GlobalPlatform Technology

TPS Keystore Protocol Specification

Version 1.0

Public Release

March 2025

Document Reference: GPP_SPE_005

Contents

Tables

Figures

1 Introduction

TPS Keystore is a TPS Service that manages cryptographic keys, performs cryptographic operations, and provides attestations on the properties of the TPS Keystore and the keys within it. A TPS Client connects with TPS Keystore using the TPS Client API ([TPS Client API]), then communicates with TPS Keystore using the TPS Keystore Protocol defined in this specification

The TPS Keystore Protocol is particularly suited to managing COSE keys, which are used to manage COSE structures ([COSE Struct]) as used, for example, in the GlobalPlatform Entity Attestation Protocol Specification ([TPS EAT]).

Recommendations for best practices and acceptable cryptography usage can be found in GlobalPlatform's Cryptographic Algorithm Recommendations ([Crypto Rec]). Relevant sections of that document MAY be applied to the interfaces and API offered by this specification. As always, the developer should refer to appropriate security guidelines.

If you are implementing this specification and you think it is not clear on something:

1. **Check with a colleague.**

And if that fails:

2. **Contact GlobalPlatform at TEE-issues-GPP_SPE_005@globalplatform.org**

1.1 Audience

This document is suitable for software developers implementing:

- TPS Keystores running in a Secure Component (SC)
- TPS Clients running in the Device and which use the TPS Keystores via the TPS Client API

1.2 IPR Disclaimer

Attention is drawn to the possibility that some of the elements of this GlobalPlatform specification or other work product may be the subject of intellectual property rights (IPR) held by GlobalPlatform members or others. For additional information regarding any such IPR that have been brought to the attention of GlobalPlatform, please visit https://globalplatform.org/specifications/ip-disclaimers/. GlobalPlatform shall not be held responsible for identifying any or all such IPR, and takes no position concerning the possible existence or the evidence, validity, or scope of any such IPR.

1.3 References

The tables below list references applicable to this specification. The latest version of each reference applies unless a publication date or version is explicitly stated.

Table 1-1: Normative References

Standard / Specification	Description	Ref
GPP_SPE_009	GlobalPlatform Technology TPS Client API Specification	[TPS Client API]
IETF RFC 2119	Key words for use in RFCs to Indicate Requirement Levels	[RFC 2119]
IETF RFC 3610	Counter with CBC-MAC (CCM) https://datatracker.ietf.org/doc/html/rfc3610	[CCM]
IETF RFC 3629	UTF-8, a transformation format of ISO 10646 https://datatracker.ietf.org/doc/html/rfc3629	[UTF8]
IETF RFC 4122	A Universally Unique IDentifier (UUID) URN Namespace https://datatracker.ietf.org/doc/html/rfc3629	[UUID]
IETF RFC 4493	The AES-CMAC Algorithm	[RFC 4493]
IETF RFC 4648	The Base16, Base32, and Base64 Data Encodings	[RFC 4648]
IETF RFC 5280	Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile https://datatracker.ietf.org/doc/html/rfc5280	[PKIX]
IETF RFC 7517	JSON Web Key (JWK) https://datatracker.ietf.org/doc/html/rfc7517	[JWK]
IETF RFC 8017	PKCS #1: RSA Cryptography Specifications Version 2.2 https://datatracker.ietf.org/doc/html/rfc8017	[PKCS 1]
IETF RFC 8174	Amendment to RFC 2119	[RFC 8174]
IETF RFC 8230	Using RSA Algorithms with CBOR Object Signing and Encryption (COSE) Messages https://datatracker.ietf.org/doc/html/rfc8230	[COSE RSA]
IETF RFC 8439	ChaCha20 and Poly1305 for IETF Protocols https://datatracker.ietf.org/doc/html/rfc8439	[ChaCha-Poly]
IETF RFC 8610	Concise data definition language (CDDL): A Notational Convention to Express CBOR Data Structures https://datatracker.ietf.org/doc/html/rfc8610	[CDDL]
IETF RFC 8812	CBOR Object Signing and Encryption (COSE) and JSON Object Signing and Encryption (JOSE) Registrations for Web Authentication (WebAuthn) Algorithms https://datatracker.ietf.org/doc/html/rfc8812	[COSE Reg]
IETF RFC 8949	Concise Binary Object Representation (CBOR) https://datatracker.ietf.org/doc/html/rfc8949	[CBOR]

Standard / Specification	Description	Ref
IETF RFC 9052	CBOR Object Signing and Encryption (COSE): Structures and Process https://datatracker.ietf.org/doc/html/rfc9052	[COSE Struct]
IETF RFC 9053	CBOR Object Signing and Encryption (COSE): Initial Algorithms https://datatracker.ietf.org/doc/html/rfc9053	[COSE Algs]
IETF RFC 9054	CBOR Object Signing and Encryption (COSE): Hash Algorithms https://datatracker.ietf.org/doc/html/rfc9054	[COSE Hash]
IETF RFC 9360	CBOR Object Signing and Encryption (COSE): Header parameters for carrying and referencing X.509 certificates https://datatracker.ietf.org/doc/html/rfc9360	[COSE X509]
IANA COSE Assignments	CBOR Object Signing and Encryption (COSE) https://www.iana.org/assignments/cose/cose.xhtml	[IANA COSE]
NIST.SP.800-38D	Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC https://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf	[AES-GCM]
GB/T 32918.1-2016	SM2 public key cryptographic algorithm based on elliptic curves; Part 1: General	[SM2 General]
GB/T 32918.2-2016	Public key cryptographic algorithm SM2 based on elliptic curves; Part 2: Digital signature algorithm	[SM2 DSA]
GB/T 32905-2016	SM3 Cryptographic Hash Algorithm	[SM3]
GB/T 32907-2016	SM4 Block Cipher Algorithm	[SM4]

Table 1-2: Informative References

Standard / Specification	Description	Ref
GPP_SPE_001	GlobalPlatform Technology Entity Attestation Protocol Specification *To be published.*	[TPS EAT]
GP_TEN_053	GlobalPlatform Technology Cryptographic Algorithm Recommendations	[Crypto Rec]
JCA/JCE	Java Cryptographic Architecture / Java Cryptography Extension	[JCA]
OASIS PKCS #11	PKCS #11 Cryptographic Token Interface Base Specification, Version 2.40, April 2015	[PKCS #11]
GB/T 32918.3-2016	Public Key cryptographic algorithm SM2 based on elliptic curves; Part 3: Key exchange protocol	[SM2 KEP]

Standard / Specification	Description	Ref
GB/T 32918.4-2016	Public key cryptographic algorithm SM2 based on elliptic curves; Part 4: Public key encryption algorithm	[SM2 Enc]
GB/T 32918.5-2016	Public Key cryptographic algorithm SM2 based on elliptic curves; Part 5: Parameter definition	[SM2 Defs]

1.4 Terminology and Definitions

The following meanings apply to SHALL, SHALL NOT, MUST, MUST NOT, SHOULD, SHOULD NOT, and MAY in this document (refer to [RFC 2119] as amended by [RFC 8174]):

- **SHALL** indicates an absolute requirement, as does **MUST**.
- **SHALL NOT** indicates an absolute prohibition, as does **MUST NOT**.
- **SHOULD** and **SHOULD NOT** indicate recommendations.
- **MAY** indicates an option.

Note that as clarified in the [RFC 8174] amendment, lower case use of these words is not normative.

Selected terms used in this document are included in Table 1-3.

Table 1-3: Terminology and Definitions

Term	Definition
Application	In the context of this specification, an entity in the operating system that is using a TPS Service. Typically a user space application providing services to end users via a user interface.
Binding	A mapping between a Language Specific API and the TPS Client API that translates Language Specific API calls to TPS Service Protocol messages specified in a TPS Service specification, and vice versa.
Device	An end-user product that includes at least one platform.
Execution Environment	An environment that hosts and executes software. This could be a REE, with hardware hosting Android, Linux, Windows, an RTOS, or other software; it could be a Secure Element or a TEE.
Regular Execution Environment (REE)	An Execution Environment comprising at least one Regular OS and all other components of the device (IC packages, other discrete components, firmware, and software) that execute, host, and support the Regular OSes (excluding any Secure Components included in the device). From the viewpoint of a Secure Component, everything in the REE is considered untrusted, though from the Regular OS point of view there may be internal trust structures. (Formerly referred to as a *Rich Execution Environment (REE)*.) Contrast *Trusted Execution Environment (TEE).*
Regular OS	An OS executing in a Regular Execution Environment. May be anything from a large OS such as Linux down to a minimal set of statically linked libraries providing services such as a TCP/IP stack. (Formerly referred to as a *Rich OS* or *Device OS*.)

Term	Definition
Secure Component (SC)	A trusted and secure component that is able to provide enhanced security compared to the Regular Operating System. Examples include GlobalPlatform's Secure Element and Trusted Execution Environment, and Trusted Computing Group's Trusted Platform Module.
Secure Element (SE)	A tamper-resistant secure hardware component that is used in a device to provide the security, confidentiality, and multiple application environment required to support various business models. May exist in any form factor, such as embedded or integrated SE, SIM/UICC, smart card, smart microSD, etc.
TPS Client	An entity that uses the TPS Client API to discover and communicate with a TPS Service. A TPS Client can be either an Application or another TPS Service.
TPS Client API	The API defined in GlobalPlatform's TPS Client API Specification ([TPS Client API]), which enables generic mechanisms for discovering and communicating with a TPS Service.
TPS Keystore Operation	A TPS Operation accomplished via the TPS Keystore Protocol, as described in this specification.
TPS Keystore Transaction	A TPS Transaction in support of a TPS Keystore Operation.
TPS Operation	An operation that is executed by a TPS Service upon a request from a TPS Client. A TPS Operation consists of one or more TPS Transactions.
TPS Service	A service in a Secure Component, providing a service to entities in the operating system; accessed using a TPS Service Protocol that is specified in a TPS Service specification.
TPS Service Name	Uniquely identifies a TPS Service implementation.
TPS Service Protocol	A protocol that is used to communicate with the TPS Service; consists of a set of TPS Operations.
TPS Service request message	A protocol message specified by a TPS Service specification. It is constructed and sent by the TPS Client to the TPS Service using the TPS Client API.
TPS Service response message	A protocol message specified by a TPS Service specification. It is constructed and sent by the TPS Service to the TPS Client in response to a TPS Service request message.
TPS Transaction	A single exchange of messages between the TPS Client and TPS Service: a TPS Service request message created and sent by a TPS Client to a TPS Service, and a TPS Service response message created by the TPS Service and sent to the TPS Client in response to the TPS Service request message.

Term	Definition
Trusted Execution Environment (TEE)	An Execution Environment that runs alongside but isolated from Execution Environments outside of the TEE. A TEE has security capabilities and meets certain security-related requirements: It protects TEE assets against a set of defined threats which include general software attacks as well as some hardware attacks, and defines rigid safeguards as to data and functions that a program can access. There are multiple technologies that can be used to implement a TEE, and the level of security achieved varies accordingly. Contrast *Regular Execution Environment (REE).*
Trusted Platform Module (TPM)	A computer chip (microcontroller) that can securely store artifacts used to authenticate the platform. These artifacts can include passwords, certificates, or encryption keys. A TPM can also be used to store platform measurements that help ensure that the platform remains trustworthy.
UUIDv5	In this document, UUIDv5 is used to denote a name-based Universally Unique Identifier constructed using SHA-1 hashing, as described in [UUID].

1.5 Abbreviations

Table 1-4: Abbreviations

Abbreviation	Meaning
AES-CBC-MAC	AES Message Authentication Code
AES-CMAC	AES Cipher-Based Message Authentication Code
CBOR	Concise Binary Object Representation
CDDL	Concise Data Definition Language
COSE	CBOR Object Signing and Encryption
EAT	Entity Attestation Token
ECDSA	Elliptic Curve Digital Signature Algorithm
EdDSA	Edwards-Curve Digital Signature Algorithm
HKDF	HMAC-Based Extract-and-Expand Key Derivation Function
HMACs	Hash-Based Message Authentication Codes
OKP	Octet Key Pair
REE	Regular Execution Environment
SC	Secure Component
SE	Secure Element
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
TPS	Trusted Platform Services

1.6 Revision History

GlobalPlatform technical documents numbered *n*.0 are major releases. Those numbered *n*.1, *n*.2, etc., are minor releases where changes typically introduce supplementary items that do not impact backward compatibility or interoperability of the specifications. Those numbered *n*.*n*.1, *n*.*n*.2, etc., are maintenance releases that incorporate errata and clarifications; all non-trivial changes are indicated, often with revision marks.

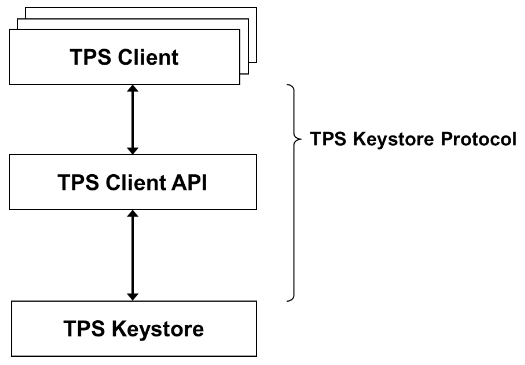
Table 1-5: Revision History

Date	Version	Description
Mar 2025	1.0	Initial Public Release

2 OVERVIEW

This specification defines a TPS Keystore Protocol for communicating with a TPS Keystore. More precisely, the TPS Keystore Protocol consists of operational instructions to a TPS Keystore implementation via the TPS Client API.

Figure 2-1: TPS Keystore Protocol



The TPS Keystore Protocol provides means for TPS Clients (i.e. Applications or other TPS Services) to communicate with the TPS Keystore and utilize the functionalities it can provide. TPS Keystore Protocol messages are conveyed between a TPS Client and the TPS Keystore using the TPS Client API.

The main functionalities of TPS Keystore include:

- Manage cryptographic keys, i.e. create, generate, or import keys, as well as manage their lifecycle in the TPS Keystore.
- Perform cryptographic operations with the keys available in the TPS Keystore.
- Provide attestations on the properties of the keys in the TPS Keystore.

A TPS Keystore conforms to TPS architecture as a TPS Service that is protected by a Secure Component (SC). An SC can be a Secure Element (SE), a Trusted Execution Environment (TEE), or another entity that provides an adequate level of security for protecting cryptographic keys and cryptographic operations performed with those keys. A TPS Keystore can be protected using two or more SCs.

This specification does not limit the environments where TPS Keystore can be implemented.

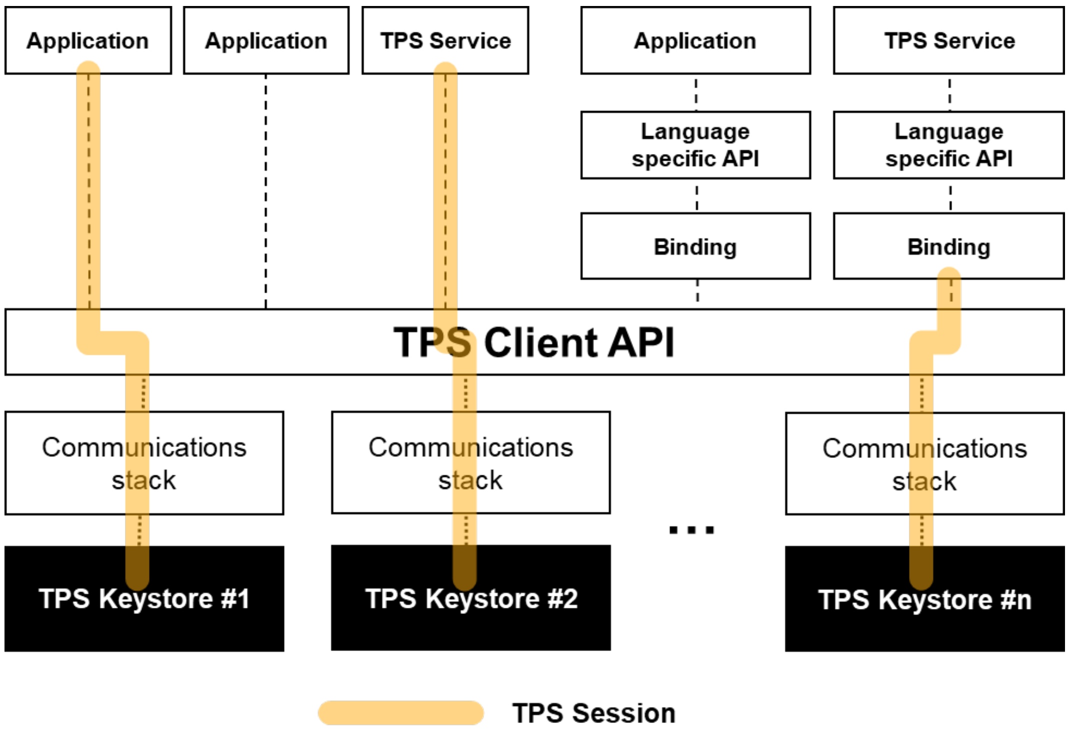
This specification fully specifies how a set of cryptographic algorithms and operations using them can be implemented in the TPS Keystore. This specification does not mandate which algorithms and operations are implemented in a specific instance of a TPS Keystore, and supports proprietary extensions. The only limitation is that implementation of functionality outside of this specification MUST use the number ranges provided for such extensions.

2.1 Architecture

2.1.1 External Architecture

Figure 2-2 outlines the relationship between TPS Keystore related components.

Figure 2-2: TPS Keystore Architecture



TPS Client API is the component used to establish a TPS Session between a TPS Client (an Application or another TPS Service) and a TPS Keystore, and to exchange TPS Keystore Protocol messages through the TPS Session. The TPS Session can be viewed as a connection or a channel between the TPS Client and the TPS Keystore through which a set of keystore operations can be executed.

A TPS Client can use a Language Specific API, such as PKCS #11 API ([PKCS #11]) or Java Cryptographic Architecture API ([JCA]), to use the cryptographic services provided by the TPS Keystore. The functions and methods of the Language Specific API are mapped to the TPS Keystore Protocol by a Binding. The Binding is responsible for converting function and method calls to TPS Keystore Protocol request messages, and sending them to the TPS Keystore using the TPS Client API. It is also responsible for decoding the incoming TPS Keystore Protocol response messages coming from the TPS Keystore via the TPS Client API, and constructing the return parameters and values for function and method calls of the Language Specific API.

A Device MAY have more than one TPS Keystore available. The TPS Client or the Binding is responsible for discovering the TPS Keystore that best meets its security requirements, as TPS Keystores available in the same Device MAY offer different levels of security services.

2.2 Security

2.2.1 Security of the TPS Keystore

The implementation of the TPS Keystore SHALL treat any input from the Regular Execution Environment as potentially malicious. A TPS Keystore SHALL assume that Applications MAY be compromised by attack or MAY be purposefully malicious.

2.2.2 Security of the Regular Operating System

In most implementations, the TPS Keystore is running in a separate operating system, i.e. within a Secure Component, which exists in parallel to the Regular OS that runs the Applications. It is important that the integration of the TPS Keystore alongside the Regular OS cannot be used to weaken the security of the Regular OS itself. The implementation of the TPS Keystore Protocol and the TPS Keystore SHALL ensure that Applications cannot use the features they expose to bypass the security sandbox used by the Regular OS to isolate processes.

2.3 Authentication

The TPS Keystore implementation authenticates the TPS Client using the session login mechanism specified in the GlobalPlatform TPS Client API specification ([TPS Client API]). The TPS Keystore implementation SHALL support at least one session login mechanism.

2.4 Access Control

The TPS Keystore implementation controls access to objects it controls. Only the creator of an object has management rights and specified usage rights to the managed object. TPS Clients SHALL be authenticated using the mechanisms specified in section 2.3.

NOTE: This version of the TPS Keystore does not support the sharing of keys; i.e., a key can be used only by the TPS Client that generated or derived the key, or that imported the key to the TPS Keystore.

2.5 TPS Service Identifiers

This section specifies the TPS Service Identifier elements that the TPS Keystore implementation will have for this version of the TPS Keystore Protocol specification. The mechanisms for selecting and generating TPS Service Identifiers are specified in [TPS Client API] section 3.2.3.

2.5.1 tps-service-name, tps-service-id, tps-service-version

All TPS Keystore implementations based on this version of the specification SHALL use the following TPS Service Identifier elements:

```
tps-service-name = "GPP TPS KEYSTORE"
tps-service-id = 1846e97d-0f5e-5cd9-b0ac-be3c5ac7799c
tps-service-version = "000000010000000000000000"          ; 1.0.0
```

2.5.2 tps-secure-component-type

The value of `tps-secure-component-type` indicates the type of security environment that supports the TPS Service. [TPS Client API] section 3.2.3.5 specifies initial values, and states that other specifications MAY define additional values.

This specification specifies additional secure component types that reflect the possibility that the TPS Keystore implementation is based on a combination of components, where a Secure Element provides the core of the security functionality, i.e., protects and operates the keys, and another component maps the TPS Keystore functionality to SE functionalities (processing the CBOR messages and mapping TPS Keystore Operations to SE functionalities).

