PSA Certified
Crypto API 1.3 PQC Extension

Document number: AES 0119

Release Quality: Beta

Issue Number: 1

Confidentiality: Non-confidential

Date of Issue: 10/06/2025

BETA RELEASE

This is a proposed update to the *PSA Certified Crypto API* [PSA-CRYPT] specification.

This is a BETA release in order to enable wider review and feedback on the changes proposed to be included in a future version of the specification.

At this quality level, the proposed changes and interfaces are complete, and suitable for initial product development. However, the specification is still subject to change.

Abstract

This document is part of the PSA Certified API specifications. It defines an extension to the Crypto API, to introduce support for Post-Quantum Cryptography (PQC) algorithms.

Contents

About this document

Release information

The change history table lists the changes that have been made to this document.

Table 1 Document revision history

Date	Version	Confidentiality	Change
March 2025	Beta 0	Non-confidential	Initial release of the 1.3 PQC Extension specification
June 2025	Beta 1	Non-confidential	Added clarifications

The detailed changes in each release are described in *Document change history* on page 53.

PSA Certified Crypto API

Copyright © 2024-2025 Arm Limited and/or its affiliates. The copyright statement reflects the fact that some draft issues of this document have been released, to a limited circulation.

License

Text and illustrations

Text and illustrations in this work are licensed under Attribution-ShareAlike 4.0 International (CC BY-SA 4.0). To view a copy of the license, visit creativecommons.org/licenses/by-sa/4.0.

Grant of patent license. Subject to the terms and conditions of this license (both the CC BY-SA 4.0 Public License and this Patent License), each Licensor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Licensed Material, where such license applies only to those patent claims licensable by such Licensor that are necessarily infringed by their contribution(s) alone or by combination of their contribution(s) with the Licensed Material to which such contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Licensed Material or a contribution incorporated within the Licensed Material constitutes direct or contributory patent infringement, then any licenses granted to You under this license for that Licensed Material shall terminate as of the date such litigation is filed.

The Arm trademarks featured here are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Please visit arm.com/company/policies/trademarks for more information about Arm's trademarks.

About the license

The language in the additional patent license is largely identical to that in section 3 of the Apache License, Version 2.0 (Apache 2.0), with two exceptions:

1. Changes are made related to the defined terms, to align those defined terms with the terminology in CC BY-SA 4.0 rather than Apache 2.0 (for example, changing "Work" to "Licensed Material").

2. The scope of the defensive termination clause is changed from "any patent licenses granted to You" to "any licenses granted to You". This change is intended to help maintain a healthy ecosystem by providing additional protection to the community against patent litigation claims.

To view the full text of the Apache 2.0 license, visit apache.org/licenses/LICENSE-2.0.

Source code

Source code samples in this work are licensed under the Apache License, Version 2.0 (the "License"); you may not use such samples except in compliance with the License. You may obtain a copy of the License at apache.org/licenses/LICENSE-2.0.

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and limitations under the License.

References

This document refers to the following documents.

Table 2 Documents referenced by this document

Ref	Document Number	Title
[PSA-CRYPT]	IHI 0086	*PSA Certified Crypto API*. arm-software.github.io/psa-api/crypto
[FIPS180-4]		NIST, *FIPS Publication 180-4: Secure Hash Standard (SHS)*, August 2015. doi.org/10.6028/NIST.FIPS.180-4
[FIPS202]		NIST, *FIPS Publication 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, August 2015. doi.org/10.6028/NIST.FIPS.202
[FIPS203]		NIST, *FIPS Publication 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard*, August 2024. doi.org/10.6028/NIST.FIPS.203
[FIPS204]		NIST, *FIPS Publication 204: Module-Lattice-Based Digital Signature Standard*, August 2024. doi.org/10.6028/NIST.FIPS.204
[FIPS205]		NIST, *FIPS Publication 205: Stateless Hash-Based Digital Signature Standard*, August 2024. doi.org/10.6028/NIST.FIPS.205
[LAMPS-MLKEM]		IETF, *Internet X.509 Public Key Infrastructure - Algorithm Identifiers for Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM)*, November 2024 (Draft 06). datatracker.ietf.org/doc/draft-ietf-lamps-kyber-certificates
[LAMPS-MLDSA]		IETF, *Internet X.509 Public Key Infrastructure: Algorithm Identifiers for ML-DSA*, November 2024 (Draft 05). datatracker.ietf.org/doc/draft-ietf-lamps-dilithium-certificates
[LAMPS-SLHDSA]		IETF, *Internet X.509 Public Key Infrastructure: Algorithm Identifiers for SLH-DSA*, November 2024 (Draft 03). datatracker.ietf.org/doc/draft-ietf-lamps-x509-slhdsa
[NIST-PQC]		NIST, *Post-Quantum Cryptography*, PQC Project page. nist.gov/pqcrypto
[SP800-208]		NIST, *NIST Special Publication 800-208: Recommendation for Stateful Hash-Based Signature Schemes*, October 2020. doi.org/10.6028/NIST.SP.800-208
[RFC8391]		IRTF, *XMSS: eXtended Merkle Signature Scheme*, May 2018. tools.ietf.org/html/rfc8391
[RFC8554]		IRTF, *Leighton-Micali Hash-Based Signatures*, April 2019. tools.ietf.org/html/rfc8554
[CFRG-LMS]		IRTF, *Additional Parameter sets for HSS/LMS Hash-Based Signatures*, October 2024 (Draft 17). datatracker.ietf.org/doc/draft-fluhrer-lms-more-parm-sets

Table 2 – continued from previous page

Ref	Document Number	Title
[LAMPS-SHBS]		IETF, *Use of the HSS and XMSS Hash-Based Signature Algorithms in Internet X.509 Public Key Infrastructure*, December 2024 (Draft 13). datatracker.ietf.org/doc/draft-ietf-lamps-x509-shbs

Terms and abbreviations

This document uses the following terms and abbreviations.

Table 3 Terms and abbreviations

Term	Meaning
AEAD	See *Authenticated Encryption with Associated Data*.
Algorithm	A finite sequence of steps to perform a particular operation. In this specification, an algorithm is a *cipher* or a related function. Other texts call this a cryptographic mechanism.
API	Application Programming Interface.
Asymmetric	See *Public-key cryptography*.
Authenticated Encryption with Associated Data (AEAD)	A type of encryption that provides confidentiality and authenticity of data using *symmetric* keys.
Byte	In this specification, a unit of storage comprising eight bits, also called an octet.
Cipher	An algorithm used for encryption or decryption with a *symmetric* key.
Cryptoprocessor	The component that performs cryptographic operations. A cryptoprocessor might contain a *keystore* and countermeasures against a range of physical and timing attacks.
Hash	A cryptographic hash function, or the value returned by such a function.
HMAC	A type of *MAC* that uses a cryptographic key with a *hash* function.
IMPLEMENTATION DEFINED	Behavior that is not defined by the architecture, but is defined and documented by individual implementations.
Initialization vector (IV)	An additional input that is not part of the message. It is used to prevent an attacker from making any correlation between cipher text and plain text. This specification uses the term for such initial inputs in all contexts. For example, the initial counter in CTR mode is called the IV.
IV	See *Initialization vector*.
KDF	See *Key Derivation Function*.
Key agreement	An algorithm for two or more parties to establish a common secret key.

Table 3 – continued from previous page

Term	Meaning
Key Derivation Function (KDF)	Key Derivation Function. An algorithm for deriving keys from secret material.
Key identifier	A reference to a cryptographic key. Key identifiers in the Crypto API are 32-bit integers.
Key policy	Key metadata that describes and restricts what a key can be used for.
Key size	The size of a key as defined by common conventions for each key type. For keys that are built from several numbers of strings, this is the size of a particular one of these numbers or strings. This specification expresses key sizes in bits.
Key type	Key metadata that describes the structure and content of a key.
Keystore	A hardware or software component that protects, stores, and manages cryptographic keys.
Lifetime	Key metadata that describes when a key is destroyed.
MAC	See *Message Authentication Code*.
Message Authentication Code (MAC)	A short piece of information used to authenticate a message. It is created and verified using a *symmetric* key.
Message digest	A *hash* of a message. Used to determine if a message has been tampered.
Multi-part operation	An *API* which splits a single cryptographic operation into a sequence of separate steps.
Non-extractable key	A key with a *key policy* that prevents it from being read by ordinary means.
Nonce	Used as an input for certain *AEAD* algorithms. Nonces must not be reused with the same key because this can break a cryptographic protocol.
Persistent key	A key that is stored in protected non-volatile memory.
Post-Quantum Cryptography (PQC)	A cryptographic scheme that relies on mathematical problems that do not have efficient algorithms for either classical or quantum computing.
PQC	See *Post-Quantum Cryptography*.
PSA	Platform Security Architecture
Public-key cryptography	A type of cryptographic system that uses key pairs. A keypair consists of a (secret) private key and a public key (not secret). A public-key cryptographic algorithm can be used for key distribution and for digital signatures.
Salt	Used as an input for certain algorithms, such as key derivations.
Signature	The output of a digital signature scheme that uses an *asymmetric* keypair. Used to establish who produced a message.
Single-part function	An *API* that implements the cryptographic operation in a single function call.
SPECIFICATION DEFINED	Behavior that is defined by this specification.

