

symKey ≔ **KDFa** (*pNameAlg, seed,* "STORAGE"*, name, NULL , bits*)

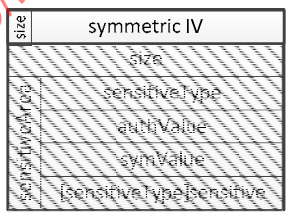
symIV ≔ bits from the RNG

3) Create *encSensitive* by encrypting the TPM2B_SENSITIVE

encSensitive ≔ **CFB**$_{pSymAlg}$ *(symKey, symIv, sensitive)*



4) Add the symmetric IV to (a TPM2B_IV) the encrypted block

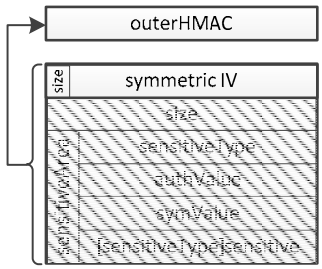


5) Compute the HMAC key

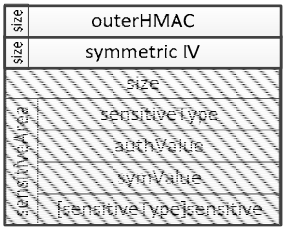
HMACkey ≔ **KDFa** (*pNameAlg, seed,* "INTEGRITY", NULL, NULL, *bits*)

6) Compute the HMAC over the symmetric IV (the full TPM2B_IV), the *encSensitive* from step 3, and the Name of the object being protected.

outerHMAC ≔ **HMAC**$_{pNameAlg}$ *(HMACkey, symIv || encSensitive || name.buffer*)



7) Marshal the *outerHMAC* into a TPM2B_DIGEST and append the symmetric IV and encrypted sensitive.



NOTE Regarding steps 6 and 7, an overall size field will be added to make the resulting TPM2B_PRIVATE structure.

Figure 22 — Creating a Private Structure

23 Protected Storage Hierarchy

23.1 Introduction

The TPM supports the creation of hierarchies of Protected Locations. A hierarchy is constructed with Storage Keys as the connectors to which other types of objects (keys, data, and other connectors) may be attached.

The hierarchical relationship of objects allows segregation of objects based on the system-operating environment (established by PCR or authorizations) as well as simplifying the management of groups of related objects.

23.2 Hierarchical Relationship between Objects

A hierarchy is rooted in a secret seed key, kept in the TPM. To create a hierarchy of keys, the seed key is used to generate an asymmetric key that uses a specific set of algorithms. If this key is a restricted decryption key, then it is a Storage Key to which other objects may be attached.

A Storage Key provides protection for the sensitive area in another object when that object is stored outside of the TPM. Protection is provided by symmetric encryption of the sensitive area. The key used for encryption either is derived from a seed value in the Storage Key or is a duplication symmetric key. A seed-derived symmetric key is applied when an object is connected to a hierarchy and a duplication symmetric key is used when an object has been encrypted to be "attached" to an additional parent.

The objects in a hierarchy have a parent-child relationship. A Storage Key that is protecting other objects is a parent and the objects that it is protecting are its children. The ancestors of an object are the parent keys that connect the object to a TPM Primary Seed. Descendants of a key are all the objects that have the key as an ancestor. Unless it is intended to be used as a parent, a child object may be of any type. Only a Storage Key may be a parent key.

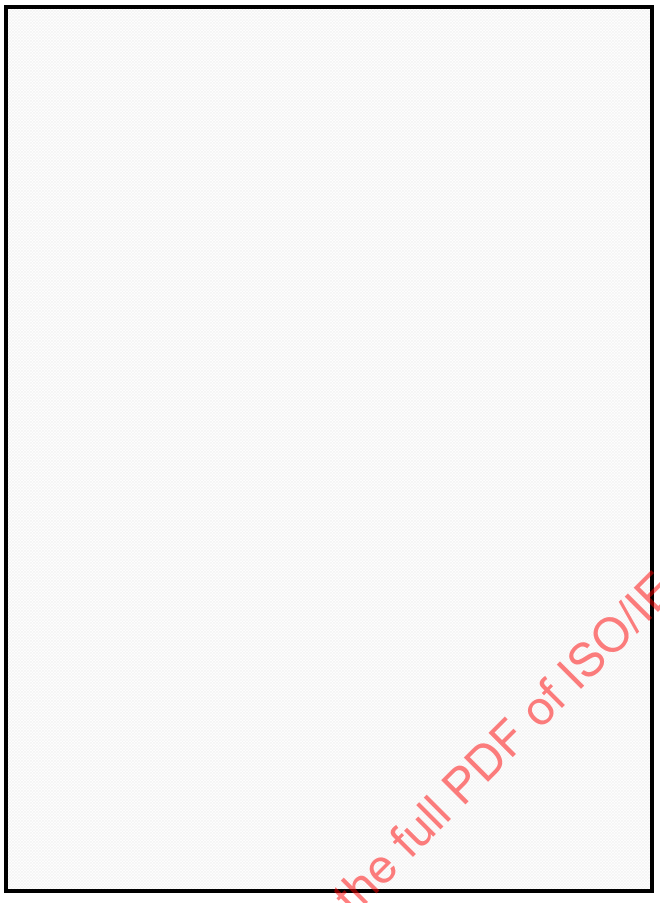


Figure 23 — Symmetric Protection of Hierarchy

While it is possible to create a hierarchy of Storage Keys using symmetric key objects instead of asymmetric key objects, it would defeat one of the primary purposes of the Storage Key, which is to provide an attachment point for import of other keys. A storage key is a symmetric protection key with an asymmetric identity (an asymmetric public key). The asymmetric identity is used to identify unambiguously the point in a hierarchy where a key can be imported. Therefore, an entity doing key duplication can know that the duplicated key can only be imported to another hierarchy at a specific location, identified by the public key.

23.3 Duplication

23.3.1 Definition

Duplication is the process of allowing an object to be a child of additional parent keys. The new parent (NP) may be in a hierarchy of the same TPM or of a different TPM.

The creator of an object controls the duplication process by selecting the duplication policy for the object.

Authorization for duplication requires a policy session. The policy sequence is required to have a command that causes the *commandCode* value of the policy context to be set to TPM_CC_Duplicate. This enables the DUP role of the policy, which is a requirement for duplication.

Duplication occurs on a loaded object and produces a new, sensitive structure that is encrypted using the methods of the NP. This new sensitive structure may not be used until TPM2_Import() has been executed to convert the object from "external" to "internal" protections.

NOTE External protections use both asymmetric and symmetric cryptography, whereas the internal protections only use symmetric cryptography.

23.3.2 Protections

23.3.2.1 Introduction

In TPM2_Duplicate(), the caller may specify that the object should be protected with an inner, symmetric encryption. That is, the sensitive area is symmetrically encrypted before it is asymmetrically encrypted using the methods of the NP. If a symmetric inner wrapper is desired, the caller may provide a key or allow the TPM to generate the key.

If the *encryptedDuplication* attribute is SET in the object being duplicated, then it is required that the object have an inner wrapper and that the new parent not be TPM_RH_NULL. For such an object, the TPM will return an error (TPM_RC_SYMMETRIC) if the *symmetricAlg* parameter in TPM2_Duplicate() is TPM_ALG_NULL and TPM_RC_HIERARARCHY if the *newParentHandle* parameter is TPM_RH_NULL.

Creation of a duplicate object uses two encryption phases. The first is used to apply an inner wrapper and the second is to encrypt using the algorithms of the NP.

The *encryptedDuplication* attribute of all objects in a duplication group are required to have the same setting. When an object is created with the *fixedParent* attribute CLEAR, then the *encryptedDuplication* attribute may be SET or CLEAR if the *fixedTPM* attribute is SET in the parent. If the *fixedTPM* attribute of a parent is not SET, then the *encryptedDuplication* attribute is required to be the same in all descendant objects of that parent.

23.3.2.2 Inner Duplication Wrapper

For the first phase, the TPM computes an integrity hash over the sensitive data. This hash includes the Name of the public area associated with this object.

$$innerIntegrity := \mathbf{H}_{nameAlg}\left(sensitive \,||\, name\right) \tag{47}$$

where

$\mathbf{H}_{nameAlg}$	hash function using the *nameAlg* of the object
$sensitive$	a TPM2B_SENSITIVE
$name$	the Name of the object being protected

A TPM2B_DIGEST containing the integrity digest value is prepended to the sensitive area and the buffer (integrity plus sensitive) is encrypted using CFB.

$$encSensitive := \mathbf{CFB}_{pSymAlg}(symKey, 0, innerIntegrity \,||\, sensitive) \quad (48)$$

where

$\mathbf{CFB}_{symAlg}$	symmetric encryption in CFB mode using the algorithm specified in the command
symKey	*encryptionKeyIn* parameter in TPM2_Duplicate() or a value from the RNG if no key is provided
innerIntegrity	value from (47)
sensitive	the sensitive value used in (47)

If no inner wrapper is specified, no integrity value is computed and no encryption occurs in this first phase and

$$encSensitive := sensitive \quad (49)$$

23.3.2.3 Outer Duplication Wrapper

In the second phase, the *encSensitive* produced by phase 1 is encrypted and integrity checked using processes similar to those defined in clause 22. However, the seed from which the protection keys are derived is protected by the asymmetric algorithm of the NP. The method of generating *seed* is determined by the asymmetric algorithm of the NP. The different methods are specified in annexes to this part of ISO/IEC 11889. The seed is selected prior to integrity generation for *encSensitive* or encryption of *encSensitive*.

NOTE For an RSA new parent, seed is not allowed to be larger than the size of the digest produced by the *nameAlg* of the object. When the TPM creates the seed, it will be exactly the size of the *nameAlg* of the new parent.

Given a value for *seed*, a symmetric encryption key (*symKey*) is created by:

$$symKey := \mathbf{KDFa}(npNameAlg, seed, \text{“STORAGE”}, Name, \text{NULL}, bits) \quad (50)$$

where

npNameAlg	the *nameAlg* of the new parent
seed	the symmetric seed value
“STORAGE”	a value used to differentiate the uses of the KDF (see clause 5.4 for the definition of this value)
Name	the Name of the object being encrypted or decrypted
bits	the number of bits required for the symmetric key

The *symKey* is used to encrypt the *encSensitive*.

$$dupSensitive := \mathbf{CFB}_{npSymAlg}(symKey, 0, encSensitive) \quad (51)$$

where

$CFB_{npSymAlg}$	symmetric encryption in CFB mode using the algorithm of the parent
symKey	symmetric key from (50)
encSensitive	value from either (48) or (49)

Next, an HMAC key is generated from seed:

$$HMACkey := \mathbf{KDFa}(npNameAlg, seed, \text{“INTEGRITY”}, \text{NULL}, \text{NULL}, bits) \quad (52)$$

where

npNameAlg	the *nameAlg* of the object's new parent
seed	the symmetric seed value used in (50)
“INTEGRITY”	a value used to differentiate the uses of the KDF (see clause 5.4 for the definition of this value)
bits	the number of bits in the digest produced by *npNameAlg*

An HMAC is then generated over the *dupSensitive* data. The Name of the associated public area is included in the HMAC computation to ensure that the sensitive area will only be decrypted when the proper public and private areas are used in TPM2_Import().

$$outerHMAC := \mathbf{HMAC}_{npNameAlg}(HMACkey, dupSensitive \,||\, Name) \quad (53)$$

where

$\mathbf{HMAC}_{npNameAlg}$	the HMAC function using nameAlg of the new parent
HMACkey	a value derived from the parent symmetric protection value according to equation (52)
dupSensitive	symmetrically encrypted sensitive area produced in (51)
Name	the Name of the object being duplicated

To complete the duplication process, the TPM2B_PUBLIC and TPM2B_ENCRYPTED_SECRET produced by TPM2_Duplicate() are used in TPM2_Import() at the TPM containing the public and private portions of the NP. If the private area is doubly encrypted, then the symmetric key used for the inner wrapper is also given to the TPM.

TPM2_Import() will recover the symmetric key according to the algorithm of the NP. The TPM2B_PRIVATE is decrypted. If an inner wrapper is present, the TPM2B_PRIVATE is decrypted using the supplied symmetric key. After symmetric decryption, the integrity value is checked.

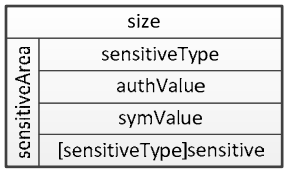
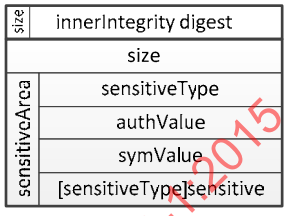
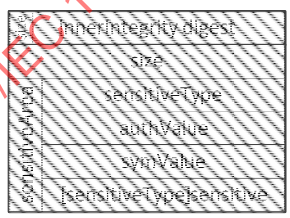
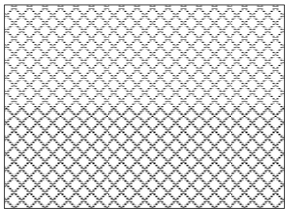
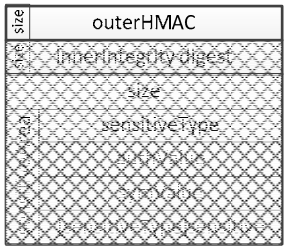
1) Marshal the *sensitive* area into a TPM2B_SENSITIVE	
2) Compute an *innerIntegrity* value $innerIntegrity := \mathbf{H}_{nameAlg}\ (sensitive \mid\mid name)$	
3) Set the encryption key (*symKey*) to *encryptionKeyIn* or a random value produced by the TPM.	
4) Create *encSensitive* by encrypting the *innerIntegrity* value and the TPM2B_SENSITIVE $encSensitive := \mathbf{CFB}_{symAlg}\ (symKey,\ 0,\ innerIntegrity \mid\mid sensitive)$	
5) Using methods of the asymmetric new parent, create a *seed* value	
6) Create a symmetric key (*symKey*): $symKey := \mathbf{KDFa}\ (npNameAlg,\ seed,\ \text{“STORAGE”},\ Name,\ \text{NULL},\ bits)$	
7) Create *dupSensitive* by encrypting *encSensitive* $dupSensitive := \mathbf{CFB}_{npSymAlg}\ (symKey,\ 0,\ encSensitive)$	
8) Compute the HMAC key from the *seed* created in step 5) $HMACkey := \mathbf{KDFa}\ (npNameAlg,\ seed,\ \text{“INTEGRITY”},\ \text{NULL},\ \text{NULL},\ bits)$	
9) Compute the HMAC over *dupSensitive* and include the object *Name* $outerHMAC := \mathbf{HMAC}_{npNameAlg}\ (HMACkey,\ dupSensitive \mid\mid Name)$	
NOTE 1 Regarding step 1, if no inner or outer wrapper is applied to the object, the structure in step 1 is returned as the *duplicate* parameter in the response for TPM2_Duplciate().	
NOTE 2 Regarding step 9, an overall size field will be added to make the resulting TPM2B_PRIVATE structure.	

Figure 24 — Duplication Process with Inner and Outer Wrapper

Figure 25 illustrates the processing of a duplication blob when no inner wrapper is used in the sensitive area.

