TRUSTED® COMPUTING GROUP

SPECIFICATION

Trusted Platform Module 2.0 Library Part 1: Architecture

Version 185
March 12, 2026

Contact: admin@trustedcomputinggroup.org

TCG Published

DISCLAIMERS, NOTICES, AND LICENSE TERMS

Copyright Licenses

Trusted Computing Group (TCG) grants to the user of the source code in this specification (the “Source Code”) a worldwide, irrevocable, nonexclusive, royalty free, copyright license to reproduce, create derivative works, distribute, display and perform the Source Code and derivative works thereof, and to grant others the rights granted herein.

The TCG grants to the user of the other parts of the specification (other than the Source Code) the rights to reproduce, distribute, display, and perform the specification solely for the purpose of developing products based on such documents.

Source Code Distribution Conditions

Redistributions of Source Code must retain the above copyright licenses, this list of conditions and the following disclaimers.

Redistributions in binary form must reproduce the above copyright licenses, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.

Disclaimers

THE COPYRIGHT LICENSES SET FORTH ABOVE DO NOT REPRESENT ANY FORM OF LICENSE OR WAIVER, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, WITH RESPECT TO PATENT RIGHTS HELD BY TCG MEMBERS (OR OTHER THIRD PARTIES) THAT MAY BE NECESSARY TO IMPLEMENT THIS SPECIFICATION OR OTHERWISE. Contact TCG Administration (admin@trustedcomputinggroup.org) for information on specification licensing rights available through TCG membership agreements.

THIS SPECIFICATION IS PROVIDED “AS IS” WITH NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, ACCURACY, COMPLETENESS, OR NONINFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

Without limitation, TCG and its members and licensors disclaim all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and TCG disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

Any marks and brands contained herein are the property of their respective owners.

DOCUMENT STYLE

Key Words

The key words “MUST,” “MUST NOT,” “REQUIRED,” “SHALL,” “SHALL NOT,” “SHOULD,” “SHOULD NOT,” “RECOMMENDED,” “MAY,” and “OPTIONAL” in this document’s normative statements are to be interpreted as described in RFC 2119: Key words for use in RFCs to Indicate Requirement Levels.

Statement Type

Please note an important distinction between different sections of text throughout this document. There are two distinctive kinds of text: *informative comments* and *normative statements*. Because most of the text in this specification will be of the kind *normative statements*, the authors have informally defined it as the default and, as such, have specifically called out text of the kind *informative comment*. They have done this by flagging the beginning and end of each informative comment and highlighting its text in gray. This means that unless text is specifically marked as of the kind *informative comment*, it can be considered a *normative statement*.

Example:

Reach out to admin@trustedcomputinggroup.org with any questions about this document.

Contents

List of Figures

List of Tables

1 Scope

This specification 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 this specification 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 this specification. Those values would be specified by a platform specific specification or alternatively determined by an implementer.

2 Specification Organization

The TPM Library is organized into the following parts:

2.1 Part 0: Introduction

Part 0 contains an informative description of the context and purpose of the design of various aspects of TPM 2.0.

2.2 Part 1: Architecture

Part 1 contains a narrative description of the properties, functions, and methods of a TPM. Unless otherwise noted, this narrative description is informative. Part 1 contains descriptions of some of the data manipulation routines that are used by this specification.

2.3 Part 2: Structures

Part 2 contains a normative description of the constants, data types, structures, and unions for the TPM interface. Unless otherwise noted, all tables and C code in Part 2 are normative.

2.4 Part 3: Commands

Part 3 contains a normative description of commands, with tables describing the command and response formats.

2.5 Reference Code

The Reference Code can be found at https://github.com/trustedcomputinggroup/tpm. Prior to version 184, most of the Reference Code was published in Part 4: Supporting Routines, with the rest provided inline in Part 3.

The Reference Code contains C code that describes the algorithms and methods used by the command code in Part 3. The Reference Code augments Parts 1-3 to provide a complete description of a TPM, including the supporting framework for the code that performs the command actions.

A TPM need not be implemented using the Reference Code. However, any implementation should provide equivalent or, in most cases, identical results as observed at the TPM interface or demonstrated through evaluation.

Note:

This specification 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 the Reference Code 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 Reference Code will be necessary in order to meet the needs of applicable evaluation regimes.

3 Terms and definitions

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

“ATH”

sequence of four octets of data containing 41 54 48 00_{16} that is used as a label in a KDF

> ***Note:***
>
> See Clause 8.4.10 for justification for the terminating octet of 00_{16}, for this and other labels below.

“CFB”

sequence of four octets containing 43 46 42 00_{16} that is used as a label in a KDF

“DUPLICATE”

sequence of 10 octets containing 44 55 50 4C 49 43 41 54 45 00_{16} that is used as a label in a KDF

“IDENTITY”

sequence of nine octets containing 49 44 45 4E 54 49 54 59 00_{16} that is used as a label in a KDF

“OBFUSCATE”

sequence of 10 octets containing 4F 42 46 55 53 43 41 54 45 00_{16} that is used as a label in a KDF

“SECRET”

sequence of seven octets containing 53 45 43 52 45 54 00_{16} that is used as a label in a KDF

“STORAGE”

sequence of eight octets containing 53 54 4F 52 41 47 45 00_{16} that is used as a label in a KDF

“XOR”

sequence of four octets containing 58 4F 52 00_{16} that is used as a label in a KDF

ancestor

<object loaded in a TPM> Storage Key that needs to be loaded prior to loading an object

authValue

octet string containing a value that is used for access authorization. The value is used as a password or to derive a key for an HMAC calculation.

authPolicy

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

bind entity

Entity whose *authValue* contributes to the derivation of an authorization session key.

bound

authValue of the Object is not included in the HMAC authorization for the authorization session

canonical form

data structure in the format used for transport to and from the TPM (see clause 4.39)

CLEAR

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

command

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

commandCode

numeric identifier of the operation to be performed by a TPM

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

cpHash

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

Derivation Parent

loadable key used to derive other keys; a TPM_ALG_KEYEDHASH Parent Key

descendant

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

digest

result of a hash operation

duplicate

allowing a Protected Object created by a TPM to be used on a different TPM

ECDH

Diffie-Hellman secure secret sharing process using elliptic curve operations

entity

a hierarchy, PCR, object, or NV Index in a TPM shielded location

Ephemeral Key

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

Empty Auth

Empty Buffer used as an authorization value

Empty Buffer

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

Empty Digest

Empty Buffer used as a digest

Empty Point

ECC point with Empty Buffers for both the x and y coordinates

Empty Policy

Empty Buffer used when a policy value is required;

> ***Note:***
>
> as a *policyValue*, an Empty Buffer will satisfy no policy.
>
> 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.

Endorsement Authorization

authorization using either *endorsementAuth* or *endorsementPolicy*

Extend

Extended

operation that replaces the current value of a digest with the hash of a buffer constructed by concatenating new data (normally a digest) to the current value of the digest (see Clause 8.4.8)

External Object

Object that can be loaded into a TPM without being a member of a specific hierarchy

Failure mode

mode in which the TPM returns `TPM_RC_FAILURE` in response to all commands except `TPM2_GetTestResult()` or `TPM2_GetCapability()`

Firmware Secret

secret entropy that is only available to the current TPM firmware on the current TPM. It is generated deterministically in an implementation-specific manner such that two different TPMs always have different Firmware Secrets, and a given TPM will always have the same Firmware Secret each time it runs the same TPM firmware.

Firmware SVN Secret

secret entropy that is bound to a given TPM firmware SVN running on the current TPM. It is generated deterministically in an implementation-specific manner such that two different TPMs always have different Firmware SVN Secrets. TPM firmware can obtain the Firmware SVN Secret for SVNs that are less than or equal to the current firmware's SVN, but not greater.

Firmware-limited Hierarchy

a hierarchy whose seed and proof values are derived from a Primary Seed and a Firmware Secret. The Primary Seed is associated with the firmware-limited Hierarchy's base hierarchy.

Firmware-limited Object

an Object with *firmwareLimited* SET. The object was created in a Firmware-limited Hierarchy and its parent cannot change.

import

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

internal form

data structure using a layout that is specific to an implementation that can be different from the canonical form

Key Derivation Function (KDF)

function that derives secret keying material from a secret

Key Derivation Method (KDM)

method by which secret keying material is derived from a shared secret (e.g., as part of a key establishment protocol) and other information.

Key Encapsulation Mechanism (KEM)

a type of key establishment protocol

Lockout Authorization

Authorization using either *lockoutAuth* or *lockoutPolicy*

LSB0, little-endian

the least-significant octet of a datum is at byte offset 0

MSB0, big-endian

the most-significant octet of a datum is at byte offset 0

LSb0

the least-significant bit of a datum is assigned the bit number of 0

MSb0

the most-significant bit of a datum is assigned the bit number of 0

non-volatile

data that is retained even when power is removed

NULL

context-sensitive value that, when applied to a pointer, is a system-defined value indicating that the pointer does not reference data; and, when applied to a structure identified by an algorithm identifier, is the TPM_ALG_NULL value indicating that no additional data is present

NULL Password

NULL Auth

authorization where the authorization value is the Empty Buffer, resulting in an authorization that is a sequence of 9 octets containing either 40 00 00 09 00 00 00 00 00_{16} or 40 00 00 09 00 00 01 00 00_{16}

NULL Signature

signature with the TPM_ALG_NULL signature scheme that contains no data

NULL-terminated

sequence of non-zero values followed by a value containing zero; most often a NULL-terminated string where the values are ASCII-encoded octets

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

NV Index, Index

user defined non-volatile shielded location

Object

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

Note:

An NV Index is not an object.

octet

eight bits of data

Note:

On most modern computers, this is the smallest addressable unit of data.

orderly shutdown

when the TPM has completed TPM2_Shutdown() before power to the TPM is removed or _TPM_Init is asserted

ordinary key

key produced with a seed taken from the TPM RNG

cf. Primary Key

Owner Authorization

authorization using either *ownerAuth* or *ownerPolicy*

Parent Key

any object with the *decrypt* and *restricted* attributes SET and the *sign* attribute CLEAR

Note:

There are two types of parent keys: Storage Parent and Derivation Parent.

PCR

one or more platform configuration registers each containing a digest

PCR.alg

hash algorithm associated with a specific PCR

PCR bank

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

PCR.digest

digest value associated with a specific PCR

Permanent Entity

TPM resource with an architecturally defined handle that does not change

Note:

The value of a Permanent Entity can change

Persistent Entity

TPM resource created by a Protected Capability that persists in TPM memory across power cycles and TPM resets

Platform Authorization

authorization using either *platformAuth* or *platformPolicy*

platform firmware

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

Note:

Commonly, but not exclusively, referred to as BIOS or UEFI or SMM code

policyDigest

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

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.

This structure member is permitted to be shared as follows:

- `TPM2_StartAuthSession()` can store the bindEntity Name
- `TPM2_PolicyCpHash()` can store the cpHash
- `TPM2_PolicyNameHash()` can store the nameHash
- `TPM2_PolicyDuplicationSelect()` can store the nameHash
- `TPM2_PolicyParameters()` can store pHash
- `TPM2_PolicyTemplate()` can store templateHash
- `TPM2_Policy_AC_SendSelect()` can store the nameHash

PolicyAuthorize Command

either `TPM2_PolicyAuthorize()` or `TPM2_PolicyAuthorizeNV()`

Primary Key

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

cf. Ordinary Key

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

private area

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

Primary Seed

large random value contained within a TPM from which Primary Keys and Primary Objects are derived

Protected Capability

operation performed by the TPM on data in a Shielded Location in response to a command sent to the TPM

Protected Object

object with an encrypted sensitive portion, the sensitive portion of which the TPM will only decrypt when it is in a Shielded Location

RAM

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

reset interval

period between two successive TPM Resets and the interval during which the *resetCount* is not changed

response

values returned by the TPM when it completes processing of a command

Resume PCR

platform configuration register with a value that is preserved over a TPM Resume sequence

Root of Trust

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

Note:

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.

rpHash

hash of the response code and the parameters of a response

Sealed Data Object

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

sensitive area

contains the confidential or secret parts of an object that needs to be encrypted and integrity protected when not in a Shielded Location on a TPM

sequence object

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

Note:

See Clause 27.

session

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

SET

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

Shielded Location

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

Shutdown(CLEAR)

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

Shutdown(STATE)

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

sizeof(x)

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

Startup(CLEAR)

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

Startup(STATE)

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

Storage Key

key used to provide integrity and confidentiality protection for descendant keys that are stored off of the TPM

Storage Parent

Storage Key that is acting as a parent key

SVN

Security Version Number, denoting the security posture of a given TPM firmware image. Incremented upon major security updates.

SVN-limited Hierarchy

a hierarchy whose seed and proof values are derived from a Primary Seed and a Firmware SVN Secret. The Primary Seed is associated with the SVN-limited Hierarchy's base hierarchy.

SVN-limited Object

an Object with *svnLimited SET*. The object was created in an SVN-limited Hierarchy and its parent cannot change.

Temporary Object

Objects that become unusable after a TPM Reset and that cannot be converted into Persistent Objects

temporary resource

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

TPM_GENERATED_VALUE

32-bit number (FF 54 43 47_{16}) that is used to tag structures that are generated by a TPM

TPM Reset

resetting of all TPM internal state to default values due to Startup(CLEAR)

TPM Resource Manager (TRM)

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

TPM Restart

Startup(CLEAR) that initializes all PCR but preserves most other TPM state from the previous Shutdown(STATE)

TPM Resume

Startup(STATE) that initializes some PCR but preserves most TPM state from the previous Shutdown(STATE)

transient object

object or sequence object that can be explicitly loaded and unloaded from TPM memory by the TRM; cleared from TPM memory when the TPM is initialized (`TPM2_Startup()`)

transient resource

object, sequence object, or session that can be explicitly loaded and unloaded from TPM memory by the TRM; cleared from TPM memory when the TPM is initialized (`TPM2_Startup()`)

Trusted Platform Module

TPM

implementation of this specification

user-installable software

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

volatile data

data that is lost when power is removed

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 `monospace` font.

Table 1: Symbols

A || B	concatenation of B to A
⌈ 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*
length(*x*)	denotes a function that returns the number of significant bits in an integer 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 `monospace` font indicates code written according to the C language standard.

4.2 Abbreviations

For the purposes of this document, the following abbreviations apply.

Table 2: Abbreviations

Abbreviation	Description
TPM	Prefix for an indication passed from the system interface of the TPM to a Protected Capability defined in this specification

(continued on next page)

(continued from previous page)

Abbreviation	Description
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
KVT	known value test
LPC	Low Pin Count
LSb	Least Significant bit
LSO	Least Significant Octet
ML-DSA	Module Lattice Digital Signature Algorithm
ML-KEM	Module Lattice Key Encapsulation Mechanism
MSb	Most Significant bit
MSO	Most Significant Octet

(continued on next page)

(continued from previous page)

Abbreviation	Description
NIST	National Institute of Standards and Technology
NP	new parent
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
SMAC	Symmetric block cipher Message Authentication Code
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 this specification
TSS	TCG Software Stack
UEFI	Unified Extensible Firmware Interface

5 Compliance

Unless the Part 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 Part 3. A platform-specific specification will indicate the commands from this specification that are required to be implemented in order to be compliant with that platform-specific specification.

Even though the code in the Reference Code 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 this specification.

Note:

Please report bugs in the Reference Code to the TCG (admin@trustedcomputinggroup.org).

Report vulnerabilities to security@trustedcomputinggroup.org.

The response codes in the specification are normative. An implementation performing a check prescribed by this specification 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 can vary from the Reference Code; in this case, the same error would not occur in the same situation (such as, a TPM implementation with more memory can be able to satisfy a request where the Reference Code would have returned an error). 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 Code.

The Reference Code contains major subsystems that can change for each instance of a TPM. For example, the NV subsystem of the Reference Code is not representative of the actual implementation of most physical NV implementations but is a crude analog. When the subsystem is rewritten, an equivalent interface should be provided, and the errors returned are required to match those of the Reference Code.

Note:

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 Code, then TPM tests that assume a modular design will be able to produce reliable test results on each TPM implementation.

The Reference Code 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.

6 Conventions

6.1 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 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:

Array indexing is zero-based.

Note:

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. This specification 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.

Note:

The TPM uses MSB0, LSb0 numbering.

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.

6.2 Sized Buffer References

The specification 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 the specification references the size field of the structure, the structure name is followed by “.size” (a period followed by the word “size”). When the specification references the octet array of the structure, the structure name is followed by “.buffer” (a period followed by the word “buffer”).

6.3 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 this specification.

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

Example:

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

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

Example:

$0100\ 1110\ 0001_2$

The number of digits indicates the number of bits in the representation.

Example:

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

Example:

$10\ 0000_2$ is a binary number that contains exactly 6 bits and has a decimal value of 32.

Example:

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

A number in a machine-readable table can 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:

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

6.4 Input and Output Buffer Marshaling

6.4.1 Integers

Integers (e.g., UINT16 or UINT32) are big endian. That is, the most significant byte occupies the lowest address.

6.4.2 Byte Arrays

Byte 0 of a byte array occupies the lowest address.

6.4.3 Structures

Structures are packed in the order that they are presented in the structure’s typedef. There must be no padding between structure members. Structures are packed on a byte boundary.

7 TPM Protections

7.1 Introduction

This part of the specification 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 can 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 can 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 this specification and the vendor-specific operations that meet the requirements of Clause 7.4.

Protected Capabilities comprise TPM commands as well as internal Protected Capabilities.

7.2 Internal Protected Capabilities

Internal Protected Capabilities augment the TPM’s interface commands defined in this TPM specification and may be used by vendor-specific functions in TPM implementations.

Internal Protected Capabilities must meet the requirements for vendor-specific operations in Clause 7.5.

Example:

Table 3 lists examples of information that can be passed by internal Protected Capabilities.

Table 3: Example information passed by Internal Protected Capabilities

Information	Used by	Description
Secure channel information	`TPM2_PolicyTransportSPDM()` `TPM2_GetCapability()`	Status whether a TPM command is protected by a secure channel session Requester and TPM secure channel keys that were used to establish the associated secure channel session
Failure mode information	Internal failure handling functions	Status whether TPM is in Failure mode
FIPS service indicator	Secured Message that wraps/ encapsulates a TPM command	FIPS service indicator of last executed TPM command

- Secure channel information may be passed by a TPM implementation's secure transport layer to its application layer, which then can evaluate this information in a policy assertion or store it in a Shielded Location.
- Failure mode information may be passed by a TPM implementation's application layer to its secure transport layer, which can then terminate active secure channel sessions (to prevent cryptographic operations during Failure mode). Failure mode information may also be passed by a TPM implementation's secure transport layer to its application layer (because Failure mode can be caused by cryptographic operations, which may be performed by a TPM implementation's secure transport layer).
- The FIPS service indicator for the last executed TPM command may be passed by a TPM's application layer to a TPM implementation's secure transport layer to include this information in the Secured Message response that encapsulates the corresponding TPM response (for convenience of reading the FIPS service indicator).

7.3 Protection of Protected Capabilities

A Protected Capability can 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.

7.4 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 can be shielded from access while the TPM is processing a command or generating a response.

7.5 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 is permitted to 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 can 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 this specification. The protection profile of that TPM should include a description of the protections of that data to ensure confidentiality and integrity of the data and to mitigate against rollback attacks.

8 TPM Architecture

8.1 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 1.

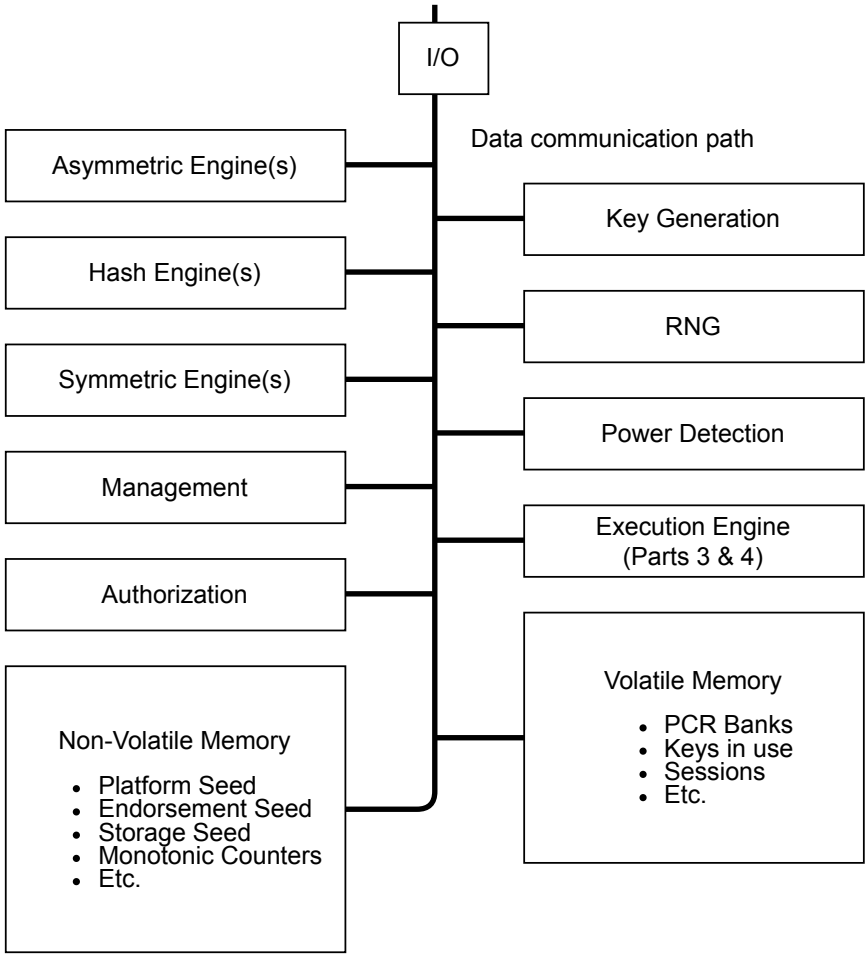


Figure 1: Architectural Overview

8.2 TPM Command Processing Overview

Figure 2 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 2 are in this clause and in clauses dedicated to those modules.

The partitioning of functions in Figure 2 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:

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 15) 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 Part 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 can 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:

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 can change TPM state.

- An authorization failure can update the dictionary attack mechanism.
- The self test mechanism has state (for example, which algorithms have been tested) that is considered to be different from the command execution state. Changes to this state can occur regardless of the command return code. For example, an implicit self test invoked to test an algorithm required by the command can 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:

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 described in Clause 15. See Clause 16 for a description of the methods for creating the values that authorize use of a TPM Shielded Location and Clause 36 for response code formatting information.

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

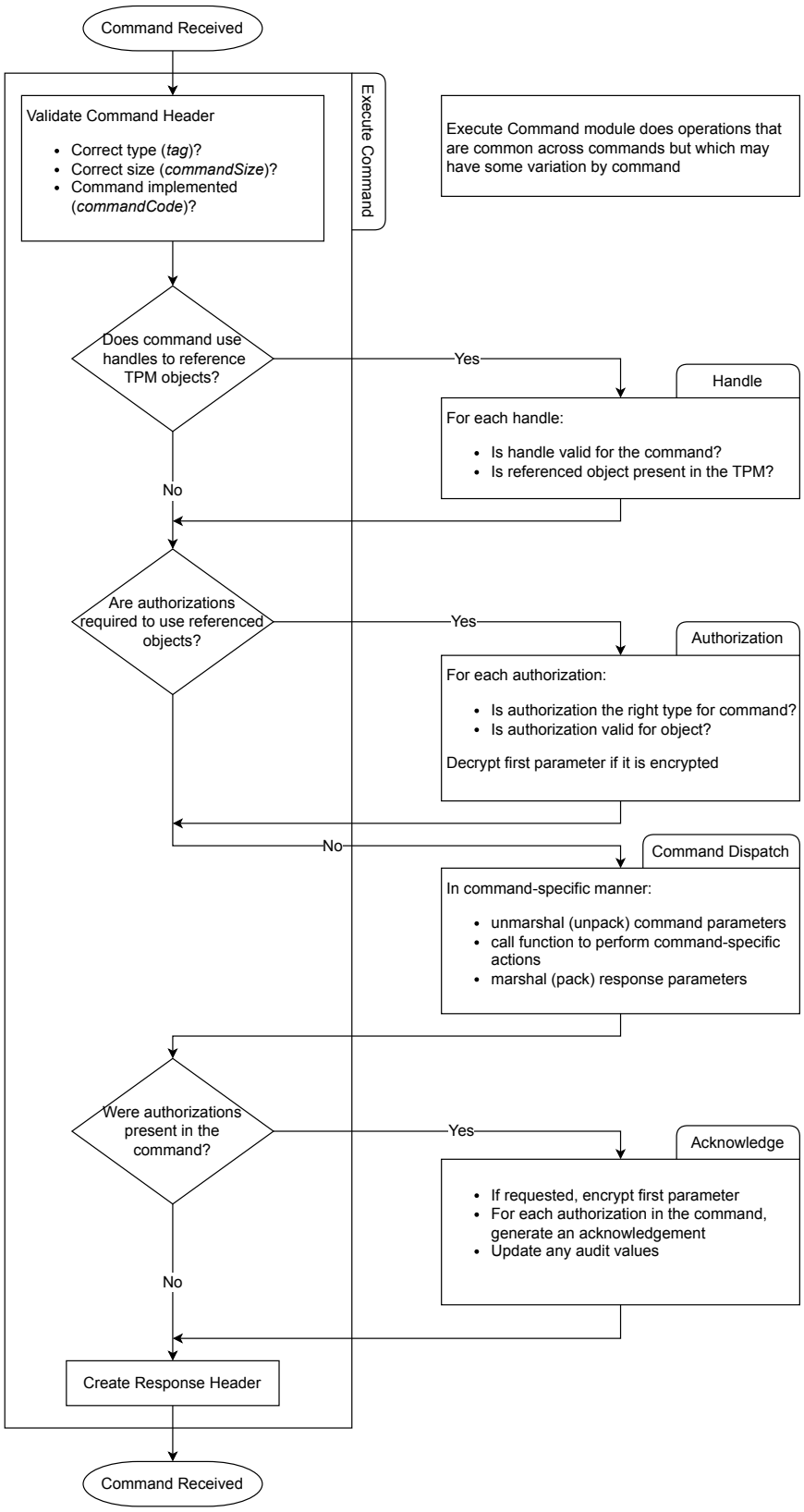


Figure 2: Command Execution Flow

8.3 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 this specification. 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. For 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 is required to be protected from modification. Before the data is modified, it is required to be in a Shielded Location.

8.4 Cryptography Subsystem

8.4.1 Introduction

The Cryptography subsystem implements the TPM's cryptographic functions. It can 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 SMAC) and signature verification, and
- key generation.

The remainder of this clause 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.

8.4.2 Symmetric Block Cipher MAC Algorithms

The TPM may implement Symmetric Block Cipher Message Authentication Code (SMAC).

An SMAC is a form of symmetric signature over some data using a symmetric block cipher algorithm. 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.

8.4.3 Hash Functions

Hash functions can 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:

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 can 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 such as PCR Extend.

8.4.4 HMAC Algorithm

The TPM implements the Hash Message Authentication Code (HMAC) algorithm specified in ISO/IEC 9797-2 [1].

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:

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

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

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

$$\mathbf{HMAC}_{hashAlg}(K,\ text)\ :=\ \mathbf{H}_{hashAlg}((\bar{K}\oplus\mathrm{OPAD})\ \|\ \mathbf{H}_{hashAlg}((\bar{K}\oplus\mathrm{IPAD})\ \|\ text)) \tag{2}$$

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

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

8.4.5 Asymmetric Operations

A TPM uses asymmetric algorithms for attestation, identification, and secret sharing. A TPM is required to implement at least one asymmetric algorithm.

8.4.5.1 General-Purpose Asymmetric Encryption and Decryption

The TPM supports several asymmetric encryption, Diffie-Hellman, and Key Encapsulation Mechanism (KEM) algorithms.

For general-purpose asymmetric encryption, Diffie-Hellman, and KEM, the following commands are provided:

Table 4: Asymmetric Encryption and Decryption Primitives

Command	Associated Algorithm(s)	Description
`TPM2_RSA_Encrypt()`	RSA	RSA encryption

(continued on next page)

(continued from previous page)

Command	Associated Algorithm(s)	Description
`TPM2_RSA_Decrypt()`	RSA	RSA decryption
`TPM2_ECDH_KeyGen()`	ECC	one-pass Diffie-Hellman with an ephemeral private key and a recipient public key
`TPM2_ECDH_ZGen()`	ECC	Diffie-Hellman with a static TPM private key and a provided public key
`TPM2_ECC_Encrypt()`	ECC	SM2 encryption as described in Clause 45.2.4
`TPM2_ECC_Decrypt()`	ECC	SM2 decryption as described in Clause 45.2.4
`TPM2_Encapsulate()`	any KEM	Key Encapsulation Mechanism public key operation
`TPM2_Decapsulate()`	any KEM	Key Encapsulation Mechanism private key operation

8.4.5.2 Labeled Key Encapsulation Mechanism (KEM)

Restricted decryption keys in the TPM cannot be used with the general-purpose asymmetric decryption commands in Table 4. They can only be used with the TPM-specific restricted-decryption protocols shown in Table 5:

Table 5: Restricted Decryption Protocols

Restricted Decryption Protocol	Label used in Labeled KEM	Relevant clause
credential protection (`TPM2_MakeCredential()`, `TPM2_ActivateCredential()`)	“IDENTITY”	Clause 21
asymmetric encryption of *salt* for salted sessions (`TPM2_StartAuthSession()`)	“SECRET”	Clause 16.6.13
object duplication and import (`TPM2_Duplicate()`, `TPM2_Rewrap()`, `TPM2_Import()`)	“DUPLICATE”	Clause 20.3

All three TPM protocols for restricted decryption use the relevant algorithm-specific Labeled KEM (Figure 3).

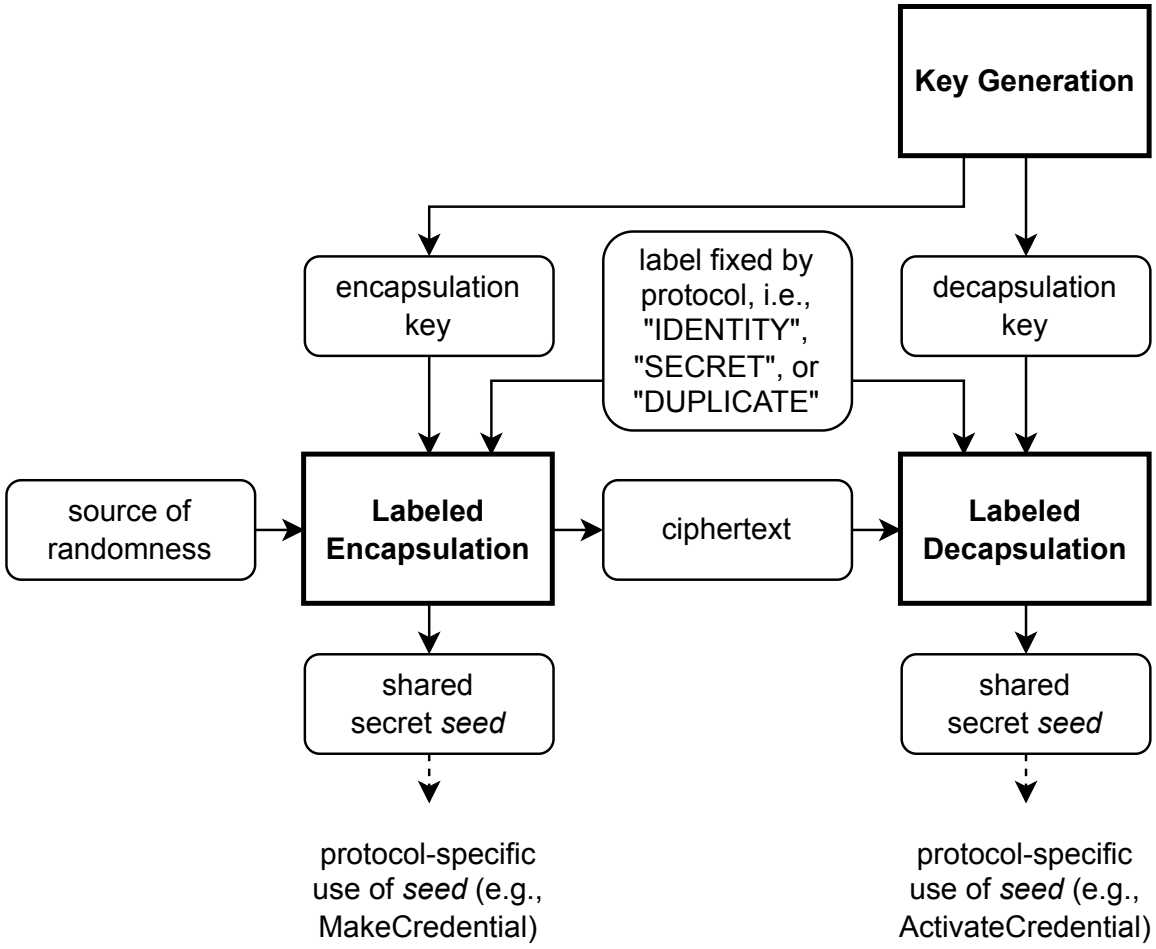


Figure 3: Labeled Key Encapsulation Mechanism

The Labeled Key Encapsulation Mechanism (KEM) in Figure 3 differs from the model of an ordinary KEM in NIST SP 800-227 [2] in that a protocol-specific label is included as a (fixed) input to the encapsulation or decapsulation. This provides domain separation, binding the shared secret to the relevant restricted-decryption protocol in use.

The party performing the labeled encapsulation uses the encapsulation (public) key, a source of randomness, and the correct label for the protocol to compute a shared secret, and a (KEM-specific) ciphertext. The party performing the labeled decapsulation (usually the TPM) decapsulates the ciphertext and produces the same shared secret. The TPM uses the shared secret to derive the key(s) used for the rest of the protocol.

See Table 5 for references to the specific clauses of this specification that discuss how the shared secret is used in each protocol.

Table 6 shows the labeled KEM for each asymmetric algorithm for a restricted decryption key.

Table 6: Algorithm-Specific Labeled KEMs

Restricted Decryption key algorithm	Relevant clause
TPM_ALG_RSA	Clause 43.10

(continued on next page)

(continued from previous page)

Restricted Decryption key algorithm	Relevant clause
TPM_ALG_ECC	Clause 44.7
TPM_ALG_MLKEM	Clause 47.4

8.4.6 Signature Operations

8.4.6.1 Signing

The TPM can 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. For symmetric signatures, HMAC and SMAC signing schemes are defined. If a key can be used for signing, then it will have the *sign* attribute.

Note:

The signing schemes are enumerated in TPMI_ALG_SIG_SCHEME in Part 2.

A key with a *sign* attribute can 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 can be used in any command that uses a signing key. For some commands, the signing scheme can 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:

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

Example:

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 can 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.

8.4.6.2 Signature Verification

TPM2_VerifySequenceComplete(), TPM2_VerifyDigestSignature(), and TPM2_VerifySignature() all validate signatures over messages or digests. Each command takes a handle of a public key, a message or its digest, and the signature over the message or its digest.

The TPM validates that the signature scheme is compatible with the selected key. In general, the TPM will be able to validate any signature it could have produced.

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

8.4.6.3 Tickets

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

> ***Note:***
>
> Hierarchy proof values are described in detail in Clause 11.5.

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:***
>
> A proof value of 256 bits is required 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_VerifySequenceComplete()`, `TPM2_VerifyDigestSignature()`, `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 (such as an authorization for `TPM2_PolicyAuthorize()`), 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.
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 can 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 can 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. An example is 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. In this case, the TPM will return a NULL Ticket. A NULL Ticket can also be used as an input parameter when the command requires a ticket, but no ticket data is available.

8.4.7 Symmetric Encryption

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 this specification.

Any symmetric block cipher supported by a TPM can be used for parameter encryption. However weak keys are not permitted to be used. Additionally, a TPM should support XOR obfuscation, which is a hash-based stream cipher. XOR obfuscation can be used only for confidential parameter passing.

Note:

XOR allows an application to have confidential and integrity-protected interactions with only one algorithm in common with the TPM (a hash).

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

Example:

SP 800-57 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:

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.

8.4.7.1 Block Cipher Modes

The block cipher modes referenced in this specification are defined in ISO/IEC 10116:2017 [3]. That specification allows parameterization of most of the modes. In a TPM implementation, the parameters are fixed, as defined in Table 7.

Table 7: 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 Codebook	none	

Note:

n is the input block size of the cipher.

8.4.7.2 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 [3] 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.

8.4.7.3 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.

When this specification 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	is a variable-sized buffer containing the data to be obfuscated
hashAlg	is the hash algorithm to be used in the KDF
key	is a variable-sized value containing a secret key

(continued on next page)

(continued from previous page)

contextU	is a variable-sized value used to qualify one of the parties to the operation (often a nonce value)
contextV	is 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 *contextV* parameters in a call to **KDFa**() to produce a *mask* value:

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

Note:

The “XOR” value is defined in “XOR”.

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

8.4.8 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:

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

where

$digest_{new}$	is the value of the digest (such as, a PCR) after the Extend operation
$\mathbf{H}_{hashAlg}$	is the hash function using a context-specific algorithm (such as, the hash algorithm associated with a specific bank of PCR)
$digest_{old}$	is the value of the digest before the Extend operation
$data_{new}$	is 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 can also apply to an NV Index that has the `TPM_NT_EXTEND` attribute.

8.4.9 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 SP 800-108 [4] is widely used in this specification.

This specification 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 can 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 described in detail in each clause where generation of a key is required.

8.4.10 Key Derivation Function

8.4.10.1 Introduction

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

The ECDH KDF is from SP 800-56C [5]. The Counter mode KDF, from SP 800-108 [4], uses HMAC as the pseudo-random function (PRF). It is referred to in the specification as **KDFa**().

8.4.10.2 KDFa()

With the exception of ECDH, **KDFa**() is used in all cases where a KDF is required. **KDFa**() uses Counter mode from SP 800-108 [4], with HMAC as the PRF.

As defined in SP 800-108, the inner loop for building the key stream is:

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

where

$K(i)$	is the i-th iteration of the KDF inner loop
$\mathbf{HMAC}()$	is the HMAC algorithm using an approved hash algorithm
K_{IN}	is the secret key material
$[i]_2$	is a 32-bit counter that starts at 1 and increments on each iteration
$Label$	is an octet stream indicating the use of the key produced by this KDF
00_{16}	is added only if *Label* is not present or if the last octet of *Label* is not zero.
$Context$	is a binary string containing information relating to the derived keying material
$[L]_2$	is 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.

As shown in Equation 6, there is an octet of zero that separates *Label* from *Context*. In SP 800-108 [4], *Label* is a sequence of octets that can or cannot have a final octet that is zero. If *Label* is not present, a zero octet is added. If *Label* is present and the last octet is not zero, a zero octet is added.

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 this specification calls for use of this KDF, it uses a function reference to **KDFa**(). The function prototype is:

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

where

hashAlg	is a TPM_ALG_ID to be used in the HMAC in the KDF
key	is a variable-sized value used as K_{IN}
label	is a variable-sized octet stream used as Label
contextU	is a variable-sized value concatenated with *contextV* to create the *Context* parameter used in Equation 6 above
contextV	is a variable-sized value concatenated to $contextU$ to create the *Context* parameter used in Equation 6 above
bits	is 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.

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 0'th 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.

8.4.10.3 KDFe for ECDH

Producing a symmetric encryption key for an ECC-protected object uses "(Cofactor) One-Pass Diffie-Hellman, C(1e, 1s, ECC CDH)" from SP 800-56A [6], using the concatenation format for *FixedInfo*. The key derivation method (KDM) used is the "One-Step Key Derivation" from SP 800-56C [5], where the Auxiliary Function is HMAC (Option 2).

Note:

KDFe is equivalent to TPM_ALG_KDF1_SP800_56A.

The inner loop of KDFe uses:

$$digest_i := \mathbf{H}(counter \parallel Z \parallel OtherInfo) \quad (9)$$

where

$digest_i$	is the digest generated on the i-th iteration of the loop (i starts at 1)
$\mathbf{H}$	is an approved hash function
$counter$	is a 32-bit counter that is initialized to 1 and incremented on each iteration
Z	is the X coordinate of the product of a public ECC key and a different private ECC key
$OtherInfo$	is 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 SP 800-56A [6], *OtherInfo* is specified as:

$$OtherInfo := AlgorithmID \parallel PartyUInfo \parallel PartyVInfo \{ \parallel SuppPubInfo \} \{ \parallel SuppPrivInfo \} \quad (10)$$

where

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

This specification requires that *OtherInfo* be constructed as:

$$OtherInfo := Use \parallel PartyUInfo.buffer \parallel PartyVInfo.buffer \quad (11)$$

where

Use	is a null-terminated string indicating the use of the key (e.g., “DUPLICATE”, “IDENTITY”, “SECRET”, etc.) (see Clause 3 for the definition of these values). This field satisfies the requirements of SP 800-56A [6] since the parsing of parsing of keying material is determined by the use.
PartyUInfo.buffer	is the x-coordinate of the public point of an ephemeral key
PartyVInfo.buffer	is 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 this specification calls for use of this KDF, it uses a function reference to **KDFe**(). The function prototype is:

$$\mathbf{KDFe}(hashAlg,\ Z,\ Use,\ PartyUInfo,\ PartyVInfo,\ bits) \tag{12}$$

where

hashAlg	is the hash algorithm to be used as $\mathbf{H}()$ in Equation 9 above
Z	is the x-coordinate of the product of a public point and a private key
Use	is a null-terminated string indicating the use of the key (e.g., “DUPLICATE”, “IDENTITY”, “SECRET”, etc.) (see Clause 3 for the definition of these values).
PartyUInfo	is the x-coordinate of the public point of an ephemeral key
PartyVInfo	is the x-coordinate of the public point of a static TPM key
bits	is 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.**

8.4.10.4 Rejection of weak keys

Some algorithms have known weak keys. If such a key is generated, it must be discarded, and a new key generated by starting over with another iteration of the KDF. In the case of DES, there are 64 known weak or semi-weak keys. None of them are allowed. In the case of AES, at least one bit in the upper half of the key must be set. Again, if this is not true, the key must be discarded, and a new key generated by starting over with another iteration of the KDF.

8.4.11 Random Number Generator (RNG) Module

8.4.11.1 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 can be implemented with a pseudo-random number generator (a PRNG). A PRNG can produce numbers that are apparently random from a non-random input (such as, a counter). 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.

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 can 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.

8.4.11.2 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.

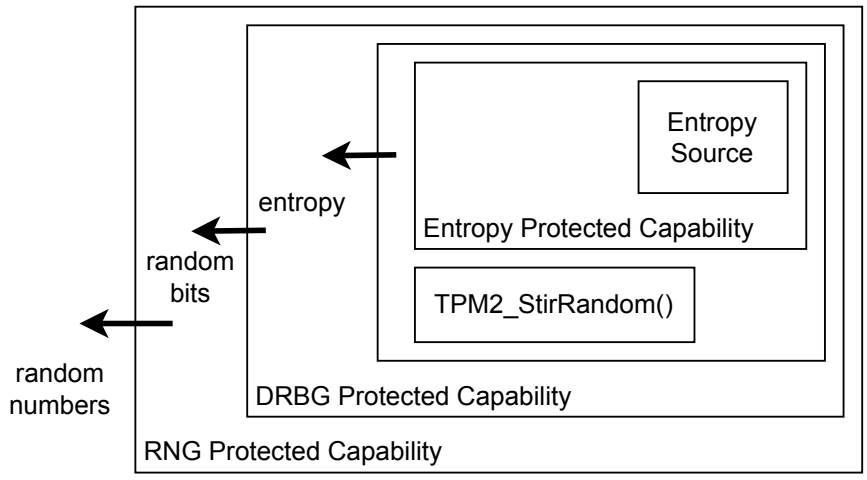


Figure 4: Random Number Generation

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:

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

Note:

It is recommended that the DRBG Protected Capability of a non-FIPS TPM consists of a DRBG mechanism that complies with NIST Recommendation SP 800-90A [7], revised March 2012; except it does not comply with its Clause 11.

Note:

The DRBG Protected Capability of a FIPS TPM consists of a DRBG mechanism that complies with NIST Recommendation SP 800-90A, revised March 2012.

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:

Each TPM can 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.

8.4.11.3 Nonce Creation

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

8.4.12 Algorithms

8.4.12.1 Algorithm Identifiers

The structures and commands in this specification are constructed with minimal reliance on algorithm defaults.

In most cases, an algorithm identifier identifies a family of algorithms followed by qualifiers. Since this specification depends on being able to discern the hash output size from the algorithm ID, its hash algorithm identifiers imply a digest size.

Example:

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, such as RSA, ECC, AES, etc. For these algorithms, supplementary information is required to define parameters.

Example:

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

8.4.12.2 Algorithm Support

This specification 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.

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. For example, the TPM may 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 can 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 can be supported in any combination.

Note:

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:

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 can only implement algorithms that have a TCG-assigned algorithm ID.

8.5 Authorization Subsystem

The Authorization Subsystem is called by the Command Dispatch module at the beginning and end of command execution. Before the command can 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 16.

8.6 Random Access Memory

8.6.1 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 this specification they are referred to as being volatile, even if the data loss is implementation-dependent.

When the specification 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 described in Clause 8.3.

8.6.2 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 can 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 can 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 can 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:

Since banks may have different numbers of PCR, a PCR Index value may not be valid for all banks. The allocation of PCR can also be changed by TPM2_PCR_Allocate() using Platform Authorization. Changing the PCR allocation does not change the attributes of the PCR.

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

Although listed in this clause, 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:

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.

8.6.3 Object Store

TPM RAM holds keys and data that are loaded into the TPM from external memory. In most cases, an object cannot 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 can 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 has to be loaded before being accessed, and the loaded blob can 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.

8.6.4 Session Store

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

A session can 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.

8.6.5 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 this specification 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.

8.7 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 the specification 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 can define a special NV data structure (an NV Index) in order to store persistent data values. NV memory can 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 can 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.

8.8 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.

9 TPM Operational States

9.1 Introduction

This clause describes TPM operational states and state transitions.

9.2 Basic TPM Operational States

9.2.1 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.

9.2.2 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. For example, if the TPM is a component on the PC's motherboard, `_TPM_Init` should coincide with a reset of the processor and chipset. 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.

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:

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

Note:

If `TPM2_Startup()`/`TPM2_FieldUpgradeData()` 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 cannot be 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:

For compliance with other standards, such as FIPS 140, it is 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_FieldUpgradeData().This validation can require use of a digital signature or message authentication code.

Occasionally, some TPM state could 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 can 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.

9.2.3 Startup State

9.2.3.1 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.

9.2.3.2 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 this specification 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:

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).

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 Shutdown(STATE) (see Clause 9.2.4). 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.

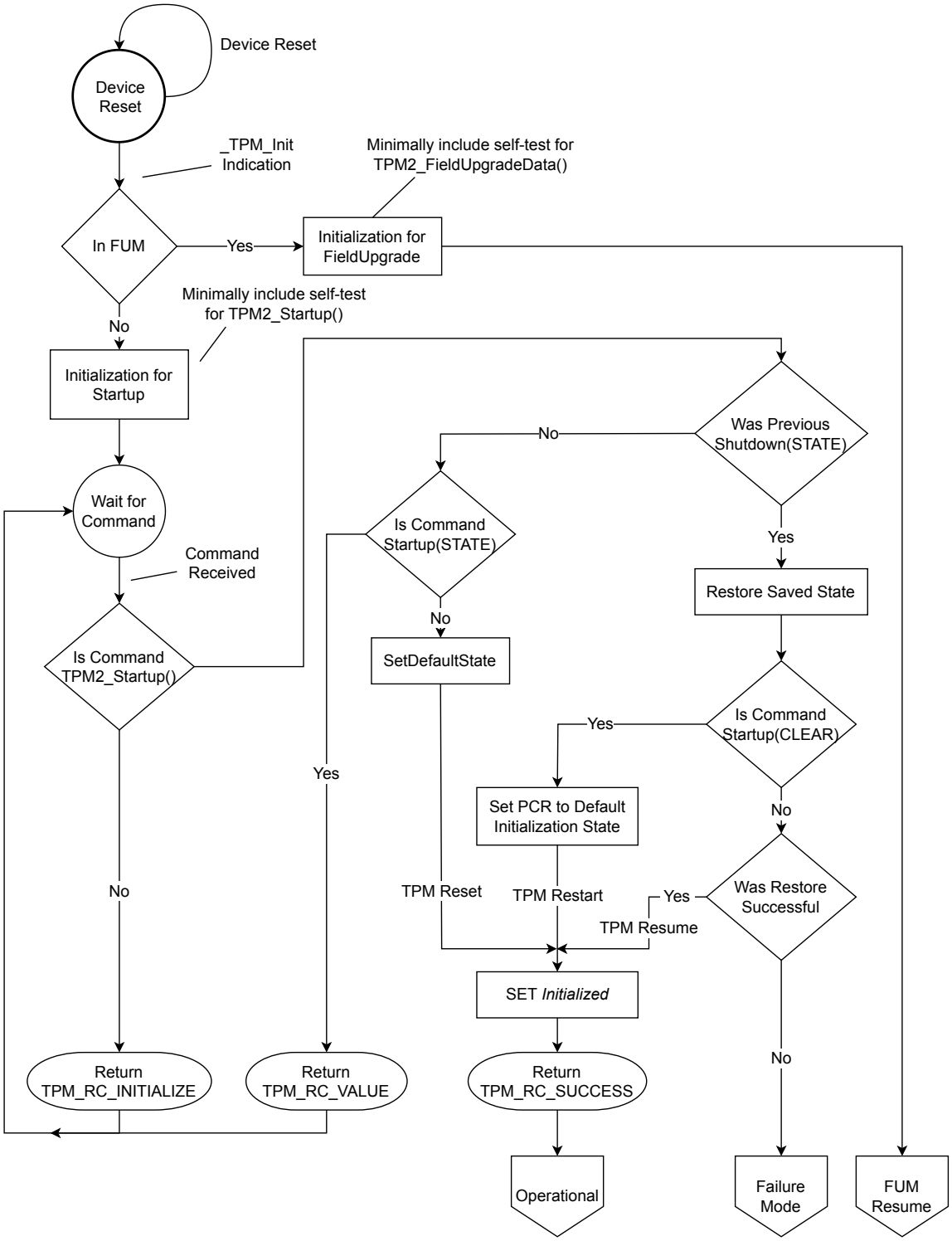


Figure 5: TPM Startup Sequences

9.2.4 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.

Shutdown(STATE) preserves the majority of the TPM operational state so that it can be restored on a subsequent TPM2_Startup(). Shutdown(CLEAR) preserves a minimal amount of state, mostly to ensure

continuity of the TPM timing functions.

Note:

The timing functions are described in Clause 33.

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 the specification 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:

This is not an exhaustive 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()`.

9.2.5 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 can 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.

9.3 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:

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.

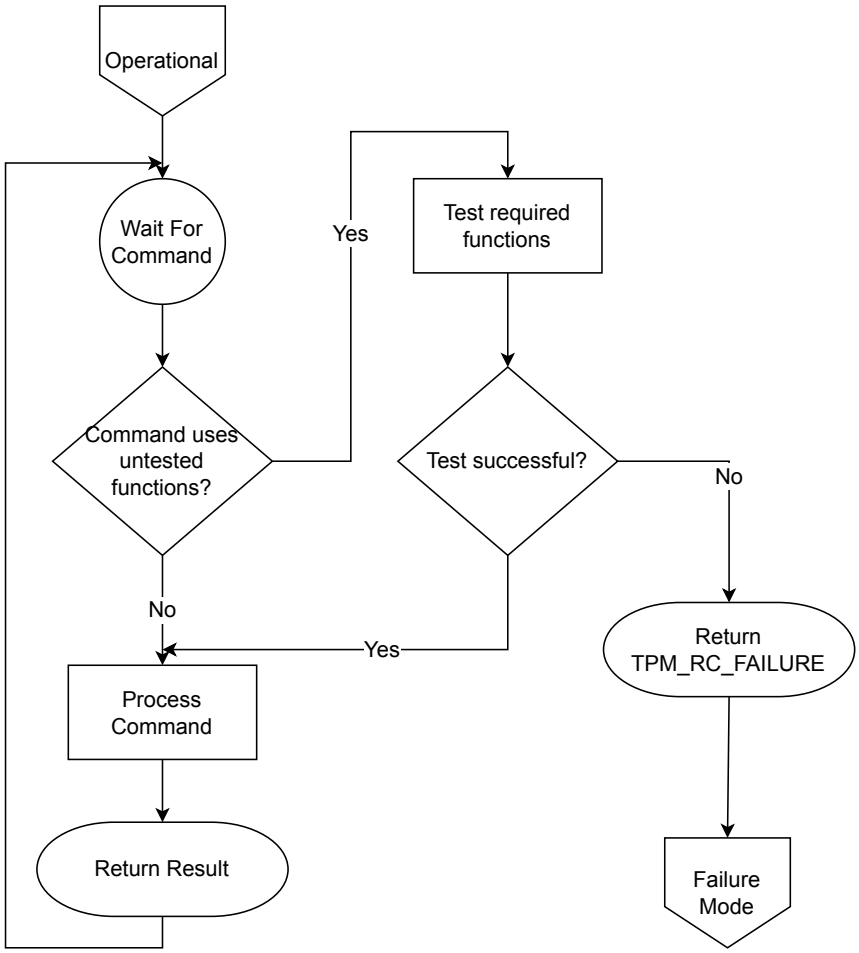


Figure 6: On-Demand Self-Test

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.

Certification standards often constrain behavior with respect to power-on self-testing. For more information about satisfying these requirements, see the TCG Protection Profiles [8] or FIPS guidance documents [9] [10].

Note:

Authenticated tests can 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.

9.4 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 is allowed to return other property values.