Table 3 – continued from previous page

Term	Meaning
Symmetric	A type of cryptographic algorithm that uses a single key. A symmetric key can be used with a block cipher or a stream cipher.
Volatile key	A key that has a short lifespan and is guaranteed not to exist after a restart of an application instance.

Conventions

Typographical conventions

The typographical conventions are:

italic	Introduces special terminology, and denotes citations.
`monospace`	Used for assembler syntax descriptions, pseudocode, and source code examples.
	Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.
SMALL CAPITALS	Used for some common terms such as IMPLEMENTATION DEFINED.
	Used for a few terms that have specific technical meanings, and are included in the *Terms and abbreviations*.
Red text	Indicates an open issue.
Blue text	Indicates a link. This can be
	• A cross-reference to another location within the document
	• A URL, for example example.com

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by 0b, and hexadecimal numbers by `0x`.

In both cases, the prefix and the associated value are written in a monospace font, for example `0xFFFF0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

Current status and anticipated changes

This document is at Beta quality status which has a particular meaning to Arm of which the recipient must be aware. A Beta quality specification will be sufficiently stable & committed for initial product development, however all aspects of the architecture described herein remain SUBJECT TO CHANGE. Please ensure that you have the latest revision.

Feedback

We welcome feedback on the PSA Certified API documentation.

If you have comments on the content of this book, visit github.com/arm-software/psa-api/issues to create a new issue at the PSA Certified API GitHub project. Give:

- The title (Crypto API).
- The number and issue (AES 0119 1.3 PQC Extension Beta (Issue 1)).
- The location in the document to which your comments apply.
- A concise explanation of your comments.

We also welcome general suggestions for additions and improvements.

1 Introduction

1.1 About Platform Security Architecture

This document is one of a set of resources provided by Arm that can help organizations develop products that meet the security requirements of PSA Certified on Arm-based platforms. The PSA Certified scheme provides a framework and methodology that helps silicon manufacturers, system software providers and OEMs to develop more secure products. Arm resources that support PSA Certified range from threat models, standard architectures that simplify development and increase portability, and open-source partnerships that provide ready-to-use software. You can read more about PSA Certified here at www.psacertified.org and find more Arm resources here at developer.arm.com/platform-security-resources.

1.2 About the Crypto API PQC Extension

This document defines an extension to the *PSA Certified Crypto API* [PSA-CRYPT] specification, to provide support for *Post-Quantum Cryptography* (PQC) algorithms. Specifically, for the NIST-approved schemes for LMS, HSS, XMSS, XMSS^{MT}, ML-DSA, SLH-DSA, and ML-KEM.

When the proposed extension is sufficiently stable to be classed as Final, it will be integrated into a future version of [PSA-CRYPT].

This specification must be read and implemented in conjunction with [PSA-CRYPT]. All of the conventions, design considerations, and implementation considerations that are described in [PSA-CRYPT] apply to this specification.

1.3 Objectives for the PQC Extension

1.3.1 Background

The justification for developing new *public-key cryptography* algorithms due to the risks posed by quantum computing are described by NIST in *Post-Quantum Cryptography* [NIST-PQC].

Extract from *Post-Quantum Cryptography*:

In recent years, there has been a substantial amount of research on quantum computers — machines that exploit quantum mechanical phenomena to solve mathematical problems that are difficult or intractable for conventional computers. If large-scale quantum computers are ever built, they will be able to break many of the public-key cryptosystems currently in use. This would seriously compromise the confidentiality and integrity of digital communications on the Internet and elsewhere. The goal of post-quantum cryptography (also called quantum-resistant cryptography) is to develop cryptographic systems that are secure against both quantum and classical computers, and can interoperate with existing communications protocols and networks.

The question of when a large-scale quantum computer will be built is a complicated one. While in the past it was less clear that large quantum computers are a physical possibility, many scientists now believe it to be

---

merely a significant engineering challenge. Some engineers even predict that within the next twenty or so years sufficiently large quantum computers will be built to break essentially all public key schemes currently in use. Historically, it has taken almost two decades to deploy our modern public key cryptography infrastructure. Therefore, regardless of whether we can estimate the exact time of the arrival of the quantum computing era, we must begin now to prepare our information security systems to be able to resist quantum computing.

NIST is hosting a project to collaboratively develop, analyze, refine, and select cryptographic schemes that are resistant to attack by both classical and quantum computing.

1.3.2 Selection of algorithms

NIST PQC project finalists

PQC algorithms that have been standardized are obvious candidates for inclusion in the Crypto API. The current set of standards is the following:

- *FIPS Publication 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard* [FIPS203]
- *FIPS Publication 204: Module-Lattice-Based Digital Signature Standard* [FIPS204]
- *FIPS Publication 205: Stateless Hash-Based Digital Signature Standard* [FIPS205]

Although the NIST standards for these algorithms are now finalized, the definition of keys in the Crypto API depends on import and export formats. To maximize key exchange interoperability with other specifications, the default export format in the Crypto API should be aligned with the definitions selected for X.509 public-key infrastructure. As the IETF process for defining the X.509 key formats is still ongoing at the time of publishing this document, the interfaces within this document are at BETA status.

However, it is not expected that other aspects of the API in this document will change when it becomes FINAL.

Note:

Although PQC algorithms that are draft standards could be considered, any definitions for these algorithms would be have to be considered experimental. Significant aspects of the algorithm, such as approved parameter sets, can change before publication of a final standard, potentially requiring a revision of any proposed interface for the Crypto API.

Other NIST-approved schemes

In *NIST Special Publication 800-208: Recommendation for Stateful Hash-Based Signature Schemes* [SP800-208], NIST approved use of the following stateful hash-based signature (HBS) schemes:

- The Leighton-Micali Signature (LMS) system, and its multi-tree variant, the Hierarchical Signature System (HSS/LMS). These are defined in *Leighton-Micali Hash-Based Signatures* [RFC8554].
- The eXtended Merkle Signature Scheme (XMSS), and its multi-tree variant XMSS^{MT}. These are defined in *XMSS: eXtended Merkle Signature Scheme* [RFC8391].

HBS schemes have additional challenges with regards to deploying secure and resilient systems for signing operations. These challenges, outlined in [SP800-208] sections §1.2 and §8.1, result in a recommendation

to use these schemes in a limited set of use cases, for example, authentication of firmware in constrained devices.

At present, it is not expected that the Crypto API will be used to create HBS private keys, or to carry out signing operations. However, there is a use case with the Crypto API for verification of HBS signatures. Therefore, for these HBS schemes, the Crypto API only provides support for public keys and signature verification algorithms.

2 API Reference

This chapter is divided into sections for each of the PQC algorithms in the Crypto API:

2.1 Additional Hash algorithms

These algorithms extend those defined in *PSA Certified Crypto API* [PSA-CRYPT] §10.2 *Message digests*. They are used with the hash functions and multi-part operations, or combined with composite algorithms that are parameterized by a hash algorithm.

2.1.1 SHA-256-based hash algorithms

`PSA_ALG_SHA_256_192` (macro)

The SHA-256/192 message digest algorithm.

Added in version 1.3.

```
#define PSA_ALG_SHA_256_192 ((psa_algorithm_t)0x0200000E)
```

SHA-256/192 is the first 192 bits (24 bytes) of the SHA-256 output. SHA-256 is defined in [FIPS180-4].

2.1.2 SHAKE-based hash algorithms

`PSA_ALG_SHAKE128_256` (macro)

The SHAKE128/256 message digest algorithm.

Added in version 1.3.

```
#define PSA_ALG_SHAKE128_256 ((psa_algorithm_t)0x02000016)
```

SHAKE128/256 is the first 256 bits (32 bytes) of the SHAKE128 output. SHAKE128 is defined in [FIPS202].

This can be used as pre-hashing for SLH-DSA (see `PSA_ALG_HASH_SLH_DSA()`).

> **Note:**
>
> For other scenarios where a hash function based on SHA3 or SHAKE is required, SHA3-256 is recommended. SHA3-256 has the same output size, and a theoretically higher security strength.

PSA_ALG_SHAKE256_192 (macro)

The SHAKE256/192 message digest algorithm.

Added in version 1.3.

```
#define PSA_ALG_SHAKE256_192 ((psa_algorithm_t)0x02000017)
```

SHAKE256/192 is the first 192 bits (24 bytes) of the SHAKE256 output. SHAKE256 is defined in [FIPS202].

PSA_ALG_SHAKE256_256 (macro)

The SHAKE256/256 message digest algorithm.

Added in version 1.3.

```
#define PSA_ALG_SHAKE256_256 ((psa_algorithm_t)0x02000018)
```

SHAKE256/256 is the first 256 bits (32 bytes) of the SHAKE256 output. SHAKE256 is defined in [FIPS202].

2.2 Module Lattice-based key encapsulation

2.2.1 Module Lattice-based key-encapsulation keys

The Crypto API supports Module Lattice-based key ecapsulation (ML-KEM) as defined in *FIPS Publication 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard* [FIPS203].

PSA_KEY_TYPE_ML_KEM_KEY_PAIR (macro)

ML-KEM key pair: both the decapsulation and encapsulation key.