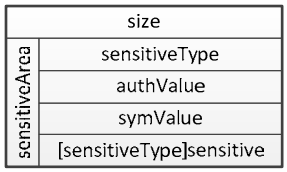
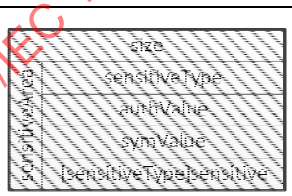
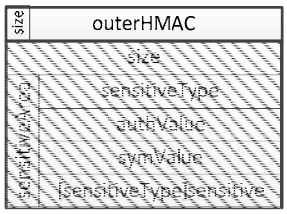
1) Marshal the *sensitive* area into a TPM2B_SENSITIVE	
2) Since there is no inner wrapper set $encSensitive \coloneqq sensitive$	
3) Using methods of the asymmetric new parent, create a *seed* value	
4) Create a symmetric key for encryption: $symKey \coloneqq \mathbf{KDFa}\ (npNameAlg, seed,$ "STORAGE", $name,$ NULL , $bits)$	
5) Create *dupSensitive* by encrypting *encSensitive* $dupSensitive \coloneqq \mathbf{CFB}_{npSymAlg}\ (symKey,$ 0, $sensitive)$	
6) Compute the HMAC key from the *seed* created in step 3 $HMACkey \coloneqq \mathbf{KDFa}\ (npNameAlg, seed,$ "INTEGRITY", NULL, NULL, $bits)$	
7) Compute the HMAC over the *dupSensitive* $outerHMAC \coloneqq \mathbf{HMAC}_{npNameAlg}\ (HMACkey, dupSensitive \mid\mid name)$	
NOTE Regarding step 7, an overall size field will be added to make the resulting TPM2B_PRIVATE structure.	

Figure 25 — Duplication Process with Outer Wrapper and No Inner Wrapper

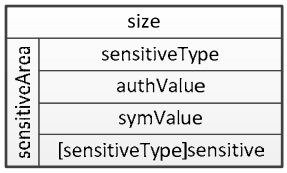
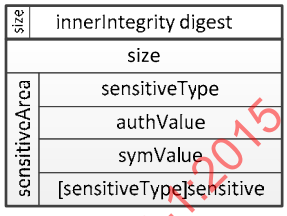
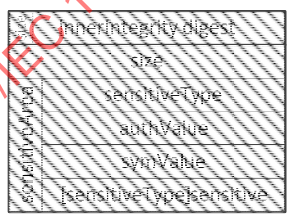
1) Marshal the *sensitive* area into a TPM2B_SENSITIVE	
2) Compute an *innerIntegrity* value $innerIntegrity := \mathbf{H}_{nameAlg}(sensitive \mid\mid Name)$	
3) Set the encryption key (*symKey*) to *encryptionKeyIn* or a random value produced by the TPM.	
4) Create *encSensitive* by encrypting the *innerIntegrity* value and the TPM2B_SENSITIVE $encSensitive := \mathbf{CFB}_{symAlg}(symKey, 0, innerIntegrity \mid\mid sensitive)$	
NOTE Regarding step 4, an overall size field will be added to make the resulting TPM2B_PRIVATE structure.	

Figure 26 — Duplication Process with Inner Wrapper and TPM_RH_NULL as NP

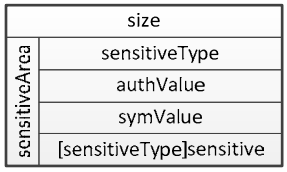
1) Marshal the *sensitive* area into a TPM2B_SENSITIVE	
NOTE An overall size field will be added to make the resulting TPM2B_PRIVATE structure. This will result in a TPM2B_SENSITIVE being the only contents of the TPM2B_PRIVATE buffer.	

Figure 27 — Duplication Process with no Inner Wrapper and TPM_RH_NULL as NP

23.4 Duplication Group

The duplication process allows an object or segment of a hierarchy to be duplicated for use in another hierarchy. This ability facilitates key distribution and backup. A duplication group is a group of objects in a hierarchy under a duplication root. The entire duplication group may be moved to another hierarchy by duplicating the duplication root.

When an object is created, its duplication attribute (*fixedParent*) is selected. If *fixedParent* is CLEAR, then the object may be operated on by TPM2_Duplicate(). This command allows the sensitive area of an object to be encrypted under a new parent so that it may be used in a different TPM hierarchy. The act of duplicating a Storage Key has the side effect of duplicating all of its descendants regardless of the setting

of their *fixedParent* attribute. That is, if a parent key is usable in a different hierarchy, then all the descendants of the parent key are also usable in the different hierarchy as well.

NOTE 1 No modification of the encryption of a child object is required to make it usable on another hierarchy. This is because the Storage Key that is duplicated contains the information used to protect its children. Duplication of the protection information has the effect of duplicating the objects protected by that information.

NOTE 2 If a particular Storage Key is usable in multiple hierarchies, then descendants of that Storage Key are usable in the same hierarchies regardless of when they are created. That is, if they are created after the duplication of the parent, they are still usable in multiple hierarchies.

If an object has *fixedParent* CLEAR, it is the root of a duplication group. If the object is not a Storage Key, then the group will have a single member. For a Storage Key, the duplication group consists of all objects that are duplicated as a direct consequence of duplicating the group root.

Objects that have *fixedParent* SET cannot be directly duplicated (that is, they may not be the referenced *objectHandle* in TPM2_Duplicate()). However, they can be implicitly duplicated if an ancestor has *fixedParent* CLEAR and that ancestor is duplicated.

Objects that have *fixedParent* SET and have no ancestors with *fixedParent CLEAR* are the only objects that are not part of a duplication group. These objects are identified by having their *fixedTPM* attribute SET. All objects that are in a duplication group have their *fixedTPM* attribute CLEAR.

An object may be a member of more than one duplication group. This would occur if more than one of its ancestor Storage Keys has *fixedParent CLEAR* or if an object and one of its ancestors has *fixedParent* CLEAR.

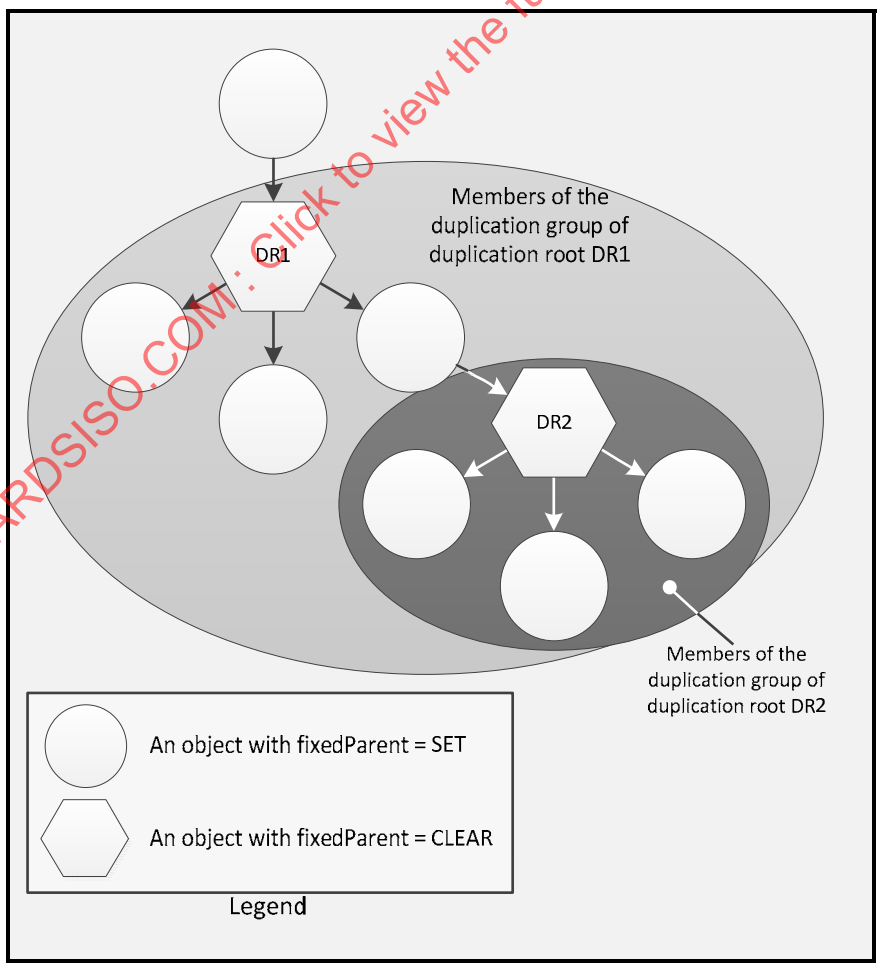


Figure 28 — Duplication Groups

23.5 Protection Group

The algorithms (asymmetric, symmetric, and hash) and key sizes used to protect child keys are consistent within a protection group. The protection group is all of the descendants of a duplication root not including other duplication roots or their descendants.

By requiring all of the non-duplicable Storage Keys to use the same algorithm, it is easier to determine the security properties of a hierarchy. If an object's *fixedTPM* attribute is SET, then all of the ancestor keys of that object use the same set of algorithms. If an object's *fixedTPM* is not SET, then the protections are determined by the duplication authority for each of the duplication roots in the object's hierarchy.

The reason that the protections are determined by the duplication authority and not by the algorithms of the key is that a duplication authority can attach a duplication root to a software-generated new parent. Inspecting the hierarchy in which an object exists does not guarantee the protections of the object unless the object's *fixedTPM* is SET.

Change of the algorithm set at a duplication root is illustrated in Figure 29.

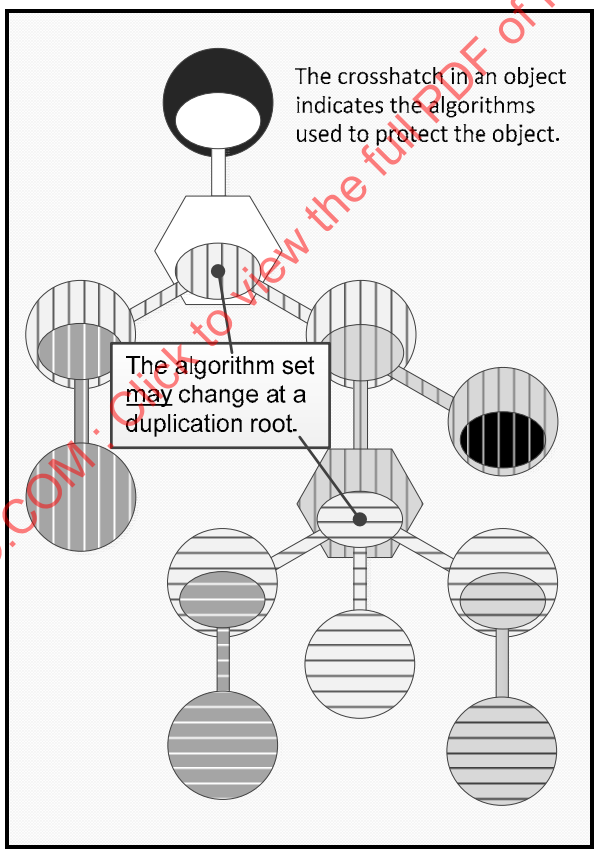


Figure 29 — Protection Groups

23.6 Summary of Hierarchy Attributes

The hierarchy attributes of an object indicate how the object is connected to the hierarchy. They indicate if the object could be extant in other hierarchies and if the object may be duplicated directly by TPM2_Duplicate().

Table 23 lists the possible combinations of an object's hierarchy attributes and the interpretation of each combination.

Table 23 — Mapping of Hierarchy Attributes

fixedParent	***fixedTPM***	**Description**
0	0	This combination represents a duplication root.
0	1	This combination is not allowed.
1	0	This combination indicates an object that is permanently in the protection group of its parent. It cannot be the *objectHandle* reference in TPM2_Duplicate().
1	1	This combination indicates an object that was created on a specific TPM and no duplicate of the object is possible.

23.7 Primary Seed Hierarchies

A Primary Object is an object that is derived from a Primary Seed value. A Primary Object is unique in that the sensitive area of the object is protected using a symmetric key that is derived from a Primary Seed and the Name of the Primary Object. In contrast, other objects are encrypted using the symmetric key of their parent.

Once created, a Primary Object may be managed like any other loadable object, including being context-saved/restored.

NOTE Since a Primary Object is symmetrically encrypted in both the loadable form and in the context-saved form, the performance advantages of using context save and context load on a Primary Object could actually be negative. It could take longer to save and restore the object than to simply reload it.

A Primary Object may have *fixedParent* SET or CLEAR. If a Primary Object has *fixedParent* SET, then *fixedTPM* is required to be SET.

23.8 Hierarchy Attributes Settings Matrix

Table 24 shows the combinations of hierarchy settings allowed for an object. In the table, the check marks ("✓") indicate that the combination is allowed.

Table 24 — Allowed Hierarchy Settings

***fixedTPM* setting in**		**Object's *fixedParent***		
parent	**object**	**CLEAR**	**SET**	**Comments**
CLEAR	CLEAR	✓	✓	if the parent's *fixedTPM* attribute is CLEAR, the child's

CLEAR	SET			*fixedTPM* is required to be CLEAR
SET	SET		✓	if the parent of an object has *fixedTPM* SET, then *fixedParent* and *fixedTPM* must have the same setting in the child[(1)(2)]
SET	CLEAR	✓		

NOTE 1 For purposes of this table, the parent of a Primary Object is considered to have a *fixedTPM* attribute that is always SET.

NOTE 2 If the parent has fixedTPM SET, then a child could be duplicable (*fixedParent* == CLEAR) or not (*fixedParent* == SET). If the child is not duplicable, then it needs to have the same setting of *fixedTPM* as its parent.

The consistency of the hierarchy settings is checked in object templates (TPM2_Create() and TPM2_CreatePrimary()) and in public areas for loaded objects (TPM2_Load()) or duplicated objects (TPM2_Import()).

Consistency of an object's hierarchy settings are not checked on an object loaded with TPM2_LoadExternal(). This is because the external object is not part of a hierarchy and its settings are not significant. Also, TPM2_LoadExternal() is used to load the public area of a key that may exist on another TPM. This would occur when using the TPM to check a signature or when loading the public area of a new parent during duplication. Those keys have hierarchy properties that are not relevant to the TPM that is using those keys so the hierarchy properties are not checked.

24 Credential Protection

24.1 Introduction

The TPM supports a privacy preserving protocol for distributing credentials for keys on a TPM. The process allows a credential provider to assign a credential to a TPM object, such that the credential provider cannot prove that the object is resident on a particular TPM but the credential is not available unless the object is resident on a device that the credential provider believes is an authentic TPM.

24.2 Protocol

The initiator of the credential process will provide, to a credential provider, the public area of a TPM object for which a credential is desired along with the credentials for a TPM key (usually an EK). The credential provider will inspect the credentials of the “EK” and the properties indicated in the public area to determine if the object should receive a credential. If so, the credential provider will issue a credential for the public area.

The credential provider may require that the credential only be useable if the public area is a valid object on the same TPM as the “EK.” To ensure this, the credential provider encrypts the credential and then "wraps" the credential encryption key with the public key of the “EK.”

NOTE “EK” is used to indicate that an EK is typically used for this process but any storage key may be used. It is up to the credential provider to decide what is acceptable for an “EK.”

The encrypted credential and the wrapped encryption key are then delivered to the initiator. The initiator can decrypt the credential by loading the “EK” and the object onto the TPM and asking the TPM to return the credential encryption key. The TPM will decrypt the credential encryption key using the private “EK” and validate that the credentialed object (public and private) is loaded on the TPM. If so, the TPM has validated that the properties of the object match the properties required by the credential provider and the TPM will return the credential encryption key.

This process preserves privacy by allowing TPM objects to have credentials from the credential provider that are not tied to a specific TPM. If the object is a signing key, that key may be used to sign attestations, and the credential can assert that the signing key is on a valid TPM without disclosing the exact TPM.

A second property of this protocol is that it prevents the credential provider from proving anything about the object for which it provided the credential. The credential provider could have produced the credential with no information from the TPM as the TPM did not need to provide a proof-of-possession of any private key in order for the credential provider to create the credential. The credential provider can know that the credential for the object could not be in use unless the object was on the same TPM as the “EK” but the credential provider cannot prove it.

