TRUSTED® COMPUTING GROUP

REFERENCE

Trusted Platform Module 2.0 Library Part 0: Introduction

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.

ACKNOWLEDGEMENTS

The writing of a specification, particularly a security specification, takes many hours for both development and review. The TPM Workgroup would like to acknowledge the contribution of those individuals (listed below) and the companies who allowed them to volunteer their time to the development of this specification.

The TPM Workgroup would like to acknowledge that this specification would not be possible without the efforts of previous contributors, as recognized in previous versions.

The TPM Workgroup would like to acknowledge the special contribution of David Wooten in the development of the TPM 2.0 architecture. We also acknowledge the generosity of Microsoft in contributing the original TPM reference code, and the following contributors to the reference code for 185: Brad Litterell, Chris Fenner, Dennis Mattoon, Ga-Wai Chin, Heike Gilg, Ivan Velevski, Joe Richey, Ken Goldman, Liran Perez, Lucas Brunetta Pozza, Nicholas Noonarby, Shai Sarfati, Shenghu Liu, Stefan Berger, Todd Johnson, Yanai Moyal, Zecharye Galitzky.

Special thanks are due to Brad Litterell, Chris Fenner, Ken Goldman, David Challener, David Wooten, David Grawrock, Julian Hammersley, Graeme Proudler, and Ari Singer who served as Chair of the TPM Working Group at different times during the development of this specification.

The TPM Workgroup would also like to give special thanks to David Grawrock, David Wooten, and Ken Goldman, who were the editors at different times during the development of this specification.

Contributors

Organization	Contributors
Advanced Micro Devices, Inc	Jen Ye, Parth Joshi, Venkata Krishna Reddy Sagili
Ampere Computing LLC	Sachhidh Kannan
Analogix Semiconductor	Greg Stewart
ARM Ltd	Dong Wei, Prachotan Reddy Bathi, Stuart Yoder
Broadcom	Ivan Velevski, Jesse Pool
BSI	Dominik Klein
Cisco Systems	Altanai Bisht, Bill Sulzen, Charlie Hsu, Doug Ambrisko, Goutam Madhukeshwar Hegde, Scott Fluhrer
Crypto Quantique Ltd	Quentin Grouillet, Ranga Desikachari, Shahram Mossayebi
Dell Inc	Amy Nelson, Antonio Fontes, Travis Gilbert
Fraunhofer SIT	Henk Birkholz
Gapfruit AG	Stefan Thöni
Google Inc	Chris Fenner, David Bird, Dionna Glaze, Jeff Andersen, Jessica McClintock, Joe Richey, Nicholas Noonarby, Ronald Aigner, Vadim Sukhomlinov
Hewlett Packard Enterprise	Ben Lytle, Dhilip Kumar Baroda Ramaswamy, Kevin Micciche, Robert Elliott, Subramanian Swaminathan
HP Inc	Adrian Shaw, Duane Gatlin, Joshua Schiffman
Huawei Technologies Co. Ltd	Bent Jepsen, Silviu Vlasceanu, Thomas Deleuran, Xiaoyu Ge, Zhang Li
IBM	Arthur Savage, Dimitrios Pendarakis, Ken Goldman

(continued on next page)

(continued from previous page)

Organization	Contributors
Infineon Technologies	Antonio Javier Cabrera Gutierrez, Daniel Flaschka, Ga-Wai Chin, Josef Kohn, Rahul Venkatram, Yoshiaki Ujino
Integrity Security Services, LLC	Hank Cohen, Martin Bergenwall
Intel Corporation	Liran Perez, Magid Latif, Masahide Kakeda, Thomas Bowen
Kioxia Corporation	James Borden
Lenovo Inc	Collin Parker, Elizabeth Stremlau, Kazuo Shiba, Kinjo Raiya
MediaTek Inc	Ahmad Khalifeh, Thomas Zeng
Microsoft	Alan Ludwig, Anaya Samuel-Hartfield, Austin Fisher, Brad Litterell, Dennis Mattoon, Derek Adam, Lucas Brunetta Pozza, Tien Thanh Dang, Yucong Tao
NEC Corporation	Toru Tomita
NetApp	Kan Itani
NSING Technologies Inc	Vision Liu, Xin Liu
Nuvoton Technology Corporation	Dan Morav, Dima Baskin, Galit Heller, Yaakov Hayun
NVIDIA Corporation	Ludovic Jacquin, Steven Bellock, Yazan Siam
NXP Semiconductors	Melissa Azouaoui, Tobias Schneider
Oracle Corporation	Deirdre Connolly, Monty Wiseman
Phoenix Technologies	Dick Wilkins
SEALSQ	Alain Isnel, Guillaume Aymard, Jean Fioretti, Patrick Debaenst, Raphaël Joud, Steve Clark
SecEdge Inc	Rob Komar
STMicroelectronics	Charly Villette, Fabien Arrivé, Gaëlle Grignard, Joe Pilozzi, Kyle Palmer, Laurent Charpentier, Xavier Boussin, Yves Magnaud
SuperMicro	Calbert Lo
Technical University of Denmark	Thanassis Giannetsos
Texas Instruments	Greg North
Toyota Motor Corporation	Eoin Carroll, Gongyuan Zhuang, Scott Ludwin
United States Government	John Kucera, Mike Boyle, Padma Krishnaswamy, Patrick Gallo, Zachary Blum
Invited Experts	David Challener, David Grawrock, Graeme Proudler, Kuniyasu Suzaki, Liqun Chen

Contents

List of Figures

List of Tables

1 Changes from TPM 1.2

1.1 Overview

The TPM 2.0 specification introduces these additional features:

- Definition of an interface that allows variability of underlying cryptographic algorithms - TPM 1.2 is constrained by its data structures to using RSA and SHA1. The TPM 2.0 structure and interface defines support for a wide range of hash and asymmetric algorithms along with limited support for use of various block, symmetric ciphers. Of particular note is the addition of support for the elliptic curve (ECC) family of asymmetric algorithms.
- Unification of authorization methods - TPM 1.2 has different schemes to authorize the use, delegated use, and migration of objects. This 2.0 specification provides a uniform framework for using authorization capabilities, so they can be combined in unique ways to provide more flexibility.
- Expansion of authorization methods - TPM 2.0 allows authorization with clear-text passwords and Hash Message Authentication Code (HMAC). It also allows construction of an arbitrarily complex authorization policy for an object using multiple authorization qualifiers.
- Dedicated BIOS support - TPM 2.0 adds a Storage hierarchy controlled by platform firmware, letting the OEM benefit from the cryptographic capabilities of the TPM regardless of the support provided to the OS.
- Simplified control model - TPM 2.0 needs no special provisioning process to be useful to applications. Although objects on which the TPM operates can have limitations, all commands are available all the time. This lets application developers rely on TPM capabilities being available whenever a TPM is present.

A TPM compatible with this specification need not be compatible with previous TPM specifications.

This specification defines the operations a TPM performs and the structures used for communication between the TPM and the host system. It does not define an electrical interface to the TPM, nor does it specify which subset of TPM 2.0 commands and resources are required for a specific platform. Please refer to platform-specific TPM specifications for this information.

1.2 Tag Values

- The command or response *tag* indicates whether a command is formatted according to TPM 1.2 or this 2.0 specification. If the latter, the *tag* indicates if any session data is present.

1.3 Handle Values

- The format of NV index handles is different between TPM 1.2 and 2.0.
- The handle range for PCR is defined to be the same as the handle range for PCR in TPM 1.2.

1.4 Algorithm Agility

- TPM 2.0 algorithm identifiers are 16 bits and do not include key sizes. This differs from algorithm identifiers in TPM 1.2, which are 32 bits and which often include the key size in the algorithm identifier (e.g., `TPM_ALG_AES128`).
- TPM 1.2 supports only the RSA algorithm with a limited number of commonly used parameters, while TPM 2.0 supports many more types of algorithms.
- In TPM 1.2, the cryptographic primitives are not exposed for general purpose use. For example, the RSA engine cannot be used for regular decryption or signing. TPM 2.0 provides commands that allow access to the cryptographic primitives of the TPM.