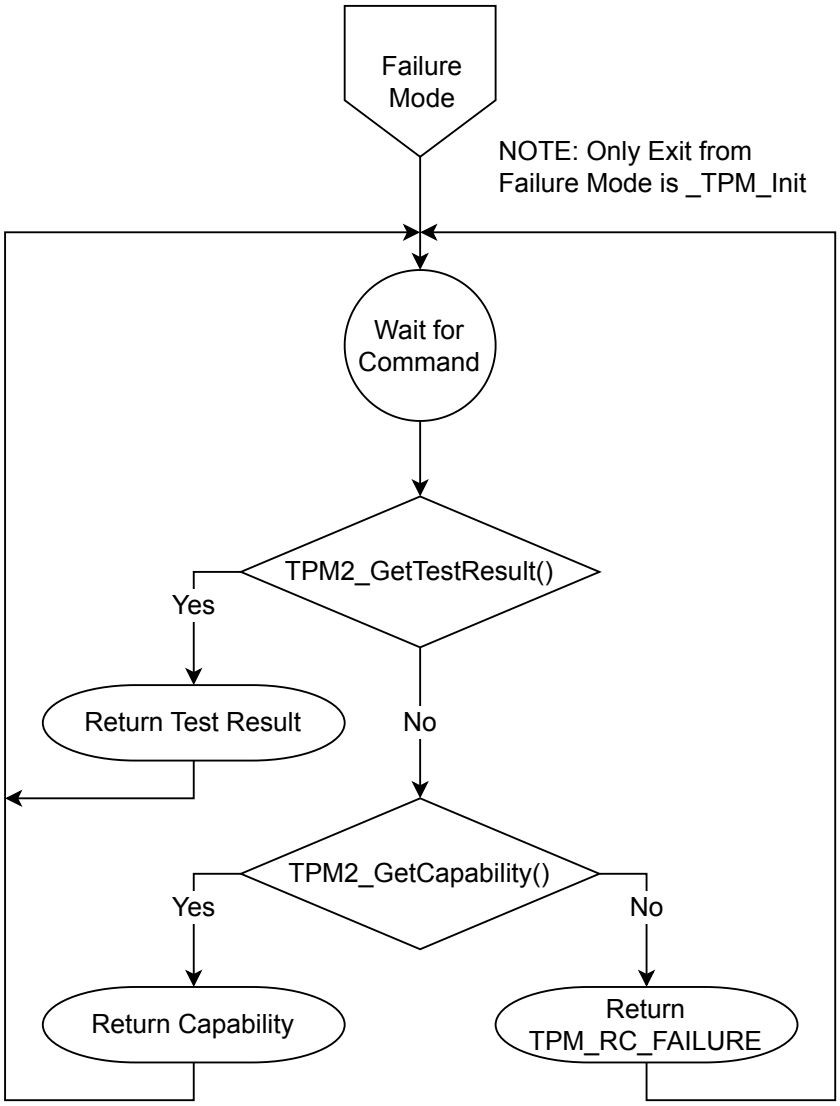


Figure 7: Failure Mode Behavior

9.5 Field Upgrade

9.5.1 Introduction

This specification describes optional Protected Capabilities for upgrading the TPM firmware. The methods described in this specification 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 described in this specification. In particular, the authorizations for the upgrade should be the same as the field upgrade commands in this specification.

9.5.2 Field Upgrade Mode

This specification 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 8). 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_FieldUpgradeStart()`, 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 9). 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. `TPM2_FirmwareRead()` may 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.

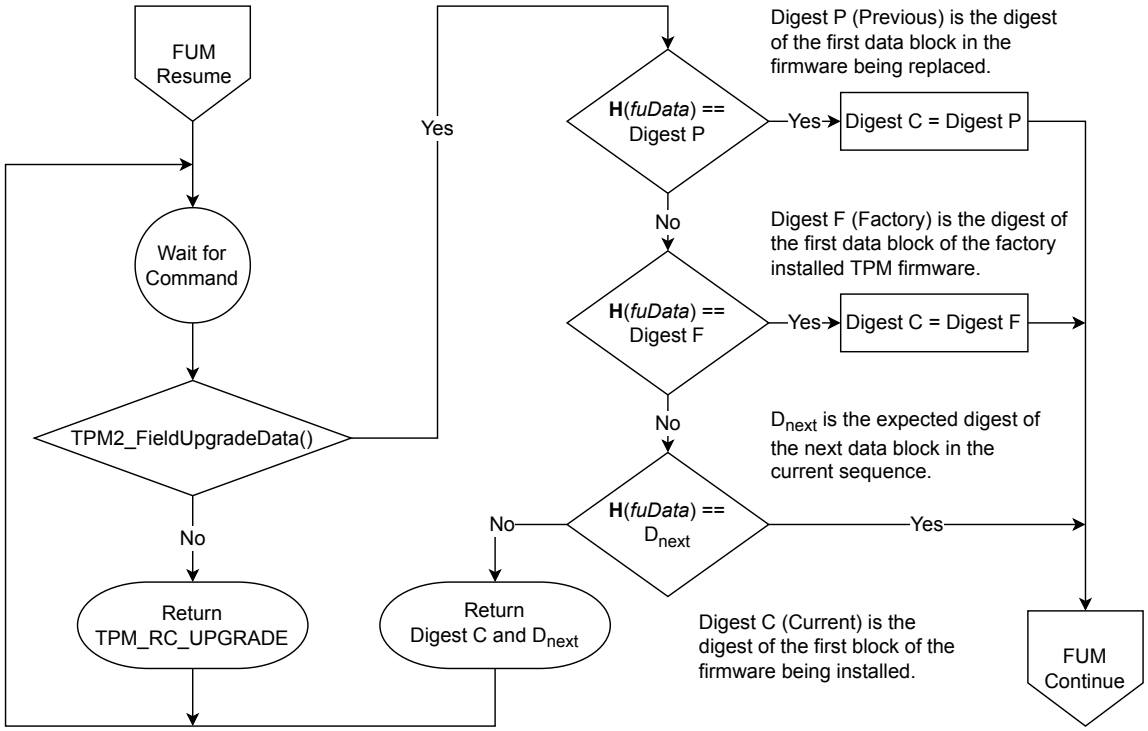


Figure 8: Resuming Field Upgrade Mode after `_TPM_Init`

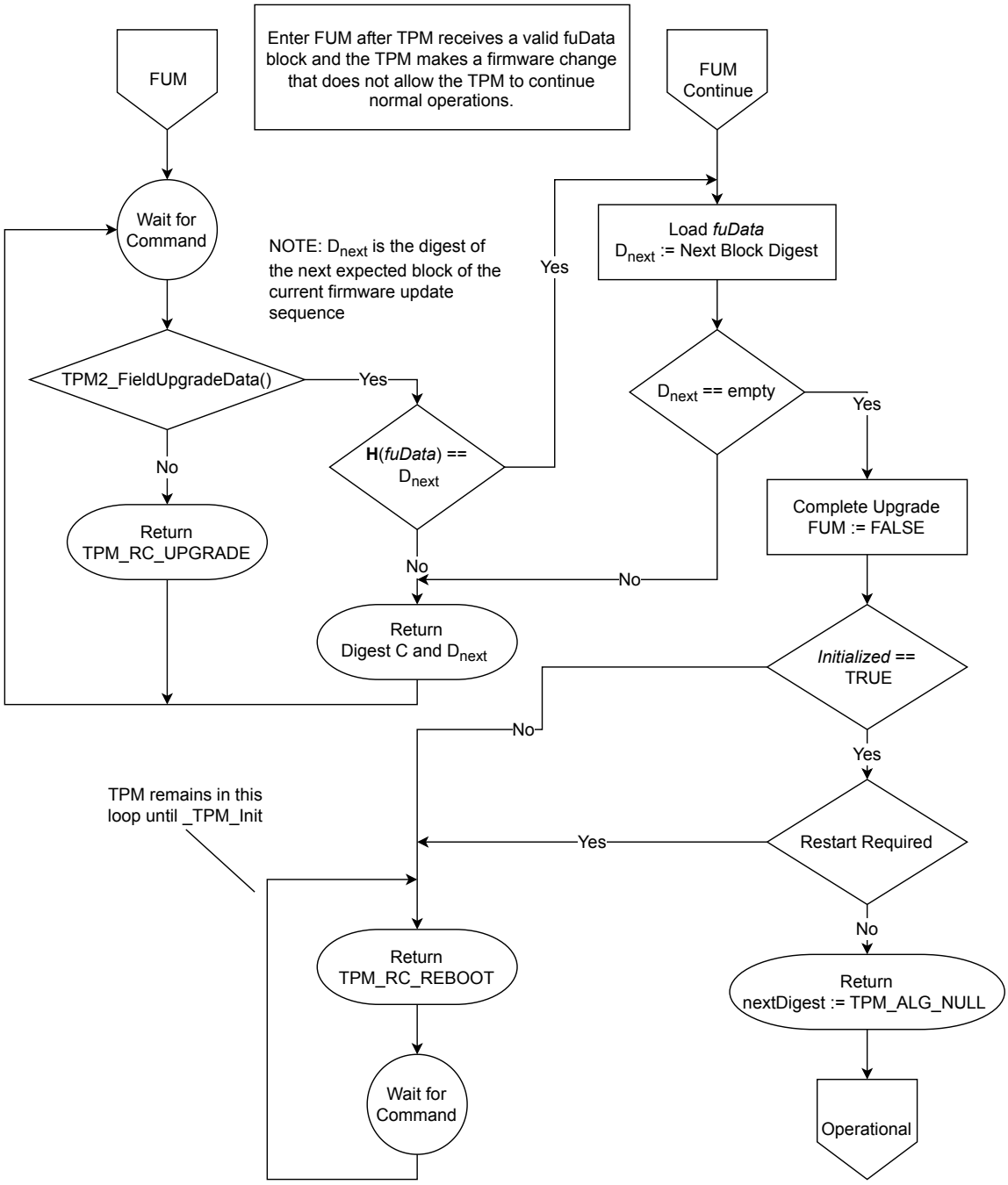


Figure 9: Field Upgrade Mode

9.5.3 Preserved TPM State

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

- Primary Seeds (and the primary keys generated from them);
- 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.

Note:

A platform manufacturer may provide a means to change preserved data to accommodate a case where a field upgrade fixes a flaw that might have compromised TPM secrets.

9.5.4 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 this specification.

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

10 TPM Control Domains

10.1 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 is 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. For example, the platform hierarchy can be disabled while still permitting access to platform firmware NV space.

10.2 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 11). 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 the specification indicates that an action can be authorized with an authorization value, the corresponding policy is also allowed. For instance, 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* can 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 8 shows the control combinations.

Table 8 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 8 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 Clause 16.7 for the description of *policyDigest* generation and use).

Table 8: Hierarchy Control Setting Combinations

hierarchy *enable*	***authValue***	***authPolicy***	**Description**
SET	Known	Set	The hierarchy is enabled, and objects in it can be loaded. Either authValue or authPolicy can manage resources related to the hierarchy.
SET	Unknown	Set	The authValue can 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. An example is delegating control of creating Primary Objects in a hierarchy to one entity while delegating control of related NV resources to a different entity.
SET	Known	Empty	When the authPolicy is empty, it cannot match any policyDigest value so the use of authPolicy is disabled.
CLEAR	N/A	N/A	When an enable is FALSE, the corresponding authValue and authPolicy cannot be used to authorize any TPM action.

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

10.3 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 can 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 manufacturer can 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 can 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 can 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:

This is not a comprehensive list. The uses of the platform controls are documented in Part 3. In that document, 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 described 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 can 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:

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

Note:

phEnable has to be SET before `TPM2_Startup()` when accommodating the case of an interrupted field upgrade that prevents startup from running. phEnable has to be SET to permit field upgrade authorization.

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

10.4 Owner Controls

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

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

The *shEnable* gates use of both *ownerAuth/ownerPolicy* and the SPS hierarchy, as described in Clause 10.2.

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

The *ownerAuth* 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 can be explicitly changed by designated commands.

10.5 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 can be considered to be privacy-sensitive require use of objects in the Endorsement hierarchy. For 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 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.

The *ehEnable* gates use of *endorsementAuth/endorsementPolicy* and the EPS hierarchy, as described in Clause 10. 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:

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 *ehEnable* 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 *ehEnable* can be CLEAR using either Endorsement Authorization or Platform Authorization. When *ehEnable* is CLEAR, it can be SET using Platform Authorization

Note:

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

The *endorsementAuth* 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 can be explicitly changed by designated commands.

10.6 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. For Primary Objects created in firmware-limited or SVN-limited hierarchies, the required authorization is that of the associated base hierarchy.

10.7 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, such as smartcards, 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.

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:

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:

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:

`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 *lockoutAuth* value can be changed (`TPM2_HierarchyChangeAuth()`) only when its current value is provided. *LockoutPolicy* can 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 Clause 16.8 for full details on setting of parameters associated with dictionary attack logic and other aspects of the dictionary attack protection.

10.8 TPM Ownership

10.8.1 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. The OS is expected to change these values and manage them on behalf of the platform Owner.

The platform can 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.

10.8.2 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()` can be authorized using either Platform Authorization or Lockout Authorization. If the control is SET, then `TPM2_Clear()` is not functional.

Note:

TPMA_PERMANENT.*disableClear* can 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*.

11 Primary Seeds

11.1 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 described in Clause 24.5. 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 can then be context saved and loaded like any other object. It can 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.

11.2 Rationale

The algorithm flexibility provided by this specification makes it possible for the TPM to support many different asymmetric key types. The addition of ECC support significantly increases the number of parameters because curve parameters can vary based on application.

While this flexibility is a major benefit of TPM 2.0, it creates new challenges for managing TPM Endorsement Keys (EKs) and EK certificates. As mentioned in Part 0, "Root of Trust for Reporting (RTR)", 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 this specification 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 this specification 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. . When the key parameters or the seed change, the TPM generates a different EK.

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. Since an EK is not generally duplicable, the owner has a choice to make. They can either re-create the EK whenever it is needed or tell the TPM to save the EK in persistent memory.

The seed key concept can 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.

It is preferred that a TPM manufacturer generate a certificate for at least one EK before the device ships. This certificate would be based on the EPS that is present in the TPM at that time.

The Primary Seed 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 can 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.

11.3 Considerations for Expensive-to-Generate EKs

The EK public key has to be known at manufacturing time in order to issue the EK certificate. For some EK algorithms (for example, RSA), some TPM implementers may prefer to deterministically generate the key pair from the EPS at manufacturing time (for example, for performance reasons).

In such cases, the TPM manufacturer has, for example, the following options, which are described in the subsequent sections:

- Inject the EPS.
- Inject the EK directly.

11.3.1 Injecting the EPS

The TPM manufacturer could generate the EPS outside of the TPM and inject it at manufacturing time. The time taken to inject an EPS would be deterministic, and one or more EK key pairs could be generated (and certified) for that EPS outside of the TPM. This could save considerable time during manufacturing.

The external algorithm for generating a key pair from the EPS has to be the same as the algorithm used by the TPM; otherwise, they will generate different keys. The generation times for the external and TPM processes will be proportional, so the manufacturer can use the external key generation time as an indicator of the time that the TPM will take when the end user attempts to recreate the EK. If the manufacturer does decide to inject an EPS and generate EKs outside of the TPM, there is an opportunity to benefit the customer by discarding EPS values that result in long key pair generation times for the certified values.

11.3.2 Injecting the EK

Another possible option for the TPM manufacturer is to inject precomputed EK(s) compatible with specific template(s) (“compatible” meaning that the algorithm and all parameters are consistent between the template and the injected EK). The TPM will access that injected EK when the associated template is used. After the EPS is changed, `TPM2_CreatePrimary()` will create a different EK even if the same template is used. This means that a different EK will be produced by `TPM2_CreatePrimary()` before and after the EPS was changed.

11.4 Primary Seed Properties

11.4.1 Introduction

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

Example:

RSA2048 has historically been considered to have a security strength of 112 bits. If it were the strongest algorithm on the TPM, then the minimum size for all Primary Seeds would be 224 bits.

Example:

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

Example:

ML-KEM-1024 claims a security level that is equivalent to breaking a block cipher with 256-bit keys.

If ML-KEM-1024 were implemented, then the minimum size for all Primary Seeds would be 512 bits.

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

After a field upgrade that changes the Primary Seed strength, or that changes the algorithm that uses the Primary Seed, the TPM shall generate the original EKs corresponding to the EK certificates provisioned by the TPM manufacturer if the same template is provided to the `TPM2_CreatePrimary()` command until such time as `TPM2_ChangeEPS()` command changes the EPS.

For a field upgrade that does not change the Primary Seed strength or the algorithm that uses the Primary Seed and does not otherwise affect the security of Primary Objects based on the seeds, `TPM2_CreatePrimary()` with the same inputs should produce the same Primary Object in the platform, storage, and endorsement hierarchies after the field upgrade as it did before the field upgrade.

This requirement shall not be in effect for other keys derived from the EPS or for keys derived from the SPS or PPS.

Example:

A field upgrade can cause `TPM2_CreatePrimary()` to generate a different key for the same input template. For example, versions prior to version 1.38 used KDFa, while version 1.38 and after use DRBG. In addition, the security strength requirement could cause a change in the seed length if the field upgrade implements a stronger algorithm.

11.4.2 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()` can 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 can 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 can 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 *endorsementAuth* or *endorsementPolicy* is required to use the EPS for creation of a Primary Object in the Endorsement hierarchy.

11.4.3 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 (such as, OS and applications).

Note:

The platform firmware can 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()` can 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:

The TPM does not enforce this imperative.

Note:

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 can be the same for all objects in the Platform hierarchy and for all platforms that implement the chosen policy hash.

11.4.4 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()` can be used to change the SPS if the TPM owner wants to ensure that no previously generated keys in the Storage hierarchy can 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 can be recreated.

11.4.5 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.

11.5 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 cannot be used when its associated hierarchy is disabled.

Example:

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:

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:

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 *ehProof* in the protection of Ordinary Objects in the Endorsement Hierarchy ensures that those Objects will be deleted when the Owner changes, because *ehProof* also changes 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. Examples are saved contexts and tickets. A hierarchy proof value may be used in other computations as long as the result of the computation does not leave the TPM.

Proofs for firmware-limited hierarchies are derived from the base hierarchy's proof, as well as the Firmware Secret. Proofs for SVN-limited hierarchies are derived from the base hierarchy's proof, as well as the Firmware SVN secret associated with the given SVN.

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:

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

Note:

According to SP 800-57, 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.

12 TPM Handles

12.1 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 can refer to only one TPM-resident resource at a time.

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

To reduce confusion, PCR are assigned handles that have the same values as in previous versions of the specification. A PCR handle is an Index into an array of PCR. A PCR's Index and handle value are the same.

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

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

12.4 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 can 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:

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:

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:

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:

The “version number” is how the TPM prevents replay of an authorization.

When an authorization session is no longer needed, `TPM2_FlushContext()` can be used to delete all context associated with the session from TPM memory (see Clause 27.6). The session handle for this command can use an upper octet of either 02_{16} or 03_{16}.

Note:

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:

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 can 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 Clause 15.6.4).

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

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

Fixed resource handles refer to Shielded Locations that are always associated with the same handle. These resources have handles with an MSO of 40_{16}. Examples of these resources are Owner, Platform, and Endorsement hierarchy controls and the Lockout authorization value.

Another type of permanent resource handle is the vendor-specific authorization value. These optional resources can be populated with authorization values that are known only by the TPM manufacturer or some other privileged entity. The update of these authorization values is TPM-manufacturer-dependent.

If present, a vendor-specific authorization value can be used as a bind value within an authorization session or to authorize a policy using the `TPM2_PolicySecret()` command. In the former case, an entity that knows the authorization value could create an auditable authorization session that only that entity could execute. In the latter case, the entity could create and/or use TPM resources with an authorization policy that only that entity could execute.

Since vendor-specific authorization values might be usable by an entity who knows them to identify the TPM, the use of these authorization values is under the control of the privacy administrator. These authorization values are only usable when the Endorsement Hierarchy is enabled as described in Clause 10.5.

Note:

A use case for the vendor-specific authorization values is to recover in the field from a flaw in the TPM firmware. For example, TPM vendors may provide a mechanism that updates one or more of these authorization values based on the measurement of the TPM firmware. This update mechanism could be used to give the manufacturer confidence that a valid, uncompromised version of the TPM firmware is running. In this scenario, if the manufacturer wished to provide a certificate for an endorsement key generated in the field after a field upgrade to a trusted firmware version occurred, the manufacturer could use an auditable authorization session using the vendor-specific authorization value to verify the properties of the endorsement key and then create a certificate for that new endorsement key.

12.6 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 can be flushed from TPM memory using TPM2_FlushContext() and indicating the handle of the loaded context to be flushed.

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

TPM2_EvictControl() can 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 can be assigned to a new Persistent Object if the first Object is deleted from persistent storage.

13 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:

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:

The caller should use its copy of the NV public area and calculate the Name before using it in an HMAC authorization calculation. Otherwise, an invalid authorization can trigger the dictionary attack protection depending on TPMA_NV_NO_DA.

The TPM must check access control before checking authorization. For example, it should 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 9 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 9: Equations for Computing Entity Names

MSO of Handle	Entity Type	Equation for Computing the Name
0x00	PCR	$Name := handle$ No hash is performed on the handle to produce the name and the name is only the size of the handle.
0x02	HMAC Session	
0x03	Policy Session	
0x40	Permanent Values	
0x01 0x11 0x12	NV Index	$Name := nameAlg \parallel \mathbf{H}_{nameAlg}(handle \rightarrow nvPublicArea)$ where $nameAlg$: algorithm used to compute *Name* $\mathbf{H}_{nameAlg}$: hash using the *nameAlg* parameter in the NV Index location associated with *handle* $nvPublicArea$: contents of the TPMU_NV_PUBLIC_2 associated with *handle*
0x80	Transient Objects (1)	$Name := nameAlg \parallel \mathbf{H}_{nameAlg}(handle \rightarrow publicArea)$ where $nameAlg$: algorithm used to compute *Name* $\mathbf{H}_{nameAlg}$: hash using the *nameAlg* parameter in the object associated with *handle* $publicArea$: contents of the TPMT_PUBLIC associated with *handle*
0x81	Persistent Objects	

Note:

(1) The Name of a sequence object is an Empty Buffer (see Clause 29.4.6).

Note:

For a legacy TPM_HT_NV_INDEX, nvPublicArea is a TPMS_NV_PUBLIC, so this calculation is unchanged from earlier versions of this specification, where an NV index's public area was always a TPMS_NV_PUBLIC.

The bare TPMU_NV_PUBLIC_2 contents are used instead of the full TPMT_NV_PUBLIC_2 contents for consistency with earlier versions of this specification, which did not have a tagged union for NV public areas. Every member of the TPMU_NV_PUBLIC_2 union begins with a TPM_HT (as the most significant byte of the index's handle), and TPM_HT is the structure tag for TPMU_NV_PUBLIC_2. Therefore, it is infeasible to find a Name that is shared by NV indices of two different types, because the type (TPM_HT) is already included in the Name hash.

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.

14 PCR Operations

14.1 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(TPM_SU_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 can also be reset by TPM2_PCR_Reset() or by a Dynamic Root of Trust (D-RTM) sequence (see Clause 31.2). 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.

The default initial condition for any PCR, other than PCR[0], is either all bits CLEAR or all bits SET. For PCR[0], the default initial condition may all bits CLEAR, all bits SET, the locality at which TPM2_Startup() was received, or an indicator that 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 can choose from the options list above.

Example:

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

Example:

A platform-specific specification can 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:

A virtual TPM may 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 Clause 14.7).

14.2 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

$$PCR_{\text{new}} := \mathbf{H}_{alg}(PCR_{\text{old}} \parallel digest) \tag{13}$$

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 Clause 14.7).

14.3 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:

$$PCR.digest[pcrNum][alg]_{new} := \mathbf{H}_{alg}(PCR.digest[pcrNum][alg]_{old} \parallel digest) \quad (14)$$

where

$\mathbf{H}_{alg}$	is a hash function using the algorithm associated with the PCR instance
$PCR.digest$	is the digest value in a PCR
$pcrNum$	is the PCR numeric selector
alg	is the PCR algorithmic selector
$digest$	is the 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.

14.4 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 supported by the TPM. Events no larger than 1024 octets can use TPM2_PCR_Event(). Events exceeding 1024 octets can 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 each implemented hash algorithm.

Example:

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

TPM2_EventSequenceComplete() requires that proper authorization be provided (see Clause 14.7).

Recording of an event can 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 create 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].

14.5 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. Examples are TPM2_PCR_Read(), TPM2_Quote(), and TPM2_PolicyPCR() that allow the caller to make arbitrary selections of PCR in multiple 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.

The bit correspondence to PCR is that the bit corresponding to PCR[*n*] is the (*n* mod 8) bit in the $\lfloor n/8 \rfloor$ octet of the array.

Example:

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 $\lfloor 0/8 \rfloor = 0^{th}$ octet (the octet with the 01_{16} value) and the bit for PCR[13] is the 13 mod 8 = 5^{th} bit in the $\lfloor 13/8 \rfloor = 1^{st}$ 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.

14.6 Reporting on PCR

14.6.1 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 can modify the selection structure and make another read request to get additional PCR values.

Since the PCR can 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 Clause 14.9 for exemptions). If this counter value changes between calls, the sequence can be repeated until the desired PCR are all returned with no change to the counter value.

14.6.2 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 can be compared to a required digest (TPM2_PolicyPCR()) or returned in an attestation (TPM2_Quote()).

To validate an attestation quote, a remote caller will typically use the PCR to recalculate the digest value. TPM2_Quote() returns only the digest, and the PCR values must be retrieved separately.

This can lead to a race condition. The PCR values can change between the time of the quote and the time they are read. There are several solutions. The PCR can be read before and after the quote to ensure that they did not change. Alternatively, the quote digest can be validated locally against the PCR before returning results to a remote caller, and the quote can be rerun until the validation succeeds.

14.7 PCR Authorizations

TPM2_PCR_Reset(), TPM2_PCR_Extend(), TPM2_PCR_Event(), and TPM2_EventSequenceComplete() require authorization for the PCR being modified. The type of the authorization can differ based on the PCR being modified. A PCR can 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 can be read using TPM2_GetCapability(*capability* == TPM_CAP_PCR_PROPERTIES, *property* == TPM_PT_PCR_AUTH) and the PCR in a policy set can be read using TPM2_GetCapability(*capability* == TPM_CAP_PCR_PROPERTIES, *property* == TPM_PT_PCR_POLICY).

Note:

The Reference Code only provides support for one set of each type. If additional sets are needed, the property types for TPM_CAP_PCR_PROPERTIES can be extended.

Note:

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 can be provided in either an HMAC session using a zero-length *authValue* in the HMAC calculation or as a zero-length password.

Neither a password session, an HMAC session, nor a policy session failure causes an update of the dictionary attack protection.

Note:

PCR have an implied *noDA* attribute SET.

14.7.1 PCR Not in a Set

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

14.7.2 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* can be changed using `TPM2_PCR_SetAuthValue()` by an entity with knowledge of the *authValue*.

14.7.3 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 can 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 can 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`).

14.7.4 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.

14.8 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 `_TPM_Init` and persist until the next `TPM2_PCR_Allocate()`.

Note:

Because of RAM limitations, an implementation may 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 cannot be made for PCR if the attributes for the PCR are not defined by the platform-specific specification of that TPM.

Note:

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 can 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()`.

14.9 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 Clause 16.7.7.6).

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* == `TPM_CAP_PCR_PROPERTY`, *property* == `TPM_PT_PCR_NO_INCREMENT`).

14.10 Other Uses for PCR

The PCR-related commands defined in this library cover common use cases: for example, logging of components during boot or a runtime-switch in the TCB. Platform-specific specifications define PCR attributes that control this behavior and describe how PCR should be used by external software.

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. For instance, 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 or 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. This specification demands no particular behavior or value-semantics for such PCR.

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.

15 TPM Command/Response Structure

15.1 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:***
>
> 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:***
>
> 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:***
>
> 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.

Note:

Not all sessions in the Authorization Area are required to be used for authorization. Sessions can also be used for audit or parameter encryption.

15.2 Command/Response Header Fields

A command or response header always contains three values, displayed in Table 10.

Table 10: Command/Response Header Structure

tag
commandSize or *responseSize*
commandCode or *responseCode*

15.2.1 tag

A *tag* is present in all commands sent to the TPM and in responses received from the TPM. Table 11 lists the *tag* values used for commands and response defined in this specification.

Note:

The tags for commands defined in this specification 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 can be parsed to find the start of the next session.

Table 11: Tag Values

Value	Description
TPM_ST_NO_SESSIONS	This value indicates that the command or response is formatted according to this specification 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 this specification 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.

15.2.2 commandSize/responseSize

The *commandSize*/*responseSize* value indicates the total number of octets of this command/response, starting with the first octet of *tag*.

15.2.3 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*).

15.2.4 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.

15.3 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 (such as, TPM2_Startup()) require 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:

TPM2_ObjectChangeAuth() has two handles, one (*objectHandle*) that uses an authorization session, and one (*parentHandle*) that does not. The standard command syntax requires that *objectHandle* occur first.

A response can 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.

15.4 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 (such as, TPM2_Clear()) require 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:

If a parameter is encrypted, it is included in the *cpHash/rpHash* after encryption. Because audit also uses *cpHash* and *rpHash*, audit of an encrypted session, although valid, is unlikely to be useful at the application level.

A response can 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:

This is the limit of parameters in the parameter list, not the number of values that can be in the parameter area. If a command needs more than 15 parameters, a new structure can be defined that encapsulates two or more of those parameters into a single structure, which can then be unmarshaled as a unit. The only loss is that error reporting will not provide as much detail when a compound parameter has an error.

As described in Clause 18, for a command or response parameter to be encrypted, it must be the first parameter and it must be a TPM2B type.

Note:

In order to encrypt more than one parameter, they have to be encapsulated in a TPM2B, making them a single parameter.

Example:

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.

15.5 *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:

The maximum value depends on the size of the largest digest produced by any hash implemented on the TPM.

Note:

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* can have a value of zero.

authorizationSize is not included in *cpHash*, and *parameterSize* is not included in the *rpHash*.

15.6 Authorization Area

15.6.1 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 Part 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 can be added to the Authorization Area. Those sessions can be designated as being for encryption, decryption, or audit; in any combination, in any order. However, in a single command, only one session is allowed 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 can 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 can have the *encrypt*, *decrypt*, and *audit* attributes all set. A password authorization cannot 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:

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 12.

Table 12: Use of Authorization/Session Blocks

Position	password authorization (1)(6)	authorization session (2)(6)	encryption session (3)	decryption session (4)	audit session (5)
1	[x]	[x]	[x]	[x]	[x]
2	[x]	[x]	[x]	[x]	[x]
3			[x]	[x]	[x]

Note:
[1] a password authorization cannot be used for encryption, decryption, or audit.
[2] an HMAC authorization session can also be used for encryption, decryption, and audit and a policy authorization session can also be used for encryption and decryption
[3] only one session can be designated as being used for encryption
[4] only one session can be designated as being used for decryption
[5] password authorization sessions and policy sessions cannot be used for audit
[6] authorization sessions come before sessions used only for encryption, decryption, or audit

In Part 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:

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 can 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.

15.6.2 Authorization Structure

15.6.2.1 Command

In a command, each authorization structure has the format shown in Table 13.

Table 13: Authorization Layout for Command

Field	Description
session handle	a four-octet value indicating the session handle 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

15.6.2.2 Response

In a response, each session structure has the format shown in Table 14.

Table 14: Authorization Layout for Response

Field	Description
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

Clause 16.6.7 describes the methods for creating an authorization session.

15.6.3 Session Handles

Session handles are described in Clause 12.4. 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 can occur only once in the Authorization Area. The handle representing a password authorization (TPM_RS_PW) can occur multiple times.

15.6.4 Session Attributes

Each session has a *sessionAttributes* octet to indicate how the session is to be applied. Table 15 explains the meaning of the fields in this octet.

If a session is not being used for authorization, at least one of decrypt, encrypt, or audit must be SET.

Table 15: 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 (see the exception for password authorization below). NOTE In this context, "echo" means that the value of a session attribute will be the same in the response as it was in the command. 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 18). The parameter will be decrypted after the HMAC computations are successfully completed. This attribute can 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 can 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 can be added just for using the session's secret for parameter decryption. This attribute can be SET in combination with any other session attribute.

(continued on next page)

(continued from previous page)

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 18). 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 can 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 can be added just for using the session's secret for parameter decryption. This attribute can be SET in combination with any other session attribute.
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 can 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 concatenation of cpHash for the command and rpHash for the response. This attribute will be echoed by the TPM in the response. This attribute can be used in combination with any other session attributes but only one session in each command can have this attribute SET.

(continued on next page)

(continued from previous page)

Attribute	Meaning
auditExclusive	This attribute is used 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. NOTE 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 below). If the session was used for all auditable commands, then it is said to be “exclusive” (see Clause 17.2 for an 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 can only be SET when the *audit* attribute is SET which excludes this attribute from being SET on a password authorization or a policy session.
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 can only be SET if audit is SET. The TPM will echo this attribute in the response.

15.7 Command Parameter Hash

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* can also be used in the authorization). The *cpHash* is computed from the parameters of the command as follows:

$$cpHash := \mathbf{H}_{sessionAlg}(commandCode \{\|\ Name1 \{\|\ Name2 \{\|\ Name3\}\}\} \{\|\ parameters\}) \quad (15)$$

where

$\mathbf{H}_{sessionAlg}$	is the hash function using the algorithm selected for the session when it was initialized
$commandCode$	is the command code for the command
$Name1$	is the unique identity of the entity associated with the first handle

(continued on next page)

(continued from previous page)

Name2	is the unique identity of the entity associated with the second handle
Name3	is the unique identity of the entity associated with the third handle
parameters	is the remaining command parameters

15.8 Response Parameter Hash

The response parameter hash (*rpHash*) 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:

$$rpHash := \mathbf{H}_{sessionAlg}(responseCode \parallel commandCode \{\parallel parameters\}) \quad (16)$$

where

$\mathbf{H}_{sessionAlg}$	is the hash function using the algorithm selected for the session when it was initialized
responseCode	is the command result code
commandCode	is the commandCode from the command
parameters	is the 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_RC_SUCCESS, which means that it is redundant to include the response code. It is retained for legacy reasons.

15.9 Command Example

Table 16 shows an example of a command schematic used in this specification. 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 can 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.

Table 16: Command Layout for Example Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Example
Handles		
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
Parameters		
UINT32	dataSize	example data size
OCTET	data[dataSize]	example data

Table 17 illustrates all command octets for the command in Table 16. In this example, the nonce size is 20 octets and the authorization HMAC is computed using SHA256. Some parameters below are not shown in the TPM 2.0 Part 3 schematic of the command but are included in the command data sent to the TPM.

Table 17: Example Command Showing authorizationSize

Offset	Size	Parameter	Value
0	2	tag	TPM_ST_SESSIONS
2	4	commandSize	**209**
6	4	commandCode	TPM_CC_Example
Handles			
10	4	handleA	
14	4	handleB	
18	4	*authorizationSize*	**61**

(continued on next page)

(continued from previous page)

Offset	Size	Parameter	Value
22	4	*authHandle*	
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>
Parameters			
83	2	dataSize	**32**
85	124	data[dataSize]	<124 octet buffer>
209			

15.10 Response Example

Table 18 shows an example schematic as it would appear in Part 3. The example is for a response sent from the TPM after successful completion of the example command in Table 16. The response has the same number of sessions in the same order as did the command.

Table 18: Response Layout for Example Command

Type	Name	Description
TPM_ST	tag	TPM_ST_SESSIONS
UINT32	responseSize	
TPM_RC	responseCode	response code of the operation
Handles		
TPM_HANDLE	handle	not included in the rpHash
Parameters		
UINT32	dataSize	size in octets of the following data
OCTET	data[dataSize]	a returned block of information

Table 19 illustrates the full response for the command in Table 16. As in the command, the nonce size is 20 octets and the acknowledgment HMAC is computed using SHA256. Some parameters below are not shown in the TPM 2.0 Part 3 schematic of the response but are present in the response data from the TPM.

Table 19: Example Response Showing parameterSize

Offset	Size	Parameter	Value
0	2	tag	TPM_ST_SESSIONS
2	4	responseSize	**203** <size in octets of the response>

(continued on next page)

(continued from previous page)

Offset	Size	Parameter	Value
6	4	responseCode	**0** <success>
Handles			
10	4	handle	<a valid TPM_HANDLE>
14	4	*parameterSize*	**128**
Parameters			
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			

16 Authorizations and Acknowledgments

16.1 Introduction

Many commands to the TPM reference TPM-resident structures and use of these structures can 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, this specification defines three authorization types:

1. password;
2. HMAC; and
3. policy.

Depending on the command, zero, one, or two authorizations can 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.

16.2 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. The role and attributes determine whether a password or HMAC can be used for authorization. A policy (if not the Empty Policy) can always be used.

- **USER** - this authorization role is used for the normal uses of a key (e.g., signing with a signing key, or loading the child of a Storage 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 can be provided with a password authorization or an HMAC session. If *userWithAuth* is CLEAR, then a password and HMAC authorizations cannot be used to provide USER role authorizations. A policy session that satisfies the *authPolicy* of the entity can be used regardless of the setting of *userWithAuth*.

  Note:

  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:

  If use of an object is to be gated based on PCR values, a policy session is required (see Clause 16.7). 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*.

- **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 can be provided using the *authValue* of the object if the *adminWithPolicy* attribute of the object is CLEAR. As with USER role authorization, ADMIN role can always be provided with a policy session as long as the policy session satisfies the *authPolicy* of the object.

Note:

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:

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.

- **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.

16.3 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 can take any number of forms, such as a dedicated pin in the TPM, a special signaling method through the TPM interface, or any desired alternative. Whatever the form, the out-of-band authorization is referred to in this specification 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 this specification because it was used in previous TPM specifications to refer to a similar concept.

The TPM maintains a table of the commands that require that PP be asserted to authorize command execution. Only certain commands can be included in this table. If, in Part 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 cannot.

Note:

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 may be required if indicated by the table.

Note:

A platform-specific specification may 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() cannot be removed from the list of commands that require PP.

Note:

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 can be read using TPM2_GetCapability(*capability* == TPM_CAP_PP_COMMANDS).

16.4 Password Authorizations

A plaintext password value can be used to authorize an action when use of an *authValue* is allowed. A plaintext password can 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:

While it can 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. *sessionAttributes→continueSession* is ignored.

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 can be used for more than one authorization.

Password authorization data sent to the TPM has the format shown in Table 20.

Table 20: 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	nonceCaller	required to be an Empty Buffer
TPMA_SESSION	sessionAttributes	only *continueSession* can be SET
TPM2B_AUTH	password	authorization compared to the *authValue* of the TPM entity

Table 21 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.

Table 21: 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, *continueSession* will be SET
TPM2B_AUTH	hmac	zero-length buffer for a password authorization

Note:

This structure is used to provide symmetry between password and other response sessions.

16.5 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 can exist both on the TPM and in saved contexts (see Clause 27). When a session context is created, only one copy of that context can 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 can 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 can also be used to provide keys for encryption of command or response parameters. A policy session used to authorize cannot also be used as an audit session. An HMAC session used to authorize can 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 can also be used to provide secrets for encryption of command or response parameters and can be used for authorization of an HMAC session.
3. **encryption** - A session that is not used for authorization or audit can be present for the purpose of encrypting command or response parameters. If an encryption-only session exists, it will follow the authorization sessions and can come before or after a session used only for audit.

A command can have as many as three authorization blocks. All three uses require an authorization session in the authorization structure as shown in Clause 15.6.2 (including the HMAC authorization field), and Table 17 and Table 19. Password blocks can only be used for authorization, so the maximum number of password blocks is equal to the number of authorizations required by the command.

16.6 Session-Based Authorizations

16.6.1 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 can 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.

Note:

Using the endorsement key for which the certificate chain has been validated as the salt key can ensure that the caller is connected to an authentic TPM.

16.6.2 Authorization Session Formats

For a session-based authorization session, the authorization structure for a command is as shown in Table 22.

Table 22: 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 23.

Table 23: 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.

(continued on next page)

(continued from previous page)

Type	Name	Description
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

16.6.3 Session Nonces

16.6.3.1 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.

16.6.3.2 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 may allow larger nonce sizes but the caller cannot 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.

16.6.3.3 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* can have a long lifetime. To prevent replay attacks on a long-lived *authValue,* use of large nonces is recommended.

Note:

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:

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 has to 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.

16.6.3.4 Nonce Binding

A command can 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.

16.6.4 Authorization Values

16.6.4.1 Overview

An object can 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* can be sent as a password that does not provide confidentiality (see Clause 16.4), or in an HMAC-based authorization session that can provide confidentiality of the *authValue*.

16.6.4.2 authValue Size

An *authValue* can 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.

For hierarchies, `TPM2_HierarchyChangeAuth()`, the authorization value can be no larger than the digest produced by the hash algorithm used for context integrity.

For `TPM2_HMAC_Start()`, `TPM2_MAC_Start()`, and `TPM2_HashSequenceStart()`, the authorization value can be no larger than the largest digest produced by any hash implemented on the TPM.

16.6.4.3 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 largest value in Clause 16.6.4.2.

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*.

By convention, if the password/phrase, with trailing zeros removed, is larger than the largest value in Clause 16.6.4.2, then the password/phrase - with trailing octets of zero removed - is hashed using

- *nameAlg* for entities with a Name algorithm
- the hash algorithm used for context integrity for entities lacking a Name algorithm

and the resulting hash given to the TPM as the *authValue* for the object. The TPM does not enforce this transformation.

16.6.5 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 *sessionKey* and *authValue* are both the Empty Buffer, see Clause 16.6.16.

$$data := (pHash \parallel nonceNewer \parallel nonceOlder \{\parallel nonceTPM_{\text{decrypt}}\} \{\parallel nonceTPM_{\text{encrypt}}\} \parallel sessionAttributes)$$

$$authHMAC := \mathbf{HMAC}_{sessionAlg}((sessionKey \parallel authValue), data) \quad (17)$$

where

$\mathbf{HMAC}_{sessionAlg}$	is the HMAC function using the hash algorithm specified when the session was started
$sessionKey$	is 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.

(continued on next page)

(continued from previous page)

authValue is 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:***
>
> For policy sessions, the *authValue* is not included in the HMAC calculation unless the policy session include `TPM2_PolicyAuthValue()` and it was not superseded by `TPM2_PolicyPassword()`.

> ***Note:***
>
> Trailing zeros are always removed from an *authValue* before it is used in an authorization computation.

pHash is the digest of the command (cpHash) or response parameters (rpHash) using the session hash algorithm.

nonceNewer is 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 is 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 Clause 16.6.3.4). The nonce size field is not included in the HMAC.

> ***Note:***
>
> The *decrypt* session is used by the TPM to decrypt a parameter in the command.

> ***Note:***
>
> The nonce of the *decrypt* session is included even if that session is also used for authorization.

(continued on next page)

(continued from previous page)

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 Clause 16.6.3.4). The nonce size field is not included in the HMAC.

> ***Note:***
>
> The *encrypt* session is used by the TPM to encrypt a parameter in the response.

> ***Note:***
>
> The nonce of the *encrypt* session is included even if that session is also used for authorization.

> ***Note:***
>
> If the same session (not the first session) is used for decrypt and encrypt, its *nonceTPM* is only used once. If different sessions are used for decrypt and encrypt, both *nonceTPM*s are included.

sessionAttributes — is 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 can include *nonceTPM*$_{\text{decrypt}}$ or *nonceTPM*$_{\text{encrypt}}$ is indicated by "*nonceOlder**" (asterisk added).

16.6.6 Note on Use of Nonces in HMAC Computations

In Equation 17, 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.

16.6.7 Starting an Authorization Session

`TPM2_StartAuthSession()` is used to start an authorization session. The parameters of this command can be chosen to produce sessions with different properties.

Table 24: Schematic of TPM2_StartAuthSession() Command

Type	Name	Description
TPM_ST	tag	

(continued on next page)

(continued from previous page)

Type	Name	Description
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StartAuthSession
Handles		
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
Parameters		
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 can provide an *authValue* to the computation for the *sessionKey*. The four variations for *tpmKey* and *bind* give sessions with different properties.

Table 25: Handle Parameters for TPM2_StartAuthSession()

tpmKey	bind	session properties
TPM_RH_NULL	TPM_RH_NULL	Unbound and unsalted session
TPM_RH_NULL	TPM entity	Bound session
TPM key	TPM_RH_NULL	Salted session
TPM key	TPM entity	Salted and bound session

16.6.8 *sessionKey* Creation

A *sessionKey* value is used in the HMAC computation as shown in Equation 17. If both *tpmKey* and *bind* are TPM_RH_NULL, then *sessionKey* is set to an Empty Buffer. Otherwise, the *sessionKey* is created as follows:

$$sessionKey := \mathbf{KDFa}(sessionAlg, (authValue \parallel salt), \text{"ATH"}, nonceTPM, nonceCaller, bits) \quad (18)$$

where

sessionAlg	is 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"	is a four-octet label value
nonceTPM	is a TPM2B_NONCE that is generated by the TPM when the session was started
nonceCaller	is a TPM2B_NONCE that is provided by the caller when the session was started.
bits	is the number of bits returned in 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 Clause 16.8).

16.6.9 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.

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:

$$data := (pHash \parallel nonceNewer.buffer \parallel nonceOlder*.buffer \parallel sessionAttributes)$$

$$authHMAC := \mathbf{HMAC}_{sessionAlg}(authValue_{entity.buffer}, data) \quad (19)$$

If the size of *authValue* is zero, then the caller may omit the HMAC from the authorization (see Clause 16.6.16).

Table 26: Format to Start Unbound, 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	0x00 00
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.

16.6.10 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*.

The *sessionKey* is computed using the *authValue* from *bind* and an Empty Buffer in place of the *salt* value.

$$sessionKey := \mathbf{KDFa}(sessionAlg,\ authValue_{bind},\ \text{"ATH"},\ nonceTPM,\ nonceCaller,\ bits) \tag{20}$$

Note:

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 calculated as follows:

1. When the session is an HMAC session:
    1. If the authorization is not for the entity to which the session is bound, the HMAC key is the concatenation of the entity's *authValue* to the session's *sessionKey* (created at `TPM2_StartAuthSession()` (see Equation 21).
    2. If the authorization is for the entity to which the session is bound, the HMAC key is the session's *sessionKey* (created at `TPM2_StartAuthSession()` (see Equation 22).

2. When the session is a policy session:

    1. If the session has *isAuthValueNeeded* SET (by `TPM_PolicyAuthValue()`), the HMAC key is the concatenation of the entity's *authValue* to the session's *sessionKey* (see Equation 21).
    2. If the session has *isAuthValueNeeded* CLEAR, the HMAC key is the session's *sessionKey* (created at `TPM2_StartAuthSession()` (see Equation 22).

$$data := (pHash \parallel nonceNewer \parallel nonceOlder* \parallel sessionAttributes)$$

$$authHMAC := \mathbf{HMAC}_{sessionAlg}((sessionKey \parallel authValue_{entity}),\ data) \tag{21}$$

$$authHMAC := \mathbf{HMAC}_{sessionAlg}(sessionKey,\ data) \tag{22}$$

Note:

Binding to an entity different from the one being authorized is a way of adding entropy to the session key. It is useful in cases where the entity being authorized has a low entropy authorization value.

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:

In the Reference Code, the authorization value is combined with the Name and stored in the SESSION→boundEntity member.

Note:

Recording of the NV Index authorization is required 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:

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:

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 entity is recorded in the session context. For a description of the rationale, see Clause 16.8.7.

Table 27: 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 0016
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

16.6.11 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*.

$$sessionKey := \mathbf{KDFa}(sessionAlg,\ salt,\ \text{"ATH"},\ nonceTPM,\ nonceCaller,\ bits) \tag{23}$$

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.

$$data := (pHash \parallel nonceNewer \parallel nonceOlder* \parallel sessionAttributes)$$

$$authHMAC := \mathbf{HMAC}_{sessionAlg}((sessionKey \parallel authValue_{entity}),\ data) \tag{24}$$

Table 28: 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

16.6.12 Salted and Bound Session Key Generation

This version of `TPM2_StartAuthSession()` creates a session that has properties that are similar to the session type in Clause 16.6.10 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.

$$sessionKey := \mathbf{KDFa}(sessionAlg, (authValue_{bind} \parallel salt), \text{"ATH"}, nonceTPM, nonceCaller, bits) \quad (25)$$

If the session is an HMAC session:

- If the authorization is not for the entity to which the session is bound, the HMAC key is the concatenation of the entity's *authValue* to the session's *sessionKey* (created at `TPM2_StartAuthSession()` (see Equation 26).
- If the authorization is for the entity to which the session is bound, the HMAC key is the session's *sessionKey* (created at `TPM2_StartAuthSession()` (see Equation 27).

If the session is a policy session:

- If the session has *isAuthValueNeeded* SET (by `TPM_PolicyAuthValue()`), the HMAC key is the concatenation of the entity's *authValue* to the session's sessionKey (see Equation 26).
- If the session has *isAuthValueNeeded* CLEAR, the HMAC key is the session's *sessionKey* (created by `TPM2_StartAuthSession()` (see Equation 27).

$$data := (pHash \parallel nonceNewer \parallel nonceOlder* \parallel sessionAttributes)$$

$$authHMAC := \mathbf{HMAC}_{sessionAlg}((sessionKey \parallel authValue_{entity}),\ data) \quad (26)$$

$$authHMAC := \mathbf{HMAC}_{sessionAlg}(sessionKey,\ data) \quad (27)$$

- The *noDA* attribute of the bind entity is recorded in the session context. For a description of the rationale, see Clause 16.8.7.

Table 29: 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

16.6.13 Encapsulation of *salt*

16.6.13.1 Overview

The *salt* parameter for `TPM2_StartAuthSession()` is encapsulated to the TPM using the methods described in this clause.

When the value of *salt* is determined, it is used in the computation of *sessionKey* as shown in Equation 18.

As illustrated in Figure 10, the salt is the shared-secret output of a Labeled KEM as discussed in Clause 8.4.5.2. The Labeled KEM protocol label for an encrypted salt is “SECRET”.

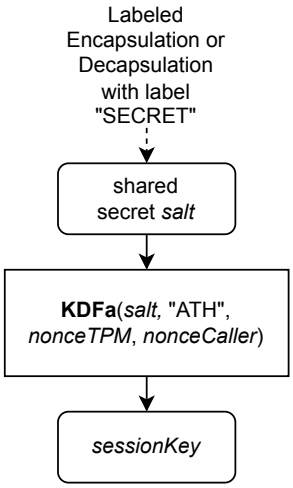


Figure 10: Encapsulation of Salt

The TPM decapsulates the shared secret as *salt* as used in the calculation of *sessionKey* in Clause 16.6.11 and Clause 16.6.12.

16.6.14 Caution on use of Unrestricted Salt Keys

If an unrestricted *tpmKey* is used for salted session generation, then the encapsulated *salt* may be recoverable by a user or attacker that can call a decryption primitive (e.g., `TPM2_RSA_Decrypt()` or `TPM2_ECDH_ZGen()`). Users are urged to only use restricted keys for salted sessions.

The ability to use an unrestricted key for salted sessions is deprecated. See Part 0.

16.6.15 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:

$$data := (rpHash \parallel nonceTPM \parallel nonceCaller \parallel sessionAttributes)$$

$$authHMAC := \mathbf{HMAC}_{sessionAlg}(authValue, data) \quad (28)$$

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:

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:

$$data := (pHash \parallel nonceNewer \parallel nonceOlder \parallel sessionAttributes)$$

$$authHMAC := \mathbf{HMAC}_{sessionAlg}((sessionKey \parallel authValue_{entity}), data) \quad (29)$$

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:

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. For instance, if the *bind* entity *authValue* has high-entropy, then there would be greater computational complexity in guessing *sessionKey* $\parallel$ $authValue_{entity}$. 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.

16.6.16 No HMAC Authorization

For a session-based authorization, both HMAC and policy, an *authHMAC* value is computed as shown in Equation 17 and that value is used as *hmac* in an authorization or acknowledgement as shown in Table 22 and Table 23 respectively. If an authorization session is started with *bind* and *tpmKey* both set to `TPM_RH_NULL`, then *sessionKey* in Equation 17 will be an Empty Buffer. If the *authValue* in Equation 17 is also an Empty Buffer, 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 17 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 22) 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 17 will only be an Empty Buffer if the *authValue* of the authorized object is an EmptyAuth, the session is a bind session and the authorization is for the entity to which the session is bound, or if the session is not an authorization session.

For a policy session, two situations will result in *authValue* being an Empty Buffer:

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 Clause 16.7.7.6).

For these two cases, if *sessionKey* is an Empty Buffer, *hmac* is allowed to be either a valid *authHMAC* or an Empty Buffer. For a bound or salted policy session, *sessionKey* is not an Empty Buffer, and *hmac* must be valid.

Note:

A policy session that does not use TPM2_PolicyAuthValue() would use a bound or salted session if that session is also used for encryption.

For a policy session that contains TPM2_PolicyPassword(), the password takes precedence and must be present in *hmac*.

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 17 and use the result as *hmac*. If *hmac* was an Empty Buffer in the command, it will be an Empty Buffer in the response.

16.6.17 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 can be provided with an HMAC session or a password. If this attribute is CLEAR, then the *authValue* cannot be used for USER role authorization, meaning that authorization cannot 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 can only be provided with a policy session. If this attribute is CLEAR, then authorization can 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 can simply assume that the attributes are SET and act accordingly.

16.6.18 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.

- on TPM Reset, all authorization sessions are terminated; and
- on TPM Resume or TPM Restart, authorization sessions in TPM memory will be terminated but sessions context saved off the TPM will remain active.

16.7 Enhanced Authorization

16.7.1 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 can have an associated *authPolicy*. A policy defines the conditions for use of an entity. For example,

- a policy can limit the use of a key unless selected PCR have specific values;
- a policy cannot allow use of a key after a specific time;
- a policy can require that authorization to change an NV Index be provided by two different entities; or
- a policy can limit a particular signing key to attest to PCR values but not to certify another TPM key.

A policy can be arbitrarily complex. However, the policy is expressed as one (statistically unique) digest called the *authPolicy*.

The digest representing a particular policy can 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 described in subsequent parts of this clause). 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.

Note:

The entity digest algorithm uses the entity Name algorithm. The session digest algorithm is set at `TPM2_StartAuthSession()`. For the policy to be satisfied, the two algorithms must match.

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 Clause 16.7.8).

The mechanisms of policy creation and evaluation are explained in the remainder of this clause

16.7.2 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. For example, an assertion can be that a set of PCR must have specific values to allow an object to be authorized for use in a specific command. The list of all policy assertions defined by this specification is in Clause 16.7.7.6.

A combination of one or more assertions is used to construct an authorization policy.

16.7.3 Policy AND

A policy can be expressed in an equation as a set of assertions that must all be satisfied before the policy is valid. For example, a policy that requires that 4 assertions be true could be written as:

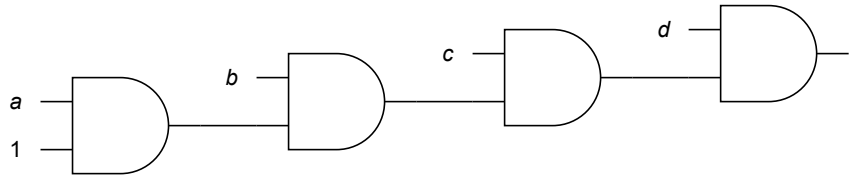
a & *b* & *c* & *d*

A possible implementation of the policy logic 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.

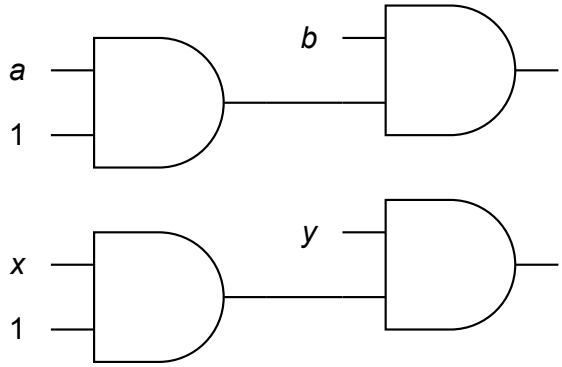
(((1 & *a*) & *b*) & *c*) & *d*

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:



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. For example, these two policies can both evaluate to the same logic value (TRUE), but they do not represent the same policies.



So that it can differentiate (*a* & *b*) from (*x* & *y*), 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.