Table 2-1: `tps-secure-component-type` Values

Secure Component	UUID Name Field	Generated UUIDv5
Combination of GlobalPlatform compliant Secure Element and Trusted Execution Environment	"GPP-SE-TEE"	`4e0f892b-9ef7-5eb6-b060-65678c4320f0`
Combination of GlobalPlatform compliant Secure Element and Regular Execution Environment (e.g., Linux, Windows, RTOS)	"GPP-SE-REE"	`ab630d91-22c3-580f-9044-b8f4ba8b2979`

2.5.3 tps-secure-component-instance

The value of `tps-secure-component-instance` is calculated as specified in [TPS Client API] section 3.2.3.6. If the value of `tps-secure-component-type` is "GPP-SE-TEE" or "GPP-SE-REE", the “Secure Element Instances” calculation is used, except that the value of `sc` is "GPP-SE-TEE" or "GPP-SE-REE", respectively.

2.5.4 tps-service-instance

The value of `tps-service-instance` is calculated as specified in [TPS Client API] section 3.2.3.7. If the value of `tps-secure-component-type` is "GPP-SE-TEE" or "GPP-SE-REE", the “Secure Element Hosted Services” calculation is used.

2.6 Key Attestation

TPS Keystore SHALL be capable of providing *key attestations* of the keys protected by the TPS Keystore. The main attributes of these key attestations are:

- Proof of key origin, where the attestation proves that the key is protected by a TPS Keystore
- Key attributes, where the attributes describe the functionalities of the key, including operations and algorithms that are allowed to be performed with the key

The attestation SHALL be signed by a key that the TPS Keystore has access to. This key is called an *attestation key*. The attestation key MAY be shared with other services in the TPS Architecture (e.g., Entity Attestation Token Service) or other entities outside of the TPS Architecture. All entities including TPS Keystore SHALL ensure that the attestation key is used only for signing attestations; e.g., the attestation key is not used to sign arbitrary data provided by an external entity.

The attestation verifying entity SHALL be able to determine that the key that is used to sign the key attestations is indeed an attestation key of a TPS Keystore. Hence, the attestation key SHALL have a certificate from a PKI system that ensures that the attestation key is only used for attestations and that it is adequately protected against theft and unauthorized use in the device hosting the TPS Keystore.

The processes and procedures to make the attestation key available for the TPS Keystore are outside the scope of this specification.

2.7 Message Encoding

TPS Keystore Protocol messages are CBOR encoded ([CBOR]) and specified in CDDL ([CDDL]).

2.8 TPS Session

A TPS Session is a session between a TPS Client and the TPS Keystore. It is initiated when the TPS Client opens a TPS Session with the TPS Keystore using the `TPSC_OpenSession` function of the TPS Client API. It is destroyed when the TPS Client uses the `TPSC_CloseSession` function of the TPS Client API to close the TPS Session with the TPS Keystore, or when the TPS Communication Stack or TPS Client API determines that the TPS Client is no longer active.

When a TPS Session is destroyed between the TPS Client and the TPS Keystore, all non-persistent keys and pending operations related to this TPS Client are destroyed and cleared in the TPS Keystore.

2.9 TPS Keystore Operation

A TPS Keystore Operation is a single operation initiated by the TPS Client with the TPS Keystore. The operation is one of the following:

- Key management operation, which manages a key in the TPS Keystore (e.g., creates a key, removes a key, etc.)
- Cryptographic operation, which executes a cryptographic operation in the TPS Keystore (e.g., calculates a hash, generates a signature, decrypts data, etc.)
- Manage end entity certificate, which associates a private key in the TPS Keystore with an end entity certificate
- Validate a PKI certificate, which validates an end entity certificate by using a list of trusted certificates in the TPS Keystore
- Manage trusted certificates, which manages the trusted certificates in the TPS Keystore (e.g., adds or removes a trusted certificate)

A TPS Keystore Operation is accomplished via one or more request-response message pairs; i.e. via one or more TPS Keystore Transactions.

2.10 TPS Keystore Transaction

A TPS Keystore Transaction is a TPS Service request-response pair initiated by the TPS Client with the TPS Keystore.

If more than one TPS Keystore Transaction is required to accomplish a TPS Keystore Operation, the stream of messages always starts with a single *init* message, followed by zero or more *update* messages, and ends with a single *finish* message. These messages are identified by including the operational phase (`op_phase`) parameter (section 3.2.45) in the messages. When the TPS Keystore receives the init message, it generates the transaction identifier (`tid`) parameter (section 3.2.44), which is returned to the TPS Client as part of the response to the init message. The TPS Client and the TPS Keystore SHALL then include the transaction identifier in all subsequent update and finish messages.

The TPS Keystore SHALL maintain a list of active transaction identifiers. A transaction identifier SHALL remain on the list until one of the following conditions is met:

- The TPS Keystore receives the finish message that includes the transaction identifier.
- The TPS Keystore receives a `TPSK_Abort` message including the transaction identifier.
- Processing of any message belonging to the transaction is not successful.
- The TPS Session from the TPS Client to the TPS Keystore is closed.

The TPS Keystore is expected to process an update or finish message if the message contains an active transaction identifier. If the transaction identifier is not active, the TPS Keystore SHALL respond with an error message.

2.11 TPS Keystore Error Handling

When a TPS Keystore encounters an error situation processing a request message sent by the TPS Client, the TPS Keystore SHALL always respond with a corresponding response message whose status SHALL NOT indicate success. The status SHOULD only indicate the rough reason for the error.

2.12 Key Identifiers

The COSE specification ([COSE Struct]) states that Applications SHALL NOT assume that key identifier (`kid`) values are unique when using the `kid` parameter in the COSE Header to locate a cryptographic key. In COSE, the key identifier is used as a hint to find the cryptographic key to process a `COSE_Messages` object (see [COSE Struct] section 3.1 and the `kid` parameter definition in section 3.2.7 of this specification). There MAY be more than one key with the same `kid` value, so all of those keys MAY need to be checked when processing the `COSE_Messages` object.

For the above reason, this specification does not mandate that the key identifier (`kid`) is unique.

The TPS Keystore specifies a unique key identifier (`ukid`) to provide an unambiguous reference to each key used in the TPS Keystore Protocol. The `ukid` SHALL be used with TPS Keystore Protocol messages while the `kid` is intended to be used in `COSE_Messages` objects as the hint of the key.

3 PARAMETERS AND MESSAGES

3.1 Data Formats and Encoding Rules

3.1.1 CBOR Map Keys

This specification uses negative integers and unsigned integers as CBOR map keys. The integers are used for compactness of encoding and easy comparison.

The presence in a CBOR map of a map key that is neither a text string nor an integer is an error. Applications can either fail processing or process messages by ignoring incorrect CBOR map keys; however, they SHALL NOT create messages with incorrect CBOR map keys.

3.1.2 Encoding

The following encoding types are used for basic data elements:

- Text string (`tstr`)
- Byte string (`bstr`)
- Integer (`int`)
- Boolean (`bool`; true/false)

Text strings SHALL be encoded as UTF-8 as specified in [UTF8]. If a text string is converted from binary data, Base64 encoding as specified in [RFC 4648] SHALL be used.

Data elements are encoded and serialized according to CBOR as specified in [CBOR].

[CBOR] section 4.2.1 describes "Core Deterministic Encoding Requirements" with three restrictions. The following restrictions SHALL be implemented for all CBOR structures specified in the specification:

- Preferred serialization SHALL be used as specified in [CBOR].
- Indefinite-length items SHALL NOT appear as specified in [CBOR].
- Maps SHALL NOT have multiple entries with the same key.

This specification does not require sorting of the keys in maps as specified in [CBOR].

3.2 Parameters

This section specifies all parameters that are used in the TPS Keystore Protocol.

3.2.1 CBOR Map Key Assignments

The following table lists all parameters specified in this specification that can be present in a TPS Message (section 3.4) and assigns a CBOR map key for each, where the key is an integer. Subsequent sections describe the parameters in more detail.

Table 3-1: TPS Keystore Parameters and Assigned CBOR Map Keys Used in Messages

Key	Name	Section	Description
`-1`	`key`	3.2.2	Unique key identifier `ukid`
`-2`	`pubkey`	3.2.2	Unique key identifier `ukid` or `TPS_COSE_Key` map specifying a public key
`-3`	`key_spec`	3.2.4	`TPS_COSE_Key` map determining key specification template for key generation and derivation
`-4`	`wrapping_key`	3.2.31	Wrapping key encoded as `key` or `pubkey`
`-5`	`wrapped_key`	3.2.32	Wrapped key that is either: • `bstr .cbor COSE_Encrypt`, where the encrypted key is a `TPS_COSE_Key`, or • `bstr` containing an encrypted key, where the encrypted key is one of the following: ○ raw key data as octets for a symmetric key ○ DER-encoded ASN.1 structure of `SubjectPublicKeyInfo` for a public key ○ DER-encoded ASN.1 structure of `PrivateKeyInfo` or `OneAsymmetricKey` for a private key
`-6`	`alg`	3.2.8	Algorithm of an operation encoded as `int`
`-7`	`iv`	3.2.15	Initialization vector or Nonce value for encryption and decryption operation encoded as `bstr`
`-8`	`partial_iv`	3.2.16	Partial initialization vector encoded as `bstr`
`-9`	`aad`	3.2.18	Additional authenticated data encoded as `bstr`
`-10`	`nonce`	3.2.19	Nonce value for a key derivation operation encoded as `bstr`
`-11`	`input`	3.2.20	Operation input encoded as `bstr`
`-12`	`output`	3.2.21	Operation output encoded as `bstr` (not including `signature` or `mac`)
`-13`	`signature`	3.2.22	Signature of data encoded as `bstr`
`-14`	`mac`	3.2.23	Message authentication code of data encoded as `bstr`
`-15`	`salt`	3.2.24	Salt value encoded as `bstr`
`-16`	`label`	3.2.25	Label value encoded as `bstr`
`-17`	`info`	3.2.26	Info value encoded as `bstr`

Key	Name	Section	Description
-18	counter	3.2.27	Counter value encoded as int
-19	tag_length	3.2.28	Tag length value encoded as int
-20	key_format	3.2.35	Key format encoded as int
-21	challenge	3.2.36	Challenge for attestation encoded as bstr
-22	key_attestation	3.2.37	Key attestation encoded as bstr .cbor COSE_Sign1
-23	key_attestation_ type	3.2.38	Key attestation type encoded as int
-24	result	3.2.39	Operation result encoded as bool
-25	key_list	3.2.41	Key list encoded as a CBOR array
-26	certificates	3.2.42	Certificate or certificate list encoded as x5chain / x5bag
-27	mid	3.2.43	Message identifier encoded as int
-28	tid	3.2.44	Transaction identifier encoded as int
-29	op_phase	3.2.45	Operational phase encoded as int
-30	status	3.2.47	Status of an operation encoded as int
-31	length	3.2.40	Length of data encoded as int
-32	hidden	3.2.46	Is key listed in the response of TPSK_ListKeys operation, encoded as bool

3.2.2 Key (key) and Public Key (pubkey)

The key parameter is the unique key identifier ukid, which is used to uniquely identify a key in the TPS Keystore.

The pubkey parameter can be either:

- ukid, which is used to uniquely identify a key in the TPS Keystore, or
- TPS_COSE_Key map, which is used to:
  - Import a key to the TPS Keystore, where key wrapping is not used.
  - Export a key from the TPS Keystore, where key wrapping is not used.
  - Use a key in a single cryptographic operation where the key will not be stored in the TPS Keystore.

In CDDL, the key and pubkey parameters are defined as:

```
key = ukid
pubkey = ukid / TPS_COSE_Key
```

3.2.3 TPS Extended COSE_Key (TPS_COSE_Key)

The `TPS_COSE_Key` map specifies parameters for a cryptographic key, and is used for:

- Importing or exporting a key, in which case the `TPS_COSE_Key` SHALL contain the key material of the key specific to the key type (`kty`). See section 3.2.6 for details.
- Generating or deriving a key, in which case the `TPS_COSE_Key` SHALL contain needed parameters to identify the kind of key to be generated or derived, including a potential allowed algorithm (`alg`) and allowed key operations (`key_ops`). See section 3.2.4 for details.

The `TPS_COSE_Key` map is extended from the `COSE_Key` map defined in [COSE Struct].

In CDDL, the `TPS_COSE_Key` is defined as:

```
TPS_COSE_Key = {
  1 => tstr / int,                 ; kty
  ? 2 => bstr,                     ; kid
  ? 3 => int,                      ; alg
  ? 4 => [+ key_op ],              ; key_ops
  ? 5 => bstr,                     ; base_iv
  ? 512 => TPS_Key_params,         ; TPS Keystore specific key parameters
  ? ec2_key_params,                ; EC key parameters, if applicable
  ? okp_key_params,                ; OKP key parameters, if applicable
  ? symmetric_key_params,          ; symmetric key parameters, if applicable
  ? rsa_key_params,                ; RSA key parameters, if applicable
  * $$tps_cose_key_ext             ; optional extended parameters
}
```

`TPS_COSE_Key` contains the following attributes associated with a key that MAY also be present in `COSE_Key`:

- 1: key type (`kty`) specifies the key type of the key.
- 2: key identifier (`kid`) indicates the key identifier of the key.
- 3: algorithm (`alg`), if present, indicates the only cryptographic algorithm that is allowed to be used with the key. If not present, any applicable cryptographic operation is allowed.
- 4: key operations (array of `key_op`), if present, indicates the cryptographic operations that are allowed to be used with the key. If not present, any applicable cryptographic operation is allowed.
- 5: base initialization vector (`base_iv`), if present, identifies the base initialization vector.

The `TPS_Key_params` map is the extension specified by this specification to the `COSE_Key` map. The `TPS_Key_params` map contains TPS Keystore specific attributes associated with a key.

- 512: TPS Keystore specific attributes, if present, are included in the CBOR map `TPS_Key_params` for the key.

The key parameters, if present, contain the key material as specified in [COSE Struct] and [COSE RSA]:

- If `kty` indicates an EC2 type key, then `ec2_key_params` SHALL be present and contain either the public key or private key material of the key.
- If `kty` indicates an OKP type key, then `okp_key_params` SHALL be present and contain the public key or private key material of the key.
- If `kty` indicates a Symmetric type key, then `symmetric_key_params` SHALL be present and contain the symmetric key material.
- If `kty` indicates an RSA type key, then `rsa_key_params` SHALL be present and contain either the public key or private key material of the key.

A TPS Keystore implementation MAY extend the `TPS_COSE_Key` with implementation specific parameters with the CDDL Socket/Plug functionality using `$$tps_cose_key_ext` definition. The implementation specific parameters SHALL NOT overlap with the parameters specified in this specification.

3.2.4 Key Specification Template (key_spec)

The `key_spec` parameter specifies a key specification template, which determines parameters for a key when it is generated or derived in the TPS Keystore.

In CDDL, the `key_spec` parameter is defined as:

```
key_spec = TPS_COSE_Key
```

When the `TPS_COSE_Key` object is used as a key specification template, it SHALL NOT contain any key material, and the parameters have the following meaning:

- `kty`: The mandatory key type, as described in section 3.2.6.
- `kid`: The optional external key identifier, which will be used by external entities to provide a hint of the key to be used when processing a `COSE_Message`. The `kid` parameter is defined in section 3.2.7.
- `alg`: The optional algorithm, which identifies the only algorithm that the key SHALL be used for. The TPS Keystore implementation SHALL allow usage of the key only with the indicated algorithm. If no algorithm is indicated, the TPS Keystore SHALL allow usage of the key with any applicable algorithm. The `alg` parameter is defined in section 3.2.8.
- `key_ops`: The optional set of key operations that are allowed to be used with the generated key. If present, the array SHALL contain at least one key operation as defined in section 3.2.9.
- `base_iv`: The optional base initialization vector parameter carries the base portion of an Initialization Vector as defined in section 3.2.17.
- `TPS_Key_params`:
  - `key_exportable`: The optional key exportable parameter indicates whether the key can be exported from the TPS Keystore. See section 3.2.29.
  - `key_lifetime`: The optional key lifetime parameter indicates the key lifetime of the key. See section 3.2.30.
  - `ukid`: The unique key identifier SHALL NOT be present.
  - `key_size`: The key size parameter indicates the size of the key in bits. (See section 3.2.34.) The key size parameter SHALL be present only if key type is an RSA key or a symmetric key.
- `ec2_key_params`, `okp_key_params`, `symmetric_key_params`, and `rsa_key_params` SHALL NOT be present.

When the `TPS_COSE_Key` object is used for importing a key, the parameters have the following meaning:

- `kty`: The mandatory key type, as described in section 3.2.6.
- `kid`: The optional external key identifier, which will be used by external entities to provide a hint of the key to be used when processing a `COSE_Message`. The `kid` parameter is defined in section 3.2.7.
- `alg`: The optional algorithm, which identifies the only algorithm that the key SHALL be used for. The TPS Keystore implementation SHALL allow usage of the key only with the indicated algorithm. If no algorithm is indicated, the TPS Keystore SHALL allow usage of the key with any applicable algorithm. The `alg` parameter is defined in section 3.2.8.
- `key_ops`: The optional set of key operations that are allowed to be used with the key. If present, the array SHALL contain at least one key operation as defined in section 3.2.9.
- `base_iv`: The optional base initialization vector parameter carries the base portion of an Initialization Vector as defined in section 3.2.17.

- `TPS_Key_params`:
  - `key_exportable`: The key exportable indication, if present, SHALL be true. See section 3.2.29.
  - `key_lifetime`: The optional key lifetime parameter indicates the key lifetime of the key. See section 3.2.30.
  - `ukid`: The unique key identifier SHALL NOT be present.
  - `key_size`: The key size parameter SHALL NOT be present. (See section 3.2.34.)
- One of the following values SHALL be present, depending on the key type value (`kty`): `ec2_key_params`, `okp_key_params`, `symmetric_key_params`, and `rsa_key_params`. The values SHALL contain either public key material only if the key is an asymmetric public key, private key material if the key is an asymmetric private key, or secret key material if the key is a symmetric key.

3.2.5 TPS Key Parameters (TPS_Key_params)

The `TPS_Key_params` parameter contains TPS Keystore specific parameters associated with a key. They are initially set by the TPS Client when generating a key or importing the plaintext non-wrapped key. When a wrapped key is imported, the TPS Key parameters MAY be included in the `wrapped_key`.

The TPS Keystore specific parameters MAY include:

- `key_exportable`: Determines if private key or a symmetric key is allowed to be exported. If not present, the default value SHALL be `false` if key is being generated, and `true` if key is being imported.
- `key_lifetime`: Determines the key lifetime of the key. If not present, the default value SHALL be `persistent`.
- `ukid`: Identifies the unique key identifier of the key. SHALL NOT be present in the TPS request messages when generating, deriving, or importing a key; otherwise SHALL be present.
- `key_size`: Indicates the size of the key in bits. SHALL be present only when generating or deriving a key; only if the key type is RSA or symm; and only if the key size is not otherwise determinable (for example, omit if the key size is specified in the algorithm identifier).
- `hidden`: Indicates whether the key is hidden, i.e., not listed in the key enumeration. Default value SHALL be `false`.
- `challenge`: Contains the provided challenge parameter when doing key attestation.
- `$$tps_key_param_ext`: A TPS Keystore implementation MAY extend `TPS_Key_params` with implementation specific parameters using `$$tps_key_param_ext`. The implementation specific parameters SHALL NOT overlap with the parameters specified in this specification.

In CDDL, the `TPS_Key_params` parameter is defined as:

```
TPS_Key_params = {
  ? 1 => key_exportable,
  ? 2 => key_lifetime,
  ? 3 => ukid,
  ? 4 => key_size,
  ? 5 => hidden,
  ? 6 => challenge,
  * $$tps_key_param_ext
}
```

3.2.6 Key Type (kty)

The kty parameter identifies the key type for a key object as specified in [COSE Struct]. Key type values are specified in [COSE Struct].

In CDDL, the kty parameter is defined as:

```
kty = okp / ec2 / rsa / symm

okp  = 1                          ; Octet Key Pair
ec2  = 2                          ; Elliptic curve with two co-ordinates
rsa  = 3                          ; RSA key
symm = 4                          ; Symmetric key
```

The kty parameter SHALL be present in all TPS_COSE_Key objects.

The TPS Keystore implementation SHALL verify that the key type is appropriate for the algorithm being processed and SHALL NOT use the key object if this is not the case.

The kty parameter can be used to determine which other key-specific parameters are present in the key object.

The normative location for standards-controlled kty values is [IANA COSE].

3.2.7 Key Identifier (kid)

The kid parameter identifies a key identifier of a key object as specified in [COSE Struct]. The kid identifies one or more keys within the TPS Keystore.

In CDDL, the kid parameter is defined as:

```
kid = bstr
```

This specification states that the kid MAY be used as an external hint of the key within the TPS Keystore. The kid is typically present in a COSE_Message, as part of the unprotected header. The TPS Client processing a COSE_Message can extract the kid, then search the TPS Keystore for keys with the same kid value. If more than one is found, the TPS Client SHALL identify the key in the TPS Keystore to be used with the COSE_Message based on the key type and other available parameters of the key, i.e., determine a key that is applicable to be used when processing the COSE_Message. If more than one key is identified after this determination, then the TPS Client MAY need to try to process the COSE_Message with each identified key until the processing is successful.