Added in version 1.3.

```
#define PSA_KEY_TYPE_ML_KEM_KEY_PAIR ((psa_key_type_t)0x7004)
```

The Crypto API treats decapsulation keys as private keys and encapsulation keys as public keys.

The key attribute size of an ML-KEM key is specified by the numeric part of the parameter-set identifier defined in [FIPS203]. The parameter-set identifier refers to the key strength, and not to the actual size of the key. The following values for the `key_bits` key attribute are used to select a specific ML-KEM parameter set:

- ML-KEM-512 : `key_bits = 512`
- ML-KEM-768 : `key_bits = 768`
- ML-KEM-1024 : `key_bits = 1024`

See also §8 in [FIPS203].

Compatible algorithms

- `PSA_ALG_ML_KEM`

Key format

> ⚠️ **Warning**
>
> The key format may change in a final version of this API. The standardization of exchange formats for ML-KEM public and private keys is in progress, but final documents have not been published. See *Internet X.509 Public Key Infrastructure - Algorithm Identifiers for Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM)* [LAMPS-MLKEM].
>
> The current proposed format is based on the expected outcome of that process.

An ML-KEM key pair is the (ek, dk) pair of encapsulation key and decapsulation key, which are generated from two secret 32-byte seeds, d and z. See [FIPS203] §7.1.

The data format for import and export of the key pair is the concatenation of the two seed values: $d \parallel z$.

> **Rationale**
>
> The IETF working group responsible for defining the format of the ML-DSA keys in *SubjectPublicKeyInfo* and *OneAsymmetricKey* structures is discussing the formats at present (September 2024), with the current consensus to using just the seed values as the private key, for the following reasons:
>
> - ML-KEM decapsulation keys are 1.5–3.0 kB in size, but can be recomputed efficiently from the initial 64-byte seed-pair.
> - There is no need to validate an imported ML-KEM key pair — every 64-byte pair of seed values is valid.
> - It is better for the standard to choose a single format to improve interoperability.

See `PSA_KEY_TYPE_ML_KEM_PUBLIC_KEY` for the data format used when exporting the public key with `psa_export_public_key()`.

> **Implementation note**
>
> An implementation can optionally compute and store the dk value, which also contains the encapsulation key ek, to accelerate operations that use the key. It is recommended that an implementation retains the seed pair (d, z) with the decapsulation key, in order to export the key, or copy the key to a different location.

Key derivation

A call to `psa_key_derivation_output_key()` will construct an ML-KEM key pair using the following process:

1. Draw 32 bytes of output as the seed value d.
2. Draw 32 bytes of output as the seed value z.

The key pair (ek, dk) is generated from the seed as defined by `ML-KEM.KeyGen_internal()` in [FIPS203] §6.1.

PSA_KEY_TYPE_ML_KEM_PUBLIC_KEY (macro)

ML-KEM public (encapsulation) key.

Added in version 1.3.

```
#define PSA_KEY_TYPE_ML_KEM_PUBLIC_KEY ((psa_key_type_t)0x4004)
```

The key attribute size of an ML-KEM public key is the same as the corresponding private key. See PSA_KEY_TYPE_ML_KEM_KEY_PAIR.

Compatible algorithms

- PSA_ALG_ML_KEM (encapsulation only)

Key format

> ⚠️ **Warning**
>
> The key format may change in a final version of this API. The standardization of exchange formats for ML-KEM public and private keys is in progress, but final documents have not been published. See *Internet X.509 Public Key Infrastructure - Algorithm Identifiers for Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM)* [LAMPS-MLKEM].
>
> The current proposed format is based on the expected outcome of that process.

An ML-KEM public key is the ek output of `ML-KEM.KeyGen()`, defined in [FIPS203] §7.1.

The size of the public key depends on the ML-KEM parameter set as follows:

Parameter set	Public-key size in bytes
ML-KEM-512	800
ML-KEM-768	1184
ML-KEM-1024	1568

PSA_KEY_TYPE_IS_ML_KEM (macro)

Whether a key type is an ML-DSA key, either a key pair or a public key.

Added in version 1.3.

```
#define PSA_KEY_TYPE_IS_ML_KEM(type) /* specification-defined value */
```

Parameters

`type`	A key type: a value of type `psa_key_type_t`.

2.2.2 Module Lattice-based key-encapsulation algorithm

These algorithms extend those defined in *PSA Certified Crypto API* [PSA-CRYPT] §10.10 *Key encapsulation*, for use with the key-encapsulation functions.

Note:

The key-encapsulation functions, `psa_encapsulate()` and `psa_decapsulate()`, were introduced in version 1.3 of the Crypto API.

ML-KEM is defined in *FIPS Publication 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard* [FIPS203]. ML-KEM has three parameter sets which provide differing security strengths.

The generation of an ML-KEM key depends on the full parameter specification. The encoding of each parameter set into the key attributes is described in *Module Lattice-based key-encapsulation keys* on page 14.

See [FIPS203] §8 for details on the parameter sets.

`PSA_ALG_ML_KEM` (macro)

Module Lattice-based key-encapsulation mechanism (ML-KEM).

Added in version 1.3.

```
#define PSA_ALG_ML_KEM ((psa_algorithm_t)0x0c000200)
```

This is the ML-KEM key-encapsulation algorithm, defined by [FIPS203]. ML-KEM requires an ML-KEM key, which determines the ML-KEM parameter set for the operation.

When using ML-KEM, the size of the encapsulation data returned by a call to `psa_encapsulate()` is as follows:

Parameter set	Encapsulation data size in bytes
ML-KEM-512	768
ML-KEM-768	1088
ML-KEM-1024	1568

The 32-byte shared output key that is produced by ML-KEM is pseudorandom. Although it can be used directly as an encryption key, it is recommended to use the output key as an input to a key-derivation operation to produce additional cryptographic keys.

Compatible key types

`PSA_KEY_TYPE_ML_KEM_KEY_PAIR`

`PSA_KEY_TYPE_ML_KEM_PUBLIC_KEY` (encapsulation only)

2.3 Module Lattice-based signatures

2.3.1 Module Lattice-based signature keys

The Crypto API supports Module Lattice-based digital signatures (ML-DSA), as defined in *FIPS Publication 204: Module-Lattice-Based Digital Signature Standard* [FIPS204].

PSA_KEY_TYPE_ML_DSA_KEY_PAIR (macro)

ML-DSA key pair: both the private and public key.

Added in version 1.3.

```
#define PSA_KEY_TYPE_ML_DSA_KEY_PAIR ((psa_key_type_t)0x7002)
```

The key attribute size of an ML-DSA key is a measure of the security strength of the ML-DSA parameter set in [FIPS204]:

- ML-DSA-44 : `key_bits = 128`
- ML-DSA-65 : `key_bits = 192`
- ML-DSA-87 : `key_bits = 256`

See also §4 in [FIPS204].

Compatible algorithms

- PSA_ALG_ML_DSA
- PSA_ALG_HASH_ML_DSA
- PSA_ALG_DETERMINISTIC_ML_DSA
- PSA_ALG_DETERMINISTIC_HASH_ML_DSA

Key format

> ⚠️ **Warning**
>
> The key format may change in a final version of this API. The standardization of exchange formats for ML-DSA public and private keys is in progress, but final documents have not been published. See *Internet X.509 Public Key Infrastructure: Algorithm Identifiers for ML-DSA* [LAMPS-MLDSA].
>
> The current proposed format is based on the expected outcome of that process.

An ML-DSA key pair is the (pk, sk) pair of public key and secret key, which are generated from a secret 32-byte seed, ξ. See [FIPS204] §5.1.

The data format for import and export of the key pair is the 32-byte seed ξ.

> **Rationale**
>
> The IETF working group responsible for defining the format of the ML-DSA keys in *SubjectPublicKeyInfo* and *OneAsymmetricKey* structures is discussing the formats at present (September 2024), with the current consensus to using just the seed value as the private key, for the following reasons:

- ML-DSA key pairs are several kB in size, but can be recomputed efficiently from the initial 32-byte seed.
- There is no need to validate an imported ML-DSA private key — every 32-byte seed values is valid.
- The public key cannot be derived from the secret key, so a key pair must store both the secret key and the public key. The size of the key pair depends on the ML-DSA parameter set as follows:

Parameter set	Key-pair size in bytes
ML-DSA-44	3872
ML-DSA-65	5984
ML-DSA-87	7488

- It is better for the standard to choose a single format to improve interoperability.

See `PSA_KEY_TYPE_ML_DSA_PUBLIC_KEY` for the data format used when exporting the public key with `psa_export_public_key()`.

Implementation note

An implementation can optionally compute and store the (pk, sk) values, to accelerate operations that use the key. It is recommended that an implementation retains the seed ξ with the key pair, in order to export the key, or copy the key to a different location.

Key derivation

A call to `psa_key_derivation_output_key()` will draw 32 bytes of output and use these as the 32-byte ML-DSA key-pair seed, xi. The key pair (pk, sk) is generated from the seed as defined by `ML-DSA.KeyGen_internal()` in [FIPS204] §6.1.

Implementation note

It is IMPLEMENTATION DEFINED whether the seed xi is expanded to (pk, sk) at the point of derivation, or only just before the key is used.