24.3 Protection of Credential

The credential blob (which typically contains the information used to decrypt the actual credential) from the credential provider contains a value that is returned by the TPM if the TPM2_ActivateCredential() is successful. The value may be anything that the credential provider wants to place in the credential blob but is expected to be values that are used to decrypt a blob containing the actual credentials of an object.

The credential provider protects the credential value (CV) with an integrity HMAC and encryption in much the same way as a credential blob. The difference is, when *seed* is generated, the label is "IDENTITY" instead of "DUPLICATE". (See clause 5.4 for the definition of these labels.)

24.4 Symmetric Encrypt

A seed is derived from values that are protected by the asymmetric algorithm of the "EK". The methods of generating the seed are determined by the asymmetric algorithm of the "EK" and are specified in an annex to this part of ISO/IEC 11889. In the process of creating *seed*, the label is required to be "INTEGRITY."

NOTE If a duplication blob is given to the TPM, its HMAC key will be wrong and the HMAC check will fail.

Given a value for *seed*, a key is created by:

$$symKey := \mathbf{KDFa}\ (ekNameAlg,\ seed,\ \text{"STORAGE"},\ name,\ \text{NULL}\ ,\ bits) \tag{54}$$

where

ekNameAlg	the *nameAlg* of the key serving as the "EK"
seed	the symmetric seed value produced using methods specific to the type of asymmetric algorithms of the "EK"
"STORAGE"	a value used to differentiate the uses of the KDF (see clause 5.4 for the definition of this label)
name	the Name of the object associated with the credential
bits	the number of bits required for the symmetric key

The *symKey* is used to encrypt the CV. The IV is set to 0.

$$encIdentity := \mathbf{CFB}_{ekSymAlg}\ (symKey,\ 0,\ CV) \tag{55}$$

where

$CFB_{ekSymAlg}$	symmetric encryption in CFB mode using the symmetric algorithm of the key serving as "EK"
symKey	symmetric key from (54)
CV	the credential value (a TPM2B_DIGEST)

24.5 HMAC

A final HMAC operation is applied to the *encIdentity* value. This is to ensure that the TPM can properly associate the credential with a loaded object and to prevent misuse of or tampering with the CV.

The HMAC key ($HMACkey$) for the integrity is computed by:

$$HMACkey := \mathbf{KDFa}\ (ekNameAlg, seed, \text{“INTEGRITY”}, \text{NULL}, \text{NULL}, bits) \quad (56)$$

where

$ekNameAlg$	the *nameAlg* of the target “EK”
$seed$	the symmetric seed value used in (54); produced using methods specific to the type of asymmetric algorithms of the “EK”
“INTEGRITY”	a value used to differentiate the uses of the KDF (see clause 5.4 for the definition of this label)
$bits$	the number of bits in the digest produced by *ekNameAlg*

NOTE Even though the same value for label is used for each integrity HMAC, $seed$ is created in a manner that is unique to the application. Since $seed$ is unique to the application, the HMAC is unique to the application.

$HMACkey$ is then used in the integrity computation.

$$identityHMAC := \mathbf{HMAC}_{ekNameAlg}\ (HMACkey, encIdentity \,||\, Name) \quad (57)$$

where

$\mathbf{HMAC}_{ekNameAlg}$	the HMAC function using *nameAlg* of the “EK”
$HMACkey$	a value derived from the “EK” symmetric protection value according to equation (56).
$encIdentity$	symmetrically encrypted sensitive area produced in (55)
$Name$	the Name of the object being protected

The integrity structure is constructed by placing the *identityHMAC* (size and hash) in the buffer ahead of the *encIdentity*.

24.6 Summary of Protection Process

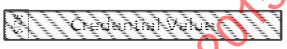
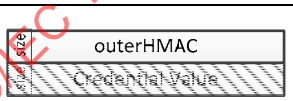
1)	Marshal the CV into a TPM2B_DIGEST	
2)	Using methods of the asymmetric “EK”, create a *seed* value	
3)	Create a symmetric key for encryption: $symKey := \mathbf{KDFa}\ (ekNameAlg, seed, \text{“STORAGE”}, Name, \text{NULL}, bits)$	
4)	Create *encIdentity* by encrypting the CV $encIdentity := \mathbf{CFB}_{ekSymAlg}\ (symKey, 0, CV)$	
5)	Compute the HMAC key $HMACkey := \mathbf{KDFa}\ (ekNameAlg, seed, \text{“INTEGRITY”}, \text{NULL}, \text{NULL}, bits)$	
6)	Compute the HMAC over the *encIdentity* from step 4 $outerHMAC := \mathbf{HMAC}_{ekNameAlg}\ (HMACkey, encIdentity \mid\mid Name)$	

Figure 30 — Creating a Identity Structure

25 Object Attributes

25.1 Base Attributes

25.1.1 Introduction

Three attributes are used to determine how the TPM may use an object. These attributes are designated as *restricted*, *sign*, and *decrypt*. The Boolean combinations of these attributes are used to express the full range of behaviors for objects.

25.1.2 *Restricted* Attribute

When the *restricted* attribute of a key is SET, the key may only operate on other objects that follow strict, but simple, format rules. A restricted key is not usable in all commands that use a key of that type. The restrictions on each type of key are explained in the clauses describing the *sign* and *decrypt* attributes.

The *restricted* attribute has no meaning when applied to an object that has both *sign* and *decrypt* CLEAR and *restricted* is required to be CLEAR for those objects.

25.1.3 *Sign* Attribute

This attribute may apply either to symmetric or asymmetric keys. A signing key uses its sensitive area key to sign data. The signature is returned by the TPM.

An asymmetric signing key may perform signing according to the key family and the signing method selected. An external entity may use the public portion of an asymmetric key to validate that the information was signed by someone with knowledge of the private portion of the key.

EXAMPLE RSA or ECC are examples of key families.

A symmetric key that can sign is used for performing an HMAC computation. This signature can be checked by another entity that knows the HMAC secret key in order to validate the source of the information.

NOTE No signing algorithm for a symmetric block cipher is currently defined by the TCG. If one is defined, then the limitation of this paragraph would change.

A restricted signing key may only sign a digest that has been produced by the TPM. The digest may be over externally supplied data or an internally generated structure. An internally generated structure that is to be signed will have the characteristic TPM_GENERATED_VALUE as the first octets in the structure to be hashed and signed. When the TPM generates a digest over externally provided data, the TPM validates that the first octets of the data are not equal to the TPM_GENERATED_VALUE. When a digest is signed by a restricted signing key, there is no ambiguity about whether or not the signed data was generated by the TPM.

A restricted signing key is occasionally referred to in ISO/IEC 11889 as an Attesting or Attestation Key.

25.1.4 *Decrypt* Attribute

An asymmetric decryption key uses the private asymmetric key in its sensitive area to decrypt data blobs that have been encrypted using the public portion of the key. A symmetric decryption key uses the key in its sensitive area to decrypt data that has been encrypted by that key.

A key that has both *decrypt* and *restricted* attributes SET only accepts data that has a specific structure. The encrypted data block must have as its first element an integrity value for the remainder of the structure. This integrity value is an HMAC of the encrypted data. This format allows the TPM to prevent misuse of the restricted decryption keys that are the basis of the protected storage hierarchy.

If the sensitive data is a child object, the symmetric and HMAC keys are derived from the symmetric seed value in the sensitive area of the parent. If the sensitive data is a duplication or certification blob, the symmetric and HMAC keys are derived from a single use seed. That seed is then protected using the asymmetric public key of the intended recipient of the protected blob.

When loading a protected blob, the TPM validates the integrity value before decrypting the data. The only way that the integrity value can be correct is if it were created by some entity with access to the unencrypted sensitive data.

NOTE The specific threat scenario that is addressed by this scheme is that an attacker will use a protected blob in a command that is not appropriate for that blob.

EXAMPLE An attacker could load the sensitive portion of an asymmetric key and attempt to access the sensitive area using TPM2_Unseal(). The TPM will unseal data, but not a key. The attacker could attempt to modify the public area of the key in order to trick the TPM into thinking that the protected blob contains a sealed data rather than a private key. The integrity value prevents these deceptions.

A restricted decryption key is often referred to in ISO/IEC 11889 as a Storage Key.

25.1.5 Uses

Table 25 shows the combinations of an object's functional attributes and describes the resulting properties.

Table 25 — Mapping of Functional Attributes

sign	*decrypt*	*restricted*	Description
0	0	0	A data blob. Can be accessed using TPM2_Unseal().
0	0	1	Not allowed. The TPM will not load or create an object with this setting.
0	1	0	A key that can be used in any operation that requires a decryption key, except that the key may not be a storage key.
0	1	1	Indicates that only the default schemes and modes of the key may be used In ISO/IEC 11889, an asymmetric key with these properties is referred to as a Storage Key. The TPM only allows this key to be used on objects that have a specific structure. Use includes create, load, unseal, and activate identity.
1	0	0	Indicates a key that may be used with any signing operation including quote, certify, and sign. The recipient of signatures generated by this key should be aware that quotes and certifications can be

			forged so the trust would not be in the key but in the entity that knows the key authorization value. If use with object type TPM_ALG_KEYEDHASH, then the key may be used for HMAC operations.
1	0	1	This combination indicates a key that can sign any digest that the TPM has created. The TPM only signs a digest over externally provided data that did not have as its first octets TPM_GENERATED_VALUE. This key can be used reliably for quoting, certifying, and signing. No signing command is prohibited for this type of key. Only the default schemes and modes of the object may be used.
1	1	0	A general-purpose key that can be used with any command that requires a key as long as the command is compatible with the key algorithm. However, this key may not be a Storage Key (the parent of other keys).
1	1	1	This type of key is currently not supported because use of a signing key as a storage node could prevent an application from being able to use the TPM in a way that is compliant with FIPS.

Table 26 shows the correspondence between the ISO/IEC 11889 (first edition) method of identifying key properties and the method in ISO/IEC 11889.

Table 26 — ISO/IEC 11889 (first edition) Correspondence

ISO/IEC 11889 (first edition) Name	*sign*	*decrypt*	*restricted*	Comments
TPM_KEY_SIGNING	1	0	0	In ISO/IEC 11889 (first edition), keys had restricted schemes. In ISO/IEC 11889, the scheme is defined in the command.
TPM_KEY_STORAGE	0	1	1	The functional properties are nearly the same as in ISO/IEC 11889 (first edition). This key could only be used to protect and unprotect items in a Protection hierarchy.
TPM_KEY_IDENTITY	1	0	1	In ISO/IEC 11889 (first edition), an Identity key was highly constrained. In ISO/IEC 11889, the restricted signing key can sign (within the limits defined in clause 25.1.3) a digest produced by the TPM.
TPM_KEY_AUTHCHANGE	-	-	-	This is not used in ISO/IEC 11889 and its use was deprecated in ISO/IEC 11889 (first edition). The functionality is provided by session encryption.
TPM_KEY_BIND	0	1	0	Functionality is roughly equivalent between the ISO/IEC 11889 (first edition) type and the unrestricted decryption key. In ISO/IEC 11889 TPM2_RSA_Decrypt() is similar to TPM_Unbind() from ISO/IEC 11889 (first edition).
TPM_KEY_LEGACY	1	1	0	Use of these keys is only constrained by the key family properties.
TPM_KEY_MIGRATE	0	1	1	A Storage Key may be the object of a re-wrap if the new parent is allowed within the policy for the object. The policy for duplication of the object is always visible in the public area.
Sealed Data	0	0	0	a blob containing user defined data

EXAMPLE 1 An Identity key in ISO/IEC 11889 (first edition) was highly constrained because it could not sign a structure that was not produced by the TPM.

EXAMPLE 2 Because in ISO/IEC 11889 the restricted signing key can sign (within the limits defined in clause 25.1.3) a digest produced by the TPM, this means it can use an Attestation Key to sign a PKCS#10 certificate request.

EXAMPLE 3 An example of how TPM_KEY_LEGACY is only constrained by the key family properties is that an

ECC key will not perform TPM2_RSA_Decrypt().

25.2 Other Attributes

25.2.1 fixedTPM and fixedParent

These attributes are specified in detail in clause 23.

25.2.2 stClear

This attribute indicates an object that will need to be reloaded after any Startup(CLEAR). Objects may be loaded into the TPM and their context saved by the TPM resource manager. Normally, these saved contexts may be reloaded at any time before the next TPM Reset. However, if this attribute is SET, then the saved context associated with the object will be invalidated on each TPM Restart as well as on TPM Reset.

An object that has this attribute SET may not be made persistent.

25.2.3 sensitiveDataOrigin

When a symmetric object (TPM_ALG_KEYEDHASH or TPM_ALG_SYM) is created, the caller may provide the secret data or have the TPM generate it. If the TPM is to be the source of the data, then the caller will SET this attribute otherwise, this attribute will be CLEAR and the caller-provided data will be used.

This attribute may not be SET in an asymmetric object. The public part of an asymmetric object is determined by its private key. If the caller has control over both the public and sensitive areas, then the TPM cannot ensure that the key is statistically unique. This is not an issue unless the object also has *fixedTPM* SET. One of the assumptions of a *fixedTPM* object is that it is statistically unique. This would not be the case for an asymmetric key if the caller provided the data. To avoid the possibility of creating a *fixedTPM* object on multiple TPMs, an asymmetric key is required to have its private key generated by the TPM or the object may be imported. If it is imported, *fixedTPM* will not be SET.

25.2.4 userWithAuth

This attribute indicates that the object's *authValue* may be used to provide the USER role authorizations for the object. If this attribute is CLEAR then USER role authorizations may only be provided by satisfying the object's *authPolicy* in a policy session. A policy session may be used for USER mode authorizations when this attribute is SET or CLEAR.

25.2.5 adminWithPolicy

This attribute indicates that authorization for an action requiring the ADMIN role requires that the *authPolicy* of the object be satisfied. If this attribute is CLEAR, then the *authValue* may be used in an HMAC session to perform operations that require ADMIN role.

As with USER role authorizations, any ADMIN role action may be authorized with a policy session that satisfies the *authPolicy*.

The primary reason for having a set of operations that require ADMIN role is to allow each of the actions to be individually controlled. When a policy is used for an ADMIN role action, the policy must contain a

command that sets the *commandCode* for the policy to the specific command. This allows each ADMIN role action to be individually enabled and controlled without having to group them.

25.2.6 noDA

If this attribute is SET in an object, then authorization failures of the object will not invoke dictionary attack protections. In addition, actions on an object with this attribute SET are not subject to lockout. This attribute is used to ensure that access to objects used by the OS is not blocked due to actions by users. An OS would be expected either to use objects with well-known values or to use high-entropy authorization values. In neither case is dictionary attack protection required.

25.2.7 encryptedDuplication

If this attribute is CLEAR, then an object may be duplicated with *newParentHandle* set to TPM_RH_NULL, which means that there is no outer wrapper for the object. If the caller does not specify an inner wrapper, then the object may be exported with this sensitive area in the clear.

While the entity that controls duplication is expected to be trusted to maintain the confidentiality of the sensitive area of a key during duplication, conformance to some standards requires that the sensitive area be encrypted when it leaves the TPM and reliance on the caller is not adequate for those standards. This attribute provides a method of producing objects that conform to those standards.

NOTE It is understood that the duplication authority can still arrange to have access to the sensitive area of the key by creating a software key and having the TPM duplicate to that key.

26 Object Structure Elements

26.1 Introduction

The TPM is intended to provide a means of creating a Storage hierarchy to protect data and keys (keys generated by the TPM or some other entity). Each of these objects (keys and data) has two components. The first is a public area that contains the attributes of the object and a public identity. The second is the sensitive area that contains the elements of the object that require TPM protections. These elements include an authorization value, one or more secret key values, and, in some cases, sealed data values.