NOTE: The unique key identifier (ukid) is used in TPS Keystore Protocol messages to identify a key in the TPS Keystore uniquely. See section 2.12 for more details on the usage of kid and ukid.

3.2.8 Algorithm Identifier (alg)

The `alg` parameter identifies which algorithm is used with a key object as specified in [COSE Struct].

In CDDL, the `alg` parameter is defined as:

```
alg = int
```

The `alg` parameter is optional for a `TPS_COSE_Key` object. If it is present, the TPS Keystore implementation SHALL allow the key to be used only with the indicated algorithm and only encoded as `int`.

The normative location for the definitions of standards-controlled `alg` values is [IANA COSE].

3.2.9 Key Operation (key_op)

The `key_op` parameter identifies a cryptographic operation that is allowed to be performed with the associated key. The key operation values are specified in [COSE Struct].

In CDDL, the `key_op` parameter is defined as:

```
key_op = sign / verify / encrypt  / decrypt  / wrap  / unwrap /
         derive_key  / derive_bits / mac_create / mac_verify

sign        =  1
verify      =  2
encrypt     =  3
decrypt     =  4
wrap        =  5
unwrap      =  6
derive_key  =  7
derive_bits =  8
mac_create  =  9
mac_verify  = 10
```

The `key_op` values are used in the `key_ops` array in a `TPS_COSE_Key` object. The `key_ops` parameter is optional for a `TPS_COSE_Key` object. If `key_ops` is present, the implementation SHALL verify that:

- The `key_op` values in the `key_ops` array match the `alg` parameter of the same `TPS_COSE_Key` object, if it is present.
- The indicated key operations can be done with the type of the key identified by the `kty` of the same `TPS_COSE_Key` object.

3.2.10 Elliptic Curve (crv)

The `crv` parameter identifies the elliptic curve as specified in [COSE Struct]. In addition, SM2 curve ([SM2 General]) is supported by this specification.

The supported elliptic curves are specified in the CDDL below.

```
crv = p256 / p384 / p521 / x25519 / x448 / ed25519 / ed448 / sm2

p256        = 1
p384        = 2
p521        = 3
x25519      = 4
x448        = 5
ed25519     = 6
ed448       = 7
sm2         = -65538
```

3.2.11 Elliptic Curve Key Parameters (ec2_key_params)

The `ec2_key_params` parameter contains attributes specifying an elliptic curve key as specified in [COSE Struct]:

- -1: Curve type identifier
- -2: x coordinate of the key
- -3: y coordinate of the key
- -4: Private key, present only if the `ec2_key_params` specifies a private key

In CDDL, the `ec2_key_params` parameter is defined as:

```
ec2_key_params = (
  -1 => crv,                           ; EC identifier
  -2 => bstr,                          ; x co-ord
  -3 => bstr / bool,                   ; y co-ord
  ? -4 => bstr                         ; private key
)
```

3.2.12 Octet Key Pair Parameters (okp_key_params)

The `okp_key_params` parameter contains attributes specifying an octet key as specified in [COSE Struct]:

- -1: Curve type identifier
- -2: Public key
- -4: Private key, present only if the `okp_key_params` specifies a private key

In CDDL, the `okp_key_params` parameter is defined as:

```
okp_key_params = (
  -1 => crv,
  -2 => bstr,                       ; public key
  ? -4 => bstr                      ; private key
)
```

3.2.13 Symmetric Key Parameters (symmetric_key_params)

The `symmetric_key_params` parameter contains attributes specifying a symmetric key as specified in [COSE Struct]:

- -1: The symmetric key

In CDDL, the `symmetric_key_params` parameter is defined as:

```
symmetric_key_params = (
  -1 => bstr                        ; key
)
```

3.2.14 RSA Key Parameters (rsa_key_params)

The `rsa_key_params` parameter contains attributes specifying an RSA key.

In CDDL, the `rsa_key_params` and `rsa_crt_prime_factor` parameters are defined as:

```
rsa_key_params = (
  -1 => bstr,                       ; modulus
  -2 => bstr,                       ; public exponent
  ? -3 => bstr,                     ; private exponent
  ? -4 => bstr,                     ; first prime factor p of n
  ? -5 => bstr,                     ; second prime factor q of n
  ? -6 => bstr,                     ; first factor’s CRT exponent dP
  ? -7 => bstr,                     ; second factor’s CRT exponent dQ
  ? -8 => bstr,                     ; CRT coefficient qInv
  ? -9 => [ + rsa_crt_prime_factor ]  ; additional prime factors if any,
                                      ; starting from 3rd prime factor
)

rsa_crt_prime_factor = (
  -10 => bstr,                      ; i-th prime factor r_i
  -11 => bstr,                      ; i-th prime factor’s CRT exponent d_i
  -12 => bstr                       ; i-th prime factor’s CRT coefficient
)
```

The `rsa_key_params` parameter contains attributes specifying an RSA key as specified in [COSE RSA].

- -1: The modulus *n*
- -2: The public exponent *e*
- -3: The private exponent *d*, present only if the `rsa_key_params` specifies a private key

Optionally, the `rsa_key_params` parameter MAY contain the CRT attributes, if the `rsa_key_params` specifies a private key:

- -4: The first prime factor *p*.
- -5: The second prime factor *q*.
- -6: The first factor’s CRT exponent *dP*.
- -7: The second factor’s CRT exponent *dQ*.
- -8: The (first) CRT coefficient *qInv*.
- -9: The additional prime factors represented as one or more `rsa_crt_prime_factor` triplets. If there are more than two prime factors, then the third, the fourth, to nth factors are represent by one triplet per additional prime. If there are only two prime factors, then the additional prime factors parameter is not present in the `rsa_key_params`.

The `rsa_crt_prime_factor` parameter specifies a prime factor of an RSA key. It is a triplet specifying:

- the i-th factor *r_i*
- the i-th factor’s CRT exponent *d_i*
- the i-th factor’s CRT coefficient *t_i*

3.2.15 Initialization Vector (iv)

The `iv` parameter contains the initialization vector or the nonce value for an encryption and decryption operation.

In CDDL, the `iv` parameter is defined as:

```
iv = bstr
```

NOTE: A `nonce` parameter is also commonly used in key derivation operations. This specification uses the `nonce` parameter to pass the nonce value for the key derivation operation to the TPS Keystore, while the initialization vector (`iv`) is used to pass the nonce value for the encryption and decryption operation to the TPS Keystore.

3.2.16 Partial Initialization Vector (partial_iv)

The `partial_iv` parameter contains the partial initialization vector value for an operation. This is a COSE specific parameter and is intended to be used in conjunction with the base initialization vector `base_iv` to generate the initialization vector that is to be used for the cryptographic operation.

In CDDL, the `partial_iv` parameter is defined as:

```
partial_iv = bstr
```

3.2.17 Base Initialization Vector (base_iv)

The `base_iv` parameter contains the base portion of an Initialization Vector. This is a COSE specific parameter that MAY be associated with a COSE key. The base initialization vector is intended to be used in conjunction with the partial initialization vector `partial_iv` to generate the initialization vector that is to be used for the cryptographic operation.

In CDDL, the `base_iv` parameter is defined as:

```
base_iv = bstr
```

3.2.18 Additional Authenticated Data (aad)

The `aad` parameter contains the additional authenticated data of a cryptographic operation.

In CDDL, the `aad` parameter is defined as:

```
aad = bstr
```

3.2.19 Nonce (nonce)

The `nonce` parameter contains the nonce of a key derivation operation.

In CDDL, the `nonce` parameter is defined as:

```
nonce = bstr
```

NOTE: A `nonce` parameter is also commonly used in encryption and decryption operations. This specification uses the initialization vector (`iv`) parameter to pass the nonce value for the encryption and decryption operation to the TPS Keystore, while the `nonce` parameter is used to pass the nonce value for the key derivation operation to the TPS Keystore.

3.2.20 Input Data (input)

The `input` parameter contains the input data of a cryptographic operation.

In CDDL, the `input` parameter is defined as:

```
input = bstr
```

3.2.21 Output Data (output)

The `output` parameter contains the output data of a cryptographic operation.

In CDDL, the `output` parameter is defined as:

```
output = bstr
```

3.2.22 Signature (signature)

The `signature` parameter contains the signature value.

In CDDL, the `signature` parameter is defined as:

```
signature = bstr
```

3.2.23 MAC (mac)

The `mac` parameter contains the MAC value.

In CDDL, the `mac` parameter is defined as:

```
mac = bstr
```

3.2.24 Salt (salt)

The `salt` parameter contains the salt of a cryptographic operation.

In CDDL, the `salt` parameter is defined as:

```
salt = bstr
```

3.2.25 Label (label)

The `label` parameter contains the label of a cryptographic operation.

In CDDL, the `label` parameter is defined as:

```
label = bstr
```

3.2.26 Info (info)

The `info` parameter contains the info of a cryptographic operation.

In CDDL, the `info` parameter is defined as:

```
info = bstr
```

3.2.27 Counter (counter)

The `counter` parameter contains the counter of a cryptographic operation.

In CDDL, the `counter` parameter is defined as:

```
counter = int
```

3.2.28 Tag Length (tag_length)

The `tag_length` parameter contains the length (in bits) of the tag of a cryptographic operation.

In CDDL, the `tag_length` parameter is defined as:

```
tag_length = int
```

3.2.29 Key Exportable (key_exportable)

The `key_exportable` parameter defines whether a key can be exported from the keystore.

In CDDL, the `key_exportable` parameter is defined as:

```
key_exportable = bool
```

If `key_exportable` is `false`, a keystore implementation SHALL NOT allow the associated key object to be exported.

It is possible to change the value of the `key_exportable` attribute from `true` to `false`, but it SHALL NOT be possible to change the `key_exportable` attribute from `false` to `true`.

3.2.30 Key Lifetime (key_lifetime)

The `key_lifetime` parameter defines how long a key remains valid.

In CDDL, the `key_lifetime` parameter is defined as:

```
key_lifetime = ephemeral / persistent / immutable

ephemeral = 1
persistent = 2
immutable = 3
```

An `ephemeral` key exists for as long as its `ukid` is valid. The `ukid` can be invalidated explicitly by a `TPSK_RemoveKey` operation (see section 3.4.3). An implementation SHALL invalidate the `ukid` of an `ephemeral` key when the keystore is power cycled.

A `persistent` key exists until it is explicitly destroyed by a `TPSK_RemoveKey` operation or due to an action that causes the key storage area to be wiped, such as a factory reset. Keys defined as persistent SHALL continue to exist across power cycles.

An `immutable` key exists for the lifetime of the keystore. A form of immutable storage, such as masked ROM or fuses, is used to hold such a key, and as such the `TPSK_RemoveKey` operation is not valid for a key with this `key_lifetime`.

3.2.31 Wrapping Key (wrapping_key)

The `wrapping_key` parameter contains the key that is used to wrap a key or unwrap a wrapped key.

In CDDL, the `wrapping_key` parameter is defined as:

```
wrapping_key = key / pubkey
```

3.2.32 Wrapped Key (wrapped_key)

The `wrapped_key` parameter contains a wrapped key. The key format inside the wrapping is either a `raw_key` format or a `cose_key` format as specified in section 3.2.35.

In CDDL, the `wrapped_key` parameter is defined as:

```
wrapped_key = wrapped_raw_key / wrapped_cose_key

wrapped_raw_key = bstr
wrapped_cose_key = bstr .cbor COSE_Encrypt
```

The `wrapped_cose_key` is a `COSE_Encrypt0` object, which contains an encrypted `TPS_COSE_Key` as specified in section 3.2.3, including the key parameters of the associated key values.

The `wrapped_raw_key` is a byte string that contains the wrapped raw key as specified in section 3.2.35.

The key wrapping and unwrapping procedure is described in section 4.10.

3.2.33 Unique Key Identifier (ukid)

The `ukid` parameter contains a unique key identifier.

In CDDL, the `ukid` parameter is defined as:

```
ukid = bstr
```

The `ukid` parameter is a unique key identifier generated by the TPS Keystore, which can be used to uniquely reference a key in the TPS Keystore.

The TPS Keystore implementation SHALL make sure that the `ukid` is unique in the TPS Keystore when it is generated. For instance, the `ukid` can be the digest of the key values and a random salt value.

This specification states that the key identifier (`kid`) specified in [COSE Struct] is used as a hint to help identify a key in the TPS Keystore. See sections 2.12 and 3.2.7 for more details.

3.2.34 Key Size (key_size)

The `key_size` parameter contains the `key_size` of the key.

In CDDL, the `key_size` parameter is defined as:

```
key_size = int
```

The `key_size` parameter is used in defining an RSA key or a symmetric key, where the `key_size` indicates the size of the key in bits.

3.2.35 Key Format (key_format)

The key_format parameter determines the format of an exported key. The key_format parameter defines the format of the key, i.e., how it is encoded. The exported key is also wrapped (encrypted) if the key is a symmetric key and an asymmetric private key. A public key is now wrapped.

In CDDL, the key_format parameter is defined as:

```
key_format = raw_key / cose_key

raw_key = 1
cose_key = 2
```

The raw_key format means that the key is in the raw format:

- For symmetric key, the key data is the raw key bytes of the symmetric key.
- For asymmetric private key, the key data is either in PrivateKeyInfo or OneAsymmetricKey format.
- For asymmetric public key, the key data is in SubjectPublicKeyInfo format.

The cose_key format means that the key is in the TPS_COSE_Key format as specified in section 3.2.3.

3.2.36 Challenge (challenge)

The challenge parameter contains the challenge that is to be included in the key attestation.

In CDDL, the challenge parameter is defined as:

```
challenge = bstr
```

3.2.37 Key Attestation (key_attestation)

The key_attestation parameter contains the key attestation of a private key that is present in the TPS Keystore. The key attestation data SHALL contain a TPS_COSE_Key structure of the attested key. For key attestation procedure, please refer to section 4.11.

In CDDL, the key_attestation parameter is defined as:

```
key_attestation = bstr .cbor COSE_Sign1
```

The key_attestation parameter is a COSE_Sign1 object, which contains the key attestation as the payload.

3.2.38 Key Attestation Type (key_attestation_type)

The `key_attestation_type` parameter identifies the type of attestation.

This specification specifies one key attestation type: `tps_key_attestation`, where the key attestation format is the `TPS_COSE_Key` structure of the attested key. For key attestation procedure, please see section 4.11.

In CDDL, the `key_attestation_type` parameter is defined as:

```
key_attestation_type = tps_key_attestation

tps_key_attestation = 1
```

3.2.39 Result (result)

The `result` parameter indicates the result of the operation as a Boolean value.

In CDDL, the `result` parameter is defined as:

```
result = bool
```

3.2.40 Length of Data (length)

The `length` parameter indicates the length (or the expected length) of data in a response.

In CDDL, the `length` parameter is defined as:

```
length = int
```

3.2.41 Key List (key_list)

The `key_list` parameter is a list of one or more `key` objects in a CBOR array, where each `key` object is a `TPS_COSE_Key` object.

In CDDL, the `key_list` parameter is defined as:

```
key_list = [ + TPS_COSE_Key ]
```

3.2.42 Certificates (certificates)

The `certificates` parameter contains either:

- a certificate chain of X.509 certificates (`x5chain`) ([COSE X509]), or
- an unordered bag of X.509 certificates (`x5bag`) ([COSE X509]).

If a single certificate is in the `certificates` parameter, it is placed in a CBOR byte string. If multiple certificates are in the `certificates` parameter, a CBOR array of byte strings is used, with each certificate in its own byte string.

In CDDL, the `certificates` parameter is defined as:

```
certificates = x5chain / x5bag

; From [COSE X509]
x5chain = COSE_X509
x5bag = COSE_X509
COSE_X509 = bstr / [ 2*certs: bstr ]
```

A certificate chain (`x5chain`) contains an ordered array of X.509 certificates. The certificates are to be ordered starting with the certificate containing the end-entity key followed by the certificate which signed it and so on.

A certificate bag (`x5bag`) contains a bag of X.509 certificates. The certificates in this parameter are unordered and MAY include self-signed certificates.

3.2.43 Message Identifier (mid)

The `mid` parameter contains a message identifier that can be used to associate a TPS Keystore request and response.

In CDDL, the `mid` parameter is defined as:

```
mid = int
```

If the TPS Client generates a message identifier and includes it in a request message sent to the TPS Keystore, then the TPS Keystore SHALL include the received message identifier in the response.

3.2.44 Transaction Identifier (tid)

Selected messages, such as `TPSK_Sign` and `TPSK_Encrypt`, require two or more request-response message pairs, as discussed in section 3.3.2. The transaction identifier is used to associate the message pairs.

In CDDL, the `tid` parameter is defined as:

```
tid = int
```

TPS Keystore generates the transaction identifier when it receives an init request message and has processed the message successfully. The transaction identifier is returned to the TPS Client as part of the init response message. All subsequent update messages and the finish request and response messages SHALL contain the transaction identifier.

If TPS Client needs to cancel an operation after the init request message but before the finish request message, the `TPSK_Abort` message SHALL be sent to cancel the ongoing operation.

3.2.45 Operation Phase (op_phase)

The `op_phase` parameter, if present, specifies the phase of an operation. The default value is `op_oneshot`, which indicates that the operation consists of a single request-response message pair; i.e. a single TPS Keystore Transaction.

If applicable, an operation can be divided into three phases, as discussed in section 3.3.2. `op_init` is used to initialize the operation, and typically contains the operation specific setup details. `op_update` is used to perform a part of the operation, typically providing input data to the operation, and optionally producing output data from the operation. `op_finish` is used to finalize the operation, typically producing the output data from the operation.

In CDDL, the `op_phase` parameter is defined as:

```
op_phase = op_oneshot / op_init / op_update / op_finish

op_oneshot = 0                      ; default,
op_init = 1                         ; init message of a transaction
op_update = 2                       ; update message of a transaction
op_finish = 3                       ; finish message of a transaction
```

3.2.46 Hidden (hidden)

The `hidden` parameter defines whether a key is listed when executing the `TPSK_ListKeys` operation.

In CDDL, the `hidden` parameter is defined as:

```
hidden = bool
```

If `hidden` is `true`, a TPS Keystore implementation SHALL NOT include the key in the list of keys when the `TPSK_ListKeys` command is received and the `TPSK_ListKeysResponse` is prepared.

The `hidden` attribute is set during key generation and cannot be changed afterwards.

3.2.47 Operation Status (status)

The status parameter indicates the status of an operation.

In CDDL, the status parameter is defined as:

```
status = (
  SUCCESS /
  IO_ERROR /
  NOT_SUPPORTED /
  INVALID_ARGUMENT /
  BAD_STATE /
  NOT_ALLOWED /
  GENERAL_FAILURE
)

SUCCESS = 0
IO_ERROR = -1
NOT_SUPPORTED = -2
INVALID_ARGUMENT = -3
BAD_STATE = -4
NOT_ALLOWED = -5
GENERAL_FAILURE = -254
```

Table 3-2: Meaning of Status Values

Status Value in Response	Meaning
SUCCESS	The operation requested in the corresponding request was successful.
IO_ERROR	The operation failed due to an unspecified I/O error.
NOT_SUPPORTED	Part or all of the functionality of the requested operation is not supported by the TPS Keystore.
INVALID_ARGUMENT	One or more arguments in the request are invalid.
BAD_STATE	The TPS Keystore is in a bad state.
NOT_ALLOWED	The requested operation is not allowed.
GENERAL_FAILURE	The TPS Keystore has suffered a general failure.

If the status is anything other than SUCCESS, the requested operation in the TPS Keystore is aborted.

3.3 Common Message Parameters

3.3.1 Common Parameters

Description

Every TPS Keystore request message MAY include a message identifier. If one is provided in the request message, it will be returned in the response message, which also includes the status of the requested operation.

Request

```
TPSK_CommonRequestParams = (
   ? -27 => mid        ; message identifier
)
```

Request parameters

- `mid`: The optional message identifier set by the TPS Client.

Response

```
TPSK_CommonResponseParams = (
  ? -27 => mid         ; message identifier
  -30 => status      ; status of the corresponding request
)
```

Response parameters

- `mid`: The message identifier from the corresponding request message, if one was provided.
- `status`: Indicates the status of the requested operation.

Implementation notes

3.3.2 Transaction Parameters

Description

For each TPS Keystore Operation that requires more than one TPS Keystore Transaction, the messages SHALL occur in the following order:

- *init* request message, which SHALL be the first message of the transaction, and a corresponding init response message
- *update* request message, which MAY be used zero or more times between the init and finish request messages, and a corresponding update response message for each
- *finish* request message, and a corresponding finish response message, which SHALL be the last message of the transaction

A TPS Keystore Transaction MAY be aborted using the `TPSK_Abort` message at any time between the init request message and the finish request message.

If any init, update, or finish response message indicates `status` other than `SUCCESS`, the TPS Keystore SHALL abort the TPS Keystore Transaction and the TPS Client SHALL mark the related transaction aborted.

The TPS Keystore Transaction parameters are conveyed from the TPS Client to the TPS Keystore as part of `TPSK_OpPhaseRequestParams`:

Request

```
TPSK_OpPhaseRequestParams = (
  ? -28 => tid,           ; transaction identifier
  ? -29 => op_phase       ; operational phase
)
```

Request parameters

- `tid`: The transaction identifier.
- `op_phase`: The operational phase.