  One assumption in TPM 1.2 is that the host processor usually has much greater performance than the processor used for the TPM so there was no point in having the TPM do something that the host could

do much faster. In addition, TPM 1.2 is a passive device with limited bandwidth. While it is true that the host processor will usually have more capability than the TPM, this will not be true in all cases. In fact, on some systems, the main processor will be able to switch execution environments and perform the TPM operations. In others, the TPM may be built around a cryptographic co-processor that has significantly greater processing capability for cryptographic operations than the host. These higher performance implementations will not be performance-limited by being attached to the system with a low-bandwidth interface. These performance differences mean that exposure of the cryptographic primitives of TPM 2.0 makes more sense that it does in TPM 1.2.

Another reason to make the cryptographic primitives available is that not all software will implement all the algorithms that may be in the TPM. For example, a BIOS may not implement the RSA algorithm but would want to check the RSA signature of some code.

- In TPM 1.2, a PCR Event is hashed using SHA-1 and then the 20-octet digest is extended to a PCR using `TPM_Extend()`. In TPM 2.0, a PCR Event may be sent directly to the TPM to be hashed using the specified algorithm for each PCR bank. For more details, see Part 3: “Integrity Collection (PCR).”

1.5 XOR Obfuscation

- TPM 2.0’s XOR scheme differs from that used in TPM 1.2: it uses a different formulation for input into the hash function.

1.6 Startup Behavior

- The startup behavior defined by this specification is different than TPM 1.2 with respect to Startup(STATE). A TPM 1.2 device will enter Failure Mode if no state is available when the TPM receives Startup(STATE). This is not the case in this specification. It is up to the CRTM to take corrective action if the TPM returns `TPM_RC_VALUE` in response to Startup(STATE).

1.7 Authorization

- TPM 2.0 supports authorizing commands by either using an authorization value or by satisfying an Enhanced Authorization policy. TPM 1.2 only supports using authorization values and PCR state.
- TPM 2.0 defines multiple hierarchies that can be used to authorize commands, while TPM 1.2 only has a single hierarchy (“ownerAuth”). For example, in TPM 1.2, *ownerAuth* authorizes `TPM_OwnerClear()`, but in TPM 2.0, Platform Authorization authorizes `TPM2_Clear()`.
- TPM 2.0 session types differ from TPM 1.2 session types. Part 1, “Unbound and Unsalted Session Key Generation” describes the TPM 2.0 session type that is most similar to OIAP from TPM 1.2. Part 1, “Bound Session Key Generation” describes the TPM 2.0 session type that is most similar to OSAP from TPM 1.2.

1.8 Attestation

- TPM 1.2’s Quote command returns the PCR values along with the digest. In TPM 2.0, because of hash agility, the PCR set could have exceeded the response buffer size, so `TPM2_Quote()` only returns the digest.

1.9 Key Properties

- The *fixedTPM* attribute is the logical inverse of the *migratable* attribute in 1.2. That is, when this attribute is CLEAR, it is the equivalent to a 1.2 object with *migratable* SET.
- Table 1 shows the correspondence between the TPM 1.2 method of identifying key properties and the method in TPM 2.0. In earlier versions of this Specification, it was found in Part 1, “Object Attributes.”

Table 1: TPM 1.2 Correspondence

TPM 1.2 Name	*sign*	*decrypt*	*restricted*	Comments
TPM_KEY_SIGNING	1	0	0	In TPM 1.2, keys had restricted schemes. In this specification, the scheme is defined in the command.
TPM_KEY_STORAGE	0	1	1	The functional properties are nearly the same as TPM 1.2. This key could only be used to protect and unprotect items in a Protection hierarchy.
TPM_KEY_IDENTITY	1	0	1	In TPM 1.2, an Identity key was highly constrained and could not, for example, sign a structure that was not produced by the TPM. In this specification, the restricted signing key can sign (within the limits defined in TPMA_OBJECT.sign) a digest produced by the TPM. This allows, for example, an Attestation Key to sign a PKCS#10 certificate request.
TPM_KEY_AUTHCHANGE	-	-	-	This is not used in this specification and its use was deprecated in TPM 1.2. The functionality is provided by session encryption.
TPM_KEY_BIND	0	1	0	Functionality is roughly equivalent between the TPM 1.2 type and the unrestricted decryption key. The specification would use TPM2_RSA_Decrypt() in place of the TPM 1.2 TPM_Unbind().
TPM_KEY_LEGACY	1	1	0	Use of these keys is only constrained by the key family properties. For example, an ECC key will not perform TPM2_RSA_Decrypt().
TPM_KEY_MIGRATE	0	1	1	A Storage Key can be the object of a re-wrap if the new parent is allowed within the policy for the object. The policy for duplication of the object is always visible in the public area.
Sealed Data	0	0	0	A blob containing user defined data

1.10 Response Codes

- TPM 2.0 response codes are designed to be distinguishable from TPM 1.2 response codes. For more details, see Part 1, “Response Code Details.”
- A TPM 2.0 may also be TPM 1.2 compatible, which impacts the response code if the request contained an unrecognized command tag. For more details, see Part 3, “Response Values.”

2 Change History

2.1 Introduction

This clause describes the major changes between published versions of the TPM 2.0 Specification.

Version 0.96

Initial version.

Version 0.99

Removed the "+" from the handle parameter in TPM2_HMAC_Start().

Changed TPM_RC_BAD_TAG to 0x01e so that its value would match TPM_BADTAG from 1.2

Removed TPM_NV_INDEX entity name space.

Authorization check includes locality.

Added phEnableNV to make NV enable independent of the platform hierarchy enable.

Added TPM2_PolicyNvWritten() to permit a policy based on whether or not NV has been written

Added TPM_PT_NV_BUFFER_MAX, the maximum data size in an NV write.

Return code when an NV hierarchy is disabled is TPM_RC_HANDLE.

TPM2_Shutdown() state may be nullified on any subsequent command.

CTR mode increments the entire IV, not just 32 bits.

TPM2_PolicySecret() cannot have a null authHandle.

Version 1.16

Added Definitions for Endorsement Authorization, Owner Authorization, Platform Authorization.

An error may change TPM state under certain conditions.

A restricted signing key cannot have a scheme of TPM_ALG_NULL.

Added TPMS_EMPTY.

TPM2_Sign(): The signing scheme hash algorithm determines the size of the hash to be signed. However, this may be removed in a future revision.

TPM2_PCR_Allocate() may return an error if the allocation fails.

Handle errors always return TPM_RC_HANDLE, not TPM_RC_HIERARCHY.

TPM_PCR_Allocate does not change allocation for a bank not listed.

For a policy ticket, if expiration is non-negative, a NULL ticket is returned.

Added *lockoutPolicy.*

Added vendor-specific handles.

Added detection of a clock discontinuity to tickets.

Reworked TPM2_Import() description.

Some reworking of H-CRTM, D-RTM.

Some clarification of policy expiration.

Changed PPS, EPS Clear flush resident transient and persistent objects.

Any field upgrade preserves state, not just the standard commands.

Added Part 1 description of vendor-specific authorization values.

Refined description of PCR interaction with H-CRTM, TPM2_Startup(), and locality. _TPM_Hash_Start indicates the start of an H-CRTM sequence, not D-RTM.

A non-authorization session must have at least one of encrypt, decrypt, or audit set

A policy session timeout can only change to a shorter value.

Added defines for ECC curves and removed some redundant values in the Part B annex.

TPM2_Sign() can use a symmetric key.

TPM2_NV_UndefineSpace() fails if TPMA_NV_POLICY_DELETE is set.

TPM2_ContextSave() encrypts just the TPM2B_CONTEXT_SENSITIVE structure.

Part 2 structures removed algorithms and added notation referring to algorithm registry.

HMAC commands cannot be used with a restricted key.