The structure definitions for both the public and sensitive areas of an object define how the information is to be arranged when it crosses the TPM interface. The organization of these structures as they exist within the TPM is at the discretion of the TPM vendor. However, the actions of commands in ISO/IEC 11889 are defined in terms of these presumptive structures and any implementation will need to produce equivalent results.

26.2 Public Area

The public area contains the information for identification of an object and its properties. The fields of the public area are listed and specified in Table 27.

Table 27 — Public Area Parameters

Parameter	Description
type	This identifies the type of the object. An algorithm ID is used as the type identifier because the structures contain parameters that are specific to the types of operations that can be performed on or with the object.
nameAlg	This is a second algorithm ID that identifies the hash algorithm used for computing the Name of the object. It is also the default hash algorithm for operations on or with this object that use a hash
objectAttributes	This contains the set of attributes of the object. These attributes are in five classes: 1) usage (*sign*, *encrypt*, *restricted*); 2) authorization (*userWithAuth*, *adminWithPolicy*, *noDA*); 3) duplication (*fixedParent*, *fixedTPM*); 4) creation (*sensitiveDataOrigin*); and 5) persistence (*stClear*).
authPolicy	This will contain the authorization policy for the object if one is defined.
[type]parameters	The parameters of an object are dependent on the object *type*. For symmetric key object, the parameters would indicate the size of the key and the default encryption mode. For an asymmetric object (RSA or ECC), the parameters would indicate the key size, signing scheme, and symmetric encryption methods associated with the key.
[type]unique	The unique value of an object is also dependent on the object *type*. For an asymmetric object, this will be the public key. For a symmetric object, this will be a value computed by hashing values in the sensitive area.

EXAMPLE 1 Regarding the type parameter, an RSA type would contain an RSA key pair that could be used for operations defined for RSA

EXAMPLE 2 Regarding the type parameter, an AES type would be used for symmetric encryption or decryption.

NOTE An object that is intended to be duplicated needs have an *authPolicy* enabling the duplication.

26.3 Sensitive Area

The sensitive area is related to the public area and contains the data that are required to be encrypted when not in a Shielded Location on the TPM. It contains the authorization value and the item-specific information. If an object is a Storage Key, it contains the symmetric key that is used to encrypt its child object.

EXAMPLE An example of item-specific information is the private or secret portion of a key.

The structure of the sensitive area is shown in Table 28.

Table 28 — Sensitive Area Parameters

Parameter	Description
sensitiveType	This identifies the type of the object for this sensitive area. This value and the type parameter of the public area are the same.
authValue	This is the authorization value for the object. It is a octet array of zero or more octets. The authorization value for an object may not have more octets than the digest produced by the object's *nameAlg*.
seedValue	For an asymmetric key, this value is required for Storage Keys and is the seed used to generate the protection values for the child objects of the Key. This is optional for other asymmetric keys, and is not used if present. For all other object types, this is an obfuscation value. It is hashed with the *sensitive* field to produce the *unique* value in the public area. Including this value in the computation obfuscates *unique* so that the *sensitive* value cannot be determined from the *unique* field.
[sensitiveType]sensitive	The contents of this parameter are dependent on *sensitiveType.* For an asymmetric key, this will contain the private key. For an HMAC or symmetric key, this will be the key. For a data object, this will be the sensitive data.

Each sensitive area created by the TPM contains some TPM-created data that makes each sensitive area statistically unique. This will be either an asymmetric key or a large random number. The unique values in the sensitive area are cryptographically linked to values in the public area in a way that makes each public area statistically unique. The fact that a sensitive area is statistically unique and cryptographically linked to a public area ensures that a TPM can detect any attempt to substitute the sensitive area associated with a public area. Such a substitution would allow subversion of secrets-based policy authorization. If an attacker could use an arbitrary sensitive area with a public area with a known Name, the attacker could perform TPM2_PolicySecret() and cause the *policyDigest* to be updated with the chosen Name even though the attacker does not know the authorization value of the correct sensitive area. Cryptographic linking of the sensitive area to the public area ensures that this type of attack is not practical.

26.4 Private Area

When a sensitive area is not in a Shielded Location on a TPM, it is integrity-protected and symmetrically encrypted. There is more than one format for a protected sensitive area but the loadable (TPM2_Load()) form of the protected sensitive area is called a "private" area.

NOTE 1 Another format is a saved context.

The process of converting a sensitive area to a private area requires that the sensitive area be marshaled to its canonical form. This marshaled structure is then encrypted using a key derived from the parent's symmetric seed. An HMAC is performed over the data with the Name of the associated sensitive area

include in the HMAC. The combination of the HMAC and the encrypted sensitive area is a key's private area.

NOTE 2 Similar protections are used when an object is context saved or duplicated.

26.5 Qualified Name

The Qualified Name (QN) of an object is the digest of all of the Names of all of the ancestor keys back to the Primary Seed at the root of the hierarchy. The QN of an object includes the Name of the object. The QN uses the Name hash of the current object to compute the QN for the object.

EXAMPLE 1 Assuming that key A is the parent of object B, then the Qualified Name of B (QN_B) is:

$$QN_B := \mathbf{H}_B\ (QN_A\ ||\ NAME_B)$$

The QN is not a digest of all of the entities loaded into the TPM. It is a digest of all of the entities in a chain.

EXAMPLE 2 Assume two entities with public areas of A and B and different Name hash algorithms (H_A and H_B). Also assume that they share the same parent P with a QN of QN_P. The QN for A is $QN_A := H_A(QN_P\ ||\ H_A(A))$ and the QN for B is $QN_B := H_B(QN_P\ ||\ H_B(B))$.

The primary purpose of the Qualified Name is to supplement the environmental information relating to object creation and object use. The environment of an object includes its hierarchy. The hierarchy starts at a Primary Seed and includes all ancestor keys for the object. The Qualified Name of an object is included in its creation data. The Qualified Name permits validation that a list of ancestor Names is correct. The Qualified Name of an object is included in its certification to indicate that the key is being used in a different environment (ancestry) than the one in which it was created.

EXAMPLE 3 Because the Qualified Name permits validation that a list of ancestor Names is correct, it is then possible to determine if all ancestor keys use sufficient cryptographic strength.

Both the Name and Qualified Name for a Primary Seed are the handle of the Primary Seed. If the parent handle is TPM_RH_NULL, Name and QN are also TPM_RH_NULL. This makes the QN of a Primary Object or Temporary Object equal to:

$$QN := \mathbf{H}_{nameAlg}(A\ hierarchy\ handle\ ||\ Primary\ Object\ Name) \quad (58)$$

NOTE The Name and QN of the parent of an object are included in the creation data of that object.

26.6 Sensitive Area Encryption

When a sensitive area is in a loadable format (a private area), the symmetric encryption key is derived from the secret seed of the parent.

When a sensitive area has been encrypted for duplication, the sensitive area is symmetrically encrypted with a key that is protected using asymmetric methods associated with the new parent. Before a duplicated object may be loaded, it must be "imported" (TPM2_Import()) and encrypted using the symmetric key derived from the secret seed of the new parent.

NOTE Clause 30.3 describes the protections that are applied to a sensitive area when it is part of a saved context.

All symmetric encryption of the sensitive area uses Cipher Feedback (CFB) mode.

The method of generating the encryption key and IV for the encryption is specified in clause 22.

26.7 Sensitive Area Integrity

When an object is not in a Shielded Location, it is susceptible to modification through means other than through a Protected Capability. An HMAC-based integrity scheme allows these modifications to be detected. The integrity HMAC includes the sensitive data and some representation of the public area. Inclusion of the public area preserves the binding between the two elements of the object.

The HMAC key is generated from the same seed that is used for generating the symmetric encryption key and IV. The HMAC of the protected structure is required to be checked before the sensitive area is decrypted.

27 Object Creation

27.1 Introduction

TPM2_Create() and TPM2_CreatePrimary() are used to create the objects (keys and data) that are part of a TPM's Storage hierarchy. TPM2_CreatePrimary() is used to create Primary Objects that are derived from a Primary Seed. TPM2_Create() is used to create other Objects that are generated with values from the TPM RNG. The "parent" of a Primary Object is a Primary Seed value, and the parent of an Ordinary Object is a Storage key.

Authorization to use the parent is required in order to create a child

All of the objects created by these commands are similar in most respects and the parameters required to create an object are the same for each command. They are:

- a public area template,
- the sensitive values, and
- the creation PCR selection.

Any type of object that can be created with TPM2_Create() can be created with TPM2_CreatePrimary().

The sensitive area of a Primary Object does not leave the TPM except in a saved context or by duplication. If the Primary Object is not made persistent in the TPM (TPM2_EvictControl()) it will need to be recreated after each TPM Reset. If it is not context saved, it will need to be recreated after the next TPM2_Startup().

27.2 Public Area Template

27.2.1 Introduction

A public area template describes the desired attributes of the object to be created. The TPM uses this template to guide the creation of the new object.

The format of the template has to match the desired format of the object to be created, in all details. The item-specific information (*unique*) will be replaced by the TPM in the creation process.

In general, the fields in the public area are checked as if the object were being loaded under the parent indicated in the creation command.

27.2.2 type

This parameter indicates the basic type of the object and determines the format of the *parameters* and *unique* fields. The type may indicate a symmetric key, an asymmetric key, or a data value.

The allowed values for *type* are: TPM_ALG_SYMCIPHER, TPM_ALG_KEYEDHASH TPM_ALG_RSA, or TPM_ALG_ECC.

NOTE The list of types could change. If an algorithm ID is allowed for use as a public area type, it is denoted by an "O" in the "Type" column of the TPM_ALG_ID constants table published in the TCG Algorithm Registry.

27.2.3 nameAlg

The *nameAlg* parameter in the template is set according to the object type. If the object is a restricted-decryption key, then the object is required to have the same *nameAlg* as the parent. For all other cases, the *nameAlg* may be any supported hash algorithm.

In the case of TPM2_LoadExternal(), *nameAlg* is allowed to be TPM_ALG_NULL. When this value is selected, the TPM does not validate the cryptographic linkage between the public and sensitive portions of the object. Since the *nameAlg* is TPM_ALG_NULL, the object has no Name.

NOTE Certification of the key with no Name has no meaning as the certification will have no Name for the certified object.

27.2.4 objectAttributes

These flags must be set according to the rules appropriate for loading the object. The required settings are found in ISO/IEC 11889-2, in the definition of TPMA_OBJECT.

27.2.5 authPolicy

If use of an object is to be gated by a policy (including PCR), the template will contain the policy hash. Otherwise, this entry will be set to TPM_ALG_NULL.

27.2.6 parameters

This field contains parameters that describe the details of the object indicated in *type*.

For a Storage Key that has *fixedParent* SET in its *objectAttributes*, these parameters will be identical to the parameters of the parent. For other objects, these parameters may vary according to the *type* and application.

27.2.7 unique

The *unique* field of the template is the only field in the public area that is replaced by the TPM during the object creation process. The caller may place any value in this field as long as the structure of the value is consistent with the *type* field. That is, this field should be structured in the same way as the data that will be placed in this field by the TPM. The caller may also set the size of this field to zero and the TPM will replace it with a correctly sized structure.

27.3 Sensitive Values

27.3.1 Overview

The sensitive values that are provided when the object is created allow initial setting of the *authValue* for the object and may provide some other object-sensitive value. The sensitive value may be an encryption key or sealed data.

The sensitive values provided to the TPM in TPM2_Create() and TPM2_CreatePrimary() (the inSensitive parameter) may optionally be encrypted using standard session-based encryption techniques. Since session-based encryption allows use of a different session for authorization and encryption, the session used for encrypting the authorization and other sensitive data does not have to be the same as the authorization session for the parent of the newly created object. This ensures that the entity that controls the parent does not automatically gain access to the secret values of a child.

27.3.2 userAuth

The *userAuth* value is the initial *authValue* for the created object. This value may be no larger than the digest produced by the *nameAlg* of the object.

NOTE This limitation ensures that any valid *authValue* will be usable on any TPM that can load the key. If this limitation were not imposed, then some TPM might not be able to load a duplicated object because the *authValue* was too large for the implementation.

27.3.3 data

This contains information that the caller wants to be incorporated in the sensitive part of the created object. This may be either a symmetric key or user data. If *data* is an Empty Buffer, then the *sensitiveDataOrigin* attribute of the template is required to be SET. If data is not empty, then *sensitiveDataOrigin* is required to be CLEAR.

If the object type is TPM_ALG_KEYEDHASH and both *sign* and *encrypt* are CLEAR, then the created object is a Sealed Data Object and the TPM will return an error (TPM_RC_SIZE) if *data* is an Empty Buffer.

If the created object is an asymmetric key, then *data* is required to be an Empty Buffer and *sensitiveDataOrigin* in the template is required to be SET.

NOTE If the caller were allowed to specify the private key, then for some types of asymmetric algorithms (such as, ECC) the actions of the TPM would not determine the Name of the object. Since the TPM has no effect on the creation of such an object, the preferred means of having such a key become part of a hierarchy is to import it with TPM2_Import().

27.4 Creation PCR

The PCR selection that is present in TPM2_Create() or TPM2_CreatePrimary() parameters is used to select the PCR values that will best represent the environment in which the object was created. The selection and the PCR are hashed according to the creation data algorithm and included in the creation data (a TPM2B_CREATION_DATA) that is returned in the command response.

NOTE When an Object is created, the TPM produces a ticket that it (the TPM) can use to verify that it created the Object. This allows the TPM to certify that it created the Object (TPM2_CertifyCreation()).

27.5 Public Area Creation

27.5.1 Introduction

This clause describes how the TPM uses the parameters of TPM2_Create() and TPM2_CreatePrimary() to set the values in the public area of the created object.

This clause does not describe the error conditions if the parameters are bad. That information is provided in the description of TPM2_Create() and TPM2_CreatePrimary() in ISO/IEC 11889-3.

27.5.2 type, nameAlg, objectAttributes, authPolicy, and parameters

The TPM will validate that these parameters are consistent in the template and then copy them from template into the created structure without modification.

27.5.3 unique

27.5.3.1 Introduction

This parameter will contain a *type*-specific structure. It is used to ensure that each object has a statistically unique identity. The methods used to create *unique* ensure that it is cryptographically bound to the contents of the sensitive area. Creation of *unique* from the sensitive data uses non-invertible processes so that the *unique* value does not compromise the confidentiality of the sensitive area.

EXAMPLE A hash is an example of a non-invertible processes.

The computation of *unique* uses one or more values in the sensitive area of the object. At least one of the sensitive area values will be provided by the TPM to ensure that *unique* is, in fact, unique. For asymmetric keys, uniqueness is provided by the public key and the public key is mathematically linked to the private key in the sensitive area.

For symmetric objects (symmetric keys, HMAC keys, and data blobs), the key (or data) is hashed with a TPM-generated obfuscation value and the resulting digest is used as the *unique* value.

There are two reasons for generating the *unique* parameter for symmetric objects in this way. The first is that it protects the contents of the user-provided data. If the secret data has low entropy, then making the *unique* parameter a simple digest of that data would allow an offline attack to determine what the secret data might be. The large, random, obfuscation value generated by the TPM is not known to an attacker, which mitigates this threat.

The second reason for this method is that it prevents an attacker from stealing an object's identity. If the identity were not based on the contents of the sensitive area, then an attacker could create a sensitive structure and associate it with the public area of any symmetric object. Having the sensitive area contain information that can cryptographically link the sensitive area to the public area prevents this kind of substitution.

The methods for producing *unique* for each of the object types are specified in the remainder of 27.5.3.