Response

```
TPSK_OpPhaseResponseParams = (
  ? -28 => tid            ; transaction identifier
)
```

Response parameters

- `tid`: The transaction identifier.

The TPS Keystore Transaction parameters SHALL be as listed below.

Table 3-3: TPS Keystore Transaction Parameters

Message	Parameters			Notes
init request message	`op_phase`	M	Operational phase: `op_init`	Omit `tid`
init response message	`tid`	M	Transaction identifier	
update request message	`tid`	M		
	`op_phase`	M	`op_update`	
update response message	`tid`	M		
finish request message	`tid`	M		
	`op_phase`	M	`op_finish`	
finish response message	`tid`	M		

Implementation notes

3.4 Messages

As with all TPS-defined CBOR messaging, messages are tagged such that the lowest four digits are the message number and higher digits indicate the specification number.

In this specification, message tags 50000 to 50999 are reserved.

3.4.1 TPS_GetFeatures_Rsp

Description

As specified in [TPS Client API] section 5.2.2, the TPS Keystore implementation SHALL respond with the following information in the `TPS_GetFeatures_Rsp` message.

Response parameters

- The `svc_name` parameter SHALL contain the `tps-service-name` as specified in section 2.5.
- The array of `login_method` parameters SHALL contain the supported session login methods for the TPS Keystore implementation as specified in section 2.3. The TPS Keystore implementation SHALL support at least one session login method.
- The array of `profile_name` parameters SHALL contain the named configurations supported by the TPS Keystore implementation. This specification names configurations in section 5.
- The `$$svc_features` parameter SHALL NOT be used for this version of TPS Keystore Protocol specification.

3.4.2 TPSK_GenerateKey

Description

Generate a new key in the TPS Keystore; see section 4.3 for details.

Request

```
TPSK_GenerateKeyRequest = #6.50001 ({
  TPSK_CommonRequestParams,
  -3 => key_spec           ; key_spec
})
```

Request parameters

- key_spec: The key generation parameters.

Response

```
TPSK_GenerateKeyResponse = #6.50002 ({
  TPSK_CommonResponseParams,
  ? -1 => key              ; ukid of the generated key
})
```

Response parameters

- key: The unique key identifier of the generated key, if key generation was successful.

Implementation notes

The implementer SHOULD ensure that the implementation has a random source of such quality as to securely and unpredictably generate key material.

3.4.3 TPSK_ChangeKey

Description

Change the properties of an existing key from the TPS Keystore. The properties of the key SHALL only be further restricted.

Request

```
TPSK_ChangeKeyRequest = #6.50003 ({
  TPSK_CommonRequestParams,
  -1 => key                ; ukid of the key to be changed
  -3 => key_spec           ; new properties of the key
})
```

Request parameters

- `key`: The `ukid` of the key to be changed.
- `key_spec`: The `key_spec` containing the new properties of the key. The properties in the `key_spec` that are allowed to be further restricted are as follows:
  - `kid`: Can be changed to any other value.
  - `key_ops`: If `key_ops` has not been specified for the key, an array with meaningful `key_op` value(s) can be set. If `key_ops` has been specified for the key, `key_op` values can only be removed from the array, and they cannot be added.

Response

```
TPSK_RemoveKeyResponse = #6.50004 ({
  TPSK_CommonResponseParams
})
```

Response parameters

- No response specific parameters.

Implementation notes

3.4.4 TPSK_RemoveKey

Description

Remove a key from the TPS Keystore.

Request

```
TPSK_RemoveKeyRequest = #6.50005 ({
  TPSK_CommonRequestParams,
  -1 => key              ; ukid of the key to be removed
})
```

Request parameters

- `key`: The `ukid` of the key to be removed.

Response

```
TPSK_RemoveKeyResponse = #6.50006 ({
  TPSK_CommonResponseParams
})
```

Response parameters

- No response specific parameters.

Implementation notes

3.4.5 TPSK_WrapKey

Description

Wrap and export a key from the TPS Keystore; see section 4.10 for details.

Request

```
TPSK_WrapKeyRequest = #6.50007 ({
  TPSK_CommonRequestParams,
  -1 => key,             ; ukid of the key to be exported
  -4 => wrapping_key,    ; ukid of the wrapping key
  -6 => alg,             ; wrapping algorithm
  ? -7 => iv,            ; initialization vector
  ? -9 => aad,           ; additional authenticated data
  ? -19 => tag_length,   ; tag length
  -20 => key_format      ; key format to be used
})
```

Request parameters

- `key`: The `ukid` of the key to be exported.
- `wrapping_key`: The `ukid` of the key to be used to wrap the exported key.
- `alg`: The key wrapping algorithm.
- `iv`: If present, the initialization vector.
- `aad`: If present, the additional authenticated data.
- `tag_length`: If present, the tag length. The tag value SHALL be returned as part of the wrapped key data in the `TPSK_EncryptResponse`.
- `key_format`: Key format to be used in key wrapping.

Response

```
TPSK_WrapKeyResponse = #6.50008 ({
  TPSK_CommonResponseParams,
  ? -5 => wrapped_key        ; the wrapped key
})
```

Response parameters

- `wrapped_key`: The wrapped key as specified in section 3.2.32, if successfully created. If the wrapping process produced a tag value, it SHALL be present in the end of the wrapped key data.

Implementation notes

When exporting an RSA private key, note that the `TPS_COSE_Key` structure does not contain the public key information. With many Elliptic Curve based public key schemes it is possible to determine the public key with knowledge of the private key, but with RSA this is not generally possible. Users can export the corresponding public key if there is any possibility that this MAY be required.

3.4.6 TPSK_ExportPublicKey

Description

Export the public key part of a key in the TPS Keystore.

Request

```
TPSK_ExportPublicKeyRequest = #6.50009 ({
  TPSK_CommonRequestParams,
  -1 => key            ; ukid of the public key to be exported
})
```

Request parameters

- `key`: The `ukid` of the key whose public key part is to be exported.

Response

```
TPSK_ExportPublicKeyResponse = #6.50010 ({
  TPSK_CommonResponseParams,
  ? -1 => key          ; the exported public key as TPS_COSE_Key
})
```

Response parameters

- `key`: The `TPS_COSE_Key` object, if export was successful, with public key values of the key.

Implementation notes

3.4.7 TPSK_UnwrapKey

Description

Import and unwrap a wrapped key into the TPS Keystore; see section 4.10 for details.

Request

```
TPSK_UnwrapKeyRequest = #6.50011 ({
  TPSK_CommonRequestParams,
  ? -3 => key_spec                    ; key_spec for the wrapped key
  ? -4 => wrapping_key,               ; ukid of wrapping key for wrapped key
  ? -5 => wrapped_key,                ; wrapped key to be imported
  ? -6 => alg,                        ; key wrapping algorithm
  ? -7 => iv,            ; initialization vector
  ? -9 => aad,           ; additional authenticated data
  ? -19 => tag_length,   ; tag length
})
```

Request parameters

- `key_spec`: The key attributes for the wrapped key. Values in `key_spec` SHALL NOT override values contained within the wrapped key if the wrapped key contains the key in `TPS_COSE_Key` format.
- `wrapping_key`: The `ukid` of the wrapping key in the TPS Keystore.
- `wrapped_key`: The wrapped key to be imported. If the wrapping process produced a tag value, it SHALL be present in the end of the wrapped key data.
- `alg`: The key wrapping algorithm.
- `iv`: If present, the initialization vector.
- `aad`: If present, the additional authenticated data.
- `tag_length`: If present, the tag length. The tag value SHALL be present in the end of the wrapped key data.

Response

```
TPSK_UnwrapKeyResponse = #6.50012 ({
  TPSK_CommonResponseParams,
  ? -1 => key            ; ukid of the imported key
})
```

Response parameters

- `key`: The `ukid` of the unwrapped key in the TPS Keystore, if key unwrapping was successful.

Implementation notes

3.4.8 TPSK_ImportKey

Description

Import a key into the TPS Keystore.

Request

```
TPSK_ImportKeyRequest = #6.50013 ({
  TPSK_CommonRequestParams,
  -3 => key_spec         ; key to be imported
})
```

Request parameters

- `key_spec`: A `TPS_COSE_Key` object with key values as specified in section 3.2.4.

Response

```
TPSK_ImportKeyResponse = #6.50014 ({
  TPSK_CommonResponseParams,
  ? -1 => key            ; ukid of the imported key
})
```

Response parameters

- `key`: The `ukid` of the imported key, if key import was successful.

Implementation notes

3.4.9 TPSK_AttestKey

Description

Generate an attestation for a key in the TPS Keystore. The attestation SHALL be generated only for keys that have been generated in the TPS Keystore and that are not exportable. See section 4.11 for details.

Request

```
TPSK_AttestKeyRequest = #6.50015 ({
  TPSK_CommonRequestParams,
  -1 => key,                        ; ukid of key to be attested
  -21 => challenge,                 ; challenge
  -23 => key_attestation_type       ; requested attestation type
})
```

Request parameters

- `key`: The `ukid` of the key to be attested.
- `challenge`: The challenge to be included in the attestation.
- `key_attestation_type`: The attestation type that is requested.

Response

```
TPSK_AttestKeyResponse = #6.50016 ({
  TPSK_CommonResponseParams,
  ? -22 => key_attestation,   ; the attestation
  ? -26 => certificates       ; the certificate chain of the attestation key
})
```

Response parameters

- `key_attestation`: The attestation of the key, if successfully created.
- `certificates`: The certificate chain of the key that was used to sign the attestation

Implementation notes

3.4.10 TPSK_AgreeKey

Description

Perform key agreement; see section 4.9 for details.

Request

```
TPSK_AgreeKeyRequest = #6.50017 ({
  TPSK_CommonRequestParams,
  -1 => key,                      ; ukid of private key
  -2 => pubkey,                   ; public key
  -3 => key_spec,                 ; key spec for the agreed key
  ? -6 => alg,                    ; algorithm of KDF
  ? -10 => nonce,                 ; nonce for KDF
  ? -15 => salt,                  ; salt for KDF
  ? -17 => info                   ; info for KDF
})
```

Request parameters

- `key`: The `ukid` of the private key to be used in key agreement.
- `pubkey`: Either the `ukid` or the `TPS_COSE_Key` of the public key to be used in key agreement.
- `key_spec`: The key spec for the agreed key in `TPS_COSE_Key` format.
- `alg`: If present, the algorithm of the key derivation function to be used.
- `nonce`: If present, the nonce for the key derivation function.
- `salt`: If present, the salt for the key derivation function.
- `info`: If present, the info for the key derivation function.

Response

```
TPSK_AgreeKeyResponse = #6.50018 ({
  TPSK_CommonResponseParams,
  ? -1 => key              ; ukid of the agreed key
})
```

Response parameters

- `key`: The `ukid` of the agreed key, if key agreement was successful.

Implementation notes

3.4.11 TPSK_Hash

Description

Compute the digest of specified data.

Request

```
TPSK_HashRequest = #6.50019 ({
  TPSK_CommonRequestParams,
  TPSK_OpPhaseRequestParams,
  ? -6 => alg,                ; algorithm,
  ? -11 => input,             ; input data
})
```

Request parameters

- `alg`: The digest algorithm to be used.
- `input`: The data to be digested.

Response

```
TPSK_HashResponse = #6.50020 ({
  TPSK_CommonResponseParams,
  TPSK_OpPhaseResponseParams,
  ? -12 => output             ; output data (hash)
})
```

Response parameters

- `output`: The calculated digest, included in the finish response message if successfully created.

TPS Keystore Transaction parameter usage

The TPS Keystore SHALL keep state between request-response message pairs. The digest returned in the finish response message SHALL be calculated over the whole content received in multiple request messages based on the digest algorithm specified in the init request message.

The TPS Keystore Transaction parameters for `TPSK_Hash` SHALL be as listed below.

Table 3-4: `TPSK_Hash` TPS Keystore Transaction Parameters

Message	Parameters			Notes
init request message	`alg`	M	Identifies digest algorithm	
	`input`	O	Contains first part of content	
init response message			*no response specific parameters*	
update request message	`input`	M	Contains next part of content	Omit `alg`
update response message			*no response specific parameters*	
finish request message	`input`	O	Contains last part of content	Omit `alg`
finish response message	`output`	M	Returns calculated digest	

Implementation notes

3.4.12 TPSK_DeriveKey

Description

Derive a key from a key; see section 4.4 for details.

Request

```
TPSK_DeriveKeyRequest = #6.50021 ({
  TPSK_CommonRequestParams,
  -1 => key,                  ; key
  -3 => key_spec,              ; key spec for the derived key
  -6 => alg,                  ; key derivation algorithm
  ? -10 => nonce,             ; nonce
  ? -15 => salt,              ; salt
  ? -17 => info               ; info
})
```

Request parameters

- `key`: The `ukid` or the `TPS_COSE_Key` of the key that the derived key is to be derived from.
- `key_spec`: The key spec for the derived key.
- `alg`: The key derivation algorithm.
- `nonce`: If present, the nonce for the key derivation.
- `salt`: If present, the salt for the key derivation.
- `info`: If present, the info for the key derivation.

Response

```
TPSK_DeriveKeyResponse = #6.50022 ({
  TPSK_CommonResponseParams,
  ? -1 => key             ; ukid of the derived key
})
```

Response parameters

- `key`: The `ukid` of the derived key, if key derivation was successful.

Implementation notes

3.4.13 TPSK_Sign

Description

Generate a signature with a key; see section 4.6 for details.

Request

```
TPSK_SignRequest = #6.50023 ({
  TPSK_CommonRequestParams,
  TPSK_OpPhaseRequestParams,
  ? -1 => key,                ; ukid of the signing key
  ? -6 => alg,                ; signature algorithm
  ? -11 => input,             ; input data (to be signed)
  ? -17 => info               ; optional additional parameter
})
```

Request parameters

- `key`: The `ukid` of the signing key.
- `alg`: The signature algorithm.
- `input`: The content to be signed.

Response

```
TPSK_SignResponse = #6.50024 ({
  TPSK_CommonResponseParams,
  TPSK_OpPhaseResponseParams,
  ? -13 => signature         ; the signature
})
```

Response parameters

- `signature`: The signature, included in the finish response message if signing was successful.

TPS Keystore Transaction parameter usage

The TPS Keystore SHALL keep state between request-response message pairs. The signature returned in the finish response message SHALL be calculated over the whole content received in multiple request messages based on the signature algorithm and the signing key specified in the init request message.

The TPS Keystore Transaction parameters for `TPSK_Sign` SHALL be as listed below.

Table 3-5: `TPSK_Sign` TPS Keystore Transaction Parameters

Message	Parameters			Notes
init request message	`key`	M	Identifies signing key	
	`alg`	M	Identifies signature algorithm	
	`input`	O	Contains first part of content	
	`info`	O	Optional additional parameter	
init response message			*no response specific parameters*	
update request message	`input`	M	Contains next part of content	Omit `key`, `alg`, `info`
update response message			*no response specific parameters*	
finish request message	`input`	O	Contains last part of content	Omit `key`, `alg`, `info`
finish response message	`signature`	M	Returns calculated signature	

Implementation notes

3.4.14 TPSK_Verify

Description

Verify signature with a key; see section 4.6 for details.

Request

```
TPSK_VerifyRequest = #6.50025 ({
  TPSK_CommonRequestParams,
  TPSK_OpPhaseRequestParams,
  ? -2 => pubkey,                 ; verification key
  ? -6 => alg,                    ; signature algorithm
  ? -11 => input,                 ; input data (to be signed)
  ? -17 => info,                  ; optional additional parameter
  ? -13 => signature              ; signature
})
```

Request parameters

- `pubkey`: The `ukid` or the `TPS_COSE_Key` of the verification key.
- `alg`: The signature algorithm.
- `input`: The content to be signed.
- `signature`: The signature.

Response

```
TPSK_VerifyResponse = #6.50026 ({
  TPSK_CommonResponseParams,
  TPSK_OpPhaseResponseParams,
  ? -24 => result         ; result of signature verification
})
```

Response parameters

- `result`: The verification result, included in the finish response message.

TPS Keystore Transaction parameter usage

The TPS Keystore SHALL keep state between request-response message pairs. The signature to be verified is sent in the finish request message, and the signature verification status is returned in the finish response message. Verification SHALL be over the whole content received in multiple request messages based on the signature algorithm and the verification key specified in the init request message.

The TPS Keystore Transaction parameters for `TPSK_Verify` SHALL be as listed below.

Table 3-6: `TPSK_Verify` TPS Keystore Transaction Parameters

Message	Parameters			Notes
init request message	`key`	M	Identifies signing key	Omit `signature`
	`alg`	M	Identifies signature algorithm	
	`input`	O	Contains first part of content	
	`info`	O	Optional additional parameter	
init response message			*no response specific parameters*	
update request message	`input`	M	Contains next part of content	Omit `key, alg, signature, info`
update response message			*no response specific parameters*	
finish request message	`input`	O	Contains last part of content	Omit `key, alg, info`
	`signature`	M	Signature to be verified	
finish response message	`result`	M	Returns status of verification result	

Implementation notes

3.4.15 TPSK_Encrypt

Description

Encrypt data with a key; see section 4.8 for details.

Request

```
TPSK_EncryptRequest = #6.50027 ({
  TPSK_CommonRequestParams,
  TPSK_OpPhaseRequestParams,
  ? -1 => key,            ; ukid of symmetric encryption key
  ? -2 => pubkey,         ; asymmetric public key
  ? -6 => alg,            ; encryption algorithm
  ? -7 => iv,             ; initialization vector
  ? -8 => partial_iv,     ; partial initialization vector
  ? -9 => aad,            ; additional authenticated data
  ? -16 => label,         ; label
  ? -19 => tag_length,    ; tag length
  ? -11 => input          ; data to be encrypted
})
```

Request parameters

- `key`: The `ukid` of the symmetric encryption key. SHALL be used if the key is a symmetric key.
- `pubkey`: The RSA public key referenced with `ukid` or given as a `TPS_COSE_Key`. SHALL be used if the encryption key is RSA public key.
- `alg`: The encryption algorithm.
- `iv`: If present, the initialization vector. SHALL be used if the key does not have the base initialization vector set, and the encryption operation requires initialization vector.
- `partial_iv`: If present, the partial initialization vector. SHALL be used only if the key has the base initialization vector set, and the encryption operation requires initialization vector.
- `aad`: If present, the additional authenticated data.
- `label`: If present, the label.
- `tag_length`: If present, the tag length. The tag value SHALL be returned as part of the output parameter of the `TPSK_EncryptResponse`.
- `input`: The content to be encrypted.

Response

```
TPSK_EncryptResponse = #6.50028 ({
  TPSK_CommonResponseParams,
  TPSK_OpPhaseResponseParams,
  ? -12 => output         ; encrypted data
})
```

Response parameters

- `output`: The encrypted data, if encryption was successful. If the encryption produces a tag value, it is present in the end of the output data.

TPS Keystore Transaction parameter usage

The TPS Keystore SHALL keep state between request-response message pairs. The combined encrypted data received in subsequent response messages from the TPS Keystore SHALL be the result of the encryption of data received in subsequent request messages by the TPS Keystore according to the encryption algorithm and encryption algorithm parameters.

The TPS Keystore Transaction parameters for `TPSK_Encrypt` SHALL be as listed below.

Table 3-7: `TPSK_Encrypt` TPS Keystore Transaction Parameters

Message	Parameters			Notes
init request message	`key / pubkey`	M	Identifies encryption key	
	`alg`	M	Identifies encryption algorithm	
	`iv`	C		Each SHALL be present if applicable for the encryption algorithm
	`partial_iv`	C		
	`aad`	C		
	`label`	C		
	`tag_length`	C		
	`input`	O	Contains first part of content	
init response message	`output`	O	Contains first part of encrypted data	
update request message	`input`	M	Contains next part of content	Omit `key`, `alg`, `iv`, `partial_iv`, `aad`, `label`, `tag_length`
update response message	`output`	O	Contains next part of encrypted data	
finish request message	`input`	O	Contains last part of content	Omit `key`, `alg`, `iv`, `partial_iv`, `aad`, `label`, `tag_length`
finish response message	`output`	M	Contains last part of encrypted data	

Implementation notes

3.4.16 TPSK_Decrypt

Description

Decrypt data with a key; see section 4.8 for details.

Request

```
TPSK_DecryptRequest = #6.50029 ({
  TPSK_CommonRequestParams,
  TPSK_OpPhaseRequestParams,
  ? -1 => key,            ; ukid of encryption key
  ? -6 => alg,            ; encryption algorithm
  ? -7 => iv,             ; initialization vector
  ? -8 => partial_iv,     ; partial initialization vector
  ? -9 => aad,            ; additional authenticated data
  ? -16 => label,         ; label
  ? -19 => tag_length,    ; tag length
  ? -11 => input          ; encrypted data
})
```

Request parameters

- `key`: The `ukid` of the encryption key. This key is either a symmetric key or RSA key pair depending on the encryption algorithm and key type.
- `alg`: The encryption algorithm.
- `iv`: If present, the initialization vector. SHALL be used if the key does not have the base initialization vector set, and the decryption operation requires initialization vector.
- `partial_iv`: If present, the partial initialization vector. SHALL be used only if the key has the base initialization vector set, and the decryption operation requires initialization vector.
- `aad`: If present, the additional authenticated data.
- `label`: If present, the label.
- `tag_length`: If present, the tag length. The tag value SHALL be present in the end of the input data.
- `input`: The encrypted data to be decrypted. If the encryption process produced a tag value, it SHALL be present in the end of the input data.