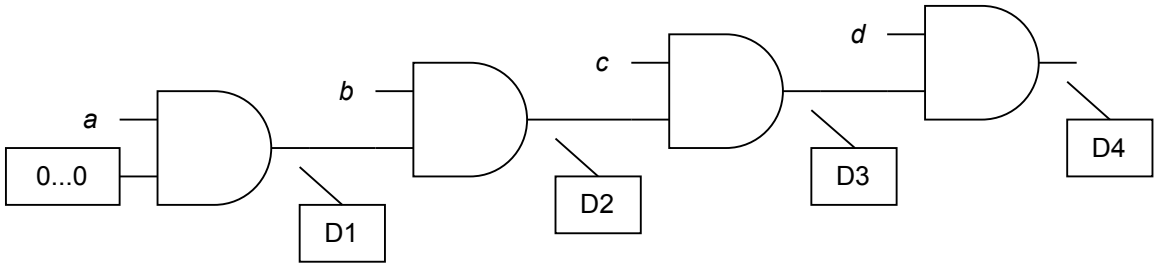
The running digest value is called the *policyDigest*. The *policyDigest* is initialized to a Zero Digest (0…0) when the policy session is started (`TPM2_StartAuthSession()`). Then, as each policy assertion is evaluated, the *policyDigest* is updated.

$$policyDigest_{\text{new}} := \mathbf{H}(policyDigest_{\text{old}} \parallel PolicyAssertion)$$

Note:

This is essentially an Extend operation.

The *policyDigest* will only be updated if a policy assertion is valid (TRUE) (see Clause 16.7.10 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 becomes:



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:

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.

16.7.4 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:

(*a* & *b*) | (*x* & y)

or as:

(((0 ... 0) & *a*) & *b*) | (((0 ... 0) & *x*) & *y*)

Evaluating the AND branches individually, the left side evaluates to:

$$D_{left} := \mathbf{H}(\mathbf{H}(0...0 \parallel a) \parallel b)$$

and the right side to:

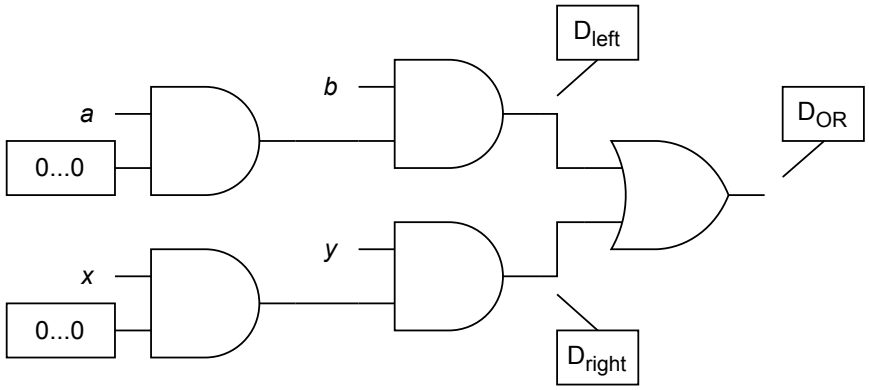
$$D_{right} := \mathbf{H}(\mathbf{H}(0...0 \parallel x) \parallel y)$$

Then, the output from a 2-input policy OR operation will be defined to be

$$policyDigest_{\text{new}} := \mathbf{H}(D_{left} \parallel D_{right})$$

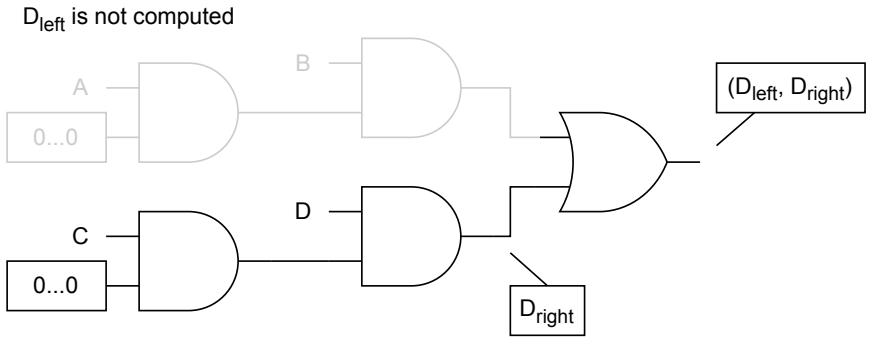
Notice that the OR operation replaces the *policyDigest* with a new value instead of Extending it as is done in an AND operation.

Pictorially, a policy with an OR is:



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. For example, 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*. Note that 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. For example, if C and D assertions were valid, then only the *right* branch would need to be 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. Figure 11 illustrates one of the many ways to construct a 12 input OR.

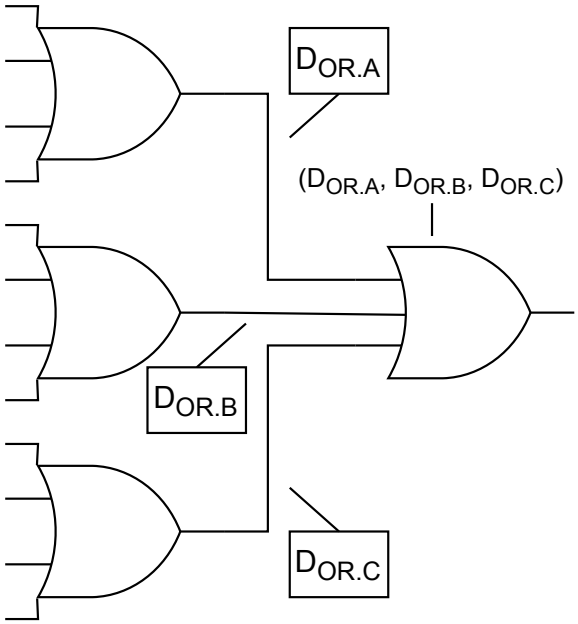


Figure 11: A 12-input OR Policy

When the OR list can contain 8 digests, 64 different branches can be ORed in just two levels.

The result of an OR operation can be an input to an AND assertion allowing construction of arbitrarily complex policies.

16.7.5 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}).

16.7.6 Policy Session Creation

`TPM2_StartAuthSession`(*sessionType* == `TPM_SE_POLICY`) is used to start an authorization session. The authorization session can use any of the four options for *tpmKey* and *bind.*

Note:

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 can be Empty Buffers (see TPM 2.0 Part 3, `TPM2_PolicyAuthValue()`).

Note:

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_StartAuthSession()` that starts the session so that a secure *sessionKey* is created.

16.7.7 Policy Assertions (Policy Commands)

16.7.7.1 Introduction

In Part 3 of the specification 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. For example, TPM2_PolicySigned() is a policy assertion that an authorization was signed by a specific entity; and 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. For example, TPM2_PolicyLocality() modifies the policy state that indicates the locality that is allowed when the policy session is used for authorization. TPM2_PolicyCommandCode() changes the policy state so that the policy can 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 Part 3. The description also indicates the policy state that is modified.

The assertions fall into three different groups: immediate, deferred, and combined.

16.7.7.2 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. 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.

16.7.7.3 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. 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.

16.7.7.4 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. 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 can 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.

16.7.7.5 Repetition of Assertions

In general, any policy assertion can occur multiple times within a policy as long as the assertion is compatible with previous assertions. An example of an incompatible set of assertions is two occurrences of TPM2_PolicyCommandCode() that indicate 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. For example, a TPM2_PolicyCpHash()

can be incompatible with a TPM2_PolicySigned(). If they indicate 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.

16.7.7.6 List of Assertions

The assertions listed in this clause will all update the *policyDigest* of the policy session being operated on if the assertion condition is met. They can also cause a change to other policy session, context values (the list of policy session context values is in Clause 16.7.8) 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 described in more detail in Clause 16.7.11.
- **TPM2_PolicyAuthorizeNV()** - valid if the specified NV Index contains a hash algorithm identifier and a digest value that match the hash algorithm and *policySession→policyDigest*. This immediate assertion changes *policySession→policyDigest* and is described in more detail in Clause 16.7.11.
- **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 a 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* (see Clause 33 for 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_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*.
- **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:

  A session can use TPM2_PolicyAuthValue() and TPM2_PolicyPassword() interchangeably. If TPM2_PolicyAuthValue() and TPM2_PolicyPassword() 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 can be either a 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:

  The Reference Code 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 can modify *policySession→cpHash* and *policySession→timeout.*

  Note:

  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 can modify *policySession→cpHash* and *policySession→timeout.*
- **TPM2_PolicyTemplate()** - valid if the hash of the *inPublic* parameter of TPM2_Create(), TPM2_CreatePrimary(), or TPM2_CreateLoaded() matches the *templateHash* in this command. This deferred assertion sets *policySession→cpHash.*

- **`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 can modify *policySession→cpHash* and *policySession→timeout*.
- **`TPM2_PolicyTransportSPDM()`** - valid if the command being authorized is protected by an SPDM secure channel. Optionally, the policy may be tied to a specified requester and/or TPM secure channel key. This deferred assertion sets *policySession→checkSecureChannel*, and optionally *policySession→checkReqKey*, *policySession→checkTpmKey* and *policySession→scKeyNameHash*.

16.7.8 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 can be changed by a policy assertion. The policy assertions are listed in Clause 16.7.7.6 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 policy 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→nameHash* 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.
- ***templateHash*** - set by `TPM2_PolicyTemplate()` and binds the policy to a specific creation template. This context parameter occupies the same location as *policySession→cpHash*. If an assertion would set *policySession→templateHash* and a previous assertion has set *cpHash* to a different value, then the policy assertion will fail. The default for *policySession→templateHash* is an Empty Buffer.
- ***boundEntity*** - set by `TPM2_StartAuthSession()` to track the entity to which the session is bound. The default for *policySession→boundEntity* 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 can 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 described in Clause 14.9). 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 cannot 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.
- ***checkSecureChannel*** - SET by TPM2_PolicyTransportSPDM() to indicate that the command being authorized by the policy must be protected by an SPDM secure channel. The default value is CLEAR.
- ***checkReqKey*** - SET by TPM2_PolicyTransportSPDM() to indicate that the secure channel has to be established with a specific requester key. This attribute has no meaning when ***checkSecureChannel*** is not SET. The default value is CLEAR.
- ***checkTpmKey*** - SET by TPM2_PolicyTransportSPDM() to indicate that the secure channel has to be established with a specific TPM key. This attribute has no meaning when ***checkSecureChannel*** is not SET. The default value is CLEAR.
- ***scKeyNameHash*** - set by TPM2_PolicyTransportSPDM() to track the Names of the required requester and/or TPM secure channel key. This attribute has no meaning when neither ***checkReqKey*** nor ***checkTpmKey*** is SET. The default value is an Empty Buffer.

16.7.9 Policy Example

A TPM 2.0 policy that checks PCR state in addition to requiring the caller to prove knowledge of the authorization value of the object being used 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*.

16.7.10 Trial Policy

The policy evaluation to determine the value for the *authPolicy* can be done in software that does the same *policyDigest* computation as the TPM. Alternatively, a trial policy session can 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 cannot be used for authorization.

16.7.11 Modification of Policies

Some policies, such as those associated with the hierarchies, can be altered directly by changing the *authPolicy* value. Policies associated with Objects and NV Indices cannot be directly altered. The reason that these policies cannot be altered is that the policy can affect the trust that someone places in the use of that entity. For example, a key can only be trusted if it can 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.

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. 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 TPM 2.0 if the policy was completely fixed.

Figure 12 and Figure 13 illustrate 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 Clause 16.7.12).

Note:

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:

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 can provide other assurances but, in the case of PCR, it is not necessary to warrant anything other than that the PCR values are expected.

An example of how of this assertion type can be used to avoid PCR brittleness is shown in Figure 12 and Figure 13. This shows the example policy in Clause 16.7.9 but with the ability to satisfy the policy with different PCR values.

Note:

The actual *authPolicy* in the authorized entity would contain (`TPM2_PolicyAuthorize()` AND `TPM2_PolicyAuthValue()`).

As shown, a `TPM2_PolicyPCR()` assertion is followed by `TPM2_PolicyAuthorize()`. If there is an authorization signed by KEY for the current *policyDigest* (in this case, $D_{PCR.A}$), then the result of the `TPM2_PolicyAuthorize()` will be D_{KEY}. This is the same output that would be produced if the input to the `TPM2_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* reset to a Zero Digest and then extended with the Name of the key. The *policyDigest* value D_{final} no longer reflects the previous value ($D_{PCR.A}$ or $D_{PCR.B}$).

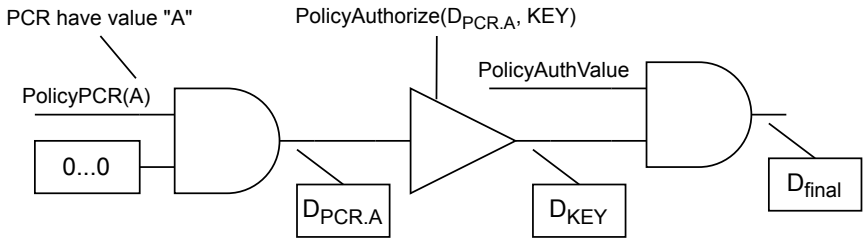


Figure 12: Use of PolicyAuthorize to Avoid PCR Brittleness (PCR A)

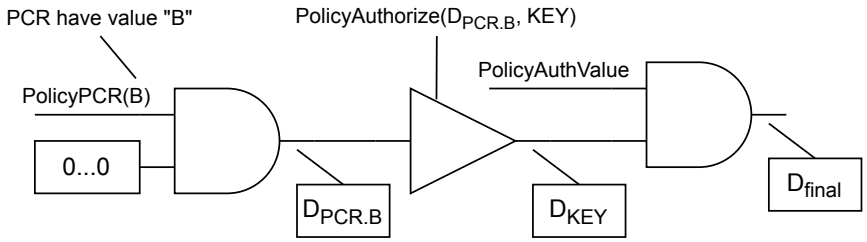


Figure 13: Use of PolicyAuthorize to Avoid PCR Brittleness (PCR B)

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.

An additional way to indirectly modify a policy uses `TPM2_PolicyAuthorizeNV()`. This command indicates an NV Index location of a policy that will be effective for an entity, the effective policy being the policy digest stored in the data at that NV index. When a policy is formed using `TPM2_PolicyAuthorizeNV()`, the NV Index Name is specified. When the entity is to be authorized, the policy stored in the data of the named index is satisfied, and then `TPM2_PolicyAuthorizeNV()` is executed. If the policy stored in that index matches the *policySession→policyDigest*, then the *policySession→policyDigest* is replaced with results of first setting the *policySession→policyDigest* to the Zero Digest, and then extending it with the command code concatenated with the Name of the NV index, which will contain the modified policy. The main difference between this

command and TPM2_PolicyAuthorize() is that multiple policies can satisfy TPM2_PolicyAuthorize(), as long as they are all signed with the appropriate key. Only the policy that is currently stored in the NV index can satisfy TPM2_PolicyAuthorizeNV().

16.7.12 TPM2_PolicySigned(), TPM2_PolicySecret(), and TPM2_PolicyTicket()

The set of assertions discussed in this clause have properties that enable a number of authorization scenarios. Among these are:

- the ability to grant an authorization that can persist for a specific amount of time (because in many protocols, access to a resource (such as, a network) is granted for some time interval); and
- the ability to associate an authorization with a policy of the authorizing entity (because in many instances, the authorizing entity can 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.

The policy assertions of TPM2_PolicySigned() and TPM2_PolicySecret() can be time limited. When a policy's authorization is time limited, it expires (is no longer valid) when TPM *Time* is greater than the indicated value for the authorization. An expiration time can be expressed in two ways. A *timeout* is an absolute value of *Time*. An *expiration* value is a relative value in seconds from the start *Time* of the policy session to which the authorization applies.

Both TPM2_PolicySigned() and TPM2_PolicySecret() can produce tickets that enable authorizations to be used and reused over a period of time and in different policy sessions. These tickets are used in TPM2_PolicyTicket().

These three commands have several input parameters in common:

- **nonceTPM** - the value returned by TPM2_StartAuthSession() or when a session is used for an authorization. It is used to limit the use of a policy assertion to a specific policy session. If a policy command includes a *nonceTPM,* then the TPM will return an error if it does not match *policySession→nonceTPM*.
- **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. For example, a fingerprint reader can 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. The TPM includes this value in the *policyDigest*. In the example of the fingerprint reader, this would mean that the *policyDigest* would only have the correct value if the fingerprint reader scanned a finger from the correct person. If this parameter is not part of the signed authorization, then this parameter should be an Empty Buffer.

  Note:

  Because TPM2_PolicySigned() does not include the *cpHashA* and *policyRef* size in the *aHash* calculation, it is recommended that the size of *policyRef* not be the same size as that of the entity Name algorithm.

- **expiration** this parameter is used to place a time limit on an authorization. It is either the number of seconds from the last time that the *nonceTPM* of a policy session was changed, or the value of Time after which a policy assertion is no longer valid.
- **timeout -** this indicates the value of *Time* after which a policy assertion is no longer valid.

If a TPM2_PolicySigned() or TPM2_PolicySecret() has a non-zero *expiration* parameter:

- If *nonceTPM* is not included, *expiration* is a *timeout.*
- If expiration is a negative number, a ticket will be produced.

Note:

For TPM2_PolicySecret(), if *authHandle* references a PIN Pass Index, then no ticket will be produced even if *expiration* is negative. This prevents use of a ticket to bypass the limit count on an PIN Pass Index.

When a policy session is started, a *nonceTPM* is generated and the current value of *Time* is copied to *policySession→startTime*. When a policy assertion includes a non-zero *expiration* and a *nonceTPM*, then *policySession→startTime* is added to the absolute value of *expiration* to determine the *timeout* for the policy assertion. If a policy assertion includes a non-zero value in its *expiration* parameter but no *nonceTPM*, then the *expiration* parameter is used directly as a *timeout*

When an assertion produces a *timeout*, the *timeout* value is placed in *policySession→timeout.* If *policySession→timeout* has previously been set, then it will be updated with the lesser of *timeout* and *policySession→timeout*.

When *Time* has a greater value than *policySession→timeout,* the policy session expires and cannot be used for authorization. If the authorization used a *timeout* (no *nonceTPM*), then the authorization will also expire on the next TPM Reset.

Note:

A policy assertion **includes** an *expiration* when the *expiration* parameter is non-zero. A policy assertion **includes** a *nonceTPM* when its *nonceTPM* parameter is not the Empty Buffer.

expiration may need to be converted to milliseconds before being added to *policySession→startTime.*

When an *expiration* parameter is used directly as a *timeout, expiration* is the value of Time in seconds when the assertion expires. When an assertion contains a *timeout* parameter (only in TPM2_PolicyTicket()), *timeout* is an implementation-dependent value.

A ticket contains a digest of the command parameters of the assertion along with a ticket *timeout*. As long as a ticket has not expired, its effect on a *policySession→policyDigest* and *policySession→timeout* will be the same as the TPM2_PolicySigned() or TPM2_PolicySecret() command that generated the ticket.

Example:

For example, one can use a TPM2_PolicySigned() command with an expiration of -3600 (the negative of the number of seconds in an hour) to return a ticket. For the next hour, that ticket can be used with TPM2_PolicyTicket() to grant whatever other permissions were approved by the TPM2_PolicySigned() command. The ticket will also expire on a TPM Reset.

When the TPM is not able to report the passage of time (*Time* does not advance), accurate timing of assertions is not possible. To prevent having a timed assertion persist past the intended timeout, a TPM is required to invalidate any time-based assertion that was created before a discontinuity in the TPM's measurement of time. Such a discontinuity can occur when *Time* does not advance or when *Time* is reset. This requirement is met by having a number (a counter or a nonce) that changes each time that there is a time discontinuity (an epoch*)* and

by including *timeEpoch* in the computation of time-based assertions. This implies that each policy session will need to:

- record *timeEpoch* when the session is created (in *policySession→timeEpoch*);
- validate that the *timeEpoch* associated with a time-limited assertion is the same as *policySession→timeEpoch* before the assertion is accepted; and
- when a time-limited policy is used for authorization, verify that the current TPM *timeEpoch* matches *policySession→timeEpoch.*

If a counter is used for *timeEpoch*, it needs to be saved in NV memory whenever it changes. If the number used for *timeEpoch* is a nonce, it can be kept in RAM and changed on each time discontinuity.

Note:

A *timeEpoch* nonce needs to be large enough that a replay is infeasible. That is, a ticket issued with a given nonce will not 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.

16.7.13 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()` can then be used to read the final *policyDigest* from the TPM. That *policyDigest* value can 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 can 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 ensure that they are the same. If desired, `TPM2_PolicyGetDigest()` can be called after each policy command.

16.7.14 Trial Policy Session

If a policy requires a signed (symmetric or asymmetric) authorization for an action, that authorization cannot 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:

If the Object is to have a duplication authorization, the duplication authority cannot 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 cannot 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:

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.

16.7.15 Use of TPM2_PolicySigned() and TPM2_PolicySecret() without *nonceTPM*

The primary purpose of including *nonceTPM* in TPM2_PolicySigned() and TPM2_PolicySecret() is to restrict the use of the policy assertion to a specific policy session so that the assertion can only be used once.

nonceTPM serves a different purpose when the assertion is structured so that a ticket is produced. In that case, the intent is that the assertion can apply to more than one policy session, so *nonceTPM* serves a different purpose - to associate the assertion with a specific TPM. That is, a non-zero expiration causes the TPM to produce a ticket. If the signer did not include *nonceTPM*, its signature could be used with any session and therefore on any TPM. Including *nonceTPM* binds the signature to a specific session, and thus a specific TPM. The signer forces the ticket to be created on a specific TPM, which ties the ticket to a timer on that TPM.

Each *nonceTPM* is expected to be statistically unique and not appear on any other TPM (this is just expected to be true and not required to be true). Therefore, when an assertion includes *nonceTPM,* the assertion will only be usable on one policy session on a specific TPM.

Note:

When a ticket is produced, that ticket is always restricted to use on a specific TPM because of the use of TPM- and hierarchy-specific proof values in the ticket computation.

When the policy assertion does not include *nonceTPM,* then is it possible to use the assertion with any policy session. For TPM2_PolicySecret() the assertion can still be associated with a specific TPM if the authorization for the *authObject* uses an HMAC or policy session. This is because that authorization session will be TPM specific. For TPM2_PolicySigned(), when there is no *nonceTPM* in the assertion, then the assertion can be used on any policy session on any TPM where the public part of *authObject* can be loaded (that is, any TPM that is compatible with this specification). This can be suitable when an assertion is limited to performing specific actions (through *cpHash*) or specific policies (though *policyRef*), but this capability should be used with caution.

16.8 Dictionary Attack Protection

16.8.1 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.

If the entity is authorized in a bind session, it receives DA protection if the bind entity receives 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 can 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* can be used for authorization in three ways:

1. a password;
2. the *authValue* parameter in the HMAC computation of Equation 17; or
3. the *authValue* parameter in the computation of *sessionKey* for a bound session as shown in Equation 18.

All uses of a DA protected *authValue* receive DA protection.

Note:

A TPM PIN Index provides a type of DA protection for an individual TPM entity. This is described in Clause 34.2.8.

16.8.2 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:

The "NV" notation indicates that these values are required to be held in persistent memory and be updated in NV when they change

1. ***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:

If the TPM has a trusted source of time that runs when TPM power is lost, then *failedTries* may 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*.

Note:

The TPM may keep track of the time elapsed toward *recoveryTime* at shutdown and use that against the *recoveryTime* upon power up.

2. ***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.
3. ***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.
4. ***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.

16.8.3 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:

An exception is that `TPM2_DictionaryAttackLockReset()` is allowed to execute even though *lockoutAuth* is DA protected.

Note:

If there is an authorization failure that does not increment *failedTries*, the TPM returns `TPM_RC_BAD_AUTH`

An authorization failure can 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:

A policy authorization does not always have an associated HMAC.

16.8.4 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:

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:

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.

16.8.5 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 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 `TPM2_Startup()` 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*.

16.8.6 Non-orderly Shutdown

A TPM may be implemented such that the command execution unit does not always have access to NV memory (see Clause 34.7.2). 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* may not be necessary if it is impractical for an attacker to prevent update of the NV version of *failedTries*.

An alternative implementation sets an NV flag indicating that access to a DA protected object occurred during this boot cycle. After a non-orderly restart, if the flag is set, the TPM increments *failedTries* and clears the flag. If the flag is clear, there is no need to increment *failedTries*.

Example:

This handles the case where a platform repeatedly does a non-orderly shutdown, possibly due to a low battery. Without the flag, *failedTries* would increment on each reboot and the TPM would go into lockout.

Note:

Some versions of the Reference Code did not correctly implement the behavior described above if a DA protected object is accessed after a `TPM2_Shutdown()`. In this case, the NV flag (indicating that access to a DA protected object occurred during this boot cycle) is not set correctly. When a power loss happens, failedTries is not incremented on the next `TPM2_Startup()`.

The check and increment of failedTries on `TPM2_Startup()` prevents the case that a failed authorization attempt is not recorded by the TPM (because NV memory is unavailable).

16.8.7 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 entity. 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 can 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 entity.

The attack is as follows:

1. An attacker creates an entity (D) that has no DA protection and *authValue* known to the attacker.
2. An attacker guesses a possible *authValue* for a DA protected entity (entity A).
3. The attacker uses entity A as the *bind* entity in `TPM2_StartAuthSession()` to create a session (S).
4. The attacker uses session S to authorize an action on entity D.
5. If the authorization fails, the attacker goes to step 2. and tries a new value.

By retaining the DA state of entity 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.

16.8.8 Sample Configurations for Lockout Parameters

16.8.8.1 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).

16.8.8.2 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, such as one. The enterprise will use this value to recover the TPM when suitable non-automated validation procedures have been performed.

The enterprise would likely set *maxTries* to a relatively low value (such as, 10).

For a server or data center, the *recoveryTime* would be set to a large value (such as, $2^{32} - 1$) implying manual recovery. For laptops, a setting of a few hours would provide adequate protection for PINs.

16.8.8.3 Home or Unmanaged Use

In this application, the *lockoutAuth* can 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.

17 Audit Session

17.1 Introduction

An audit session allows for the auditing of a selected sequence of commands so that evidence can be provided that the commands were executed.

Any HMAC authorization session can be designated for auditing but only one session can be used for audit in each command. A session is designated for auditing by setting the *audit* attribute in the session. A policy session cannot be designated for auditing.

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 Clause 16.6.10 and Clause 16.6.12), 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 can save other session contexts and retry the command.

Note:

The TPM is required to have sufficient memory to allow three sessions to be simultaneously loaded, one of which can 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:

$$auditDigest_{\text{new}} := \mathbf{H}_{auditAlg}(auditDigest_{\text{old}} \parallel cpHash \parallel rpHash) \tag{30}$$

The hash algorithm is the algorithm designated in `TPM2_StartAuthSession()`.

Note:

Audit within an encrypted session will record the encrypted cpHash and/or rpHash, which is unlikely to be useful at the application level.

Unless a command description indicates that no sessions are allowed, an audit session can be used with any command. A command can 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:

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.

For commands that do not require authorization, a bound or salted audit session causes an HMAC based on a shared secret to be generated. This provides assurance that the command was executed on the TPM. 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 can 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.

Note:

This assurance does not require a signed audit digest to be used.

Example:

`TPM2_PCR_Read()` does not require authorization. A bound or salted audit session will cause an HMAC to be used, and thus provide integrity for the command and response.

17.2 Exclusive Audit Sessions

An exclusive audit session permits the TPM to prove that a series of commands were executed with no intervening commands that were not audited by the exclusive audit session.

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. If not, the audit session is exclusive and the *auditExclusive* attribute will be SET.

The TPM keeps track of the current exclusive session. At most, one active session can have the *auditExclusive* status. A session becomes the current exclusive audit session when it is first used as an audit session, regardless of the setting of *auditReset*. It can also become the current exclusive audit session if the *auditReset* attribute of the session is SET in the command. In the response, the *auditExclusive* attribute of the session will be SET and the session is exclusive. 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. *auditReset* can only be SET if *audit* is also SET.

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:

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.

17.3 Command Gating Based on Exclusivity

If the *auditExclusive* attribute of an audit session is SET in the command, then the TPM will return `TPM_RC_EXCLUSIVE` if the audit session is not the current exclusive audit session.

Note:

As with other error returns, no change is made to the state of the session and it remains active.

Note:

auditExclusive in a command only determines whether the command is executed. It does not affect the exclusive status of the session.

Note:

auditExclusive SET requires *audit* to be SET.

17.4 Audit Session Reporting

The audit status of an audit session can be determined with TPM2_GetSessionAuditDigest(). This command returns a data structure that includes the audit session digest. If the handle for the signing key is not TPM_RH_NULL, the TPM returns a signature over the data structure.

In the atypical case where the TPM2_GetSessionAuditDigest() sessionHandle is the same as the handle of the audit session, 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.

17.5 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 session 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.

17.6 Audit Alternative

Both TPM2_GetSessionAuditDigest() and TPM2_GetCommandAuditDigest() require Endorsement Authorization. If an application does not have Endorsement Authorization, it can still obtain proof that a command was run on a particular TPM. The application must have a *fixedTPM* asymmetric encryption key that is trusted to be on the TPM. This key would have similar trust properties to the signing key that would be used with the TPM2_GetSessionAuditDigest() and TPM2_GetCommandAuditDigest() commands. The application uses an audit session that is a salted session with the trusted key specified as *tpmKey*. The salt forces an HMAC session. The HMAC verification is proof that the command was run on that TPM.

18 Session-based encryption

18.1 Introduction

Several commands have parameters that could need to be encrypted going to or from the TPM. An example is the authorization data that is passed to the TPM when an object is created or when the authorization value is changed. Session-based encryption can be used to ensure confidentiality of these parameters.

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 can be encrypted. That parameter must have an explicit size field. Only the data portion of the parameter is encrypted. The TPM should support session-based encryption using XOR obfuscation. Support for a block cipher using CFB mode is platform specific. These 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 can 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.

Note:

Parameter encryption is enabled on a command by command basis. It is not specified when the session is started.

If a session is also being used for authorization, *sessionValue* (see Clause 18.2 and Clause 18.3) is *sessionKey* ‖ *authValue*. The binding of the session is ignored. If the session is not being used for authorization, *sessionValue* is *sessionKey*.

Note:

A policy session that is used for parameter encryption uses *authValue* to calculate *sessionValue* even if the policy does not include `TPM2_PolicyAuthValue()`.

As observed in Clause 16.6.15, if the session is unbound and unsalted, the *sessionValue* entropy is entirely based on the *authValue*. For commands with no *authValue*, such as `TPM2_LoadExternal()` or `TPM2_RSA_Encrypt()`, a bind or salt session must be used to secure the parameter encryption.

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 can only be SET in one session that is used in a command and the *decrypt* attribute can only be SET in one session per command. The attributes can 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.

Note:

When *cpHash* is calculated on encrypted data, it cannot be precalculated for a policy, `TPM2_PolicyCpHash()` is unlikely to be useful in this case.

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 are discarded by the TPM.

The two methods of session-based encryption used in this specification 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 by the HMAC authorization session verification.

18.2 XOR Parameter Obfuscation

For session-based obfuscation using **XOR**(), the operation is:

$$\mathbf{XOR}(parameter,\ hashAlg,\ sessionValue,\ nonceNewer,\ nonceOlder) \tag{31}$$

where

parameter	is a variable sized buffer containing the parameter to be obfuscated
hashAlg	is the hash algorithm associated with the session
sessionValue	see Clause 18.1
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:

The **XOR**() function is defined in Clause 8.4.7.3.

Note:

The obfuscated size of parameter is the same as the size of the underlying parameter. That is, if a `TPM2B_CREATE` is obfuscated, the size of the obfuscated data is the same as the size of the data.

18.3 CFB Mode Parameter Encryption

When session-based encryption uses a symmetric block cipher, an encryption key and IV will be generated from:

$$\mathbf{KDFa}(hashAlg,\ sessionValue,\ \text{"CFB"},\ nonceNewer,\ nonceOlder,\ bits) \tag{32}$$

where

hashAlg	is the hash algorithm associated with the session

(continued on next page)

(continued from previous page)

sessionValue	see Clause 18.1
"CFB"	is a label to differentiate use of **KDFa**()
nonceNewer	*nonceCaller* for a command and *nonceTPM* for a response
nonceOlder	*nonceTPM* for a command and *nonceCaller* for a response
bits	is the number of bits required for the symmetric key plus an IV

Note:

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:

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.

19 Protected Storage

19.1 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.

19.2 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.

19.3 Protection Values

The protection of a sensitive area uses two keys. These values are created from a secret value associated with an object's Storage 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. See Table 30. 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 Storage 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 clauses 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 33. 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 Clause 20.3).

Note:

This method is also used to generate the symmetric key used for the protection of credential values (see Clause 21.4).

To produce the HMAC key, the seed is used in **KDFa**() as shown in Equation 35. 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:

This method is also used to generate the HMAC key for credential values (see Clause 21.4).

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.

Table 30: Protection Values

Protection Type	**Object**	**Duplication Outer Wrapper**	**Credential**
Clause	Clause 19.4, Clause 19.5	Clause 20.3.2.3	Clause 21.3, Clause 21.4
***seed* Source** (for KDFa)	*seedValue* in sensitive area of Storage Parent	Labeled Key Encapsulation (Clause 8.4.5.2)	Labeled Key Encapsulation (Clause 8.4.5.2)
***seed* Encryption Algorithm**	N/A (*seed* not encrypted)	Asymmetric algorithm of the new parent	Asymmetric algorithm of the new parent
***seed* Label** (for Labeled KEM)	N/A (*seed* not encrypted)	“DUPLICATE”	“IDENTITY”
***symKey* Generation Label**	“STORAGE”	“STORAGE”	“STORAGE”
***hmacKey* Generation Label**	“INTEGRITY”	“INTEGRITY”	“INTEGRITY”
Related Commands	`TPM2_Create`, `TPM2_CreateLoaded`, `TPM2_Import`	`TPM2_Duplicate`, `TPM2_Import`, `TPM2_Rewrap`	`TPM2_MakeCredential`, `TPM2_ActivateCredential`

19.4 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 Storage 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 Storage Parent.

The symmetric key for the encryption is computed by:

$$symKey := \mathbf{KDFa}(pNameAlg, seedValue, \text{"STORAGE"}, name, \text{NULL}, bits) \quad (33)$$

where

pNameAlg	is the *nameAlg* of the object's Storage Parent
seedValue	is the symmetric seed value in the sensitive area of the object's Storage Parent (see Clause 24.7.4)
"STORAGE"	is a value used to differentiate the uses of the KDF
name	is the Name of the object being encrypted / decrypted
bits	is the 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.

$$encSensitive := \mathbf{CFB}_{pSymAlg}(symKey, symIv, sensitive) \quad (34)$$

where

$\mathbf{CFB}_{pSymAlg}$	is symmetric encryption in CFB mode using the symmetric algorithm of the Storage Parent
symKey	is the symmetric key from Equation 33
symIv	is an IV from RNG or 0
sensitive	is 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.

19.5 Integrity

The HMAC key (*HMACkey*) for the integrity is computed by:

$$HMACkey := \mathbf{KDFa}(pNameAlg,\ seedValue,\ \text{"INTEGRITY"},\ \text{NULL},\ \text{NULL},\ bits) \tag{35}$$

where

$pNameAlg$	is the *nameAlg* of the object's Storage Parent
$seedValue$	is the symmetric seed value in the sensitive area of the object's Storage Parent (see Clause 24.7.4)
"INTEGRITY"	is a value used to differentiate the uses of the KDF
$bits$	is 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 Equation 34.

Note:

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.

$$outerHMAC := \mathbf{HMAC}_{pNameAlg}(HMACkey,\ symIv \parallel encSensitive \parallel name.buffer) \tag{36}$$

where

$\mathbf{HMAC}_{pNameAlg}$	is the HMAC function using *nameAlg* of the object's Storage Parent
$(HMACkey$	is a value derived from the Storage Parent's symmetric protection value (*seedValue*) according to Equation 35
$symIv$	is a marshaled `TPM2B_IV` containing the symmetric IV value used in Equation 34. Both the size and buffer fields are included in the HMAC
$encSensitive$	is the encrypted `TPM2B_SENSITIVE` produced in Equation 34; after encryption, the size and buffer fields are not separable
$name.buffer$	is the Name of the object being protected (does not include a size field)

The integrity value is placed before the symmetric IV.

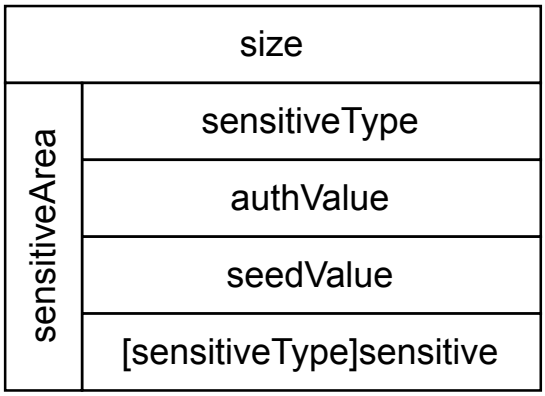
Note:

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:

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



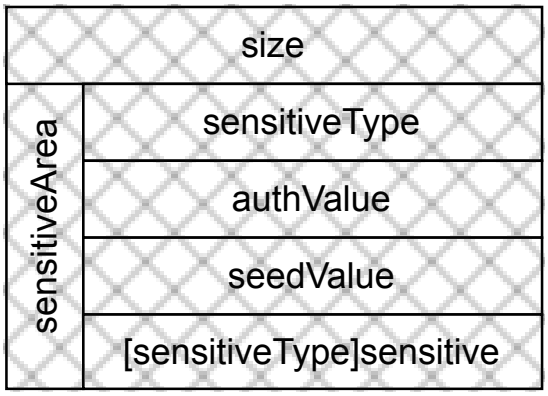
2. Create a symmetric key and IV for encryption:

$$symKey := \mathbf{KDFa}(pNameAlg, seed, \text{"STORAGE"}, name, \text{NULL}, bits)$$

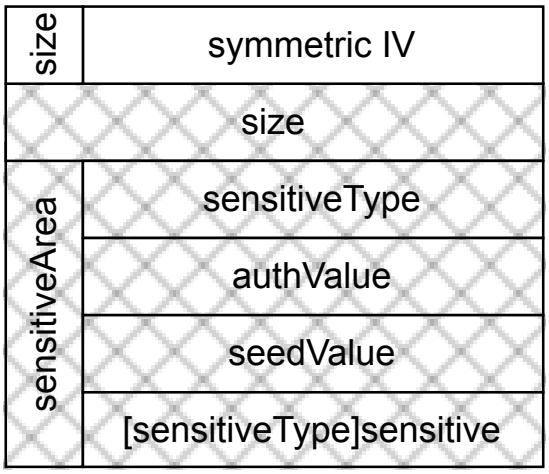
$$symIv := \text{bitsfromtheRNG}$$

3. Create *encSensitive* by encrypting the TPM2B_SENSITIVE

$$encSensitive := \mathbf{CFB}_{pSymAlg}(symKey, symIv, sensitive)$$



4. Add the symmetric IV to (a TPM2B_IV) the encrypted block



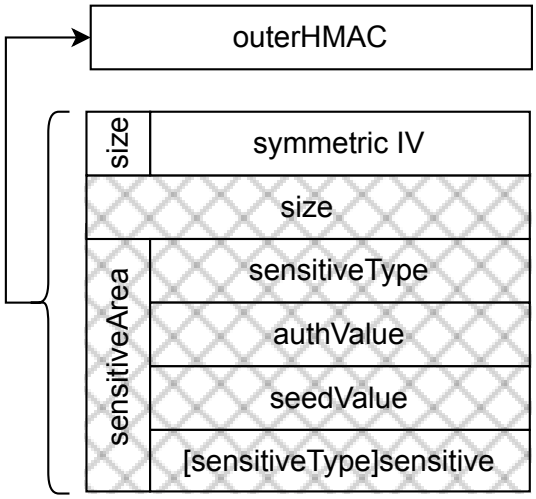
5. Compute the HMAC key

$$HMACkey := \mathbf{KDFa}(pNameAlg, seed, \text{"INTEGRITY"}, \text{NULL}, \text{NULL}, bits)$$

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.

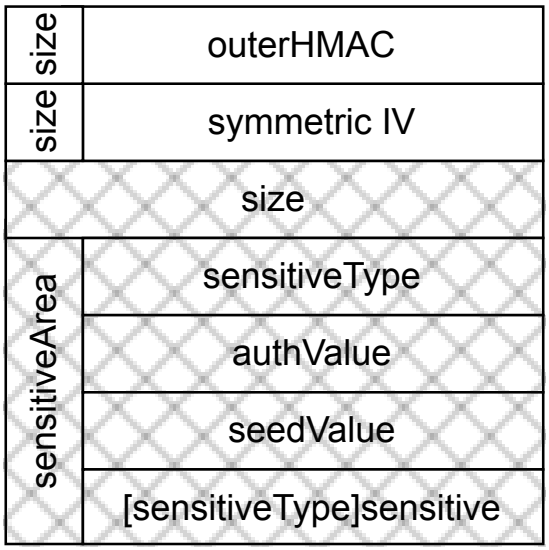
$$outerHMAC := \mathbf{HMAC}_{pNameAlg}(HMACkey, symIv \parallel encSensitive \parallel name.buffer$$

> ***Note:***
>
> An overall size field will be added to make the resulting TPM2B_PRIVATE structure.



7. Marshal the *outerHMAC* into a TPM2B_DIGEST and append the symmetric IV and encrypted sensitive.

> ***Note:***
>
> An overall size field will be added to make the resulting TPM2B_PRIVATE structure.


size
outerHMAC
size
symmetric IV
size
sensitiveArea
sensitiveType
authValue
seedValue
[sensitiveType]sensitive

20 Protected Storage Hierarchy

20.1 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) can 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.

20.2 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 (Primary Seed) is used to generate a key that uses a specific set of algorithms. If this key is a restricted decryption key, then it is a Parent Key. If it is not a Derivation Parent, then it is a Storage Parent under which other objects can be created or attached.

A Storage Parent provides protection for the sensitive area in another object (a child) when that object is stored outside of the TPM. Protection is provided by symmetric encryption and HMAC-based integrity protection of the sensitive area. There are two different cases for sensitive area encryption: storage and duplication.

When an Ordinary Object is created (`TPM2_Create()` or `TPM2_CreateLoaded()`) the keys used for protection of the Object's sensitive area are derived from a seed value (*seedValue*) in the sensitive area of the Storage Key. When an Object is prepared for duplication, its sensitive area is protected by a random key and a form of Diffie-Hellman is used to convey the key to the duplication target.

The objects in a hierarchy have a parent-child relationship. A Storage Key that is protecting other objects is a Storage 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 that Parent as an ancestor. Unless it is intended to be used as a parent, a child object can be of any type.

A Derived Object is a child of its Derivation Parent. Both Primary Objects and Derived Objects are derived from seed values. For a Primary Object, the seed value is a Primary Seed and for a Derived Object the seed value is the sensitive value in the Derivation Parent.

The sensitive part of an object created from a seed is not stored off of the TPM except in a context blobs (see Clause 27). This means that the seed used to create the Object is not also used to generate protection keys for the Object. When an Object is duplicated, its sensitive area is protected by a random key, so the creation seed is not put at risk by the duplication process (see Clause 20.3).

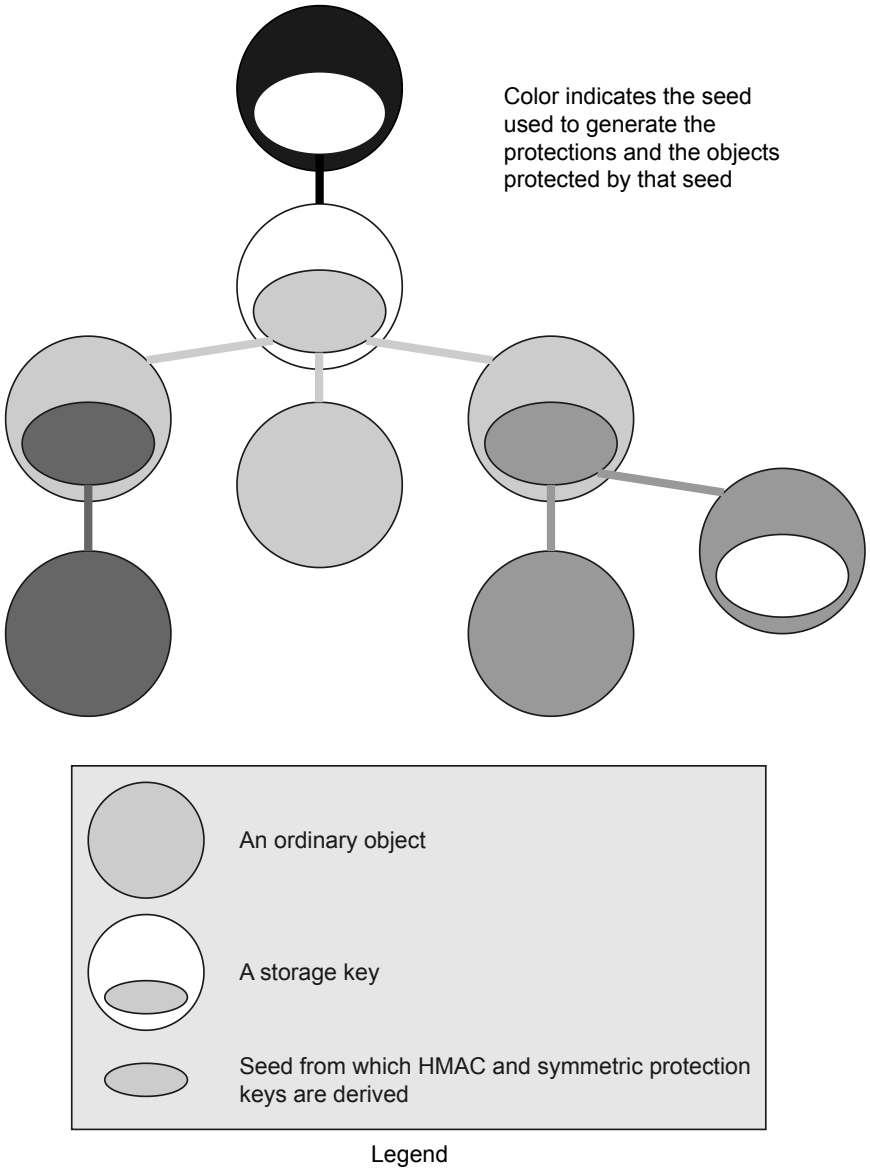


Figure 14: Symmetric Protection of Hierarchy

There are two classes of Storage Keys: asymmetric and symmetric. All Storage Keys contain a symmetric protection key. An asymmetric Storage Key has a public identity that can be used as the target of an identity-based or secret-based duplication operation. An object that is a symmetric block cipher Object can also be a Storage Key, but can only be the target of secret-based duplication

20.3 Duplication

20.3.1 Definition

Duplication is the process of allowing an object to be a child of additional Storage Parent keys. The new parent (NP) can 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 cannot 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.

When an Object is duplicated, its sensitive area can be protected with an outer wrapper, an inner wrapper, or both. The outer wrapper uses the Labeled KEM of the parent (Clause 8.4.5.2) and provides identity-based duplication. The inner wrapper uses a symmetric key that is under control of the duplication authority for the Object. Duplication using an inner wrapper is secret-based duplication.

Note:

The duplication authority is the entity that controls the conditions under which an Object can be duplicated.

20.3.2 Protections

20.3.2.1 Introduction

In TPM2_Duplicate(), the caller can indicate 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 can 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_HIERARCHY 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 can be SET or CLEAR if the *fixedTPM* attribute is SET in the Storage Parent. If the *fixedTPM* attribute of a Storage Parent is not SET, then the *encryptedDuplication* attribute is required to be the same in all descendant objects of that Storage Parent.

20.3.2.2 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.

$$innerIntegrity := \mathbf{H}_{nameAlg}(sensitive \parallel name) \tag{37}$$

where

$\mathbf{H}_{nameAlg}$	is the hash function using the *nameAlg* of the object
$sensitive$	is a TPM2B_SENSITIVE
$name$	is 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.

$$encSensitive := \mathbf{CFB}_{pSymAlg}(symKey, 0, innerIntegrity \parallel sensitive) \quad (38)$$

where

$\mathbf{CFB}_{pSymAlg}$	is 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
$innerIntegrity$	is the value from Equation 37
$sensitive$	is the sensitive value used in Equation 37

If no inner wrapper is specified, no integrity value is computed, and no encryption occurs in this first phase and

$$encSensitive := sensitive \quad (39)$$

20.3.2.3 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 19. However, the seed from which the protection keys are derived is the shared-secret output of the Labeled KEM (Clause 8.4.5.2) of the NP. The Labeled KEM protocol label for the duplication wrapper is “DUPLICATE”.

The seed is encapsulated prior to integrity generation for *encSensitive* or encryption of *encSensitive*, as illustrated in Figure 15.

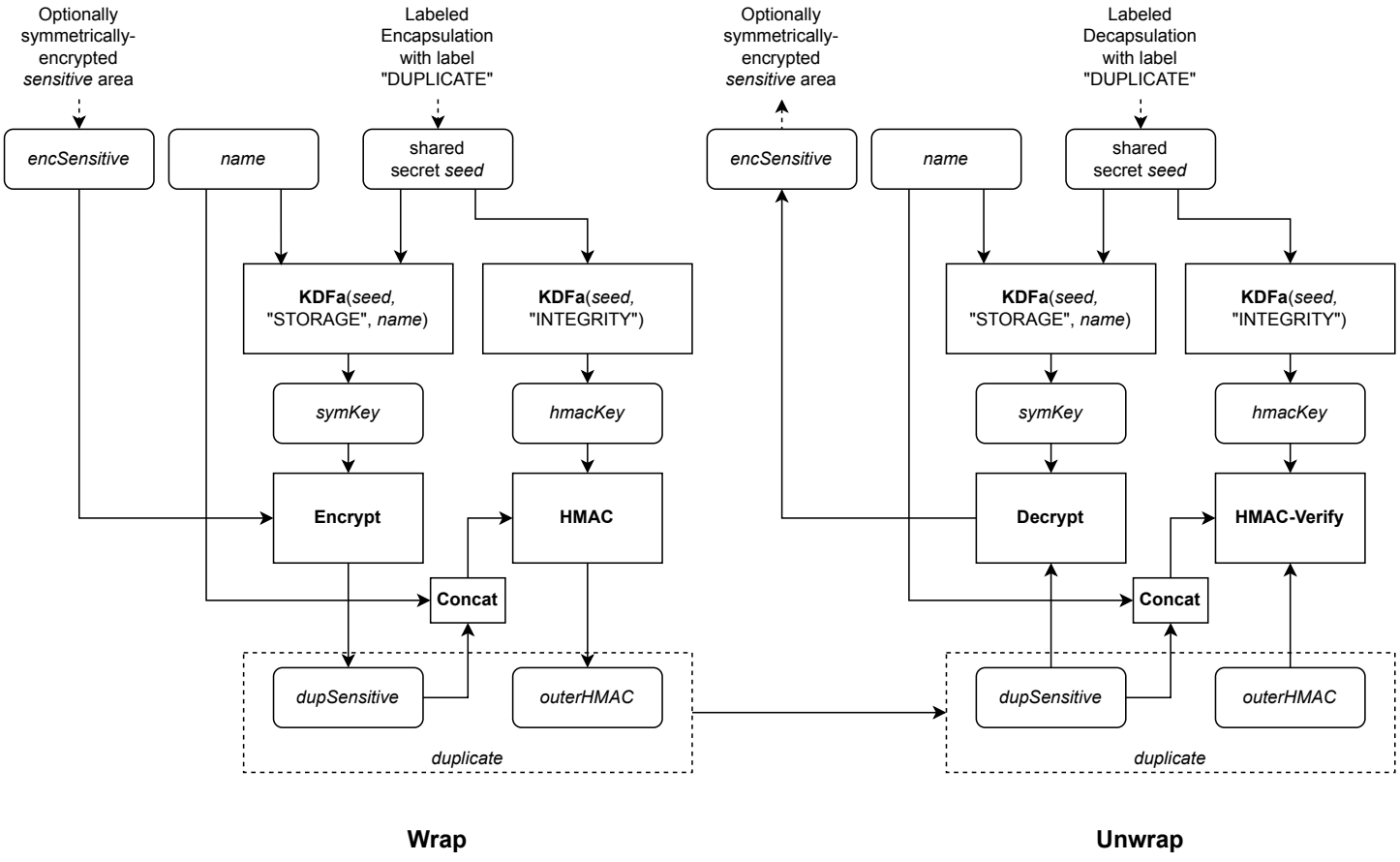


Figure 15: Outer Duplication Wrapper

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 *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:

$$symKey := \mathbf{KDFa}(npNameAlg, seed, \text{"STORAGE"}, Name, \text{NULL}, bits) \quad (40)$$

where

npNameAlg	is the *nameAlg* of the new parent
seed	is the symmetric seed value
"STORAGE"	is a value used to differentiate the uses of the KDF
Name	is the Name of the object being encrypted or decrypted
bits	is the number of bits required for the symmetric key

The *symKey* is used to encrypt the *encSensitive*.

$$dupSensitive := \mathbf{CFB}_{npSymAlg}(symKey, 0, encSensitive) \quad (41)$$

where

$\mathbf{CFB}_{npSymAlg}$	is symmetric encryption in CFB mode using the algorithm of the new parent
$symKey$	is the symmetric key from Equation 40
$encSensitive$	is the value from either Equation 38 or Equation 39

> ***Note:***
>
> The entire TPM2B_PRIVATE, including the `size` field, is used.

Next, an HMAC key is generated from seed:

$$HMACkey := \mathbf{KDFa}(npNameAlg, seed, \text{"INTEGRITY"}, \text{NULL}, \text{NULL}, bits) \quad (42)$$

where

$npNameAlg$	is the *nameAlg* of the object's new parent
$seed$	is the symmetric seed value used in Equation 40
"INTEGRITY"	is a value used to differentiate the uses of the KDF.
$bits$	is 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().

$$outerHMAC := \mathbf{HMAC}_{npNameAlg}(HMACkey, dupSensitive \parallel Name) \quad (43)$$

where

$\mathbf{HMAC}_{npNameAlg}$	is the HMAC function using *nameAlg* of the new parent
$HMACkey$	is a value derived from the new parent symmetric protection value according to Equation 42
$dupSensitive$	symmetrically encrypted sensitive area produced in Equation 41
$Name$	is the Name of the object being duplicated

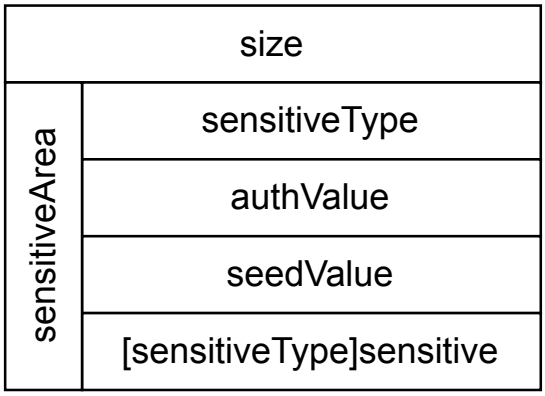
To complete the duplication process, the TPM2B_PUBLIC and TPM2B_ENCRYPTED_SECRET produced by TPM2_Duplicate() are used in TPM2_Import() at the TPM containing the public and private portions of

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.