Clarified Auth Role for hierarchies and NV Index.

Added password check to authorization checks.

Indicated that handles returned by the TPM are TPM_HT_TRANSIENT (three places).

FIPS 186-4 note.

Return codes for tag requires vs. actual mismatch.

Version 1.38

Introduced NV PIN indices.

A trial session cannot use encrypt or decrypt.

HMAC is optional when the HMAC key is the Empty Buffer. If present, it must be correct.

CFB uses *sessionValue* in the KDF, not *sessionKey*

FIPS-140 requires NV to be erased when an Index is deleted. NV data must be initialized on a first partial write.

TPM2_Create() for a keyed hash object must have TPM_ALG_NULL if *sign* and *decrypt* are both SET or CLEAR.

For an unrestricted HMAC key, if both the key and parameter have a non-NULL scheme, they must match.

Defined transient object and made the use of object and sequence object more consistent.

Refined the description of an exclusive audit session, the definition of auditReset, and its relationship to the audit attribute.

Explained that the TPM clock must be accurate even if there is no reliable external clock.

Updated the informative algorithm ID table.

TPM2_HMAC() and TPM2_HMAC_Start() return code change.

All signing commands, including attestation commands, return TPM_RC_KEY for a non-signing key.

TPM2_SetCommandCodeAuditStatus() is not audited when used to change the algorithm.

Trial policy sessions check authorizations.

DA protection does apply to TPM_RH_LOCKOUT.

continueAuthSession is ignored for a password session.

Reworked NV attributes to accommodate more NV types. Defined TPM_NT.

For a hybrid counter Index, the first write always writes through to NV memory.

Added ECC point padding description.

The algorithm ID table in this specification is informative.

Context gap must be 2^n-1.

Handle type 0x03 is for saved sessions, not active session.

Timeout is of length TPM2B_DIGEST, not UINT64.

nullProof can be used in a ticket.

TPM2_EncryptDecrypt() uses an unrestricted key. The sign attribute is used as an encrypt attribute. A non-null mode cannot be overridden.

A TPM2_PolicySecret() being satisfied by a policy requires a password or auth value. The object must permit password or HMAC authorization.

TPM2_PolicyNV() is an immediate assertion.

NULL password can have continue set or clear.

Sign attribute becomes encrypt attribute for a symmetric cipher object.

Saved context metadata is normative. Encrypted data is vendor-specific.

TPMU_SYM_MODE, TPMS_SCHEME_XOR selector permits NULL.

If the session requires a policy session, returns TPM_RC_AUTH_TYPE.

TPM2_NV_Certify() returns TPM_RC_NV_UNINITIALIZED if unwritten even if size is zero.

Advised that callers should not use NV read public to calculate the Name.

Removed advice that FIPS may require an *authValue* size of half the hash algorithm digest size.

Clarified that *nonceTPM* is only used once in an HMAC calculation when the session is being used for both encrypt and decrypt.

Clarified that *authValue* is an Empty Buffer if a session is not an authorization session.

Clarified that *sessionValue* for authorization sessions that are encrypt or decrypt sessions is *sessionKey* ‖ *authValue* regardless of binding.

Clarified that nameAlg is the *authPolicy* hash algorithm.

Structure definition lower limits apply to TPM inputs. Upper limits refer to inputs and outputs.

The year and day of year can indicate an errata date.

TPM_RC_NONCE is returned for a nonce value mismatch.

TPMS_ALGORITHM_DETAIL_ECC kdf can be TPM_ALG_NULL.

TPMS_CONTEXT *savedHandle* indicates the context type.

If a handle in handle area references a session and the session is not present, returns TPM_RC_REFERENCE_H0 + N.

Clarified that the size of an encrypted parameter can be zero.

TPM2_Startup() can result in the PCR update counter non-zero because of PCR resets.

For RSA salt key, the size of an encrypted salt must be the same as the size of the public modulus.

TPM2_ECDH_KeyGen() requires *restricted* CLEAR and *decrypt* SET.

TPM2_Commit() does not require the *sign* attribute.

TPM2_PolicyOR() extends the digest into a Zero Digest PolicyDigest. It does not replace the digest.

TPM2_PolicyPCR() with a trial policy may use the TPM PCR if the caller does provide PCR settings.

TPM2_PolicyNV(), TPM2_PolicyCounterTimer(), TPM2_NV_Certify(), can return TPM_RC_VALUE if the offset is greater than the data size.

Clarified that TPM2B_DATA is the size of a TPMT_HA but is not required to contain an algorithm ID.

Clarified that time can be set to zero at _TPM_Init or TPM2_Startup().

TPM2_StartAuthSession() rejects a symmetric salt key. Session-based encryption should support XOR, but a block cipher is platform-specific.

Added TPM_PT_MODES for FIPS and other indications. Added TPMA_MODES.

Clarified the TPMA_STARTUP_CLEAR attribute (enable flags) settings on the various startup types.

_PRIVATE structure changed from TPMT_SENSITIVE to TPM2B_SENSITIVE.

Added restrictions on *unique* input for TPM2_Create() and TPM2_CreatePrimary(). Removed obsolete TPM_CC_PP_FIRST and TPM_CC_PP_LAST.

Removed symmetric salt.

sensitiveDataOrigin is set for an asymmetric object.

Clarified that only the template *unique* field may be altered when an object is created.

ehProof is changed on TPM2_Clear().

TPM2_SetPrimaryPolicy() requires a policy length consistent with the hash algorithm.

Augmented “Object Creation / Introduction” by adding the table “Creation Commands” and a description of that table.

Augmented “Entropy Creation / Introduction” by adding the table “Deriving Cryptographic Values” and a description of that table.

Added TPM2_PolicyTemplate(), TPM2_CreateLoaded(), TPMI_DH_PARENT.

Added TPM2_PolicyAuthorizeNV(), TPM2_EncryptDecrypt2().

Noted that TPM2_Create() may require transient resources.

TPM2_Clear() increments the *pcrUpdateCounter*, permitting a policy that can be invalidated on TPM2_Clear().

TPM_PT_NV_BUFFER_MAX returns the maximum size for NV read and NV certify as well as NV write.

Noted that TPMA_NV_POLICY_DELETE with a policy that cannot be satisfied defines an Index that can never be deleted.

TPM2_NV_Read() ignores *offset* for bits and counter indexes.

Added application note on audit alternative.

Added command code for TPM2_PolicyAuthorizeNV() and TPM2_EncryptDecrypt2().

Added GetCapability TPM_CAP_AUTH_POLICIES for hierarchy policies, and new structure TPMS_TAGGED_POLICY.

Offset is ignored when reading counter and bits NV indexes.

ReadClock can have audit session.

Added additional option to ticket expiration, and *timeEpoch*.

TPM2B_PRIVATE always has authorization value padded.

Clarified GPIO inputs and outputs.

EC Schnorr computation changes.

Salt always uses OAEP.

KDF must reject weak keys.

TPM2_Create() for a *fixedParent* storage key only requires the symmetric algorithm of the parent and child to match.

Policy ticket creation also digests the *timeEpoch*.

Weak symmetric keys will not be generated and cannot be loaded.

OAEP uses the object's scheme. If the object's scheme is TPM_ALG_NULL, uses the objects Name algorithm.

GPIO input and output settings are platform or vendor-specific.

Modified the ECDAA signature calculation.

Added TPM2_PolicyAuthorize() definition.

Noted that weak symmetric keys are not permitted.

OAEP uses the key's scheme unless it is NULL.

Modifications to the ECDAA sign operation.

Parents use CFB mode, and cannot have a NULL symmetric algorithm

The salt key scheme must be NULL or OAEP.

Updated the interaction between nonceTPM and expiration.

Data may be a non-Empty Buffer when a primary key is created.

TPM2_PolicySecret() referencing a PIN Pass Index returns a NULL ticket.

TPM2_SelfTest() returns TPM_RC_FAILURE on failure.

phEnableNV is set on TPM Reset or TPM Restart

TPM2_Create() and TPM2_CreatePrimary() input is actually TPM2B_PUBLIC even though the parameter says TPM2B_TEMPLATE.