`PSA_KEY_TYPE_ML_DSA_PUBLIC_KEY` (macro)

ML-DSA public key.

Added in version 1.3.

```
#define PSA_KEY_TYPE_ML_DSA_PUBLIC_KEY ((psa_key_type_t)0x4002)
```

The key attribute size of an ML-DSA public key is the same as the corresponding private key. See `PSA_KEY_TYPE_ML_DSA_KEY_PAIR`.

Compatible algorithms

- PSA_ALG_ML_DSA
- PSA_ALG_HASH_ML_DSA
- PSA_ALG_DETERMINISTIC_ML_DSA
- PSA_ALG_DETERMINISTIC_HASH_ML_DSA

Key format

> ⚠️ **Warning**
>
> The key format may change in a final version of this API. The standardization of exchange formats for ML-DSA public and private keys is in progress, but final documents have not been published. See *Internet X.509 Public Key Infrastructure: Algorithm Identifiers for ML-DSA* [LAMPS-MLDSA].
>
> The current proposed format is based on the expected outcome of that process.

An ML-DSA public key is the pk output of `ML-DSA.KeyGen()`, defined in [FIPS204] §5.1.

The size of the public key depends on the ML-DSA parameter set as follows:

Parameter set	Public-key size in bytes
ML-DSA-44	1312
ML-DSA-65	1952
ML-DSA-87	2592

`PSA_KEY_TYPE_IS_ML_DSA` (macro)

Whether a key type is an ML-DSA key, either a key pair or a public key.

Added in version 1.3.

```
#define PSA_KEY_TYPE_IS_ML_DSA(type) /* specification-defined value */
```

Parameters

`type`	A key type: a value of type `psa_key_type_t`.

2.3.2 Module Lattice-based signature algorithms

These algorithms extend those defined in *PSA Certified Crypto API* [PSA-CRYPT] §10.7 *Asymmetric signature*, for use with the signature functions.

The ML-DSA signature and verification scheme is defined in *FIPS Publication 204: Module-Lattice-Based Digital Signature Standard* [FIPS204]. ML-DSA has three parameter sets which provide differing security strengths.

ML-DSA keys are large: 1.2–2.5kB for the public key, and triple that for the key pair. ML-DSA signatures are much larger than those for RSA and Elliptic curve schemes, between 2.4kB and 4.6kB, depending on the selected parameter set.

See [FIPS204] §4 for details on the parameter sets, and the key and generated signature sizes.

The generation of an ML-DSA key depends on the full parameter specification. The encoding of each parameter set into the key attributes is described in *Module Lattice-based signature keys* on page 18.

[FIPS204] defines pure and pre-hashed variants of the signature scheme, which can either be hedged (randomized) or deterministic. Four algorithms are defined to support these variants: `PSA_ALG_ML_DSA`, `PSA_ALG_DETERMINISTIC_ML_DSA`, `PSA_ALG_HASH_ML_DSA()`, and `PSA_ALG_DETERMINISTIC_HASH_ML_DSA()`.

Hedged and deterministic signatures

Hedging incorporates fresh randomness in the signature computation, resulting in distinct signatures on every signing operation when given identical inputs. Deterministic signatures do not require additional random data, and result in an identical signature for the same inputs.

Signature verification does not distinguish between a hedged and a deterministic signature. Either hedged or deterministic algorithms can be used when verifying a signature.

When computing a signature, the key's permitted-algorithm policy must match the requested algorithm, treating hedged and deterministic versions as distinct. When verifying a signature, the hedged and deterministic versions of each algorithm are considered equivalent when checking the key's permitted-algorithm policy.

Note:

The hedged version provides message secrecy and some protection against side-channels. [FIPS204] recommends that users should use the hedged version if either of these issues are a concern. The deterministic variant should only be used if the implementation does not include any source of randomness.

Implementation note

[FIPS204] recommends that implementations use an approved random number generator to provide the random value in the hedged version. However, it notes that use of the hedged variant with a weak RNG is generally preferable to the deterministic variant.

Rationale

The use of fresh randomness, or not, when computing a signature seems like an implementation decision based on the capability of the system, and its vulnerability to specific threats, following the recommendations in [FIPS204].

However, the Crypto API gives distinct algorithm identifiers for the hedged and deterministic variants, to enable an application use case to require a specific variant.

Pure and pre-hashed algorithms

The pre-hashed signature computation *HashML-DSA* generates distinct signatures to a pure signature *ML-DSA*, with the same key and message hashing algorithm.

An ML-DSA signature can only be verified with an ML-DSA algorithm. A HashML-DSA signature can only be verified with a HashML-DSA algorithm.

Contexts

Contexts are not supported in the current version of this specification because there is no suitable signature interface that can take the context as a parameter. A empty context string is used when computing or verifying ML-DSA signatures.

A future version of this specification may add suitable functions and extend this algorithm to support contexts.

`PSA_ALG_ML_DSA` (macro)

Module lattice-based digital signature algorithm without pre-hashing (ML-DSA).

Added in version 1.3.

```
#define PSA_ALG_ML_DSA ((psa_algorithm_t) 0x06004400)
```

This algorithm can only be used with the `psa_sign_message()` and `psa_verify_message()` functions.

This is the pure ML-DSA digital signature algorithm, defined by *FIPS Publication 204: Module-Lattice-Based Digital Signature Standard* [FIPS204], using hedging. ML-DSA requires an ML-DSA key, which determines the ML-DSA parameter set for the operation.

This algorithm is randomized: each invocation returns a different, equally valid signature. See the notes on hedged signatures.

When `PSA_ALG_ML_DSA` is used as a permitted algorithm in a key policy, this permits:

- `PSA_ALG_ML_DSA` as the algorithm in a call to `psa_sign_message()`.
- `PSA_ALG_ML_DSA` or `PSA_ALG_DETERMINISTIC_ML_DSA` as the algorithm in a call to `psa_verify_message()`.

Note:

To sign or verify the pre-computed hash of a message using ML-DSA, the HashML-DSA algorithms (`PSA_ALG_HASH_ML_DSA()` and `PSA_ALG_DETERMINISTIC_HASH_ML_DSA()`) can also be used with `psa_sign_hash()` and `psa_verify_hash()`.

The signature produced by HashML-DSA is distinct from that produced by ML-DSA.

Compatible key types

`PSA_KEY_TYPE_ML_DSA_KEY_PAIR`
`PSA_KEY_TYPE_ML_DSA_PUBLIC_KEY` (signature verification only)

`PSA_ALG_DETERMINISTIC_ML_DSA` (macro)

Deterministic module lattice-based digital signature algorithm without pre-hashing (ML-DSA).

Added in version 1.3.

```
#define PSA_ALG_DETERMINISTIC_ML_DSA ((psa_algorithm_t) 0x06004500)
```

This algorithm can only be used with the `psa_sign_message()` and `psa_verify_message()` functions.

This is the pure ML-DSA digital signature algorithm, defined by *FIPS Publication 204: Module-Lattice-Based Digital Signature Standard* [FIPS204], without hedging. ML-DSA requires an ML-DSA key, which determines the ML-DSA parameter set for the operation.

This algorithm is deterministic: each invocation with the same inputs returns an identical signature.

> ⚠️ **Warning**
>
> It is recommended to use the hedged `PSA_ALG_ML_DSA` algorithm instead, when supported by the implementation. See the notes on deterministic signatures.

When `PSA_ALG_DETERMINISTIC_ML_DSA` is used as a permitted algorithm in a key policy, this permits:

- `PSA_ALG_DETERMINISTIC_ML_DSA` as the algorithm in a call to `psa_sign_message()`.

- `PSA_ALG_ML_DSA` or `PSA_ALG_DETERMINISTIC_ML_DSA` as the algorithm in a call to `psa_verify_message()`.

Note:

To sign or verify the pre-computed hash of a message using ML-DSA, the HashML-DSA algorithms (`PSA_ALG_HASH_ML_DSA()` and `PSA_ALG_DETERMINISTIC_HASH_ML_DSA()`) can also be used with `psa_sign_hash()` and `psa_verify_hash()`.

The signature produced by HashML-DSA is distinct from that produced by ML-DSA.

Compatible key types

`PSA_KEY_TYPE_ML_DSA_KEY_PAIR`

`PSA_KEY_TYPE_ML_DSA_PUBLIC_KEY` (signature verification only)

`PSA_ALG_HASH_ML_DSA` (macro)

Module lattice-based digital signature algorithm with pre-hashing (HashML-DSA).