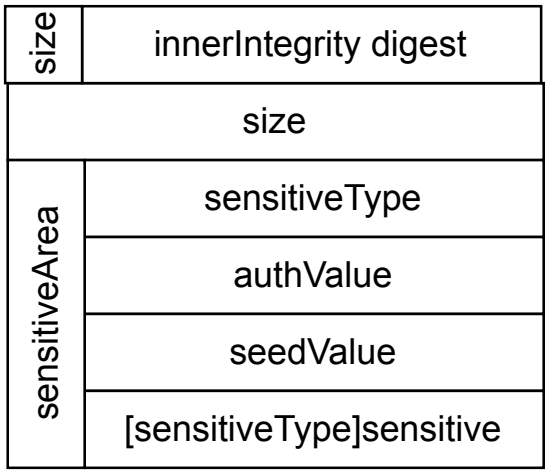
1. Marshal the *sensitive* area into a `TPM2B_SENSITIVE`

> ***Note:***
>
> If no inner or outer wrapper is applied to the object, this structure is returned as the *duplicate* parameter in the response for `TPM2_Duplicate()`.



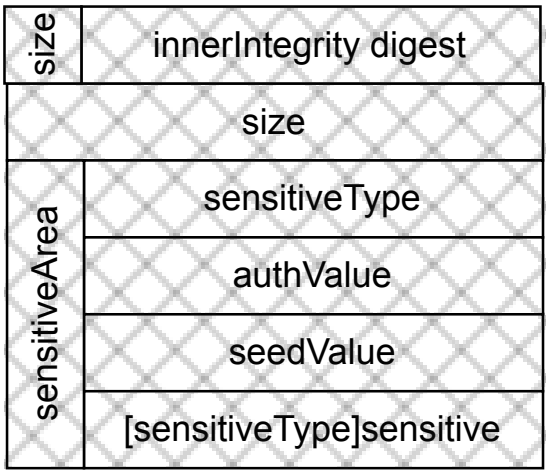
2. Compute an *innerIntegrity* value

$$innerIntegrity := \mathbf{H}_{nameAlg}(sensitive \parallel name)$$



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`

$$encSensitive := \mathbf{CFB}_{symAlg}(symKey, 0, innerIntegrity \parallel sensitive)$$

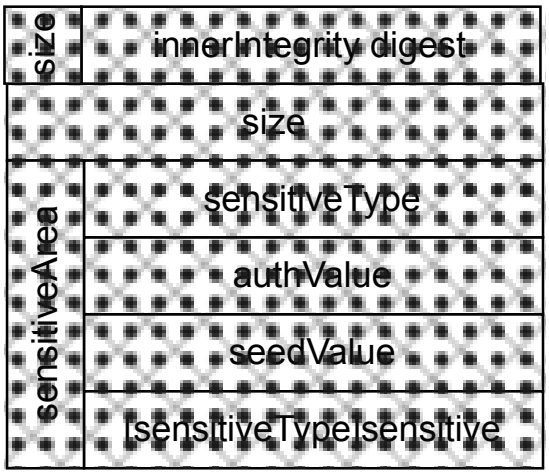


5. Using methods of the asymmetric new parent, create a *seed* value
6. Create a symmetric key (*symKey*):

$$symKey := \mathbf{KDFa}(npNameAlg, seed, \text{"STORAGE"}, Name, \text{NULL}, bits)$$

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)

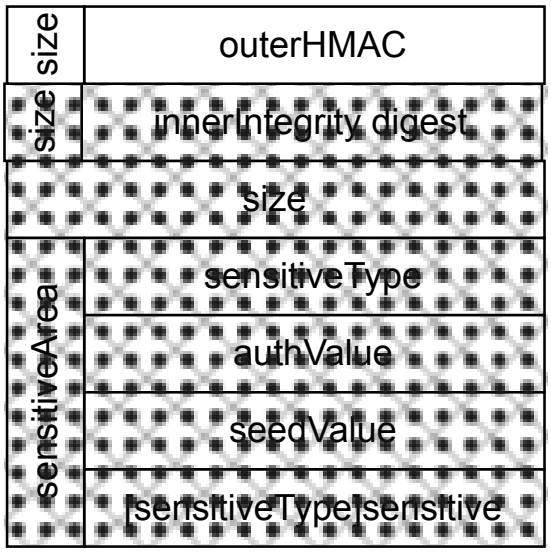
$$HMACkey := \mathbf{KDFa}(npNameAlg, seed, \text{"INTEGRITY"}, \text{NULL}, \text{NULL}, bits)$$

9. Compute the HMAC over *dupSensitive* and include the object *Name*

$$outerHMAC := \mathbf{HMAC}_{npNameAlg}(HMACkey, dupSensitive \parallel Name)$$

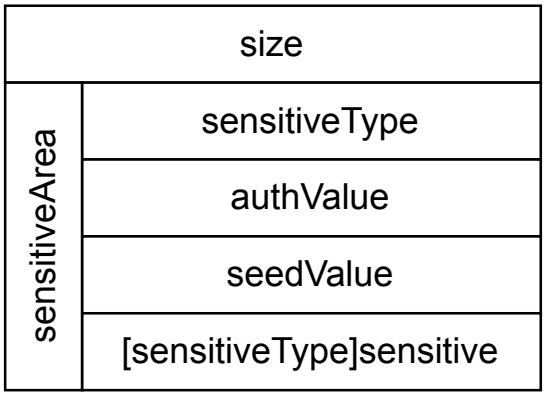
Note:

An overall size field will be added to make the resulting TPM2B_PRIVATE structure.



The text below 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`

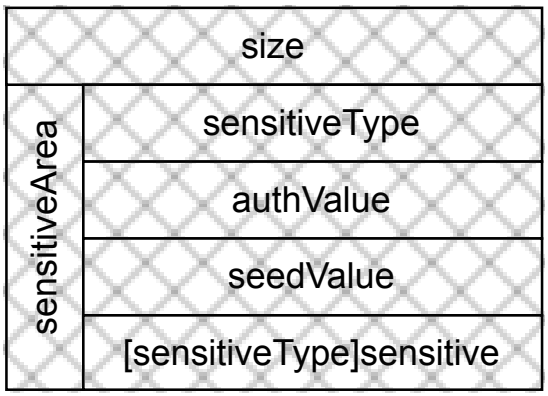


2. Since there is no inner wrapper set *encSensitive* := *sensitive*
3. Using methods of the asymmetric new parent, create a *seed* value
4. Create a symmetric key for encryption:

$$symKey := \mathbf{KDFa}(npNameAlg, seed, \text{"STORAGE"}, name, \text{NULL}, bits)$$

5. Create *dupSensitive* by encrypting *encSensitive*

$$dupSensitive := \mathbf{CFB}_{npSymAlg}(symKey, 0, sensitive)$$



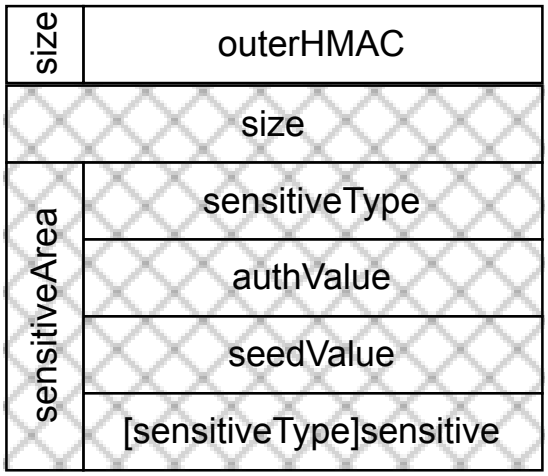
6. Compute the HMAC key from the *seed* created in step 3

$$HMACkey := \mathbf{KDFa}(npNameAlg, seed, \text{"INTEGRITY"}, \text{NULL}, \text{NULL}, bits)$$

7. Compute the HMAC over the *dupSensitive*

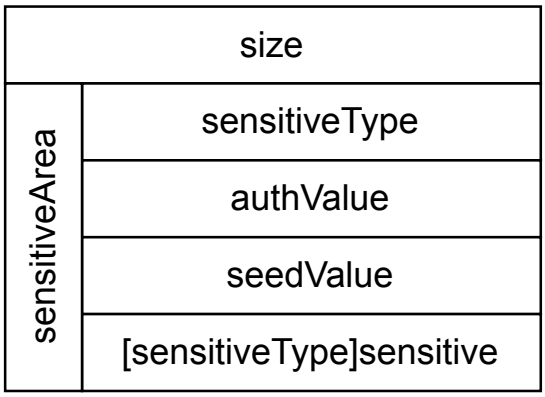
$$outerHMAC := \mathbf{HMAC}_{npNameAlg}(HMACkey, dupSensitive \parallel name)$$

> ***Note:***
> An overall size field will be added to make the resulting TPM2B_PRIVATE structure.



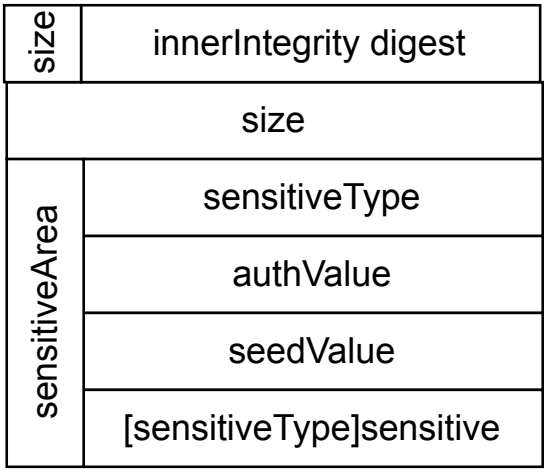
The text below illustrates the processing of a duplication blob when an outer wrapper, but no inner wrapper is used in the sensitive area.

1. Marshal the *sensitive* area into a TPM2B_SENSITIVE



2. Compute an *innerIntegrity* value

$$innerIntegrity := \mathbf{H}_{nameAlg}(sensitive \parallel Name)$$

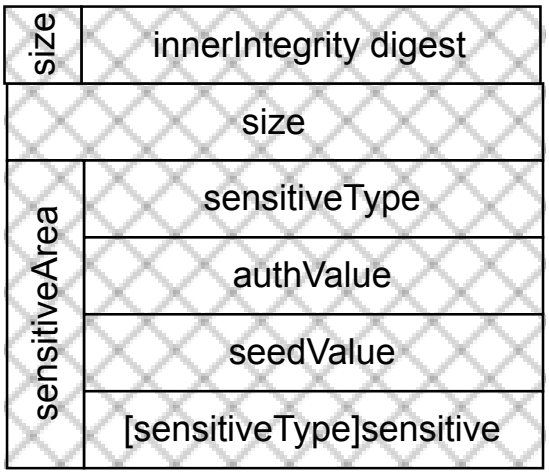


3. Set the encryption key (*symKey*) to *encryptionKeyIn* or a random value produced by the TPM.
4. Create *encSensitive* by encrypting the *innerIntegrity* value and the TPM2B_SENSITIVE

$$encSensitive := \mathbf{CFB}_{symAlg}(symKey, 0, innerIntegrity \parallel sensitive)$$

Note:

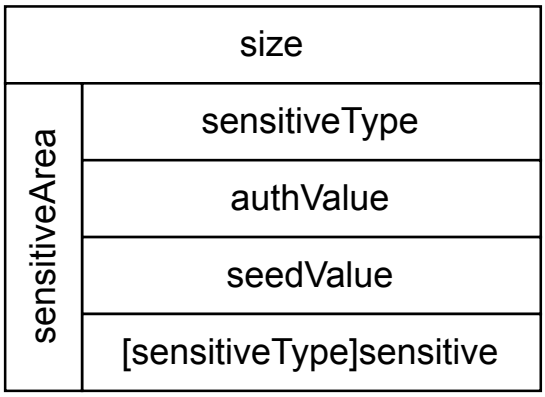
An overall size field will be added to make the resulting TPM2B_PRIVATE structure.



The text below illustrates the processing of a duplication blob 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.



20.3.3 Rewrap

20.3.3.1 Introduction

TPM2_Rewrap() is a primitive of an exemplar key recovery service that performs all its security-sensitive processes on TPMs.

The effect of the recovery service is indistinguishable from duplication of a source key directly from a source platform to a destination platform.

The advantage of the recovery service is that

- registration of a source key with the recovery service relies upon an operational source platform, but not upon an operational destination platform, and
- delivery of the source key by the recovery service relies upon an operational destination platform, but not upon an operational source platform.

The recovery service keeps a source key from a source platform, irrespective of whether the destination platform is known. The source key is protected from the recovery service by virtue of a backup password that must be kept hidden from the recovery service but revealed to a destination platform. When the destination

platform is revealed to the recovery service, the recovery service facilitates the installation of the source key in the destination platform.

1. While the source platform is operational, the source platform uses `TPM2_Duplicate()` to create a doubly wrapped duplication BLOB using a source key *TpmPrivateKey*, a backup password, and the recovery service's public key. (Duplication BLOBs are described earlier in this sub-clause. Note that the "Outer Duplication Wrapper" sub-clause explains that the outer wrapping is symmetric encryption that depends on a seed generated from a public key.)
2. While the source platform is operational, the source platform sends the duplication BLOB (Source BLOB in Figure 16) to the recovery service, which stores the BLOB.
3. When a destination platform is revealed to the recovery service, the recovery service uses `TPM2_Rewrap()` to derive another doubly wrapped duplication BLOB using the original doubly wrapped duplication BLOB, the recovery service's key, and the destination platform's public key.
4. When the destination platform is operational, the recovery service sends the derived duplication BLOB (Recovery BLOB in Figure 16) to the destination platform.
5. While the destination platform is operational, the destination platform uses `TPM2_Import()` to create a normal key BLOB from the derived duplication BLOB, the destination platform's key, and the backup password.

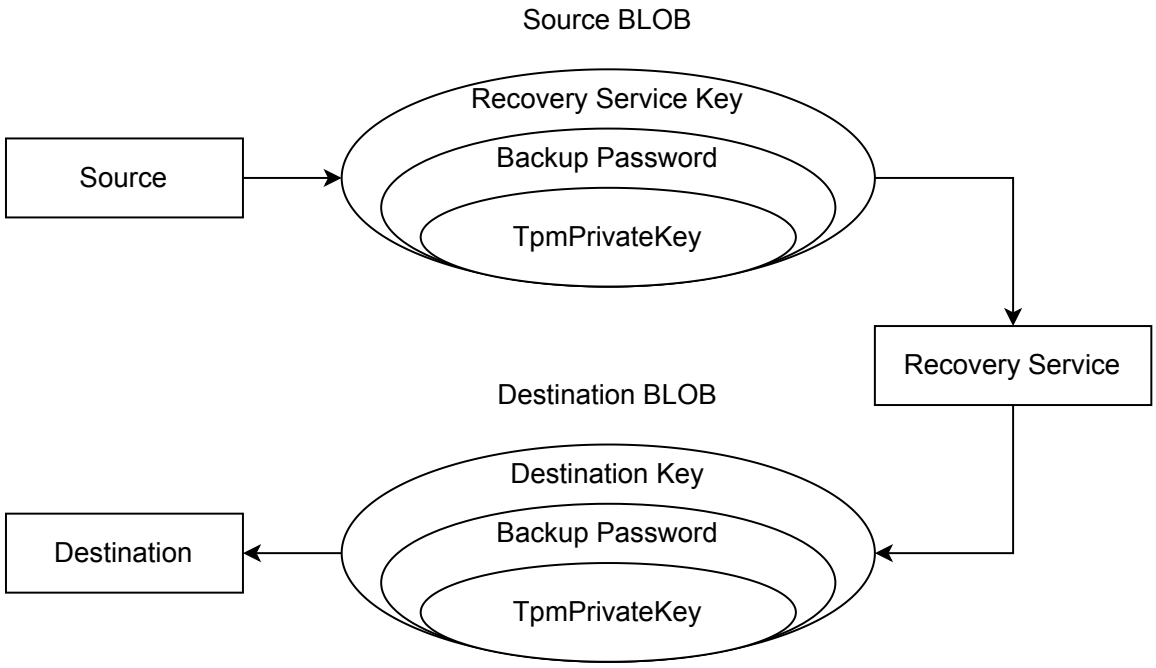


Figure 16: Key Recovery Process

20.3.3.2 Creating a backed-up key

1. At the source platform, a key to be backed up, *sourceKey* == [*sourcePubKey*, *sourceSensitiveKey*], and the recovery service's public key *recoveryServicePubKey* are loaded in the source TPM.
2. At the source TPM, `TPM2_Duplicate()` is used to create the doubly wrapped duplication BLOB, which is *sourceSensitiveKey,* wrapped by *encryptionKeyIn*, wrapped by *recoveryServicePubKey*. The parameters to `TPM2_Duplicate()` are:
    1. *objectHandle* - references the key *sourceKey* to be sent to the recovery service

2. *newParentHandle* - references the recovery service's public key *recoveryServicePubKey*
3. *encryptionKeyIn* - is the backup password (this is an optional parameter and if the caller does not provide a value, the TPM will generate one)
4. *symmetricAlg* - the encryption algorithm for the inner wrapper

3. The TPM returns:
    1. *encryptionKeyOut* - returned only if the TPM generated the key used for the inner wrapper
    2. *duplicate* - the wrapped sensitive area of *objectHandle;* the Source BLOB.
    3. *outSymSeed* - a protected version of the seed used to make the symmetric key used for outer wrapping encryption
4. The duplication BLOB duplicate is sent to the recovery service

20.3.3.3 Recovering a backed-up key

1. At the recovery service's platform, the recovery service's key *recoveryServiceKey* == [*recoveryServicePubKey*, *recoveryServiceSensitiveKey*], and the destination platform's public key *destinationPubKey* are loaded into the recovery service's TPM.
2. At the recovery service's TPM, `TPM2_Rewrap()` is used to is used to replace the outer wrapper of the Source BLOB with an outer wrapper tied to the Destination Key *destinationPubKey*. The parameters for `TPM2_Rewrap()` are:
    1. *oldParent* - references the recovery service's key *recoveryServiceKey*
    2. *newParent* - references the destination platform's public key *destinationPubKey*
    3. *induplicate* - the Source BLOB
    4. *inSymSeed* - this is *outSymSeed* from the source platform. It is needed to derive the symmetric key used by the source platform for outer wrapping encryption
3. At the recovery service, the TPM will return
    1. *outDuplicate* - the rewrapped Destination BLOB
    2. *outSymSeed* - a protected version of the seed used to make the symmetric key used by the recovery service for outer wrapping encryption
4. At the destination platform, the destination platform's key *destinationKey* == [*destinationPubKey*, *destinationSensitiveKey*] is loaded into the TPM.
5. At the destination platform, `TPM2_Import()` is used to create *outPrivate* , which is a normal key BLOB that can be loaded into the TPM on the platform. The parameters to `TPM2_Import()` are:
    1. *parentHandle* - a reference to the destination platform's key (this will become the Storage Parent for the imported object)
    2. *encryptionKey* - the backup password (*encryptionKeyIn* or *encryptionKeyOut*)
    3. *objectPublic* - the public area of the key being imported
    4. *duplicate* - the Destination BLOB *outDuplicate* from the recovery service
    5. *inSymSeed* is *outSymSeed* from the recovery service. It is needed to derive the symmetric key used by the recovery service for outer wrapping encryption
6. At the destination platform, the TPM returns
    1. *outPrivate* - the sensitive area of the imported object

20.4 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 can 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 can 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 can 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 Storage Parent is usable in a different hierarchy, then all the descendants of the Storage Parent are also usable in the different hierarchy as well.

Note:

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:

If a particular Storage Parent is usable in multiple hierarchies, then descendants of that Storage Parent are usable in the same hierarchies regardless of when they are created. That is, if they are created after the duplication of the Storage 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 cannot 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 can 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.

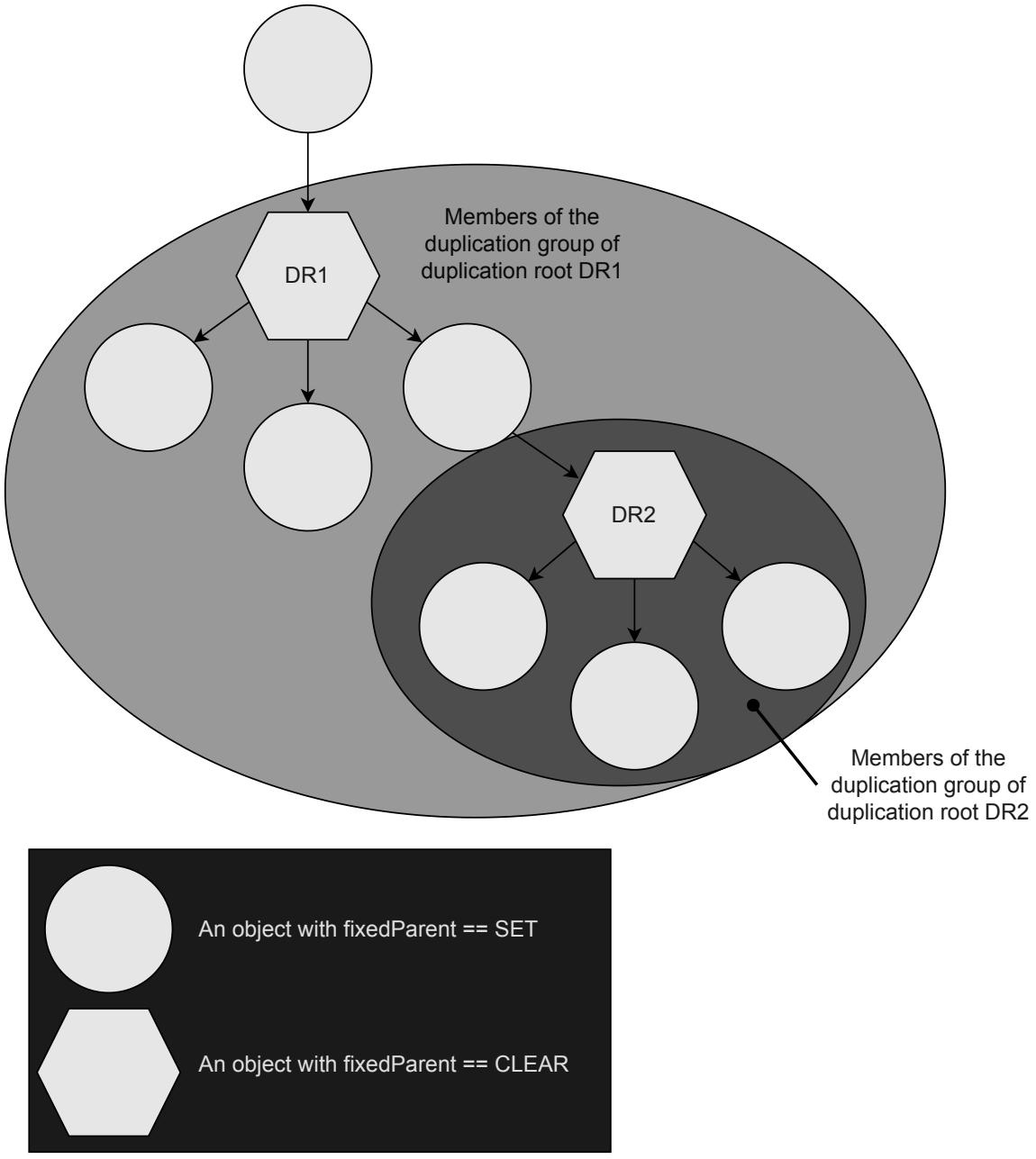


Figure 17: Duplication Groups

20.5 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 18.

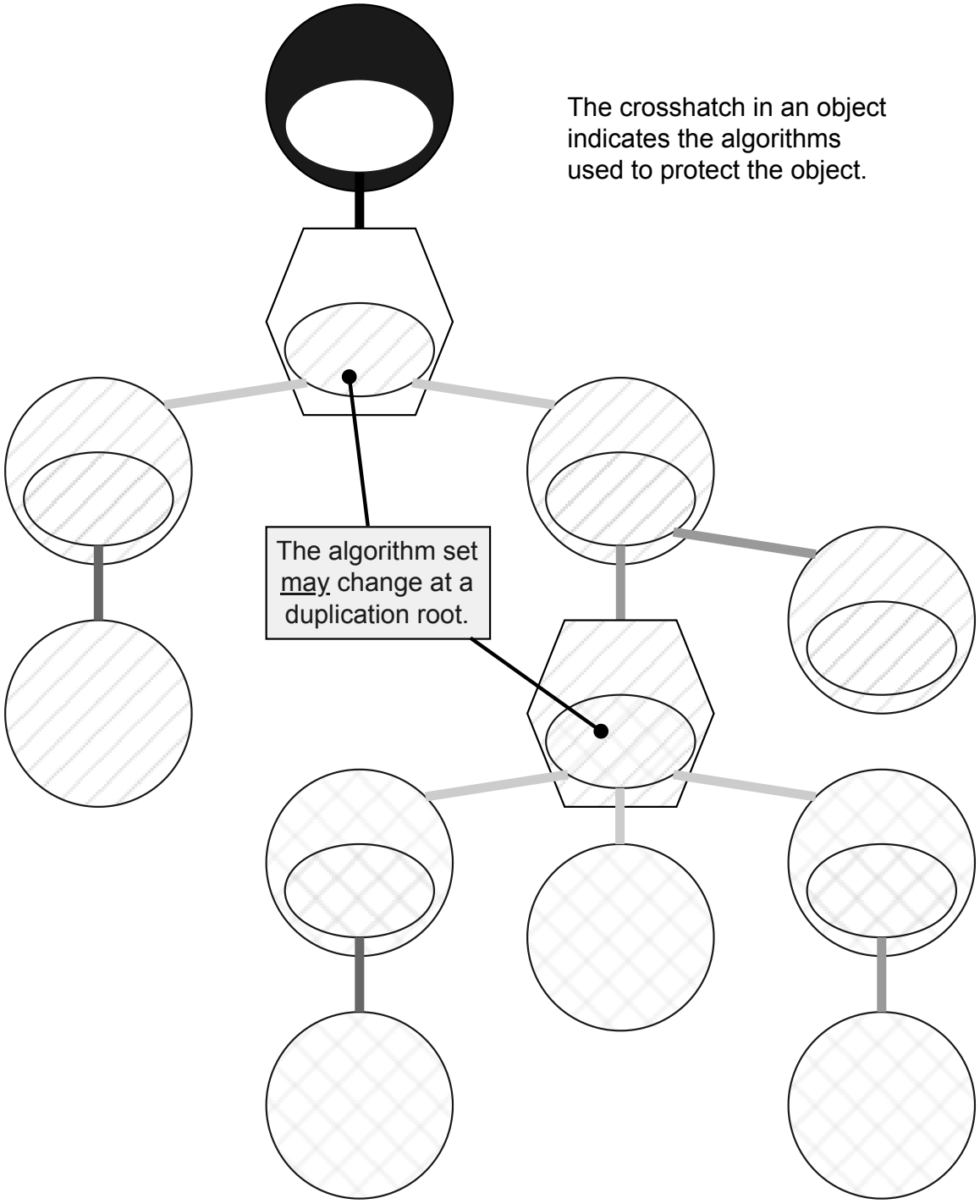


Figure 18: Protection Groups

20.6 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 can be duplicated directly by `TPM2_Duplicate()`.

Table 31 lists the possible combinations of an object's hierarchy attributes and the interpretation of each combination.

Table 31: 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 Storage 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.

20.7 Primary Seed Hierarchies

A Primary Object is an object that is derived from a Primary Seed value. The sensitive area of a Primary Object is not returned in the `TPM2_CreatePrimary()` or `TPM2_CreateLoaded()` response. The Primary Object will need to be regenerated each time it is needed, or it can be made persistent in NV memory on the TPM (`TPM2_EvictControl()`).

Note:

A Primary Object can be duplicated in which case its sensitive area will be stored off of the TPM.

Note:

The reason for not allowing a Primary Object to be returned is to prevent certain types of power analysis attacks on the Primary Seed values.

Once created, a Primary Object can be context-saved/restored.

A Primary Object can have *fixedParent* SET or CLEAR. If a Primary Object has *fixedParent* SET, then *fixedTPM* is required to be SET.

Hierarchy Attributes Settings Matrix

Table 32 shows the combinations of hierarchy settings allowed for an object. In the table, the check marks ("X") indicate that the combination is allowed.

Table 32: Allowed Hierarchy Settings

***fixedTPM* setting in**		**Object's *fixedParent***		**Comments**
parent	**object**	**CLEAR**	**SET**	
CLEAR	CLEAR	X	X	if the parent's fixedTPM attribute is CLEAR, the child's fixedTPM is required to be CLEAR
CLEAR	SET			
SET	SET		X	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	X		

Note:
[1] For purposes of this table, the parent of a Primary Object is considered to have a *fixedTPM* attribute that is always SET.
[2] If the parent has *fixedTPM* SET, then a child can be duplicable (*fixedParent*== CLEAR) or not (*fixedParent* == SET). If the child is not duplicable, then it is required 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()`).

21 Credential Protection

21.1 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.

Example:

The enroller sends an AK along with 1000 authentic TPM EK certificates. The credential provider runs `TPM2_MakeCredential()` against the AK and all 1000 EKs. The enroller runs `TPM2_ActivateCredential()` against its TPM and recovers the credential. The credential provider is ensured that the AK is from an authentic TPM, but it does not know which of the 1000 TPMs the AK resides on.

21.2 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 can 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 a challenge and then “wraps” the challenge 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 can be used. It is up to the credential provider to decide what is acceptable for an “EK.”

The encrypted challenge and the wrapped encryption key are then delivered to the initiator. The initiator can decrypt the challenge by loading the “EK” and the object onto the TPM and asking the TPM to return the challenge. The TPM will decrypt the challenge 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 challenge.

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 can 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.

21.3 Protection of Credential

The credential blob (which typically contains the information used to decrypt the challenge) from the credential provider contains a value that is returned by the TPM if the `TPM2_ActivateCredential()` is successful. The value can be anything that the credential provider wants to place in the credential blob but is expected to be simply a large random number.

As illustrated in Figure 19, the credential provider protects the credential value (CV) with an integrity HMAC and encryption based on a seed. The seed is the shared-secret output of the Labeled KEM as discussed in Clause 8.4.5.2. The Labeled KEM protocol label for a protected credential is “IDENTITY”.

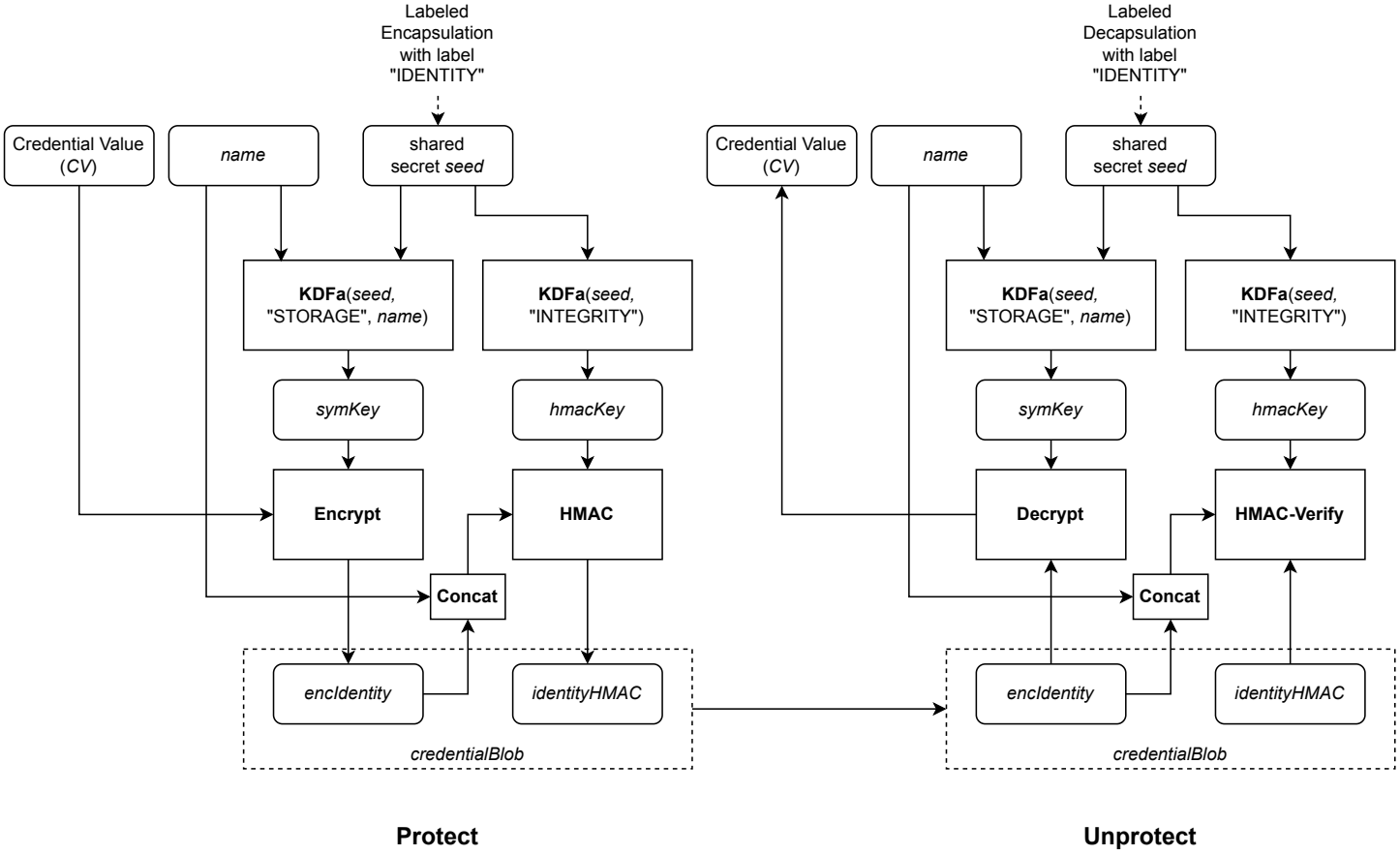


Figure 19: Protection of Credential

The credential provider provides *encIdentity*, *identityHMAC*, and the Labeled KEM ciphertext to the initiator. The initiator can decrypt the CV using `TPM2_ActivateCredential()` with *credentialBlob* consisting of *encIdentity* and *identityHMAC*, and the Labeled KEM ciphertext as *secret*.

The encryption of *encIdentity* is discussed in Clause 21.4. The calculation of *identityHMAC* is discussed in Clause 21.5.

21.4 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 described in a clause in Part 1. In the process of creating the seed, the label is required to be “IDENTITY” instead of “DUPLICATE”.

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 using the label “STORAGE”:

$$symKey := \mathbf{KDFa}(ekNameAlg, seed, \text{"STORAGE"}, name, \text{NULL}, bits) \quad (44)$$

where

ekNameAlg	is the *nameAlg* of the key serving as the “EK”
seed	is the symmetric seed value produced using methods specific to the type of asymmetric algorithms of the “EK”
"STORAGE"	is a value used to differentiate the uses of the KDF
name	is the Name of the object associated with the credential
bits	is the number of bits required for the symmetric key

The *symKey* is used to encrypt the CV. The IV is set to 0.

$$encIdentity := \mathbf{CFB}_{ekSymAlg}(symKey,\ 0,\ CV) \tag{45}$$

where

$\mathbf{CFB}_{ekSymAlg}$	is symmetric encryption in CFB mode using the symmetric algorithm of the key serving as “EK”
symKey	is the symmetric key from Equation 44
CV	is the credential value (a TPM2B_DIGEST)

> ***Note:***
>
> The entire TPM2B_DIGEST, including the size field, is used.

21.5 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:

$$HMACkey := \mathbf{KDFa}(ekNameAlg,\ seed,\ \text{"INTEGRITY"},\ \text{NULL},\ \text{NULL},\ bits) \tag{46}$$

where

ekNameAlg	is the *nameAlg* of the target “EK”
seed	is the symmetric seed value used in Equation 44; produced using methods specific to the type of asymmetric algorithms of the “EK”
"INTEGRITY"	is a value used to differentiate the uses of the KDF
bits	is 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.

$$identityHMAC := \mathbf{HMAC}_{ekNameAlg}(HMACkey, encIdentity \parallel Name) \quad (47)$$

where

$\mathbf{HMAC}_{ekNameAlg}$	is the HMAC function using *nameAlg* of the "EK"
$HMACkey$	is a value derived from the "EK" symmetric protection value according to Equation 46.
$encIdentity$	is the symmetrically encrypted sensitive area produced in Equation 45
$Name$	is 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*.

21.6 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

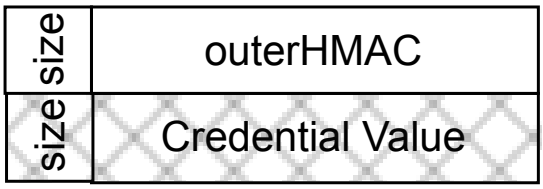
$$encIdentity := \mathbf{CFB}_{ekSymAlg}(symKey, 0, CV)$$

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

$$outerHMAC := \mathbf{HMAC}_{ekNameAlg}(HMACkey, encIdentity \parallel Name)$$



22 Object Attributes

22.1 Base Attributes

22.1.1 Introduction

Three attributes are used to determine how the TPM can 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.

22.1.2 *Restricted* Attribute

When the *restricted* attribute of a key is SET, the key can 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.

22.1.3 *Sign* Attribute

This attribute can 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 can perform signing according to the key family (e.g., RSA or ECC) and the signing method selected. An external entity can 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.

For a symmetric cipher object, this attribute and the object's mode determines whether the key can encrypt or sign (SMAC).

A symmetric key that can sign is used for performing an HMAC or an SMAC computation. This signature can be checked by another entity that knows the HMAC or SMAC secret key in order to validate the source of the information.

A restricted signing key can only sign a digest that has been produced by the TPM. The digest can 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 this specification as an Attesting or Attestation Key.

22.1.4 *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 part of a child object, the symmetric and HMAC keys are derived from the symmetric seed value (*seedValue*) in the sensitive area of the Storage 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. For example, an attacker can 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 can 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 this specification as a Storage Key.

22.1.5 Uses

Table 33 shows the combinations of an object's functional attributes and describes the resulting properties.

Table 33: Mapping of Functional Attributes

sign	***decrypt***	***restricted***	**Description**
0	0	0	A data blob. Can be accessed using TPM2_Unseal(). NOTE: This attribute set can only be used for a *keyedHash* object.
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 cannot be a storage key.
0	1	1	Indicates that only the default schemes and modes of the key can be used In this specification, a key with these properties is referred to as a Parent Key. Asymmetric keys and symmetric keys with these attributes are Storage Parents, and *keyedHash* objects with these attributes are Derivation Parents. The TPM only allows keys with these attributes to be used on objects that have a specific structure. For Storage Parents, use includes create, load, and activate credential.
1	0	0	Indicates a key that can 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 can be used for HMAC operations.

(continued on next page)

(continued from previous page)

sign	*decrypt*	*restricted*	Description
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 can 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 cannot 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.

22.2 Other Attributes

22.2.1 fixedTPM and fixedParent

These attributes are described in detail in Clause 20.

22.2.2 stClear

This attribute indicates an object that will need to be reloaded after any Startup(CLEAR). Objects can be loaded into the TPM and their context saved by the TPM resource manager. Normally, these saved contexts can 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 cannot be made persistent.

22.2.3 sensitiveDataOrigin

The meaning and allowed settings for this attribute are different for Created and Derived Objects. For a Derived Object, this attribute is required always to be CLEAR. For a Created Object, this attribute is SET if the sensitive data of the object is to be generated by the TPM.

Note:

The reason that *sensitiveDataOrigin* is to be CLEAR for a derived object is that it is impractical to use it to indicate anything about the provenance of the seed value used in deriving an object. The only case in which the *sensitiveDataOrigin* of the Derivation Parent might reasonably be reflected in the derived key is when *sensitiveDataOrigin* and *fixedTPM* are both SET in the parent. For all other cases, there is no way for the TPM to provide any assurance about the setting of *sensitiveDataOrigin.* However, for a derived key with fixedTPM SET, it is a relatively simple matter to check the setting of this attribute in the Derivation Parent. Rather than add to the TPM the complexity of validating that a Derivation Parent has the correct combination of attributes to allow this attribute to be SET, it was decided to require that this attribute be CLEAR rather than ignored. This is because this attribute does not change the way that Object derivation takes place as it does with Object creation.

When a symmetric object (TPM_ALG_KEYEDHASH or TPM_ALG_SYMCIPHER) is created, the caller can 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.

When an asymmetric object is created, this attribute must be SET. 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 can be imported. If it is imported, *fixedTPM* will not be SET.

Note:

The requirement that *sensitiveDataOrigin* be SET for asymmetric objects is enforced indirectly. When an asymmetric key is created, the caller is not allowed to provide the sensitive data of the key. Because the caller does not provide the sensitive data, *sensitiveDataOrigin* is required to be SET. Since this relationship is only checked when the object is created, *sensitiveDataOrigin* is allowed to have any setting when an object is loaded or imported.

22.2.4 userWithAuth

This attribute indicates that the object's *authValue* can be used to provide the USER role authorizations for the object. If this attribute is CLEAR, then USER role authorizations can only be provided by satisfying the object's *authPolicy* in a policy session. A policy session can be used for USER mode authorizations when this attribute is SET or CLEAR.

22.2.5 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* can be used in an HMAC session to perform operations that require ADMIN role.

As with USER role authorizations, any ADMIN role action can 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.

22.2.6 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.

22.2.7 encryptedDuplication

If this attribute is CLEAR, then an object can 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 can 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.

23 Object Structure Elements

23.1 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 this specification are defined in terms of these presumptive structures and any implementation will need to produce equivalent results.

23.2 Public Area

The public area contains the information for identification of an object and its properties. The fields of the public are listed and described in Table 34.

Table 34: 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. For example, an RSA type would be used for an RSA key. A SYMCIPHER type would be used for symmetric encryption or decryption.
nameAlg	This is a second algorithm ID that identifies the hash algorithm used for computing the Name of the object.
objectAttributes	This contains the set of attributes of the object. These attributes are in five classes: usage (*sign*, *encrypt*, *restricted*); authorization (*userWithAuth, adminWithPolicy, noDA*); duplication (*fixedParent, fixedTPM, encryptedDuplication*); creation (*sensitiveDataOrigin*); and persistence (*stClear*).
authPolicy	This will contain the authorization policy for the object if one is defined. *nameAlg* is used as the *authPolicy* hash algorithm, NOTE An object that is intended to be duplicated must have an *authPolicy* enabling the duplication.
[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, the parameters indicate the key size, signing scheme, and symmetric encryption methods associated with the key.

(continued on next page)

(continued from previous page)

Parameter	Description
[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.

23.3 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 such as the private or secret portion of a key. If an object is a Storage Key, it contains the symmetric key that is used to encrypt its child object.

The structure of the sensitive area is shown in Table 35.

Table 35: 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 an octet array of zero or more octets. The authorization value for an object cannot have more octets than the digest produced by the object's *nameAlg*.
seedValue	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 asymmetric keys that are not Storage 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 a symmetric key, this will be the key. For an HMAC key this is the HMAC key value. 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.

Note:

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.

23.4 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:

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 Storage Parent’s symmetric seed (*seedValue*). 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:

Similar protections are used when an object is context saved or duplicated.

23.5 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 handle of 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:

Assuming that key *A* is the Storage Parent of object *B*, then the Qualified Name of *B* (QN_B) is:

$$QN_B := \mathbf{H}_B(QN_A \parallel NAME_B)$$

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:

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

$$QN_A := \mathbf{H}_A(QN_P \parallel \mathbf{H}_A(A))$$

and the *QN* for B is

$$QN_B := \mathbf{H}_B(QN_P \parallel \mathbf{H}_B(B))$$

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. It is then possible to determine if, for example, all ancestor keys use sufficient cryptographic strength. 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.

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:

$$QN := \mathbf{H}_{nameAlg}(A\ \ hierarchy\ \ handle \parallel Primary\ \ Object\ \ Name)$$

Note:

The creation data for an object includes both the Name and QN of the Storage Parent of that object.

23.6 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 (*seedValue*) 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 can 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 27 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 described in Clause 19.

23.7 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.

24 Object Creation

24.1 Introduction

TPM2_Create(), TPM2_CreatePrimary() and TPM2_CreateLoaded() 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 Ordinary Objects that are generated with values from the TPM RNG. TPM2_CreateLoaded() can be used to create a Primary or Ordinary Object.

Note:

TPM2_CreateLoaded() can also be used for Derived Objects. This is covered in more detail in Clause 25.

Deprecated:

TPM2_CreateLoaded() was deprecated in version 184. See Part 0.

Table 36: Creation Commands

Creation command	**TPM2_CreatePrimary**	**TPM2_Create**	**TPM2_CreateLoaded**		
Parent Handle Type	Primary Seed	Storage Parent	Primary Seed	Storage Parent	Derivation Parent
Created Object Type	Primary	Ordinary	Primary	Ordinary	Derived
Public Area Returned	yes	yes	yes	yes	yes
Sensitive Area Returned	no	yes	no	yes	no
creationData Returned	yes	yes	no	no	no

Table 36 compares and contrasts the creation of objects by TPM2_CreatePrimary(), TPM2_Create(), and TPM2_CreateLoaded(). In particular, when creating keys:

- TPM2_CreatePrimary() - creates and loads Primary Objects for immediate use and provides creationData.
- TPM2_Create() - creates Ordinary Objects for later use (via TPM2_Load()). TPM2_Create() returns a BLOB containing the sensitive area of an Ordinary Object and provides creationData.
- TPM2_CreateLoaded() - depending on the type of the parent, generates and loads a Primary Object, an Ordinary Object; or Derived Object.

Authorization to use the Parent is required in order to generate a child. Authorization to use a Primary Seed is required in order to create a Primary Object.

All of the objects created by these commands are similar in most respects. For TPM2_Create() and TPM2_CreatePrimary(), the parameters required to create an object are the same for both commands. They are:

- a public area template,
- the sensitive values,
- optional user-provided identification data, and

- the optional creation PCR selection.

For TPM2_CreateLoaded(), the only parameters are the public area template and the sensitive values.

Note:

The user-data and PCR parameters are not used for TPM2_CreateLoaded() as it does not return the *creationData* used for creation certification. For objects where the creation certification is necessary, the TPM2_Create() or TPM2_CreatePrimary() functions are available.

Any type of object that can be created with TPM2_Create() can be created with TPM2_CreatePrimary() or TPM2_CreateLoaded().

Note:

TPM2_CreateLoaded() can be used for creation of asymmetric keys but it cannot be used for derivation of certain types of asymmetric keys. This limitation is because of the variability in algorithms for some asymmetric key types (such as RSA).

The sensitive area of an Object created from a seed does not leave the TPM except in a saved context or by duplication. If a Primary Object is not context saved (and not persistent), it will need to be recreated after the next TPM2_Startup(). Even if context saved, if a Primary Object is not made persistent in the TPM (TPM2_EvictControl()), it will need to be recreated after each TPM Reset.

24.2 Public Area Template

24.2.1 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 (the *unique* field) will be replaced by the TPM in the creation process, but all other fields in the created object will be identical to those in the template.

In general, the fields in the public area are checked as if the object were being loaded under the Storage Parent indicated in the creation command.

24.2.2 type

This parameter indicates the basic type of the object and determines the format of the *parameters* and *unique* fields. The type can indicate a symmetric key, an asymmetric key, or a data value.

The allowed values for *type* are listed in TPMI_ALG_PUBLIC in Part 2.

24.2.3 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 Storage Parent. For all other cases, the *nameAlg* can 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.

24.2.4 objectAttributes

These flags must be set according to the rules appropriate for loading the object. The required settings are found in Part 2, in the definition of TPMA_OBJECT.

24.2.5 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 the Empty Policy.

24.2.6 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 Storage Parent. For other objects, these parameters can vary according to the *type* and application.

24.2.7 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 can 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 can also set the size of this field to zero and the TPM will replace it with a correctly sized structure.

24.3 Sensitive Values

24.3.1 Overview

The sensitive values that are provided when the object is created allow initial setting of the *authValue* for the object and can provide some other object-sensitive value. The sensitive value can be an encryption key or sealed data.

The sensitive values provided to the TPM in TPM2_Create() and TPM2_CreatePrimary() (the inSensitive parameter) can 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 Storage Parent of the newly created object. This ensures that the entity that controls the Storage Parent does not automatically gain access to the secret values of a child.

24.3.2 userAuth

The *userAuth* value is the initial *authValue* for the created object. This value can 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.

24.3.3 data

This contains information that the caller wants to be incorporated in the sensitive part of the created object. This can 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 and not a primary key, then *data* is required to be an Empty Buffer and *sensitiveDataOrigin* in the template is required to be SET. For a primary key, *data* permits personalization of the key with private data, data that can be provided as an encrypted parameter.

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().

24.4 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()).

24.5 Public Area Creation

24.5.1 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 Part 3.

24.5.2 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.

24.5.3 unique

24.5.3.1 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 (such as, a hash) so that the *unique* value does not compromise the confidentiality of the sensitive area.

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 described in the remainder of Clause 24.2.7.

24.5.3.2 TPM_ALG_KEYEDHASH

This type is used for an HMAC key or data block. The computation for *unique* for a KeyedHash object is:

$$unique := \mathbf{H}_{nameAlg}(obfuscate \parallel key) \tag{48}$$

where

$\mathbf{H}_{nameAlg}$	is the hash using *nameAlg* from the object template
$obfuscate$	is the contents of *seedValue.buffer* in the object's sensitive area
key	is 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.

24.5.3.3 TPM_ALG_SYMCIPHER

This type is used for a symmetric block cipher key. The *unique* value is computed as shown in Equation 48.

24.5.3.4 TPM_ALG_RSA

For an RSA key, *unique* is the public modulus of the key. It is computed as described in Clause 43.8.

24.5.3.5 TPM_ALG_ECC

For an ECC key, *unique* is the public point computed as described in Clause 44.6.

24.5.3.6 TPM_ALG_MLDSA

For an ML-DSA key, *unique* is the public key as described in Clause 46.2.

24.5.3.7 TPM_ALG_MLKEM

For an ML-KEM key, *unique* is the public key as described in Clause 47.2.

24.6 Creation Entropy

24.6.1 Introduction

Table 37 compares and contrasts the methods used to create the cryptographic values of primary, ordinary, and derived keys.

Table 37: Deriving Object Entropy

Object Type	Source of Object Entropy	Described in Clause
Primary	DRBG initialized using a hierarchy seed and the hash of the input template	Clause 24.6.3
Ordinary	TPM's default DRBG	Clause 24.6.2
Derived	KDF	Clause 25.4

- A Primary Object is intended to be created multiple times, in the absence of any other key, with the minimum amount of persistent storage. As a result, the cryptographic values of primary keys are created by instantiating a DRBG.
- An Ordinary Object is intended to be created exactly once and persistently stored. As a result, the cryptographic values of ordinary objects are created by the DRBG that is used by default when a TPM requires random data. This DRBG is seeded with entropy when the TPM was created and topped-up with additional entropy added at intervals.
- A derived key is intended to be derived multiple times from a parent key, and not persistently stored. As a result, the cryptographic values of derived keys are created by applying a KDF and hash algorithm specified in the Derivation Parent to the Derivation Parent's symmetric key, using label and context values provided by the caller.

24.6.2 Entropy for Ordinary Objects

For an Ordinary Object, the random number generator will use the default random number generator of the TPM which produces numbers that are as random as the TPM is able to produce.

24.6.3 Entropy for Primary Objects

For a Primary Object, the DRBG is seeded with a primary seed, a template hash, and a use string. This produces a sequence of bits that have as much entropy as the primary seed and which have a property that is required for generating a Primary Object - the DRBG state can be reinstantiated each time the same Primary Object is created.

Choice of the entropy generation for Primary Objects is a vendor option.

Note:

The Reference Code uses a DRBG based on SP 800-90A in order to minimize compliance issues.

24.7 Sensitive Area Creation

24.7.1 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 can 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 can 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*.

24.7.2 type

The *type* parameter of the object's sensitive area is a copy of the type parameter from the object's public-area template.

24.7.3 authValue

The *authValue* of the object is copied from the *userAuth* field of the *inSensitive* parameter of commands such as `TPM2_Create()`, `TPM2_CreateLoaded()`, or `TPM2_CreatePrimary()`, or from *newAuth* in commands such as `TPM2_ObjectChangeAuth()`.

When the TPM returns a `TPM2B_PRIVATE` structure, the TPM pads the `TPM2B_AUTH` to its maximum size.

Note:

This prevents the TPM from leaking the size of the authorization value in cases where trailing zeros are stripped.

24.7.4 *seedValue*

For a symmetric object, *seedValue* field is used as an *obfuscation* value. It is also used to hold the symmetric seed value for Storage Keys.

For an asymmetric key that is not a Storage Key, *seedValue* is not needed and the TPM will ignore the value if it is present.

For a Storage Key, *seedValue* is used as a seed for generating the integrity and confidentiality values for protecting child objects of the key.

For all object types, when the TPM generates *seedValue*, it is the size of the digest produced by the *nameAlg* of the object.

Note:

Presuming that the protection algorithms of a Storage Key are reasonably balanced (a requirement), then this size of seed will provide adequate entropy required for protection of the child Object.

For an imported symmetric object, *seedValue* is required to be the size of the digest produced by the *nameAlg* of the object.

For an imported Storage Key, *seedValue* is required to be at least to be at least half the size of the digest produced by the nameAlg.

Note:

This requirement is for backward compatibility.

For Imported asymmetric non-Storage Keys, *seedValue* is not required.

Note:

The rationale for these requirements derives from the use of *seedValue*. When *seedValue* is used in a hash, it is the full size. When used in an HMAC, it can be half the size.

seedValue is generated using the “entropy” source used for the object type (see Clause 24.6).

When creating a Primary Object in the Endorsement Hierarchy, it is required that the entropy source be updated to reflect the current SPS. This allows the *sensitiveValue* to remain the same after a change of the SPS but prevents any previously-generated Child Objects in the Endorsement Hierarchy from being loaded after the SPS changes.

Note:

In the Reference Code, this is accomplished by reseeding the DRBG state with the proof value of the storage hierarchy.

24.7.5 sensitive

24.7.5.1 Symmetric Objects

Symmetric objects have a *type* of `TPM_ALG_SYMCIPHER` or `TPM_ALG_KEYEDHASH`. For a symmetric object, the sensitive object data can 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()`. For `TPM2_CreateLoaded()`, if the Parent is a Derivation Parent, then *sensitiveDataOrigin* is required to be CLEAR in the template.

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:

If the key has fewer significant digits than necessary, pad octets of zero are required. 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:

The HMAC algorithm requires that keys larger than the hash block size be hashed before use. This may 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 have to 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.

24.7.5.2 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 described in an algorithm-specific clause of Part 1.

Example:

RSA key generation is described in Clause 43.8 and ECC key generation is described in Clause 44.6.

24.8 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 Storage Parent. If the parent is a Primary Seed, then the algorithm will be TPM `_ALG_NULL`.
- the Name of the Storage Parent. If the parent is a Primary Seed, then the Name will be the handle of the seed.
- the Qualified Name of the Storage 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 can 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.

24.9 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`.

25 Object Derivation

25.1 Introduction

Deprecated:

`TPM2_CreateLoaded()` was deprecated in version 184. See Part 0.

This clause describes the differences between Object creation and Object derivation. If no difference is stated, then there is none.