TPM2_PolicySecret() for PIN and non-PIN Index clarifications.

TPM2_PolicyNV(), TPM2_NV_Read(), TPM2_NV_Certify() may ignore offset parameter.

TPM2_NV_GlobalWriteLock(), TPM2_NV_ReadLock() may write NV.

Added back expiration comment that timeout cannot become smaller.

Explained the result of TPM_CAP_AUTH_POLICIES.

Version 1.59

Added Attached Component (AC Send) description, structures, and functions.

TPM2_ECC_Parameters() may zero pad results.

TPM2_DictionaryAttackParameters() does not reset failedTries.

Clarified that the KDFa 0x00 byte is only explicitly added if Label is not present or if it is not NULL-terminated.

Clarified that recoveryTime may be tracked through a shutdown.

Added TDES annex explaining parity generation.

Clarified HMAC key calculation for bound policy session with and without TPM2_PolicyAuthValue(). Similar clarification for encrypted policy session.

Added the TPM2_MAC() commands and merged with TPM2_HMAC() commands. Added TPMI_ALG_MAC_SCHEME.

Changed TPM2B_TIMEOUT back to a UINT64.

TPM2_FlushContext() for sessions ignores the upper byte of the handle.

Minor updates for TPM2_MAC().

Added TPMI_ALG_CIPHER_MODE, used for EncryptDecrypt.

Salt key must be a decrypt key.

seedValue is the size of the *nameAlg* digest.

Field upgrade should preserve the TPM vendor-provisioned EKs.

Salt can only use asymmetric key encryption.

Alternative implementation of *failedTries* on non-orderly shutdown.

Added description of entropy usage for derived objects.

Alternate implementations for NV counter index initialization.

TPM_PT_NV_COUNTERS_MAX - zero value indicates no specified maximum.

Added TPMI_DH_SAVED for handle values that can be used in TPM2_ContextSave() or TPM2_FlushContext().

TPMS_SCHEME_XOR cannot have a NULL hash algorithm

TPM2_PolicyTemplate() error codes if command is sent twice or if *cpHash* is already set.

Reworked the attestation key certification to indicate that an encrypted challenge response is a more likely use case than an encrypted certificate.

Field upgrade should not affect TPM2_CreatePrimary() outputs under certain conditions.

The reset of the Time circuit is related to TPM power, not TPM_Init.

MAX_SYM_DATA 128 changed from shall to should.

Sign and decrypt both CLEAR or SET and scheme not TPM_ALG_NULL returns TPM_RC_SCHEME.

TPM2_PCR_Allocate() takes effect at _TPM_Init(), not TPM2_Startup().

Clarified in the text that TPM2_PolicyDuplicationSelect() Names do not include the size.

The TPM may enter Failure mode if TPM2_Startup() is not TPM_SU_CLEAR after an algorithm set change that affects PCR banks. It was previously not a may.

After a field upgrade, preserving seeds, etc. was changed from shall to should.

Part 1 added phEnableNV to STATE_CLEAR_DATA, clearCount increments on TPM Restart, not TPM Resume

Noted that TPM2_EventSequenceComplete() always returns all hashes.

Noted that TPM2_PCR_Allocate() requirement for TPM_SU_CLEAR only applies until after the next _TPM_Init.

Added some notes about the interaction between audit and parameter encryption. Clarified that the audit digest is a single hash of *cpHash* and *rpHash*.

The random commit value has to be at least equal to the security strength of the signing key. KDFa for the commit calculation uses *vendorAlg*, not *nameAlg*.

+ decoration only applies to command parameters, not response parameters.

TPM2_Startup() does not clear the written bit for an orderly counter Index.

TPM2_Hash() and TPM2_SequenceComplete() creates a ticket, with an Empty digest if in the NULL hierarchy.

TPM2_VerifySignature() returns a ticket with an Empty digest if the key is in the NULL hierarchy.

Updated ECC key generation and point padding in the Part 1 annex.

The TPM_PT_PS_REVISION value is platform specific.

Moved implementation-specific description of the *Clock Safe* flag to an example.

Clarified that the GetCapability returning TPML_PCR_SELECTION must return a selection for allocated banks but can return additional selections.

Added a first draft of TPM2_CertifyX509().

C.5 ECC Key Generation changed d to c and G to Q.

TPM2B_PRIVATE_KEY_RSA is permitted to be larger for *fixedTPM* keys. The TPM2B_PRIVATE structure in TPM2_Create() and TPM2_Load() may contain five CRT primes (instead of one).

Assigned TPM_CC_CertifyX509 command code, x509Sign attribute, TPMA_X509_KEY_USAGE, Added TPM2_CertifyX509() description, parameters, and actions.

Defined NV digest attestation structure, TPMS_NV_DIGEST_CERTIFY_INFO, and added certifying an NV digest to TPM2_NV_Certify().

Clarified that *label* in KDFa is an octet stream and the conditions for the KDFa zero byte.

Clarified that the required size of an object sensitive area *seedValue* for TPM generated and imported objects.

Clarified that the L parameter in OAEP is a byte stream with the last byte zero, not a null terminated string.

Added an Annex with a Library Profile Guide.

Clarified that TPM_PT_NV_BUFFER_MAX applies to NV extend or NV certify.

The TPMA_X509_KEY_USAGE keyAgreement and encipherOnly attributes require the decrypt attribute.

Explained that most of TPM2_SetAlgorithmSet() is vendor-dependent.

Explained that the initialization of the list of commands requiring physical presence is platform-specific.

Part 2 removed , S AND <IO> from several table titles.

TPM_ECC_CURVE add + to TPM_ECC_NONE

Added the ACT feature.

Explained that the TPM2_CertifyX509() *partialCertificate* and *addedToCertificate* are a DER encoded SEQUENCEs. Explained the encoding of the TPMA_OBJECT element. Noted that *tbsDigest* is returned as a debugging aid. Changed *qualifyingData* to *reserved* and that it must be an Empty Buffer.

Added a requirement that, if a command resets PCR in multiple banks, the PCR Update Counter must be incremented only once. If a command causes PCR in multiple banks to change, the PCR Update Counter must be incremented once for each bank.

Reversed the TPMA_X509_KEY_USAGE bit map.

Version 1.83

Added a note about the Reference Code not handling *failedTries* correctly in a non-orderly shutdown,

Added a note about the Reference Code handling the PIN Pass and PIN Fail *pinCount* incorrectly when the index was authorized with a policy session.

ACT: *preserveSignaled* is a state reflecting *signaled*, not a TPM attribute. TPM2_Startup() clears ACT items. TPM2_ACT_SetTimeout() clears *preservedSignaled*.

Based on a proposal from Greater China Regional Forum (GCRF) members, this version adds SM2 encryption/decryption support. The suggestion to add this capability was received by TCG as a comment from the ISO/IEC JTC 1 Chinese National Body on the Final Draft International Standard ballot for ISO/IEC 11889:2015.

Added TPM2_ECC_Encrypt() and TPM2_ECC_Decrypt() commands.

Documented the policy session context values and which ones are shared. Observed that a policy session cannot be used for session audit.

Explained that TPM2_CreateLoaded() can create child keys with the same key but different attributes.

Reworded the ACT *preserveSignaled* behavior on a power cycle.

Reworded the Part 2 TPMA_NV_AUTHWRITE and TPMA_NV_READ descriptions to match the other attributes. The behavior does not change.

The RSASSA-PSS salt size is now the largest size permitted by FIPS 186-4. A note explains that older TPMs may use a different maximum size.

TPM_RC_AUTH_FAIL is returned when *lockoutAuth* is disabled.

TPM2_PolicySecret() returns TPM_RC_AUTH_UNAVAILABLE for an NV Index with TPMA_NV_AUTHREAD CLEAR. The command will increment an NV Index *pinCount*.