Added in version 1.3.

```
#define PSA_ALG_HASH_ML_DSA(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true. This includes `PSA_ALG_ANY_HASH` when specifying the algorithm in a key policy.

Returns

The corresponding HashML-DSA signature algorithm, using `hash_alg` to pre-hash the message.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This algorithm can be used with both the message and hash signature functions.

This is the pre-hashed ML-DSA digital signature algorithm, defined by *FIPS Publication 204: Module-Lattice-Based Digital Signature Standard* [FIPS204], using hedging. ML-DSA requires an ML-DSA key, which determines the ML-DSA parameter set for the operation.

> **Note:**
>
> For the pre-hashing, [FIPS204] §5.4 recommends the use of an approved hash function with an equivalent, or better, security strength than the chosen ML-DSA parameter set.

This algorithm is randomized: each invocation returns a different, equally valid signature. See the notes on hedged signatures.

When `PSA_ALG_HASH_ML_DSA()` is used as a permitted algorithm in a key policy, this permits:

- `PSA_ALG_HASH_ML_DSA()` as the algorithm in a call to `psa_sign_message()` and `psa_sign_hash()`.
- `PSA_ALG_HASH_ML_DSA()` or `PSA_ALG_DETERMINISTIC_HASH_ML_DSA()` as the algorithm in a call to `psa_verify_message()` and `psa_verify_hash()`.

> **Note:**
>
> The signature produced by HashML-DSA is distinct from that produced by ML-DSA.

Usage

This is a hash-and-sign algorithm. To calculate a signature, use one of the following approaches:

- Call `psa_sign_message()` with the message.
- Calculate the hash of the message with `psa_hash_compute()`, or with a multi-part hash operation, using the `hash_alg` hash algorithm. Note that `hash_alg` can be extracted from the signature algorithm using `PSA_ALG_GET_HASH(sig_alg)`. Then sign the calculated hash with `psa_sign_hash()`.

Verifying a signature is similar, using `psa_verify_message()` or `psa_verify_hash()` instead of the signature function.

Compatible key types

`PSA_KEY_TYPE_ML_DSA_KEY_PAIR`

`PSA_KEY_TYPE_ML_DSA_PUBLIC_KEY` (signature verification only)

PSA_ALG_DETERMINISTIC_HASH_ML_DSA (macro)

Deterministic module lattice-based digital signature algorithm with pre-hashing (HashML-DSA).

Added in version 1.3.

```
#define PSA_ALG_DETERMINISTIC_HASH_ML_DSA(hash_alg) \
    /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true. This includes `PSA_ALG_ANY_HASH` when specifying the algorithm in a key policy.

Returns

The corresponding deterministic HashML-DSA signature algorithm, using `hash_alg` to pre-hash the message.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This algorithm can be used with both the message and hash signature functions.

This is the pre-hashed ML-DSA digital signature algorithm, defined by *FIPS Publication 204: Module-Lattice-Based Digital Signature Standard* [FIPS204], without hedging. ML-DSA requires an ML-DSA key, which determines the ML-DSA parameter set for the operation.

> **Note:**
>
> For the pre-hashing, [FIPS204] §5.4 recommends the use of an approved hash function with an equivalent, or better, security strength than the chosen ML-DSA parameter set.

This algorithm is deterministic: each invocation with the same inputs returns an identical signature.

> ⚠️ **Warning**
>
> It is recommended to use the hedged `PSA_ALG_HASH_ML_DSA()` algorithm instead, when supported by the implementation. See the notes on deterministic signatures.

When `PSA_ALG_DETERMINISTIC_HASH_ML_DSA()` is used as a permitted algorithm in a key policy, this permits:

- `PSA_ALG_DETERMINISTIC_HASH_ML_DSA()` as the algorithm in a call to `psa_sign_message()` and `psa_sign_hash()`.
- `PSA_ALG_HASH_ML_DSA()` or `PSA_ALG_DETERMINISTIC_HASH_ML_DSA()` as the algorithm in a call to `psa_verify_message()` and `psa_verify_hash()`.

> **Note:**
>
> The signature produced by HashML-DSA is distinct from that produced by ML-DSA.

Usage

See `PSA_ALG_HASH_ML_DSA()` for example usage.

Compatible key types

`PSA_KEY_TYPE_ML_DSA_KEY_PAIR`

`PSA_KEY_TYPE_ML_DSA_PUBLIC_KEY` (signature verification only)

PSA_ALG_IS_ML_DSA (macro)

Whether the specified algorithm is ML-DSA, without pre-hashing.

Added in version 1.3.

```
#define PSA_ALG_IS_ML_DSA(alg) /* specification-defined value */
```

Parameters

    `alg`   An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a pure ML-DSA algorithm, `0` otherwise.

This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

> **Note:**
>
> Use `PSA_ALG_IS_HASH_ML_DSA()` to determine if an algorithm identifier is a HashML-DSA algorithm.

PSA_ALG_IS_HASH_ML_DSA (macro)

Whether the specified algorithm is HashML-DSA.

Added in version 1.3.

```
#define PSA_ALG_IS_HASH_ML_DSA(alg) /* specification-defined value */
```

Parameters

    `alg`   An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a HashML-DSA algorithm, `0` otherwise.

This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

> **Note:**
>
> Use `PSA_ALG_IS_ML_DSA()` to determine if an algorithm identifier is a pre-hashed ML-DSA algorithm.

PSA_ALG_IS_DETERMINISTIC_HASH_ML_DSA (macro)

Whether the specified algorithm is deterministic HashML-DSA.

Added in version 1.3.

```
#define PSA_ALG_IS_DETERMINISTIC_HASH_ML_DSA(alg) \
    /* specification-defined value */
```

Parameters

    `alg`   An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a deterministic HashML-DSA algorithm, `0` otherwise.

This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

See also `PSA_ALG_IS_HASH_ML_DSA()` and `PSA_ALG_IS_HEDGED_HASH_ML_DSA()`.

`PSA_ALG_IS_HEDGED_HASH_ML_DSA` (macro)

Whether the specified algorithm is hedged HashML-DSA.

Added in version 1.3.

```
#define PSA_ALG_IS_HEDGED_HASH_ML_DSA(alg) /* specification-defined value */
```

Parameters

    `alg`   An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a hedged HashML-DSA algorithm, `0` otherwise.

This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

See also `PSA_ALG_IS_HASH_ML_DSA()` and `PSA_ALG_IS_DETERMINISTIC_HASH_ML_DSA()`.

2.4 Stateless Hash-based signatures

2.4.1 Stateless Hash-based signature keys

The Crypto API supports Stateless Hash-based digital signatures (SLH-DSA), as defined in *FIPS Publication 205: Stateless Hash-Based Digital Signature Standard* [FIPS205].

`psa_slh_dsa_family_t` (typedef)

The type of identifiers of a Stateless hash-based DSA parameter set.

Added in version 1.3.

```
typedef uint8_t psa_slh_dsa_family_t;
```

The parameter-set identifier is required to create an SLH-DSA key using the `PSA_KEY_TYPE_SLH_DSA_KEY_PAIR()` or `PSA_KEY_TYPE_SLH_DSA_PUBLIC_KEY()` macros.

The specific SLH-DSA parameter set within a family is identified by the `key_bits` attribute of the key.

The range of SLH-DSA family identifier values is divided as follows:

    `0x00`   Reserved. Not allocated to an SLH-DSA parameter-set family.

```
0x01 - 0x7f
```
> SLH-DSA parameter-set family identifiers defined by this standard. Unallocated values in this range are reserved for future use.

```
0x80 - 0xff
```
> Invalid. Values in this range must not be used.

The least significant bit of an SLH-DSA family identifier is a parity bit for the whole key type. See *SLH-DSA key encoding* on page 49 for details of the encoding of asymmetric key types.

PSA_KEY_TYPE_SLH_DSA_KEY_PAIR (macro)

SLH-DSA key pair: both the private key and public key.

Added in version 1.3.

```
#define PSA_KEY_TYPE_SLH_DSA_KEY_PAIR(set) /* specification-defined value */
```

Parameters

set	A value of type `psa_slh_dsa_family_t` that identifies the SLH-DSA parameter-set family to be used.

Description

The key attribute size of of an SLH-DSA key pair is the bit-size of each component in the SLH-DSA keys defined in [FIPS205]. That is, for a parameter set with security parameter n, the bit-size in the key attributes is $8n$. See the documentation of each SLH-DSA parameter-set family for details.

Compatible algorithms

- PSA_ALG_SLH_DSA
- PSA_ALG_HASH_SLH_DSA
- PSA_ALG_DETERMINISTIC_SLH_DSA
- PSA_ALG_DETERMINISTIC_HASH_SLH_DSA

Key format

> ⚠️ **Warning**
>
> The key format may change in a final version of this API. The standardization of exchange formats for SHL-DSA public and private keys is in progress, but final documents have not been published. See *Internet X.509 Public Key Infrastructure: Algorithm Identifiers for SLH-DSA* [LAMPS-SLHDSA].
>
> The current proposed format is based on the expected outcome of that process.

A SLH-DSA key pair is defined in [FIPS205] §9.1 as the four n-byte values, SK.seed, SK.prf, PK.seed, and PK.root, where n is the security parameter.

The data format for import and export of the key pair is the concatenation of the four octet strings:

$$SK.\text{seed} \mathbin{||} SK.\text{prf} \mathbin{||} PK.\text{seed} \mathbin{||} PK.\text{root}$$

See PSA_KEY_TYPE_SLH_DSA_PUBLIC_KEY for the data format used when exporting the public key with `psa_export_public_key()`.

Key derivation

A call to `psa_key_derivation_output_key()` will draw output bytes as follows:

- 32 bytes are drawn as SK.seed.
- 32 bytes are drawn as SK.prf.
- 32 bytes are drawn as PK.seed.

The private key (SK.seed, SK.prf, PK.seed, PK.root) is generated from these values as defined by `slh_keygen_internal()` in [FIPS205] §9.1.