The `TPM2_CreateLoaded()` command is used for derivation. This command can be used to create or derive any type of object with the type of Object determined by the type of the entity referenced by the *parentHandle* parameter. If *parentHandle* references a Primary Seed, then a Primary Object is created; if *parentHandle* references a Storage Parent, then an Ordinary Object is created; and if *parentHandle* references a Derivation Parent, then a Derived Object is generated.

Note:

For a given template (*inPublic*), the same Primary Object is created by both `TPM2_CreatePrimary()` and `TPM2_CreateLoaded()`.

25.2 Derivation Parameters

For object derivation the TPM uses the *sensitive* value in a Derivation Parent as a key in a key derivation function (KDF). The KDF that is to be used in Object derivation is a property of the Derivation Parent and can include the hash algorithm to use in the derivation process.

Note:

KDFa (`TPM_ALG_KDF1_SP800_108`) is the only KDF that is currently supported by the Reference Code.

Most KDFs require additional parameters in order to have different types of keys derived for different applications. The TPM allows two additional parameters (*label* and *context*) to be provided in `TPM2_CreateLoaded()`. These additional parameters can be provided in two ways: in the *unique* field of the *inPublic* value, or in the *data* field of the *inSensitive* parameter. If provided in the *unique* field, the corresponding value in the *inSensitive.data* field is ignored.

Since `TPM2_CreateLoaded()` does not use the public attributes in the KDF, it can create child keys with the same private key but different attributes.

Note:

`TPM2_PolicyTemplate()` on the parent can be used to restrict the child attributes.

Example:

Once the parent is duplicated, one TPM can derive a key that can only be used for encryption and a different TPM can derive the same key that is restricted to be used for decryption. Or an HMAC key can be restricted to signing on one TPM and verification on another.

25.3 Public Area Template

For `TPM2_CreateLoaded()`, a `TPM2B_TEMPLATE` is used for the *inPublic* parameter instead of a `TPM2B_PUBLIC`. The difference in parameters is to allow overloading of the *unique* field in the *inPublic* parameter. For a `TPM2B_PUBLIC`, the *unique* field is unmarshaled based on the *type* of *inPublic.* For a

TPM2B_TEMPLATE, the *inPublic* is unmarshaled as a byte array and passed to the TPM2_CreateLoaded() action code where it is unmarshaled based on the type of parent and *type* of *inPublic.*

When using TPM2_CreateLoaded() to create a Primary or Ordinary Object, the caller should use the same format for the unique field that would be used when creating the Object with TPM2_CreatePrimary() or TPM2_Create(). The derivation-specific format is required when *parentHandle* references a Derivation Parent.

For object creation, *sensitiveDataOrigin* indicates to the TPM whether the caller is providing the sensitive data or if the TPM is to generate it. For Object Derivation, the caller provides values that influence the derivation process, but the caller does not explicitly set the sensitive value. For this reason, *sensitiveDataOrigin* is required to be CLEAR in the template for a Derived Object.

25.4 Entropy for Derived Objects

25.4.1 Conceptual Description

The 'entropy' for a Derived Object is provided by a protected value in the sensitive area of the derivation parent (the sensitive value). That entropy, along with caller-provided values, is used in a KDF to spread the entropy across values in the derived object. Those derived values are the *sensitive* and *seedValues*. The remainder of the Derived Object is provided by the template or computed from the two derived values.

KDFa is used for the derivation. The parameters for the derivation are:

$$\mathbf{KDFa}(hashAlg,\ sensitive,\ [label,\ ]\ [context,\ ],\ 0,\ 8192) \tag{49}$$

where:

hashAlg	the *nameAlg* of the derivation parent
sensitive	the *sensitive* value in the sensitive area of the derivation parent
label	is an optional string provided by the caller
context	is an optional string provided by the caller

KDFa has a counter and a bits parameter that are set, as shown above, to 0 and 8192 respectively.

Note:

In order to be compliant with SP 800-108 [4], the KDFa function will increment counter to 1 before using it in the generation of the first HMAC block.

This call will cause the KDF to generate 1024 bytes of data, with the results of the first digest being the most significant bytes.

During the derivation process, the data is removed from the 1024-byte buffer as needed for each use. The data is used from most significant byte to least significant byte with no bytes skipped. For most key generations, a deterministic number of bytes will be removed for each of the derived fields (*sensitive* and *seedValue*).

Example:

For a 128-bit AES key in a SYMCIPHER object having SHA-256 as its *nameAlg*, the most significant 16 bytes of the KDF data are used for the AES key and the next-most-significant 32 bytes are used for the *seedValue*.

Example:

For ECC, the TPM uses the method of FIPS 186-5 A.2.1 Key Pair Generation Using Extra Random Bits [11]. For a 256-bit ECC key, the most-significant 40 bytes are used to generate the private key and, if the *nameAlg* of the derived object is SHA-256, the next-most-significant 32 bytes will be used for the *seedValue*.

In some cases, the number of bytes used for the *sensitive* value is indeterminate. This is because some of the generated values are unsuitable for the application and could need to be discarded. In such cases, bytes are taken from the 1024-byte buffer until suitable values for *sensitive* have been found and the next most significant bytes are used for *seedValue*.

Example:

Not all 64-bit values are suitable for use as DES keys. When the derivation process produces one of these values, the key will be discarded, and the next most significant bytes are taken from the 1024-byte KDF buffer.

When suitable values for both *sensitive* and *seedValue* have been extracted from the 1024-byte KDF buffer, the remaining bytes are discarded.

25.4.2 Caution on use of Derivation Parents

Users of Derived Objects are advised to ensure that *label* and *context* are not re-used between different objects derived from the same Derivation Parent.

If the same Derivation Parent, *label*, and *context* are provided in two different invocations of CreateLoaded, the Derived Objects resulting from the derivation will share the same keying material (that is, output from the KDF used to create the Derived Object's sensitive and seedValue). This is true even if two different templates are provided to CreateLoaded.

Authorization values and/or policies can be used to protect Derivation Parents from misuse by attackers. `TPM2_PolicyTemplate()` can be used to restrict the template(s) that can be used with a given Derivation Parent.

Although the TPM can produce attestations of Derived Objects (e.g., with `TPM2_Certify()`), these attestations are untrustworthy because sensitiveDataOrigin can never be SET for a Derived Object. Verifiers should always ensure that sensitiveDataOrigin is SET for attested objects.

25.4.3 Implementation Alternatives

There are various ways to produce an implementation that is compatible with the conceptual description above without actually having to generate 1024 bytes of data. Some examples are given here:

- An implementation may compute the number of bytes that will be needed to produce the *sensitive* and *seedValue* and the KDF would only need to generate that number of bytes. The bits parameter in the call to generate the data would still need to be 8192. In order to facilitate this type of implementation, the *CryptKDFa*() function in the Reference Code has a blocks parameter that limits the number of returned blocks, regardless of the size of the *sizeInBits* parameter.
- An implementation may generate blocks on a demand basis. This is fairly complex but allows the derivation process to be used as if it were any other RNG. To implement this process, the calling parameters of the KDF are saved in a structure so that they are available for multiple calls. This structure is passed to the functions that use a random number generator. When random bits are needed, the generator checks the type of the random number generator context and, if it contains KDF parameters, they are used in a call to the KDF. After each call, the counter value is incremented so that the net effect of generating one block at a time is the same as generating all of them at the same time. The additional

complexity of this implementation is that it is required that all of the bytes from the KDF be used in order, with none skipped. In order to deal with a call that does not use a full block, a buffer is added to the KDF structure in which residual bytes are saved. When a call is made to fetch bytes from the KDF, the residual buffer is checked first and any bytes in that buffer are returned before the KDF is called to produce additional bytes. If the KDF produces a block and not all bytes are returned, the residual bytes are placed in the buffer. This provides continuity of bytes as required by Clause 25.4.1.

25.5 Derivation Process

The derivation process is required to be the same for all TPMs. That is, with the same inputs, all TPMs will generate the same Derived Objects.

When generating a Derived Object, the TPM will create the entropy structure for a KDF and pass a pointer to the structure to the function that creates Objects. The algorithm for generating an Object is as described in Clause 24.7.

Note:

The method of generating RSA keys is highly variable and is normally chosen according to the constraints of the application. In some cases, compliance is the overriding factor and in others, performance can be the determining factor. Since no single algorithm seems to be optimum for all the constraints and it would not be acceptable to require that TPMs implement one RSA key generation for compliance and one for interoperability, the TCG has chosen not to support derivation for RSA keys.

26 Object Loading

26.1 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 can require that the USER role authorization for the Storage Parent be provided

26.2 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:

1. 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 Storage Parent.
2. 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.

3. 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 could have been created by software using inconsistent values. The integrity check could pass, but the attribute values could be unacceptable or inappropriate.

   1. If *fixedTPM* is SET, *fixedTPM* must also be SET in the Storage Parent.
   2. If *fixedParent* is CLEAR, then *fixedTPM* must also be CLEAR.

      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.

   3. If *restricted* is SET, only one of *sign* or *decrypt* can be SET.

26.3 Public-only Load

There are several cases, such as duplication or signature verification, 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.

A public-only load occurs when the *inPrivate* parameter to `TPM2_LoadExternal()` has a size of zero.

26.4 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 can be loaded so that the TPM can be used to validate a signature. A symmetric key can be loaded so that the symmetric engines of the TPM can 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.

Example:

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 Storage Parent but can 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 such as `TPM2_VerifySequenceComplete()` and `TPM2_VerifyDigestSignature()`.

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.

27 Context Management

27.1 Introduction

To allow the TPM to be shared among many applications, the TPM supports context management. The objects, sequence 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 Clause 27.3.1 and Clause 27.3.2). 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:

In the Reference Code, *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 Code, 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:

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:

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 TPM2B_CONTEXT_DATA may be defined by the vendor, but a saved context is required to have its integrity and confidentiality protected by cryptographic means. Parts 3 and 4 of this specification implement the normative requirements for providing confidentiality and integrity protection for saved contexts. These protections are described in more detail in subsequent parts of Clause 27.

Note:

The algorithms chosen for integrity and confidentiality protection of a saved context are vendor specific. However, the cryptographic strengths of the algorithms used are required to be the highest of any algorithm of the same type implemented on the TPM.

27.2 Context Data

27.2.1 Introduction

The data structure TPMS_CONTEXT returned by TPM2_ContextSave() contains context metadata as well as the actual context TPM2B_CONTEXT_DATA. The context metadata contains:

- a sequence number,
- a handle, *savedHandle*, and

  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.

The actual context contains:

- an integrity HMAC, and
- an encrypted data blob.

The structure of the metadata is normative. The internal structure TPMS_CONTEXT_DATA of the actual context is vendor specific. 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 this clause.

The structure of the context contains both confidential and non-confidential data. This specification requires encryption of the confidential data. The TPMS_CONTEXT structure is normative. The structure of the enclosed TPMS_CONTEXT_DATA is vendor-specific, and its confidential data must be encrypted.

27.2.2 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 Clause 27.3.1 and Clause 27.3.2). 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 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 described in more detail in Clause 27.5). 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 sequence number.

In the Reference Code, *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 Clause 27.5). 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.

27.2.3 Handle

The *savedHandle* 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 Code, 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 or sequence object. For an object, the *savedHandle* 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 *savedHandle* 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 *savedHandle* 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 Clause 29.4.6); 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. Therefore, the TRM can optimize by saving the Transient Object context only once.

27.2.4 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.

27.3 Context Protections

27.3.1 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 TPM2B_CONTEXT_SENSITIVE context before it is encrypted. When the context is loaded, the value of *sequence* is compared to the value in the loaded TPM2B_CONTEXT_SENSITIVE 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:

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:

$$(symKey,\ symIv)\ :=\ \mathbf{KDFa}(hashAlg,\ hProof,\ vendorString,\ sequence,\ handle,\ bits) \tag{50}$$

where

hashAlg	is a hash algorithm chosen by the vendor

(continued on next page)

(continued from previous page)

hProof	is the proof value associated with the hierarchy associated with the context
vendorString	is a value used to differentiate the uses of the KDF
sequence	is the sequence parameter of the TPMS_CONTEXT
handle	is the handle parameter of the TPMS_CONTEXT
bits	is the number of bits needed for a symmetric key and IV for the context encryption

Note:

The value of *vendorString* is required to be different from any other label string used in a **KDFa**() call. The Reference Code uses “CONTEXT_ENCRYPT”

Note:

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 Equation 50 are used to encrypt the object or session context

$$encContext := \mathbf{CFB}_{symAlg}(symKey, symIv, context) \quad (51)$$

where

$\mathbf{CFB}_{symAlg}$	is symmetric encryption in CFB mode using a symmetric algorithm chosen by the TPM vendor
symKey	is the symmetric key from Equation 50
symIv	is the IV from Equation 50
context	is the context being protected (a TPM2B_CONTEXT_DATA)

Note:

The *size* field and the *buffer* field of *context* are encrypted.

27.3.2 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 (see Part 2 for the description of TPM_PT_CONTEXT_HASH).

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:

$$data := resetValue \ \{\| \ clearCount\} \ \| \ sequence \ \| \ handle \ \| \ encContext$$

$$contextHMAC := \mathbf{HMAC}_{vendorAlg}(hProof, \ data) \tag{52}$$

where

$\mathbf{HMAC}_{vendorAlg}$	is HMAC using a vendor-defined hash algorithm
hProof	is the hierarchy proof as selected by the hierarchy parameter of the TPMS_CONTEXT
resetValue	is 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	is a counter value that is incremented on each TPM Restart 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	is the sequence parameter of the TPMS_CONTEXT
handle	is the handle parameter of the TPMS_CONTEXT
encContext	is the encrypted context blob

27.4 Object Context Management

When an object's context is saved, a copy of the object context is integrity protected, 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.

27.5 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 saved 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. If, for example, 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 has 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 has to indicate the limit while there are still three available session sequence numbers in the 'even' interval. This allows the sessions in the 'odd' interval to be loaded and saved with an 'even' interval session sequence 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 TPM Restart or TPM Resume. Saved session contexts are invalidated on a TPM Reset.

27.6 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 copied to persistent TPM NV memory with TPM2_EvictControl(). When made persistent, TPM2_FlushContext() and TPM2_Startup(TPM_SU_CLEAR) have no effect on the persistent copy of 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 evicted from persistent memory using either Owner Authorization or Platform Authorization. An object made persistent using Platform Authorization may only be evicted from persistent memory using Platform Authorization.

27.7 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.

27.8 Sequence Context Management

Unlike a session context (see Clause 27.5), a saved sequence object does not include replay protection. Thus, an attacker could undo a TPM2_SequenceUpdate() by context saving the sequence object context before the update and context loading the stale context after the update. The application can mitigate this attack by adding a length to the update packet. The verifier can check the length and detect a missing update.

Example:

Commands such as TPM2_HMAC_Start(), TPM2_MAC_Start(), or TPM2_HashSequenceStart() create a sequence object.

Note:

Using an HMAC session during TPM2_SequenceUpdate() can prevent an alteration of the update packet (by a man in the middle).

28 Attestation

28.1 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_GetCommandAuditDigest()` 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`.

28.2 Standard Attestation Structure

The contents of the standard attestation structure are described in Table 38.

Table 38: 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) 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 this specification.

(continued on next page)

(continued from previous page)

Parameter	Type	Description
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

28.3 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 (see Clause 33.7). 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.

28.4 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.

28.5 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:

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:

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.

28.6 X.509 Certificate Signing

Deprecated:

TPM2_CertifyX509() was deprecated in version 184. See Part 0.

TPM2_CertifyX509() signs an X.509-formatted certificate. Prior to constructing and signing an X.509 certificate, TPM2_CertifyX509() verifies that the key-to-be-certified is loaded in the TPM, and that some of the permissions in the key's proposed X.509 certificate are compatible with the key-to-be-certified.

A typical use of TPM2_CertifyX509() is the enrollment of any TPM key into an X.509 Public Key Infrastructure. TPM2_CertifyX509() signs an X.509 formatted certificate describing the TPM key, rather than enabling the TPM key to self-certify by creating an X.509 formatted Certificate Signing Request (CSR). This is because a CSR doesn't provide a way for a TPM to communicate to a Certification Authority that the TPM key is protected by a TPM, or the restraints on the TPM key that are enforced by the TPM. The X.509 formatted certificate signed by the TPM is inspected by a Certification Authority prior to the CA signing its own certificate describing the operations on the key that are approved by the CA. A simpler CA will use conventional X.509 methods to just verify the TPM's certificate and verify the X.509 certificate of the key that signed the TPM's certificate. A more sophisticated CA that understands TPMs will also interpret the TPM certificate's Extensions element, which describes the certified key's detailed TPMA_OBJECT attributes and indicates precisely what operations a TPM can and cannot perform on the certified key.

In a typical usage of TPM2_CertifyX509():

- First, the CA (or its proxy) loads a certifying signing key in the TPM. The certifying signing key is typically a key with the x509Sign attribute SET.
- Next, the CA uses the certifying key with TPM2_CertifyX509() to certify a TPM key. The DER-encoded partialCertificate parameter for TPM2_CertifyX509() describes the validity time for the certificate as well as the X.509 Name for the certifying and certified keys. Additionally, partialCertificate is required to contain a KeyUsage in the Extension field and may contain a TPMA_OBJECT Extension. TPM2_CertifyX509() verifies that the key is compatible with approved operations, constructs a complete X.509 RFC 5280-defined certificate, and signs that complete certificate.
- TPM2_CertifyX509() returns the part of the complete X.509 certificate that was constructed by the TPM, plus the TPM's signature over the complete certificate. This data and partialCertificate are assembled outside the TPM into a complete X.509 RFC 5280-defined signed certificate for inspection by the Certification Authority using conventional X.509 tools.
- Finally, the Certification Authority recertifies the TPM key using the CA's normal certifying key, for consumption by entities that trust the CA.

Certification Authorities should be wary of certificates signed by TPM2_CertifyX509() with keys that do not have the x509sign attribute SET, because an X.509 certificate can be signed using TPM2_Sign(). In that case, the TPM will not have verified that the certified key is loaded in the TPM and will not have verified that the certified key is compatible with the X.509 certificate. Nevertheless, an X.509 certificate signed by TPM2_CertifyX509() with an ordinary signing key or a restricted signing key may be acceptable when the CA trusts the entity controlling usage of that signing key: the CA itself may have exclusive control over the signing key, for example.

An x509sign key is even more restricted than a restricted key: a restricted key won't use TPM2_Sign() to sign any data that starts with the "magic" parameter, and an x509sign key won't use TPM2_Sign() to sign any data at all. Hence the command TPM2_CertifyX509() is the only way that an x509sign key can sign an X.509 certificate. An x509sign key will also sign any of the non-X.509 certification commands (e.g., TPM2_Certify(), TPM2_CertifyCreation(), TPM2_NV_Certify()), which sign data that start with the TCG "magic" value. This enables an x509sign key to be an Attestation Key with a certificate proving that the

x509sign key is bound to a single TPM and is safe because none of the certification commands sign data that could masquerade as a fraudulent certificate.

TPM2_CertifyX509() verifies that a key-to-be-certified is compatible with permissions granted by the X.509 certificate. Broadly speaking, the TPM verifies that the private key can decrypt if the certificate says the key is approved for decryption and verifies that the private key can sign if the certificate says the key is approved for signing. The TPM verifies that a key has its fixedTPM attribute SET if the certificate approves the key for non-repudiation/contentCommitment operations, because only the TPM that has a particular fixedTPM key can use that particular key.

TPM2_CertifyX509() cannot guarantee that a key will perform only the operations approved by an X.509 certificate. A TPM cannot control operations on the public part of a key and doesn't usually know why the key is being used, whereas a certificate can approve operations on the public part of a key and can approve the objective of operations on a key. Relying parties are cautioned that not all the information in a certificate signed using TPM2_CertifyX509() is validated by the TPM. The TPM validates the public key of the target, the KeyUsage, and the TPMA_OBJECT extended attributes.

Some partialCertificate inputs to TPM2_CertifyX509() cannot be verified by the TPM but must be present because they are necessary to create a signature over an X.509 certificate. One such input is the certifying signing scheme OIDs that are necessary to create the signature. The caller does not need to supply OIDs if the TPM is able to generate the OIDs for the certifying signing scheme. However, there are so many OIDs that the TPM is unable to generate all possible OIDs, and some OIDs have not yet been assigned. Thus, the caller must supply the OIDs for the certifying signing scheme if the TPM is unable to generate them. Should the caller supply OIDs, they take precedence over any OIDs that the TPM would have assigned.

29 Cryptographic Support Functions

29.1 Introduction

This clause describes the cryptographic primitives that may be provided by a TPM compliant to this specification.

29.2 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 Clause 29.4).

If the data used to create the digest does not have TPM_GENERATED_VALUE as its first octets, then the response to TPM2_Hash() or TPM2_SequenceComplete() will contain a ticket indicating that the digest may be signed with a restricted signing key.

Note:

The creation of the ticket may be suppressed by using TPM_RH_NULL as the hierarchy parameter in TPM2_Hash() or TPM2_SequenceComplete().

29.3 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.

29.4 Hash, MAC, Event, and Signature Sequences

29.4.1 Introduction

When the amount of data to be included in a request cannot or will not be sent to the TPM in one of the atomic commands (e.g., TPM2_Hash(), TPM2_HMAC()) then a sequence of commands can be used to provide incremental updates to the request.

There are five types of sequences:

1. Hash sequences, initialized with TPM2_HashSequenceStart() and completed with TPM2_SequenceComplete().
2. Event sequences, also initialized with TPM2_HashSequenceStart() and completed with TPM2_EventSequenceComplete().
3. MAC / HMAC sequences, initialized with TPM2_MAC_Start() / TPM2_HMAC_Start() and completed with TPM2_SequenceComplete().
4. Signature signing sequences, initialized with TPM2_SignSequenceStart() and completed with TPM2_SignSequenceComplete().
5. Signature verification sequences, initialized with TPM2_VerifySequenceStart() and completed with TPM2_VerifySequenceComplete().

Increments of data are added to a sequence using TPM2_SequenceUpdate().

TPM2_SequenceComplete(), TPM2_EventSequenceComplete(), and TPM2_SignSequenceComplete() (but not TPM2_VerifySequenceComplete()) can also be used to provide the last data to be included in the sequence. If the entire data can be provided at this time, then TPM2_SequenceUpdate() is not necessary.

29.4.2 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:

1. TPM2_HashSequenceStart() (*hashAlg* is a supported hash algorithm), followed by
2. TPM2_SequenceUpdate() (zero or more), followed by
3. TPM2_SequenceComplete()

29.4.3 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 SHA256 and a SHA384 bank, then the list will contain two digests.

An Event Sequence is:

1. TPM2_HashSequenceStart() (*hashAlg* is TPM_ALG_NULL), followed by
2. TPM2_SequenceUpdate() (zero or more) followed by
3. TPM2_EventSequenceComplete() (will do an Extend if *pcrHandle* is a PCR and not TPM_RH_NULL)

29.4.4 MAC / HMAC Sequence

For a MAC or HMAC sequence, the TPM will use the indicated key as the MAC or HMAC key and perform an MAC or HMAC computation over the data of the sequence using the specified hash algorithm or symmetric MAC.

TPM2_SequenceComplete() completes the MAC or HMAC sequence and returns the MAC or 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

A MAC or HMAC sequence is:

1. TPM2_MAC_Start() or TPM2_HMAC_Start() followed by
2. TPM2_SequenceUpdate() (zero or more) followed by
3. TPM2_SequenceComplete()

29.4.5 Signature Sequences

A signature sequence can be used to pass an arbitrarily large message to the TPM for signing or signature verification.

29.4.5.1 Signature Signing Sequence

A signature signing sequence is:

1. `TPM2_SignSequenceStart()`, followed by
2. `TPM2_SequenceUpdate()` (zero or more), followed by
3. `TPM2_SignSequenceComplete()`

Note:

For EdDSA signing, `TPM2_SequenceUpdate()` is not supported, and the entire message to be signed has to be passed into the `TPM2_SignSequenceComplete()` command. See Clause 44.3.2 for more details.

29.4.5.2 Signature Verification Sequence

A signature verification sequence is:

1. `TPM2_VerifySequenceStart()`, followed by
2. `TPM2_SequenceUpdate()` (one or more), followed by
3. `TPM2_VerifySequenceComplete()`

Note:

`TPM2_VerifySequenceComplete()` does not allow passing additional data into the sequence, so `TPM2_SequenceUpdate()` needs to be used at least once.

29.4.6 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 other Objects because the sequence context is updated on each use. In addition, unlike other 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()`, `TPM2_SequenceComplete()`, `TPM2_SignSequenceComplete()`, or `TPM2_VerifySequenceComplete()` 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.

29.5 Symmetric Encryption

`TPM2_EncryptDecrypt2()` 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.

Deprecated:

`TPM2_EncryptDecrypt()` was deprecated in version 1.38 in favor of `TPM2_EncryptDecrypt2()`. See Part 0 for more information.

29.6 Asymmetric Encryption and Signature Operations

The algorithm-specific clauses in Part 1 contain descriptions of the cryptographic encryption/decryption and signature primitives that are defined for each of the asymmetric algorithms supported by the specification.

30 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. 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 previous families of this specification.

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.

31 Hardware Core Root of Trust Measurement (H-CRTM) Event Sequence

31.1 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.

Note:

The interface is permitted to be designed such that it is not possible to interrupt this sequence.

31.2 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:

There is no special designation for when the H-CRTM occurs before `TPM2_Startup()`

Note:

The D-RTM sequence may be repeated one or more times after TPM2_Startup(). On each invocation of the D-RTM sequence, the RTM has to 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.

$$PCR[17][hashAlg] := \mathbf{H}_{hashAlg}(0...0 \parallel H_{hashAlg}(hash\_data)) \quad (53)$$

where

$hash\_data$	is 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.

31.3 H-CRTM before TPM2_Startup() and TPM2_Startup() without H-CRTM

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.

$$PCR[0][hashAlg] := \mathbf{H}_{hashAlg}(0...04 \parallel H_{hashAlg}(hash\_data)) \quad (54)$$

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*
$hash\_data$	is 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(TPM_SU_CLEAR) will not change the value of PCR[0]. Otherwise, TPM2_Startup(TPM_SU_CLEAR) will set PCR[0] to the locality of the TPM2_Startup() command.

If there is an H-CRTM event before a TPM2_Startup(TPM_SU_CLEAR), there must be an H-CRTM event before a subsequent TPM2_Startup(TPM_SU_STATE). The locality of the TPM2_Startup(TPM_SU_STATE) is not checked against the locality of the previous TPM2_Startup(TPM_SU_CLEAR)

If there is no H-CRTM event before TPM2_Startup(TPM_SU_CLEAR), there must be no H-CRTM event before a subsequent TPM2_Startup(TPM_SU_STATE) and the TPM2_Startup(TPM_SU_STATE) must have the same locality as the previous TPM2_Startup(TPM_SU_CLEAR).

Note:

If the H-CRTM sequence is sent multiple times before TPM2_Startup(), each will perform the action of Equation 54. That is, PCR 0 will be set back to 0…04 each time. However, the intent is that H-CRTM would be sent only once before TPM2_Startup()

32 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:

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 may 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:

$$audit_{\text{new}} := \mathbf{H}_{auditAlg}(audit_{\text{old}} \parallel cpHash \parallel rpHash) \tag{55}$$

where

$\mathbf{H}_{auditAlg}$	is the hash function using the currently selected audit hash algorithm
$audit_{\text{old}}$	is the previously computed audit digest
$cpHash$	is the command parameter hash using the audit hash
$rpHash$	is the response parameter hash using the audit hash

Note:

Clause 15.7 describes the process for computing *cpHash* and Clause 15.8 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:

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 a `TPM2_SetCommandCodeAuditStatus()` that changes the audit digest algorithm *auditAlg*.

An audit report structure contains the current value of the audit digest register and the value of the audit counter.

Note:

The signed audit structure is a `TPM2B_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:

The command audit session counter is incremented on the first auditable command in a session. This is typically 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.

33 Timing Components

33.1 Introduction

The TPM has timing components for use in time-stamping of attestations and for gating policy

Time is a free-running hardware value that is not under software control. *Time* advances when the *Time* circuit is powered and is reset to zero when power to the *Time* hardware is lost.

> ***Note:***
>
> Typically, the *Time* hardware will be powered down when the rest of the TPM is powered down.

Clock is a value that is derived from *Time* and advances as *Time* advances. *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*. The Safe flag indicates whether *Clock* might have been wound backwards, in which case the current *Clock* value would be unsafe. 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 (such as, the right to use or duplicate a key for 1 hour);
- 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.

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.

33.2 Time

Time is a 64-bit value that contains the time in milliseconds that the circuit providing *Time* has been powered.

Note:

Depending on the frequency of the TPM oscillator and the setting of the frequency divisor (`TPM2_ClockRateAdjust()`), the rate at which *Time* advances can be in error by as much as 32.5%.

Time is unaffected by `TPM2_ClockSet()`.

The circuit providing *Time* may be powered independently from the rest of the TPM. However, *Time* must be powered whenever the TPM is powered. The *Time* hardware needs to provide a reliable indication that it has lost power or has been reset. *Time* should not be reset unless the TPM requires a `_TPM_Init` indication before resuming operation.

Time need not advance continuously when powered. The *Time* hardware is required to provide a reliable indication if *Time* has stopped advancing.

33.3 Clock

33.3.1 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 advances at the rate that *Time* advances. 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 described in subsequent parts of this clause.

The accuracy of *Clock* is approximate. The causes of inaccuracy are

- 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 this specification, 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*.

33.3.2 *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* is 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* can 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. For example, assume that the update interval for *NV Clock* is the maximum allowed value (2^{22} milliseconds or approximately 70 minutes). Power may be removed from the TPM and *Time* 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 may 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 described in more detail in the following clause.

33.3.3 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:

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:

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.

Clock remains safe as long as *Time* is powered. That is, if there is a non-Orderly shutdown and the TPM is powered down but *Time* is powered, then *Clock* will be updated the next time the TPM starts. Since time is not lost, *Clock* will not appear to go backwards and *Safe* can be SET. During the time that the TPM is powered down it is not necessary for *Time* to advance, it simply needs not to be set to a lower time value.

33.3.4 *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. For example, assuming that the NV Clock update interval is 2^{12} (approximately every 4 seconds), the TPM may perform an update of *NV Clock* whenever the low-order 12 bits of volatile *Clock* are zero.

Note:

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:

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.

33.3.5 Setting *Clock*

The value in the volatile *Clock* may be set forward using TPM2_ClockSet(). The *newTime* parameter of TPM2_ClockSet() is required to have a greater value than the volatile *Clock*. So that policies that rely on *Clock* do not have to contend with the possibility of the value of *Clock* wrapping, *newTime* may not be greater than FF FF 00 00 00 00 00 00_{16}.

If TPM2_ClockSet() causes the volatile and non-volatile versions of *Clock* to differ by more than the implementation-dependent update interval, then *NV Clock* will be updated before TPM2_ClockSet() returns.

Note:

It is not necessary that all the bits of *NV Clock* be updated. Only the bits of *NV Clock* that are updated in the normal update process need to be changed.

Example:

Assume the update of *NV Clock* occurs every 2^{12} milliseconds (00 00 10 00_{16}), that the low-order 32 bits of *NV Clock* are 00 00 00 00_{16} and *Clock* are 00 00 0F 00_{16}, and that a *newTime* advances *Clock* to 00 00 11 00_{16}. Since this makes the difference between *Clock* and *NV Clock* more than the update interval (2^{12}), *NV Clock* is updated to 00 00 11 00_{16}.

The expected management for *Clock* is for a coarse (large) update to be made after TPM2_Startup() in order to recover the time lost when the TPM was not powered. After that single large change, *Clock* is expected to be updated with relatively small values to keep it synchronized with real time. If software manages *Clock* in this manner, TPM2_ClockSet() will not have to be throttled in order to avoid NV wear-out.

Note:

System software may purposely cause the rate of *Clock* advance to be slower than real time and just make minor adjustments when an attestation of some sort is required. If managed in this way, TPM2_ClockSet() may be executed many times between update intervals. Because update of the NV portion of *Clock* is not allowed unless the difference between the two versions is at least as large as the update interval, TPM2_ClockSet() will not need throttling to avoid wear-out.

Note:

The specification could have been written so that TPM2_ClockSet() would never invoke NV throttling. That is, the value for *newTime* could have been set such that the rate of *NV Clock* update would be at an acceptable rate or TPM2_ClockSet() would fail. This logic is complex, and under normal circumstances, redundant. As a consequence, the specification does not place restrictions on the values of *newTime* other than those listed above. The fact that TPM2_ClockSet() requires Owner Authorization or Platform Authorization will provide some level of protection against an attacker using TPM2_ClockSet() for a wear-out attack on the TPM. TPMs can implement wear-protection if extraordinary rates of update are observed.

33.3.6 *Clock* Periodicity

The TPM clock may be driven by an internal or external frequency source or be derived from a time source supplied by its operating environment. TPM profiles shall specify the time source to be used and the required accuracy.

External software may make limited adjustments to the rate of advance of *Clock* to provide a better approximation to real time.

This specification requires that the nominal rate of advance of *Clock* when powered is within 15% of the rate of UTC. If the external clock is not reliable, the TPM must provide its own clock with the necessary accuracy. External software may indicate that *Clock* is not advancing at the rate of UTC and that the rate needs to be increased or decreased. The command to adjust the clock rate is TPM2_ClockRateAdjust(). The *newRate* parameter of this command allows fine or coarse upward or downward adjustments to the current counting rate. This specification does not define coarse or fine adjustment percentages, and software that manages the TPM must infer this from observed behavior.

The range of adjustment of the rate is dependent on the design of the TPM. It is required that the variation in the rate be large enough that it will allow software to adjust the rate of *Clock* advance to be the same as UTC. The TPM should not allow rate adjustments that are larger than the design tolerance of the TPM.

Example:

A TPM is designed to have a nominal internal oscillator frequency of 10 MHz with a tolerance of +/-15% and a presettable counter that is used to count the oscillator clocks and generate an output every second that is used to advance *Clock*. To cover the tolerance of the oscillator, the preset for the counter would have to be between 8,500,000 and 11,500,000.

Example:

A TPM is designed as above but with the additional ability to accept an outside frequency reference as long as that reference is at least +/-15%. If the external source is more accurate than +/-15%, then the TPM may still allow an adjustment over the 8,500,000 to 11,500,000 range.

Note:

In the worst case, an attacker who knows either the Platform Authorization or Owner Authorization value may be able to make the TPM run 32.5% (1.15^2) fast or slow. However, an attacker who knows the Platform Authorization or Owner Authorization could also set *Clock* arbitrarily far into the future.

An error is returned if external software tries to adjust the clock rate outside specified bounds.

The TPM may store adjustments to the nominal clock rate in volatile memory. If it does, then adjustment should only be stored on an orderly shutdown and not during the actions of TPM2_ClockRateAdjust(). That is, the adjustment value should be in volatile memory and only saved to nonvolatile memory on an orderly shutdown.

Note:

This constraint on TPM2_ClockRateAdjust() is so that software may make changes to the rate at arbitrarily high rates without causing an NV event that might require throttling.

33.4 resetCount

The *resetCount* is a non-volatile, 32-bit counter that is incremented on a successful TPM Reset. It may be read using TPM2_ReadClock() and be used in an authorization policy (TPM2_PolicyCounterTimer()). Additionally, the contents of the *resetCount* are included in the attestation data for any of the attestation commands.

Note:

Depending on the hierarchy of the signing key, the value of *resetCount* can be obfuscated so that a verifier can tell that the counter has changed but cannot know the absolute value of the counter.

The purposes of *resetCount* are to indicate when the static trust state of the platform may have changed and to indicate a possible discontinuity in *Clock*.

Example:

Without the *resetCount*, the sequence - (1) attest to trusted values (2) transition to an untrusted state (3) perform a transaction (4) TPM Reset (5) attest to trusted value - would hide the fact that the transaction may have occurred during an untrusted state.

Note:

Since the volatile *Clock* is reloaded from the *NV Clock* on each _TPM_Init, the volatile *Clock* will lose some time in nearly all circumstances.

resetCount is incremented whenever the TPM starts up and all previous state is lost (i.e., on a TPM Reset). *resetCount* is set to zero in TPM2_Clear().

33.5 restartCount

In addition to TPM Reset, other events may cause a discontinuity in TPM-recorded time or in the Root of Trust for Reporting (RTR). A suspend-resume cycle will cause a time discontinuity. _TPM_Hash_Start can cause an RTR discontinuity in the dynamic Root of Trust for Measurement (D-RTM) PCR. The *restartCount* is used to provide an indication of these discontinuities.

The *restartCount* is a non-volatile, 32-bit counter that increments when the TPM executes TPM Resume, TPM Restart, or _TPM_Hash_Start. Since *resetCount* increments on each TPM Reset, the combination of *resetCount* and *restartCount* accounts for the cases when a discontinuity may occur, allowing TPM *Time* to fall behind real time.

Note:

When software sets *Clock* forward, that is a positive time discontinuity under control of software. The negative discontinuities of *Clock* are due to hardware actions that may be outside of the control of software.

The combination of *resetCount* and *restartCount* also accounts for the discontinuities of the RTR. A change in *resetCount* indicates a discontinuity in the static RTR, and a change in *restartCount* indicates a change in the dynamic RTR.

restartCount is reset to 0 on TPM Reset - when *resetCount* is incremented. This does not cause a loss of information about the dynamic RTR because a change to *resetCount* also implies a change to the dynamic RTR.

33.6 Note on the Accuracy and Reliability of *Clock*

Clock is designed to allow a managed environment, such as enterprise, to maintain a small deviation between *Clock* and real time. If the platform is not managed, if the platform falls into the hands of an adversary, or if the platform is controlled by malware, then accuracy of *Clock* is diminished. This note addresses considerations that influence the applicability of *Clock* for time-stamping and for time-limited objects.

This analysis assumes that the TPM is not physically attacked, but that adversaries may manipulate external software and local clocks like the CMOS clock on PC platforms.

It is assumed that, under normal operation, external software adjusts *Clock* at platform startup and subsequently makes occasional additional rate and forwarding adjustments to ensure that *Clock* remains within acceptable tolerances. Enterprise management servers or web services may occasionally request time-stamped nonces to check that *Clock* meets network policy.

If *Clock* is used to time-stamp event log entries, then server software should ensure that *Clock* is accurate (as described above), and client software may occasionally record TPM *Time* values counter-signed by external authoritative time-stamping services to provide fiduciary time markers. These services may include the *Clock* and *Time* values as well as the initialization counters (*resetCount* and *restartCount*). The minimal security guarantees provided by the TPM in this case are

- proper ordering of events logged at times greater than 1 millisecond apart (apart from when associated with discontinuities in the *resetCount* and *restartCount*), and
- that time stamps can never be forged to indicate a time in the past. If the value of *Clock* could be “stale,” *Safe* will indicate as much. If *Clock* has occasionally been reported to other authorities or has been counter-signed, then the accuracy of the other time stamps can be interpolated more accurately.

If *Clock* is used to lifetime-limit objects, then when the platform is properly managed, objects will become inaccessible with temporal accuracy related to the precision of clock management and the update interval of *NV Clock*. If the lifetime has the granularity of *NV Clock* update, then once it becomes inaccessible, it cannot be recovered because, at that granularity, *Clock* will not move backward. If the granularity of the lifetime needs to be shorter than the update interval of *Clock*, then the *Safe* flag can be checked to see if the value of *Clock* may be “stale” or not.

If the platform falls into adversarial hands, the attacker will never be able to recover already revoked objects. However, for objects with lifetimes in the future, an adversary may effectively stop the passage of time so that objects never expire.

Example:

To make TPM *Time* “stop,” the platform might be turned on briefly to access the time-limited object and then turned off in a way that prevents an orderly shutdown of the TPM. If the TPM is left on for less than the update interval and the platform does not have an orderly shutdown, *Clock* will continue to repeat values within the range of an update interval. In a managed environment, a platform with a *Clock* that has a value that is substantially different from real time will likely be denied further network services. For a system in an unmanaged environment, a more complex policy using *resetCount* and *Time* may be used to limit access to objects even if time does not advance (for example, the policy may allow access for 20 minutes or 2 reboots).

When the owner of the platform changes (new SPS generated) *Clock* is reset to zero. Using *Clock* to do time stamping with a non-duplicable key does not constitute a vulnerability because the signing key also becomes inaccessible when the owner changes, so no new events can be created. If the time-stamping key is duplicable, then a more detailed security analysis is needed - for instance, examination of the Qualified Name in the signing structure.

If *Clock* is used in other policy settings, similar considerations apply. If an object is destroyed when the owner is changed, then *Clock* reset is benign. However, if an object survives an owner change (such as, an NV Index created by the platform), then use of *Clock* in its access policy may not be appropriate.

33.7 Privacy Aspects of Clock

The attestation structures return several values that, when taken together, may be sufficiently unique to identity a specific platform. For example, the difference between *Clock* and *Time* is, during the interval of a boot, likely to be somewhat unique for a platform. When combined with *resetCount* and *restartCount*, the values can become very indicative of a specific platform. If these values allow signatures from two keys to be correlated, then those keys remain correlated as long as they are in use. The TPM uses authorizations and obfuscation values to prevent this type of unwanted correlation.

All attestations contain a `TPMS_CLOCK_INFO` structure. That structure contains *Clock*, *resetCount*, *restartCount*, and *Safe*. The attestation structure also contains a 64-bit value that is indicative of the firmware version number. When these values are going to be signed by a key that is not in the Platform or Endorsement hierarchy, *resetCount*, *restartCount*, and firmware version number have a key-specific value added to them before they are put into the attestation structure. The addition allows the determination of change in values but prevents disclosure of the exact value.

Each Attestation Key has a different 128-bit obfuscation value that is constant for the lifetime of the key. It is computed by:

$$obfuscation := \mathbf{KDFa}(signHandle{\rightarrow}nameAlg,\ shProof,\ \text{"OBFUSCATE"},\ signHandle{\rightarrow}QN,\ 0,\ 128)$$

34 NV Memory

34.1 Introduction

Each TPM is required to have some non-volatile memory. This memory is used to retain values across power events. The NV memory is used to hold:

- NV Index values,
- objects made persistent by TPM2_EvictControl(),
- state saved by TPM2_Shutdown(), and
- Persistent NV data.

34.2 NV Indices

34.2.1 Definition

An NV Index is space that is defined by a user of the TPM. The Index is identified by a unique handle value. An NV Index handle has an MSO of TPM_HT_NV_INDEX.

The NV Index structure has:

- An identifying handle - this handle is assigned by the caller when the Index is defined and is used to reference the Index. The handle associated with an Index has an MSO of TPM_HT_NV_INDEX.
- A *nameAlg* - this parameter indicates the hash algorithm used in the computation of the Name of the Index (see Clause 13).
- An authorization policy - this parameter is optional and is the digest of the policy for the NV Index. For the policy to apply to an operation, the corresponding TPMA_NV_POLICY_READ, TPMA_NV_POLICY_WRITE, or TPMA_NV_POLICY_DELETE attribute needs to be SET. Different policies for read, write, and delete can be achieved using policy OR terms and TPM2_PolicyCommandCode().
- A set of NV Index attributes - this parameter determines the nature of the Index and who may manipulate or read the Index.
- An authorization value that is no larger than the size of the digest produced by the *nameAlg* of the NV Index.
- A value indicating the size of the Index data - this parameter indicates the number of octets that are required to hold the NV data. For some Index types, the size is fixed.
- The NV Index data that may be modified according to the type of the NV Index.

All the parts of the NV Index structure, except for the *authValue* and Index data, constitute the public portion of the Index. They are hashed using the *nameAlg* to produce the Name of the Index.

The public area of the Index may be read using TPM2_NV_ReadPublic().

Note:

TPM2_NV_ReadPublic() also returns the Name of the NV Index.

An NV Index can be designated as a hybrid Index. A hybrid Index is intended for applications where frequent updates are expected. High frequency updates are generally not compatible with the technology currently used for nonvolatile storage on a TPM. A hybrid Index maintains a volatile (RAM) and a non-volatile copy of its Index data. A write to a non-hybrid Index is immediately written to NV memory but a write to a hybrid Index only updates the copy of the Index data in RAM. See Clause 34.2.4.3 for hybrid counter operation. The non-volatile copy of a hybrid NV Index is updated on TPM2_Shutdown().

If an NV Index has TPMA_NV_ORDERLY SET, then it is a hybrid Index.

Note:

The user of a hybrid NV Index will understand that data may be lost if the TPM does not shut down in an orderly fashion so that the volatile data can be written to NV memory.

Whether or not NV Index is a hybrid, when an NV Index is defined (TPM2_NV_DefineSpace()), the persistent values of the NV Index are written to NV if the command completes successfully.

Any NV Index type can be defined as a hybrid. The conditions under which the write to NV memory occur vary and are described below.

Note:

An implementation is not required to support an arbitrary number of hybrid indices and is not required to support any ordinary hybrid Index with a size of more than eight octets.

34.2.2 NV Index Allocation

An NV Index is allocated with TPM2_NV_DefineSpace(). Either Platform Authorization or Owner Authorization is required in order to allocate an Index. The caller indicates the NV Index to assign to the NV location, the access controls for the Index, and the type and or size of the data buffer that should be reserved for writing. While the allocation process does write the metadata for the Index to NV, it does not write to the data area of the Index data and a read of the NV location before it is written will return an error (TPM_RC_NV_UNINITIALIZED).

When an NV Index is defined (TPM2_NV_DefineSpace()), its TPMA_NV_WRITTEN attribute will be CLEAR. Until the Index is written by a party that can satisfy the write policy, the Index is defined but has no data, and TPM2_PolicyNV() and TPM2_NV_Read() will fail.

TPMA_NV_WRITTEN is SET when an authorized party first writes the Index. This permits a relying party to know that the value in the Index was written by an authorized party. It is not simply a default value that was present when the Index was defined (or deleted and redefined to attempt a roll back.)

A replying party can read the Index attributes and policy, which are public, to determine the authorized party.

Note:

The metadata of an NV Index is the data relating to the NV Index description (Index number, policy, attributes, data size, and *authValue*) along with any additional information that the TPM needs to manage the NV Index memory.

Different types of NV Index may be supported.

- **Ordinary** - an Index with an NV Index type of TPM_NT_ORDINARY contains data that is opaque to the TPM that is modified using TPM2_NV_Write().
- **Counter** - an Index with an NV Index type of TPM_NT_COUNTER contains a 64-bit counter that is modified using TPM2_NV_Increment().
- **Bit field** - an Index with an NV Index type of TPM_NT_BITS contains 64 bits that are initialized to 0 and are modified using TPM2_NV_SetBits().
- **Extend** - an Index with an NV Index type of TPM_NT_EXTEND contains a value that has behavior similar to a PCR and is modified using TPM2_NV_Extend().
- **PIN Fail** - an Index with an NV Index type of TPM_NT_PIN_FAIL that contains a TPMS_NV_PIN_COUNTER_PARAMETERS structure that is modified using TPM2_NV_Write() or by using the *authValue* of the Index. *pinCount* is reset when an authorization attempt using *authValue*

succeeds. *pinCount* is incremented after an authorization attempt using *authValue* fails. *pinCount* cannot increment beyond *pinLimit* because *authValue* authorization is locked out if *pinCount* >= *pinLimit*. A Pin Fail Index can be modified with TPM2_NV_Write().

- **PIN Pass** - an Index with an NV Index type of TPM_NT_PIN_PASS that contains a TPMS_NV_PIN_COUNTER_PARAMETERS structure that is modified using TPM2_NV_Write() or by using the *authValue* of the Index. *pinCount* is incremented after an authorization attempt using *authValue* succeeds. *pinCount* cannot increment beyond *pinLimit* because *authValue* authorization is locked out if *pinCount* >= *pinLimit*. A Pin Pass Index can be modified with TPM2_NV_Write().

TPM2_NV_DefineSpace() can fail if an Index with the requested handle already exists or if there is insufficient NV memory for the allocation. Creation of a hybrid Index will fail if there is insufficient RAM available for the allocation. The command will fail if an Index type is not supported.

Example:

If the TPM does not implement TPM2_NV_Extend(), then the TPM will not allow creation of an NV Index that has the TPM_NT_EXTEND attribute.

If the Index to be created has its TPMA_NV_POLICY_DELETE attribute SET, then platform authorization is required for allocation. This attribute is only allowed to be selected if TPM2_NV_UndefineSpaceSpecial() is implemented on the TPM.

Note:

This attribute indicates that a policy is required to delete the Index. It permits creation of an Index that can never be deleted. One example is an Empty Policy, which can never be satisfied. Another example is a policy that does not include TPM2_PolicyCommandCode() with TPM_CC_NV_UndefineSpaceSpecial(). Requiring platform authorization protects against the current TPM owner creating such an Index.

34.2.3 NV Index Deletion

An NV Index can be removed using either TPM2_NV_UndefineSpace() or TPM2_NV_UndefineSpaceSpecial().

If the TPMA_NV_POLICY_DELETE attribute is SET, then the Index can only be deleted if ADMIN role authorization is provided. ADMIN role authorization is provided by a policy session with the *commandCode* of the policy set to TPM2_NV_UndefineSpaceSpecial().

TPM2_NV_UndefineSpace() is used to delete other Indices from the NV. The authorization given for deleting the Index is required to be the same as the authorization given to allocate the Index.

TPM2_Clear() will remove any NV Index that used Owner Authorization to define the Index. TPM2_Clear() uses either TPM_RH_LOCKOUT or TPM_RH_PLATFORM.

TPM2_ChangePPS() does not cause any NV Index to be removed.

To comply with FIPS-140, the data contents and authorization value must be zeroized when the NV Index is deleted.

34.2.4 High-Endurance (Hybrid) Indices

34.2.4.1 Description

Some applications need the ability to make frequent updates to non-volatile values such as monotonic counters. A high update rate is generally not compatible with the technology currently used for non-volatile storage on a

TPM. To allow the TPM to support high-update rates while protecting the endurance of the NV memory, a hybrid Index type is defined.

When an NV Index is defined with the TPMA_NV_ORDERLY attribute SET, the TPM will allocate the required NV memory as well as space in TPM RAM for the data value. Updates to the Index will modify the RAM copy of the Index data with updates to the NV on Shutdown() or whenever the RAM copy of a counter is divisible by a set modulus. In some cases, the data write may never occur.

Note:

The value of the modulus is implementation specific and can be accessed using TPM2_GetCapability(*capability* == TPM_CAP_TPM_PROPERTY, *property* == TPM_PT_ORDERLY_COUNT). The returned value is the modulus - 1. This value is referred to as MAX_ORDERLY_COUNT.

If the TPMA_NV_ORDERLY attribute of an Index is SET, the TPM will perform special processing on the Index at TPM2_Startup(). The processing is dependent on the type of the Index.

34.2.4.2 Hybrid Indices Other than Counter Indices

For hybrid Indices that are not Counters, the NV Index data in volatile RAM memory is copied to non-volatile memory on a Shutdown(STATE), The data need not be copied to non-volatile memory on Shutdown(CLEAR).

1. On TPM Resume, the non-volatile copy of the Index data is copied into the volatile version of the NV Index data.
2. On TPM Reset, the TPMA_NV_WRITTEN attribute will be initialized to CLEAR. On a subsequent update of the Index, it will be initialized before it is updated.
3. On TPM Restart, if TPMA_NV_CLEAR_STCLEAR is SET, the NV Index is initialized as in b) above. If TPMA_NV_CLEAR_STCLEAR is CLEAR, then the NV Index is initialized as in a) above.

Note:

TPMA_NV_CLEAR_STCLEAR cannot be SET if the NV Index type is TPMA_NV_COUNTER. Counters are either restored (on an orderly startup) or set to a higher value (on a non-orderly startup).

34.2.4.3 Counter Hybrid Indices

The hybrid counter Index is designed so that it will be monotonically increasing and not miss an increment command regardless of the type of shutdown or startup.

For a Counter NV Index with the TPMA_NV_ORDERLY attribute, Index data in non-volatile memory is written to NV on any Shutdown().

Note:

For a Counter (or any other Index) that has TPMA_NV_ORDERLY CLEAR, non-volatile memory is written on any update of the NV Index.

On any orderly startup of the TPM (TPM2_Startup() following an orderly shutdown), the NV value of a hybrid counter Index will be copied to the RAM version. The count will be able to continue without any discontinuity.

On a non-orderly startup, the value of the counter in NV is adjusted before it is copied to RAM. A counter is adjusted by logical OR of the value of MAX_ORDERLY_COUNT to the NV value. This sets the RAM version of the counter to the maximum value it could have had before being updated due to the modulus test. This ensures that the RAM counter value is no less than any previously used counter value.

Example:

Assume that MAX_ORDERLY_COUNT contains 0F FF_{16} and that the TPM lost power without an orderly shutdown. On a startup, if an orderly counter is found to have a value of 00 00 00 00 00 01 73 $A1_{16}$, then the RAM version is updated to 00 00 00 00 00 01 7F FF_{16}.

Note:

When the RAM version of the counter is set this way, it is not necessary to immediately update the counter to NV. If the counter is incremented, then it will be automatically saved to NV when the low bits become zero.

Note:

If the RAM counter were initialized so that the low bits were zero and a subsequent un-orderly shutdown occurred, the counter would have to be advanced again, whether it had been incremented or not. By setting the counter to the maximum value before NV update, there is no need to advance the count on a subsequent unorderly shutdown unless the counter was used.

34.2.5 Reading an NV Index

Read access to an NV Index is provided with TPM2_NV_Read(), TPM2_NV_Certify(), and TPM2_PolicyNV(). For all of these commands, read authorization is required. The attributes of the Index determine what authorizations are allowed. TPMA_NV_PPREAD allows the Index to be read using Platform Authorization; TPMA_NV_OWNERREAD allows the Index to be read using Owner Authorization; TPMA_NV_AUTHREAD allows the Index to be read using the *authValue* of the Index; and TPMA_NV_POLICYREAD allows the Index to be read if the *authPolicy* of the Index is satisfied.

At least one of TPMA_NV_PPREAD, TPMA_NV_OWNERREAD, TPMA_NV_AUTHREAD or TPMA_NV_POLICYREAD needs to be SET or the TPM will not allocate the Index.

An access control (TPMA_NV_READ_STCLEAR) allows reading of the Index to be temporarily blocked. When this attribute is SET, TPM2_NV_ReadLock() may be used to temporarily disable read access to the Index. When the Index has been locked for read, the TPMA_NV_READLOCKED attribute of the Index will be SET. TPMA_NV_READLOCKED will be CLEAR on the next TPM Reset or TPM Restart. If the TPMA_NV_READLOCKED attribute is SET when the Index is read, the TPM returns TPM_RC_NV_LOCKED.

The *authPolicy* of the NV Index may be constructed such that it only applies for reading or for writing. It may be constructed to allow general reading and limited writing or general writing and limited reading. If reading or writing of the Index is to be restricted based on PCR values, then read authorization needs to use *authPolicy*.

34.2.6 Updating an Index

34.2.6.1 Introduction

The command used to update an Index is determined by the NV Index type. TPM2_NV_Write() is used to modify an Ordinary Index or a PIN Index, TPM2_NV_Increment() is used to modify a Counter Index, TPM2_NV_SetBits() is used to modify a Bit Field Index, and TPM2_NV_Extend() is used to modify an Extend Index. For all of these commands, write authorization is required.