TPM2_Startup() with a TPM Reset changes *nullProof* and *nullSeed*.

Observed that a saved sequence object does not have replay protection.

TPM2_Quote() may return TPM_RC_SCHEME if *signHandle* is NULL.

Added TPM2_PolicyCapability().

TPM2_NV_DefineSpace() returns an error if *dataSize* is greater than the maximum NV buffer size and TPMA_NV_WRITEALLis SET, since the Index could never be written.

Explained that long password hashing is by convention, not enforced by the TPM.

Added to localities supported by TPM2_Startup() to Library Profile Guide.

Explained that the TPM2_SequenceComplete() *hierarchy* parameter determines the ticket lifetime.

Added to TPM2_PolicyCounterTimer() comparison description.

Clarified that hybrid NV Indexes are not written on each write command.

TPM_EO signed operations always use two's complement.

TPM2_StartAuthSession() *nonceTPM* is a minimum of 16 bytes.

Clarified salted and bound session key generation to agree with the other clauses.

Clarified that the policy session digest algorithm must match the entity name algorithm.

Deprecated unused TPMS_ALGORITHM_DESCRIPTION.

Added TPML_VENDOR_PROPERTY.

Reworded the interaction between NV authorization, the NV lock commands, and the NV lock bits. The only normative change is that the TPM may return an error if a lock command is sent to a locked Index with invalid authorization.

Clarified that, if an object is subject to DA protection, its use in a bind session subjects the session to DA protection.

Clarified that PCR has implied *noDA*.

Clarified that *platformAuth* is the ACT *authValue*.

Removed the note that read lock requires read privilege.

Added TPM2_PolicyParameters() and reworded the relationships between TPM2_PolicyCpHash(), TPM2_PolicyNameHash(), TPM2_PolicyParameters(), TPM2_PolicyTemplate(), and bound sessions.

Added TPMA_OBJECT.*fixedFirmware* and added a section to Part 1 on using *fixedFirmware* to attest to the firmware version of the TPM.

Changed appendix to indicate that the Reference Code requires an RSA public exponent to be odd, not prime. Part 2 says the exponent must be a prime.

Added FIPS_140_3_INDICATOR to TPMA_MODES.

Added statement that the TPM does not verify key attributes in TPM2_ECC_Encrypt().

Added an informative TPM2B_VENDOR_PROPERTY and TPML_VENDOR_PROPERTY to support the specification parser.

Added informative changes to the algorithm ID sections and macro expansions.

Added value for TPM_HT_EXTERNAL_NV.

Added TPM_HT_PERMANENT_NV for a platform-specific NV Index that cannot be defined or undefined. Added TPM_HT_NV_INDEX_A64 for an internal NV Index with 64-bit attributes, Specified external and permanent NV with 64-bit attributes. Changed TPMA_NV to 64 bits. Added TPMI_RH_NV_DEFINED_INDEX to support permanent index. Explained how TPMS_NV_PUBLIC is serialized to support 32-bit and 64-bit attributes on the interface.

Added informative description of policySession→cpHash shared contents.

Added note that H-CRTM sets PCR 0 back to 0…04 each time.

Added informative expansion of the TPM_ALG_ALG! Macros.

Added TPM2_NV_DefineSpace2() and TPM2_NV_ReadPublic2() to support the TPM2B_NV_PUBLIC_2 structure. Removed TPM_HT_NV_INDEX_A64.

Added Key Generation seed table. Storage key uses the label “STORAGE”.

Added TPM_CC_NV_DefineSpace2 and TPM_CC_NV_ReadPublic2 command codes.

TPM2_CertifyX509() signHandle cannot be null.

Field upgrade preserves primary keys, not just the seeds.

fixedFirmware is clear if it is clear in the parent.

For TPM2_Load(), the integrity shall be checked before private area decryption.

Bind object changed to bind entity.

Rewrote injected EPS and EK sections.

Changed MAX_DIGEST_SIZE to MAX_2B_BUFFER_SIZE

Firmware limited and SVN limited:

- Added firmware-limited and SVN-limited hierarchies. Updated TPMI_DH_PARENT and TPMI_RH_HIERARCHY. Add TPMI_RH_BASE_HIERARCHY.
- TPM2_EvictControl() does not support firmware-limited or SVN limited objects.
- TPM2_GetCapability() supports SVN.
- Added permanent handles for firmware and SVN.
- TPMA_OBJECT.*fixedFirmware* moved to Bit 8 and renamed as *firmwareLimited.*
- Introduced TPMA_OBJECT.*svnLimited.*

TPM2_PolicyCapability() does not support TPM_CAP_PCRS.

Updated the informative to Algorithm Registry 1.34. Added LMS, XMSS, XOF, KMAC. ECC curve 488.

SVN limited objects can be in the NULL hierarchy.

Added TPM2_SetCapability().

TPMI_RH_HIERARCHY_AUTH has optional TPM_RH_NULL.

Added clarifying guidance on Derived Objects for consistency with Part 3.

Added note that attestation keys typically use sensitiveDataOrigin.

Version 184

Switched to TPM_SPEC_VERSION-based specification version scheme (184 instead of 1.84).

Introduced Part 0: Introduction.

Removed references to Part 4.

Introduced TPM2_ReadOnlyControl() and Read-Only mode.

Introduced TPM2_PolicyTransportSPDM().

Introduced TPM_CAP_PUB_KEYS and TPM_CAP_SPDM_SESSION_INFO.

Deprecated TPM_ALG_TDES and TPM_ALG_SHA1.

Deprecated TPM2_CreateLoaded() and TPM2_CertifyX509().

Deprecated Attached Components.

Version 185

Introduced support for ML-DSA, HashML-DSA, and ML-KEM.

Introduced support for EdDSA, HashEdDSA, and DHKEM.

Introduced the sequenced signing commands:

- TPM2_VerifySequenceStart()
- TPM2_SignSequenceStart()
- TPM2_VerifySequenceComplete()
- TPM2_SignSequenceComplete()

These commands can be used to sign and verify messages of arbitrary size (except for EdDSA, where messages of arbitrary size can be verified, but signing is limited to MAX_BUFFER).

Introduced the new digest signing commands:

- TPM2_VerifyDigestSignature()
- TPM2_SignDigest()

These commands replace TPM2_Sign() and TPM2_VerifySignature() and can be used with digest-signing algorithms (e.g., ECDSA but not EdDSA; HashML-DSA but not ML-DSA except when *allowExternalMu* is TRUE).

Introduced the new Key Encapsulation Mechanism (KEM) commands:

- TPM_CC_Encapsulate
- TPM_CC_Decapsulate

These commands can be used with KEMs such as ML-KEM and DHKEM.

Reframed the existing “Secret Sharing” for Restricted Decryption operations as “Labeled KEM”, defining the RSA and ECDH-based Labeled KEMs. Defined the ML-KEM-based Labeled KEM.

Enabled persisting public-only objects.

Renamed TPM_PT_DAY_OF_YEAR to TPM_PT_ERRATA and TPM_SPEC_DAY_OF_YEAR to TPM_SPEC_ERRATA. The Errata version implemented by a TPM is now reported as a version number rather than as a date.

Introduced TPM_PT_ML_PARAMETER_SETS.

Removed the !ALG_ macro tokens (by expanding them). Structures that relied on the macros are now written out legibly.

Deprecated TPM2_Sign() and TPM2_VerifySignature() (in favor of TPM2_SignDigest() and TPM2_VerifyDigestSignature()).

Deprecated keys having both *sign* and *decrypt* SET.

Deprecated keys with a NULL *scheme* except for Storage Keys and keys intended for use with the raw RSA primitive.

Deprecated the bits of TPMA_ALGORITHM and deleted TPMS_ALGORITHM_DESCRIPTION, which was unused.

Deprecated TPMS_ALGORITHM_DETAIL_ECC.*kdf*.

Removed normative (algorithm-specific) annexes in Part 1, making them normal clauses instead.