`PSA_KEY_TYPE_SLH_DSA_PUBLIC_KEY` (macro)

SLH-DSA public key.

Added in version 1.3.

```
#define PSA_KEY_TYPE_SLH_DSA_PUBLIC_KEY(set) /* specification-defined value */
```

Parameters

`set`	A value of type `psa_slh_dsa_family_t` that identifies the SLH-DSA parameter-set family to be used.

Description

The key attribute size of an SLH-DSA public key is the same as the corresponding private key. See `PSA_KEY_TYPE_SLH_DSA_KEY_PAIR()` and the documentation of each SLH-DSA parameter-set family for details.

Compatible algorithms

- `PSA_ALG_SLH_DSA`
- `PSA_ALG_HASH_SLH_DSA`
- `PSA_ALG_DETERMINISTIC_SLH_DSA`
- `PSA_ALG_DETERMINISTIC_HASH_SLH_DSA`

Key format

> ⚠️ **Warning**
>
> The key format may change in a final version of this API. The standardization of exchange formats for SHL-DSA public and private keys is in progress, but final documents have not been published. See *Internet X.509 Public Key Infrastructure: Algorithm Identifiers for SLH-DSA* [LAMPS-SLHDSA].
>
> The current proposed format is based on the expected outcome of that process.

A SLH-DSA public key is defined in [FIPS205] §9.1 as two n-byte values, PK.seed and PK.root, where n is the security parameter.

The data format for export of the public key is the concatenation of the two octet strings:

$$PK.\text{seed} \parallel PK.\text{root}$$

PSA_SLH_DSA_FAMILY_SHA2_S (macro)

SLH-DSA family for the SLH-DSA-SHA2-*NNN*s parameter sets.

Added in version 1.3.

```
#define PSA_SLH_DSA_FAMILY_SHA2_S ((psa_slh_dsa_family_t) 0x02)
```

This family comprises the following parameter sets:

- SLH-DSA-SHA2-128s : `key_bits = 128`
- SLH-DSA-SHA2-192s : `key_bits = 192`
- SLH-DSA-SHA2-256s : `key_bits = 256`

They are defined in [FIPS205].

PSA_SLH_DSA_FAMILY_SHA2_F (macro)

SLH-DSA family for the SLH-DSA-SHA2-*NNN*f parameter sets.

Added in version 1.3.

```
#define PSA_SLH_DSA_FAMILY_SHA2_F ((psa_slh_dsa_family_t) 0x04)
```

This family comprises the following parameter sets:

- SLH-DSA-SHA2-128f : `key_bits = 128`
- SLH-DSA-SHA2-192f : `key_bits = 192`
- SLH-DSA-SHA2-256f : `key_bits = 256`

They are defined in [FIPS205].

PSA_SLH_DSA_FAMILY_SHAKE_S (macro)

SLH-DSA family for the SLH-DSA-SHAKE-*NNN*s parameter sets.

Added in version 1.3.

```
#define PSA_SLH_DSA_FAMILY_SHAKE_S ((psa_slh_dsa_family_t) 0x0b)
```

This family comprises the following parameter sets:

- SLH-DSA-SHAKE-128s : `key_bits = 128`
- SLH-DSA-SHAKE-192s : `key_bits = 192`
- SLH-DSA-SHAKE-256s : `key_bits = 256`

They are defined in [FIPS205].

PSA_SLH_DSA_FAMILY_SHAKE_F (macro)

SLH-DSA family for the SLH-DSA-SHAKE-*NNN*f parameter sets.

Added in version 1.3.

```
#define PSA_SLH_DSA_FAMILY_SHAKE_F ((psa_slh_dsa_family_t) 0x0d)
```

This family comprises the following parameter sets:

- SLH-DSA-SHAKE-128f : `key_bits = 128`
- SLH-DSA-SHAKE-192f : `key_bits = 192`
- SLH-DSA-SHAKE-256f : `key_bits = 256`

They are defined in [FIPS205].

PSA_KEY_TYPE_IS_SLH_DSA (macro)

Whether a key type is an SLH-DSA key, either a key pair or a public key.

Added in version 1.3.

```
#define PSA_KEY_TYPE_IS_SLH_DSA(type) /* specification-defined value */
```

Parameters

type   A key type: a value of type `psa_key_type_t`.

PSA_KEY_TYPE_IS_SLH_DSA_KEY_PAIR (macro)

Whether a key type is an SLH-DSA key pair.

Added in version 1.3.

```
#define PSA_KEY_TYPE_IS_SLH_DSA_KEY_PAIR(type) \
    /* specification-defined value */
```

Parameters

type   A key type: a value of type `psa_key_type_t`.

PSA_KEY_TYPE_IS_SLH_DSA_PUBLIC_KEY (macro)

Whether a key type is an SLH-DSA public key.

Added in version 1.3.

```
#define PSA_KEY_TYPE_IS_SLH_DSA_PUBLIC_KEY(type) \
    /* specification-defined value */
```

Parameters

type   A key type: a value of type `psa_key_type_t`.

PSA_KEY_TYPE_SLH_DSA_GET_FAMILY (macro)

Extract the parameter-set family from an SLH-DSA key type.

Added in version 1.3.

```
#define PSA_KEY_TYPE_SLH_DSA_GET_FAMILY(type) /* specification-defined value */
```

Parameters

> `type`   An SLH-DSA key type: a value of type `psa_key_type_t` such that
>   `PSA_KEY_TYPE_IS_SLH_DSA`(type) is true.

Returns: `psa_dh_family_t`

The SLH-DSA parameter-set family id, if `type` is a supported SLH-DSA key. Unspecified if `type` is not a
supported SLH-DSA key.

2.4.2 Stateless Hash-based signature algorithms

These algorithms extend those defined in *PSA Certified Crypto API* [PSA-CRYPT] §10.7 *Asymmetric signature*,
for use with the signature functions.

The SLH-DSA signature and verification scheme is defined in *FIPS Publication 205: Stateless Hash-Based
Digital Signature Standard* [FIPS205]. SLH-DSA has twelve parameter sets which provide differing security
strengths, trade-off between signature size and computation cost, and selection between SHA2 and
SHAKE-based hashing.

SLH-DSA keys are fairly compact, 32, 48, or 64 bytes for the public key, and double that for the key pair.
SLH-DSA signatures are much larger than those for RSA and Elliptic curve schemes, between 7.8kB and
49kB depending on the selected parameter set. An SLH-DSA signature has the structure described in
[FIPS205] §9.2, Figure 17.

See [FIPS205] §11 for details on the parameter sets, and the public key and generated signature sizes.

The generation of an SLH-DSA key depends on the full parameter specification. The encoding of each
parameter set into the key attributes is described in *Stateless Hash-based signature keys* on page 27.

[FIPS205] defines pure and pre-hashed variants of the signature scheme, which can either be hedged
(randomized) or deterministic. Four algorithms are defined to support these variants: `PSA_ALG_SLH_DSA`,
`PSA_ALG_DETERMINISTIC_SLH_DSA`, `PSA_ALG_HASH_SLH_DSA()`, and `PSA_ALG_DETERMINISTIC_HASH_SLH_DSA()`.

Hedged and deterministic signatures

Hedging incorporates fresh randomness in the signature computation, resulting in distinct signatures on
every signing operation when given identical inputs. Deterministic signatures do not require additional
random data, and result in an identical signature for the same inputs.

Signature verification does not distinguish between a hedged and a deterministic signature. Either hedged
or deterministic algorithms can be used when verifying a signature.

When computing a signature, the key's permitted-algorithm policy must match the requested algorithm,
treating hedged and deterministic versions as distinct. When verifying a signature, the hedged and
deterministic versions of each algorithm are considered equivalent when checking the key's
permitted-algorithm policy.

> **Note:**
>
> The hedged version provides message secrecy and some protection against side-channels. [FIPS205]
> recommends that users should use the hedged version if either of these issues are a concern. The

deterministic variant should only be used if the implementation does not include any source of randomness.

Implementation note

[FIPS205] recommends that implementations use an approved random number generator to provide the random value in the hedged version. However, it notes that use of the hedged variant with a weak RNG is generally preferable to the deterministic variant.

> **Rationale**
>
> The use of fresh randomness, or not, when computing a signature seems like an implementation decision based on the capability of the system, and its vulnerability to specific threats, following the recommendations in [FIPS205].
>
> However, the Crypto API gives distinct algorithm identifiers for the hedged and deterministic variants for the following reasons:
>
> - [FIPS205] §9.1 recommends that SLH-DSA signing keys are only used to compute either deterministic, or hedged, signatures, but not both. Supporting this recommendation requires separate algorithm identifiers, and requiring an exact policy match for signature computation.
> - Enable an application use case to require a specific variant.

Pure and pre-hashed algorithms

The pre-hashed signature computation *HashSLH-DSA* generates distinct signatures to a pure signature *SLH-DSA*, with the same key and message hashing algorithm.

An SLH-DSA signature can only be verified with an SLH-DSA algorithm. A HashSLH-DSA signature can only be verified with a HashSLH-DSA algorithm.

Contexts

Contexts are not supported in the current version of this specification because there is no suitable signature interface that can take the context as a parameter. A empty context string is used when computing or verifying SLH-DSA signatures.