Response

```
TPSK_DecryptResponse = #6.50030 ({
  TPSK_CommonResponseParams,
  TPSK_OpPhaseResponseParams,
  ? -12 => output         ; decrypted plain data
})
```

Response parameters

- `output`: The plain text data, if decryption was successful.

TPS Keystore Transaction parameter usage

The TPS Keystore SHALL keep state between request-response message pairs. The combined decrypted (plain text) data received in subsequent response messages from the TPS Keystore SHALL be the result of the decryption of the encrypted data received in subsequent request messages by the TPS Keystore according to the encryption algorithm and encryption algorithm parameters.

The TPS Keystore Transaction parameters for `TPSK_Decrypt` SHALL be as listed below.

Table 3-8: `TPSK_Decrypt` TPS Keystore Transaction Parameters

Message	Parameters			Notes
init request message	`key`	M	Identifies the encryption key	
	`alg`	M	Identifies the encryption algorithm	
	`iv`	C		Each SHALL be present if applicable for the encryption algorithm
	`partial_iv`	C		
	`aad`	C		
	`label`	C		
	`tag_length`	C		
	`input`	O	Contains first part of encrypted content	
init response message	`output`	O	Contains first part of decrypted data	
update request message	`input`	M	Contains next part of encrypted content	Omit `key`, `alg`, `iv`, `partial_iv`, `aad`, `label`, `tag_length`
update response message	`output`	O	Contains next part of decrypted data	
finish request message	`input`	O	Contains last part of encrypted content	Omit `key`, `alg`, `iv`, `partial_iv`, `aad`, `label`, `tag_length`
finish response message	`output`	M	Contains last part of decrypted data	

Implementation notes

3.4.17 TPSK_SignMAC

Description

Generate a MAC of data with a key; see section 4.7 for details.

Request

```
TPSK_SignMACRequest = #6.50031 ({
  TPSK_CommonRequestParams,
  TPSK_OpPhaseRequestParams,
  ? -1 => key,           ; ukid of mac key
  ? -6 => alg,           ; mac algorithm
  ? -11 => input         ; input data
})
```

Request parameters

- `key`: The `ukid` of the MAC signing key.
- `alg`: The MAC algorithm.
- `input`: The content to be MAC signed.

Response

```
TPSK_SignMACResponse = #6.50032 ({
  TPSK_CommonResponseParams,
  TPSK_OpPhaseResponseParams,
  ? -14 => mac          ; mac
})
```

Response parameters

- `mac`: The signed MAC, included in finish response message if successfully created.

TPS Keystore Transaction parameter usage

The TPS Keystore SHALL keep state between request-response message pairs. The MAC returned in the finish response message SHALL be calculated over the whole content received in multiple request messages based on the MAC algorithm and the MAC key specified in the init request message.

The TPS Keystore Transaction parameters for `TPSK_SignMAC` SHALL be as listed below.

Table 3-9: `TPSK_SignMAC` TPS Keystore Transaction Parameters

Message	Parameters			Notes
init request message	`key`	M	Identifies the MAC key	
	`alg`	M	Identifies the MAC algorithm	
	`input`	O	Contains first part of content	
init response message			*no response specific parameters*	
update request message	`input`	M	Contains next part of content	Omit `key` and `alg`
update response message			*no response specific parameters*	
finish request message	`input`	O	Contains last part of content	Omit `key` and `alg`
finish response message	`mac`	M	Contains calculated MAC	

Implementation notes

3.4.18 TPSK_VerifyMAC

Description

Verify a MAC of data with a key; see section 4.7 for details.

Request

```
TPSK_VerifyMACRequest = #6.50033 ({
  TPSK_CommonRequestParams,
  TPSK_OpPhaseRequestParams,
  ? -1 => key,           ; ukid of mac key
  ? -6 => alg,           ; mac algorithm
  ? -11 => input,        ; input data
  ? -14 => mac           ; mac
})
```

Request parameters

- `key`: The `ukid` of the MAC signing key.
- `alg`: The MAC algorithm.
- `input`: The content to be MAC signed.
- `mac`: The signed MAC.

Response

```
TPSK_VerifyMACResponse = #6.50034 ({
  TPSK_CommonResponseParams,
  TPSK_OpPhaseResponseParams,
  ? -24 => result         ; result of mac verification
})
```

Response parameters

- `result`: The result of the MAC verification, included in the finish response message.

TPS Keystore Transaction parameter usage

The TPS Keystore SHALL keep state between request-response message pairs. The MAC to be verified in sent in the finish request message, and the MAC verification status is returned in the finish response message. Verification SHALL be over the whole content received in multiple request messages based on the MAC algorithm and the MAC key specified in the init request message.

The TPS Keystore Transaction parameters for `TPSK_VerifyMAC` SHALL be as listed below.

Table 3-10: `TPSK_VerifyMAC` TPS Keystore Transaction Parameters

Message	Parameters			Notes
init request message	`key`	M	Identifies the MAC key	Omit `mac`
	`alg`	M	Identifies the MAC algorithm	
	`input`	O	Contains first part of content	
init response message			*no response specific parameters*	
update request message	`input`	M	Contains next part of content	Omit `key, alg, mac`
update response message			*no response specific parameters*	
finish request message	`input`	O	Contains last part of content	Omit `key and alg`
	`mac`	M	Contains MAC to be verified	
finish response message	`result`	M	Status of verification result	

Implementation notes

3.4.19 TPSK_GenerateRandom

Description

Produce random data of a specified length.

Request

```
TPSK_GenerateRandomRequest = #6.50035 ({
  TPSK_CommonRequestParams,
  -31 => length              ; how many bytes of random data is requested
})
```

Request parameters

- `length`: Requested length of random data in bytes.

Response

```
TPSK_GenerateRandomResponse = #6.50036 ({
  TPSK_CommonResponseParams,
  ? -12 => output            ; the random data
})
```

Response parameters

- `output`: The random data, if created.

Implementation notes

3.4.20 TPSK_HasKey

Description

Determine whether a key is stored in the TPS Keystore and available to the TPS Client.

Request

```
TPSK_HasKeyRequest = #6.50037 ({
  TPSK_CommonRequestParams,
  -1 => key              ; ukid of key to be queried
})
```

Request parameters

- key: The ukid of the key to be queried.

Response

```
TPSK_HasKeyResponse = #6.50038 ({
  TPSK_CommonResponseParams,
  ? -1 => key            ; the found key
})
```

Response parameters

- key: The TPS_COSE_Key object of the requested key, if available to the TPS Client, regardless of whether it is hidden or not.

Implementation notes

3.4.21 TPSK_ListKeys

Description

List all keys that are stored and available in the TPS Keystore for the TPS Client.

Request

```
TPSK_ListKeysRequest = #6.50039 ({
  TPSK_CommonRequestParams
})
```

Request parameters

- No request specific parameters.

Response

```
TPSK_ListKeysResponse = #6.50040 ({
  TPSK_CommonResponseParams,
  ? -25 => key_list       ; list of keys
})
```

Response parameters

- `key_list`: Array of one or more keys in the TPS Keystore as specified in section 3.2.41, if at least one key is available to the TPS Client. If a key is hidden, it SHALL NOT be present in the list.

Implementation notes

3.4.22 TPSK_GetCertificateChain

Description

Get the certificate chain of a key.

Request

```
TPSK_GetCertificateChainRequest = #6.50041 ({
  TPSK_CommonRequestParams,
  -1 => key                      ; ukid of key
})
```

Request parameters

- `key`: The `ukid` of the key whose certificate chain is requested.

Response

```
TPSK_GetCertificateChainResponse = #6.50042 ({
  TPSK_CommonResponseParams,
  ? -26 => certificates          ; the certificate chain
})
```

Response parameters

- `certificates`: The certificate chain of the key, if available.

Implementation notes

3.4.23 TPSK_SetCertificateChain

Description

Set the certificate chain for a key. The key SHALL be a private key. If the key already has a certificate chain, the existing certificate chain will be removed.

Request

```
TPSK_SetCertificateChainRequest = #6.50043 ({
  TPSK_CommonRequestParams,
  -1 => key                         ; ukid of key
  -26 => certificates               ; certificate chain
})
```

Request parameters

- `key`: The `ukid` of the key for which a certificate chain is set.
- `certificates`: The certificate chain.

Response

```
TPSK_SetCertificateChainResponse = #6.50044 ({
  TPSK_CommonResponseParams
})
```

Response parameters

- No response specific attributes.

Implementation notes

3.4.24 TPSK_ValidateCertificate

Description

Validate a certificate. Validation procedure SHALL follow the steps specified in [PKIX] section 6.1, with the assumption that policy is `anyPolicy`. See section 4.12 for details.

The trusted certificate list used for the certificate validation is determined the same way as described in section 3.4.25.

Request

```
TPSK_ValidateCertificateRequest = #6.50045 ({
  TPSK_CommonRequestParams,
  -26 => certificates                ; certificate to be validated
})
```

Request parameters

- `certificates`: The certificate to be validated (single certificate).

Response

```
TPSK_ValidateCertificateResponse = #6.50046 ({
  TPSK_CommonResponseParams,
  ? -24 => result                   ; result of validation
})
```

Response parameters

- `result`: The validation result.

Implementation notes

3.4.25 TPSK_ListTrustedCertificates

Description

List all trusted certificates in the TPS Keystore. These certificates are either root certificate or intermediate CA certificates that are explicitly trusted by the TPS Keystore. These certificates are used to validate other certificates and are used with `TPSK_ValidateCertificate` message.

The list of trusted certificates SHALL include both

- common trusted certificates, i.e., the persistent list of trusted certificated stored in the TPS Keystore, and
- TPS Client specific trusted certificate, i.e., the certificates that has been imported using the `TPSK_ImportTrustedCertificate` message by the TPS Client.

Request

```
TPSK_ListTrustedCertificatesRequest = #6.50047 ({
  TPSK_CommonRequestParams
})
```

Request parameters

- No request specific parameters.

Response

```
TPSK_ListTrustedCertificatesResponse = #6.50048 ({
  TPSK_CommonResponseParams,
  ? -26 => certificates            ; list of trusted certificates
})
```

Response parameters

- `certificates`: The list of trusted certificates, if any exist.

Implementation notes

3.4.26 TPSK_ImportTrustedCertificate

Description

Import a certificate to the trusted certificate list of the TPS Keystore. A trusted certificate is either a root certificate or an intermediate CA certificate. After such a certificate is imported to the TPS Keystore, it is explicitly trusted by the TPS Keystore.

The certificate SHALL be added to TPS Client specific trusted certificate list, and the imported certificate SHALL be used for certificate validation only for the TPS Client that imported the certificate.

Request

```
TPSK_ImportTrustedCertificateRequest = #6.50049 ({
  TPSK_CommonRequestParams,
  -26 => certificates          ; trusted certificate to be imported
})
```

Request parameters

- `certificates`: The trusted certificate to be imported.

Response

```
TPSK_ImportTrustedCertificateResponse = #6.50050 ({
  TPSK_CommonResponseParams
})
```

Response parameters

- No response specific parameters.

Implementation notes

3.4.27 TPSK_RemoveTrustedCertificate

Description

Remove a certificate from the trusted certificate list of the TPS Keystore.

The TPS Client SHALL be able remove the certificate only if the same TPS Client also imported the certificate to the TPS Keystore using `TPSK_ImportTrustedCertificate`.

Request

```
TPSK_RemoveTrustedCertificateRequest = #6.50051 ({
  TPSK_CommonRequestParams,
  -26 => certificates           ; trusted certificate to be removed
})
```

Request parameters

- `certificates`: The trusted certificate to be removed (single certificate).

Response

```
TPSK_RemoveTrustedCertificateResponse = #6.50052 ({
  TPSK_CommonResponseParams
})
```

Response parameters

- No response specific parameters.

Implementation notes

3.4.28 TPSK_Abort

Description

Abort a TPS Keystore Transaction operation identified by the transaction identifier `tid`. The `tid` SHALL be active, i.e., the transaction has been initialized but not yet finished.

Request

```
TPSK_AbortRequest = #6.50053 ({
  TPSK_CommonRequestParams,
  TPSK_OpPhaseRequestParams
})
```

Request parameters

- `TPSK_OpPhaseRequestParams` SHALL include only the `tid` parameter.
- No request specific parameters.

Response

```
TPSK_AbortResponse = #6.50054 ({
  TPSK_CommonResponseParams,
  TPSK_OpPhaseResponseParams
})
```

Response parameters

- `TPSK_OpPhaseResponseParams` SHALL include only the `tid` parameter.
- No response specific parameters.

Implementation notes

4 OPERATIONS

This section details how the Messages specified in section 3.4 are used with supported cryptographic operations.

4.1 Introduction

The TPS Keystore divides the operations to *atomic operations* and *combined operations*. With an atomic operation, a single cryptographic operation is performed, e.g., SHA-256 digest of data, AES-GCM encryption or decryption of data, EdDSA signature. With a combined operation, multiple cryptographic operations are performed, e.g., key agreement followed by key derivation, or key agreement and key derivation followed by key wrapping procedure. COSE typically specifies algorithm identifiers where the identifier determines a combination of cryptographic operations.

The TPS Keystore MAY support combined operations in such a way that the combined operation can be done with single TPS Keystore message request and response pair. However, it SHALL be possible also to perform the combined operation with multiple TPS Keystore message request and response pairs. For instance, key agreement with key derivation could be performed with the `TPSK_AgreeKey` message only, where the algorithm would also specify the key derivation function, which would result in the actual key to be used with a cryptographic operation afterwards. Alternatively, the basic key agreement can be performed with `TPSK_AgreeKey` which would result in the shared secret. Then `TPSK_DeriveKey` would be used with the previously agreed shared secret to get the actual key to be used with a cryptographic operation afterwards.

COSE typically specifies algorithm identifiers where the identifier determines also the lengths of the input and output parameters of a cryptographic operation, e.g., `AES-CCM-16-128-128` where the cryptographic operation is AES-CCM with 128-bit key, 128-bit tag, and 13-byte nonce ([COSE Algs]). This specification also specifies common algorithm identifiers, where the input and output parameters of the cryptographic operations MAY vary in length provided that the input parameters and their lengths are otherwise acceptable to the cryptographic operation in question.

NOTE: This specification does not support the following cryptographic algorithms:

- public key encryption with RSA using PKCS1 scheme (RSAES-PKCS1-v1_5)
- Chinese SM2 curve based key establishment protocol (SM2-KEP)
- Chinese SM2 curve based public key encryption with Chinese SM2 curve (SM2-PKE)

4.2 Limiting Key Usage

4.2.1 Cryptographic Operations

When a key is generated, the cryptographic operations that can be done with the key can be limited by the caller. This is done by the `key_ops` array parameter in the `key_spec` structure (see 3.2.4). If one or more operations are listed in the `key_ops` array, then only those cryptographic algorithms belonging to the listed operations are allowed to be performed with the key.

4.2.2 Cryptographic Operation Inheritance

When the cryptographic algorithm is either a key derivation function (e.g., HKDF) or a key agreement algorithm (e.g., ECDH), which results to a new key, then the operations in the `keys_ops` array of the input key (i.e., a private key or a symmetric key) that is used to perform the operation SHALL affect the operations of the new key the following way:

- When the input key does not have any operations specified in the `key_ops` array, or it contains only the `derive_key` operation, then the new key SHALL NOT inherit any operations by default from the input key. The caller MAY restrict the allowed operations to one or more of the following operations: `sign`, `verify`, `derive_key`, `encrypt`, `decrypt`, `mac_create`, and `mac_verify`. The `wrap` and `unwrap` operations are not allowed to be included.
- When the input key has more than one operation specified in the `keys_ops` array, the array SHALL contain the `derive_key` operation. In this case, the new key SHALL inherit all the operations assigned to the input key by default. The caller MAY further restrict the allowed operations to one or more of the operations that were inherited from the input key. The call SHALL NOT be allowed add new operations that were not inherited from the input key.

The caller MAY restrict the allowed operations of the new key as listed above by using `key_ops` array of the `key_spec` input parameter of the corresponding operation (i.e., `TPSK_DeriveKey`, see section 3.4.12, and `TPSK_AgreeKey`, see section 3.4.10).

4.2.3 Cryptographic Algorithm

When a key is generated, the cryptographic algorithm that can be performed with the key can be limited by the caller. This is done by the `alg` parameter in the `key_spec` structure (see section 3.2.4). If the `alg` parameter is specified for the key, then only the indicated cryptographic algorithm MAY be used with the key and no other algorithm use is allowed.

4.2.4 Limiting Operations and Algorithm

The TPS Keystore implementation SHALL enforce the limitation of key usage based on cryptographic operation list (`key_ops`) and a single cryptographic algorithm (`alg`) if they are specified for the key. The caller is responsible for setting up these limitations logically so that they are mutually interoperable and that they are usable with the key type in question. Otherwise, the generated key would be unusable due to mismatching restrictions on the key.

Hence, the following rules SHALL apply:

- When the cryptographic algorithm is limited for a key pair, then it SHALL be meaningful given the type of the key pair.
- If the key pair has limitations on the key operations, the cryptographic algorithm limitation SHALL be compatible with the allowed key operations.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the respective response message (e.g., `TPSK_GenerateKeyResponse`).

4.3 Key Generation

A new key is generated to the TPS Keystore using the `TPSK_GenerateKey` operation.

4.3.1 Elliptic Key Pair Generation

An Elliptic curve key pair is generated with the parameters specified in the table below.

Table 4-1: `TPSK_GenerateKeyRequest` Parameters for Generating EC Key Pair

`TPSK_GenerateKeyRequest` Params	O/M	Possible Values
`.key_spec (-3)`	M	
`.kty (1)`	M	`ec2` or `okp`
`.kid (2)`	O	If present, any UTF-8 encoded string
`.alg (3)`	O	See section 4.2.
`.key_ops (4)`	O	See section 4.2 and Table 4-3.
`.TPS_Key_params (512)`	O	SHALL be present, if any enclosing parameters are to be present.
`.key_exportable (1)`	O	`true` or `false`; default `false`
`.key_lifetime (2)`	O	ephemeral or persistent; default persistent
`.crv (-1)`	M	See Table 4-2.

This specification supports key generation based on the following elliptic curves:

Table 4-2: Supported Elliptic Curves and their Key Types

Elliptic Curve	`.kty`	`.crv` Value	Comments
secp256r aka P-256	`ec2 (2)`	`1`	
secp384r aka P-384	`ec2 (2)`	`2`	
secp521r aka P-521	`ec2 (2)`	`3`	
x25519	`okp (3)`	`4`	
x448	`okp (3)`	`5`	
ed25519	`okp (3)`	`6`	
ed448	`okp (3)`	`7`	
sm2	`ec2 (2)`	`-65538`	

Usage of the elliptic key pair can be limited using the `key_ops` parameter and the TPS Keystore implementation SHALL enforce the limitations.

Table 4-3: Allowed Key Operations for Elliptic Key Pair

`.key_ops`	Description
not present	For `ec2` key type, the allowed operations are `sign`, `verify`, and `derive_key`. For the symmetric key obtained via the `derive_key` operation, the allowed operations are `encrypt`, `decrypt`, `mac_create`, and `mac_verify`. The `wrap` and `unwrap` operations are not allowed for the symmetric key. For `okp` key type, when key is ed25519 or ed448, the allowed key operations are `sign` and `verify`. For `okp` key type, when key is x25519 or x448, the allowed key operation is `derive_key`. For the symmetric key obtained via the `derive_key` operation, the allowed operations are `encrypt`, `decrypt`, `mac_create`, and `mac_verify`. The `wrap` and `unwrap` operations are not allowed.

`sign,`
`?verify`
	For `ec2` key type, and for `okp` key type when key is ed25519 or ed448, the key pair is allowed to sign data and verify a signature.
	The `key_ops` of the original key pair MAY not include the `verify` operation.
	If the key pair to be generated is `okp` key type where key is x25519 or x448, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_GenerateKeyResponse`.

`derive_key`	For `ec2` key type, and for `okp` key type when key is x25519 or x448, the key pair is allowed to perform a key agreement operation. By default, for the symmetric key obtained via key derivation operation, the allowed operations are `derive_key`, `encrypt`, `decrypt`, `mac_create`, and `mac_verify`. The `wrap` and `unwrap` operations are not allowed. Key pair is used for key agreement to generate a secret. A symmetric key SHOULD be generated from the secret using a key derivation. The usage rights of the generated secret and the generated symmetric key are not limited. The key derivation operation MAY further limit the allowed key operations but SHALL NOT include the `wrap` or `unwrap` operations. If the key pair to be generated is `okp` key type where key is ed25519 or ed448, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_GenerateKeyResponse`.

`derive_key,`
`?encrypt,`
`?decrypt`
	The same requirements apply as listed for the `derive_key` operation above with following exceptions:
	If, in addition to the `derive_key` operation, the `key_ops` of the original key pair includes the `encrypt` or the `decrypt` operation, then the `key_ops` of the original key pair SHALL NOT include any other key operations.
	In this case, for the symmetric key obtained via the `derive_key` operation, the allowed operations are the same as those for the original key pair.
	The key derivation operation MAY further limit the allowed key operations but SHALL NOT include any operations that were not listed as allowed key operations for the original key pair.