27.5.3.2 TPM_ALG_KEYEDHASH

This type is used for an HMAC key or data block. The computation for *unique* for a KeyedHash object is:

$$unique := \mathbf{H}_{nameAlg}(obfuscate \,||\, key) \tag{59}$$

where

$\mathbf{H}_{nameAlg}$	hash using *nameAlg* from the object template
obfuscate	the contents of *seedValue.buffer* in the object's sensitive area
key	the contents of *sensitive.bits.buffer* in the object's sensitive area; this will be either an HMAC key, a data blob, or a symmetric key.

27.5.3.3 TPM_ALG_SYMCIPHER

This type is used for a symmetric block cipher key. The *unique* value is computed as shown in (59).

27.5.3.4 TPM_ALG_RSA

For an RSA key, *unique* is the public modulus of the key. It is computed as specified in D.2.

27.5.3.5 TPM_ALG_ECC

For an ECC key, *unique* is the public point computed as specified in D.3.

27.6 Sensitive Area Creation

27.6.1 Introduction

This clause indicates how the TPM creates the sensitive portion of an object (a TPMT_SENSITIVE).

The process for computing the contents of a sensitive area is determined by the type of the object, indicated in the *type* field of *template*.

Some of the sensitive area fields may contain data that is provided by the caller. Some of the fields are always provided by the TPM. When a TPM-provided field is in a Primary Object, the TPM-provided data is always derived, in some way, from the associated Primary Seed such that the same Primary Object can be reproduced as long as the associated Primary Seed remains unchanged. For Ordinary Objects, an implementation may either get the TPM-provided data from the RNG, or compute the fields of the object as if it were a Primary Object; but with a random number used in place of a Primary Seed.

The performance difference between the two methods of producing asymmetric objects is negligible as the majority of the work is in validating the choices rather than in generating them. For symmetric objects, the difference might be worth having different methods for Primary and Ordinary Objects but there is an added cost in development and testing that could offset the benefit of any slight performance advantage.

For Ordinary Objects, the method used for generating *sensitive* should be used for generating *seedValue*. That is, if *sensitive* is generated by taking values from the RNG, then *seedValue* should be generated by taking values from the RNG. If *sensitive* is generated by creating a random seed and using the methods used for Primary Keys, then that same seed should be used for generating *seedValue*.

27.6.2 type

The *type* parameter of the object's sensitive area is a copy of the type parameter from the object's public-area template.

27.6.3 authValue

The *authValue* of the object is copied from the *userAuth* field of the *inSensitive* parameter of TPM2_Create() or TPM2_CreatePrimary().

27.6.4 *seedValue*

The *seedValue* field is used for the *obfuscation* value of symmetric keys, HMAC keys, and data objects. It is also used to hold the symmetric seed value for asymmetric Storage Keys. For all object types, when present, *seedValue* is the size of the digest produced by the *nameAlg* of the object.

seedValue is only needed if the asymmetric key is a Storage Key. This value is used as a seed for generating the integrity and confidentiality values for protecting the child objects of the key. The size of *seedValue* is the digest size of the *nameAlg* of the object. Presuming that the protection algorithms of a Storage Key are reasonably balanced (a requirement), then this size of seed will provided adequate entropy for generation of the various keys required for protection of the child object.

For an Ordinary Object, *seedValue* can be created by taking bits from the RNG or generated using the same method used for Primary Objects. That method uses **KDFa**() as shown below.

$$seedValue := \mathbf{KDFa}\ (hashAlg,\ seed,\ \text{seedValue},\ tName,\ proof,\ bits) \quad (60)$$

where

hashAlg	in TPM2_Create(), the *nameAlg* of the parent; in TPM2_CreatePrimary(), the context integrity hash algorithm
seed	for a Primary Object, the Primary Seed; for all other objects, a random number with the same size as a Primary Seed
seedValue	a null-terminated, vendor-specific string different from any other label used for **KDFa**()
tName	Name of the creation template computed using the *nameAlg* in the template
proof	if the object being created is a non-duplicable Primary Object in the Endorsement Hierarchy, then this is *ehProof* and for all other objects, this is an Empty Buffer
bits	the number of bits in the digest produced by *nameAlg* in the creation template

NOTE The use of *ehProof* in Primary Storage Keys in the Endorsement Hierarchy ensures that user created child keys in that hierarchy are no longer useable after the owner is changed using TPM2_Clear().

27.6.5 sensitive

27.6.5.1 Symmetric Objects

Symmetric objects have a *type* of TPM_ALG_SYMCIPHER or TPM_ALG_KEYEDHASH. For a symmetric object, the sensitive object data may be provided by the caller or generated by the TPM.

If *sensitiveDataOrigin* attribute in the object template is CLEAR, then the sensitive data is provided by the caller. If provided by the caller, the sensitive data will be in the *data* field of the *inSensitive* parameter of TPM2_Create() or TPM2_CreatePrimary().

If *sensitiveDataOrigin* is SET, it indicates that the TPM is the source of the sensitive data and the data field of the *inSensitive* parameter is required to be an Empty Buffer.

A user provided symmetric key is required to be the size indicated by *parameters.symDetail.keyBits.sym* in the template. It is the number of octets required to hold the number of bits indicated.

NOTE 1 If the key has fewer significant digits that necessary, pad octets of zero are needed. The pad octets are added to the high-order end of the key.

A user provided HMAC key is not allowed to be larger than the smaller of the block size of the hash algorithm or 128 octets. Limiting the size to 128 octets is for compatibility of structures between TPM.

NOTE 2 The HMAC algorithm needs keys larger than the hash block size be hashed before use. This could result in fewer bits of entropy in the HMAC key than expected by the caller. The TPM will not allow the caller to specify an overly large value for the HMAC key. If the caller desires to use a larger value, they can perform the digest externally and pass the resulting digest to the TPM for use as the HMAC key.

If not provided by the caller, *sensitive* is generated by the TPM. For a TPM_ALG_KEYEDHASH object, the size is the digest size of the nameAlg of the object. For a TPM_ALG_SYMCIPHER object, the size is equal to (*parameters.symDetail.keyBits.sym* + 7) / 8.

For an Ordinary Object, *sensitive* can be created by taking bits from the RNG. *Sensitive* may also be generated using the same method used for Primary Objects. The method used for Primary Objects uses **KDFa**() as shown below.

$$sensitive := \mathbf{KDFa}\ (hashAlg,\ seed,\ \text{sensitive},\ tName,\ \text{NULL},\ bits) \quad (61)$$

where

hashAlg	in TPM2_Create(), the *nameAlg* of the parent; in TPM2_CreatePrimary(), the context integrity hash algorithm
seed	for a Primary Object, the Primary Seed; for all other objects, a random number with the same size as a Primary Seed
sensitive	a null-terminated, vendor-specific string different from any other label used for **KDFa**()
tName	Name of the object template computed using the *nameAlg* in the template
bits	for a TPM_ALG_SYMCIPHER object, the value in *parameters.symDetail.keyBits.sym*; for all other objects, the number of bits in the digest produced by the object's *nameAlg.*

27.6.5.2 Asymmetric Objects

The *sensitive* field in an asymmetric key object is the private key. The key is generated in a way that is specific to the algorithm and is specified in an algorithm-specific annex of this part of ISO/IEC 11889.

EXAMPLE RSA key generation is specified in D.2 and ECC key generation is specified in D.3.

27.7 Creation Data and Ticket

When it creates an object, the TPM also creates a data structure that describes the environment in which the object was created. This data includes:

- a digest of selected PCR at the time of object creation and a bit-map indicating the PCR that were included in the list. The PCR selection is those PCR indicated in the call to TPM2_Create() and TPM2_CreatePrimary().
- the locality at which the object was created
- the *nameAlg* of the parent. If the parent is a Primary Seed, then the algorithm will be TPM _ALG_NULL.
- the Name of the parent. If the parent is a Primary Seed, then the Name will be the handle of the seed.
- the Qualified Name of the parent. If the parent is a Primary Seed, then the Qualified Name will be the handle of the seed.
- some additional data provided by the caller that is to be associated with the new object

In addition to these values, the TPM will create a ticket that will allow the TPM to validate that the creation data was generated by the TPM.

The creation data will act as a form of certification of the object that is most useful when *fixedTPM* is CLEAR in the created object. Without this information, it would not be possible to determine how the object came to be in the hierarchy where it is found. When the object is moved, it would be up to the duplication authority to provide some certification of the duplication process. If there is no creation data

indicating that the object was created in the place where it was found, and there is no certificate from the duplication authority for the object, then it may be difficult to establish the trustworthiness of the object.

NOTE In this case, the trustworthiness of the object refers to determining that the sensitive area of the object has only ever been accessible by trusted entities such as other TPMs.

27.8 Creation Resources

When a Primary Object is created, it is also loaded in a TPM object slot and the handle is returned. If no free object slot is available, the TPM will return TPM_RC_OBJECT_MEMORY.

When creating an ordinary object, the TPM may use an object slot as scratch memory in which it builds the object. If the implementation does use this scheme and no object slot is available, then the TPM will return TPM_RC_OBJECT_MEMORY.

28 Object Loading

28.1 Introduction

An object is either a key or data that can be loaded into the TPM for use. An object must be loaded before the TPM can use or modify the object. Loading may require that the USER role authorization for the parent be provided

28.2 Load of an Ordinary Object

It is possible to load just the public portion of an object into the TPM (TPM2_LoadExternal()), or to load both the public and private portions (TPM2_Load()). If the sensitive area is to be manipulated or used, then both portions are required to be loaded.

When loading an object, multiple consistency checks are performed. Among these checks:

a) Is the HMAC of the encrypted private area correct – this ensures that the sensitive area was not modified, that the sensitive area and the provided public area are matched, and that the object is a descendant of the parent.

b) Is the unique parameter of the public area cryptographically bound to the sensitive data – this is required to prevent improper association of a public area with a sensitive area. If this check were not done, an attacker could use a public area that had a Name that is the same as a different object and associate a different sensitive area with the public area. If the object were used in TPM2_PolicySecret(), the attacker could get the TPM to create a *policyDigest* with any desired hash value.

EXAMPLE A legitimate policy uses signature validation of a key with Name1. An attacker could create an object with Name1 (copy the data from the legitimate key) and then create a sensitive area that had an *authValue* known to the attacker, instead of using TPM2_PolicySigned() to create the policy.

c) Are the attributes consistent – these values need to be checked even if the integrity check indicates that the values were not modified. This is because the object may have been created by software using inconsistent values. The integrity may be good but the values may be wrong.

1) If *fixedTPM* is SET, *fixedTPM* must also be SET in the parent.

NOTE If *fixedTPM* is properly SET, then the other checks need not be made because the object is verified to have been created on the TPM that loaded the object, so the other attributes are known to be correct.

2) If *fixedParent* is CLEAR, then *fixedTPM* must also be CLEAR.

3) If *restricted* is SET, only one of *sign* or *decrypt* may be SET.

28.3 Public-only Load

There are several cases when only the public portion of an asymmetric key can be loaded. The public-only load of an object requires that the caller associate the object with one of the hierarchies. This association is needed when the key is used for signature verification so that the TPM can determine which proof value to use in the ticket.

EXAMPLE Duplication or signature verification are example cases when only the public portion of an asymmetric key can be loaded.

A public-only load occurs when the *inPrivate* parameter to TPM2_LoadExternal() has a size of zero.

28.4 External Object Load

External Objects allow the cryptographic processes of the TPM to be used on keys that are not part of a TPM hierarchy. The public portion of an asymmetric key may be loaded so that the TPM can be used to validate a signature. A symmetric key may be loaded so that the symmetric engines of the TPM may be used to encrypt or decrypt data.

TPM2_LoadExternal() is used to load an External Object. When only the public portion is loaded, the attributes of the object are arbitrary but the structures are required to be consistent with the type. That is, if an RSA signing key is loaded, the signing scheme must be a valid scheme for an RSA key.

When the sensitive portion of the object is loaded (such as, a symmetric key), the sensitive area is not encrypted by a parent but may be encrypted using parameter encryption. The *fixedParent* and *fixedTPM* attributes are required to be CLEAR when both parts are loaded. This check allows the object to be used in any command that is valid for the type including certification.

NOTE If an entity has access to both the public and sensitive portions of a key, then the entity could import the key and then certify it.

An external object can be associated with a hierarchy when it is loaded. This allows creation of tickets that are specific to a hierarchy in commands.

EXAMPLE An example command that creates tickets specific to a hierarchy is TPM2_VerifySignature().

If the hierarchy with which an External object is associated is disabled, the object will be flushed. If the associated hierarchy is disabled when TPM2_LoadExternal() is called, the object will not load.

29 Object Creation in Reference Implementation

A Primary seed is used in the creation of a Primary Object. When a TPM-generated value is needed in an Object, an additional iteration of the KDF using the Primary seed produces additional pseudo-random values.

EXAMPLE When the object has a *seedValue* (a keyedHash, symCipher, or parent object), the *seedValue* will be populated with a value based on the Primary Seed. This ensures that all of the TPM-generated values of a specific Primary Object can be recreated as long as the Primary Seed remains the same.

NOTE 1 For a given seed, the uniqueness of a Primary Object is determined by the template used in its creation.

For simplicity in the implementation, the same methods are used for both Primary and Ordinary Objects. The difference being that, instead of using a Primary Seed, a random seed is generated from the TPM's RNG.

Implementations are not required to use the seed-and-KDF method for generating objects. One of the more likely places for using a different method is in the generation of primes for use in RSA keys. For most other object types, there is little advantage in using different methods for generation of Primary and Ordinary Objects but such a possibility is not prohibited by ISO/IEC 11889.

If using the seed-and-KDF method for key creation, the seed value for use in the object creation should be twice the size as the security strength of the object. For a *symCipher* object, its security strength will generally be the size of the symmetric key. For a *keyedHash* object, the security strength will generally be the same as the digest used in the *keyedHash* object. For an asymmetric key, the security strength varies according to the algorithm. For a data object, the security strength is assumed to be the digest size of the *nameAlg*.

NOTE 2 This arbitrary assumption ensures that the *unique* field of the public area has sufficient entropy.

30 Context Management

30.1 Introduction

To allow the TPM to be shared among many applications, the TPM supports context management. The objects and sessions used by an application may be loaded into the TPM when needed and saved when a different application is using the TPM. The TPM Resource Manager (TRM) is responsible for swapping the contexts so that the necessary resources are present in the TPM when needed.

There are two types of contexts: those associated with Transient Objects, and those associated with authorization sessions.

The four commands used to manage the contexts are

1) **TPM2_ContextSave()** – the TPM integrity protects, encrypts, and returns the context associated with a handle,
2) **TPM2_ContextLoad()** – allows a previously saved context to be loaded to TPM RAM and have a handle assigned,
3) **TPM2_FlushContext()** – the context information associated with the specified handle is erased from TPM RAM, and
4) **TPM2_EvictControl()** – allows the owner or the platform firmware to designate objects that are to remain TPM-resident over TPM2_Startup() events. This command will return a new handle.

A saved context is cryptographically bound to a specific TPM so that it may not be loaded on a different TPM. This binding is provided by using a statistically unique proof value in the generation of the protection values for a context (see 30.3 and 30.3.2). When the proof value of a hierarchy changes, saved object contexts belonging to that context can no longer be loaded into the TPM. The proof value for a context will change when its Primary Seed changes. Additionally, *ehProof* will change when either the SPS or EPS changes.

NOTE 1 In the reference implementation, *ehProof* is a non-volatile value from the RNG. It is allowed that the *ehProof* be generated by a KDF using both the EPS and SPS as inputs. If generated with a KDF, the ehProof can be generated on an as-needed basis or made a non-volatile value.