The attributes of the Index determine what authorizations are allowed. TPMA_NV_PPWRITE allows the Index to be modified using Platform Authorization; TPMA_NV_OWNERWRITE allows the Index to be modified using Owner Authorization; TPMA_NV_AUTHWRITE allows the Index to be modified using the *authValue* of the Index; and TPMA_NV_POLICYWRITE allows the Index to be modified if the *authPolicy* of the Index is satisfied.

At least one of TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE or TPMA_NV_POLICYWRITE needs to be SET or the TPM will not allocate the Index. For a PIN Index, TPMA_NV_AUTHWRITE may not be SET and at least one of the other three write methods is required to be selected.

Note:

A method other than TPMA_NV_AUTHWRITE is required for a PIN Index because the *authValue* of a PIN Index is not accessible until the Index is written.

If the access control attribute TPMA_NV_WRITEDEFINE is SET, TPM2_NV_WriteLock() or TPM2_NV_GlobalWriteLock() may be used to permanently disable modify access to the Index. When the Index has been locked for modify, the TPMA_NV_WRITELOCKED attribute of the Index will be SET. This attribute will remain SET until the Index is deleted (TPM2_NV_UndefineSpace()).

If TPMA_NV_WRITEDEFINE is CLEAR, the TPMA_NV_WRITELOCKED attribute can be SET using TPM2_NV_WriteLock() if TPMA_NV_WRITE_STCLEAR is SET or TPM2_NV_GlobalWriteLock() if TPMA_NV_GLOBALLOCK is SET. In this case, TPMA_NV_WRITELOCKED will be CLEAR on the next TPM Reset or TPM Restart.

Note:

If TPMA_NV_WRITELOCKED is SET, but TPMA_NV_WRITTEN is CLEAR, then TPMA_NV_WRITELOCKED is CLEAR by TPM Reset or TPM Restart. This is true even if the TPMA_NV_WRITEDEFINE attribute is set. It prevents an NV Index from being defined that can never be written and permits a use case where an Index is defined, but the user wants to prohibit writes until after a reboot.

If the TPMA_NV_WRITELOCKED attribute is SET when an attempt is made to modify the Index, the TPM returns TPM_RC_NV_LOCKED.

For a PIN Fail Index, the TPM will return TPM_RC_NC_ATTRIBUTES if TPMA_NV_NO_DA is CLEAR.

34.2.6.2 NV Ordinary Index Update

TPM2_NV_Write() is used to modify the contents of an ordinary Index. The modification may be to the entire Index or, if the Index attributes allow (TPMA_NV_WRITE_ALL CLEAR), the size of the data to write can be as small as zero octets.

When a partial write is allowed, the *offset* parameter of TPM2_NV_Write() may be non-zero or the *size* of the data parameter may be less than the *size* of the Index. The TPM checks the TPMA_NV_WRITTEN attribute. If it is CLEAR, then the TPM will initialize the remainder of the Index to either all zero or all one. Alternatively, the TPM can initialize the entire Index at the time the Index is defined.

If the sum of the size of the *data* parameter and the *offset* parameter in TPM2_NV_Write() is greater than the size of the Index, then the TPM will not perform the write and will return an error.

On any TPM2_NV_Write() (including a size of zero), if the modification is successful, then the TPMA_NV_WRITTEN attribute of the Index will be SET. Any octets not initialized by the first write will have a value of all zero or all one.

Example:

If the Index is defined to contain 2 octets, and the first write of the Index is a single octet of 55_{16}, to offset 0, then the next read of the full Index will return 55 00_{16}.

If the Ordinary Index has the TPMA_NV_ORDERLY attribute, then only the RAM version of the Index is written. The data is preserved on a Shutdown(STATE).

34.2.6.3 NV Counter Index

When an Index has the TPMA_NV_COUNTER attribute, it behaves as a monotonic counter and may only be modified using TPM2_NV_Increment().

When an NV counter is created, it has no value and the TPMA_NV_WRITTEN attribute will be CLEAR.

On each TPM2_NV_Increment() the TPM checks the TPMA_NV_WRITTEN attribute of the Index. If it is CLEAR, then the TPM will initialize the 8-octet counter value such that the first increment will set a value that is greater than any value that a counter Index with the same Name has had over the lifetime of the TPM. The TPMA_NV_WRITTEN attribute will be SET.

Note:

This check ensures that an Index cannot be deleted and another Index with the same Name defined with a lower value.

Note:

The Reference Code implements this by tracking and using the largest count of any deleted NV Counter. An alternative implementation could track the largest count of any NV Counter, deleted or currently defined.

After checking TPMA_NV_WRITTEN and performing any required initialization operations, the TPM will increment the Counter.

Note:

The TPM will need to maintain a largest-count value. It is not necessary to update this value except when a NV Index is deleted. If the NV Index being deleted has the largest value held by an NV Index, then this value would be copied to the largest-count value. The value of an NV Counter Index after the first increment is larger than the largest-count value.

Note:

Since no counter can ever repeat a previous value ever contained in any NV Counter Index, a counter with a particular Name cannot be rolled back by deleting it and redefining it.

If the TPMA_NV_ORDERLY attribute is CLEAR, the increment will occur on the NV version of the counter (no RAM version exists). If the TPMA_NV_ORDERLY attribute is SET, the increment will occur on the RAM version of the counter, and if this causes a rollover, the NV version of the counter is updated. However, if TPMA_NV_WRITTEN is CLEAR, the NV version of the counter is also written. Once SET, TPMA_NV_WRITTEN of a counter is never CLEAR.

An Index may be defined with the TPMA_NV_ORDERLY attribute to indicate that the Index is expected to be modified at a high frequency and that the data is only required to persist when the TPM goes through an orderly shutdown process. For a counter, it also means that it will be written to NV when the counter has reached some threshold value. The threshold value for counters (MAX_ORDERLY_COUNT) is implementation dependent and can be read using TPM2_GetCapability(*capability* == TPM_CAP_PT, *property* == TPM_PT_ORDERLY_COUNT). This property has one of 32 values that can be expressed as $(2^N - 1)$ where N is between 1 and 32.

Example:

If MAX_ORDERLY_COUNT is 00 00 0F FF_{16}, then whenever the RAM version of a counter is incrementing, causing the low-order 12 bits to be zero, the NV version of the counter is updated.

The meaning of this threshold value is that when the counter is incremented such that the counter value ANDed with MAX_ORDERLY_COUNT is zero, then the NV version of the counter will be updated.

Note:

Another way to express this is to simply say that the NV version of the counter will be updated when the low order bits of the counter “roll-over”.

The TPM is required to ensure that, when an NV Counter is read, its value is not less than a previously reported value of the counter. That is, it may not go backward. If the shutdown was orderly, then, regardless of the type of the NV Counter, the NV value of a counter will not be less than the last reported value. If the shutdown was not orderly and the NV Counter has the TPMA_NV_ORDERLY attribute, then a value of the Counter may have been read from the RAM version of the counter but the NV version may not have been updated. To handle this case, if the TPMA_NV_ORDERLY attribute of an NV Counter is SET, and the TPM shutdown was not orderly, then, at TPM2_Startup() the TPM will OR the value of MAX_ORDERLY_COUNT to the contents of the non-volatile counter and set that as the current count in the RAM version of the counter.

Note:

The TPM has to prevent a rollback attack caused by a counter being deleted and then being recreated with a lower value. To do this, the TPM may keep track of the value of the highest count of a deleted counter using a phantom counter. When a counter is deleted, the current value of the counter is compared to the current phantom counter and other counters. If the value is larger than the phantom counter and other counters, the phantom counter is updated. When a new NV counter is created, it starts with the highest value of all the counters, including the phantom counter.

For a Counter with the TPMA_NV_ORDERLY attribute SET, the NV copy of the data will be updated whenever a specified number of low order bits of the RAM copy become all zeros. That number of low order bits is TPM implementation-dependent. The setting for a TPM may be found using TPM2_GetCapability(*capability* == TPM_CAP_TPM_PROPERTIES, *property* == TPM_PT_ORDERLY_COUNT). That capability is MAX_ORDERLY_COUNT.

The TPMA_NV_CLEAR_STCLEAR attribute has no effect on an NV Counter Index and it may be SET or CLEAR in the template.

34.2.6.4 NV Bit Field Index

When an Index has the TPMA_NV_BITS attribute it may only be modified by TPM2_NV_SetBits().

When an NV Bit Field Index is created, it has no value and the TPMA_NV_WRITTEN attribute will be CLEAR.

On each TPM2_NV_SetBits(), the TPM will check the TPMA_NV_WRITTEN attribute of the Index. If it is CLEAR, the TPM will set the 64 bits of the Index to zero. The TPM will then SET the TPMA_NV_WRITTEN attribute for the Index.

After checking TPMA_NV_WRITTEN and doing any necessary initialization, the TPM will OR the *bits* parameter to the Index.

If the TPMA_NV_ORDERLY attribute is not SET, the NV value of the Index is written with the modified value. If no bits were SET in the *bits*, the NV Index data will only be updated if TPMA_NV_WRITTEN was CLEAR when the command execution was started.

If TPMA_NV_ORDERLY is SET, the RAM version of the Bit Field data is updated but it is not written to NV. The data is only preserved to NV on a Shutdown(STATE), and on TPM Reset, the TPMA_NV_WRITTEN attribute of the Index will be CLEAR.

34.2.6.5 NV Extend Index

When an Index has the TPM_NT_EXTEND attribute, it may only be modified by TPM2_NV_Extend().

When an NV Extend Index is created, it has no value and the TPMA_NV_WRITTEN attribute will be CLEAR.

On each TPM2_NV_Extend(), the TPM will check the TPMA_NV_WRITTEN attribute of the Index. If it is CLEAR, the TPM will initialize the Index to a Zero Digest that is the size of the digest produced by the *nameAlg* of the Index. The TPM will then SET the TPMA_NV_WRITTEN attribute for the Index.

After checking TPMA_NV_WRITTEN and doing any necessary initialization, the TPM will extend the Index using:

$$nvIndex{\rightarrow}data_{\text{new}} := H_{nameAlg}(nvIndex{\rightarrow}data_{\text{old}} \parallel data.buffer) \quad (56)$$

where

$nameAlg$	is the hash algorithm indicated in *nvIndex → nameAlg*
$nvIndex{\rightarrow}data$	is the value of the data field in the Index
$data.buffer$	is the data buffer of the command parameter

If the TPMA_NV_ORDERLY attribute is not SET, the NV value of the Index is written with the modified value.

If TPMA_NV_ORDERLY is SET, the RAM version of the Index is updated but it is not written to NV. The data is only preserved on a Shutdown(STATE), and on TPM Reset, the TPMA_NV_WRITTEN attribute of the Index will be CLEAR.

Note:

To use an NV Index as a PCR, define the Index with these attributes: TPMA_NV_ORDERLY and TPMA_NV_CLEAR_STCLEAR SET and TPM_NT_EXTEND.

34.2.6.6 NV PIN Index

TPM2_NV_Write() is used to modify the contents of a PIN Index. The modification may be to the entire Index or, if the Index attributes allow (TPMA_NV_WRITE_ALL CLEAR), the size of the data to write can be as small as zero octets.

When a partial write is allowed, the *offset* parameter of TPM2_NV_Write() may be non-zero or the *size* of the data parameter may be less than the *size* of the Index. The TPM checks the TPMA_NV_WRITTEN attribute. If it is CLEAR, then the TPM will initialize the remainder of the Index to either all zero or all one. Alternatively, the TPM can initialize the entire Index at the time the Index is defined.

If the sum of the size of the *data* parameter and the *offset* parameter in TPM2_NV_Write() is greater than the size of the Index, then the TPM will not perform the write and will return an error.

On any TPM2_NV_Write() (including a size of zero), if the modification is successful, then the TPMA_NV_WRITTEN attribute of the Index will be SET. Any octets not initialized by the first write will have a value of zero.

Example:

If the Index is defined to contain 2 octets, and the first write of the Index is a single octet of 55_{16}, to offset 0, then the next read of the full Index will return 55 00_{16}.

If the Index has the TPMA_NV_ORDERLY attribute SET, then only the RAM version of the Index is written. The data is only preserved to NV on a Shutdown(STATE), and on TPM Reset, the TPMA_NV_WRITTEN attribute of the Index will be CLEAR.

If the *authValue* of an PIN Index is used for authorization, then the authorization will fail if the *pinCount* field of the Index is not less than the *pinLimit* field or if the TPMA_NV_WRITTEN attribute of the Index is CLEAR.

When the *authValue* of a PIN Index is used for authorization and the authorization succeeds, the *pinCount* field is set to zero if the Index is PIN Fail and incremented if the Index is PIN Pass. If the authorization fails, *pinCount* is incremented for a PIN Fail Index and left unchanged for a PIN Pass Index.

34.2.7 NV Index in a Policy

`TPM2_PolicyNV()` may be used to include the contents of an NV Index in a policy command. `TPM2_PolicyNV()` allows various comparisons of the value of the NV data with a reference value.

`TPM2_PolicyNV()` is an immediate assertion (see Clause 16.7.7.2). If the comparison succeeds, the TPM will update the *policyDigest* with the comparison values and the access controls on the referenced Index, including the *authPolicy*. Inclusion of the update policy of the Index provides a means of identifying the update properties of the Index. To make effective use of this command, writing of the Index should be dependent on *authPolicy.* If the policy must be met in order to write the Index, then it is possible to ensure that only the correct entity may recreate the Index. If other write authorizations are allowed, then it is not possible to know if the Index was written by a known entity.

If an NV Index is used in `TPM2_PolicyNV()` after it is defined but before it is first written, then the TPM will return an error.

The nominal use of a PIN Index is to reference the Index in an entity's policy in `TPM2_PolicySecret()`. The `TPM2_PolicySecret()` will succeed if *pinCount* is less than *pinLimit* and the caller is able to provide the *authValue* of the Index in the authorization. If the rest of the policy is satisfied, access to the PIN-protected entity will be allowed.

Note:

A PIN Fail Index provides a form of individual Dictionary Attack defense that is not affected by the TPM's global Dictionary Attack mechanism. In particular, it can be used to allow the TPM to emulate the behavior of a smart card.

Note:

A PIN Pass Index allows count-limited use of a TPM object. An example use would be to only allow access to a decryption key for protected content.

34.2.8 PIN Index Considerations

34.2.8.1 Restricting the number of uses of an object with PIN Pass

It is possible to limit the number of *authValue* (PIN) authorizations of a particular key or entity.

A key or object has a limited number of authorizations when its policy has a `TPM2_PolicySecret()` assertion pointing to a PIN Pass NV Index.

A PIN Pass's *pinLimit* is the number of correct authorization attempts that are permitted before authorization via *authValue* is locked out. If *pinCount* is less than its *pinLimit*, *pinCount* is incremented immediately by the TPM after *authValue* authorization succeeds. There is no automatic reset or decrement method for *pinCount*. Once *pinCount* equals *pinLimit*, an administrator must reduce *pinCount* and/or increase *pinLimit* using `TPM2_NV_Write()` or delete the Index.

34.2.8.2 Localized Dictionary Attack protection with PIN Fail

It is possible to authorize a particular key or object via an *authValue* (PIN) that has its own individual Dictionary Attack defense and does not use (and is not affected by) the TPM's global Dictionary Attack defense mechanism. This may be useful when a TPM is used to emulate a smartcard, for example.

A key or object has localized Dictionary Attack protection if its policy has a TPM2_PolicySecret() assertion pointing to an PIN Fail NV Index.

A PIN Fail's *pinLimit* is the number of incorrect authorization attempts that are permitted before authorization via *authValue* is locked out. If *pinCount* is less than its *pinLimit*, *pinCount* is incremented immediately by the TPM after *authValue* authorization fails. *pinCount* is reset to zero by the TPM whenever *authValue* authorization succeeds.

Note:

In the Reference Code prior to version 1.59, a successful authorization with the PIN Index *authPolicy* has the same effect on *pinCount* as a successful authorization with the PIN Index *authValue*. That means, for a PIN Pass NV Index, *pinCount* is incremented after *authPolicy* authorization succeeds, and for a PIN Fail NV Index, *pinCount* is reset to zero after *authPolicy* authorization succeeds. This behavior of the Reference Code was incorrect. Authorization with *ownerAuth*/*ownerPolicy*, *platformAuth*/*platformPolicy* is not affected by this issue, and will not increment or reset *pinCount*.

To avoid this issue on some older implementations, it is recommended that Owner or Platform authorization be used to read the PIN Index. It is recommended that the PIN Index *authPolicy* not be used to read the PIN Index unless *authValue* is part of the policy for reading the Index. The following setting is recommended for a PIN Pass or PIN Fail NV Index when the Index is defined:

- If the Owner authorized the creation of the index, TPMA_NV_OWNERREAD is SET and TPMA_NV_OWNERWRITE is CLEAR
- If the Platform authorized the creation of the index, TPMA_NV_PPREAD is SET and TPMA_NV_PPWRITE is CLEAR
- TPMA_NV_POLICYREAD is CLEAR (unless *authValue* is part of the policy for reading the index)
- TPMA_NV_POLICYWRITE is SET
- TPMA_NV_WRITEALL is SET

34.2.8.3 PIN Index Attributes

A PIN Index may be read or write locked. If read or write locked, the Index may still be referenced by TPM2_PolicySecret(). An Index disabled using *phEnableNV* (if platform created) or *shEnable* (if owner created) cannot be used in a policy. If a policy points to an unwritten PIN Pass or PIN Fail Index, the Index's authorization check must fail because *pinLimit* is not written.

Note:

Allowing a PIN Index to be used when write locked allows it to be used as a PIN but prevents writing of the *pinLimit.*

TPMA_NV_ORDERLY may be SET or CLEAR, however, if SET the Index will revert to unwritten on TPM Reset and possibly on TPM Restart (depending on TPMA_NV_CLEAR_STCLEAR). This will cause the Index to not be usable for PIN authorization until it is reinitialized.

TPM2_PolicyAuthValue() and TPM2_PolicyPassword() cannot be used in the policy that does the initial write to a PIN Index. This is because these policy commands require that the *authValue* of the PIN Index to be used and the *authValue* of a PIN Index cannot be used until it is first written. Therefore, it may be desirable that TPMA_NV_POLICYWRITE is SET so that the PIN Index value may be initialized.

If TPMA_NV_POLICYREAD, TPMA_NV_PPREAD, or TPMA_NV_OWNERREAD is SET then the Index may read using TPM2_NV_Read() (with those authorizations) without affecting the contents of the Index. If

TPMA_NV_AUTHREAD is the only method of reading the Index, then the act of reading the Index could change its *pinCount.*

Note:

Using the NV Index authorization value for the read would consume a PIN Pass Index authorization or reset the PIN Fail *pinCount.* In addition, *authValue* can't be used for authorization once *pinCount* >= *pinLimit*.

Note:

In a PIN Fail Index, it may be desirable that TPMA_NV_AUTHREAD is SET, so *pinCount* can be reset by reading the NV Index with valid *authValue* authorization.

TPMA_NV_AUTHREAD is SET, so pinCount can be reset by reading the NV Index with valid authValue authorization.

It is recommended that the Index have a policy that includes a TPM2_PolicySigned() assertion, to unambiguously identify the Index and the entity authorized to initialize the Index.

Note:

This prevents covert attacks where an Index is secretly deleted and replaced.

If the *authObject* parameter of TPM2_PolicySecret() references a PIN Pass Index, then the command may succeed, but a NULL ticket will be returned. The reason is that the ticket could allow more accesses to a count limited object than allowed by the PIN Pass Index.

Note:

Without this restriction, a caller could get a ticket for a count limited object and use the ticket instead of using the PIN Pass Index. This could, potentially, allow unlimited access to a PIN Pass entity.

If a PIN Pass or PIN Fail Index is referenced as a bind entity, the TPM must return TPM_RC_HANDLE. Otherwise, the sequence in which the TPM processes authorizations would enable a hammering attack on the Index.

Restrictions on PIN Pass and PIN Fail Indexes are specified in Part 3 TPM2_NV_DefineSpace().

34.3 Owner and Platform Evict Objects

In some applications, it is desirable for an object to be made persistent in the TPM so that it is always available. An example of when this would be useful is for a Primary Key. Having the Primary Key be always available avoids the time penalty of re-computing the Primary Key after each TPM Reset.

TPM2_EvictControl() is used to make a loaded object persistent by saving it to the TPM's NV memory. This command is also used to remove a persistent object.

To be made persistent, the object cannot be in the NULL hierarchy, the object cannot have the *stClear* attribute SET, and the object cannot be a descendant of a key with the *stClear* attribute SET.

Note:

Older versions of the specification did not allow an object to be persisted when only the public portion of the object was loaded (for NV space efficiency). Support for persisting public-only objects was added in version 185.

The type of the *objectHandle* parameter of TPM2_EvictControl() determines if the Object is to be made persistent or to be removed from persistent memory. If *objectHandle* is a Transient Object, it is made persistent and, if *objectHandle* is a persistent object, it is deleted. The Transient Object is not affected.

When making a Transient Object persistent, the *persistentHandle* parameter of TPM2_EvictControl() indicates which handle is to be assigned to the persistent version of the object. The TPM will not allow assignment of a persistent handle if that handle is already assigned to a persistent object.

If *objectHandle* is a Transient Object in the Platform Hierarchy, Platform Authorization must be provided. If *objectHandle* is in the Endorsement or Storage Hierarchy, Owner Authorization is required.

The persistent handle space is divided evenly between the Platform and the Owner. The persistent handles that may be assigned when Owner Authorization is provided are in the range 81 00 00 00_{16} to 81 7F FF FF_{16}. Handles in the range 81 80 00 00_{16} to 81 FF FF FF_{16} may be assigned when Platform Authorization is provided. When removing a persistent object, the authorization used to persist the object is required to remove it.

34.4 State Saved by TPM2_Shutdown()

34.4.1 Background

TPM2_Shutdown() is used for an orderly shutdown of the TPM. When doing an orderly shutdown, the TPM will save some state to NV memory. In the reference implantation, the state saved is separated into three groups:

1. NV Orderly Data - data that is saved on any Shutdown and is not reset,
2. NV Clear Data - data that is saved on Shutdown(STATE) and is reset on TPM Restart or TPM Reset (such as, PCR), and
3. NV Reset Data - data that is saved on Shutdown(STATE) and is reset on TPM Reset (such as session context tracking information).

34.4.2 NV Orderly Data

The data in this structure is saved to NV on any Shutdown type and restored on any Startup. It may have special initialization performed if the Startup is not orderly. In the Reference Code, this data is collected into a special data structure (ORDERLY_DATA) the contents of which are illustrated in Table 39.

Table 39: Contents of the ORDERLY_DATA Structure

Parameter	Description	Changed By:
clock	This is the version of *Clock* that is updated on any Shutdown and on any rollover of the RAM version of *Clock*.	TPM2_Clear(), TPM2_Startup(), passage of time
clockSafe	used to determine the *Safe* value reported in the TPMS_CLOCK_INFO structure. This value is CLEAR when a Startup is not orderly and once CLEAR, is not SET until the RAM value of *Clock* rolls over.	TPM2_Clear(), TPM2_Startup(), passage of time

34.4.3 NV Clear Data

Data in this structure is saved to NV on any Shutdown(STATE) but is set to its default initialization value if the subsequent Startup is either TPM Reset or TPM Restart. In the Reference Code, data of this type is collected into a single data structure (`STATE_CLEAR_DATA`) as illustrated in Table 40.

Note:

The default reset value is applied on either TPM Reset or TPM Restart. These change conditions are not listed in the “Changed By” column.

Table 40: Contents of the STATE_CLEAR_DATA Structure

Parameter	Description	Changed By
shEnable	the enable for the storage hierarchy. The default initialization value is SET.	TPM2_HierarchyControl()
ehEnable	the enable for the endorsement hierarchy. The default initialization value is SET.	TPM2_HierarchyControl()
phEnableNV	the enable for the platform hierarchy NV indices. The default initialization value is SET.	TPM2_HierarchyControl()
platformAlg	the hash algorithm used for *platformPolicy.* The default initialization value is TPM_ALG_NULL	TPM2_SetPrimaryPolicy()
platformPolicy	the policy used if the authorization session is a policy session and the authorized handle is TPM_RH_PLATFORM. The default initialization value is an Empty Buffer.	TPM2_SetPrimaryPolicy()
platformAuth	the authorization value used if the authorization handle is TPM_RH_PLATFORM and the authorization is provided by password or an HMAC session. The default initialization value is an Empty Buffer.	TPM2_HierarchyChangeAuth()

(continued on next page)

(continued from previous page)

Parameter	Description	Changed By
pcrSave	a data structure that holds the PCR that are preserved across Startup(STATE). The PCR in this structure are determined by a platform-specific TPM specification. The default initialization value for each PCR is determined by the relevant platform-specific specification but is normally a Zero Digest for each PCR in the structure.	TPM2_PCR_Extend(), TPM2_PCR_Event()
readOnly	the Read-Only mode of the TPM. The default initialization value is CLEAR.	TPM2_ReadOnlyControl()

34.4.4 NV Reset Data

Data in this structure is saved to NV on any Shutdown(STATE) and restored by a subsequent Startup of any type. In the case of a TPM Reset, the values are set to their specified initialization value. In the Reference Code, data of this type is collected into a single data structure (`STATE_RESET_DATA`) as illustrated in Table 41.

Table 41: Contents of the STATE_RESET_DATA Structure

Parameter	Description	Changed By(1)
nullProof	proof value used with entities associated with the TPM_RH_NULL hierarchy (including all session contexts, sequences, and Temporary Objects); initialization value is from the RNG	
nullSeed	seed value used for creating Temporary Objects with TPM_RH_NULL as a parent; initialization value is from the RNG	
clearCount	a value that is incremented each time the TPM performs a TPM Restart; used to tag contexts for *stClear* objects so that they may not be reloaded after a TPM Restart; initialization value is zero	TPM2_Startup(TPM_SU_CLEAR)
objectContextID	counter that is incremented each time an object is context saved; used to ensure that the encryption key and IV for each saved object is unique; initialization value is zero	TPM2_ContextSave()
contextArray	an array for keeping the version numbers of the associated saved session contexts; used to prevent replay of authorization sessions; each element is initialized to zero indicating that it is not assigned	TPM2_ContextLoad(), TPM2_ContextSave(), TPM2_StartAuthSession()

(continued on next page)

(continued from previous page)

Parameter	Description	Changed By(1)
contextCount	the value used to set the version number for each saved context; initialization value is 0.	TPM2_ContextSave(), TPM2_StartAuthSession()
commandAuditDigest	the current command code audit digest; initialization value is an Empty Digest.	Any audited command, TPM2_GetCommandAuditDigest()
restartCount	counts the number of TPM Resume, TPM Restart, or D-RTM events. Initialization value is zero.	TPM2_Startup(), _TPM_Hash_End
pcrUpdateCounter	counts the number of changes to PCR; because this value is used in policy sessions, it is not reset until the context protections for saved session contexts are changed. Initialization value is zero	TPM2_PCR_Extend(), TPM2_PCR_Event(), TPM2_PCR_Reset()
commitCounter	the number of times TPM2_Commit() is executed; initialization value is zero.	TPM2_Commit()
commitNonce	value used to create the pseudo-random values used in two-phase signing operations; initialization value is from the random number generator.	
commitArray	bit vector used to indicate that only one first phase of a two phase signing operation has occurred; initialization value is all bits CLEAR.	sign-phase of two-phase sign, TPM2_Commit()
NOTE (1) The default reset value is applied on each TPM Reset. This change condition is not listed in the “Changed By” column.		

34.5 Persistent NV Data

The data in this category is data that is always present in the TPM. This does not mean that the data cannot be changed, but that there is always a value associated with the location. The data can be changed by a Protected Capability.

In the Reference Code, the persistent NV data is in the `PERSISTENT_DATA` structure. It contents are listed in Table 42. While this table shows the context of the structure in the Reference Code, it is only illustrative. An implementation may change the contents in order to satisfy the requirements of the implementation.

Table 42: Contents of the PERSISTENT_DATA Structure

Parameter	Description	Changed By
disableClear	This value is CLEAR if `TPM_RH_OWNER` is allowed for authorization of `TPM2_Clear()`	`TPM2_ClearControl()`, `TPM2_Clear()`
ownerAlg	the hash algorithm used for the *ownerPolicy*	`TPM2_SetPrimaryPolicy()`, `TPM2_Clear()`

(continued on next page)

(continued from previous page)

Parameter	Description	Changed By
ownerPolicy	the policy used if the authorization session is a policy session and the authorized handle is TPM_RH_OWNER	TPM2_SetPrimaryPolicy(), TPM2_Clear()
endorsementAlg	the hash algorithm used for the *endorsementPolicy*	TPM2_SetPrimaryPolicy(), TPM2_Clear()
endorsementPolicy	the policy used if the authorization session is a policy session and the authorized handle is TPM_RH_ENDORSEMENT	TPM2_SetPrimaryPolicy(), TPM2_Clear()
ownerAuth	the authorization value used if the authorization handle is TPM_RH_OWNER and the authorization is provided by password or an HMAC session	TPM2_HierarchyChangeAuth(), TPM2_Clear()
endorsementAuth	the authorization value used if the authorization handle is TPM_RH_ENDORSEMENT and the authorization is provided by password or an HMAC session	TPM2_HierarchyChangeAuth(), TPM2_Clear()
lockoutAuth	the authorization value used if the authorization handle is TPM_RH_LOCKOUT and the authorization is provided by password or an HMAC session	TPM2_HierarchyChangeAuth(), TPM2_Clear()
lockoutAlg	the hash algorithm used for the *lockoutPolicy*	TPM2_SetPrimaryPolicy(), TPM2_Clear()
lockoutPolicy	the policy used if the authorization session is a policy session and the authorized handle is TPM_RH_LOCKOUT	TPM2_SetPrimaryPolicy(), TPM2_Clear()
epSeed	the seed value for the Endorsement Hierarchy	TPM2_ChangeEPS()
ehProof	the proof value for the Endorsement Hierarchy It is used to tag tickets and saved object contexts for objects in the Endorsement Hierarchy.	TPM2_ChangeEPS(), TPM2_Clear()
spSeed	the seed value for the Storage Hierarchy	TPM2_Clear()
shProof	the proof value for the Storage Hierarchy It is used to tag tickets and saved object contexts for objects in the Storage Hierarchy.	TPM2_Clear()
ppSeed	the seed value for the Platform Hierarchy	TPM2_ChangePPS()

(continued on next page)

(continued from previous page)

Parameter	Description	Changed By
phProof	the proof value for the Platform Hierarchy It is used to tag tickets and saved object contexts for objects in the Platform Hierarchy.	TPM2_ChangePPS()
resetCount	a counter that increments on each TPM Reset	TPM Reset, TPM2_Clear()
totalResetCount	a value that increments on each TPM Reset This value is used as *resetValue* in Equation 52 to tag saved contexts.	TPM Reset
pcrPolicies	This structure is used when a platform-specific specification requires that update of certain PCR requires policy authorization.	TPM2_PCR_SetAuthPolicy()
pcrAuthValues	This structure is used when a platform-specific specification requires that update of certain PCR requires HMAC or password authorization	TPM2_PCR_SetAuthValue()
pcrAllocated	This structure is used when a platform-specific specification requires support for TPM2_PCR_Allocate() to change the algorithms used for PCR and the population of the PCR in each bank.	TPM2_PCR_Allocate()
ppList	In the Reference Code, this is an array of bits that is used to indicate the commands that require assertion of Physical Presence when TPM_RH_PLATFORM is used for authorization.	TPM2_PP_Commands()
failedTries	count of the number of authorization failures for objects that are subject to Dictionary Attack protection This value can count down if no authorization failures occur for lockoutRecovery time	TPM2_DictionaryAttackLockReset(), authorization failures, passage of time (*recoveryTime*)
maxTries	the maximum value for *failedTries* before the TPM enters lockout	TPM2_DictionaryAttackParameters()
recoveryTime	the time that must pass before failedTries is decremented	TPM2_DictionaryAttackParameters()
lockoutRecovery	the time that must pass after an authorization failure using TPM_RH_LOCKOUT	TPM2_DictionaryAttackParameters()

(continued on next page)

(continued from previous page)

Parameter	Description	Changed By
lockoutAuthEnabled	when CLEAR, TPM_RH_LOCKOUT may not be used for authorization	TPM_RH_LOCKOUT auth failure, passage of time (*lockoutRecovery*)
orderlyState	between a TPM2_Shutdown() and _TPM_Init, no TPM command caused a change to the TPM's state to make the state in NV inconsistent with the state in TPM RAM	many
auditCommands	in the Reference Code, a bit array indicating which commands are audited	TPM2_SetCommandCodeAuditStatus()
auditHashAlg	the hash algorithm used for the command audit	TPM2_SetCommandCodeAuditStatus()
auditCounter	a counter that increments on the first audited command following a reset of the command audit digest The count is only incremented if the command completes with PM_RC_SUCCESS	audited command
algorithmSet	a vendor-specific value that indicates the algorithm set that is in use on the TPM This value may be used selectively to disable algorithms implemented in the TPM.	TPM2_SetAlgorithmSet()
firmwareV1	the more significant 32-bits of the vendor-assigned, firmware revision	TPM2_FieldUpgradeStart(), TPM2_FieldUpgradeData()
firmwareV2	the less significant 32-bits of the vendor-assigned, firmware revision	TPM2_FieldUpgradeStart(), TPM2_FieldUpgradeData()

34.6 NV Rate Limiting

The TPM is allowed to limit the rate at which updates are made to NV memory. An update occurs when an NV Index is defined or undefined, when an NV Index is modified, and when the persistence of an object is changed with TPM2_EvictControl(). An NV modification is allowed for other commands in an implementation dependent way. The rate for limiting the updates is TPM dependent.

When the TPM will prevent execution of a command because it is rate-limiting NV updates, the TPM will return TPM_RC_NV_RATE. This code is in the group of warning return codes meaning that the command might succeed if retried later.

Note:

Checking to see whether the NV is being rate limited is allowed to occur at any part of the command execution. This means that the TPM is permitted to return TPM_RC_NV_RATE before it has validated all of the parameters of the command. As a consequence, when the command is retried when the TPM is not rate limiting, it can fail due to incorrect parameters.

TPM2_GetCapability(*capability* == TPM_CAP_PROPERTIES, *property* == TPM_PT_NV_WRITE_RECOVERY) will provide an estimate of the number of milliseconds before the TPM will be able to accept a command that will modify the TPM NV.

Note:

After TPM2_Shutdown(), any command is allowed to cause a change of the TPM's orderly shutdown state and the TPM is permitted to return TPM_RC_NV_RATE in response to commands that are not normally allowed to make modifications to the TPM NV state.

34.7 NV Other Considerations

34.7.1 Power Interruption

A TPM is not required to maintain the integrity of the data in an NV Index if a power loss interrupts the write. After the interruption, the TPM should indicate that the Index no longer exists. The interruption of a write to one Index is not allowed to affect the integrity of other Indices.

34.7.2 External NV

34.7.2.1 Introduction

An implementation is allowed to use an external device for storing non-volatile TPM data. This may include all application defined NV (NV Indices and persistent objects) as well as all TPM state data. When stored in an external device, the data is required to be encrypted, integrity checked, and rollback protected using algorithms that have the highest security strength of any algorithm implemented on the TPM.

The encryption keys used to encrypt the data in the NV shall be protected in a manner which is defined by the TPM profile which is being implemented. The level and manner of protection for these keys shall also be specified and shall be at least as strong as the keys themselves. For a chip-based implementation, the encryption keys used to encrypt the data stored in NV are not allowed to be exposed outside of the TPM, even if encrypted.

The protection keys used to protect external NV data will be contained in or derived from a persistent value that does not leave the physical TPM. That persistent value must not be a global secret.

Note:

In many implementations, it is expected that the persistent values will be stored in fuses.

34.7.2.2 Access Interruptions

When an external device is used for non-volatile storage, that device may not always be accessible to the TPM command execution engine. When the memory is not accessible, operations that require update of NV will return TPM_RC_NV_UNAVAILABLE.

Note:

When updates to NV are being rate limited (but the NV is accessible), the TPM will return TPM_RC_NV_RATE.

During the time when NV is not available for update, *Clock* should not advance and *Safe* should be NO when accessed.

When NV is not available, the implementation may or may not advance *Clock.* If *Clock* is not being advanced, the TPM will return TPM_RC_NV_UNAVAILABLE for commands that do comparisons to *Clock* or adjustments of *Clock.* These commands are:

- TPM2_PolicySigned() or TPM2_PolicySecret() with a non-zero *expiration*;
- TPM2_PolicyTicket(); and

- TPM2_PolicyCounterTimer() if any part of TPMS_TIME_INFO.*clockInfo.clock* is used in the operation.

When NV is not available, the implementation may or may not advance *Time*. If *Time* is not being advanced, then TPM2_PolicyCounterTimer() will return TPM_RC_NV_UNAVAILABLE if any part of TPMS_TIME_INFO.time is used in the operation.

34.7.3 PCR in NV

If a TPM implementation places PCR in NV space, it should also use a caching scheme to prevent NV wear out.

35 Multi-Tasking

An implementation of the TPM may use cycles of a host processor for execution. The operating system on the host processor may not be able to operate properly if the TPM uses large blocks of time to complete execution of a command. In such systems, the TPM may be designed to yield after completion of a portion of the command so that the command may be resumed later.

When the TPM yields before completion of a command, it may return TPM_RC_YIELDED. This code indicates that the exact command that the TPM was executing may be resubmitted later. If the next command to the TPM is not the yielded command, the TPM may lose any state associated with the command that yielded so that when the yielded command is restarted, it may restart from the beginning.

36 Errors and Response Codes

36.1 Error Reporting

When a command fails, the TPM will return a 10-octet response that indicates the response code. No auxiliary information is provided by an error other than what may be inferred from the context of the error.

36.2 TPM State After an Error

When the TPM returns an error that is related to command execution, the TPM is required to preserve the TPM state. Except for the possible effect on the dictionary attack logic, it should be as though the command had not been received.

In some cases, an otherwise asynchronous operation may cause the TPM to create an error. For example, if the TPM is doing self-test of functions on an as-needed basis, the TPM may return an error due to failure of the self-test. The TPM should preserve the fact that it has failed the self-test but it should not preserve any command-specific results.

When a command modifies NV RAM, the action of writing the NV may fail and it may not be recoverable. If the TPM cannot recover from the NV write failure, then it should disable the NV so that the affected NV locations cannot be accessed.

36.3 Resource Exhaustion Warnings

36.3.1 Introduction

The executable specification has been optimized for comprehension and correctness. In particular, the Reference Code has been designed to minimize the locations in the code where resource exhaustion can occur, so that recovery from these situations is simplified. This is known not to achieve an efficient use of limited RAM resources, and other implementations may choose methods that are more aggressive in their use of memory. This clause describes the methods that are recommended for reporting of these errors.

Allocated resources are classified by their persistence relative to a command's execution. A transient resource is one that can be moved to or from TPM memory using a context management command (`TPM2_ContextLoad()`, `TPM2_ContextSave()`). These resources may continue to occupy TPM memory after completion of a command. A temporary resource is used in the processing of a command but is disposed of before the command completes. The following two clauses describe the expected behavior of the TPM when it is unable to create either of these resource types.

36.3.2 Transient Resources

The TPM Reference Code allocates space for a configuration-defined number of transient resources of the maximum size supported by the configuration parameters. This allocation occurs during the compilation process of the Reference Code. The maximum size of the objects is determined by the structure definitions in TPM 2.0 Part 2. The Reference Code presumes that, if a resource slot is available, then any object that might be stored in that slot will fit.

A practical consequence of this approach is that the only resource allocation failure for a transient resource occurs when all the dedicated slots of the appropriate type (object, sequence object, or session) are full. For objects, the number of available slots determines when the resources are all used. For sessions, there are two slot resources: handles and session contexts.

When the TPM is out of object slots, it returns `TPM_RC_OBJECT_MEMORY`. When out of session context slots, it returns `TPM_RC_SESSION_MEMORY`. When the TPM is out of handle slots for sessions, the response code is `TPM_RC_SESSION_HANDLES`.

For a system using dynamic allocation of memory for transient resources, the TPM should return an error response code that indicates the type of resource that needs to be removed from the TPM for the command

to complete. If removal of either an object or a session from TPM memory would free memory for the command, then the TPM may return TPM_RC_MEMORY. If removal of a specific resource is required, the TPM should return a code that indicates the specific resource (TPM_RC_OBJECT_MEMORY or TPM_RC_SESSION_MEMORY).

36.3.3 Temporary Resources

The TPM Reference Code is designed so that temporary resources are allocated on the execution stack. Static analysis of the code allows the maximum size of the stack to be determined so that resource exhaustion for a temporary resource cannot occur.

This construction vastly simplifies the control flow of the normative command actions, since no additional memory management code is required. However, other memory management schemes for temporary resources are allowed. Error handling for these implementations is complex and beyond the scope of this specification. However, the TPM is required to follow the standard error reporting rules.

- If the TPM returns an error, the state of the TPM is required to be restored to the state that existed before the command execution began.

  Note:

  One exception is state that would change even if a command were not executed, such as *Clock*, *Time*, dictionary attack lockout recovery, and related state. Another exception is state deliberately changed as a result of the error, such as the count of authorization failures and NV PIN Fail index values.

- The TPM will return TPM_RC_MEMORY if removal of one or more transient resources will allow the command to complete.

  Note:

  If the TPM requires the removal of a specific type of resource, then it needs to return the specific response code (TPM_RC_SESSION_MEMORY or TPM_RC_OBJECT_MEMORY) rather than the non-specific TPM_RC_MEMORY response.

- If a session must be flushed before a new session can be created, the TPM will return TPM_RC_SESSION_HANDLES.

The consequence of these requirements is that the TPM is required to be able to return the memory allocation to the same state that existed before the command execution began. It is also required that no change to NV memory be made before all temporary resources required for completion of the command have been allocated.

36.4 Response Code Details

The response code from the TPM is a 32-bit value but the TPM only uses the low-order 12 bits to communicate its warnings or errors, leaving the remaining 20 bits for use by software.

The response codes are encoded so that certain errors can be associated with the component in which the error occurred, and the specific element of the component. In cases where the error cannot be associated with a specific parameter of the command, the response code will be sufficiently differentiated to allow determination of the cause of the error.

Example:

If the second handle in the handle area was the wrong type for the command, the TPM would return TPM_RC_VALUE + TPM_RC_H + TPM_RC_2.

Example:

If the TPM can determine that the error was in the handle area but not the handle in error, the TPM would return TPM_RC_VALUE + TPM_RC_H.

The design of the response codes was constrained so that the response codes returned for commands defined in this specification would be different from the response codes defined by a previous family of the specification.

An algorithm for evaluating the response code to determine the nature of the error and the command handle, session, or parameter value in error is shown in Figure 20.

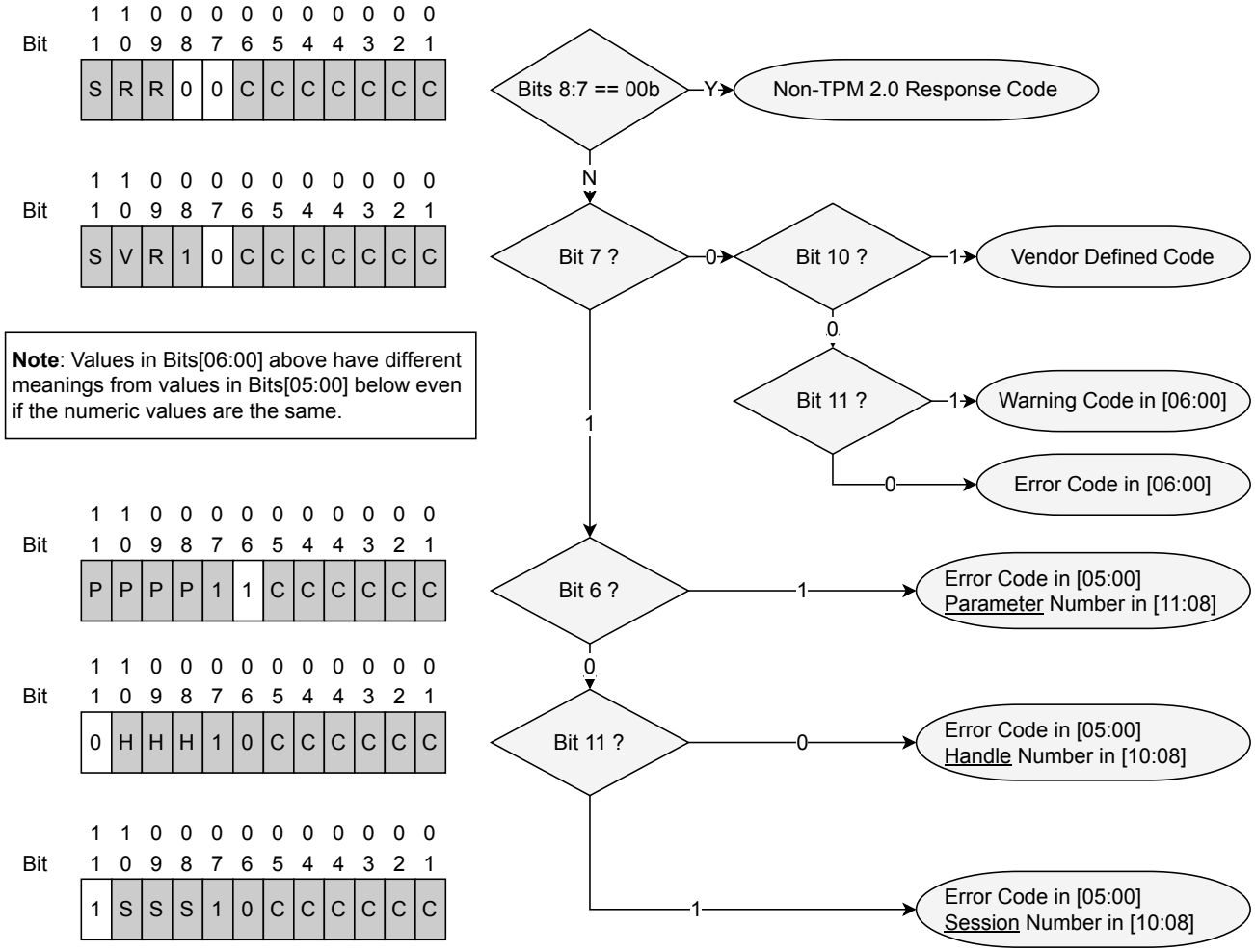


Figure 20: Response Code Evaluation

37 General Purpose I/O

A TPM may have one or more I/O pins that inputs or outputs a logic state. TPM2_NV_Read() and TPM2_NV_Write() may be used to access the value of GPIO using normal access controls.

A platform-specific specification defines the mapping of NV Indices to individual General Purpose I/O (GPIO). Whether the TPM reserves any NV storage for the indicated GPIO is platform specific.

This specification does not require the NV Indices associated with GPIO pins to be pre-allocated. When one of the Indices reserved for GPIO pins is defined, it is automatically associated with the corresponding GPIO pin.

Note:

The owner and platform space are segregated and it is expected that the GPIO pins will be assigned to Index values in the Index space reserved for the platform.

Note:

The TCG maintains a registry of reserved NV Index values.

The controls that let the GPIO pin be used either as an input or an output are vendor or platform specific.

For outputs, if the Index has the TPMA_NV_ORDERLY attribute SET, the output state is volatile, and becomes non-volatile on an orderly shutdown. If the TPMA_NV_ORDERLY attribute is CLEAR, the output state is non-volatile.

For inputs, a read of the Index returns TPMI_YES_NO, where YES indicates a logic 1 and NO indicates a logic 0 on the input pin.

38 Minimums

38.1 Introduction

This clause lists the minimums for specific functional blocks where a minimum is needed to ensure proper TPM operation.

Platform-specific TPM specifications may impose other minimums but those minimums are not allowed to be less than the minimums in this specification.

38.2 Authorization Sessions

An active authorization session is a session that is currently loaded into TPM memory and can be addressed with a session handle in a command. A concurrent session is an authorization session that either is loaded on the TPM or has its context saved.

A command may require no more than three sessions divided according to the needs of the command. The TPM is required to be able to support execution of a command with three authorization sessions.

The management of sessions is different from the management of objects. Management software can keep the contexts for an indefinite number of objects and load them as required. The number of concurrent sessions, however, is limited by the resources that the TPM can devote to tracking those sessions.

The TPM should support a minimum of 64 concurrent sessions. Fewer sessions would impair the ability of the TPM to conduct concurrent operations with multiple users.

38.3 Transient Objects

In order to be able to execute all commands, the TPM needs to have two active, loaded objects of any type. A Transient Object is an object that occupies TPM memory and may be referenced by handle. The number of Transient Objects that the TPM supports does not include those objects that have been placed in persistent TPM memory.

Note:

A TPM implementation is permitted to copy an object from persistent storage into a Transient Object slot in order to speed up access to the object data.

38.4 NV Counters and Bit Fields

All TPM implementations should allow at least one NV Index to be allocated for use as a monotonic counter (`TPMA_NV_COUNTER`) or bit field (`TPMA_NV_BITS`). The number of these Index types determines how many different policies may include revocation as part of their logic. When the number of these Index types is too small, the software complexity of handling revocation becomes too complex to manage.

Note:

This minimum (1) may be adequate for a TPM in a simple embedded system but is too low for a TPM in a complex system such as a PC. It is recommended that platform-specific specifications for more complex systems mandate support for at least sixteen (16) counter or bit field Indices.

Note:

The requirement that a TPM support the `TPMA_NV_COUNTER` or `TPMA_NV_BITS` attribute implies that the TPM is required to implement either `TPM2_NV_Increment()` or `TPM2_NV_SetBits()`.

39 Attached Components

39.1 Introduction

Deprecated:

Attached Components (`TPM2_AC_GetCapability()`, `TPM2_AC_Send()`, and `TPM2_Policy_AC_SendSelect()`) were deprecated in version 184. See Part 0.

39.1.1 Purpose

The TPM has an extensive set of commands that allow implementation of flexible access control for objects contained in the Shielded Locations of the TPM. The pair of commands `TPM2_AC_Send()` and `TPM2_Policy_AC_SendSelect()` enable the access control mechanisms of a TPM to be used to manage sensitive data (such as keys) for other components (called the Attached Components) that are attached to the TPM.

Examples of Attached Components are encrypting disk controllers, crypto accelerators, and network adapters with crypto capabilities.

39.1.2 Concept

An Attached Component (AC) is physically connected to a TPM through a data channel other than the TPM's command and response buffer (the details of the TPM-to-AC data channel are beyond the scope of this document). A properly authorized `TPM2_AC_Send()` will cause the TPM to copy the selected TPM Object to a selected AC over the provided data channel. This avoids having to "bounce" the Object via an out of band transfer from the TPM through memory then to the component. Object data is, therefore, duplicated outside the TPM's "Shielded Location". The platform manufacturer provisions the connection between the TPM and the Attached Component, and hence determines the protection of an object in transit from the TPM to the Attached Component.

`TPM2_AC_Send()` uses the DUP auth role that requires authorization with a policy with *policySession→commandCode* set to `TPM_CC_AC_Send`.

Note:

`TPM2_AC_Send()` will only send objects that are constructed with a policy that allows `TPM2_AC_Send()`.

39.2 TPM2_AC_Send()

`TPM2_AC_Send()` instructs the TPM to copy portions (possibly all) of the Object referenced by *sendObject* to the Attached Component referenced by *ac*. The methods used to send the Object and the properties of the Attached Component are outside the scope of this specification. The command does not cause the referenced object to be flushed.

The applications using this TPM and the creators of the TPM's objects are expected to understand the properties of the Attached Component and the connection between the TPM and the Attached Component.

The use of Enhanced Authorization allows the object creator to restrict sending the object using any combination of Policies. For example, the policy may only allow an Object to be sent when PCR and Locality have specific values. `TPM2_AC_Send()` requires DUP role authorization for the *sendObject.* This means that *sendObject* authorization requires a policy session that has *policySession→commandCode* set to `TPM_CC_AC_Send`. This requirement ensures that `TPM2_AC_Send()` is only used on Objects that are intended to be sent to an Attached Component.

The *acDataIn* parameter allows `TPM2_AC_Send()` to send qualifying data to an Attached Component along with the Object. For example, *acDataIn* may be used to identify a key slot in the AC into which the Object is to be loaded.

An Attached Component optionally returns *acDataOut* information to the caller. A possible use of *acDataOut* would be for the AC to indicate the key slot into which it has placed the Object.

After TPM2_AC_Send() completes, the Object in the TPM may be flushed without effecting the values in the AC. Similarly, the AC may modify (or delete) the Object data from its memory without affecting any Object in TPM Shielded Locations.

39.3 Send Object Types

The TPM does not restrict the type of Object that may be sent to an AC. However, an AC may not be able to process all types of TPM Objects. The AC may be designed to reject unknown Object types or it may be the responsibility of system software to ensure that only the proper Object types be sent to an AC.

39.4 Send Object Attributes

The *sendObject* shall have *fixedTPM*, *fixedParent* and *encryptedDuplication* CLEAR.

39.5 Attached Component Authorization

TPM2_AC_Send() requires authorization for the Object to be sent and for the AC that is the target of the send operation.

Example:

If an Attached Component is an encrypting disk controller, unauthorized software, such as ransomware, cannot set the encryption key without getting access to the proper authorization. Use of the PCR as part of the policy for the object can ensure that the disk encryption key can only be set early in the boot phase.

Proper authorization to send to an AC is determined by the presence of an aliased NV Index. The range of AC handles is AC_FIRST to AC_LAST. The range for aliased AC Indexes is NV_AC_FIRST to NV_AC_LAST. If the Owner or Platform creates an NV Index in the range AC Indexes and a corresponding AC exists in the AC handle range, then the write authorization settings of the AC Index (TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE, and TPMA_NV_POLICYWRITE) determine how the send may be authorized. If no aliased AC Index exists, then either Owner Authorization or Platform Authorization may be used to allow the send to the AC.

The Platform and Owner may use TPM2_NV_DefineSpace() to delegate the rights to access a specific AC The Platform or Owner may change the access rights to the AC by calling TPM2_NV_UndefineSpace() and then TPM2_NV_DefineSpace() with new parameters.

TPM2_NV_ChangeAuth() allows the current key and object manager to change the *authValue* by using the current AC authorization policy.

The following is a summary of states and allowed actions:

1. Attached Component has no aliased NV Index defined
    1. Can use TPM2_AC_Send() to send objects to the Attached Component with *ownerAuth* or *platformAuth*
    2. Can use TPM2_AC_GetCapability() to get optional information about the AC
2. Attached Component has an aliased NV Index
    1. Can use TPM2_AC_Send() to send objects to the Attached Component with the write authorization types allowed by the aliased Index.
    2. Can change authorization using ADMIN role (TPM2_NV_ChangeAuth()).

TPM2_Clear() will delete any AC Index alias defined by the Owner but not by the Platform.

39.6 Attached Component Object Management

39.6.1 Discovery

Applications may use information from the platform manufacturer (for example, a platform certificate) to determine whether a platform has an Attached Component. Software may also discover the number of attached components by calling TPM2_GetCapability(*capability* == TPM_CAP_HANDLES) with a TPM_HT_AC Property Type to find available AC handles.

Specific information about each Attached Component may be provided by calling the TPM2_AC_GetCapability() command. In response to this command, the TPM returns vendor-specific information about a specific Attached Component. For example, *capabilitiesData* may be used to return a pairing value that can be also obtained from the AC itself by an AC-specific API (see Part 2, *Attached Component Structures*).