`.key_ops`	Description
`derive_key,` `?mac_create,` `?mac_verify`	The same requirements apply as listed for the `derive_key` operation above with following exceptions: If, in addition to the `derive_key` operation, the `key_ops` of the original key pair includes the `mac_create` operation or the `mac_verify` operation, then the `keys_ops` of the original key pair SHALL NOT include any other key operations. In this case, for the symmetric key obtained via the `derive_key` operation, the allowed operations are the same as those listed for the original key pair. The key derivation operation MAY further limit the allowed key operations but SHALL NOT include any operations that were not listed as allowed key operations for the original key pair.
`derive_key,` `?wrap,` `?unwrap`	The same requirements apply as listed for the `derive_key` operation above with following exceptions: If, in addition to the `derive_key` operation, the `key_ops` of the original key pair includes the `wrap` operation or the `unwrap` operation, then the `key_ops` of the original key pair SHALL NOT include any other key operations. By default, for the symmetric key obtained via the `derive_key` operation, the allowed operations are the same as those listed for the original key pair. The key derivation operation MAY further limit the allowed key operations but SHALL NOT include any operations that were not listed as allowed key operations for the original key pair. NOTE: Key pair can be used for wrapping and unwrapping operations only if the `key_ops` parameter explicitly lists the `wrap` and `unwrap` operations.
any other combination	The TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_GenerateKeyResponse`.

The usage of the elliptic key pair can be limited with using the `alg` parameter and the TPS Keystore implementation SHALL enforce the limitations. The supported algorithms are listed in the following sections in this chapter.

4.3.2 RSA Key Pair Generation

An RSA key pair is generated with the parameters specified in the table below.

Table 4-4: `TPSK_GenerateKeyRequest` Parameters for Generating RSA Key Pair

`TPSK_GenerateKeyRequest` Params	O/M	Possible Values
`.key_spec (-3)`	M	
`.kty (1)`	M	`rsa` (3)
`.kid (2)`	O	Any UTF-8 encoded string
`.alg (3)`	O	See section 4.2 and Table 4-3.
`.key_ops (4)`	O	See section 4.2 and Table 4-5.
`.TPS_Key_params (512)`	M	
`.key_exportable (1)`	O	`true` or `false`; default `false`
`.key_lifetime (2)`	O	ephemeral or persistent; default persistent
`.key_size (4)`	M	RSA key size in bits: 2048, 3072, 4096
`.public_exponent (-2)`	O	RSA public exponent to be used; default 65537

Usage of the RSA key pair can be limited by using the `key_ops` parameter and the TPS Keystore implementation SHALL enforce the limitations.

Table 4-5: Allowed Key Operations for RSA Key Pair

`.key_ops`	Description
not present	For `rsa` key type, the allowed operations are `sign`, `verify`, `encrypt`, and `decrypt`. The `wrap` and `unwrap` operations are not allowed.
`sign, ?verify`	For `rsa` key type, the key pair is allowed to sign data and verify a signature. The `key_ops` of the key pair MAY not include the `verify` operation.
`?encrypt, decrypt`	For `rsa` key type, the key pair is allowed to encrypt and decrypt data. The `key_ops` of the key pair MAY not include the `encrypt` operation.
`?wrap, unwrap`	For `rsa` key type, the key pair is allowed to wrap and unwrap key material. The `key_ops` of the key pair MAY not include the `wrap` operation. NOTE: Key pair can be used for wrapping and unwrapping operations only if the `key_ops` parameter explicitly lists the `wrap` and `unwrap` operations.
any other combination	The TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_GenerateKeyResponse`.

The usage of the RSA key pair can be limited with using the `alg` parameter and the TPS Keystore implementation SHALL enforce the limitations. The supported algorithms are listed in the following sections in this chapter.

4.3.3 Symmetric Key Generation

A symmetric key is generated with the parameters specified in the table below.

Table 4-6: `TPSK_GenerateKeyRequest` Parameters for Generating Symmetric Key

`TPSK_GenerateKeyRequest` Params	O/M	Possible Values
`.key_spec (-3)`	M	
`.kty (1)`	M	`symm (4)`
`.kid (2)`	O	Any UTF-8 encoded string
`.alg (3)`	O	See section 4.2.
`.key_ops (4)`	O	See section 4.2 and Table 4-7.
`.TPS_Key_params (512)`	M	
`.key_exportable (1)`	O	`true` or `false`; default `false`
`.key_lifetime (2)`	O	ephemeral or persistent; default persistent
`.key_size (4)`	M	key size in bits: 80-1024 bits, SHALL be multiple of 8 bits

Usage of the symmetric key can be limited with using the `key_ops` parameter and the TPS Keystore implementation SHALL enforce the limitations.

Table 4-7: Allowed Key Operations for Symmetric Key

`.key_ops`	Description
not present	For symmetric key type, the allowed operations are `mac_create`, `mac_verify`, `encrypt`, and `decrypt`. The `wrap` and `unwrap` operations are not allowed.
`?mac_create`, `?mac_verify`	For symmetric key type, the key is allowed to create a MAC and verify the MAC of data. If the `key_ops` of the key includes the `mac_create` operation or the `mac_verify` operation, then no other key operations are allowed in the `key_ops`.
`?encrypt`, `?decrypt`	For symmetric key type, the key is allowed to encrypt and decrypt data. If the `key_ops` of the key includes the `encrypt` operation or the `decrypt` operation, then no other key operations are allowed in the `key_ops`.
`?wrap`, `?unwrap`	For symmetric key type, the key is allowed to wrap and unwrap key material. If the `key_ops` of the key includes the `wrap` operation or the `unwrap` operation, then no other key operations are allowed in the `key_ops`. NOTE: Key pair can be used for wrapping and unwrapping operations only if the `key_ops` parameter explicitly lists `wrap` and `unwrap` operations.
any other combination	The TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_GenerateKeyResponse`.

The usage of the RSA key pair can be limited with using the `alg` parameter and the TPS Keystore implementation SHALL enforce the limitations. The supported algorithms are listed in the following sections in this chapter.

4.4 Key Derivation

4.4.1 General

For key derivation, the following rules SHALL apply:

- If the input key material has key usage limitations:
  - If `key_ops` values for the input key material are set, the `key_spec` of the derived key SHALL NOT contain any key operations that are not present in the `.key_ops` values of the input key material.
  - If the `alg` value for the input key material is set, the `alg` value for key derivation request SHALL be equal to that value.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_DeriveKeyResponse`.

4.4.2 HMAC-Based Extract-and-Expand Key Derivation Function (HKDF)

A symmetric key is derived with HKDF with the parameters specified in the table below.

Table 4-8: `TPSK_DeriveKeyRequest` Parameters for Generating Symmetric Key

`TPSK_DeriveKeyRequest` Params	O/M	Possible Values
`.key (-1)`	M	The symmetric key to be used as input key material.
`.key_spec (-3)`	M	Same as for symmetric key generation, section 4.3.3.
`.alg (-6)`	M	Any supported algorithm listed in Table 4-9.
`.salt (-15)`	O	Optional salt value of any length, divisible by 8
`.info (-17)`	O	Optional context and application specific information

Supported digest algorithms are listed in the table below.

Table 4-9: HKDF Algorithms

`.alg`	Value	Description
direct+HKDF-SHA-512	-10	Direct use of key material with HKDF using SHA-512 [COSE Algs]
direct+HKDF-SHA-256	-11	Direct use of key material with HKDF using SHA-256 [COSE Algs]
direct+HKDF-AES-256	-12	Direct use of key material with HKDF using AES-MAC-256 [COSE Algs]
direct+HKDF-AES-128	-13	Direct use of key material with HKDF using AES-MAC-128 [COSE Algs]
direct+HKDF-SM3	-65541	Direct use of key material with HKDF using SM3 [SM3]

4.5 Hash Functions

A digest is generated with the parameters specified in the table below.

Table 4-10: `TPSK_HashRequest` Parameters for Generating Digests

`TPSK_HashRequest` Params	O/M	Possible Values
`.alg (-6)`	M	Any supported algorithm listed in Table 4-11
`.input (-11)`	M	Data to be hashed

Supported digest algorithms are listed in the table below.

Table 4-11: Digest Algorithms

`.alg`	Value	Description
SHA-1	-14	SHA-1 hash [COSE Hash]
SHA-256	-16	SHA-256 hash [COSE Hash]
SHA-384	-43	SHA-384 hash [COSE Hash]
SHA-512	-44	SHA-512 hash [COSE Hash]
SHA-256/64	-15	SHA-256 hash truncated to 64 bits [COSE Hash]
SHA-512/256	-17	SHA-512 hash truncated to 256 bits [COSE Hash]
SHAKE128	-18	SHAKE-128 256-bit hash value [COSE Hash]
SHAKE256	-45	SHAKE-256 512-bit hash value [COSE Hash]
SM3	-65539	SM3 hash [SM3]

4.6 Signature Algorithms

4.6.1 General

For generating a signature, the following rules SHALL apply for the `TPSK_SignRequest`:

- If the key pair has key operation limitations defined, the operation parameters SHALL include the `sign` operation.
- If the key pair has algorithm limitation defined, the algorithm limitation of the key pair SHALL be equal to the signature algorithm in `TPSK_SignRequest`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_SignResponse`.

For verifying a signature, the following rules SHALL apply for the `TPSK_VerifyRequest`:

- If the key pair has key operation limitations defined, the operation parameters SHALL include the `verify` operation.
- If the key pair has algorithm limitation defined, the algorithm of the key pair SHALL be equal to the algorithm in `TPSK_VerifyRequest`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_VerifyResponse`.

4.6.2 Elliptic Curve Digital Signature Algorithm (ECDSA)

4.6.2.1 Signature Generation

An ECDSA signature is generated with the parameters specified in the table below.

Table 4-12: `TPSK_SignRequest` Parameters for Generating ECDSA Signatures

`TPSK_SignRequest` Params	O/M	Possible Values
`.key (-1)`	M	`ukid of ec2` key pair
`.alg (-6)`	M	See Table 4-14.
`.input (-11)`	M	Data to be signed

For ECDSA signature generation, in addition to the general rules listed in section 4.6.1, the following rules SHALL also apply:

- Key type of the key SHALL be `ec2`, and curve SHALL NOT be `sm2`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_SignResponse`.

4.6.2.2 Signature Verification

An ECDSA signature is verified with the parameters specified in the table below.

Table 4-13: `TPSK_VerifyRequest` Parameters for Verifying ECDSA Signatures

`TPSK_VerifyRequest` Params	O/M	Possible Values
`.key (-1)`	M	`ukid of ec2` key pair
`.alg (-6)`	M	See Table 4-14.
`.input (-11)`	M	Data to be signed
`.signature (-13)`	M	The signature to be verified

For ECDSA signature verification, in addition to the general rules listed in section 4.6.1, the following rules SHALL also apply:

- Key type of the key SHALL be `ec2`, and curve SHALL NOT be `sm2`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_VerifyResponse`.

4.6.2.3 Signature Algorithms

Table 4-14: Possible Algorithm Values for ECDSA Signature Generation and Verification

`.alg`	Value	Notes
ES256	-7	ECDSA with SHA-256 [COSE Algs]
ES384	-35	ECDSA with SHA-384 [COSE Algs]
ES512	-36	ECDSA with SHA-512 [COSE Algs]

4.6.3 Edwards-Curve Digital Signature Algorithm (EdDSA)

4.6.3.1 Signature Generation

An EdDSA signature is generated with the parameters specified in the table below.

Table 4-15: `TPSK_SignRequest` Parameters for Generating EdDSA Signatures

`TPSK_SignRequest` Params	O/M	Possible Values
`.key (-1)`	M	`ukid` of `ec2` key pair
`.alg (-6)`	M	See Table 4-17.
`.input (-11)`	M	Data to be signed

For ECDSA signature generation, in addition to the general rules listed in section 4.6.1, the following rules SHALL also apply:

- Key type of the key SHALL be `okp`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_SignResponse`.

4.6.3.2 Signature Verification

An EdDSA signature is verified with the parameters specified in the table below.

Table 4-16: `TPSK_VerifyRequest` Parameters for Verifying EdDSA Signatures

`TPSK_VerifyRequest` Params	O/M	Possible Values
`.key (-1)`	M	`ukid` of `okp` key pair
`.alg (-6)`	M	See Table 4-17.
`.input (-11)`	M	Data to be signed
`.signature (-13)`	M	Signature to be verified

For ECDSA signature verification, in addition to the general rules listed in section 4.6.1, the following rules SHALL also apply:

- Key type of the key SHALL be `okp`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_VerifyResponse`.

4.6.3.3 Signature Algorithms

Table 4-17: Possible Algorithm Values for EdDSA Signature Generation and Verification

`.alg`	Value	Notes
EdDSA	-8	EdDSA signature algorithm [COSE Algs]

4.6.4 SM2 Signature Algorithm

4.6.4.1 Signature Generation

An SM2 signature ([SM2 DSA]) is generated with the parameters specified in the table below.

Table 4-18: `TPSK_SignRequest` Parameters for Generating SM2 Signatures

`TPSK_SignRequest` Params	O/M	Possible Values
`.key (-1)`	M	`ukid of sm2` key pair
`.alg (-6)`	M	See Table 4-20.
`.input (-11)`	M	Data to be signed
`.info (-17)`	M	Identifier value (required by SM2 signature algorithm)

For SM2 signature generation, in addition to the general rules listed in section 4.6.1, the following rules SHALL also apply:

- Key type of the key SHALL be `ec2`, and curve SHALL be `sm2`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_SignResponse`.

4.6.4.2 Signature Verification

An SM2 signature is verified with the parameters specified in the table below.

Table 4-19: `TPSK_VerifyRequest` Parameters for Verifying SM2 Signatures

`TPSK_VerifyRequest` Params	O/M	Possible Values
`.key (-1)`	M	`ukid of okp` key pair
`.alg (-6)`	M	See Table 4-20.
`.input (-11)`	M	Data to be signed
`.info (-17)`	M	Identifier value (required by SM2 signature algorithm)
`.signature (-13)`	M	Signature to be verified

For SM2 signature verification, in addition to the general rules listed in section 4.6.1, the following rules SHALL also apply:

- Key type of the key SHALL be `ec2`, and curve SHALL be `sm2`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_VerifyResponse`.

4.6.4.3 Signature Algorithms

Table 4-20: Possible Algorithm Values for SM2 Signature Generation and Verification

`.alg`	Value	Notes
SM2-DSA	-65540	SM2 signature algorithm [SM2 DSA]

4.6.5 RSASSA-PSS Signature Algorithm

4.6.5.1 Signature Generation

An RSASSA-PSS signature [PKCS 1] is generated with the parameters specified in the table below.

Table 4-21: `TPSK_SignRequest` Parameters for Generating RSASSA-PSS Signatures

`TPSK_SignRequest` Params	O/M	Possible Values
`.key (-1)`	M	`ukid of rsa` key pair
`.alg (-6)`	M	See Table 4-23.
`.input (-11)`	M	Data to be signed

For RSASSA-PSS signature generation, in addition to the general rules listed in section 4.6.1, the following rules SHALL also apply:

- Key type of the key SHALL be `rsa`, and private key SHALL be present.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_SignResponse`.

4.6.5.2 Signature Verification

An RSASSA-PSS signature [PKCS 1] is verified with the parameters specified in the table below.

Table 4-22: `TPSK_VerifyRequest` Parameters for Verifying RSASSA-PSS Signatures

`TPSK_VerifyRequest` Params	O/M	Possible Values
`.key (-1)`	M	`ukid of rsa` key pair
`.alg (-6)`	M	See Table 4-23.
`.input (-11)`	M	Data to be signed
`.signature (-13)`	M	Signature to be verified

For RSASSA-PSS signature verification, in addition to the general rules listed in section 4.6.1, the following rules SHALL also apply:

- Key type of the key SHALL be `rsa`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_VerifyResponse`.

4.6.5.3 Signature Algorithms

Table 4-23: Possible Algorithm Values for RSASA-PSS Signature Generation and Verification

`.alg`	Value	Notes
PS256	-37	RSASSA-PSS with SHA-256 [COSE RSA]
PS384	-38	RSASSA-PSS with SHA-384 [COSE RSA]
PS512	-39	RSASSA-PSS with SHA-512 [COSE RSA]

4.6.6 RSASSA-PKCS1-v1_5 Signature Algorithm

4.6.6.1 Signature Generation

An RSASSA-PKCS1 signature [PKCS 1] is generated with the parameters specified in the table below.

Table 4-24: `TPSK_SignRequest` Parameters for Generating RSASSA-PKCS1 Signatures

`TPSK_SignRequest` Params	O/M	Possible Values
`.key (-1)`	M	`ukid of rsa` key pair
`.alg (-6)`	M	See Table 4-26.
`.input (-11)`	M	Data to be signed

For RSASSA-PKCS1 signature generation, in addition to the general rules listed in section 4.6.1, the following rules SHALL also apply:

- Key type of the key SHALL be `rsa`, and private key SHALL be present in TPS Keystore.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_SignResponse`.

4.6.6.2 Signature Verification

An RSASSA-PKCS1 signature [PKCS 1] is verified with the parameters specified in the table below.

Table 4-25: `TPSK_VerifyRequest` Parameters for Verifying RSASSA-PKCS1 Signatures

`TPSK_VerifyRequest` Params	O/M	Possible Values
`.key (-1)`	M	`ukid of rsa` key pair
`.alg (-6)`	M	See Table 4-26.
`.input (-11)`	M	Data to be signed
`.signature (-13)`	M	Signature to be verified

For RSASSA-PKCS1 signature verification, in addition to the general rules listed in section 4.6.1, the following rules SHALL also apply:

- Key type of the key SHALL be `rsa`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_VerifyResponse`.

4.6.6.3 Signature Algorithms

Table 4-26: Possible Algorithm Values for RSASSA-PKCS1 Signature Generation and Verification

`.alg`	Value	Notes
RS1	-65535	RSASSA-PKCS1-v1_5 with SHA-1 [COSE Reg]
RS256	-257	RSASSA-PKCS1-v1_5 with SHA-256 [COSE Reg]
RS384	-258	RSASSA-PKCS1-v1_5 with SHA-384 [COSE Reg]
RS512	-259	RSASSA-PKCS1-v1_5 with SHA-512 [COSE Reg]

4.7 Keyed Message Authentication Codes Algorithms

4.7.1 General

For generating a MAC, the following rules SHALL apply for the `TPSK_SignMACRequest`:

- If the key pair has key operation limitations, the operation parameters SHALL include the `mac_create` operation.
- If the key pair has algorithm limitation, the algorithm limitation of the key pair SHALL be equal to the signature algorithm in `TPSK_SignMACRequest`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_SignMACResponse`.

For verifying a signature, the following rules SHALL apply for the `TPSK_VerifyMACRequest`:

- If the key pair has key operation limitations, the operation parameters SHALL include the `mac_verify` operation.
- If the key pair has algorithm limitation, the algorithm of the key pair SHALL be equal to the algorithm in `TPSK_VerifyMACRequest`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_VerifyMACResponse`.

4.7.2 Hash-Based Message Authentication Codes (HMACs)

4.7.2.1 MAC Generation

An HMAC value is generated as specified in [COSE Algs] with the parameters specified in the table below.

Table 4-27: `TPSK_SignMACRequest` Parameters for Generating an HMAC

`TPSK_SignMACRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key, variable length
`.alg (-6)`	M	See Table 4-29.
`.input (-11)`	M	Data to be MAC'd

For MAC generation using HMAC, in addition to the general rules listed in section 4.7.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_SignMACResponse`.

4.7.2.2 MAC Verification

An HMAC value is verified as specified in [COSE Algs] with the parameters specified in the table below.

Table 4-28: `TPSK_VerifyMACRequest` Parameters for Generating and Verifying an HMAC

`TPSK_VerifyMACRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key, variable length
`.alg (-6)`	M	See Table 4-29.
`.input (-11)`	M	Data to be verified with the MAC
`.mac (-14)`	M	MAC to be verified

For MAC verification using HMAC, in addition to the general rules listed in section 4.7.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_VerifyMACResponse`.

4.7.2.3 MAC Algorithms

Table 4-29: Possible Algorithm Values for HMAC Generation

`.alg`	Value	Notes
HMAC 256/256	5	HMAC with SHA-256 [COSE Algs]
HMAC 384/384	6	HMAC with SHA-384 [COSE Algs]
HMAC 512/512	7	HMAC with SHA-512 [COSE Algs]
HMAC 256/64	4	HMAC with SHA-256 truncated to 64 bits [COSE Algs]

4.7.3 AES Message Authentication Code (AES-CBC-MAC)

4.7.3.1 MAC Generation

An AES-CBC-MAC value is generated as specified in [COSE Algs] with the parameters specified in the table below.

Table 4-30: `TPSK_SignMACRequest` Parameters for Generating an AES-CBC-MAC

`TPSK_SignMACRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key; key sizes 128, 192, or 256 bits
`.alg (-6)`	M	See Table 4-32.
`.input (-11)`	M	Data to be MAC'd

For MAC generation using AES-CBC-MAC, in addition to the general rules listed in section 4.7.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.
- The size of the key SHALL be compatible with the key size indicated by the selected algorithm.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_SignMACResponse`.