3 Deprecated Functionality

3.1 Introduction

The following features are deprecated in this version of the TPM 2.0 specification, and may be removed from a future version.

3.1.1 Deprecated Algorithms

3.1.1.1 TPM_ALG_TDES

TPM_ALG_TDES was deprecated in TPM 2.0 version 184.

TDES was withdrawn by NIST on January 1, 2024 [1]. Users should prefer a more modern block cipher, such as AES, instead.

3.1.1.2 TPM_ALG_SHA1

TPM_ALG_SHA1 was deprecated in TPM 2.0 version 184.

SHA1 will be withdrawn by NIST on January 1, 2031 [2]. Users should prefer a more modern hash algorithm, such as SHA256, instead.

3.1.2 Deprecated Structures

3.1.2.1 TPMA_ALGORITHM

All bits in TPMA_ALGORITHM were deprecated in version 185, due to a lack of use case for the TPM classifying cryptographic algorithms for the user.

Users can query the TPM for general support for a given algorithm ID (e.g., TPM_ALG_AES) with TPM2_GetCapability(*capability* = TPM_CAP_ALGS). A user may use TPMS_ALG_PROPERTY.*alg* to indicate the presence or absence of support. A user may ignore TPMS_ALG_PROPERTY.*algProperties* altogether.

Users can query the TPM for support for a given command code (e.g., TPM_CC_EncryptDecrypt2) with TPM2_GetCapability(*capability* = TPM_CAP_COMMANDS).

Users can query the TPM for whether a key with specific parameters can be created (e.g., a 192-bit AES key) using TPM2_TestParms().

3.1.3 Deprecated Commands

3.1.3.1 TPM2_EncryptDecrypt

TPM2_EncryptDecrypt() was deprecated in TPM 2.0 version 1.38.

It was replaced by TPM2_EncryptDecrypt2(), which allows for parameter-encryption of *inData*. It is the same as TPM2_EncryptDecrypt() in all other respects.

3.1.3.2 TPM2_CreateLoaded

TPM2_CreateLoaded() was deprecated in TPM 2.0 version 184.

Where possible, users should adopt TPM2_Create() and TPM2_CreatePrimary() instead.

Users with use cases for derived objects should refer to Part 1, “Caution on use of Derivation Parents” for guidance.

[1] https://csrc.nist.gov/pubs/sp/800/67/r2/final
[2] https://www.nist.gov/news-events/news/2022/12/nist-retires-sha-1-cryptographic-algorithm

3.1.3.3 TPM2_CertifyX509

TPM2_CertifyX509() was deprecated in TPM 2.0 version 184 because its nuances caused confusion.

Where possible, users should adopt TPM2_Certify() with an Attestation Key instead.

3.1.3.4 TPM2_AC_GetCapability, TPM2_AC_Send, TPM2_Policy_AC_SendSelect

While there is potential value in connecting a TPM directly to another component so that the TPM can manage secrets for it, TPM2_AC_GetCapability(), TPM2_AC_Send(), and TPM2_Policy_AC_SendSelect() were deprecated in TPM 2.0 version 184 due to complexity and lack of adoption.

3.1.3.5 TPM2_Sign, TPM2_VerifySignature

TPM2_Sign() was deprecated in TPM 2.0 version 185. Its functionality is replaced by TPM2_SignDigest() and the sequenced signing commands, which were introduced in order to accommodate post-quantum algorithms.

TPM2_VerifySignature() was deprecated in TPM 2.0 version 185. Its functionality is replaced by TPM2_VerifyDigestSignature() and the sequenced signing commands, which were introduced in order to accommodate post-quantum algorithms.

3.1.4 Other Deprecations

3.1.4.1 Both *sign* and *decrypt* in TPMA_OBJECT

When unmarshaling a TPMA_OBJECT, having both *objectAttributes.sign* SET and *objectAttributes.decrypt* SET was deprecated in TPM 2.0 version 185 unless *type* is TPM_ALG_SYMCIPHER.

A future version of the TPM 2.0 specification could allow or require the TPM to respond with an error code in this case.

3.1.4.2 Null *scheme* in TPMT_PUBLIC

In the context of TPM2_Create(), TPM2_CreatePrimary(), TPM2_CreateLoaded(), and TPM2_LoadExternal(), the TPM will sometimes allow a signing or encryption scheme of TPM_ALG_NULL. This allows users to use the same key in multiple cryptographic schemes, which is poor cryptographic hygiene.

When unmarshaling a TPMT_PUBLIC, the following cases were deprecated in TPM 2.0 version 185:

- *parameters.keyedHashDetail.scheme* = TPM_ALG_NULL if *objectAttributes.sign* is SET
- *parameters.rsaDetail.scheme* = TPM_ALG_NULL unless the object is a Restricted Decryption key or intended for use with raw RSAEP/RSADP operations
- *parameters.eccDetail.scheme* = TPM_ALG_NULL unless the object is a Restricted Decryption key

A future version of the TPM 2.0 specification could allow or require the TPM to respond with an error code in these cases.

3.1.4.3 Salted Session Key Generation with Unrestricted Keys

Part 1, “Salted Session Key Generation” describes how a decryption key (which can be restricted or unrestricted) can be used to establish a secure session with the TPM.

TPM2_StartAuthSession() with an unrestricted *tpmKey* was deprecated in TPM 2.0 version 185.

Users who are using unrestricted keys for salted sessions should switch to using restricted keys for salted sessions, to ensure that the session salt can only ever be decrypted for internal TPM use.

A future version of the TPM 2.0 specification could allow or require the TPM to respond with an error code in this case.

3.1.4.4 TPMS_ALGORITHM_DETAIL_ECC.*kdf*

`TPM2_ECC_Parameters()` reports information about the requested ECC curve in a `TPMS_ALGORITHM_DETAIL_ECC` structure. This structure contains a field called *kdf*, which describes the default KDF algorithm recommended for use with keys that use the given curve. The TPM does not internally use this information. Therefore, this field was deprecated in TPM 2.0 version 185.

A future version of the TPM 2.0 specification could allow or require the TPM to populate this field with `TPM_ALG_NULL` in all cases.

4 Trusted Platforms

4.1 Trust

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

Example:

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

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

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

4.2 Trust Concepts

4.2.1 Trusted Building Block

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

Example:

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

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

4.2.2 Trusted Computing Base

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

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

4.2.3 Trust Boundaries

The combination of TBB and Roots of Trust form a trust boundary, within which measurement, storage, and reporting can be accomplished for a minimal configuration. In systems that are more complex, it can be necessary for the CRTM to establish trust in other code, by making measurements of that other code and recording the measurement in a Platform Configuration Register (PCR). If the CRTM transfers control to that other code regardless of the measurement, then the trust boundary is expanded. If the CRTM will not run that code unless its measurement is the expected value, the trust boundary remains the same because the measured code is an expected extension of the CRTM.

4.2.4 Transitive Trust

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

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

4.2.5 Trust Authority

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

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

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

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

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

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

Note:

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

4.3 Trusted Platform Module

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

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

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

Example:

An example of a low-performance interface is the Low Pin Count, or LPC, bus.

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

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

4.4 Roots of Trust

4.4.1 Introduction

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

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

Example:

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

This certification provides confidence in the Roots of Trust implemented in the TPM. In addition, a certificate from a platform manufacturer can provide assurance that the TPM was properly installed on a machine that is compliant with TCG specifications so that the Root of Trust provided by the platform can be trusted (see Clause 4.5.2 for more information on certification).

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

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

Trust in the Roots of Trust can be achieved through a variety of means but is expected to include technical evaluation by competent experts.

4.4.2 Root of Trust for Measurement (RTM)

The RTM sends integrity-relevant information (measurements) to the RTS. Typically, the RTM is the CPU controlled by the Core Root of Trust for Measurement (CRTM). The CRTM is the first set of instructions executed when a new chain of trust is established. When a system is reset, the CPU begins executing the CRTM. The CRTM then sends values that indicate its identity to the RTS. This establishes the starting point for a chain of trust (see Clause 4.5.5 for a more detailed description of integrity measurement).