The association between ACs and AC handle is vendor specific, but the association is required to be constant during the platform's lifetime. If an AC is replaced, the new AC should have the same AC handle as the AC that was replaced.

After reboot or other TPM power state changes, TPM2_GetCapability() should be used to obtain the list of the available ACs.

Note:

It is suggested that no handle be associated with a defective AC. However, it is also allowed that a defective AC be reported using TPM2_AC_GetCapability().

TPM2_GetCapability() can be used at any time to determine the current AC configuration.

Note:

There is no mechanism for the TPM to proactively signal the change in state of an AC. However, a platform can have a method for the AC to signal software and for that indication to trigger an enumeration process.

39.6.2 Setup

The AC's configuration should be performed by the AC's Manufacturer. AC configuration is out of scope of the TCG.

39.6.3 Sending

If the desired Object is not already loaded, the application (such as a key manager) may load it using an existing TPM command such as TPM2_Load(). This removes the wrapping of the object and returns an object handle. However, if an object is evicted and then re-loaded, there is no assurance the TPM will assign the object the same object handle. For this reason, applications should not associate the TPM's assigned object handle with the object when sent to the Attached Component, or when the object is within the Attached Component's domain. Key and object managers using the TPM2_AC_Send() command may assign, if necessary, a handle for the transferred object within the Attached Component domain using the *acDataIn* parameter. Therefore, in common usages there is no association between the TPM generated object handle assigned to the object and any handle for that object within the Attached Component's domain. Any association between the TPM's object and the Attached Component's object is done by the application or the application's resource and object manager external to the TPM.

39.7 Power States

The AC authorization values and policies that are setup by TPM2_NV_DefineSpace() are persistent across all TPM power state changes, like the authorization values and policies of other NV Indexes.

TPM power state changes have no effect on the Attached Component's copy of objects, because that copy of the object is no longer in the TPM's "Shielded Location".

39.8 Attached Component Format

The format and security properties of the connection between the TPM and the Attached Component are outside the scope of the TPM Library Specification. A vendor may provide a proprietary connection between the TPM and an Attached Component where the format and the security properties are defined by that vendor. A TCG platform-specific Working Group may define a data format that may include a key exchange method so that independently manufactured Attached Components can interoperate with different TPMs.

40 Authenticated Countdown Timer (ACT)

40.1 Introduction

The functionality and commands described in this clause enable the TPM to manage multiple authenticated countdown timers (ACT).

40.2 Description

An ACT is a 32-bit counter that, when not already zero, will decrement by one each second that the TPM is powered.

The countdown timers are used to trigger events on a platform when they count down to zero, at which point they are said to timeout or expire. `TPM2_ACT_SetTimeout()` is used to set an ACT to a non-zero value and begin the timeout. On TPM Reset or TPM Restart, all ACT timeouts are set to zero with no side effects (no event triggered). ACT timeouts are preserved across TPM Resume.

The ACT timeouts are saved by `TPM2_Shutdown(TPM_SU_STATE)`. On `TPM2_Startup(TPM_SU_STATE)`, if the TPM shutdown was orderly, the saved ACT values are restored and the ACT resumes counting. If an ACT *startTimeout* has been written (`TPM2_ACT_SetTimeout()`) since the last `TPM2_Startup()`, then the current timeout of the ACT is saved by `TPM2_Shutdown(TPM_SU_STATE)`; otherwise, the saved value is one half of the current ACT timeout. If a `TPM2_ACT_SetTimeout()` occurs after the `TPM2_Shutdown()`, then the TPM state is no longer orderly, and a subsequent `TPM2_Startup(TPM_SU_STATE)` will fail.

Note:

A system could continue to operate after a `TPM2_Shutdown(TPM_SU_STATE)`. Therefore, saving half the timeout prevents an attacker from continually extending the timeout by doing `TPM2_Shutdown(TPM_SU_STATE)` immediately after `TPM2_Startup(TPM_SU_STATE)`, and then restarting the system (TPM Resume) just before the timer expires.

An ACT has an *authValue* and an *authPolicy*. The *authValue* is the same as the current *platformAuth* and can only be used if *phEnable* is SET. The *authPolicy* is ACT-specific and is neither enabled nor disabled by *phEnable*. Each ACT has its own *authPolicy*.

`TPM2_ACT_SetTimeout()` must be properly authorized. Authorization may be provided either by *platformAuth* or by an ACT-specific *authPolicy*. The *startTimeout* parameter in `TPM2_ACT_SetTimeout()` is an integer number of seconds.

The *authPolicy* for an ACT can be changed by `TPM2_SetPrimaryPolicy()` using either *platformAuth* or the ACT-specific *authPolicy*.

Example:

The platform can delegate use of an ACT to the owner by setting its ACT policy to `TPM2_PolicySecret()` referencing *ownerAuth*.

The *authPolicy* of an ACT is initialized to an Empty Policy by TPM Reset or TPM Restart but is preserved during TPM Resume.

Note:

After TPM Reset or TPM Restart, *phEnable* is SET, allowing the platform to initialize any ACT *authPolicy.*

40.3 Typical Use

A typical example for the use of an ACT is as a watchdog timer that will cause a platform reset when the timer reaches zero (expires). In a system using an ACT, a periodic platform action outside the TPM indicates that

the timeout should be set anew using TPM2_ACT_SetTimeout(). The most common reason why timeout is not set anew is that the local system is not behaving properly because of some type of corruption (either inadvertent or malicious). The intent of the timer is that, in the absence of a properly authorized timeout extension, the platform would be reset, putting it back into a known state with the expectation that the corruption can be removed. The reason for having an authenticated timeout is to allow an external entity to make a decision about the health of the system.

The example above is not the only one supported by an ACT. In fact, this specification does not mandate that any specific platform behavior occur as a result of a timer expiring. The action on timer expiration may be chosen by a platform-specific specification or be vendor specific.

Because *platformAuth* may be used to change the *authPolicy* or set a *startTimeout* for any ACT, the platform firmware has ultimate control of the ACT. On each TPM Reset or TPM Restart, the platform firmware is expected to set the *authPolicy* for all ACT using *platformAuth*. This specification mandates no specific policy for any ACT, but it is expected that, in most cases, the platform firmware will either:

1. Initialize an ACT *authPolicy* with a policy that can only be satisfied by an entity trusted by the platform manufacturer; or
2. Initialize the ACT *authPolicy* so that the *authPolicy* can be changed using *ownerAuth.*

In the first case, the platform firmware may set an initial timeout to ensure that some corrective action will occur if malware prevents the trusted entity from setting the ACT.

Note:

This is how the platform would typically initialize a watchdog timer

In the second case, the system software is expected to take control of the ACT. The platform would not set an initial timeout as it is possible that the ACT will not be used by the system software.

Note:

If the platform firmware does not initialize the ACT *authPolicy* before *phEnable* CLEAR, then the ACT cannot be used.

40.4 Failure Mode

If the TPM enters failure mode, the ACT should continue to count down and trigger the specified event should it expire.

Note:

If the failure mode was caused by a timer failure or affects functionality that is required for the platform-specific event, the ACT might not trigger reliably.

TPM2_ACT_SetTimeout() shall not be usable while a TPM is in Failure Mode. This means that the timeout cannot be extended and that timed events will occur if the TPM is not powered down or Reset before the ACT expires. A platform-specific specification may specify that an event will have no effect if the TPM is in Failure Mode.

A TPM may allow reading of the remaining ACT time (TPM2_GetCapability(*capability* == TPM_CAP_ACT)) when the TPM is in Failure Mode.

40.5 Field Upgrade

The behaviour of a TPM during Field Upgrade is undefined. However, it is preferred that ACT continue to operate normally during Field Upgrade except that the ACT may not be changed by TPM2_ACT_SetTimeout().

Note:

Since *platformAuth* is required to start a Field Upgrade, *platformAuth* can be used to set the *startTimeout* for any active ACT to a value that is sufficient to allow a Field Upgrade to complete.

40.6 Typical ACT authPolicy

This clause describes a typical ACT authorization policy that authorizes setting of *startTimeout* with an authentication credential (key). The signature created by the authentication key is used as a cryptographically protected deferral ticket for the ACT.

The ACT *authPolicy* is constructed using TPM2_PolicySigned() and may include other policy components. Authorization by multiple entities can be achieved by combining multiple TPM2_PolicySigned() commands using AND or OR terms.

The deferral ticket is provided to the TPM in TPM2_PolicySigned() as the *auth* parameter (the signed authorization). The signature verification key, the *authObject* in TPM2_PolicySigned(), may be a symmetric or asymmetric key.

Note:

The advantage of an asymmetric signing key is that only the public key needs to be provisioned into the TPM. In the case of a symmetric HMAC key, it is recommended that the HMAC key's *authPolicy* restricts the key to be used only for TPM2_PolicySigned() and not for other commands like TPM2_HMAC(), so that the TPM cannot to issue its own deferral tickets.

For the *nonceTPM* and *expiration* parameters of TPM2_PolicySigned(), the following is recommended:

1. *nonceTPM* present and not an Empty Buffer
2. *expiration* > 0

Both settings ensure that a deferral ticket is single-use. The presence of *nonceTPM* in TPM2_PolicySigned() prevents the same signature being used multiple times within a policy session to defer the ACT indefinitely.

Note:

The *nonceTPM* for the policy session changes at the end of TPM2_ACT_SetTimeout(). This invalidates the previous signature and prevents replay of TPM2_ACT_SetTimeout() without getting a new signature from the authorized entity.

The non-negative *expiration* prevents TPM2_PolicySigned() creating a policy ticket which may be reused with TPM2_PolicyTicket() over a period of time to defer the ACT.

The ability to change *startTimeout* of TPM2_ACT_SetTimeout() should be limited by including *cpHash* in the ACT *authPolicy*. This can be achieved in two ways:

1. The ACT authorization policy includes TPM2_PolicyCpHash(). In this case, the entity setting the policy determines the *startTimeout* value.
2. The ACT authorization policy includes TPM2_PolicySigned() with *cpHashA* set. In this case, the signer (of the deferral tickets) determines the startTimeout value.

41 TPM Firmware-Limited and SVN-Limited Objects

41.1 Introduction

A TPM can be used to attest to the firmware and software running on a device. For example, a platform-specific specification can describe how each platform boot sequence stage is logged into TPM PCRs. The values of TPM PCRs (and therefore the associated firmware/software stack those values represent) can be attested by quoting them with an Attestation Key, or by binding other objects to PCR values.

In some use cases, a relying party might benefit from attestation of the firmware that the TPM itself is currently running. This capability relies upon firmware-attestation secrets and can be of interest after a flaw in a TPM implementation has been identified and replacement firmware has been installed on the TPM.

Note:

This capability requires a relying party to assess the functionality of different versions of TPM firmware and decide whether firmware-attestation secrets have been disclosed to adversaries. A TPM with fully secure TPM firmware will not disclose firmware-attestation secrets via the TPM's programmatic interface. A TPM with less-than-fully-secure TPM firmware might or might not be subverted, and disclose firmware-attestation secrets, depending upon the functionality of that TPM firmware. Even if firmware-attestation secrets could be disclosed, specific firmware-attestation secrets might or might not have been disclosed to adversaries.

Internal implementation details of TPMs vary. Figure 21 depicts a simple TPM that is composed of a bootloader and an application layer implemented in firmware. The bootloader is responsible for booting the TPM, and the firmware responds to TPM 2.0 commands from the host system via the TPM bus.

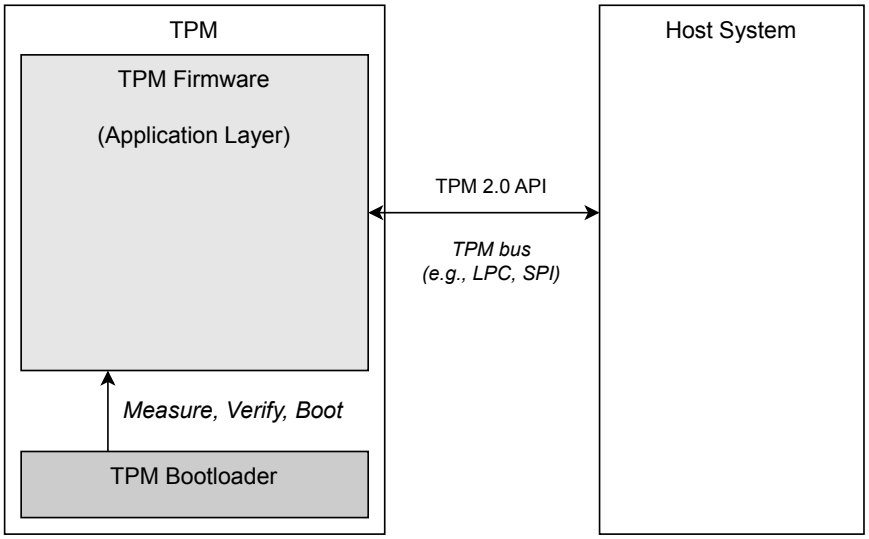


Figure 21: Example TPM Bootloader and Firmware

41.2 Security Version Numbers

A Security Version Number (SVN) denotes the security level of a given version of TPM firmware. When a security vulnerability is identified, TPM vendors should increment the SVN associated with the firmware that carries the correction.

After the correction is applied to a given TPM instance, the new firmware will be able to access SVN-limited objects associated with old SVN values. On the other hand, if TPM firmware is downgraded to one with a lower SVN, the older firmware will not be able to access SVN-limited objects associated with newer SVN values.

A given TPM firmware image's SVN can be a function of its *firmwareVersion,* or it can be managed by the vendor independently. The TPM reports its current image's SVN via `TPM2_GetCapability()`.

A TPM's SVN is architecturally defined as a UINT16. This provides a sufficient range of SVN values over a product's lifetime, while allowing the SVN to be specified in the 32-bit reserved handle range.

41.3 Description of Firmware-limited Objects

An object with the *firmwareLimited* attribute SET is cryptographically limited to the current firmware of the TPM. If the object is a Primary Object, it is partly derived with secret entropy limited to the current firmware. If it is not a Primary Object, then it is part of a hierarchy rooted with a firmware-limited Primary Key (so it is limited to the current firmware). This secret entropy is termed the Firmware Secret and is cryptographically limited to the current firmware image running on the current TPM instance. TPM firmware cannot derive a Firmware Secret bound to any other TPM firmware image or belonging to any other TPM instance. This property must be implemented in a manner that does not rely on TPM firmware to be well-behaved. The mechanism by which this property is enforced is outside the scope of this specification.

Example:

The TPM's bootloader knows *bootloaderSecret*, and derives:

$$fwSecret := \mathbf{KDFa}(hashAlg,\ bootloaderSecret,\ \text{"FW\_SECRET"},\ \mathbf{H}(tpmFw.image),\ \text{NULL},\ bits)$$

The TPM's bootloader must ensure that the Firmware Secret, as well as objects derived from or protected by it, are not available to other firmware images beyond the firmware to which the Firmware Secret is limited.

41.4 Description of SVN-limited objects

An object with the *svnLimited* attribute SET is cryptographically limited to a given SVN of TPM firmware. If the object is a Primary Object, it is partly derived with secret entropy limited to that SVN. If it is not a Primary Object, then it is part of a hierarchy rooted with an SVN-limited Primary Key (so it is cryptographically limited to the given firmware SVN). This secret entropy is termed a Firmware SVN Secret and is cryptographically limited to a given firmware SVN running on the current TPM instance.

TPM firmware with SVN X can derive every SVN Secret bound to an SVN <= X, and cannot be able to derive an SVN Secret for any SVN > X. This monotonic access property must be implemented in a manner that does not rely on TPM firmware to be well-behaved. The mechanism by which this monotonic access property is enforced is outside the scope of this specification.

Example:

The TPM's bootloader knows *bootloaderSecret*, and derives:

$$svnSecret := \mathbf{KDFa}(hashAlg, bootloaderSecret, \text{"SVN\_SECRET"}, \text{NULL}, \text{NULL}, bits)$$

$$maxSVN := 255$$

Code:

```
for (i = maxSVN; i >= tpmFw.svn; i--) { svnSecret = H(svnSecret) }
```

If TPM firmware has SVN 4 and wishes to derive the svnSecret for SVN 2, it can hash its svnSecret twice. The firmware cannot obtain the svnSecret for SVN > 4 because it cannot efficiently reverse the hash operations performed by the bootloader.

Example:

The TPM's bootloader knows *bootloaderSecret*, and the firmware's SVN, and writes them to hardware registers:

$$svnSecretSeed := \mathbf{KDFa}(hashAlg, bootloaderSecret, \text{"SVN\_SECRET"}, \text{NULL}, \text{NULL}, bits)$$

Code:

```
hw.writeRegister(SVN_SECRET_SEED, svnSecretSeed);

hw.writeRegister(LATCHED_FW_SVN, tpmFw.svn);
```

At runtime, when the user wishes to create an SVN-limited object based on a requested SVN, TPM firmware queries the hardware:

Code:

```
hw.deriveVersionedSecret(requestedSvn);
```

Hardware will reject the request if *requestedSvn* is > `LATCHED_FW_SVN`. Otherwise, it will derive and return **KDF**(*hashAlg*, *SVN_SECRET_SEED*, *requestedSvn*).

Note:

A user that wishes to create an SVN-limited object that is available to all firmware SVNs should select SVN 0.

Note:

If the firmware's current SVN equals its maximum SVN, any SVN-limited key created by the current firmware will be available to all future firmware updates.

Note:

Multiple distinct firmware images can share the same SVN. Such images will share access to the same Firmware SVN Secrets, but not the same Firmware Secret.

The TPM's bootloader must ensure that Firmware SVN Secrets, as well as objects derived from or protected by them, are not available to other firmware images with SVNs lower than that to which each given Firmware SVN Secret is bound.

41.5 EvictControl of objects in firmware-limited and SVN-limited hierarchies

Objects in a firmware-limited or SVN-limited hierarchy (including objects without *firmwareLimited/svnLimited* SET), cannot be made persistent by `TPM2_EvictControl()`.

Note:

An object can exist within a firmware-limited (or SVN-limited) hierarchy and not have *firmwareLimited* (or *svnLimited*) SET. In this case, the object is protected by the properties of its object hierarchy, the root of which is a *firmwareLimited* (or *svnLimited*) hierarchy handle.

Regardless of the setting of *firmwareLimited* (or *svnLimited*), persisting a firmware-limited (or SVN-limited) object would remove the protection of its (firmware- or SVN-limited) object hierarchy in the case of a TPM firmware update.

41.6 Firmware-limited and SVN-limited hierarchy seeds and proof values

Firmware-limited and SVN-limited hierarchies are referred to by reserved permanent handles. Each such hierarchy has an associated base hierarchy (which specifies a primary seed and proof value) and additional secret entropy. Firmware-limited and SVN-limited hierarchy primary seeds and proof values are derived from the base hierarchy's primary seed / proof value, as well as the additional secret.

Table 43: Additional Secrets for Firmware- and SVN-limited Hierarchy Proof Values

Limited hierarchy	Base hierarchy	Additional secret
TPM_RH_FW_OWNER	TPM_RH_OWNER	Firmware secret
TPM_RH_FW_ENDORSEMENT	TPM_RH_ENDORSEMENT	
TPM_RH_FW_PLATFORM	TPM_RH_PLATFORM	
TPM_RH_FW_NULL	TPM_RH_NULL	
TPM_RH_SVN_OWNER_BASE + {SVN}	TPM_RH_OWNER	Firmware SVN secret associated with {SVN}
TPM_RH_SVN_ENDORSEMENT_BASE + {SVN}	TPM_RH_ENDORSEMENT	
TPM_RH_SVN_PLATFORM_BASE + {SVN}	TPM_RH_PLATFORM	
TPM_RH_SVN_NULL_BASE + {SVN}	TPM_RH_NULL	

The seed and proof value for each limited hierarchy can be derived as follows:

$$value := \mathbf{KDFa}(hashAlg, bSecret, bSecretLabel, aSecret, aSecretLabel, bits) \quad (57)$$

where

hashAlg	is a hash algorithm chosen by the vendor
bSecret	is a base secret (seed or proof) associated with the base hierarchy
bSecretLabel	is a value used to differentiate between seed or proof derivation
aSecret	is an additional secret (Firmware Secret or Firmware SVN Secret) associated with the hierarchy
aSecretLabel	is a value used to differentiate between firmware-limited or SVN-limited hierarchies
bits	is the number of bits returned is the size of seed or proof values

Note:

If Equation 57 is used, the value of *bSecretLabel* is required to be different from any other label string used in a **KDFa**() call. The Reference Code uses “H_SEED_SECRET” and “H_PROOF_SECRET” for deriving seeds and proofs, respectively. For aSecretLabel, the Reference Code uses “H_FW_SECRET” and “H_SVN_SECRET” for deriving seeds/proofs for firmware-limited hierarchies and SVN-limited hierarchies, respectively.

41.7 Firmware-limited vs SVN-limited objects

A firmware-limited object is cryptographically limited to the current firmware image; if the image is upgraded or downgraded, all firmware-objects that were limited to the previously installed firmware image are lost. By contrast, SVN-limited objects remain available to TPM firmware updates as long as those updates’ SVNs are greater than or equal to the SVN to which the object is limited.

Both mechanisms enable TPM users to recover trust in TPMs, should there exist a firmware image with a critical vulnerability that leaks the TPM’s key material, including all its EKs. After updating firmware to patch the vulnerability, these mechanisms allow users to create keys that cannot be compromised by the older vulnerable firmware.

Firmware-limited objects are also useful for reestablishing trust in the face of a compromise of the TPM vendor’s field-upgrade firmware signing key. A compromised field-upgrade firmware signing key could sign malicious TPM firmware with an arbitrarily high SVN, thus gaining access to any SVN-limited object. However, such malicious firmware could not obtain access to firmware-limited objects created by legitimate firmware, since the objects are bound to the firmware image itself, rather than the firmware’s associated SVN.

In addition, firmware-limited objects are useful for TPM implementations that support only explicit attestation, and do not support sealing. These implementations do not require the ability to preserve sealed secrets across firmware updates. For such implementations, derivation of the Firmware SVN Secret is unnecessary overhead.

SVN-limited objects are useful for objects that should remain available across TPM firmware updates, such as keys that encrypt long-lived secrets. Such objects remain available to new TPM firmware as long as that new firmware’s SVN is greater than or equal to the SVN to which the object is limited.

41.8 Firmware-Limited Endorsement Keys

An Endorsement Key with the *firmwareLimited* attribute SET is cryptographically limited to the current version of the TPM firmware.

To do this, the user uses an EK template to create a Primary Key with *firmwareLimited* SET and sets the *primaryHandle* to TPM_RH_FW_ENDORSEMENT. Its Name will reflect the fact that *firmwareLimited* was SET, so it can be attested (using TPM2_Certify(), TPM2_CertifyCreation(), TPM2_ActivateCredential()) like any other TPM object.

Note:

A flaw in some version of TPM firmware might allow an adversary to use a conventional (non-firmware-limited) Endorsement Key to impersonate any version of TPM firmware. If a conventional Endorsement Key is used to activate a firmware-limited Endorsement Key, the user may be able to prevent impersonation by activating a firmware-limited Endorsement Key in a controlled environment.

Note:

In some circumstances, an entity such as a TPM vendor might be able to provide an EK certificate for a firmware-limited Endorsement Key. Such methods are beyond the scope of this specification.

41.9 Firmware-Limited Attestation Keys

An Attestation Key with the *firmwareLimited* attribute SET is cryptographically limited to the current version of the TPM firmware.

To use this, the user creates an Attestation Key with *firmwareLimited* SET and sets the *primaryHandle* to TPM_RH_FW_OWNER, TPM_RH_FW_PLATFORM, TPM_RH_FW_ENDORSEMENT, TPM_RH_FW_NULL, or a parent that has the *firmwareLimited* attribute. An attestation CA checks that the *firmwareLimited* attribute is SET in the key's public area and associates the key with some TPM firmware version (e.g., the value from an EK certificate, or the unobfuscated *firmwareVersion* value from an attestation structure signed by the key if it is in the Endorsement hierarchy).

Note:

A change in TPM firmware implies a change to *firmwareVersion*, which is the 64-bit value signed in TPMS_ATTEST structures. *firmwareVersion* is also reported by TPM2_GetCapability() as the TPM_PT properties TPM_PT_FIRMWARE_VERSION_1 and TPM_PT_FIRMWARE_VERSION_2.

Note:

When *firmwareVersion* is unchanged, firmware-limited objects are also unchanged. This includes situations where the TPM may run firmware version A, then firmware version B, then firmware version A again. Firmware-limited objects created on a given TPM running firmware version A are always available to that TPM while it is running firmware version A.

A firmware-limited Attestation Key's Name will reflect the fact that *firmwareLimited* was SET, so it can be attested (using TPM2_Certify(), TPM2_CertifyCreation(), TPM2_ActivateCredential()) like any other TPM object.

Note:

A flaw in some version of TPM firmware might allow an adversary to use a conventional (i.e., non-firmware-limited) Endorsement Key to impersonate any version of TPM firmware. If a conventional Endorsement Key is used to activate a firmware-limited Attestation Key, the user may be able to mitigate the risk of impersonation by activating a credential for a firmware-limited Attestation Key in a controlled environment.

41.10 Enrollment of SVN-Limited Objects

An SVN-limited object is cryptographically limited to a given minimum SVN of TPM firmware.

To do this, the user creates an object hierarchy rooted by a Storage Primary Key created using `TPM2_CreatePrimary()` or `TPM2_CreateLoaded()`, setting *primaryHandle* to the SVN-limited primary handle value:

`TPM_RH_SVN_{ENDORSEMENT`, or `OWNER`, or `PLATFORM`, or `NULL}_BASE + SVN`,

where SVN is a UINT16 value denoting the minimum Security Version Number to which the hierarchy is limited. *svnLimited* objects in this hierarchy are limited to the given minimum SVN. Creation or loading of such objects shall fail if the given SVN is greater than the TPM firmware's current SVN.

svnLimited indicates that the object is confined to an SVN-limited hierarchy but does not indicate the actual base hierarchy or SVN value to which the object is limited. If a remote registrar (e.g., a Privacy CA) wishes to confirm the actual minimum SVN value to which the object is limited, the registrar needs to compute the expected Qualified Name of the SVN-limited object in its hierarchy, rooting all the way back to the SVN-limited Primary Handle, and compare it with the attested qualifiedName of the object as reported by either `TPM2_Certify()` or `TPM2_CertifyCreation()`.

Note:

This would require transmitting the public areas of all the ancestor keys in the object's hierarchy to the CA. A Primary Key has no ancestor keys (its parent's Qualified Name is the hierarchy handle that was used in CreatePrimary), which makes it straightforward to verify a Primary Key's Qualified Name in an SVN-limited or firmware-limited hierarchy.

As the use-case for SVN-limited objects involves protection from vulnerable firmware that has access to base hierarchy seeds and proof values, users should certify SVN-limited objects via a firmware-limited or SVN-limited AK. The AK's chain of trust should traverse only other firmware-limited or SVN-limited TPM objects, chaining back to a firmware-limited EK. See the EK Credential Profile [12] for guidance on establishing trust in a TPM's EK.

42 Read-Only Mode of Operation

42.1 Introduction

This clause describes a Read-Only mode of operation that is defined to allow specialized TPM profiles or platform applications to support use cases where a TPM's ownership and/or provisioning steps may be done in ways, or at times, which vary from the typical PC manufacturing pipeline and Operating System first boot. The purpose for defining these features in this Specification is to provide a standard way for TPM clients (such as a generalized high-level operating system, custom platform firmware, or other entity) to understand whether a particular TPM supports the Read-Only mode of operation, whether it is currently operating in the Read-Only mode, and/or to request the TPM enter or exit the Read-Only mode.

42.2 Description

The Read-Only mode of operation can be considered an extension of the hierarchy disable flags in TPMA_STARTUP_CLEAR. Whereas disabling a hierarchy prevents any use of a hierarchy, the Read-Only mode of operation prevents making persistent changes, including object eviction, NV update operations, authorization changes, and so on. However, the Read-Only mode of operation is applied equally across all enabled TPM hierarchies.

A TPM can enter and exit Read-Only mode with Platform Authorization by using the TPM2_ReadOnlyControl() command, or by using a vendor-defined mechanism. A TPM exits Read-Only mode if the TPM receives a TPM2_Startup() for TPM Reset or TPM Restart. However Read-Only mode will remain enabled during TPM Resume. The current Read-Only mode of operation is reported via TPMA_STARTUP_CLEAR.

Part 2 of this Specification defines the following values to support the Read-Only mode of operation:

1. A bit in TPMA_STARTUP_CLEAR that indicates the current Read-Only mode of the TPM. The Read-Only mode bit will be SET if the TPM is in Read-Only mode and will be CLEAR otherwise. Operating systems that are unaware of, or ignore, this bit may encounter errors when attempting commands that would typically succeed.
2. The error code TPM_RC_READ_ONLY, which indicates that an operation was rejected because the operation required a change to TPM state that is unsupported while in Read-Only mode. A device supporting Read-Only mode must reject any affected command before performing authorization checks which would be evaluated during the execution of the command.
3. A command code, TPM_CC_ReadOnlyControl. A client can determine whether a given TPM supports Read-Only mode by inspecting the supported command list with TPM2_GetCapability(*capability* == TPM_CAP_COMMANDS, *property* == TPM_CC_ReadOnlyControl).

42.3 Command Behavior

While in the Read-Only mode of operation, most of the TPM commands will continue to function as otherwise documented. However, several common commands are not permitted and will return TPM_RC_READ_ONLY. A command that is not permitted will have no effect on the TPM state.

A complete list of the commands permitted and not permitted while in the Read-Only mode of operation is available in Part 3, "TPM2_ReadOnlyControl".

The following list is not exhaustive but describes many of the common operations that continue to function normally, as documented, when the TPM is in Read-Only mode.

- TPM initialization and orderly shutdown with commands _TPM_Init(), TPM2_Startup() and TPM2_Shutdown() continue to operate normally.

- TPM self-tests may be performed. This includes TPM2_SelfTest(), full or partial, as well as TPM2_IncrementalSelfTest().
- All forms of platform measurement are supported. This includes TPM2_PCR_Read(), TPM2_PCR_Extend(), TPM2_PCR_Event(), TPM2_PCR_Reset(), TPM2_HashSequenceStart(), TPM2_SequenceUpdate(), TPM2_SequenceComplete(), and related commands.
- Reading the TPM capabilities can be performed using the TPM2_GetCapability() and TPM2_TestParms() commands.
- Existing NV indexes can be read. The command TPM2_NV_Read() can be used to read the value of an NV index and the command TPM2_NV_ReadPublic() can be used to read the public area of an NV index.
- NV indexes can be used like additional PCRs with TPM2_NV_Extend(). The NV index must be defined as TPMA_NV_ORDERLY and TPMA_NV_CLEAR_STCLEAR.
- Transient Objects can be loaded. For example, TPM2_Import() can be used to import a duplicate key, and TPM2_Load() can be used to load a previously created key.
- Context management is permitted. This includes TPM2_ContextSave(), TPM2_ContextLoad() and TPM2_FlushContext().
- Operations that take a loaded handle, whether transient or persistent, can be executed. TPM2_ActivateCredential(), TPM2_Unseal(), TPM2_RSA_Encrypt(), TPM2_RSA_Decrypt(), TPM2_Certify(), TPM2_Quote(), TPM2_Sign(), TPM2_VerifySequenceComplete(), TPM2_NV_Certify(), and TPM2_Duplicate(), are all examples of commands that can be used with existing handles.
- All session types are permitted. This includes trial and policy authorization sessions, audit sessions, as well as encrypt/decrypt sessions. For example, the TPM2_StartAuthSession() command can be used to start a PCR policy session that will be specified when calling TPM2_Unseal() on an object.
- All of the policy evaluation commands, such as TPM2_PolicySecret(), TPM2_PolicyPCR() and TPM2_PolicyOR(), are permitted.
- Dictionary attack counters continue to function and are enforced. Additionally, TPM lockout due to successive authorization failures can be cleared with TPM2_DictionaryAttackLockReset().
- The non-volatile component on the TPM clock will continue to update at the rate reported by the TPM_PT_CLOCK_UPDATE property returned from TPM2_GetCapability().

The following list is not exhaustive but describes many of the common operations that are explicitly not permitted when the TPM is in Read-Only mode. The operations in this list will return TPM_RC_READ_ONLY when the TPM is in the Read-Only mode of operation.

- Creating new objects is not permitted. This includes creating an object with TPM2_Create() or TPM2_CreatePrimary(). Although creating a new object does not modify persistent TPM state, the Read-Only mode of operation permits only loading previously created or externally created objects.
- Modifying persistent handles with the use of TPM2_EvictControl() is not permitted.
- Defining new NV space or updating existing NV space is not permitted. This includes TPM2_NV_DefineSpace(), TPM2_NV_UndefineSpace(), TPM2_NV_Write(), TPM2_NV_Increment(), and so on.
- While all of the policy evaluation commands are supported, a TPM2_PolicySecret() assertion which references a PIN Pass or PIN Fail NV Index is not permitted.
- All authorization changes, for example to existing objects, NV space, PCRs, hierarchies, and so on, are not permitted. This includes TPM2_ObjectChangeAuth(), TPM2_NV_ChangeAuth(), TPM2_PCR_SetAuthValue() and TPM2_HierarchyChangeAuth().

- Hierarchy changes are not permitted. This includes TPM2_HierarchyControl(), TPM2_Clear(), TPM2_ChangeEPS() and TPM2_ChangePPS().
- While dictionary attack counters continue to function and are enforced, the lockout parameters cannot be modified using TPM2_DictionaryAttackParameters().
- Setting or adjusting the TPM clock is not permitted using the commands TPM2_ClockSet() and TPM2_ClockRateAdjust().

This specification does not define the behavior of vendor-defined commands in Read-Only mode.

43 RSA

43.1 Introduction

The RSA asymmetric algorithm is used for digital signatures, secret sharing, and encryption.

A TPM that supports RSA should support a public modulus size of at least 2,048 bits. Support for other key sizes is permitted.

When the size (*k*) of the public modulus (*n*) of an RSA key is given, then $\lfloor log_2(n) \rfloor = (k - 1)$. Additionally, for a two-prime system, the primes (*p* and *q*) satisfy $\lfloor log_2(p^2) \rfloor = (k - 1)$ and $\lfloor log_2(q^2) \rfloor = (k - 1)$.

The RSA algorithm requires the methods of encryption and signing defined in RFC 8017 [13]. This includes support for RSAES-OAEP, RSAES-PKCS1-v1.5, RSASSA-PKCS1-v1.5, and RSASSA-PSS.

The RSA structures in this specification support only public keys that are the product of two primes. Support for other numbers of primes is allowed, but it is performed in a vendor-specific manner and thus beyond the scope of this specification.

A TPM is required only to support a public exponent (*e*) of $2^{16} + 1$. Support for other exponents is allowed but discouraged.

Note:

The Reference Code does not support an exponent size smaller than 7 nor does it allow keys to be created on the TPM with a public exponent less than $2^{16} + 1$.

When loading an RSA key, the TPM validates that its public and private portions are properly paired by dividing the public modulus by the single private prime and requiring that the remainder be zero. The TPM does not validate whether input values are primes.

Note:

Validating the pairing of the public and private key portions need not be performed when the key is being loaded. However, this check is performed before the authorization value of the key or the private portion of the asymmetric key may be used.

The TPM will also validate that the provided and computed prime factors are in an acceptable range. To be acceptable, the square of the prime is required to have the same number of significant bits as the public modulus.

43.2 RSAEP

This is the RSA public key primitive defined in RFC 8017 [13], clause 5.1.1. It is a modular exponentiation of a message (*m*) with the public exponent (*e*), modulo the public modulus (*n*) to produce the cipher text (*c*). This is expressed as:

$$c := m^e \pmod{n} \tag{58}$$

where

c	is the encrypted message
m	is a value $0 < m < n$ to be encrypted
e	is the public exponent (default is $2^{16} + 1$)

(continued on next page)

(continued from previous page)

n is the public modulus

43.3 RSADP

This is the RSA private key primitive defined in PSCS#1v2.1, clause 5.1.2. This clause describes the private key in two forms: as a pair and as a quintuple. The Reference Code uses the pair form with a private exponent (*d*). Using this form, the RSADP operation recovers a message from a cipher text by:

$$m := c^d \pmod{n} \tag{59}$$

Note:

The Reference Code also supports use of the CRT form of the private exponent.

43.4 RSAES_OAEP

This encryption scheme is defined in RFC 8017 [13]. It is the only scheme used with an RSA-restricted decryption key. The algorithm identifier for this scheme is TPM_ALG_OAEP.

For RSA keys protecting a secret value (such as, an encryption key or a session secret), the *L* parameter is a byte stream, the last byte of which must be zero, indicating the intended use of the encrypted value. A command that accepts or creates an RSA-encrypted secret indicates the value of the string to use for *L*. The RSA key's *scheme* hash algorithm (or, if it is TPM_ALG_NULL, the RSA key's Name algorithm) is used to compute *lhash* := **H**(*L*), and the null termination octet is included in the digest.

MGF1 (as defined in IEEE Std 1363^{TM}-2000 [14]) computes *dbMask* and *seedMask.* The mask-generation function uses the Name algorithm of the RSA key as the hash algorithm.

43.5 RSAES-PKCS-v1_5

This encryption scheme is defined in RFC 8017 [13]. It has no parameters. The algorithm identifier for this scheme is TPM_ALG_RSAES.

43.6 RSASSA-PKCS1-v1_5

This signing scheme is defined in RFC 8017 [13]. The algorithm identifier for this scheme is TPM_ALG_RSASSA.

An RSA-restricted signing key may use either this algorithm or RSASSA_PSS, but not both. An unrestricted signing key may select as its default either this algorithm or RSASSA_PSS. If TPM_ALG_NULL is selected, the caller will indicate the scheme in the signing command.

This signature scheme prepends an OID to a digest before signing with the private key. It may be used in any command that allows an asymmetric signing operation.

For signing commands that use restricted signing keys, the TPM provides the OID that corresponds to the digest algorithm, and the OID provided by the caller is discarded.

For commands that use unrestricted signing keys, the TPM uses the caller-provided OID.

Note:

If the command does not provide a parameter for the OID, then the TPM provides the OID even if the key is not restricted.

For hash algorithms where the TCG defines a TPM_ALG_ID, the TCG provides the OID to use with restricted signing keys. Currently, the defined values are:

- SHA1

30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14_{16}

- SHA256

30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20_{16}

- SHA384

30 41 30 0d 06 09 60 86 48 01 65 03 04 02 02 05 00 04 30_{16}

- SHA512

30 51 30 0d 06 09 60 86 48 01 65 03 04 02 03 05 00 04 40_{16}

Note:

These values are from RFC 8017 [13].

Note:

The listing above is not normative. TCG maintains the normative list.

43.7 RSASSA_PSS

This signing scheme is defined in RFC 8017 [13]. The algorithm identifier for this scheme is TPM_ALG_RSAPSS.

A restricted signing key may use either this algorithm or RSASSA-PKCS1-v1_5, but not both. An unrestricted signing key may use either this algorithm, RSASSA-PKCS1-v1_5, or TPM_ALG_NULL. If TPM_ALG_NULL is selected, the caller can indicate the signing scheme in the signing command.

When used with a restricted signing key, the hash algorithms for messages (M) and M' are the same.

When used with an unrestricted signing key, the hash algorithm for M and M' can differ.

For both restricted and unrestricted signing keys, the random salt length is the largest size allowed by FIPS 186-5 [11].

Note:

TPM implementations from prior to the publication of FIPS 186-4 [15] (prior to TPM rev 1.63), which introduced the salt length limitation, may use the largest size allowed by the key size and message digest size.

43.8 RSA Key Generation

43.8.1 Background

The implementation of the RSA key-generation function should meet the requirements of the intended market. The methods in FIPS 186-5 [11] are recommended.

In the Reference Code, the primes used for the key are generated using the methods of FIPS 186-5, A.1.3 "Generation of Random Primes that are Probably Prime."

Note:

FIPS 186-5 only allows this method to be used for primes of 1024 bits or larger. For smaller primes, the methods described in clause A.1.5 “Generation of Probable Primes with Conditions Based on Auxiliary Provable Primes” or clause A.1..6 “Generation of Probable Primes with Conditions Based on Auxiliary Probable Primes” can be used if FIPS compliance is required.

43.8.2 Large Prime Generation

For generating a prime the Reference Code has two different implementations: one using testing of candidates and the other using a number sieve. The process for testing of candidates is described in this clause.

The inputs are:

- *primeSize* - this is the number of bits in the prime to be generated. It should be half the number of bits in the public modulus to be generated
- *e* - the public exponent

  Note:

  In the Reference Code, the exponent is required to be an odd number > 2^{16}

- a random number generation function according to the type of key being generated (see Clause 24.6.2 and Clause 24.6.3)

Note:

Derivation of RSA keys is not supported.

The prime generation process is:

1. set prime candidate *p* to the next *primeSize* number of bits from the provided random number generation function
2. adjust *p* so that the high-order two bits and the low order bit are one

   Note:

   In the Reference Code, when a prime is generated, the upper two octets for prime candidates are verified to be B5 05_{16} or greater. This forces the prime to be greater than 0.7071075439453125 * $2^{\frac{n}{2}}$ where n is the number of bits in the public modulus. This is slightly larger than the required value of $\frac{\sqrt{2}}{2} \cdot 2^{\frac{n}{2}}$. This value ensures that the MSb of the product of these two primes will be SET. Setting the two most significant bits would also ensure that the magnitude of the product is large enough but would reduce the range of allowed primes by a small factor (about 4.3%).

3. test *p* to determine if it is probably prime
   1. Using a greatest common divisor (**GCD**()) function, see if *p* shares any common factors with a composite number that is the product of the first 1024 primes and if so, go to (1).
   2. do *N* rounds of Miller-Rabin where *N* is determined by the size of the prime and if the test fails on any round, go to (1)

> ***Note:***
>
> The value for N is found in FIPS 186-5, Appendix B. [11]
>
> The witness values used by Miller-Rabin are from the same random number function used to generate the prime candidate.

4. return *p*

43.8.3 RSA Key Generation Algorithm

The key generation process is:

1. initialize the values of the algorithm
    1. Set *securityStrength* according to the size of the public modulus of the key to be generated as specified in SP 800-57 (part 1).
    2. $primeSize = \frac{1}{2}$ the size of the RSA modulus (*inPublic.parameters.keyBits* of the template)
2. find a first prime (*p*) using the method in Clause 43.8.2
3. find a second prime (*q*) using the method in Clause 43.8.2:
4. If $|p - q| \leq 2^{\frac{nLen}{2}-100}$, go to step (2).
5. compute the public modulus $n := p \cdot q$

> ***Note:***
>
> Depending on the starting values, the algorithm could take many iterations to find two suitable primes.

6. compute the private exponent $d := e^{-1}$ mod (p - 1)(q - 1)

> ***Note:***
>
> The Reference Code also provides an option to use the CRT form of the private exponent *d*.

7. if d < $2^{\frac{nLen}{2}}$ where *nLen* is the number of bits in the public modulus (*n*), then go to step (2).

> ***Note:***
>
> If required, a random value is encrypted with the public exponent and decrypted with the private exponent to validate that the key can be used for signing and signature verification.

8. return *n*, *p,* and *d*

43.9 RSA Cryptographic Primitives

43.9.1 Introduction

When RSA is implemented on a TPM, it may provide these additional commands to support cryptographic operations. The command description in Part 3 indicates the restrictions on the types of keys that may be used with each of the commands.

43.9.2 TPM2_RSA_Encrypt()

TPM2_RSA_Encrypt() may be used to perform encryption according to the methods described in RFC 8017 [13]. If the scheme of the key is TPM_ALG_NULL, then the encryption scheme may be specified in the command. Otherwise, the scheme specified in the key will be used. The scheme options are:

- TPM_ALG_NULL - selects RSAEP as described in Clause 43.2
- TPM_ALG_OAEP - selects RSAES_OAEP as described in Clause 43.4
- TPM_ALG_RSAES - selects RSAES-PKCS-v1_5 as described in Clause 43.5

43.9.3 TPM2_RSA_Decrypt()

TPM2_RSA_Decrypt() performs the decryption operations defined in RFC 8017, clause 7.1.2 [13]. The handle used in this command is required to have the *decrypt* attribute SET. If the scheme of the key is TPM_ALG_NULL, then the encryption scheme may be specified in the command. Otherwise, the scheme specified in the key will be used. The scheme options are:

- TPM_ALG_NULL - selects RSADP as described in Clause 43.3
- TPM_ALG_OAEP - selects RSAES_OAEP as described in Clause 43.4
- TPM_ALG_RSAES - selects RSAES-PKCS-v1_5 as described in Clause 43.5

43.10 RSA Labeled KEM

43.10.1 Overview

The RSA implementation of the Labeled KEM (Clause 8.4.5.2) is based on OAEP (Clause 43.4) and depicted in Figure 22.

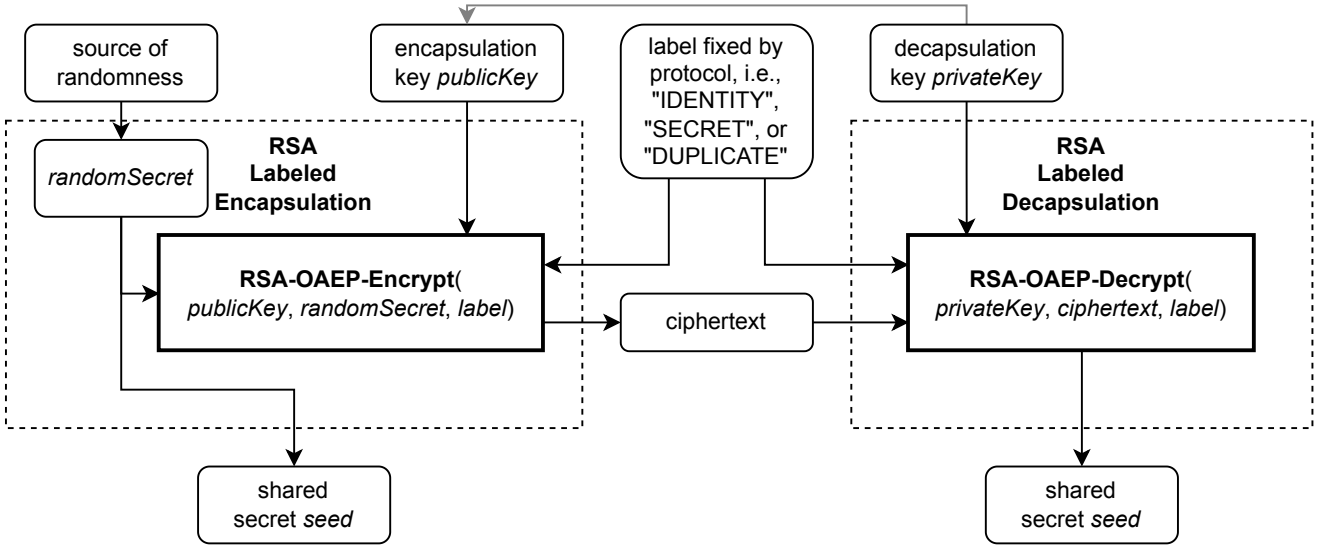


Figure 22: RSA Labeled KEM

For RSA labeled encapsulation, a random secret is OAEP-encrypted to the TPM public key. The protocol label is used as the OAEP label. The OAEP hash algorithm used is:

- The hash algorithm indicated in the key's *scheme*, if the key's *scheme* is TPM_ALG_OAEP
- The key's *nameAlg* if the key's *scheme* is TPM_ALG_NULL

If the key's *scheme* is not TPM_ALG_OAEP or TPM_ALG_NULL, the TPM must return an error (TPM_RC_VALUE).

Note:

The OAEP hash algorithm is always *nameAlg* in the case of restricted decryption keys, because the *scheme* is required to always be TPM_ALG_NULL in such cases. See Part 2, "TPMS_RSA_PARMS".

The size of the random secret is limited to the size of the digest produced by the *scheme* hash algorithm (or *nameAlg* if the scheme hash algorithm is TPM_ALG_NULL) of the object that is associated with the public key used for OAEP encryption.

Note:

It is recommended to use a random secret that is exactly the same size as the digest produced by the relevant hash algorithm.

44 ECC

44.1 Introduction

The ECC algorithm is used for digital signatures and for secret sharing.

Note:

As implemented in a TPM, ECC is not used directly for encryption of data. Rather, ECDH secret sharing is used to establish a symmetric key, and then a symmetric algorithm is used for the actual data encryption.

A TPM should support prime modulus ECC.

If the ECC algorithm is supported, the TPM is required to support ECDSA and ECDH.

The TPM should support ECC key sizes of at least 256 bits. Support for other key sizes is allowed.

Note:

It is anticipated that the recommended ECC key size will increase over time in revisions to this specification.

The TPM does not check the security of ECC curve parameters. It does check that the public and private keys are properly paired.

Note:

Validating the pairing of the key's public and private portions need not be performed when the key is being loaded. However, this check is required to be performed before the authorization value of the key or the private portion of the asymmetric key may be used.

44.2 Split Operations

44.2.1 Introduction

Several of the EC schemes us two-phase protocols in which the TPM generates an ephemeral key pair in the first phase and uses that ephemeral key in the second phase. These protocols require that the ephemeral key only be used once. Ordinary TPM keys have context that may be saved and restored by TPM context management. This clause describes the methods used to implement the required single use ephemeral keys.

44.2.2 Commit Random Value

A split operation requires two TPM commands the first of which is `TPM2_Commit()`. It uses a TPM-generated, random value in the commit computation. A second command (such as, any of the signing commands) completes the split signing operation and uses the same commit value. The random commit value is required to:

- have at least the number of bits equal to the security strength of the signing key;
- not be known outside of the TPM; and
- only be used once.

Because the random value is not allowed to be known outside of the TPM, the TPM is required to store the random value between the two commands in split sequence. To allow more than one split sequence to be in process at a time, the TPM may have an array of values and return a count value as one of the response parameters of the `TPM2_Commit()` indicating the array entry being used for the sequence. This count value is an input to the TPM in the command that completes the split sequence.

Note:

The number of split sequences supported by the TPM may be found using TPM2_GetCapability(*capability* == TPM_CAP_TPM_PROPERTIES, *property* == TPM_PT_SPLIT_MAX).

To minimize the size of the array used for storing these values, a TPM may generate pseudo-random values instead.

If using pseudo-random values, the TPM creates the value using **KDFa**(), a counter (*commitCount*), and a random value (*commitRandom*). On each TPM Reset, the TPM will select a new random value for *commitRandom* and reset *commitCount* to zero. On TPM2_Commit(), the TPM would use the current value of the *commitCounter* to generate the pseudo-random value (*r*) by

$$r := \mathbf{KDFa}(nameAlg,\ commitRandom,\ \text{"ECDAA Commit"},\ name,\ commitCount,\ bits) \quad (60)$$

where

nameAlg	is the *nameAlg* of the signing key (*signHandle*)
commitRandom	is the current value of *commitRandom*
"ECDAA Commit"	is a value used to differentiate uses of **KDFa**()
name	is the Name of *signHandle*
commitCount	is the current value of *commitCount*
bits	is the number of bits in the order of the curve of the signing key (signHandle)

Note:

When the number of bits is not a multiple of 8, it is rounded up to be a multiple of 8.

To track the usage of the *commitCount*, the TPM maintains a bit array (*A*[]) that has a power of 2 number of bits (*N*) (that is, the bits indexes of *A*[] are from 0 to $2^N - 1$). After computing the value of *r*, the low-order *N* bits of *commitCount* are used to index *A*[] and the corresponding bit is SET. The low-order 16 bits of *commitCount* are returned as the *counter* parameter.

44.2.3 TPM2_Commit()

TPM2_Commit() performs the first part of a split operation. The TPM will perform the point multiplications on the provided points and return intermediate signing values. Alternatively, the TPM will simply return a public ephemeral key based on a commit private value. The *signHandle* parameter refers to an ECC key. TPM2_Commit() has the following parameters, all of which are optional.

P_1	is a point on the curve used by *signHandle* (a TPM2B_ECC_POINT)
s_2	is an octet array used to derive x-coordinate of a base point (a TPM2B_ECC_PARAMETER)
y_2	is the y-coordinate of the point associated with s_2 (a TPM2B_ECC_PARAMETER)

Note:

P_1 is a TPM2B_ECC_POINT, a sized buffer containing a TPMS_ECC_POINT. It is not a sized buffer containing an array of bytes. A size of zero for the TPM2B_ECC_POINT will create an unmarshaling error because the minimum size for P_1 is 4 (two ECC parameters, both of which are Empty Buffers). If P_1 is an Empty Buffer, the TPM returns TPM_RC_INSUFFICIENT regardless of s_2 and y_2. If P_1 is an Empty Point and s_2 and y_2 are Empty Buffers, then the TPM will set E := [r]G where r is the commit random value and G is the generator point for the curve.

In the algorithm below, the following additional values are used in addition to the command parameters:

$\mathbf{H}_{nameAlg}$	is a hash function using the *nameAlg* of the key associated with *signHandle*
p	is the field modulus of the curve associated with *signHandle*
n	is the order of the curve associated with *signHandle*
d_s	is the private key associated with *signHandle*
G	is the generator of the curve associated with *signHandle*
c	is the counter that increments each time TPM2_Commit() is executed
A[i]	is an array of bits used to indicate when a value of *c* has been used in a signing operation; the values of *i* are 0 to 2*N*-1.
N	is the $\log_2$ of the number of values in *A*
k	is a nonce that is set to a random value each on each TPM Reset; the nonce size is twice the security strength of any ECDAA key supported by the TPM.

The commit algorithm is:

1. Validate that s_2 and y_2 are either both Empty Buffers or both not Empty Buffers (TPM_RC_SIZE)
2. If s_2 is an Empty Buffer, skip to step (5)
3. compute $x_2 := \mathbf{H}_{nameAlg}$ (s_2) (mod *p*)
4. if (x_2, y_2) is not a point on the curve of *signHandle,* return TPM_RC_ECC_POINT
5. if P_1 is not an Empty Point and P_1 is not a point on the curve of *signHandle*, return TPM_RC_ECC_POINT
6. set *K*, *L*, and *E* to be Empty Buffers
7. generate or derive *r* (see Clause 44.2.2)
8. set $r := r \bmod n$
9. if s_2 is not an Empty Buffer, set K := [d_s] (x_2, y_2) and L := [r] (x_2, y_2)
10. if P_1 is not an Empty Point, set E := [r](P_1)
11. if P_1 is an Empty Point and s_2 is an Empty Buffer, set $E := [r]G$
12. if *K, L,* or *E* is the point at infinity, return TPM_RC_NO_RESULT
13. set *counter* := *commitCount*
14. set *commitCount* := *commitCount* + 1

Note:

Depending on the method of generating *r*, it can be necessary to update the tracking array here.

15. output *K, L, E,* and *counter*

Note:

Depending on the input parameters, K and L or E can be Empty Points

44.2.4 TPM2_EC_Ephemeral()

`TPM2_EC_Ephemeral()` is similar to `TPM2_Commit()` in that it uses the commit random value to generate an ephemeral key for use in a two-phase operation. However, `TPM2_EC_Ephemeral()` only used the random value *r* to generate a corresponding public key $Q := [r]G$ where *G* is the generator point for a specified curve.