4.7.3.2 MAC Verification

An AES-CBC-MAC value is verified as specified in [COSE Algs] with the parameters specified in the table below.

Table 4-31: `TPSK_VerifyMACRequest` Parameters for Verifying an AES-CBC-MAC

`TPSK_VerifyMACRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key; key sizes 128 or 256 bits
`.alg (-6)`	M	See Table 4-32.
`.input (-11)`	M	Data to be verified with the MAC
`.mac (-14)`	M	MAC to be verified

For MAC verification using AES-CBC-MAC, in addition to the general rules listed in section 4.7.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.
- The size of the key SHALL be compatible with the key size indicated by the selected algorithm.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_VerifyMACResponse`.

4.7.3.3 MAC Algorithms

Table 4-32: Possible Algorithm Values for AES-CBC-MAC Generation and Verification

`.alg`	Value	Notes
AES-MAC 128/64	14	AES-MAC 128-bit key, 64-bit tag [COSE Algs]
AES-MAC 256/64	15	AES-MAC 256-bit key, 64-bit tag [COSE Algs]
AES-MAC 128/128	25	AES-MAC 128-bit key, 128-bit tag [COSE Algs]
AES-MAC 256/128	26	AES-MAC 256-bit key, 128-bit tag [COSE Algs]

4.7.4 AES Cipher-Based Message Authentication Code (AES-CMAC)

4.7.4.1 MAC Generation

An AES-CMAC value is generated as specified in [RFC 4493] with the parameters specified in the table below.

Table 4-33: `TPSK_SignMACRequest` Parameters for Generating an AES-CMAC

`TPSK_SignMACRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key; key size 128 bits
`.alg (-6)`	M	See Table 4-35.
`.input (-11)`	M	Data to be MAC'd

For MAC generation using AES-CBC-MAC, in addition to the general rules listed in section 4.7.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.
- The size of the key SHALL be compatible with the key size indicated by the selected algorithm.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_SignMACResponse`.

4.7.4.2 MAC Verification

An AES-CMAC value is verified as specified in [RFC 4493] with the parameters specified in the table below.

Table 4-34: `TPSK_VerifyMACRequest` Parameters for Verifying an AES-CMAC

`TPSK_VerifyMACRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key; key size 128 bits
`.alg (-6)`	M	See Table 4-35.
`.input (-11)`	M	Data to be verified with the MAC
`.mac (-14)`	M	MAC to be verified

For MAC verification using AES-CBC-MAC, in addition to the general rules listed in section 4.7.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.
- The size of the key SHALL be compatible with the key size indicated by the selected algorithm.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_VerifyMACResponse`.

4.7.4.3 MAC Algorithms

Table 4-35: Possible Algorithm Values for AES-CMAC Generation and Verification

`.alg`	Value	Notes
AES-CMAC	-65537	AES-CMAC with 128-bit key

4.8 Content Encryption Algorithms

4.8.1 General

For encrypting data, the following rules SHALL apply for the `TPSK_EncryptRequest`:

- If the key pair has key operation limitations, the operation parameters SHALL include the `encrypt` operation.
- If the key pair has algorithm limitation, the algorithm limitation of the key pair SHALL be equal to the signature algorithm in `TPSK_EncryptRequest`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_EncryptResponse`.

For decrypting data, the following rules SHALL apply for the `TPSK_DecryptRequest`:

- If the key pair has key operation limitations, the operation parameters SHALL include the `decrypt` operation.
- If the key pair has algorithm limitation, the algorithm of the key pair SHALL be equal to the algorithm in `TPSK_DecryptRequest`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_DecryptResponse`.

4.8.2 AES-GCM

4.8.2.1 Encryption

An AES-GCM encryption is done with the parameters specified in the table below.

Table 4-36: `TPSK_EncryptRequest` Parameters for AES-GCM Encryption

`TPSK_EncryptRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key; key size 128, 192, or 256 bits
`.alg (-6)`	M	See Table 4-38.
`.iv (-7)`	M	Nonce, size 8 to 128 bits, divisible by 8; recommended size 96 bits
`.aad (-9)`	O	Optional additional authenticated data
`.tag_length (-19)`	M	Tag size: 8 to 128 bits, divisible by 8
`.input (-11)`	M	Data to be encrypted

For encryption, in addition to the general rules listed in section 4.8.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.
- The size of the key SHALL be compatible with the key size indicated by the selected algorithm. If the algorithm does not indicate the key size, the key size SHALL be 128, 192, or 256 bits.
- The size of the nonce SHALL be compatible with the nonce size indicated by the selected algorithm. If the algorithm does not indicate the size of the nonce, the nonce size SHALL be between 8 and 128 bits, divisible by 8.
- The size of the tag SHALL be compatible with the tag size indicated by the selected algorithm. If the algorithm does not indicate the size of the tag, the tag size SHALL be between 8 and 128 bits, divisible by 8.
- All input parameter lengths SHALL be as specified in [AES-GCM].

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_EncryptResponse`.

4.8.2.2 Decryption

An AES-GCM decryption is done with the parameters specified in the table below.

Table 4-37: `TPSK_DecryptRequest` Parameters for AES-GCM Encryption

`TPSK_DecryptRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key; key size 128, 192, or 256 bits
`.alg (-6)`	M	See Table 4-38.
`.iv (-7)`	M	Nonce, size 8 to 128 bites, divisible by 8; recommended size 96 bits
`.aad (-9)`	O	Optional additional authenticated data
`.tag_length (-19)`	M	Tag size: 8 to 128 bits, divisible by 8
`.input (-11)`	M	Encrypted data concatenated with the tag

For decryption, in addition to the general rules listed in section 4.8.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.
- The size of the key SHALL be compatible with the key size indicated by the selected algorithm. If the algorithm does not indicate the key size, the key size SHALL be 128, 192, or 256 bits.
- The size of the nonce SHALL be compatible with the nonce size indicated by the selected algorithm. If the algorithm does not indicate the size of the nonce, the nonce size SHALL be between 8 and 128 bits, divisible by 8.
- The size of the tag SHALL be compatible with the tag size indicated by the selected algorithm. If the algorithm does not indicate the size of the tag, the tag size SHALL be between 8 and 128 bits, divisible by 8.
- All input parameter lengths SHALL be as specified in [AES-GCM].

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_DecryptResponse`.

4.8.2.3 Encryption Algorithms

Table 4-38: Possible Algorithm Values for AES-GCM Encryption/Decryption

`.alg`	Value	Notes
AES-GCM + any	-65547	AES-GCM mode with any allowed key size, nonce size, tag length
A128GCM	1	AES-GCM mode w/ 128-bit key, 128-bit tag, 96-bit nonce [COSE Algs]
A192GCM	2	AES-GCM mode w/ 192-bit key, 128-bit tag, 96-bit nonce [COSE Algs]
A256GCM	3	AES-GCM mode w/ 256-bit key, 128-bit tag, 96-bit nonce [COSE Algs]

4.8.3 AES-CCM

4.8.3.1 Encryption

An AES-CCM encryption is done with the parameters specified in the table below.

Table 4-39: `TPSK_EncryptRequest` Parameters for AES-CCM Encryption

`TPSK_EncryptRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key; key size 128, 192, or 256 bits
`.alg (-6)`	M	See Table 4-41.
`.iv (-7)`	M	Nonce, size 56 to 104 bits, divisible by 8
`.aad (-9)`	O	Optional additional authenticated data
`.tag_length (-19)`	M	Tag size: 32, 48, 64, 80, 96, 112, 128 bits
`.input (-11)`	M	Data to be encrypted

For encryption, in addition to the general rules listed in section 4.8.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.
- The size of the key SHALL be compatible with the key size indicated by the selected algorithm. If the algorithm does not indicate the key size, the key size SHALL be 128, 192, or 256 bits.
- The size of the nonce SHALL be compatible with the nonce size indicated by the selected algorithm. If the algorithm does not indicate the size of the nonce, the nonce size SHALL be between 56 and 104 bits, divisible by 8.
- The size of the tag SHALL be compatible with the tag size indicated by the selected algorithm. If the algorithm does not indicate the size of the tag, the tag size SHALL be between 32 and 128 bits, divisible by 16.
- All input parameter lengths SHALL be as specified in [CCM], and especially they SHALL result in the allowed L and M parameter choices as specified in [CCM].

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_EncryptResponse`.

4.8.3.2 Decryption

An AES-CCM decryption is done with the parameters specified in the table below.

Table 4-40: `TPSK_DecryptRequest` Parameters for AES-CCM Encryption

`TPSK_DecryptRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key; key size 128, 192, or 256 bits
`.alg (-6)`	M	See Table 4-41.
`.iv (-7)`	M	Nonce, size 56 to 104 bits, divisible by 8
`.aad (-9)`	O	Optional additional authenticated data
`.tag_length (-19)`	M	Tag size: 32, 48, 64, 80, 96, 112, 128 bits
`.input (-11)`	M	Encrypted data concatenated with the tag

For decryption, in addition to the general rules listed in section 4.8.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.
- The size of the key SHALL be compatible with the key size indicated by the selected algorithm. If the algorithm does not indicate the key size, the key size SHALL be 128, 192, or 256 bits.
- The size of the nonce SHALL be compatible with the nonce size indicated by the selected algorithm. If the algorithm does not indicate the size of the nonce, the nonce size SHALL be between 56 and 104 bits, divisible by 8.
- The size of the tag SHALL be compatible with the tag size indicated by the selected algorithm. If the algorithm does not indicate the size of the tag, the tag size SHALL be between 32 and 128 bits, divisible by 16.
- All input parameter lengths SHALL be as specified in [CCM], and especially they SHALL result in the allowed L and M parameter choices as specified in [CCM].

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_DecryptResponse`.

4.8.3.3 Encryption Algorithms

Table 4-41: Possible Algorithm Values for AES-CCM Encryption/Decryption

`.alg`	Value	Notes
AES-CCM + any	-65548	AES-CCM mode with any allowed key size, nonce size, tag length
AES-CCM-16-64-128	10	AES-CCM mode 128-bit key, 64-bit tag, 13-byte nonce [COSE Algs]
AES-CCM-16-64-256	11	AES-CCM mode 256-bit key, 64-bit tag, 13-byte nonce [COSE Algs]
AES-CCM-64-64-128	12	AES-CCM mode 128-bit key, 64-bit tag, 7-byte nonce [COSE Algs]
AES-CCM-64-64-256	13	AES-CCM mode 256-bit key, 64-bit tag, 7-byte nonce [COSE Algs]
AES-CCM-16-128-128	30	AES-CCM mode 128-bit key, 128-bit tag, 13-byte nonce [COSE Algs]
AES-CCM-16-128-256	31	AES-CCM mode 256-bit key, 128-bit tag, 13-byte nonce [COSE Algs]
AES-CCM-64-128-128	32	AES-CCM mode 128-bit key, 128-bit tag, 7-byte nonce [COSE Algs]
AES-CCM-64-128-256	33	AES-CCM mode 256-bit key, 128-bit tag, 7-byte nonce [COSE Algs]

4.8.4 ChaCha20 and Poly1305

4.8.4.1 Encryption

A ChaCha20/Poly1305 encryption is done with the parameters specified in the table below.

Table 4-42: `TPSK_EncryptRequest` Parameters for ChaCha20/Poly1305 Encryption

`TPSK_EncryptRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key; key size 256 bits
`.alg (-6)`	M	See Table 4-44.
`.iv (-7)`	M	Nonce, size 96 bits
`.aad (-9)`	O	Optional additional authenticated data
`.tag_length (-19)`	M	Tag size: 128 bits
`.input (-11)`	M	Data to be encrypted

For encryption, in addition to the general rules listed in section 4.8.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.
- The size of the key SHALL be 256 bits.
- The size of the nonce SHALL be 96 bits.
- The size of the tag SHALL be 128 bits.
- All input parameter lengths SHALL be as specified in [ChaCha-Poly].

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_EncryptResponse`.

4.8.4.2 Decryption

A ChaCha20/Poly1305 decryption is done with the parameters specified in the table below.

Table 4-43: `TPSK_DecryptRequest` Parameters for ChaCha20/Poly1305 Encryption

`TPSK_DecryptRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key; key size 256 bits
`.alg (-6)`	M	See Table 4-44.
`.iv (-7)`	M	Nonce, size 96 bits
`.aad (-9)`	O	Optional additional authenticated data
`.tag_length (-19)`	M	Tag size 128 bits
`.input (-11)`	M	Encrypted data concatenated with the tag

For decryption, in addition to the general rules listed in section 4.8.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.
- The size of the key SHALL be 256 bits.
- The size of the nonce SHALL be 96 bits.
- The size of the tag SHALL be 128 bits.
- All input parameter lengths SHALL be as specified in [ChaCha-Poly].

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_DecryptResponse`.

4.8.4.3 Encryption Algorithms

Table 4-44: Possible Algorithm Values for ChaCha20/Poly1305 Encryption/Decryption

`.alg`	Value	Notes
ChaCha20/Poly1305	24	ChaCha20/Poly1305 mode key size 256 bits, nonce size 96 bits, tag length 128 bits [COSE Algs]

4.8.5 SM4-GCM

SM4 block cipher [SM4] is used as AES is used with GCM.

4.8.5.1 Encryption

An SM4-GCM encryption is done with the parameters specified in the table below.

Table 4-45: `TPSK_EncryptRequest` Parameters for SM4-GCM Encryption

`TPSK_EncryptRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key; key size 128 bits
`.alg (-6)`	M	See Table 4-47.
`.iv (-7)`	M	Nonce, size 8 to 128 bits, divisible by 8
`.aad (-9)`	O	Optional additional authenticated data
`.tag_length (-19)`	M	Tag size: 8 to 128 bits, divisible by 8
`.input (-11)`	M	Data to be encrypted

For encryption, in addition to the general rules listed in section 4.8.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.
- The size of the key SHALL be 128 bits.
- The size of the nonce SHALL be compatible with the nonce size indicated by the selected algorithm. If the algorithm does not indicate the size of the nonce, the nonce size SHALL be between 8 and 128 bits, divisible by 8.
- The size of the tag SHALL be compatible with the tag size indicated by the selected algorithm. If the algorithm does not indicate the size of the tag, the tag size SHALL be between 8 and 128 bits, divisible by 8.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_EncryptResponse`.

4.8.5.2 Decryption

An SM4-GCM decryption is done with the parameters specified in the table below.

Table 4-46: `TPSK_DecryptRequest` Parameters for SM4-GCM Encryption

`TPSK_DecryptRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key; key size 128 bits
`.alg (-6)`	M	See Table 4-47.
`.iv (-7)`	M	Nonce, size 8 to 128 bits, divisible by 8
`.aad (-9)`	O	Optional additional authenticated data
`.tag_length (-19)`	M	Tag size: 8 to 128 bits, divisible by 8
`.input (-11)`	M	Encrypted data concatenated with the tag

For decryption, in addition to the general rules listed in section 4.8.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.
- The size of the key SHALL be 128 bits.
- The size of the nonce SHALL be compatible with the nonce size indicated by the selected algorithm. If the algorithm does not indicate the size of the nonce, the nonce size SHALL be between 8 and 128 bits, divisible by 8.
- The size of the tag SHALL be compatible with the tag size indicated by the selected algorithm. If the algorithm does not indicate the size of the tag, the tag size SHALL be between 8 and 128 bits, divisible by 8.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_DecryptResponse`.

4.8.5.3 Encryption Algorithms

Table 4-47: Possible Algorithm Values for SM4-GCM Encryption/Decryption

`.alg`	Value	Notes
SM4-GCM + any	-65549	SM4-GCM mode with 128-bit key, any allowed nonce and tag size
SM4-GCM-128	-65542	SM4-GCM mode w/ 128-bit key, 128-bit nonce, and 128-bit tag

4.8.6 SM4-CCM

SM4 block cipher [SM4] is used as AES is used with CCM.

4.8.6.1 Encryption

An SM4-CCM encryption is done with the parameters specified in the table below.

Table 4-48: `TPSK_EncryptRequest` Parameters for SM4-CCM Encryption

`TPSK_EncryptRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key; key size 128 bits
`.alg (-6)`	M	See Table 4-50.
`.iv (-7)`	M	Nonce, size 56 to 104 bits, divisible by 8
`.aad (-9)`	O	Optional additional authenticated data
`.tag_length (-19)`	M	Tag size: 32, 48, 64, 80, 96, 112, 128 bits
`.input (-11)`	M	Data to be encrypted

For encryption, in addition to the general rules listed in section 4.8.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.
- The size of the key SHALL be 128 bits.
- The size of the nonce SHALL be compatible with the nonce size indicated by the selected algorithm. If the algorithm does not indicate the size of the nonce, the nonce size SHALL be between 56 and 104 bits, divisible by 8.
- The size of the tag SHALL be compatible with the tag size indicated by the selected algorithm. If the algorithm does not indicate the size of the tag, the tag size SHALL be between 32 and 128 bits, divisible by 16.
- All input parameter lengths SHALL be as specified in [CCM], and especially they SHALL result in the allowed L and M parameter choices as specified in [CCM].

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_EncryptResponse`.

4.8.6.2 Decryption

An SM4-CCM decryption is done with the parameters specified in the table below.

Table 4-49: `TPSK_DecryptRequest` Parameters for SM4-CCM Encryption

`TPSK_DecryptRequest` Params	O/M	Possible Values
`.key (-1)`	M	Symmetric key; key size 128 bits
`.alg (-6)`	M	See Table 4-50.
`.iv (-7)`	M	Nonce, size 56 to 104 bits, divisible by 8
`.aad (-9)`	O	Optional additional authenticated data
`.tag_length (-19)`	M	Tag size: 32, 48, 64, 80, 96, 112, 128 bits
`.input (-11)`	M	Encrypted data concatenated with the tag

For decryption, in addition to the general rules listed in section 4.8.1, the following rules SHALL also apply:

- Key type of the key SHALL be `symm`.
- The size of the key SHALL be 128 bits.
- The size of the nonce SHALL be compatible with the nonce size indicated by the selected algorithm. If the algorithm does not indicate the size of the nonce, the nonce size SHALL be between 56 and 104 bits, divisible by 8.
- The size of the tag SHALL be compatible with the tag size indicated by the selected algorithm. If the algorithm does not indicate the size of the tag, the tag size SHALL be between 32 and 128 bits, divisible by 16.
- All input parameter lengths SHALL be as specified in [CCM], and especially they SHALL result in the allowed L and M parameter choices as specified in [CCM].

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_DecryptResponse`.

4.8.6.3 Encryption Algorithms

Table 4-50: Possible Algorithm Values for SM4-CCM Encryption/Decryption

`.alg`	Value	Notes
SM4-CCM + any	-65550	SM4-CCM mode with any allowed key, nonce, tag size
SM4-CCM-16-64-128	-65543	SM4-CCM mode 128-bit key, 64-bit tag, 13-byte nonce
SM4-CCM-64-64-128	-65544	SM4-CCM mode 128-bit key, 64-bit tag, 7-byte nonce
SM4-CCM-16-128-128	-65545	SM4-CCM mode 128-bit key, 128-bit tag, 13-byte nonce
SM4-CCM-64-128-128	-65546	SM4-CCM mode 128-bit key, 128-bit tag, 7-byte nonce

4.8.7 RSAES-OAEP

4.8.7.1 Encryption

An RSAES-OAEP encryption is done with the parameters specified in the table below.

Table 4-51: `TPSK_EncryptRequest` Parameters for RSAES-OAEP Encryption

`TPSK_EncryptRequest` Params	O/M	Possible Values
`.key (-1)`	M	`ukid` of RSA public key
`.alg (-6)`	M	See Table 4-53.
`.label (-16)`	O	Optional label
`.input (-11)`	M	Data to be encrypted

For encryption, in addition to the general rules listed in section 4.8.1, the following rules SHALL also apply:

- Key type of the key SHALL be `rsa`.
- The size of the key SHALL be 2048, 3072, or 4096 bits.
- All input parameter lengths SHALL be as specified in [PKCS 1].

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_EncryptResponse`.

4.8.7.2 Decryption

An RSAES-OAEP decryption is done with the parameters specified in the table below.

Table 4-52: `TPSK_DecryptRequest` Parameters for RSAES-OAEP Encryption

`TPSK_DecryptRequest` Params	O/M	Possible Values
`.key (-1)`	M	`ukid` of RSA key pair
`.alg (-6)`	M	See Table 4-53.
`.label (-16)`	O	Optional label
`.input (-11)`	M	Encrypted data

For decryption, in addition to the general rules listed in section 4.8.1, the following rules SHALL also apply:

- Key type of the key SHALL be `rsa`, and private key SHALL be present in TPS Keystore.
- The size of the key SHALL be 2048, 3072, or 4096 bits.
- All input parameter lengths SHALL be as specified in [PKCS 1].

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_DecryptResponse`.