Saved contexts for all objects and sessions are invalidated on a TPM Reset. In the reference implementation, the encryption keys for contexts are changed by TPM Reset so previously saved contexts may no longer be loaded. Saved session contexts remain valid until the session is closed, or TPM Reset. If the *stClear* attribute of an object is SET, then saved contexts for the object are invalidated on either TPM Reset or TPM Restart (that is, any time the TPM does a Startup(CLEAR). If the *stClear* attribute of an object is CLEAR, then the saved contexts for that object are valid and may be loaded into the TPM until the next TPM Reset.

NOTE 2 In the reference design, when an object context is saved, the current value of *clearCount* is placed in the context. When the context is loaded, if the object is a *stClear* object, the value in the object is compared to the current value of *clearCount*. If they are not the same, then the context load fails.

Objects and sessions are not retained in TPM memory after a TPM2_Startup() and it is necessary for the TRM to save the contexts for any session or object that is to be useable after TPM Restart or TPM Resume.

NOTE 3 The TPM might lose power between a TPM2_Shutdown(TPM_SU_STATE) and the subsequent TPM2_Startup(). With respect to context preservation, the TPM behavior is defined to be the same whether the TPM loses power or not.

The structure of a saved context may be defined by the vendor but a saved context is required to have its integrity and confidentiality protected by cryptographic means. ISO/IEC 11889-3 and ISO/IEC 11889-4 implement the normative methods for providing confidentiality and integrity protection for saved contexts. These protections are specified in more detail in subsequent parts of clause 30.

NOTE 4 The algorithms chosen for integrity and confidentiality protection of a saved context are vendor specific. However, the cryptographic strengths of the algorithms used need to be the highest of any algorithm of the same type implemented on the TPM.

30.2 Context Data

30.2.1 Introduction

When a context is saved, the saved context data structure contains:

- a sequence number,
- a handle, *savedHandle*

  NOTE For transient objects, this *savedHandle* in a saved context data structure is not the same as the handle used by the TPM to reference loaded objects and by TPM commands to describe the object being operated on.

- a hierarchy selector,
- an integrity HMAC, and
- an encrypted data blob.

The encrypted data blob contains the data necessary to reconstruct the full object or session context in the TPM. The other fields are defined in the remainder of clause 30.2.

The structure of the context is vendor-specific and may contain both confidential and non-confidential data. ISO/IEC 11889 requires encryption of the entire context.

30.2.2 Sequence Number

New protection values are generated each time a context is saved. The protection values are an HMAC key, a symmetric key, and an initial value. The values are made unique by including a counter value in the generation process (see 30.3 and 30.3.2). The counter value used for the context is stored in the sequence number field of the context structure. Two counters are used for generating the sequence numbers. One counter is used for transient and sequence object contexts. A second counter is used for session contexts.

There are two counters used to provide sequence numbers. The counter (*objectContextID*) provides sequence numbers for transient and sequence objects. This counter is incremented each time an object context is saved. The counter (*contextCounter*) is used to provide sequence numbers for sessions and increments when a session context is created or loaded (its behavior is specified in more detail in 30.5). When creating the context structure, the TPM sets the *sequence* parameter to the value of the counter used in the generation of the protection values for the context.

When a context is loaded, (TPM2_ContextLoad()), the TPM checks that the *sequence* parameter is in a viable range before starting the operation. For an object, the viable range is any number that is less than the current value of the object sequence counter. For a session, the sequence number must also be less than the session sequence number but it must also be greater that the sequence number minus the allowable range for session number.

In the reference implementation, *objectContextID* is a 64-bit counter that is initialized to zero at startup and is expected to never overflow. The size is platform-specific.

EXAMPLE For purposes of this example, assume that the sequence counter value is only 16 bits and that the session counter indicates the last assigned session context had a value of 10 10_{16}. It would then be an error if the *sequence* parameter in a loaded session context is greater than 10 10_{16}. Assume further that the TPM only allows a range of 256 between session values (explanation in 30.5). Then it would be an error if the sequence parameter of the session in TPM2_ContextLoad() is less than 10 10_{16} – 01 00_{16} = 0F 10_{16}; and the TPM will not load the context.

30.2.3 Handle

The savedH*andle* number for a context indicates the type of the context (object or authorization session). The type of the context is used to determine how to reconstruct the protection values for validation of the context. If the savedHandle value in the context is changed by software, the context will not load.

For a session, the same handle is assigned to the context whether the context is loaded in the TPM or in a saved context. That is, *savedHandle* is the same as the handle the TPM uses to refer to the session. A session handle will have an MSO of TPM_HT_HMAC_SESSION (02_{16}) or TPM_HT_POLICY_SESSION (03_{16}). The range of values in the handle index (the low-order three octets of the handle) is TPM dependent. In the reference implementation, the low order bits of the session context handles fall within a range from 0 to MAX_ACTIVE_SESSIONS – 1 and the TPM will generate an error and do no further processing of the context if the handle is outside of this range.

A savedHandle MSO of TPM_HT_TRANSIENT (80_{16}), indicates that the context is an Object. For an object, the savedH*andle* parameter of the context structure does not indicate the handle value used by the TPM to reference the object (when a transient object context is not on the TPM, the TPM retains no information about that context). Therefore, the savedH*andle* value is not used for transient object contexts in the same way that it is used for session contexts. Instead, the savedHandle is used to indicate the type of the transient object context.

Three savedH*andle* values are defined for transient object contexts:

1) 00 00 00_{16} – indicates a transient Object that does not have the *stateClear* property;

   NOTE An Object has the *stateClear* property when *stClear* is SET in the Object or in any of its ancestor keys.

2) 00 00 01_{16} – indicates a sequence Object (see 32.4.5); and

3) 00 00 02_{16} – indicates a transient Object that has the *stateClear* property.

EXAMPLE A sequence Object will have a 32-bit handle value of 80 00 00 01_{16}.

If the *savedHandle* type is TPM_HT_TRANSIENT, the TPM will not generate or load a context with any other value besides the three values described above for the handle's index.

Objects that have the *stateClear* property are invalidated by Startup(CLEAR). To enforce this, the TPM will include *clearCount* in the integrity value of the Object.

TPM processing of contexts with *savedHandle* values of 80 00 00 00_{16} or 80 00 00 01_{16} is the same. The reason for differentiating sequence Objects is to identify the context for the convenience of the TPM resource manager (TRM). The TRM needs to manage sequence objects differently from other transient Objects. Because the context of a sequence object changes each time the sequence is updated, the context needs to be saved each time the context is used. The context of a transient Object does not change on use.

30.2.4 Hierarchy

The hierarchy parameter of the context indicates which of the hierarchy proof values are used in the creation of the protection values for the context. For objects, this value is determined by the hierarchy of the object and may be TPM_RH_NULL for a Temporary Object. Sequence objects and sessions are in the Null hierarchy.

30.3 Context Protections

30.3.1 Context Confidentiality Protection

A symmetric block cipher is used to protect the confidentiality of a saved context. The algorithm is selected by the TPM vendor but is required to have the highest security strength of any symmetric block cipher implemented on the TPM.

When the context is created by TPM2_ContextSave(), the value of *sequence* is stored in the context before it is encrypted. When the context is loaded, the value of *sequence* is compared to the value in the loaded context after it is decrypted. If the values are not the same, then the TPM will enter failure mode as this is symptomatic of a specific type of power analysis attack.

The symmetric key and IV are regenerated when a context is loaded. It is required that the symmetric key and IV not be generated until the context integrity has been validated.

NOTE 1 This restriction prevents simultaneous power-analysis attacks on the integrity and encryption values of a context. Since the integrity is checked first, no attempt is made to create the symmetric key if the integrity check fails.

KDFa() is used to generate the symmetric encryption key and IV for context encryption. The parameters of the call are:

$$(symKey, symIv) := \mathbf{KDFa}\,(hashAlg, hProof, vendorString, sequence, handle, bits) \quad (62)$$

where

hashAlg	a hash algorithm chosen by the vendor
hProof	the proof value associated with the hierarchy associated with the context
vendorString	a value used to differentiate the uses of the KDF
sequence	the sequence parameter of the TPMS_CONTEXT
handle	the handle parameter of the TPMS_CONTEXT
bits	the number of bits needed for a symmetric key and IV for the context encryption

NOTE 2 The value of *vendorString* needs to be different from any other label string used in a **KDFa**() call. The reference implementation uses “CONTEXT”

NOTE 3 The *nullProof* is used as the hProof value for a context in the Null hierarchy so that the encryption keys do not repeat and so that they change on each TPM Reset.

The key and IV produced in (62) are used to encrypt the object or session context

$$encContext := \mathbf{CFB}_{symAlg}\,(symKey, symIv, context) \quad (63)$$

where

$\mathbf{CFB}_{symAlg}$	symmetric encryption in CFB mode using a symmetric algorithm chosen by the TPM vendor
$symKey$	symmetric key from (62)
$symIv$	IV from (62)
$context$	the context being protected (a TPM2B_CONTEXT_DATA)

NOTE 4 The *size* field and the *buffer* field of *context* are encrypted.

30.3.2 Context Integrity Protection

The integrity of a saved context is protected by an HMAC using a hash algorithm selected by the TPM vendor. The hash algorithm chosen is required to have the highest security strength of any hash algorithm implemented on the TPM.

The HMAC is constructed using the proof value associated with the hierarchy to which the object belongs. Since the proof value changes when the associated Primary Seed changes, HMAC validation for a previously saved context will fail when the associated Primary Seed changes, and that context may no longer be loaded. Other values in the HMAC computation serve to invalidate other context subsets without necessarily invalidating them all.

EXAMPLE The *clearCount* value is included in the HMAC of a context for an object with the *stClear* attribute so that the context will be invalidated on each TPM Restart as well as each TPM Reset.

The only TPM state change that invalidates all saved contexts is TPM Reset.

Sessions, Sequences, and Temporary Objects are in the "null" hierarchy.

The HMAC integrity computation for a saved context is:

$$contextHMAC := \mathbf{HMAC}_{vendorAlg}(hProof, resetValue \{ || clearCount\} || sequence || handle || encContext) \quad (64)$$

where

$\mathbf{HMAC}_{vendorAlg}$	HMAC using a vendor-defined hash algorithm
hProof	the hierarchy proof as selected by the hierarchy parameter of the TPMS_CONTEXT
resetValue	either a counter value that increments on each TPM Reset and is not reset over the lifetime of the TPM; or a random value that changes on each TPM Reset and has the size of the digest produced by vendorAlg
clearCount	a counter value that is incremented on each TPM Resume and may be incremented or set to zero on TPM Reset. This value is only included if the handle value is 80 00 00 02_{16}.
	NOTE The handle value is 80 00 00 02_{16} when the *stClear* attribute of the object is SET or when the *stClear* attribute is set in one of the object's ancestor keys.
sequence	the sequence parameter of the TPMS_CONTEXT
handle	the handle parameter of the TPMS_CONTEXT

encContext the encrypted context blob

30.4 Object Context Management

When an object's context is saved, a copy of the object context is integrity checked, encrypted, and returned to the caller. The original context remains in the TPM and the TPM retains its handle. A saved object context may be reloaded into the TPM with TPM2_ContextLoad(). If the TPM has sufficient memory available, it will load the object and assign a handle. If other copies of the same object are in TPM memory, they are unaffected. An object context is only removed from TPM memory with TPM2_FlushContext(), deletion of the associated hierarchy seed, or TPM2_Startup().

The handle assigned to an object when it is loaded may not be assigned to any other TPM resource, object, or session. When the object is flushed from TPM memory, its handle may be assigned to another TPM resource when it is loaded or created.

Software may create as many copies of an object context as desired. When an object is not in TPM memory, it has no associated handle. If an object context is saved and subsequently reloaded, it is likely that a different handle will be assigned to the object.

When the Primary Seed is changed for the hierarchy associated with an object, all objects associated with that hierarchy are flushed from TPM memory. The TPM will no longer load saved contexts associated with the previous Primary Seed.

When an attempt is made to load an object or an object context (TPM2_Load(), TPM2_CreatePrimary(), TPM2_LoadExternal() or TPM2_ContextLoad()) and the TPM does not have sufficient RAM to hold the object, the TPM will return TPM _RC_OBJECT_MEMORY or TPM _RC_MEMORY. This warning code is normally handled by the TRM. It indicates that an object or a session needs to be unloaded from TPM memory before the command can complete. If the TPM returns TPM_RC_OBJECT_MEMORY, it indicates that an object must be flushed from TPM memory. If the TPM returns TPM_RC_MEMORY, then it is possible that removal from TPM RAM of either an object or a session would allow the command to complete.

When a command references a persistent object, the TPM may move the object from NV into an object slot. If no slot is available, the TPM will return TPM_RC_OBJECT_MEMORY.

An implementation is allowed to use an object slot for temporary memory in execution of TPM2_Import() and return TPM_RC_OBJECT_MEMORY if a slot is not available.

If the TPM uses an object slot for temporary memory, the slot will be freed at the end of the command in which the slot was allocated.

If a TPM receives Shutdown(STATE) before the _TPM_Init, then the saved object contexts will continue to be usable after a TPM Restart or TPM Resume. An exception is that an object may be created with the *stClear* attribute. If this attribute is SET in an object or an ancestor of an object, then the saved context will be invalidated on TPM Restart. All saved object contexts are invalidated by TPM Reset.

30.5 Session Context Management

A session context is created by TPM2_StartAuthSession(). The context associated with a session is unique. That is, the data describing the session's state may be either on the TPM or saved off the TPM, but not both. Further, a session context may only be loaded once. These limitations on the session context are intended to prevent possible attacks based on replay of authorizations.

The handle associated with a session does not change as long as the session is active. The session is active until closed by the *continueSession* flag being FALSE or until the session context is flushed from the TPM by TPM2_FlushContext().

The nominal implementation uses a volatile counter (*contextCounter*) that increments each time a session is created or context loaded. This count value is assigned to the created or loaded session context and serves as a version number for the session context. If the session context is saved and reloaded, it is assigned a new version number. *contextCounter* is saved by Shutdown(STATE) and reset on TPM Reset.

The TPM maintains a database of concurrent sessions so that it can validate that a reloaded session context is the most recent version. It is required that the TPM be able to ensure that the restored context is the correct context regardless of the number of contexts created.

The size of *contextCounter* affects the size of the memory required for tracking each of the contexts. It is therefore desirable that the counter only be large enough for the majority of applications, meaning that it will not be large enough for all applications. In those applications, a method is required to handle counter rollover.

One scheme for handling rollover is to maintain an even/odd interval.

EXAMPLE If a nonce were being used for each interval, then the TPM could maintain two nonces, one to be used when the MSb of the volatile counter is 0 and the other when the MSb is 1. When the counts of all the sessions have the same MSb, then a new nonce can be created for use when the MSb changes. This scheme works unless a session has a long lifetime. That is, if the session is created when the MSb is 0, and the session is still active when the counter reaches its maximum value with all bits equal 1, then the context with an MSb of 0 will need to be discarded.

Rather than have the old session be automatically flushed, the TPM provides an indication that it is reaching its limit and that one or more saved session contexts need to have their *sequence* number updated to the current interval in preparation for the context counter rollover.

The indication that the context counter is approaching its limit is provided when an authorization session is created or loaded. If the creation or loading of a session would make it impossible for the TPM to bring all contexts into the current interval, then it would return an error (TPM_RC_CONTEXT_GAP) and not create or load the new session. On receiving this error, the management software either would explicitly flush old session contexts or would load the old session contexts to update their associated counter values.

When the TPM returns TPM_RC_CONTEXT_GAP, it will not allow an authorization session to be created and it will only allow the oldest authorization session to be loaded. When the oldest session is loaded, its *sequence* number is updated. It may be used or saved with its new *sequence* number.

NOTE The TPM needs to provide the indication of the session-tracking limit being reached before the maximum count is reached. If there are three sessions in the 'odd' interval and the end of the 'even' interval is being reached, then the TPM needs to indicate the limit while there are still three available session numbers in the 'even' interval. This allows the sessions in the 'odd' interval to be loaded and saved with an 'even' interval session number and with no session in the 'odd' interval so that a new 'odd' interval identifier can be created.