A future version of this specification may add suitable functions and extend this algorithm to support contexts.

`PSA_ALG_SLH_DSA` (macro)

Stateless hash-based digital signature algorithm without pre-hashing (SLH-DSA).

Added in version 1.3.

```
#define PSA_ALG_SLH_DSA ((psa_algorithm_t) 0x06004000)
```

This algorithm can only be used with the `psa_sign_message()` and `psa_verify_message()` functions.

This is the pure SLH-DSA digital signature algorithm, defined by *FIPS Publication 205: Stateless Hash-Based Digital Signature Standard* [FIPS205], using hedging. SLH-DSA requires an SLH-DSA key, which determines the SLH-DSA parameter set for the operation.

This algorithm is randomized: each invocation returns a different, equally valid signature. See the notes on hedged signatures.

When `PSA_ALG_SLH_DSA` is used as a permitted algorithm in a key policy, this permits:

- `PSA_ALG_SLH_DSA` as the algorithm in a call to `psa_sign_message()`.
- `PSA_ALG_SLH_DSA` or `PSA_ALG_DETERMINISTIC_SLH_DSA` as the algorithm in a call to `psa_verify_message()`.

Note:

To sign or verify the pre-computed hash of a message using SLH-DSA, the HashSLH-DSA algorithms (`PSA_ALG_HASH_SLH_DSA()` and `PSA_ALG_DETERMINISTIC_HASH_SLH_DSA()`) can also be used with `psa_sign_hash()` and `psa_verify_hash()`.

The signature produced by HashSLH-DSA is distinct from that produced by SLH-DSA.

Compatible key types

`PSA_KEY_TYPE_SLH_DSA_KEY_PAIR()`

`PSA_KEY_TYPE_SLH_DSA_PUBLIC_KEY()` (signature verification only)

`PSA_ALG_DETERMINISTIC_SLH_DSA` (macro)

Deterministic stateless hash-based digital signature algorithm without pre-hashing (SLH-DSA).

Added in version 1.3.

```
#define PSA_ALG_DETERMINISTIC_SLH_DSA ((psa_algorithm_t) 0x06004100)
```

This algorithm can only be used with the `psa_sign_message()` and `psa_verify_message()` functions.

This is the pure SLH-DSA digital signature algorithm, defined by [FIPS205], without hedging. SLH-DSA requires an SLH-DSA key, which determines the SLH-DSA parameter set for the operation.

This algorithm is deterministic: each invocation with the same inputs returns an identical signature.

⚠️ **Warning**

It is recommended to use the hedged `PSA_ALG_SLH_DSA` algorithm instead, when supported by the implementation. See the notes on deterministic signatures.

When `PSA_ALG_DETERMINISTIC_SLH_DSA` is used as a permitted algorithm in a key policy, this permits:

- `PSA_ALG_DETERMINISTIC_SLH_DSA` as the algorithm in a call to `psa_sign_message()`.
- `PSA_ALG_SLH_DSA` or `PSA_ALG_DETERMINISTIC_SLH_DSA` as the algorithm in a call to `psa_verify_message()`.

Note:

To sign or verify the pre-computed hash of a message using SLH-DSA, the HashSLH-DSA algorithms (`PSA_ALG_HASH_SLH_DSA()` and `PSA_ALG_DETERMINISTIC_HASH_SLH_DSA()`) can also be used with `psa_sign_hash()` and `psa_verify_hash()`.

The signature produced by HashSLH-DSA is distinct from that produced by SLH-DSA.

Compatible key types

`PSA_KEY_TYPE_SLH_DSA_KEY_PAIR()`

`PSA_KEY_TYPE_SLH_DSA_PUBLIC_KEY()` (signature verification only)

`PSA_ALG_HASH_SLH_DSA` (macro)

Stateless hash-based digital signature algorithm with pre-hashing (HashSLH-DSA).

Added in version 1.3.

```
#define PSA_ALG_HASH_SLH_DSA(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true. This includes `PSA_ALG_ANY_HASH` when specifying the algorithm in a key policy.

Returns

The corresponding HashSLH-DSA signature algorithm, using `hash_alg` to pre-hash the message.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This algorithm can be used with both the message and hash signature functions.

This is the pre-hashed SLH-DSA digital signature algorithm, defined by [FIPS205], using hedging. SLH-DSA requires an SLH-DSA key, which determines the SLH-DSA parameter set for the operation.

> **Note:**
>
> For the pre-hashing, [FIPS205] §10.2 recommends the use of an approved hash function with an equivalent, or better, security strength than the chosen SLH-DSA parameter set.

This algorithm is randomized: each invocation returns a different, equally valid signature. See the notes on hedged signatures.

When `PSA_ALG_HASH_SLH_DSA()` is used as a permitted algorithm in a key policy, this permits:

- `PSA_ALG_HASH_SLH_DSA()` as the algorithm in a call to `psa_sign_message()` and `psa_sign_hash()`.
- `PSA_ALG_HASH_SLH_DSA()` or `PSA_ALG_DETERMINISTIC_HASH_SLH_DSA()` as the algorithm in a call to `psa_verify_message()` and `psa_verify_hash()`.

> **Note:**
>
> The signature produced by HashSLH-DSA is distinct from that produced by SLH-DSA.

Usage

This is a hash-and-sign algorithm. To calculate a signature, use one of the following approaches:

- Call `psa_sign_message()` with the message.
- Calculate the hash of the message with `psa_hash_compute()`, or with a multi-part hash operation, using the `hash_alg` hash algorithm. Note that `hash_alg` can be extracted from the signature algorithm using `PSA_ALG_GET_HASH(sig_alg)`. Then sign the calculated hash with `psa_sign_hash()`.

Verifying a signature is similar, using `psa_verify_message()` or `psa_verify_hash()` instead of the signature function.

Compatible key types

`PSA_KEY_TYPE_SLH_DSA_KEY_PAIR()`

`PSA_KEY_TYPE_SLH_DSA_PUBLIC_KEY()` (signature verification only)

PSA_ALG_DETERMINISTIC_HASH_SLH_DSA (macro)

Deterministic stateless hash-based digital signature algorithm with pre-hashing (HashSLH-DSA).

Added in version 1.3.

```
#define PSA_ALG_DETERMINISTIC_HASH_SLH_DSA(hash_alg) \
    /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true. This includes `PSA_ALG_ANY_HASH` when specifying the algorithm in a key policy.

Returns

The corresponding deterministic HashSLH-DSA signature algorithm, using `hash_alg` to pre-hash the message.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This algorithm can be used with both the message and hash signature functions.

This is the pre-hashed SLH-DSA digital signature algorithm, defined by [FIPS205], without hedging. SLH-DSA requires an SLH-DSA key, which determines the SLH-DSA parameter set for the operation.

> **Note:**
>
> For the pre-hashing, [FIPS205] §10.2 recommends the use of an approved hash function with an equivalent, or better, security strength than the chosen SLH-DSA parameter set.

This algorithm is deterministic: each invocation with the same inputs returns an identical signature.

> **⚠ Warning**
>
> It is recommended to use the hedged `PSA_ALG_HASH_SLH_DSA()` algorithm instead, when supported by the implementation. See the notes on deterministic signatures.

When `PSA_ALG_DETERMINISTIC_HASH_SLH_DSA()` is used as a permitted algorithm in a key policy, this permits:

- `PSA_ALG_DETERMINISTIC_HASH_SLH_DSA()` as the algorithm in a call to `psa_sign_message()` and `psa_sign_hash()`.
- `PSA_ALG_HASH_SLH_DSA()` or `PSA_ALG_DETERMINISTIC_HASH_SLH_DSA()` as the algorithm in a call to `psa_verify_message()` and `psa_verify_hash()`.

Note:

The signature produced by HashSLH-DSA is distinct from that produced by SLH-DSA.

Usage

See `PSA_ALG_HASH_SLH_DSA()` for example usage.

Compatible key types

`PSA_KEY_TYPE_SLH_DSA_KEY_PAIR()`

`PSA_KEY_TYPE_SLH_DSA_PUBLIC_KEY()` (signature verification only)

`PSA_ALG_IS_SLH_DSA` (macro)

Whether the specified algorithm is SLH-DSA.

Added in version 1.3.

```
#define PSA_ALG_IS_SLH_DSA(alg) /* specification-defined value */
```

Parameters

    alg   An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is an SLH-DSA algorithm, `0` otherwise.

This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

`PSA_ALG_IS_HASH_SLH_DSA` (macro)

Whether the specified algorithm is HashSLH-DSA.

Added in version 1.3.

```
#define PSA_ALG_IS_HASH_SLH_DSA(alg) /* specification-defined value */
```

Parameters

 `alg`   An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a HashSLH-DSA algorithm, `0` otherwise.

This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

PSA_ALG_IS_DETERMINISTIC_HASH_SLH_DSA (macro)

Whether the specified algorithm is deterministic HashSLH-DSA.

Added in version 1.3.

```
#define PSA_ALG_IS_DETERMINISTIC_HASH_SLH_DSA(alg) \
    /* specification-defined value */
```

Parameters

 `alg`   An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a deterministic HashSLH-DSA algorithm, `0` otherwise.