4.4.3 Root of Trust for Storage (RTS)

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

Some of the information in TPM memory locations is not sensitive and the TPM does not protect it from disclosure. An example of non-sensitive data is the current contents of a platform configuration register (PCR) containing a digest. Other information is sensitive and the TPM does not allow access to the information without proper authority. An example of sensitive data in a Shielded Location is the private part of an asymmetric key.

Sometimes, the TPM uses the contents of one Shielded Location to gate access to another Shielded Location. For example, access to (use of) a private key for signing can be conditioned on PCRs having specific values.

4.4.4 Root of Trust for Reporting (RTR)

4.4.4.1 Description

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

Note:

Not all Shielded Locations are directly accessible. For example, the values of the private part of keys and authorizations are in Shielded Locations on which the TPM will not report.

The values on which the RTR reports typically are

- evidence of a platform configuration in PCR (such as, `TPM2_Quote()`),
- audit logs (such as, `TPM2_GetCommandAuditDigest()`), and
- key properties (such as, `TPM2_Certify()`).

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

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

4.4.4.2 Identity of the RTR

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

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

4.4.4.3 RTR Binding to a Platform

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

4.4.4.4 Platform Identity and Privacy Considerations

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

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

Note:

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

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

4.5 Basic Trusted Platform Features

4.5.1 Introduction

At a minimum, a trusted platform provides the three Roots of Trust described previously. All three roots use certification and attestation to provide evidence of the accuracy of information. A trusted platform will also offer Protected Locations (see Part 1, "Protection of Shielded Locations") for the keys and data objects entrusted to it. Finally, a trusted platform can provide integrity measurement to ensure the trustworthiness of a platform by logging changes to platform state; this is done by recording logged entries in PCR for later validation as being correct and unaltered. These basic TPM concepts are now described in detail.

4.5.2 Certification

The nominal method of establishing trust in a key is with a certificate indicating that the processes used for creating and protecting the key meets necessary security criteria. A certificate can be provided by shipping the TPM with an embedded key (that is, an Endorsement Key) along with a Certificate of Authenticity for the EK. The EK and its certificate can be used to associate credentials (certificates) with other TPM keys; this process is described in Clause 4.5.3.3. When a certified key has attributes that let it sign TPM-created data, it can attest to the TPM-resident record of platform characteristics that affect the integrity (trustworthiness) of a platform.

Note:

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

4.5.3 Attestation and Authentication

4.5.3.1 Types of Attestation

Trusted platforms employ a hierarchy of attestations:

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

   Note:

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

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

   Note:

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

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

   Note:

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

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

   Note:

   This type of attestation is done using `TPM2_Certify()`.

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

   Note:

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

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

Note:

This is commonly called “third-party certification.”

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

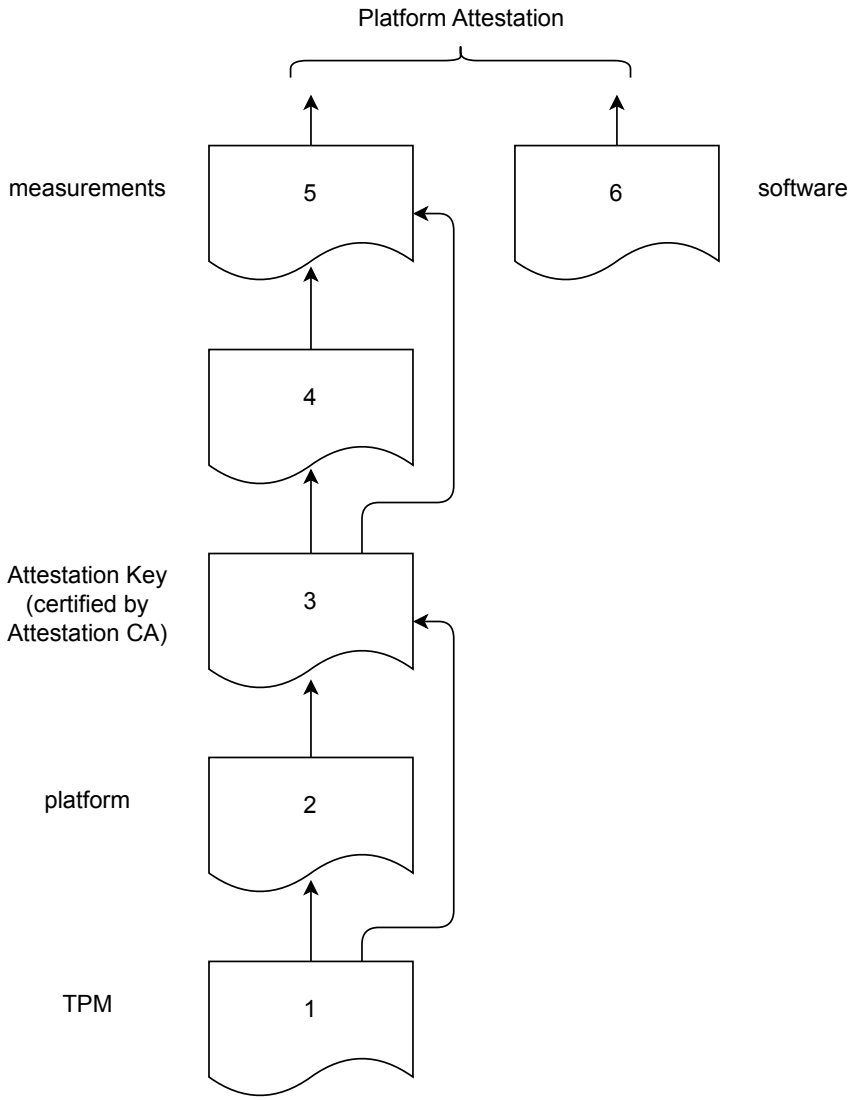

Figure 1: Attestation Hierarchy

4.5.3.2 Attestation Keys

When the TPM creates a message to sign from internal TPM state (such as, in `TPM2_Quote()`), a special value (`TPM_GENERATED_VALUE`) is used as the message header. A TPM-generated message always begins with this value.

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

Example:

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

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

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

4.5.3.3 Attestation Key Identity Certification

Any TPM user that can create a key on a TPM can create a restricted-use signing key. The key creator can then ask a third party, such as an attestation Certificate Authority (CA), to provide a certificate for it. The attestation CA can request that the caller provide some evidence that the key being certified is a TPM-resident key.

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

Note:

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

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

If the certified key is a decryption key and cannot sign, then an alternative method is used to allow the new key or data object to be reliably certified. For this alternative certification, the identity of the Object to be certified and a certificate for the decryption key (such as an EK) are provided to the CA. From the certificate, the CA determines the public key for the decryption key. The CA then produces a challenge for the Object to be certified and encrypts the challenge with the certified key. The encrypted challenge is given to the TPM containing both the certified decryption key and the key to be certified.

The challenge is protected using methods that are dependent on the type of the certified decryption key. The general method is described in Part 1, “Credential Protection.” Additional methods appropriate to RSA keys are described in Part 1, “RSA” and additional methods appropriate to ECC keys are in Part 1, “ECC”. The protection process produces an encrypted blob, an HMAC over the blob, and a secret value that can only be recovered by the certified decryption key.

TPM2_ActivateCredential() is used to access the challenge. The TPM recovers the secret value and uses it to generate the keys necessary to decrypt and validate the HMAC and encrypted blob. If the challenge is recovered successfully, and the key being certified by the credential is loaded on the TPM, then the challenge

is returned to the caller, and then provided to the CA. After the CA validates the challenge, it can issue the certificate for the key.

Note:

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

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

4.5.4 Protected Location

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

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

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

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

Primary Seeds can be changed, and when they are changed, the objects they protected will no longer be usable. For example, the Storage Primary Seed (SPS) creates the Storage hierarchy for owner-related data, and that seed changes when the owner changes.