Session contexts in TPM RAM are flushed on any TPM2_Startup(). Saved session contexts are not invalidated and may be reloaded after a STARTUP_SAVE. Saved session contexts are invalidated on ST_CLEAR.

30.6 Eviction

Eviction is the process of removing the context associated with an object or session from TPM RAM to allow for other sessions or objects to be loaded or created. Saving a session context removes the majority of the session context from TPM RAM. Saving an object context does not remove it from TPM

memory. When applied to an object, TPM2_FlushContext() will remove it from the TPM RAM but not invalidate the saved contexts of that object. When applied to a session, TPM2_FlushContext() will invalidate the session whether its context is in TPM RAM or saved.

An object may be made persistent in TPM NV memory with TPM2_EvictControl(). When made persistent, TPM2_FlushContext() and ST_CLEAR have no effect on the object.

A session may not be made persistent.

Use of TPM2_EvictControl() requires either Owner Authorization or Platform Authorization. An object made persistent using *ownerAuth* may be made volatile using either Owner Authorization or Platform Authorization. An object made persistent using Platform Authorization may only be made volatile using Platform Authorization.

30.7 Incidental Use of Object Slots

In most cases, the TRM will explicitly load and unload (flush) objects from the TPM's object memory. In three cases, the TPM will make use of object slots as a side effect and the TRM needs to deal with potential resource issues that may arise. The three cases are: TPM2_Import(), use of persistent objects, and _TPM_Hash_Start.

TPM2_Import() allows an implementation to use an object slot for its "scratch" memory while operating on the import blob. When the command completes the slot will be available. An implementation that uses this option may return TPM_RC_OBJECT_MEMORY if a needed slot is not available. This return code is in the group of response codes that are expected to be handled by the resource manager.

When a handle references a persistent object, a TPM implementation is allowed to return TPM_RC_OBJECT_MEMORY if an object slot is not available. This allows the TPM to keep the persistent image of the object in a compressed form and decompress it into an object slot for efficient processing. The version of the persistent object held in an object slot will be removed when the command completes.

When the TPM receives _TPM_Hash_Start, it will unconditionally create an Event Sequence context. If an object slot is available, the TPM will use the available slot. If an object slot is not available, the TPM will flush an arbitrary object context and use that slot. At the end of the event sequence (_TPM_Hash_End), the slot used for the Event Sequence will be vacant. The TRM should be aware that the _TPM_Hash_Start sequence may cause loss of a loaded object.

31 Attestation

31.1 Introduction

Attestation is the action of having the TPM sign some internal TPM data. Confidence in the attestation is related to the confidence in the key that is used to sign. The highest confidence is provided by a *fixedTPM*, restricted signing key that is created on a TPM with a certificate from the TPM manufacturer.

The TPM may be used to attest to several different types of data:

- PCR data – TPM2_Quote()
- *Clock* and *Time* data – TPM2_GetTime()
- Audit digests – TPM2_GetCommmandAuditDigest() and TPM2_GetSessionAuditDigest()
- Other TPM Objects – TPM2_Certify()

For all of these commands, the TPM produces a standard attestation structure and appends the command-specific data. The resulting data block is then hashed and signed by the selected signing key. The selected key may be any key that has the *sign* attribute SET. If the signing key is unrestricted, then the caller may indicate the signing scheme to be used. If the signing key is restricted, the TPM will return an error (TPM_RC_SCHEME) unless the scheme selector in the attestation command is TPM_ALG_NULL.

31.2 Standard Attestation Structure

The contents of the standard attestation structure are specified in Table 29.

Table 29 — Standard Attestation Structure

Parameter	Type	Description
magic	TPM_GENERATED	This unique value (TPM_GENERATED_VALUE) occurs as the first octets in any TPM-generated attestation structure. This field is used to prevent use of a restricted signing key to sign a forgery of an attestation. A TPM will not allow a restricted signing key to sign any external data if that data starts with this unique value. The way that the TPM enforces this restriction is that a TPM will not use a restricted key to sign a digest that the TPM did not produce. Since the TPM produced the digest, it can ensure that any external data did not start with this value.
type	TPMI_ST_ATTEST	This identifies the type of the attestation structure and indicates the contents of the *attested* parameter.
qualifiedSigner	TPM2B_NAME	This is the Qualified Name of the key used to sign the attestation data. A key that can be duplicated may be signing in different locations and this Qualified Name allows the Verifier to determine the environment in which the signature was produced.
extraData	TPM2B_DATA	external info supplied by caller (often in *qualifyingData* parameter)
clockInfo	TPMS_CLOCK_INFO	The values of *Clock*, *resetCount*, *restartCount*, and *Safe*
firmwareVersion	UINT64	This TPM-vendor-defined value changes when the firmware on the TPM changes, if that change is meaningful to the security of the TPM.
[type]attested	TPMU_ATTEST	the type-specific attestation information
NOTE	A TPM2B_DATA structure provides room for a digest and a method indicator to indicate the components of the digest. The definition of this method indicator is outside the scope of ISO/IEC	

11889.

31.3 Privacy

The attestation block contains information that could allow cross correlation of attestation values. The combination of a *firmwareVersion* and *clockInfo* could be used to identify that two attestations were signed by keys on the same TPM. This correlation is possible because the combination of *resetCount*, *restartCount*, and *firmwareVersion* could be unique. Even if the combination is not unique for all TPM, an imperfect correlation may be adequate for certain types of activity tracking.

The TPM prevents such tracking by adding obfuscation values to the reported values of *resetCount*, *restartCount*, and *firmwareVersion*. This obfuscation value is different for each key and TPM. Although the values are obfuscated, they do not lose any of their usefulness for indicating changes to the values. While the absolute values are not visible in the attestation, it is still possible to look at attestations signed by the same key and determine how many times the TPM was reset or restarted between the attestations and to see the delta in the firmware version number (if any).

It is sometimes necessary to have the non-obfuscated values of the *clockInfo* and *firmwareVersion* included in an attestation. Support for this is provided by allowing signing keys in the Endorsement hierarchy. When a key in the Endorsement hierarchy signs an attestation, no obfuscation is applied. The underlying presumption is that the TPM's Privacy Administrator controls the Endorsement hierarchy and it is possible, through policy, to limit the use of keys in that hierarchy so that authorization from the Privacy Administrator is always required.

31.4 Qualifying Data

Each of the attestation commands has a parameter called *qualifyingData*. This parameter is not interpreted by the TPM and may contain any data chosen by the caller. The most common use of this parameter is expected to be as a nonce to ensure "freshness" of an attestation.

31.5 Anonymous Signing

If an anonymous scheme (TPM_ALG_ECDAA) is used for signing in any attestation command, the *qualifiedSigner* parameter will be an Empty Buffer.

NOTE 1 If the *qualifiedSigner* field was properly populated (not the Empty Buffer), then the unique identity of the signing key would be disclosed.

For TPM2_Certify() using an anonymous signing scheme, both the *qualifiedSigner* and *qualifiedName* of the certified key are set to an Empty Buffer.

NOTE 2 If the *qualifiedName* field was not cleared, then it would be possible to establish a hierarchical relationship between to certified objects. This is not desirable for an anonymous scheme.

32 Cryptographic Support Functions

32.1 Introduction

In ISO/IEC 11889 (first edition), the cryptographic primitives were not exposed for general purpose use. ISO/IEC 11889 provides commands that allow access to the primitive cryptographic processes of the TPM.

EXAMPLE 1 An example of how in ISO/IEC 11889 (first edition), the cryptographic primitives were not exposed for general purpose use is the RSA engine could not be used for exponentiation.

One assumption in ISO/IEC 11889 (first edition) was that the host processer usually had much greater performance than the processor used for the TPM so there was no point in having the TPM do something that the host could do much faster. In addition, ISO/IEC 11889 (first edition) was a passive device with limited bandwidth. While it is true that the host processor will usually have more capability than the TPM, this will not be true in all cases. In fact, on some systems, the main processor will be able to switch execution environments and perform the TPM operations. In others, the TPM may be built around a cryptographic coprocessor that has significantly greater processing capability for cryptographic operations than the host. These higher performance implementations will not be performance-limited by being attached to the system with a low-bandwidth interface. These performance differences mean that exposure of the cryptographic primitives of ISO/IEC 11889 makes more sense that it did in ISO/IEC 11889 (first edition).

Another reason to make the cryptographic primitives available is that not all software will implement all the algorithms that may be in the TPM.

EXAMPLE 2 A BIOS might not implement the RSA algorithm but would want to check the RSA signature of some code.

This clause describes the commands and methods that may be provided by a TPM compliant with ISO/IEC 11889.

32.2 Hash

TPM2_Hash() will create a digest of a block of data using the indicated hash algorithm. If the amount of data to be hashed exceeds that input buffer size of the TPM, then a hash sequence is used (see 32.4).

If the data used to create the digest does not have TPM_GENERATED_VALUE as its first octets, then the response to TPM2_Hash() will contain a ticket indicating that the digest may be signed with a restricted signing key.

NOTE The creation of the ticket could be suppressed by using TPM_RH_NONE as the hierarchy parameter in TPM2_Hash().

32.3 HMAC

TPM2_HMAC() will compute an HMAC over a block of data using a TPM-resident value for the HMAC key. In this command, the handle parameter is required to reference an object with a *type* of TPM_ALG_KEYEDHASH with the *sign* attribute SET.

32.4 Hash, HMAC, and Event Sequences

32.4.1 Introduction

When the amount of data to be included in a digest cannot or will not be sent to the TPM in one of the atomic hash/HMAC commands (TPM2_Hash(), or TPM2_HMAC()) then a sequence of commands may be used to provide incremental updates to the digest.

A sequence is started with either TPM2_HashSequenceStart() or TPM2_HMAC_Start(); increments of data are added to the sequence digest(s) using TPM2_SequenceUpdate(); and TPM2_SequenceComplete() or TPM2_EventSequenceComplete() is used to complete a sequence. TPM2_SequenceComplete() and TPM2_EventSequenceComplete() may also provide the last data to be included in the sequence digest(s).

Three types of sequences are defined:

1) hash
2) Event
3) HMAC

32.4.2 Hash Sequence

In a hash sequence, the TPM will perform a hash over all the data in the sequence using the selected algorithm.

TPM2_SequenceComplete() completes the hash sequence and returns a digest of the data. Additionally, if the data used to create the digest did not start with TPM_GENERATED_VALUE, then a ticket is produced indicating that the digest may be signed with a restricted key.

A hash sequence is:

a) TPM2_HashSequenceStart() (*hashAlg* is a supported hash algorithm), followed by
b) TPM2_SequenceUpdate() (zero or more), followed by
c) TPM2_SequenceComplete()

32.4.3 Event Sequence

For an Event Sequence, the TPM will potentially create multiple digests over the data (a digest for each PCR bank). TPM2_EventSequenceComplete() is used to complete the sequence and return a list of digests; and, if a PCR handle is provided, each digest is extended into the corresponding PCR bank.

EXAMPLE If a TPM implements both a SHA1 and a SHA256 bank, then the list will contain two digests.

An Event Sequence is:

a) TPM2_HashSequenceStart() (*hashAlg* is TPM_ALG_NULL), followed by
b) TPM2_SequenceUpdate() (zero or more) followed by
c) TPM2_EventSequenceComplete() (will do an Extend if *pcrHandle* is a PCR and not TPM_RH_NULL)

32.4.4 HMAC Sequence

For an HMAC sequence, the TPM will use the indicated key as the HMAC key and perform an HMAC computation over the data of the sequence using the specified hash algorithm.

TPM2_SequenceComplete() completes the HMAC sequence and returns the HMAC value.

NOTE The response for TPM2_SequenceComplete() also has a *validation* parameter. This parameter is used for a hash sequence to indicate if the digest is safe to sign with a restricted key. This parameter is not used for an HMAC sequence so the TPM will set the *validation* parameter to a NULL Ticket

An HMAC sequence is:

a) TPM2_HMAC_Start() (*hashAlg* is a supported hash algorithm), followed by

b) TPM2_SequenceUpdate() (zero or more) followed by

c) TPM2_SequenceComplete()

32.4.5 Sequence Contexts

Sequences involve hashing of data and the intermediate hash state must be retained by the TPM in a protected location. This intermediate state is kept in a vendor-specific structure that may occupy an object slot on the TPM.

A sequence context is assigned a handle so that it may be saved and restored like any transient object. Its properties are not identical to a transient object because the sequence context is updated on each use. In addition, unlike transient objects, the public portion of a sequence is not readable with TPM2_ReadPublic(). A sequence context can be replayed if one has the authorization for the sequence.

If an authorization or audit for a sequence object requires computation of a *cpHash* and an *rpHash*, the Name associated with *sequenceHandle* will be the Empty Buffer.

When TPM2_EventSequenceComplete() or TPM2_SequenceComplete() completes successfully, the sequence context is flushed from the TPM.

A sequence is exempt from dictionary attack protection and authorization failures will not cause the TPM to enter lockout.

32.5 Symmetric Encryption

TPM2_EncryptDecrypt() is defined for symmetric encryption and decryption of blocks of data. Support for this command in a TPM may cause the TPM to be subject to different jurisdictions' legal import/export controls than would apply to a TPM without these commands.

The command supports chaining of encryption so that the encryption/decryption may be done incrementally as the data arrives or to handle the cases where the block of data is larger than will fit into a single TPM buffer.

32.6 Asymmetric Encryption and Signature Operations

The annexes to this part of ISO/IEC 11889 contain descriptions of the cryptographic encryption/decryption and signature primitives that are defined for each of the asymmetric algorithms supported by ISO/IEC 11889.

33 Locality

In some systems, accesses to the TPM are segregated by privilege level. The interface to the TPM may be able to discriminate the different privilege levels and provide an indication to the TPM when the access is at a privilege level other than the default level.

The indication of privilege level can be used in access control policy to ensure that the operation on an object is occurring at the right level. The privilege level of a command is called its Locality.

The method by which the TPM interface determines the Locality of an access is system-dependent. The TPM interface provides a Locality indication to the TPM each time the TPM is accessed. The contents of the command or response buffer are not changed by the Locality indication.

The definition of the modifier is platform-specific. Depending on the platform, the modifier could be a special bus cycle or additional input pins on the TPM.

EXAMPLE One example would be special cycles on the Low Pin Count (LPC) bus that inform the TPM it is under the control of a process on the PC platform. The assumption is that spoofing the modifier to the TPM requires more than just a simple hardware attack, and would require expertise and possibly special hardware.

The locality value is represented as a byte and locality values have two separate representations. Localities 0 through 4 are represented as bits in the byte with $0000\ 0001_2$ representing locality 0 and $0001\ 0000_2$ representing locality 4. This representation allows multiple localities to be represented in a single byte as long as the localities are in the range of 0-4. This representation of locality is compatible with ISO/IEC 11889 (first edition)

A second representation is for localities above 4. These are called *extended localities*. For extended localities, the locality byte is an integer value representing the locality. Because of the format for localities 0-4, the first extended locality is 32. The range of extended localities is 32-255. The locality value may indicate only one extended locality at a time.

NOTE Locality 5 through 31 cannot be selected.

34 Hardware Core Root of Trust Measurement (H-CRTM) Event Sequence

34.1 Introduction

A process that puts the system in a known state running known code creates the starting point for a chain of trust. A computer system reset puts the processor and chipset into a known state, and the processor (the root of trust for measurement) begins executing code provided by the platform manufacturer. This initial code is the core root of trust for measurement (CRTM). It is code that must be trusted as there is no way to tell what that code is other than to rely on the manufacturer. Usually, one of the actions of the CRTM is to extend a PCR with a value that represents the identity of the CRTM. This boot process starts the chain of trust with two different roots that are usually from different sources: the RTM from a CPU vendor and a CRTM from a platform manufacture.