4.8.7.3 Encryption Algorithms

Table 4-53: Possible Algorithm Values for RSAES-OEAP Encryption/Decryption

`.alg`	Value	Notes
RSAES-OAEP w/ RFC 8017 default parameters	-40	RSAES-OAEP w/ SHA-1 [COSE RSA]
RSAES-OAEP w/ SHA-256	-41	RSAES-OAEP w/ SHA-256 [COSE RSA]
RSAES-OAEP w/ SHA-512	-42	RSAES-OAEP w/ SHA-512 [COSE RSA]

4.9 Key Agreement Algorithms

4.9.1 General

For performing a key agreement, the following rules SHALL apply for the `TPSK_AgreeKeyRequest`:

- If the key pair has key operation limitations, the operation parameters SHALL include the `derive_key` operation.
- If the key pair has algorithm limitation, the algorithm limitation of the key pair SHALL be equal to the signature algorithm in `TPSK_AgreeKeyRequest`.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_AgreeKeyResponse`.

4.9.2 ECDH

An ECDH is done with the parameters specified in the table below. The result of the plain ECDH keying material SHALL only be used in a key derivation to produce a key for a cryptographic operation.

Table 4-54: `TPSK_AgreeKeyRequest` Parameters for ECDH

`TPSK_AgreeKeyRequest` Params	O/M	Possible Values
`.key (-1)`	M	A private key of an EC curve.
`.pubkey (-2)`	M	A public key of the same EC curve as the private key.
`.key_spec (-3)`	M	Same as for symmetric key generation, section 4.3.3.

For performing ECDH only, in addition to the general rules listed in section 4.9.1, the following rules SHALL also apply:

- Key type of the key SHALL be `ec2` or `okp`, curve SHALL NOT be `sm2`.
- The `key` and `pubkey` parameter SHALL be on the same curve.
- `key_ops` parameter of `key_spec` SHALL contain the `derive_key` operation.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_AgreeKeyResponse`.

4.9.3 ECDH with Key Derivation

An ECDH with key derivation is done with the parameters specified in the table below.

Table 4-55: `TPSK_AgreeKeyRequest` Parameters for ECDH with Key Derivation

`TPSK_AgreeKeyRequest` Params	O/M	Possible Values
`.key (-1)`	M	A private key of an EC curve.
`.pubkey (-2)`	M	A public key of the same EC curve as the private key.
`.key_spec (-3)`	M	Same as for symmetric key generation, section 4.3.3.
`.alg (-6)`	M	See Table 4-56.
`.nonce (-10)`	O	
`.salt (-15)`	O	Optional salt value, see Table 4-56.
`.info (-17)`	O	Optional info value, see Table 4-56.

For performing ECDH with key derivation, in addition to the general rules listed in section 4.9.1, the following rules SHALL also apply:

- Key type of the key SHALL be `ec2` or `okp`, curve SHALL NOT be `sm2`.
- The `key` and `pubkey` parameter SHALL be on the same curve.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_AgreeKeyResponse`.

Table 4-56: Possible Algorithm Values for ECDH with Key Derivation Algorithms

`.alg`	Value	Notes
ECDH-SS + HKDF-256 ECDH-ES + HKDF-256	-27 -25	Perform ECDH and derive the key using HKDF with SHA-256. [COSE Algs] `.salt` MAY contain the salt value for HKDF. `.info` MAY contain the info value for HKDF.
ECDH-SS + HKDF-512 ECDH-ES + HKDF-512	-28 -26	Perform ECDH and derive the key using HKDF with SHA-512. [COSE Algs] `.salt` MAY contain the salt value for HKDF. `.info` MAY contain the info value for HKDF.
ECDH-SS + A128KW ECDH-ES + A128KW	-32 -29	Perform ECDH and derive 128-bit key using AES Key Wrap. [COSE Algs] The resulting key can only be used to wrap and unwrap symmetric keys with length 128, 192, or 256 bits.
ECDH-SS + A192KW ECDH-ES + A192KW	-33 -30	Perform ECDH and derive 192-bit key using AES Key Wrap. [COSE Algs] The resulting key can only be used to wrap and unwrap symmetric keys with length 192 or 256 bits.
ECDH-SS + A256KW ECDH-ES + A256KW	-34 -31	Perform ECDH and derive 256-bit key using AES Key Wrap. [COSE Algs] The resulting key can only be used to wrap and unwrap symmetric keys with length 256 bits.

NOTE: ECDH-SS and ECDH-ES based algorithms are essentially the same except that in ES the EC keys are ephemeral while in SS both keys are static. For the TPS Keystore implementation, it is not relevant whether an ES or SS based algorithm identifier is used, as long as both keys are present and usable in the TPS Keystore at the time of the operation. The result of the operation is the same.

4.10 Key Wrapping and Unwrapping

4.10.1 Key Wrap

A key wrapping is done with the parameters specified in the table below.

Table 4-57: `TPSK_WrapKeyRequest` Parameters for Key Wrapping

`TPSK_WrapKeyRequest` Params	O/M	Possible Values
`.key (-1)`	M	The key to be wrapped
`.wrapping_key (-4)`	M	The key to be used in encrypting the key to be wrapped
`.alg (-6)`	M	The encryption algorithm for the key wrapping
`.iv (-7)`	O	If the encryption algorithm used for key wrapping requires initialization vector or nonce as input parameter, this parameter SHALL be present.
`.aad (-9)`	O	If the encryption algorithm used for key wrapping requires additional authentication data as input parameter, this parameter SHALL be present.
`.tag_length (-19)`	O	If the encryption algorithm used for key wrapping requires tag length to be set, this parameter SHALL be present.
`.key_format (-20)`	M	The format of the key to be wrapped

For key wrapping, the following rules SHALL apply:

- The `key_exportable` value for the `key` SHALL be `true`.
- The `key_ops` value of the `wrapping_key` SHALL contain only the `wrap` value, or only the `wrap` and `unwrap` values. No other `key_ops` values SHALL be present.
- The `alg` value SHALL be one of the encryption algorithm values specified in section 4.8 or listed in Table 4-59.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_WrapKeyResponse`.

4.10.2 Key Unwrap

A key unwrapping is done with the parameters specified in the table below.

Table 4-58: `TPSK_UnwrapKeyRequest` Parameters for Key Wrapping

`TPSK_UnwrapKeyRequest` Params	O/M	Possible Values
`.key_spec (-3)`	O	The parameters for the imported key
`.wrapping_key (-4)`	M	The key to be used in decrypting the wrapped key
`.wrapped_key (-5)`	M	The wrapped key
`.alg (-6)`	M	The encryption algorithm for the key unwrapping
`.iv (-7)`	O	If the encryption algorithm used for key unwrapping requires initialization vector or nonce as input parameter, this parameter SHALL be present.
`.aad (-9)`	O	If the encryption algorithm used for key unwrapping requires additional authentication data as input parameter, this parameter SHALL be present.
`.tag_length (-19)`	O	If the encryption algorithm used for key unwrapping requires tag length to be set, this parameter SHALL be present.

For key unwrapping, the following rules SHALL apply:

- The `key_ops` value of the `wrapping_key` SHALL contain only the `unwrap` value, or only the `wrap` and `unwrap` values. No other `key_ops` values SHALL be present.
- The `alg` value SHALL be one of the encryption algorithm values specified in section 4.8 or listed in Table 4-59.

If any rule above is false, the TPS Keystore implementation SHALL return the `INVALID_ARGUMENT` error in the `TPSK_UnwrapKeyResponse`.

4.10.3 Key Wrapping Algorithms

Table 4-59: Key Wrapping and Unwrapping Algorithms

`.alg`	Value	Notes
AES-GCM algorithms listed in Table 4-38.	See Table 4-38	AES-GCM is used for wrapping/unwrapping the key. Input parameters `iv`, `aad`, and `tag_length` are used as specified in section 4.8.2.
AES-CCM algorithms listed in Table 4-41.	See Table 4-41	AES-CCM is used for wrapping/unwrapping the key. Input parameters `iv`, `aad`, and `tag_length` are used as specified in section 4.8.3.
ChaCha20/Poly1305 listed in Table 4-44.	See Table 4-44	ChaCha20/Poly1305 is used for wrapping/unwrapping the key. Input parameters `iv`, `aad`, and `tag_length` are used as specified in section 4.8.4.
ECDH-SS + A128KW ECDH-ES + A128KW	-32 -29	Wrap the key with AES Key Wrap using previously derived wrapping key whose length is 128 bits. [COSE Algs] To be wrapped key SHALL be symmetric with size 128, 192, or 256 bits. `.key_format` SHALL be `raw`.
ECDH-SS + A192KW ECDH-ES + A192KW	-33 -30	Wrap the key with AES Key Wrap using previously derived wrapping key whose length is 192 bits. [COSE Algs] To be wrapped key SHALL be symmetric with size 192 or 256 bits. `.key_format` SHALL be `raw`.
ECDH-SS + A256KW ECDH-ES + A256KW	-34 -31	Wrap the key with AES Key Wrap using previously derived wrapping key whose length is 256 bits. [COSE Algs] To be wrapped key SHALL be symmetric with size 256 bits. `.key_format` SHALL be `raw`.

4.11 Key Attestation Procedures

4.11.1 Attestation Format

Key Attestation SHALL, on request, be performed on a key present in the TPS Keystore that was generated in the TPS Keystore and for which the value of the key parameter `key_exportable` is `false`, indicating that the key cannot be exported from the TPS Keystore in any form. A Key Attestation request for any other key SHALL be rejected with `NOT_ALLOWED` status.

Key Attestation is a `COSE_Sign1` structure as specified in [COSE Struct] section 4.2, where:

- `protected` header SHALL be empty.
- `unprotected` header SHALL contain:
  - `alg` parameter containing the signature algorithm used to generate the signature
  - `content` parameter containing the content type as `tstr: "application/tps-key-attestation"`
  - `kid` parameter containing the SHA-256 hash of the public key of the attestation key, where the public key is encoded in the DER-encoded ASN.1 structure of `SubjectPublicKeyInfo` format
- `payload` SHALL contain a `TPS_COSE_Key` structure as described below.
- `signature` SHALL contain the signature performed by the attestation key.

The `TPS_COSE_Key` structure SHALL be the following:

```
TPS_COSE_Key = {
  1 => tstr / int,          ; kty shall be present
  ? 2 => bstr,              ; kid shall be present if it has been specified
  ? 3 => int,               ; alg shall be present if it has been specified
  ? 4 => [+ key_op ],       ; key_ops shall be present if it has been specified
  ? 5 => bstr,              ; base_iv shall not be present
  ? 512 => TPS_Key_params,  ; TPS_Key_params shall be present
  ? ec2_key_params,         ; if key is ec2 key, public key parameters shall be
                            ;  present, and private key parameters shall not be
  ? okp_key_params,         ; if key is okp key, public key parameters shall be
                            ;  present, and private key parameters shall not be
  ? symmetric_key_params,   ; shall not be present
  ? rsa_key_params          ; if key is rsa key, public key parameters shall be
                            ;  present, and private key parameers shall not be
  * $$tps_cose_key_ext      ; additional tps parameters may be present,
                            ;  verifier may ignore these values
}
```

The `TPS_Key_params` structure SHALL contain the following:

```
TPS_Key_params = {
  ? 1 => key_exportable,    ; shall not be present
  ? 2 => key_lifetime,      ; shall not be present
  ? 3 => ukid,              ; shall not be present
  ? 4 => key_size,          ; shall not be present
  ? 5 => hidden,            ; shall not be present
  ? 6 => challenge,         ; shall contain the challenge given in
                            ;  TPSK_AttestKeyRequest message
  * $$tps_key_param_ext     ; additional tps parameters may be present,
                                    ;  verifier may ignore these values
}
```

4.11.2 Generation Procedure

The TPS Client requests the generation of the key attestation using the `TPSK_AttestKeyRequest` command by identifying the key using its `ukid`, and providing a `challenge` parameter. Based on the request command, the TPS Keystore will generate the key attestation and return it in the `TPSK_AttestKeyResponse` together with the key attestation certificate chain.

4.11.3 Verification Procedure

Verification of the key attestation SHALL contain at least the following steps:

- Verifier SHALL successfully validate the key attestation certificate and its certificate chain with verifier's configured trusted certificate list.
- Verifier SHALL successfully validate the signature of the key attestation using the public key in the key attestation certificate.
- Verifier SHALL successfully compare the `challenge` parameter in the key attestation against the `challenge` parameter provided to the verifier by the verification requestor.

After the above steps, the verifier or the verification requestor can inspect the key and its properties in the `TPS_COSE_Key` structure and assume that the key is protected by the TPS Keystore implementation identified by the key attestation certificate.

4.12 Certification Validation Procedures

Certificate validation SHALL follow the steps specified in [PKIX] section 6.1, with the assumption that policy is `anyPolicy`.

5 CONFIGURATIONS

The TPS Keystore implementation uses configurations to indicate its capabilities, providing configuration names in the `TPS_GetFeatures_Rsp` message (see section 3.4.1).

This specification declares the configurations listed in this chapter. Other specifications MAY declare other configurations. If TPS Client encounters a configuration that it does not recognize, it SHALL ignore that configuration.

This specification uses the prefix "`gp-tps-`" in the configuration name. Any other conforming specification that specifies additional configurations SHALL NOT use the prefix "`gp-tps-`" in its configuration names.

Subsequent versions of this specification MAY add or change configurations.

5.1 Basic Configurations

By providing one or more of the following configuration names in a `TPS_GetFeatures_Rsp` message, a TPS Keystore implementation declares that it supports the indicated set of algorithms and operations.

Table 5-1: Basic Configurations

Basic Configuration	Supported Operations and Algorithms
`gp-tps-sha`	SHA-1, SHA-256, SHA-384, SHA-512 digest algorithms
`gp-tps-sha1`	SHA-1 digest algorithm
`gp-tps-sha2`	SHA-256, SHA-384, SHA-512 digest algorithms
`gp-tps-sha256`	SHA-256 digest algorithm
`gp-tps-sha384`	SHA-384 digest algorithm
`gp-tps-sha512`	SHA-512 digest algorithm
`gp-tps-shake`	SHAKE-128, SHAKE-256 digest algorithms
`gp-tps-shake128`	SHAKE-128 digest algorithm
`gp-tps-shake256`	SHAKE-256 digest algorithms
`gp-tps-gen-skey-128`	128-bit symmetric key generation
`gp-tps-gen-skey-192`	192-bit symmetric key generation
`gp-tps-gen-skey-256`	256-bit symmetric key generation
`gp-tps-gen-skey`	80- to 1024-bit symmetric key generation
`gp-tps-rsa`	RSA key pair generation; supported sizes 2048 bits, 3072 bits, 4096 bits Supported operations: RSASSA-PSS, RSASSA-PKCS1-v1_5, RSAES-OAEP with applicable supported digest algorithms
`gp-tps-rsa-2k`	RSA key pair generation; supported size 2048 bits Supported operations: RSASSA-PSS, RSASSA-PKCS1-v1_5, RSAES-OAEP with applicable supported digest algorithms
`gp-tps-rsa-3k`	RSA key pair generation; supported size 3072 bits Supported operations: RSASSA-PSS, RSASSA-PKCS1-v1_5, RSAES-OAEP with applicable supported digest algorithms
`gp-tps-rsa-4k`	RSA key pair generation; supported size 4096 bits Supported operations: RSASSA-PSS, RSASSA-PKCS1-v1_5, RSAES-OAEP with applicable supported digest algorithms
`gp-tps-p256`	NIST P-256 key pair generation Supported operations: ECDSA, ECDH
`gp-tps-p384`	NISP P-384 key pair generation Supported operations: ECDSA, ECDH
`gp-tps-p521`	NIST P-521 key pair generation Supported operations: ECDSA, ECDH

Basic Configuration	Supported Operations and Algorithms
`gp-tps-c25519`	Ed25519 key pair generation and EdDSA X25519 key pair generation and ECDH
`gp-tps-c448`	Ed448 key pair generation and EdDSA X448 key pair generation and ECDH
`gp-tps-ed25519`	Ed25519 key pair generation and EdDSA
`gp-tps-ed448`	Ed448 key pair generation and EdDSA
`gp-tps-x25519`	X25519 key pair generation and ECDH
`gp-tps-x448`	X448 key pair generation and ECDH
`gp-tps-aes-gcm`	AES-GCM encryption and decryption with applicable supported symmetric key lengths
`gp-tps-aes-ccm`	AES-CCM encryption and decryption with applicable supported symmetric key lengths
`gp-tps-hkdf`	HKDF with applicable supported digest algorithms
`gp-tps-mac`	HMAC with applicable supported digest algorithms, AES-CBC-MAC, AES-CMAC
`gp-tps-hmac`	HMAC with applicable supported digest algorithms
`gp-tps-aes-cbc-mac`	AES-CBC-MAC algorithm
`gp-tps-aes-cmac`	AES-CMAC algorithm
`gp-tps-ka+hkdf`	Combined key agreement and key derivation supported, meaning key agreement with any supported EC and Octet Key Pair, and key derivation with HKDF with applicable supported digest algorithm
`gp-tps-ka+aes-key-wrap`	Combined key agreement and key derivation supported, meaning key agreement with any supported EC and Octet Key Pair, and AES Key Wrap algorithm
`gp-tps-sm`	SM2 key pair generation and SM2DSA SM3 digest algorithm SM4-GCM encryption and decryption SM4-CCM encryption and decryption
`gp-tps-sm2`	SM2 key pair generation and SM2DSA
`gp-tps-sm3`	SM3 digest algorithm
`gp-tps-sm4-gcm`	SM4-GCM encryption and decryption
`gp-tps-sm4-ccm`	SM4-CCM encryption and decryption
`gp-tps-wrap-unwrap`	Key wrapping and unwrapping as specified in section 4.10
`gp-tps-key-attestation`	Key attestation as specified in section 4.11
`gp-tps-certificate-validation`	Certificate validation as specified in section 4.12

5.2 Grouped Configurations

The grouped configurations below MAY be used instead of the basic configurations listed in section 5.1.

Table 5-2: Grouped Configurations

Grouped Configurations	Supported Basic Configurations

`gp-tps-full`	`gp-tps-sha`
	`gp-tps-shake`
	`gp-tps-gen-skey`
	`gp-tps-rsa`
	`gp-tps-p256`
	`gp-tps-p384`
	`gp-tps-p521`
	`gp-tps-c25519`
	`gp-tps-c448`
	`gp-tps-hkdf`
	`gp-tps-mac`
	`gp-tps-aes-gcm`
	`gp-tps-aes-ccm`
	`gp-tps-ka+hkdf`
	`gp-tps-ka+aes-key-wrap`

`gp-tps-mini`	`gp-tps-sha`
	`gp-tps-gen-skey-128`
	`gp-tps-gen-skey-256`
	`gp-tps-p256`
	`gp-tps-aes-gcm`
	`gp-tps-hkdf`
	`gp-tps-ka+hkdf`

5.3 Examples (Informative)

TPS Keystore implementation SHALL use the `TPS_GetFeatures_Rsp` message (see section 3.4.1).

If a TPS Keystore implementation declares support for the following `profile_names`, then it claims to support the full configuration:

```
profile_names = [ "gp-tps-full" ]
```

If TPS Keystore implementation declares support for the following `profile_names`, then it claims to support the full configuration with key attestation, key wrapping and unwrapping, and certificate validation:

```
profile_names = [ "gp-tps-full",
                  "gp-tps-key-attestation",
                  "gp-tps-wrap-unwrap",
                  "gp-tps-certificate-validation" ]
```

If a TPS Keystore implementation declares support for the following `profile_names`, then it claims to support the full configuration and the Chinese algorithms:

```
profile_names = [ "gp-tps-full",
                  "gp-tps-sm" ]
```

If a TPS Keystore implementation declares support for the following `profile_names`, then it claims to support the mini configuration and additionally NIST P-521 Curve key pair generation and usage:

```
profile_names = [ "gp-tps-mini",
                  "gp-tps-p521" ]
```

Annex A ALGORITHM IDENTIFIER ASSIGNMENTS

A.1 Introduction

This specification uses COSE algorithm assignments registered with IANA by IETF ([IANA COSE]) that identify algorithms within the TPS Keystore protocol messages. However, some algorithms specified in this specification have not been specified by IETF, and therefore this specification assigns algorithm identifier values for them.

NOTE: If IETF assigns a COSE algorithm identifier value for any algorithm listed in Table A-1, GlobalPlatform will update this specification to include the newly assigned algorithm identifier values. In such an occurrence, GlobalPlatform assigned algorithm identifier values in Table A-1 will not be deprecated but can continue to be used alongside the IETF assigned algorithm identifier values.

A.2 Assignments

Table A-1: Algorithm Identifier Assignments

Algorithm Identifier	Value	Comments
`AES-CMAC`	-65537	Message authentication code algorithm identifier
`SM2`	-65538	Curve identifier
`SM3`	-65539	Hash function algorithm identifier
`SM2-DSA`	-65540	Signature algorithm identifier
`direct+HKDF-SM3`	-65541	Key derivation algorithm identifier
`SM4-GCM-128`	-65542	Encryption algorithm identifier
`SM4-CCM-16-64-128`	-65543	Encryption algorithm identifier
`SM4-CCM-64-64-128`	-65544	Encryption algorithm identifier
`SM4-CCM-16-128-128`	-65545	Encryption algorithm identifier
`SM4-CCM-64-128-128`	-65546	Encryption algorithm identifier
`AES-GCM + any`	-65547	Encryption algorithm identifier
`AES-CCM + any`	-65548	Encryption algorithm identifier
`SM4-GCM + any`	-65549	Encryption algorithm identifier
`SM4-CCM + any`	-65550	Encryption algorithm identifier