4.5.5 Integrity Measurement and Reporting

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

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

An alternative method of initializing the platform is available on some processor architectures. It lets the CPU act as the CRTM and apply protections to portions of memory it measures. This process lets a new chain of trust start without rebooting the platform. Because the RTM can be re-established dynamically, this method is

called dynamic RTM (D-RTM). Both S-RTM and D-RTM can take a system in an unknown state and return it to a known state. The D-RTM has the advantage of not requiring the system to be rebooted.

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

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

The RTM (usually, code running on the CPU) makes these measurements and records them in RTS using Extend. The Extend process (see Part 1, “Extend of a PCR”) allows the TPM to accumulate an indefinite number of measurements in a relatively small amount of memory.

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

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

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

5 Policy Examples

This clause compares authorization between TPM 1.2 and this specification.

5.1 TPM 1.2 Compatible Authorization

A TPM 1.2 key may have its use gated by PCR and *authValue*. To select this authorization, the key would be created with a *pcrSelection* with at least one bit SET and the *digestAtRelease* set to indicate the digest of the selected PCR. Additionally, the key's TPM_AUTH_DATA_USAGE would be set to TPM_AUTH_ALWAYS. To perform the authorization, an authorization session is created and used to prove knowledge of the *authValue* in the authorization HMAC. If the HMAC check is successful and the digest of the selected PCR matches the *digestAtRelease*, the action is approved.

For a TPM compatible with this specification, use of PCR for access control requires a policy. The policy should be created at the time of object creation so that the policy requires selected PCR to have a specific value. This is similar to determining the *digestAtRelease* in TPM 1.2. The policy will use two factors: PCR and an *authValue*. The first policy command will be TPM2_PolicyPCR() and it will modify the *policyDigest* by:

$$policyDigest_1 := \mathbf{H}_{contextAlg}(policyDigest_0 \parallel \text{TPM\_CC\_Policy\_PCR} \parallel \text{PCRSelection} \parallel \text{PCRdigest}) \quad (1)$$

where

$\mathbf{H}_{contextAlg}$	is a hash function using the context hash algorithm
$policyDigest_0$	is an array of octets of zero equal in length to the size of the policy digest
TPM_CC_Policy_PCR	is a constant indicating the command modifying the *policyDigest* is TPM2_PolicyPCR()
PCRSelection	is a TPML_PCR_SELECTION that indicates the PCR that will be included in the PCR digest
PCRdigest	is the expected digest of the PCR selected by the PCR Selection; the PCR are hashed using the hash algorithm of the policy session

To cause the TPM to compute an HMAC using the *authValue* of the object, a TPM2_PolicyAuthValue() would be included in the policy. It would modify the *policyDigest* as:

$$policyDigest_2 := \mathbf{H}_{contextAlg}(policyDigest_1 \parallel \text{TPM\_CC\_PolicyAuthValue}) \quad (2)$$

where

$\mathbf{H}_{contextAlg}$	is a hash function using the context hash algorithm
$policyDigest_1$	is the result of performing the operation in Equation 1 above
TPM_CC_PolicyAuthValue	is the command code for TPM2_PolicyAuthValue()

The value of $policyDigest_2$ would be included in the template of the object in the *authPolicy* parameter. To use the object, a policy authorization session would be started using TPM2_StartAuthSession(). Then a TPM2_PolicyPCR() and TPM2_PolicyAuthSession() would be executed using the handle of the

authorization session. If the PCR were the same as those used when performing the operation of Equation 1, then the *policyDigest* of the policy session will match the *authPolicy* of the object. Because the policy sequence contained `TPM2_PolicyAuthValue()`, the TPM will check that the HMAC in the authorization indicates that the caller knows the *authValue* of the object (same computation as performed on an HMAC session). If both checks succeed, the object is properly authorized.

6 Library Profile Guide

6.1 Introduction

This clause provides guidance to TPM platform-specific work groups when developing platform-specific TPM specifications. The platform-specific specification must specify these items. It aggregates platform-specific information from other parts of this specification.

Functionality described in the TPM Library Specification may also be implemented by devices other than TPMs.

6.2 Platform-Specific Constants

- Constants returned by `TPM2_GetCapability()` with the capability prefix `TPM_PT_PS`. See Part 2.
- The manufacturer, vendor strings, and firmware version.

6.3 Hierarchies

- Which hierarchies to implement (including Firmware- and SVN-limited hierarchies).

6.4 PCR

- Number of PCR.
- The minimum size of a selection structure.
- Number of banks.
- Supported hash algorithms.
- PCR authorization, authorization groups, and locality.
- Which can be reset, under what conditions, and what the reset value is.
- Whether the PCR is preserved on resume.
- Which PCR increment the PCR update counter.
- D-RTM and H-CRTM behavior (discussed in Part 1), and PCR values at `_TPM_Init()`.
- Which localities are supported for `TPM2_Startup()`, which affects the initial value of PCR[0].

6.5 Algorithms

Define algorithms. See the TCG Algorithm Registry.

- Hash, asymmetric, and symmetric algorithms.
- For elliptic curve, the curves.
- For RSA, the public exponent.
- Key sizes.
- Padding modes.
- Endorsement key certificate provisioning, and the NV attributes for the certificates.
- Algorithms and modes for parameter encryption.

6.6 Commands

See Part 2 and note that some commands have prerequisites or are implemented as a set.

- Mandatory, optional, and forbidden commands.
- Whether firmware upgrade is required, and whether the library specification or a vendor-specific method is permitted.

6.7 Buffers

- The minimum for the maximum size of an NV Index.
- The minimum size for the input and output buffers.

6.8 NV Storage

- Specify as much as possible the NV storage capacity. Since NV Indexes has varying metadata requirements, the value may not be exact.
- The types of NV Indexes supported. This is linked to the supported TPM2_NV_ commands.
- Whether external NV is supported and what its attributes are encrypted, integrity protected and/or rollback protected.
- Whether permanent NV is required. If required, specify the handle and attributes.
- Whether internal NV with 64-bit attributes is supported.
- Whether NV Indexes and persistent keys may or may not come from the same NV memory pool. See TPMA_MEMORY.*sharedNV* in Part 2.
- A minimum number of total Indexes, and minimums for certain Index types, such as counters.
- Whether both orderly and non-orderly indexes are supported.
- The minimum number of persistent key slots.

6.9 Sessions and Objects

- The minimum number of loaded and active sessions.
- The minimum number of loaded objects.
- Whether authorization sessions and transient objects come from the same memory pool. See TPMA_MEMORY.*sharedRAM* in Part 2.
- Whether persistent objects are copied to RAM when referenced. See TPMA_MEMORY.*objectCopiedToRam* in Part 2.

6.10 Physical Presence

- If physical presence is implemented, specify the table of commands that require physical presence.

6.11 Dictionary Attack Lockout

- The default value for maxTries and recoveryTime.

6.12 Self Test

- Whether TPM2_SelfTest() can be blocking or non-blocking.

6.13 Authenticated Countdown Timers (ACT)

- The number of supported ACT instances (usually 0 or 1).
- Trigger event when the ACT times out.
- Whether `TPM2_ClockRateAdjust()` may affect the ACT rate, and maximum adjustment.
- The ACT behavior if TPM is in a low power state (sleep mode) (typically, ACT must advance).
- Whether the ACT state must be preserved over a power cycle (setting of the *preserveSignaled* attribute).
- Whether clearing or setting the *signaled* attribute must also clear or set the associated trigger event (e.g., when ACT signaling is turned off, or ACT signaling is preserved across TPM Resume).
- Whether the remaining ACT timeout must be retrievable in TPM Failure Mode.

7 Implementation Guide

7.1 Field Upgrade

- See Part 1, “Preserved TPM State”. If keys are derived from primary seeds, the derivation process should remain the same across field upgrades.
- See Clause 3. An object created on an old version of the firmware might have attributes that are considered deprecated or invalid by the new version of the firmware. The choice of whether to invalidate these objects or keep them as-is is up to the implementation.