Some system implementations support an alternative method of starting a chain of trust that makes the CPU the CRTM. For this method, the CPU is placed in a known state and measures the code that it will run. Before being measured, this code is protected so that it cannot be tampered with and there is assurance that the code that is measured is the code that is executed. Since the CPU is both executing the measured code and measuring it, it is both the RTM and the CRTM. This is called a hardware-based core root of trust for measurement or H-CRTM.

The TPM supports an H-CRTM by providing special interface indications that allow the TPM to determine when it is receiving data from the RTM acting as CRTM. These indications are:

- **_TPM_Hash_Start** – sent by the RTM to indicate the start of a H-CRTM Event Sequence. The TPM will initialize an H-CRTM Event Sequence context. The H-CRTM Event Sequence context contains hash state for each bank of PCR. This indication is only allowed from the RTM when it has been put into a known "good" state as defined by the RTM manufacturer. There is only one _TPM_Hash_Start per H-CRTM Event Sequence.
- **_TPM_Hash_Data** – sent by the RTM to update the digests in the H-CRTM Event Sequence contexts with H-CRTM data. An H-CRTM Event Sequence may have zero or more _TPM_Hash_Data indications.
- **_TPM_Hash_End** – sent by the RTM to indicate the end of the H-CRTM Event Sequence. On receipt of this indication, the TPM will take actions that are dependent on whether the H-CRTM occurred before or after TPM2_Startup(). The actions taken as the result of this indication will always include initialization of at least one PCR followed by a PCR being extended with the H-CRTM data.

During an H-CRTM sequence, if any indication other the _TPM_Hash_Data occurs between the _TPM_Hash_Start and _TPM_Hash_End indications (including receipt of a command), then the H-CRTM Event Sequence is abandoned, the H-CRTM Event Sequence context is flushed, and no change to any PCR occurs.

34.2 Dynamic Root of Trust Measurement

When an H-CRTM occurs after TPM2_Startup() it is called the dynamic root of trust for measurement (D-RTM).

NOTE 1 There is no special designation for when the H-CRTM occurs before TPM2_Startup()

NOTE 2 The D-RTM sequence could be repeated one or more times after TPM2_Startup. On each invocation of the D-RTM sequence, the RTM will be in the same known state.

For D-RTM, the TPM will initialize one or more PCR to zero and then extend PCR[17] in each bank with the H-CRTM data accumulated in the H-CRTM Event Sequence.

$$\text{PCR}[17][hashAlg] := \mathbf{H}_{hashAlg}(0\ldots0 \,||\, \mathbf{H}_{hashAlg}(hash\_data)) \quad (65)$$

Where

hash_data	all the octets of data received in _TPM_Hash_Data indications

The PCR that are initialized and extended as a result of a D-RTM event are specified in a platform-specific TPM specification.

34.3 H-CRTM before TPM2_Startup()

If the H-CRTM sequence occurs before TPM2_Startup(), then only PCR[0] will be affected. When _TPM_Hash_End is received, the TPM will complete the Event Sequence digests. It will then initialize PCR[0] to 4 and Extend the H-CRTM Event Sequence data. The value 0…4 represents evidence that the initial measurement was from an H-CRTM.

$$\text{PCR}[0][hashAlg] := \mathbf{H}_{hashAlg(}(0\ldots04 \,||\, \mathbf{H}_{hashAlg}(hash\_data)) \quad (66)$$

where

0…04	denotes a numeric value of 4 with high-order bits of 0 to make the value the size of a digest computed with *hashAlg*
hash_data	all the octets of data received in _TPM_Hash_Data indications

If PCR[0] is initialized by an H-CRTM event before TPM2_Startup(), then TPM2_Startup(CLEAR) will not change the value of PCR[0]. Otherwise, TPM2_Startup(CLEAR) will ensure that PCR[0] is zero.

If the previous TPM Reset or TPM Restart used H-CRTM, then each TPM Resume will need to use H-CRTM. If the previous TPM Reset or TPM Restart did not use H-CRTM, then each TPM Resume should not use H-CRTM.

35 Command Audit

The command audit mechanism allows the TPM owner to create a verifiable log of each execution of selected commands.

TPM2_SetCommandCodeAuditStatus() is used either to change the list of commands being audited or to change the audit hash algorithm (it cannot change both in the same command). This command requires either Platform Authorization or Owner Authorization. The selection may change at any time.

NOTE 1 It is anticipated that a small number of commands will be selected for audit, most likely those commands that provide identities and control of the TPM. However, there are few restrictions on which commands can be audited.

The audit log, the list of executed TPM commands and responses, is maintained outside the TPM by an untrusted party. Enabling the audit function of a TPM does not guarantee that the log will be properly maintained. The TPM audit function simply provides a means to determine if the log was properly maintained.

It is not necessary to continuously maintain the audit log in order to use the audit capability. When an audit log is started, the current contents of the audit digest register can be read to establish the starting value for the log. At the end of the audit interval, the audit digest register can be read again and the contents of the audit log over the audit interval can be verified.

An audit can be used to track use of keys and, therefore, is potentially privacy sensitive. For this reason, the privacy administrator of the TPM must authorize access to the audit digest register. Authorization from the privacy administrator is expressed using Endorsement Authorization.

The update of the audit digest register occurs when the command completes successfully and the response has been created. The command audit update is:

$$audit_{new} := \mathbf{H}_{auditAlg}(audit_{old} \,||\, cpHash \,||\, rpHash) \tag{67}$$

where

$\mathbf{H}_{auditAlg}$	hash function using the currently selected audit hash algorithm
$audit_{old}$	the previously computed audit digest
$cpHash$	the command parameter hash using the audit hash
$rpHash$	the response parameter hash using the audit hash

NOTE 2 Clause 18.7 describes the process for computing *cpHash* and clause 18.8 describes the process for computing the *rpHash*.

The audit mechanism uses two components: an audit digest register and an audit counter. The audit counter is a non-volatile register that counts the number of audit logs that are created. If the audit digest register contains all octets of zero when an audit event is recorded, then a new audit log is being created and the audit counter is incremented.

An audit log ends and the audit digest is cleared when the command TPM2_GetCommandAuditDigest() returns a signature.

NOTE 3 The audit counter is incremented when the new log starts so that a missing log cannot be dismissed as being irrelevant. Because a new audit log is started only when an auditable event occurs, any missing log is suspect.

The audit counter is non-volatile and is reset to zero by TPM2_Clear(). The audit digest register is reset when an unanticipated power event occurs (that is, loss of TPM power without an orderly shutdown). The audit digest is preserved over any orderly shutdown.

The audit digest register is reset by any TPM2_SetCommandCodeAuditStatus() that has an *auditAlg* value other than TPM_ALG_NULL.

An audit report structure contains the current value of the audit digest register and the value of the audit counter.

NOTE 4 The signed audit structure is a TPMB_ATTEST structure that contains other qualifying information about the signing environment.

Because the audit mechanism utilizes NV memory, endurance may be a factor. The endurance requirements of the audit mechanism are platform-specific.

NOTE 5 The command audit session counter is incremented on the first auditable command in a session. This would be infrequent so the endurance of the counter is not likely to be a major issue.

When the TPM is in Failure mode, command audit is not functional and command audit of TPM2_GetTestResult() and TPM2_GetCapability() will not occur.

TPM2_SetCommandAuditStatus() is audited when it changes the list of audited commands. It is not possible to disable audit of this command. If TPM2_SetCommandAuditStatus() is used to change the audit hash algorithm, then the command is not audited and evidence of this operation is provided by the change in the hash algorithm reported when the command audit value is read.

36 Timing Components

36.1 Introduction

The TPM has timing components for use in time-stamping of attestations and for gating policy. *Clock* advances while the TPM is powered. Additionally, *Clock* may be advanced in order to bring it into alignment with real time. However, *Clock* may not be set back except by installing a new owner.

The *resetCount* and *restartCount* values allow detection of power loss that could cause discontinuities in the time recorded by *Clock*. A timer (*Time*) runs when the TPM is powered and is reset by any TPM2_Startup(). The *Safe* flag indicates that the values read from the timing components are known not to be replay values.

The timing components are exposed through commands that:

- read the value of *Clock*, *Time*, *resetCount*, and *restartCount* (TPM2_GetTime());
- time-stamp externally provided data using a signature key and *Clock*, *resetCount*, and *restartCount* (TPM2_GetTime(), TPM2_Quote(), TPM2_Certify(), and other restricted signing operations);

  NOTE TPM2_ReadClock() returns uncertified (not signed) values. TPM2_GetTime() returns a structure and an optional signature over the data. TPM2_ReadClock() is used by the OS to manage the timing resources of the TPM and TPM2_GetTime() is for attestation of time and is under control of the privacy administrator.

- allow *Clock* to be adjusted forward (TPM2_SetClock());
- allow the rate of advance of *Clock* to be adjusted (TPM2_ClockRateAdjust()); and
- allow objects to be lifetime-limited using authorization policy expressions that reference *Safe*, *Clock*, *Time*, *resetCount*, and *restartCount* (TPM2_PolicyCounterTimer()).

Potential use cases for the TPM timing components include:

- lifetime limits for keys when certificate revocation is impossible or undesirable;
- time-limited delegation of rights;
- time-stamping of security event logs to ensure that events cannot be forged in the past;
- boot-counter stamping of event logs to ensure that a log associated with a particular reboot cannot be deleted without leaving a trace;
- boot-counter/PCR-counter stamping of keys to indicate they were created during OS installation;
- time-stamping of attestation values as an alternative to the use of a nonce in online protocols; and
- indication of whether a TPM/platform has rebooted since last checked.

EXAMPLE An example of ime-limited delegation of rights is the right to use or duplicate a key for 1 hour.

Clock is *not* designed to be a replacement for other online or local time sources and is not appropriate for all uses. Later clauses describe the behavior of timing resources and their specific security properties. Implementers and relying parties should understand the limitations before using these features.

36.2 Clock

36.2.1 Introduction

Clock is a time value that can be advanced but never rolled back. It may increment in volatile memory. If so, it is periodically written to NV memory.

A non-orderly shutdown may cause a write to NV memory to be missed. Other values that are written to NV on an orderly shutdown will be advanced to a known safe value on the next startup. However, Clock is not advanced because power outages would cause the clock to be advanced to a time in the future and it could not be adjusted back to an accurate value. To indicate that a value reported in Clock may be a repeat of a previously reported value, a flag (*safe*) is CLEAR after a non-orderly shutdown. After the next NV update of *Clock,* safe is SET to indicate that *Clock* is not a repeat.

Clock is a volatile value that increments each millisecond that the TPM is powered. A non-volatile value (*NV Clock*) is updated periodically from *Clock*. *NV Clock* will always move forward as *Clock* advances. However, because of unexpected power loss, it is possible that the same value of *Clock* will be reported more than once. The mitigations for this are specified in subsequent parts of clause 36.2.

The accuracy of *Clock* is approximate. The causes of inaccuracy are

- battery backup for *Clock* is not required,
- the TPM's time reference may not be accurate, and
- the TPM must rely on external software to provide initial or periodic adjustments to *Clock* settings.

The interpretation of the time-origin (t=0) is out of the scope of ISO/IEC 11889, although Coordinated Universal Time (UTC) is expected to be a common convention.

The value of *Clock* may be set forward by external software (TPM2_ClockSet()) to compensate for power interruptions or clock slew, but, except for changes in ownership (TPM2_Clear()), the TPM will not allow external software to set *Clock* backward.

The value of *Clock* may be advanced by TPM2_ClockSet() using either platform or owner authorization.

NOTE The value of *Clock* cannot be advanced beyond FF FF 00 00 00 00 00 00_{16}. This restriction prevents any possibility of *Clock* rolling over during its lifetime and simplifies use of *Clock* in policies.

The TPM may be driven by an imprecise internal or external frequency source. To compensate, the TPM allows external software with a more reliable time source to make limited (+/-15%) adjustments to the rate of advancement of *Clock*.

36.2.2 *Clock* Implementation

The technology used for non-volatile storage may make the update rate for *NV Clock* an endurance issue. To mitigate this, the interval between updates of *NV Clock* from *Clock* are allowed to be as long as once per 2^{22} milliseconds.

NOTE If *NV Clock* is implemented in a technology that allows millisecond updates and has no endurance issues, then *Clock* and *NV Clock* could be the same.

Since *NV Clock* may be updated at a low rate, a power event may cause the value in *Clock* to appear to go backward.

EXAMPLE Assume that the update interval for *NV Clock* is the maximum allowed value (2^{22} milliseconds or approximately 70 minutes). Power might be removed from the TPM just before an update of *NV Clock*. Then, when power is restored, *Clock* will be restored from *NV Clock* and *Clock* may have a value that is more than an hour older than the last reported value of *Clock*. This illustrates that the values of *Clock* reported by the TPM for the first hour of operation can have a lower value than values returned before the power outage.

The *Safe* flag in the TPMS_TIME_INFO structure is used to indicate if the reported value of *Clock* is guaranteed not to be a repeat of a previously reported value. The *Safe* flag is specified in more detail in the following clause.

36.2.3 Orderly Shutdown of *Clock*

In order to reduce the amount of time that must pass before *Safe* is SET, the TPM supports an orderly shutdown. TPM2_Shutdown() is used to indicate to the TPM that software anticipates the loss of TPM power and that the appropriate state should be preserved. When the TPM receives TPM2_Shutdown(), it will copy all of the bits of *Clock* to *NV Clock.* After an orderly shutdown, the TPM will SET a non-volatile flag to indicate that an orderly shutdown has occurred.

NOTE 1 To allow the *NV Clock* to only have to record the upper bits of *Clock*, an alternate implementation is to keep *Clock* in memory that has a copy saved on an orderly shutdown and to restore *Clock* from that memory on the next power up.

After an orderly shutdown, *Clock* continues to count and *NV Clock* will be updated at the normal rate.

Any time a command is executed that uses the value of *Clock*, the flag indicating orderly shutdown will be CLEAR even if this command occurs subsequent to TPM2_Shutdown(). This flag may be SET when *NV Clock* is updated from *Clock*.

NOTE 2 It is possible for the TPM to perform multiple shutdowns before TPM power is actually lost.

If *Safe* is not SET when TPM2_Shutdown() is received, then *NV Clock* must not be set from *Clock* and *Safe* must not be SET on the subsequent startup.

It is permitted for the low-order 10 bits of *Clock* to come from *Time* and for *NV Clock* not to implement those bits. That is, *NV Clock* does not maintain resolution to better than 2^{10} milliseconds. If an implementation uses this option, then *Safe* will be CLEAR at least for the first 2^{10} milliseconds of TPM operation.

36.2.4 *Clock* Initialization at TPM2_Startup()

On any TPM2_Startup() or _TPM_Init (vendor's choice), *Clock* is loaded from *NV Clock* and *Clock* begins incrementing at a one millisecond rate. *NV Clock* is then updated, no less frequently than the update interval. It is anticipated that the first update of *NV Clock* will occur when some number of low-order bits of the volatile *Clock* become zero, indicating the passage of the update interval.

EXAMPLE Assuming that the NV CLock update interval is 2^{12} (approximately every 4 seconds), the TPM can perform an update of *NV Clock* whenever the low-order 12 bits of volatile *Clock* are zero.

NOTE 1 If the TPM had an orderly shutdown, the low-order bits of the *NV Clock* will likely not be zero, so the first update of *NV Clock* after the _TPM_Init will occur in less than the normal update interval.

NOTE 2 If the TPM received TPM2_Shutdown() and a subsequent command that used *Clock*, then the NV value of *Clock* will likely be non-zero, but *Safe* will be CLEAR.