As with `TPM2_Commit()`, a counter value is returned. This value needs to be used in a subsequent command in order to complete the two-phase operation.

44.2.5 Recovering the Private Ephemeral Key

To complete a split or two-phase operation, the TPM uses the same random or pseudo-random value generated in `TPM2_Commit()`. The random or pseudo-random value is determined by the *counter* field provided as an input parameter for the command that is the second phase of the split operation.

If the values are stored in an array, *counter* is used to index the array and, after the value is used in the signing operation, the value is erased. If using the pseudo-random method, the following algorithm is used to reconstruct the random value.

1. set $t := $ low-order 16 bits of *commitCount*
2. verify that $t - 2^N < counter < t$ else return `TPM_RC_RANGE`
3. set $i := $ low-order *N* bits of *counter*
4. if *A[i]* is CLEAR, return `TPM_RC_VALUE`
5. set $c := commitCount - t$
6. if $counter \geq t$ then $c := c - 2^{16}$
7. $c := c + counter$
8. compute *r* as in Equation 60 using *c* in place of *commitCount*
9. if the command completes successfully set $A[i] := 0$

44.3 EC Signing

44.3.1 ECDSA

For a TPM compliant with this specification, the default ECC signing scheme (ECDSA) is as defined in ISO/IEC 14888-3 [16].

ECDSA signatures are supported by `TPM2_SignDigest()` and `TPM2_SignSequenceComplete()`, along with the (deprecated) `TPM2_Sign()` and `TPM2_VerifySignature()` commands.

44.3.2 EdDSA

44.3.2.1 Introduction

EdDSA is defined in RFC 8032 [17] and used by Curve25519 and Curve448. PureEdDSA is supported as the algorithm ID TPM_ALG_EDDSA and HashEdDSA is supported as TPM_ALG_HASH_EDDSA.

TPM_ALG_EDDSA keys can only be used with TPM2_SignSequenceComplete() and TPM2_VerifySequenceComplete(). TPM_ALG_HASH_EDDSA keys can be used with either of TPM2_SignSequenceComplete() or TPM2_SignDigest(), and either of TPM2_VerifySequenceComplete() or TPM2_VerifyDigestSignature().

Unlike other ECC schemes, TPM_ALG_EDDSA and TPM_ALG_HASH_EDDSA do not have a configurable hash algorithm as part of the signing scheme. It is fixed by RFC 8032.

For TPM_ALG_HASH_EDDSA keys, the *digest* passed to TPM2_SignDigest() and TPM2_VerifyDigestSignature() is computed using the correct pre-hashing algorithm for the curve (a 64-byte SHA-512 digest in the case of Curve25519, or a 64-byte SHAKE256-512 digest in the case of Curve448).

The same key cannot be used with both TPM_ALG_EDDSA and TPM_ALG_HASH_EDDSA.

The TPM enforces this requirement by not allowing scheme = TPM_ALG_NULL for EdDSA/HashEdDSA keys (signing keys on Curve25519 or Curve448). See Part 2, “TPMS_ECC_PARMS” for more details.

44.3.2.2 EdDSA Signature Representation

EdDSA signatures are represented in the encoding specified in RFC 8032 [17] as a single TPM2B in little-endian. See TPM2B_SIGNATURE_EDDSA in Part 2.

44.3.3 ECDAA

44.3.3.1 Introduction

If a TPM supports ECC, it is recommended that it also support the ECDAA scheme described in this clause.

Direct Anonymous Attestation based on ECC (ECDAA) is a TPM signature scheme that provides anonymous signatures (meaning that different signatures from the same signer cannot be correlated), or pseudonymous signatures (meaning that different signatures from the same signer can be correlated but the identity of the signer is still unknown). Multiple ECDAA schemes are supported in this TPM implementation.

The TPM signs data with an ECDAA key in an unconventional way. A Verifier verifies signature values using data equivalent to a public key and verifies the public key using data equivalent to a certificate (which is also called a credential) supplied by an Issuer. However, the public key and the credential are randomized by the TPM and the TPM’s Host platform before they are sent to the Verifier. This prevents both the Verifier and the Issuer from identifying the TPM that created the signature value.

It is anticipated that the most common use of ECDAA will be to certify (TPM2_Certify()) a TPM object (usually a key). A credential issuer will provide a certificate for an ECDAA key. This certificate will validate that the ECDAA key belongs to a valid TPM without disclosing the ECDAA key. That ECDAA key may then be used to certify other TPM objects. These certificates prove that the certified object belongs to a valid TPM without disclosing the identity of that TPM. If the certified key is a signing key, it may then be used to attest to various TPM states, without disclosing the identity of the TPM to which it belongs.

This scheme is substantially different from the AK scheme described in Clause 21 in that the ECDAA key may be used to provide the anonymity for keys rather than having to send each new attestation key to a privacy certificate authority (PCA) in order to have an anonymizing certificate produced. After a certificate has been obtained for an ECDAA key, it may be used to produce anonymized certificates for many TPM keys without requiring additional interaction with a privacy CA.

An ECDAA key may be used in any command that produces a signature. The TPM may not be used to verify an ECDAA signature.

44.3.3.2 ECDAA Key Generation on the TPM

While any signing key may be an ECDAA key, it is most useful as a Primary Key in the Endorsement hierarchy. This ensures that a TPM will normally produce the same ECDAA keys and receive the same credentials from a given Issuer, no matter how many times the credentialing process is performed, and no matter how many owners the TPM has had. This property is desirable because an Issuer should only give credentials to a platform after verifying that the platform has the architecture of a trusted platform. The Issuer would give replacement (different) credentials only when it is necessary to retire the old credentials. Replacement credentials erase the previous DAA history of the platform, at least as far as the credentials from that issuer are concerned. Replacement might be desirable, as when a platform changes hands, for example, in order to eliminate any association via DAA between the seller and the buyer. On the other hand, replacement might be undesirable, since it enables a rogue to rejoin a community from which it has been barred. Replacement is done by submitting a different `TPMT_PUBLIC`.*unique* field value to the TPM when the key is created (`TPM2_Create()` or `TPM2_CreatePrimary()`). Software may use any value of `TPMT_PUBLIC`.*unique* field at any time, in any order, but the Issuer can detect when a request uses a different value from the previous request and could reject the request.

The cryptographic parameters of the curve are indicated by the template in the command (`TPM2_Create()` or `TPM2_CreatePrimary()`) that creates the curve. The curve ID depends on the Issuer who is expected to provide a credential for the DAA key (different Issuers may require different curves). The TPM generates a private key (d_s) and a public key (Q_s). The non-cryptographic parameters in the template (that is, object attributes and signing scheme) are chosen by the entity that calls the command to create the DAA key. Inappropriate choice of the non-cryptographic parameters will cause the Issuer to reject an application for a DAA credential.

Due to the additional algebraic structure of the pairing-friendly curves required for ECDAA, the security strength of an ECDAA key may be less than the security strength of other ECC keys of the same size.

If the Endorsement Primary Seed is used as the DAA seed, then, like other EK, an ECDAA key will change whenever the EPS is changed.

The process for generating an ECDAA key is identical to the process used for any ECC key.

For the TCG defined ECDAA protocol, the curve described by p, n, and b is a Barreto-Naehrig (BN) elliptic curve. BN curves are of the form $y^2=x^3+b$ as defined in ISO/IEC 15946-5, Clause 7.3 [18] and IEEE P1363.3 Clause A.11.5 [14].

Note:

The linear term (a) of generic ECC curves (curves with the form $y^2=x^3+ax+b$) is zero in BN curves. All BN curves are suitable but some are less efficient than others. The BN curves recommended in this version of DAA were chosen by the DAA designers.

The cryptographic value of the public key in the resultant TPM key structure is Q_s, which is used by the Issuer when computing the membership credential on the DAA private key d_s. Q_s is not used to verify the DAA signatures produced by the TPM and corresponding host platform.

44.3.3.3 ECDAA Sign Operation

The ECDAA scheme may be used in any command that uses a signing key. These are, the attestation group and `TPM2_Sign()`.

For an attestation command using the ECDAA scheme, both the *qualifiedSigner* and *extraData* fields in the attestation block (a `TPMS_ATTEST`) are set to be the Empty Buffer before the data is hashed. The attestation

data is then marshaled and hashed. The resulting hash data is then concatenated to the first hash to produce the value to sign (*P*).

$$P := \mathbf{H}_{schemeHash}(qualifyingData \parallel \mathbf{H}_{schemeHash}(\text{TPMS\_ATTEST})) \quad (61)$$

For `TPM2_Sign()`, the value to sign is the input *digest* and

$$P := digest \quad (62)$$

To complete the ECDAA sign operation, the TPM uses the same random or pseudo-random value (*r*) used in `TPM2_Commit()`. The value is determined by the *counter* field in the scheme parameter of the signing command. This parameter is used in the process defined in Clause 44.2.5.

The signature is created using a modified Schnorr signature using the *P* and *r* values described above:

1. set *k* to a random value such that 0 < *k* < *n*
2. compute $T := \mathbf{H}(k \parallel P) \pmod n$
3. compute integer $s := (r + Td_s) \pmod n$
4. if *s* = 0, output failure (negligible probability)

The signature is the tuple (*k* , *s*).

Note:

The *k* value is returned in the *R* parameter of the `TPMT_SIGNATURE` structure.

44.3.4 EC Schnorr

44.3.4.1 Introduction

If a TPM supports ECC, it should support the `TPM_ALG_ECSCHNORR` scheme.

The scheme description uses the following values:

G	generator point for the curve of the signing key
d_S	private value of the signing key
Q_S	public point of the signing key ($Q_S := [d_S]G$)
n	order of *G*
$\mathbf{H}_{schemeHash}$	hash algorithm specified in the signing scheme

44.3.4.2 EC Schnorr Sign

An EC Schnorr signature is generated when the signing scheme for a key is `TPM_ALG_ECSCHNORR`. The scheme many be used in any signing operation

To sign a digest *P*

1. set *k* to a random value such that 0 < *k* < *n*
2. compute $E := (x_E, y_E) := [k]G$

3. if *E* is the point at infinity, go to (1)
4. compute $r := \mathbf{TRUNC}(\mathbf{H}_{schemeHash}(\mathbf{FE2BS}(x_E) \| P, n)$

> ***Note:***
>
> x_E is a field element with the same number of bits as the curve order *n*

> ***Note:***
>
> **TRUNC**() is a function that reduces the number of octets in the first argument until it has no more octets than the second argument. Truncation occurs from the less significant end of the number. If the digest produced by $\mathbf{H}_{schemeHash}$ has the same number of octets as the curve order *n*, then no truncation occurs.

> ***Note:***
>
> **FE2BS**() is a function that converts the number x_E (a field element) into a canonical value (octet or byte string) with the same number of octets as the field order *n*. This can result in a value with leading octets of zero. As x_E is computed (mod *p*) the value may be greater than *n*

5. compute integer $s := (k + rd_S) \pmod{n}$

> ***Note:***
>
> This is the same computation as step (3) in Clause 44.3.3.3.

6. if *s* = 0 or *s* = *k* go to (1)

> ***Note:***
>
> The *s* = *k* check is to eliminate the possibility that 0 = *r* (mod *n*). Optionally, an implementation could check after (4) that 0 ≠ *r* (mod *n*).

The signature is the tuple (*r, s*).

44.3.4.3 EC Schnorr Signature Validate

To validate a Schnorr signature (*r, s*) over digest *P*

1. verify that 0 < *s* < *n*
2. compute $(x_E, y_E) := [s]G + [-r]Q_S$
3. compute $r' := \mathbf{TRUNC}(\mathbf{H}_{schemeHash}(\mathbf{FE2BS}(x_E) \| P), n)$
4. the signature is valid if *r'* = *r*

> ***Note:***
>
> The comparison of *r'* and *r* is done assuming that both values are numeric and not octet strings. This reduces the chance of interoperability problems due to padding performed on *r*.

44.4 ECC Key Encapsulation Mechanism

44.4.1 Overview

An ECC key with the *decrypt* bit SET can be used with the `TPM2_Encapsulate()` and `TPM2_Decapsulate()` commands if the *kdf* field of the `TPMS_ECC_PARMS` is not `TPM_ALG_NULL`. In this case, the key can be used as a Key Encapsulation Mechanism (KEM) key, and the KEM is equivalent to the Diffie-Hellman based KEM (DHKEM) from RFC 9180: Hybrid Public Key Encryption [19], using the selected KDF.

44.4.2 Encapsulation

`TPM2_Encapsulate()` is equivalent to DHKEM's `Encaps` function.

Given:

- Recipient's public key *pkR*

The TPM will:

1. Generate an ephemeral secret key *skE* and corresponding public key *pkE*
2. Compute *dh*, the Diffie-Hellman shared secret between *pkR* and *skE*
3. Serialize *pkE* and *pkR* as *pkE_serialized* and *pkR_serialized*:
    1. For NIST P-curves, the serialization of a point is (0x04 || X || Y), where X and Y are the big-endian coordinates of the point, as specified in RFC 9180 [19].

        Note:

        This is the uncompressed serialization of the point as specified in SEC 1 [20].

    2. For Curve25519 and Curve448, the serialization of a point is the little-endian X coordinate of the point as specified in RFC 9180 [19].
4. Define *kem_context* as the concatenation of (*pkE_serialized* || *pkR_serialized*)
5. Expand *dh* and *kem_context* into a shared secret *shared_secret* using the key's KDF settings in the `ExtractAndExpand` function from RFC 9180 [19]
6. Return *shared_secret*, *pkE_serialized*.

44.4.3 Decapsulation

`TPM2_Decapsulate()` is equivalent to DHKEM's `Decaps` function.

Given:

- Ciphertext *pkE_serialized*
- Private key *skR*

The TPM will:

1. Compute *dh*, the Diffie-Hellman shared secret between *pkE* and *skR*
2. Serialize *pkR* as *pkR_serialized* in the same way as for Encapsulation (Clause 44.4.2).
3. Define *kem_context* as the concatenation of (*pkE_serialized* || *pkR_serialized*)
4. Expand *dh* and *kem_context* into a shared secret *shared_secret* using the key's KDF settings in the `ExtractAndExpand` function from RFC 9180 [19]
5. Return *shared_secret*.

44.5 ECC Parameter Representation

44.5.1 Public Points

An ECC point (a public key or an ECDH shared secret) is represented by an TPMS_ECC_POINT containing a pair of buffers *x* and *y*, padded as in Clause 44.5.3.

The point at infinity is represented by empty *x* and *y* buffers.

For the Edwards and Montgomery curves (Curve25519 and Curve448), *y* has length 0 and *x* is populated (for points other than the point at infinity) as follows:

- For ECDH (X25519 and X448), *x* contains the little-endian X coordinate as specified in RFC 7748 [21].
- For EdDSA and HashEdDSA, *x* contains the compressed point in little-endian as specified in RFC 8032 [17].

For all other curves, *x* and *y* contain the uncompressed big-endian coordinates of the ECC point.

For an ECDH public key and shared secret, the representation described above MUST also be used when input to the KDFe() function.

If the TPM receives an ECC point in the incorrect format (e.g., a compressed point for a NIST P-curve), it returns TPM_RC_ECC_POINT, just as when it encounters any invalid ECC point.

44.5.2 Private Keys

An ECC private key is represented as a TPM2B_ECC_PARAMETER.

For the Edwards and Montgomery curves (Curve25519 and Curve448), the private key is little-endian as specified in RFC 7748 [21].

For all other curves, the private key is big-endian.

44.5.3 Padding

To provide consistent behavior across all TPM implementations this clause describes the padding requirements for ECC parameters.

In ECC points returned by the TPM, the *x* and *y* values, if non-empty, are required to be the size of their associated curve (e.g., 32 bytes for NIST P-256). If necessary, leading bytes of zero MUST be added at the most significant bytes of the *x* and *y* values. (RFC 8032 [17] and RFC 7748 [21] already require Edwards and Montgomery curve points to be represented as full-length buffers.)

An TPMS_ECC_POINT representing an actual ECC point (i.e., not a nonce in a key creation template) MUST be padded as described above when it is provided to the TPM.

To ensure interoperability, any TPMS_ECC_POINT that is part of a TPM2B_PUBLIC (even if it does not represent an actual curve point) SHOULD be padded as described above when it is provided to the TPM.

Note:

The reason for requiring padding on the input of an ECC point is that an ECC point makes up the *unique* field of an ECC key. The Name of the key is computed by hashing the key's public area as input to the TPM. If the ECC point in the *unique* field were not properly padded, the Name would not be consistent.

Internally, an intermediate ECC point, such as the result of an ECC point multiplication or the public key of an ephemeral ECC key, is required to be padded if used as input to the KDFe() function.

Note:

This ensures that the secret derived from KDFe(), which is used e.g., as salt in `TPM2_StartAuthSession()`, or as protection seed of the outer wrapper in `TPM2_Duplicate()`, is the same on different implementations.

Version 1.16 of this specification also required the ECC private key in *duplicate* of `TPM2_Import()` to be padded.

When the ECC parameters are returned by the command `TPM2_ECC_Parameters()`, the numeric values will be as specified in the TCG Algorithm registry [22]. However, the size of values, other than the *x* and *y* of the generator point, may not be the same as the registry because values may or may not be zero padded.

44.5.4 Differences for EdDSA and X25519/X448

Table 44 provides a summary of the differences between the ECC parameter encoding for EdDSA and X25519/X448 compared to (conventional) ECDSA and ECDH (used with Weierstrass curves such as NIST-P).

Table 44: ECC Parameter Encoding Overview

ECC Parameter	Encoding for ECDSA, ECDH (with Weierstrass curves)	Encoding for EdDSA (with Edwards curves)	Encoding for X25519, X448 (with Montgomery curves)
Public key	big-endian uncompressed point	little-endian compressed point (contained in *x* buffer)	little-endian x-coordinate (contained in *x* buffer)
Private key	big-endian	little-endian	little-endian
Signature	big-endian (contained in structure with two sized buffers along with hash, see `TPMS_SIGNATURE_ECC`)	little-endian (contained in one sized buffer, see `TPM2B_SIGNATURE_EDDSA`)	N/A
ECDH shared secret	big-endian uncompressed point	N/A	little-endian x-coordinate (contained in *x* buffer)

44.6 ECC Key Generation

For an ECC key, the method of FIPS 186-5, Annex A.2.1 *Key Pair Generation Using Extra Random Bits* [11] is used. The caller provides a random number generation function according to the type of key being generated (see Clause 24.6.2, Clause 24.6.3, and Clause 25.4) and that function is called when random bits are required by the generation process. The key generation process is:

1. obtain a random value *c* of **length**(*n*) + 64 bits where *n* is the order of the curve
2. set $d := (c \bmod (n - 1)) + 1$
3. compute $Q := (x_Q, y_Q) := [d]G$
4. return *d* and *Q*

44.7 ECC Labeled KEM

44.7.1 Overview

The ECC implementation of the Labeled KEM (Clause 8.4.5.2) is based on the “(Cofactor) One-Pass Diffie-Hellman, C(1e, 1s, ECC CDH)” scheme from SP 800-56A [6] and depicted in Figure 23.

Note:

Unlike in the RSA Labeled KEM, the key's *scheme* is ignored for the ECC Labeled KEM. The hash algorithm used for KDFe is always the key's *nameAlg*, even if the key's *scheme* is not TPM_ALG_NULL.

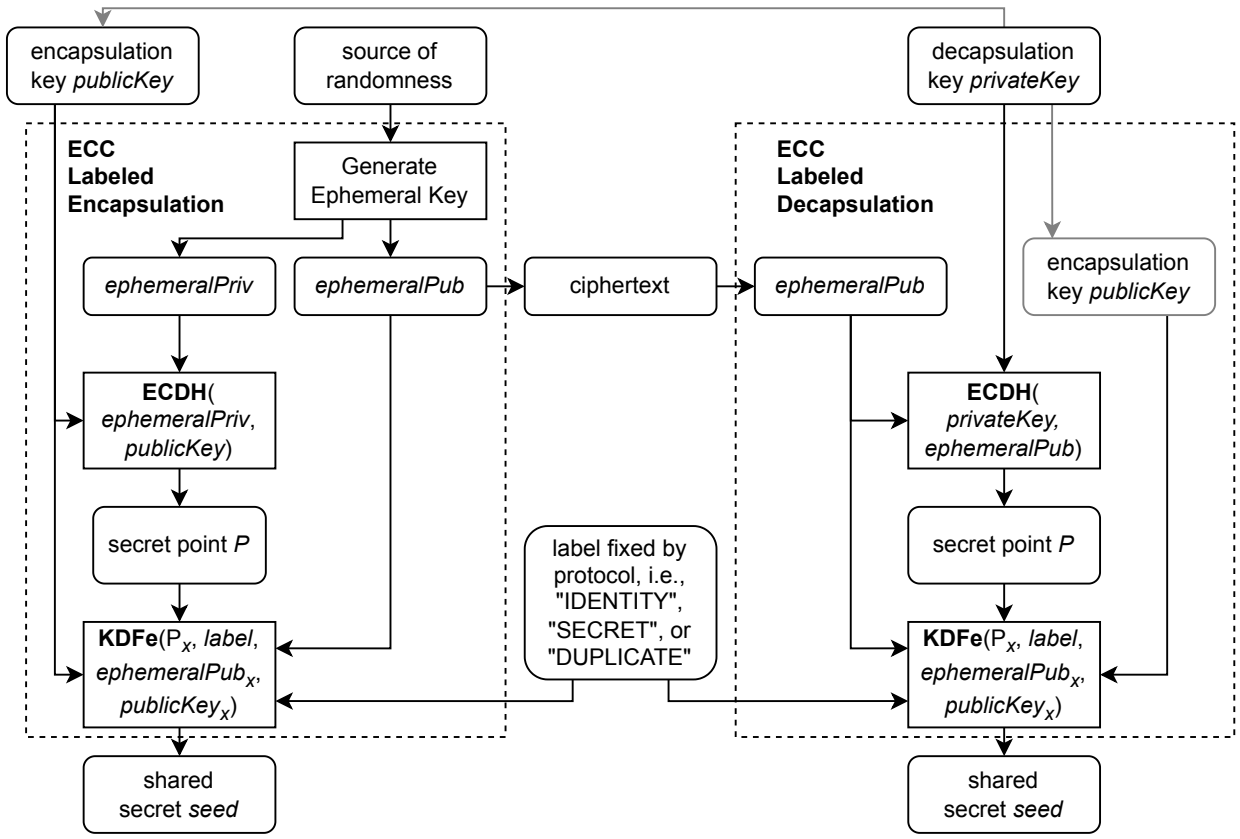


Figure 23: ECC Labeled KEM

Using the notation of SP 800-56A, the initiator generates an ephemeral key pair ($d_{e,U}$, $Q_{e,U}$) from the curve parameters. The public point of the ephemeral key($Q_{e,U}$) is used by the recipient to recover the shared secret.

The initiator uses the private portion of the ephemeral key ($d_{e,U}$) and the public portion ($Q_{s,V}$) of an ECC key of the recipient and computes the point P $:=$ h [$d_{e,U}$]$Q_{s,V}$. Then it will set $Z := x_P$ where x_P is the x-coordinate of *P*.

The recipient may compute $P := h\,[d_{s,V}]Q_{e,U}$ and $Z := x_P$.

The Z value is used in ***KDFe*** to generate a value for *seed* that is appropriate for the use of the seed. The seed will be the size of the digest produced by the *hashAlg* used in the KDF. Seed is computed by:

$$seed := \mathbf{KDFe}(hashAlg,\ Z,\ label,\ PartyUInfo,\ PartyVInfo,\ bits) \tag{63}$$

where

$hashAlg$	is the nameAlg of the recipient key
Z	is the x coordinate (x_P) of the product (P) of a public point and a private key ($P := h[d]Q$)

(continued on next page)

(continued from previous page)

label	is an application-dependent value
PartyUInfo	is the x-coordinate of the secret exchange value ($Q_{e,U}$)
PartyVInfo	is the x-coordinate of a public key ($Q_{s,V}$)
bits	is the number of bits in the digest of hashAlg

44.8 ECC Primitive Operations

44.8.1 Introduction

When ECC is implemented on a TPM, it may provide these additional commands to support cryptographic operations with unrestricted ECC keys.

44.8.2 TPM2_ECDH_KeyGen()

`TPM2_ECDH_KeyGen()` produces an ephemeral key pair. It multiplies the private ephemeral key with the public point of a loaded TPM key to produce the Diffie-Hellman shared secret.

This function can be performed by software as the public key and parameters are known. The function would be provided by the TPM as a service.

Since the operation can be performed by software, no authorization is required to use the public portion of the key and the key attributes are not checked.

44.8.3 TPM2_ECDH_ZGen()

`TPM2_ECDH_ZGen()` performs the ECDH primitive function with one static and one ephemeral key as defined in SP 800-56A, clause 6.2.2 [6]. The input point (Q_e) is multiplied by the private coordinate (d_s) to produce the point $Z = (x_Z, y_Z) := hd_sQ_e$.

Since this operation used the private portion of an ECC key, authorization is required. To prevent inadvertent compromise of a signing key, *sign* and restricted are required to be CLEAR in the referenced key.

44.8.4 Two-phase Key Exchange

44.8.4.1 Introduction

Various key exchange protocols use an ephemeral key from each party. For these protocols, each party generates an ephemeral key and that key is sent to the other party along with other information. The other party then uses the key material from the other party along with its own ephemeral key to generate the key-exchange values.

These protocols require two phases. In the first phase, the TPM generates an ephemeral key to be sent to the other party. In the second phase, the TPM combines data from the other party with the ephemeral key generated in the first phase. The protocols require that the ephemeral key generated by the TPM only be used once and be discarded after the key exchange is complete. This property of this key is the same as required for ECDAA.

`TPM2_EC_Ephemeral()` uses the commit mechanism to generate a random value (r) and a public key $P := [r]G$. The value of P is returned to the caller along with the counter value associated with r.

`TPM2_ZGen_2Phase()` is used to complete the second phase of the key exchange. The counter value returned by `TPM2_EC_Ephemeral()` is provided from which the TPM recreates r and regenerates the associated public key. When `TPM2_ZGen_2Phase()` completes successfully, the TPM will “retire” the r value so that it may not be used again.

One of the parameters of TPM2_ZGen_2Phase() is a scheme selector (*inScheme*). This indicates to the TPM which of the supported schemes is to be used. This clause describes two of the allowed schemes. They are the two EC schemes from SP 800-56A [6] that require two ephemeral and two static keys. The schemes are specified in SP 800-56A in "(Cofactor) Full Unified Model, C(2e, 2s, ECC CDH)" and "Full MQV, C(2e, 2s, ECC MQV)". These schemes use the following terms:

$d_{s,A}$	the private part of a TPM-resident ECC key referenced by the *keyA* parameter
$Q_{s,A}$	the public point of the key referenced by *keyA* equal to $[d_{s,A}]G$ with coordinates $(x_{s,A}, y_{s,A})$
$d_{e,A}$	a private ephemeral key generated by the TPM (the value of *r* associated with *counter* parameter)
$Q_{e,A}$	the public ephemeral key associated with *counter* equal to $[d_{e,A}]G$ or $[r]G$ with coordinates $(x_{e,A}, y_{e,A})$
$Q_{s,B}$	the *inQsB* parameter - a point on the curve of *keyA* assumed to be a static public key associated with the other party in the key exchange with coordinates $(x_{s,B}, y_{s,B})$
$Q_{e,B}$	the *inQeB* parameter - a point on the curve of *keyA* assumed to be an ephemeral public key associated with the other party in the key exchange with coordinates $(x_{e,B}, y_{e,B})$

44.8.4.2 Full Unified Model

When this scheme is selected for TPM2_ZGen_2Phase(), the TPM will:

1. set *outZ1* $:= [d_{s,A}]Q_{s,B}$
2. set *outZ2* $:= [d_{e,A}]Q_{e,B}$

> ***Note:***
>
> If *outZ1* or *outZ2* is the point at infinity, then both coordinate values of the point will be Empty Buffers.

44.8.4.3 Full MQV

This scheme uses an associated value function (**avf**()) that is defines as:

Inputs:

$Q = (x, y)$	a public key
n	the modulus of the curve containing *Q*

Process:

1. Set $f := \lceil \frac{\lceil log_2(n) \rceil}{2} \rceil$
2. Set $x' = 2^f + (x \bmod 2^f)$
3. return *x'*

The MQV computation:

4. validate that $Q_{s,B}$ and $Q_{e,B}$ are on the curve associated with $d_{s,A}$

5. using *counter*, recover $d_{e,A}$ = *r* as described in Clause 44.2.5
6. set $Q_{e,A} := [d_{e,A}]G$ where *G* is the generator point for the curve of $d_{s,A}$
7. set $t_A := (d_{e,A} + d_{s,A} \cdot \mathbf{avf}(Q_{e,A}))$ (mod *n*)
8. set *outZ1* := $[h \cdot t_A]$ $(Q_{e,B} + [\mathbf{avf}(Q_{e,B})](Q_{s,B}))$

Note:

if *outZ1* is the point at infinity both the coordinate values of *outZ1* will be Empty Buffers

Note:

This protocol may be susceptible to unknown key-share (UKS) attacks.

45 Support for SMx Family of Algorithms

45.1 Introduction

This clause provides additional information for implementation of the SM2, SM3, and SM4 algorithms published by the Standardization Administration of the P.R.C.

45.2 SM2

45.2.1 Introduction

SM2 contains information relating to ECC cryptography and is in five parts.

- GB/T 32918.1-2016 [23]: *General* - "provides necessary basics of mathematics and related cryptographic techniques used in public key cryptographic algorithm SM2 based on elliptic curves." The methods of this part are compatible with the EC methods in other standards and no special considerations are necessary to accommodate this standard.
- GB/T 32918.2-2016 [24]: *Digital Signature Algorithm* - defines the process for generation and verification of a digital signature using the methods described in Part 1. The signing method in this part of the standard requires addition of a new signing scheme and methods. These are described in this clause.
- GB/T 32918.3-2016 [25]: *Key Exchange Protocol* - defines a two-phase key exchange protocol using the methods of Part 1. The method in this part of the SM2 standard is supported by addition of a key exchange command (`TPM2_ZGen_2Phase()`). The algorithm is fully described in Part 3 of this TPM specification.
- GB/T 32918.4-2016 [26]: Public Key Encryption Algorithm - defines an encryption method using single pass EC Diffie-Hellman to exchange a key that is then used to generate a stream cipher.
- GB/T 32918.5-2017 [27]: Parameter definition - defines the parameters for a 256-bit ECC curve.

Note:

SM2 does not specify a specific cryptographic hash algorithm. The TPM does not automatically provide a default; if a hash algorithm is not provided by the caller, the TPM will return an error. However, per section 5.4.2 of [24], [25], and [26], this should usually be SM3 (`TPM_ALG_SM3_256`), see Clause 45.3.

45.2.2 SM2 Digital Signature Algorithm

45.2.2.1 SM2 Sign

The SM2 signing scheme has an algorithm ID of `TPM_ALG_SM2`. If the TPM implements this algorithm, then any structure that allows an ECC-based signing scheme may use this algorithm ID.

The SM2 Digital Signature Algorithm is a message signing scheme. The first step (specified in 5.5 of [24]) is to compute a digest Z_A by:

$$Z_A := \mathbf{H}(ENTL_A \parallel ID_A \parallel a \parallel b \parallel x_G \parallel y_G \parallel x_A \parallel y_A) \quad (64)$$

where:

H	the specified cryptographic hash function
$ENTL_A$	is two octets containing the length of ID_A in octets

(continued on next page)

(continued from previous page)

ID_A	is a provided octet string containing information that can identify an entity's identity unambiguously (see ISO/IEC 15946-3 3.9) [18]
a	is the coefficient for the linear term of the equation for the curve of the signing key
b	is the coefficient for the constant term of the equation for the curve of the signing key
x_G	is the *x* coordinate of the generator point for the curve of the signing key
y_G	is the *y* coordinate of the generator point for the curve of the signing key
x_A	is the *x* coordinate of the public key of the signing key
y_A	is the *y* coordinate of the public key of the signing key

The next step (specified by A1-A2 in 6.1 of [24]) is to compute a digest *e* from the message data *M* by:

$$e := \mathbf{H}(Z_A \parallel M) \tag{65}$$

where:

H	the specified cryptographic hash function
Z_A	the digest computed in Equation 64
M	the message to sign

The final step (specified by A3-A7 in 6.1 of [24]) is to sign the digest *e* using the private key:

1. set *k* to a random value such that $1 \leq k \leq n\text{-}1$
2. compute $P_1 := (x_1, y_1) := [k]G$
3. compute $r := e + x_1 \pmod{n}$
4. if *r* equals 0 or (*r* + *k*) equals *n*, go to (1)
5. compute $s := ((1 + d_s)^{-1} \cdot (k - r \cdot d_s)) \pmod{n}$
6. if *s* equals 0, go to (1)
7. the signature is the tuple (*r, s*)

where:

e	the digest computed in Equation 65
d_s	a private ECC key
n	the modulus of the curve for d_s

45.2.2.2 SM2 Signature Verification

For verification, first compute Z_A from the public key and provided ID_A as in Equation 64. Next, compute *e* from Z_A and the message *M* as in Equation 65. Finally, we validate the signature (*r, s*):

1. verify that *r* and *s* are in the inclusive interval 1 to (*n* - 1)
2. compute *t* := (*r* + *s*) (mod *n*)
3. verify that 0 < *t*. If not, the signature is not valid.
4. compute (*x*, *y*) := [*s*]*G* + [*t*]*P*
5. compute *r'* := (*e* + *x*) (mod *n*)
6. verify that *r'* = *r*. If not, the signature is not valid.

where:

e	the digest computed in Equation 65
(*r, s*)	the signature tuple
P	a public ECC key
G	the generator point for the curve of *P*
n	the modulus of the curve for d_s

45.2.2.3 Sequenced Signing Commands

The TPM exposes both sequenced singing commands (see Clause 29.4.5) and digest signing commands (see Clause 45.2.2.4). When using TPM_ALG_SM2 with the sequenced signing commands, the entire process of SM2 Signing is performed.

Specifically:

1. TPM2_SignVerifySequenceStart() is called, passing ID_A as the optional *context* parameter. The TPM then computes Z_A according to Equation 64.
2. TPM2_SequenceUpdate() is called zero or more times to provide chunks of the message data *M*. The TPM updates the corresponding hash context for Equation 65.
3. TPM2_SignSequenceComplete() or TPM2_VerifySequenceComplete() is called with remainder of the message data *M*. Then, the TPM finishes computing *e* according to Equation 65. Finally, *e* is used to compute or verify the signature, as appropriate.

Note:

The sequenced signing commands in Clause 29.4.5 were added in version 185.

45.2.2.4 Digest Signing Commands

When using TPM_ALG_SM2 with the digest signing commands (TPM2_Sign() and TPM2_SignDigest()) and digest signature verification commands (TPM2_VerifySignature() and TPM2_VerifyDigestSignature()), the caller only provides the TPM with the digest *e* computed in Equation 65 instead of the entire message *M*. The caller is responsible for computing Z_A (see Equation 64), prepending it to the message *M*, then hashing this modified message to get digest *e* to pass to the TPM.

TPM2_Sign() or TPM2_SignDigest() may be used with the TPM_ALG_SM2 scheme identifier to create a full SM2-compatible signature. To do an SM2 signature, the application would compute Z_A, and then use the resulting digest as the first data in one of the TPM hash commands (which could be a TPM2_HashSequenceStart()); with the Z_A value followed by the message data (*M*). The digest of **H**(Z_A ‖ *M*) would then be used as the *digest* parameter for TPM2_Sign() or TPM2_SignDigest().

Note:

Since Z_A is a constant value for a key, an application might choose to keep Z_A as part of the metadata for the key so that it would not need to be recomputed each time the key is used for an SM2 signature.

As the provided ID_A is only used in computing the digest *e*, it cannot be passed as the *context* parameter for `TPM2_SignDigest()` or `TPM2_VerifyDigestSignature()`. This *context* parameter must be empty when using `TPM_ALG_SM2` with these commands.

Note:

The commands `TPM2_SignDigest()` and `TPM2_VerifyDigestSignature()` were added in version 185. The `TPM2_Sign()` and `TPM2_VerifySignature()` commands were deprecated in version 185.

45.2.2.5 Implementation Issues

When the TPM creates signatures for attestation (e.g. the *signature* response parameter for `TPM2_Certify()`), the digest *e* is not computed as described in Equation 65. Instead, the digest is simply the hash of the message *M*:

$$e := \mathbf{H}(M) \tag{66}$$

One consequence of this is that attestation operations will not create a signature that is, in all details, compliant with SM2 Part 2. Instead, the attestation signatures will be TPM specific. The reason that attestations do not sign using the full scheme are:

- There is no infrastructure for the distribution of ID_A values
- Requiring the use of an ID_A value in a signature could allow correlation of a user and void the privacy assurances of the attestation
- Ensuring that an external digest does not match a valid attestation becomes intractable.

The reason that the attestation problem becomes intractable is that, using Z_A with an attestation means that the first bytes that were used to form the digest of the signed value (*e*) would vary with each key used to sign. An attacker could perform a hash using the key specific values followed by message data that has all the characteristics of an attestation. The TPM will not be able to discern the transition from Z_A data to the false attestation data.

To prevent this kind of attack without adding excessive complexity to the TPM, the attestation is done without including Z_A. Since the use of Z_A does not improve the security of the SM2 signature, leaving it out does not compromise the value of the SM2 signing process for attestations. Also, since an attestation only has meaning in the context of a TPM, having TPM-specific verification of a signature over an attestation block should not create an issue.

45.2.3 SM2 Key Exchange

45.2.3.1 Introduction

The key exchange algorithm GB/T 32918.3-2016 is a two-phase algorithm. It is similar to the scheme described in Clause 44.8.4.3.

Note:

This protocol may be susceptible to unknown key-share (UKS) attacks.

This SM2 key exchange computations use an associated value function (**avfSM2**()) that is similar to the function defined in SP 800-56A [6] with the only differencing being that the result is one bit less than the value defined in SP 800-56A. The **avfSM2**() function is:

Inputs:

Q = (x, y)	a public key
n	the modulus of the curve containing *Q*

Process:

1. set $f := \lceil \frac{\lceil log_2(n) \rceil}{2} \rceil - 1$
2. set *x'* := 2^f + (*x* (mod 2^f))
3. return *x'*

Note:

This function is similar to the function in SP 800-56A [6] except that, in the formulation in GB/T 32918.3-2016 as shown in a) above, the value of *f* is one less than the equivalent in SP 800-56A.

45.2.3.2 SM2 Key Exchange Protocol

The key exchange protocol is between two entities, A and B. The TPM performs computations as party A. Since the protocol is symmetric, both party A and party B may be TPMs and they will both perform the same operations, using the values from the other TPM as party B values.

The caller must use `TPM2_EC_Ephemeral()` to have the TPM generate a single-use ephemeral key. The ephemeral public key is sent to the other party as $Q_{e,B}$.

The inputs to the key exchange computation are:

counter	the counter parameter from `TPM2_Commit()`
$Q_{s,B}$	a public EC key from party B; usually, the public part of a static key
$Q_{e,B}$	a public EC key; usually, the public part of an ephemeral key
$d_{s,A}$	a private EC key (an unrestricted decryption key)

The protocol:

1. validate that $Q_{s,B}$ and $Q_{e,B}$ are on the curve associated with $d_{s,A}$
2. using *counter*, recover *r* as described in Clause 44.2.5
3. set $Q_{e,A} := [r]G$ where *G* is the generator point for the curve of $d_{s,A}$
4. set $t_A := (d_{s,A} + d_{e,A} \cdot \textbf{avfSM2}(Q_{e,A}))$ (mod *n*)
5. set $Z := [h \cdot t_A](Q_{s,B} + [\textbf{avfSM2}(Q_{e,B})](Q_{e,B}))$
6. if *Z* is the point at infinity, return failure

45.2.4 SM2 Encryption and Decryption

45.2.4.1 Introduction

These clauses describe the encryption and decryption process of GB/T 32918.4-2016 as implemented in the TPM.

Note:

The results of the encryption contains an encoded ECC point. This specification only supports the uncompressed version of a point.

These algorithms may be use with any ECC key with a *restricted* attribute that is CLEAR.

45.2.4.2 Encryption

The encryption process is based on ECDH using the exchanged secret key as a generator for a bit stream. That bit stream is exclusive ORed (XORed) with the clear text *M* to produce the cipher text *C.* The inputs to the encryption process are:

P_B	a public EC key for the entity receiving the encrypted data
G	the generator point for the curve of P_B
n	the modulus of the curve of P_B
M	the bit string to be encrypted
KDF	a KDF algorithm such as KDF1 or KDF2
Hash	the hash algorithm used in the KDF

The process is:

1. set *k* to a random value such that $1 \leq k \leq n - 1$
2. compute $P_1 := (x_1, y_1) := [k]G$
3. compute $P_2 := (x_2, y_2) := [k]P_B$
4. if P_1 or P_2 is the point at infinity, go to 1.
5. set $C_1 := x_1 \parallel y_1$

Note:

C_1 is the uncompressed form of an ECC point.

6. create a bit string *t* of length *len(M)* using the selected KDF and hash algorithm with $x_2 \parallel y_2$ as the key value

Note:

Currently, only KDF2 (TPM_ALG_KDF2) is supported.

7. set $C_2 := M \oplus t$
8. set $C_3 := \mathbf{H}(x_2 \parallel M \parallel y_2)$
9. return C_1, C_2, and C_3

45.2.4.3 Decryption

The inputs to the decryption are:

C_1	an ephemeral key
C_2	a block encrypted as described in Clause 45.2.4.2
C	an encrypted block as described in Clause 45.2.4.2
d_B	a private static key
KDF	a KDF algorithm such as KDF1
Hash	the hash algorithm used in the KDF

The process is:

1. extract P_1 = x_1, y_1 from C_1
2. compute $P_2 := (x_2, y_2) := [d_B]P_1$
3. create a bit string t of length len(M) using the selected KDF and hash algorithm with $x_2 \parallel y_2$ as the key value
4. set $M := C_2 \oplus t$
5. set $V := \mathbf{H}(x_2 \parallel M \parallel y_2)$
6. if V is not equal to C_3, return error
7. return M

45.3 SM3

SM3 (defined in ISO/IEC 10118-3:2018 [28]) is a hash algorithm that uses a 512-bit block and produces a digest of 256 bits.

If the TPM implements this algorithm, then the algorithm ID for SM3 (`TPM_ALG_SM3_256`) may be used in any structure that allows a hash algorithm.

45.4 SM4

SM4 (defined in GB/T 32907-2016 [29]) is a symmetric block cipher with a key and block size of 128 bits.

If the TPM implements this algorithm, then the algorithm ID for SM4 (`TPM_ALG_SM4`) may be used in any structure that allows a symmetric block cipher.

46 ML-DSA

46.1 Introduction

ML-DSA [30] is used for digital signatures.

46.2 ML-DSA Key Representation and Generation

The TPM 2.0 Library represents an ML-DSA private key as the 32-byte seed ξ (xi). While implementations might choose to represent ML-DSA keys internally in expanded form (e.g., for performance reasons), the TPM only supports importing/exporting ML-DSA keys as seeds.

ML-DSA keys are generated within the TPM by sampling the seed ξ from the entropy source selected in Clause 24.6.

ML-DSA verification (public) keys are represented in the format specified by Algorithm 1 (`ML-DSA.KeyGen()`) of [30].

46.3 ML-DSA Cryptographic Primitives

ML-DSA keys can be used with `TPM2_SignSequenceStart()`, `TPM2_SignSequenceComplete()`, `TPM2_VerifySequenceStart()`, and `TPM2_VerifySequenceComplete()`. When *allowExternalMu* is TRUE, they can also be used with `TPM2_SignDigest()` and `TPM2_VerifyDigestSignature()`, where the *digest* value will be interpreted as the 64-byte external Mu (μ) value as computed in Algorithm 7 (`ML-DSA.Sign_internal`), Line 6 of FIPS 204 [30].

HashML-DSA keys can be used with `TPM2_SignSequenceStart()`, `TPM2_SignSequenceComplete()`, `TPM2_VerifySequenceStart()`, `TPM2_VerifySequenceComplete()`, `TPM2_SignDigest()` and `TPM2_VerifyDigestSignature()`.

47 ML-KEM

47.1 Introduction

ML-KEM [31] is used for key encapsulation.

47.2 ML-KEM Key Representation and Generation

The TPM 2.0 Library represents an ML-KEM private key as the 64-byte seed $(d \parallel z)$. While implementations might choose to represent ML-KEM keys internally in expanded form (e.g., for performance reasons), the TPM only supports importing/exporting ML-KEM keys as seeds.

ML-KEM keys are generated within the TPM by sampling the seed $(d \parallel z)$ from the entropy source selected in Clause 24.6.

ML-KEM encapsulation (public) keys are represented in the format specified by Algorithm 19 (`ML-KEM.KeyGen()`) of [31].

47.3 ML-KEM Cryptographic Primitives

ML-KEM keys can be used with the `TPM2_Encapsulate()` and `TPM2_Decapsulate()` commands in Part 3.

47.4 ML-KEM Labeled KEM

The Labeled KEM for ML-KEM is constructed on top of ML-KEM by adding a **KDFa** step after the encapsulation or decapsulation. This KDF produces a secret that has the proper size, bound to a particular instance of the particular Restricted Decryption protocol (from Clause 8.4.5.2).

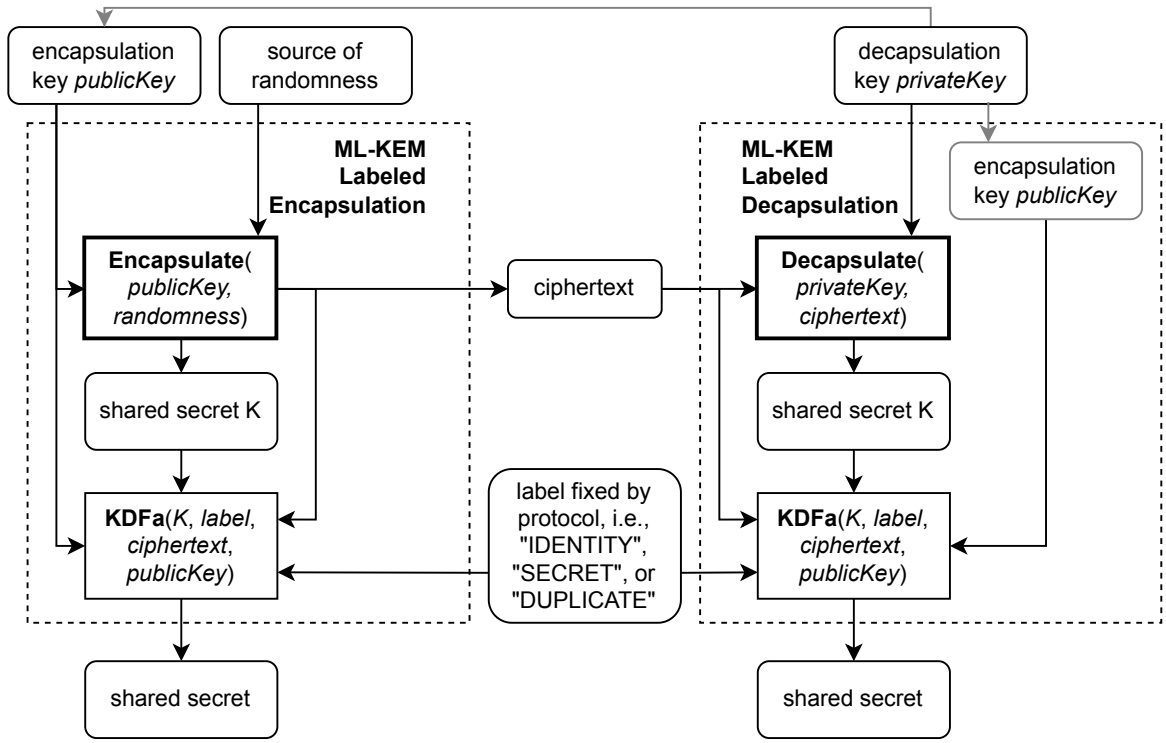


Figure 24: ML-KEM Labeled KEM

The initiator performs ML-KEM encapsulation with the recipient's encapsulation (public) key to compute the pair (shared secret *K*, *ciphertext*). The recipient uses its decapsulation (secret) key to decapsulate the shared secret

K from *ciphertext*.

Equation 67 describes how *seed* is computed using *KDFa*:

$$seed := \mathbf{KDFa}(hashAlg, K, label, ciphertext, publicKey, bits) \quad (67)$$

where

hashAlg	is the nameAlg of the recipient key
K	is the 64-byte shared secret returned from ML-KEM encapsulation or decapsulation
label	is an application-dependent value
ciphertext	is the ciphertext (`TPM2B_KEM_CIPHERTEXT` contents)
publicKey	is the encapsulation key (`TPM2B_PUBLIC_KEY_MLKEM` contents)
bits	is the number of bits in the digest of hashAlg

Note:

As discussed in Clause 8.4.10.2, only the contents of the buffers (and not the sizes) are used in **KDFa**.

References

[1] "Information security - Message authentication codes (MACs) - Part 2: Mechanisms using a dedicated hash-function." ISO/IEC, Jun. 2021. Available: https://www.iso.org/standard/75296.html

[2] G. Alagic *et al.*, "SP 800-227: Recommendations for Key-Encapsulation Mechanisms." Jan. 2025. Available: https://csrc.nist.gov/pubs/sp/800/227/ipd

[3] "Information technology - Security techniques - Modes of operation for an n-bit block cipher." ISO/IEC, Jul. 2017. Available: https://www.iso.org/standard/64575.html

[4] L. Chen, "SP 800-108: Recommendation for Key Derivation Using Pseudorandom Functions." National Institute of Standards and Technology, Aug. 2022. Available: http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf

[5] E. Barker, L. Chen, and R. Davis, "SP 800-56c: Recommendation for Key-Derivation Methods in Key-Establishment Schemes." National Institute of Standards and Technology, Aug. 2020. Available: https://doi.org/10.6028/NIST.SP.800-56Cr2

[6] E. Barker, L. Chen, A. Roginsky, A. Vassilev, and R. Davis, "SP 800-56a: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography." National Institute of Standards and Technology, Apr. 2018. Available: https://doi.org/10.6028/NIST.SP.800-56Ar3

[7] E. Barker and J. Kelsey, "NIST SP800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators." National Institute of Standards and Technology, Jun. 2015. Available: https://csrc.nist.gov/pubs/sp/800/90/a/r1/final

[8] "TCG Protection Profile for PC Client Specific TPM 2.0." Trusted Computing Group, Sep. 2021. Available: https://trustedcomputinggroup.org/resource/pc-client-protection-profile-for-tpm-2-0/

[9] "TCG FIPS 140-2 Guidance for TPM 2.0." Trusted Computing Group, Feb. 2017. Available: https://trustedcomputinggroup.org/resource/tcg-fips-140-2-guidance-for-tpm-2-0/

[10] "TCG FIPS 140-3 Guidance for TPM 2.0." Trusted Computing Group, Jun. 2024. Available: https://trustedcomputinggroup.org/resource/tcg-fips-140-3-guidance-for-tpm-2-0/

[11] "FIPS 186-5: Digital Signature Standard (DSS)." National Institute of Standards and Technology, Feb. 2023. Available: https://doi.org/10.6028/NIST.FIPS.186-5

[12] "TCG EK Credential Profile for TPM Family 2.0." Trusted Computing Group, Dec. 2024. Available: https://trustedcomputinggroup.org/resource/http-trustedcomputinggroup-org-wp-content-uploads-tcg-ek-credential-profile-v-2-5-r2_published-pdf/

[13] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, "RFC 8017 - PKCS #1: RSA Cryptography Specifications Version 2.2." RFC Editor, Nov. 2016. Available: https://www.rfc-editor.org/info/rfc8017

[14] "IEEE Standard Specifications for Public-Key Cryptography." IEEE, Aug. 2000. Available: https://standards.ieee.org/ieee/1363/2049/

[15] "FIPS 186-4: Digital Signature Standard (DSS)." National Institute of Standards and Technology, Jul. 2013. Available: https://doi.org/10.6028/NIST.FIPS.186-4

[16] "IT Security techniques - Digital signatures with appendix - Part 3: Discrete logarithm based mechanisms." ISO/IEC, Nov. 2018. Available: https://www.iso.org/standard/76382.html

[17] S. Josefsson and I. Liusvaara, "RFC 8032: Edwards-Curve Digital Signature Algorithm (EdDSA)." RFC Editor, Jan. 2017. Available: https://www.rfc-editor.org/info/rfc8032

[18] "Information technology - Security techniques - Cryptographic techniques based on elliptic curves." ISO/IEC, Jul. 2016. Available: https://www.iso.org/standard/65480.html

[19] R. Barnes, K. Bhargavan, B. Lipp, and C. Wood, "RFC 9180: Hybrid Public Key Encryption." RFC Editor, Feb. 2022. Available: https://www.rfc-editor.org/info/rfc9180

[20] "SEC 1: Elliptic Curve Cryptography." Standards for Efficient Cryptography Group, May 2009. Available: https://secg.org/sec1-v2.pdf

[21] A. Langley, M. Hamburg, and S. Turner, "RFC 7748: Elliptic Curves for Security." RFC Editor, Jan. 2016. Available: https://www.rfc-editor.org/info/rfc7748

[22] “TCG Algorithm Registry.” Trusted Computing Group, Feb. 2025. Available: https://trustedcomputinggroup.org/resource/tcg-algorithm-registry/
[23] “Information Security Technology - Public Key Cryptographic Algorithm SM2 Based On Elliptic Curves - Part 1: General.” Standardization Administration of the PRC, 2016. Available: https://webstore.ansi.org/standards/spc/gb329182016
[24] “Information Security Technology - Public Key Cryptographic Algorithm SM2 Based On Elliptic Curves - Part 2: Digital Signature Algorithm.” Standardization Administration of the PRC, 2016. Available: https://webstore.ansi.org/standards/spc/gb329182016-1676169
[25] “Information Security Technology - Public Key Cryptographic Algorithm SM2 Based On Elliptic Curves - Part 3: Key exchange protocol.” Standardization Administration of the PRC, 2016. Available: https://webstore.ansi.org/standards/spc/gb329182016-1676171
[26] “Information Security Technology - Public Key Cryptographic Algorithm SM2 Based On Elliptic Curves - Part 4: Public key encryption algorithm.” Standardization Administration of the PRC, 2016. Available: https://webstore.ansi.org/standards/spc/gb329182016-1676170
[27] “Information Security Technology - Public Key Cryptographic Algorithm SM2 Based On Elliptic Curves - Part 5: Parameter definition.” Standardization Administration of the PRC, 2017. Available: https://webstore.ansi.org/standards/spc/gb329182017
[28] “IT Security techniques - Hash-functions - Part 3: Dedicated hash-functions.” ISO/IEC, Oct. 2018. Available: https://www.iso.org/standard/67116.html
[29] “SM4 Block Cipher Algorithm.” Standardization Administration of the PRC, 2016.
[30] “Module-Lattice-Based Digital Signature Standard.” National Institute of Standards and Technology, Aug. 2024. Available: https://doi.org/10.6028/NIST.FIPS.204
[31] “Module-Lattice-Based Key-Encapsulation Mechanism Standard.” National Institute of Standards and Technology, Aug. 2024. Available: https://doi.org/10.6028/NIST.FIPS.203