This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

See also `PSA_ALG_IS_HASH_SLH_DSA()` and `PSA_ALG_IS_HEDGED_HASH_SLH_DSA()`.

PSA_ALG_IS_HEDGED_HASH_SLH_DSA (macro)

Whether the specified algorithm is hedged HashSLH-DSA.

Added in version 1.3.

```
#define PSA_ALG_IS_HEDGED_HASH_SLH_DSA(alg) /* specification-defined value */
```

Parameters

 `alg`   An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a hedged HashSLH-DSA algorithm, `0` otherwise.

This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

See also `PSA_ALG_IS_HASH_SLH_DSA()` and `PSA_ALG_IS_DETERMINISTIC_HASH_SLH_DSA()`.

2.5 Leighton-Micali Signatures

The Crypto API supports Leighton-Micali Signatures (LMS), and the multi-level Hierarchical Signature Scheme (HSS). These schemes are defined in *Leighton-Micali Hash-Based Signatures* [RFC8554].

For the Crypto API to support signature verification, it is only necessary to define a public keys for these schemes, and the default public key formats for import and export.

> **Rationale**
>
> At present, it is not expected that the Crypto API will be used to generate LMS or HSS private keys, or to carry out signing operations. However, there is value in supporting verification of LMS and HSS signatures. Therefore, the Crypto API does not support LMS or HSS key pairs, or the associated signing operations.

Note:

A full set of NIST-approved parameter sets for LMS and HSS is defined in *NIST Special Publication 800-208: Recommendation for Stateful Hash-Based Signature Schemes* [SP800-208] §4, with the additional IANA identifiers defined in *Additional Parameter sets for HSS/LMS Hash-Based Signatures* [CFRG-LMS].

2.5.1 Leighton-Micali Signature keys

PSA_KEY_TYPE_LMS_PUBLIC_KEY (macro)

Leighton-Micali Signatures (LMS) public key.

Added in version 1.3.

```
#define PSA_KEY_TYPE_LMS_PUBLIC_KEY ((psa_key_type_t)0x4007)
```

The parameterization of an LMS key is fully encoded in the key data.

The key attribute size of an LMS public key is output length, in bits, of the hash function identified by the LMS parameter set.

- SHA-256/192, SHAKE256/192 : `key_bits = 192`
- SHA-256, SHAKE256/256 : `key_bits = 256`

Compatible algorithms

- PSA_ALG_LMS

Key format

The data format for import or export of the public key is the encoded `lms_public_key` structure, defined in [RFC8554] §3.

`PSA_KEY_TYPE_HSS_PUBLIC_KEY` (macro)

Hierarchical Signature Scheme (HSS) public key.

Added in version 1.3.

```
#define PSA_KEY_TYPE_HSS_PUBLIC_KEY ((psa_key_type_t)0x4008)
```

The parameterization of an HSS key is fully encoded in the key data.

The key attribute size of an HSS public key is output length, in bits, of the hash function identified by the HSS parameter set.

- SHA-256/192, SHAKE256/192 : `key_bits = 192`
- SHA-256, SHAKE256/256 : `key_bits = 256`

Compatible algorithms

- `PSA_ALG_HSS`

Key format

> ⚠️ **Warning**
>
> The key format may change in a final version of this API. The standardization of exchange formats for HSS public keys is in progress, but final documents have not been published. See *Use of the HSS and XMSS Hash-Based Signature Algorithms in Internet X.509 Public Key Infrastructure* [LAMPS-SHBS].
>
> The current proposed format is based on the expected outcome of that process.

The data format for import or export of the public key is the encoded `hss_public_key` structure, defined in [RFC8554] §3.

2.5.2 Leighton-Micali Signature algorithms

These algorithms extend those defined in *PSA Certified Crypto API* [PSA-CRYPT] §10.7 *Asymmetric signature*, for use with the signature functions.

`PSA_ALG_LMS` (macro)

Leighton-Micali Signatures (LMS) signature algorithm.

Added in version 1.3.

```
#define PSA_ALG_LMS ((psa_algorithm_t) 0x06004800)
```

This message-signature algorithm can only be used with the `psa_verify_message()` function.

This is the LMS stateful hash-based signature algorithm, defined by *Leighton-Micali Hash-Based Signatures* [RFC8554]. LMS requires an LMS key. The key and the signature must both encode the same LMS parameter set, which is used for the verification procedure.

Note:

LMS signature calculation is not supported.

Compatible key types

`PSA_KEY_TYPE_LMS_PUBLIC_KEY` (signature verification only)

`PSA_ALG_HSS` (macro)

Hierarchical Signature Scheme (HSS) signature algorithm.

Added in version 1.3.

```
#define PSA_ALG_HSS ((psa_algorithm_t) 0x06004900)
```

This message-signature algorithm can only be used with the `psa_verify_message()` function.

This is the HSS stateful hash-based signature algorithm, defined by *Leighton-Micali Hash-Based Signatures* [RFC8554]. HSS requires an HSS key. The key and the signature must both encode the same HSS parameter set, which is used for the verification procedure.

> **Note:**
>
> HSS signature calculation is not supported.

Compatible key types

`PSA_KEY_TYPE_HSS_PUBLIC_KEY` (signature verification only)

2.6 eXtended Merkle Signature Scheme

The Crypto API supports eXtended Merkle Signature Scheme (XMSS), and the multi-tree variant XMSS^{MT}. These schemes are defined in *XMSS: eXtended Merkle Signature Scheme* [RFC8391].

For the Crypto API to support signature verification, it is only necessary to define public keys for these schemes, and the default public key formats for import and export.

> **Rationale**
>
> At present, it is not expected that the Crypto API will be used to generate XMSS or XMSS^{MT} private keys, or to carry out signing operations. However, there is value in supporting verification of XMSS and XMSS^{MT} signatures. Therefore, the Crypto API does not support XMSS or XMSS^{MT} key pairs, or the associated signing operations.

> **Note:**
>
> A full set of NIST-approved parameter sets for XMSS or XMSS^{MT} is defined in *NIST Special Publication 800-208: Recommendation for Stateful Hash-Based Signature Schemes* [SP800-208] §5.

2.6.1 XMSS and XMSS^{MT} keys

`PSA_KEY_TYPE_XMSS_PUBLIC_KEY` (macro)

eXtended Merkle Signature Scheme (XMSS) public key.

Added in version 1.3.

```
#define PSA_KEY_TYPE_XMSS_PUBLIC_KEY ((psa_key_type_t)0x400B)
```

The parameterization of an XMSS key is fully encoded in the key data.

The key attribute size of an XMSS public key is output length, in bits, of the hash function identified by the XMSS parameter set.

- SHA-256/192, SHAKE256/192 : `key_bits = 192`
- SHA-256, SHAKE256/256 : `key_bits = 256`

> **Note:**
>
> For a multi-tree XMSS key, see `PSA_KEY_TYPE_XMSS_MT_PUBLIC_KEY`.

Compatible algorithms

- `PSA_ALG_XMSS`

Key format

> ⚠️ **Warning**
>
> The key format may change in a final version of this API. The standardization of exchange formats for XMSS public keys is in progress, but final documents have not been published. See *Use of the HSS and XMSS Hash-Based Signature Algorithms in Internet X.509 Public Key Infrastructure* [LAMPS-SHBS].
>
> The current proposed format is based on the expected outcome of that process.

The data format for import or export of the public key is the encoded `xmss_public_key` structure, defined in [RFC8391] Appendix B.3.

`PSA_KEY_TYPE_XMSS_MT_PUBLIC_KEY` (macro)

Multi-tree eXtended Merkle Signature Scheme (XMSS^{MT}) public key.

Added in version 1.3.

```
#define PSA_KEY_TYPE_XMSS_MT_PUBLIC_KEY ((psa_key_type_t)0x400D)
```

The parameterization of an XMSS^{MT} key is fully encoded in the key data.

The key attribute size of an XMSS^{MT} public key is output length, in bits, of the hash function identified by the XMSS^{MT} parameter set.

- SHA-256/192, SHAKE256/192 : `key_bits = 192`
- SHA-256, SHAKE256/256 : `key_bits = 256`

Compatible algorithms

- `PSA_ALG_XMSS_MT`

Key format

> ⚠️ **Warning**
>
> The key format may change in a final version of this API. The standardization of exchange formats for XMSS^MT public keys is in progress, but final documents have not been published. See *Use of the HSS and XMSS Hash-Based Signature Algorithms in Internet X.509 Public Key Infrastructure* [LAMPS-SHBS].
>
> The current proposed format is based on the expected outcome of that process.

The data format for import or export of the public key is the encoded `xmssmt_public_key` structure, defined in [RFC8391] Appendix C.3.

2.6.2 XMSS and XMSS^MT algorithms

These algorithms extend those defined in *PSA Certified Crypto API* [PSA-CRYPT] §10.7 *Asymmetric signature*, for use with the signature functions.

`PSA_ALG_XMSS` (macro)

eXtended Merkle Signature Scheme (XMSS) signature algorithm.

Added in version 1.3.

```
#define PSA_ALG_XMSS ((psa_algorithm_t) 0x06004A00)
```

This message-signature algorithm can only be used with the `psa_verify_message()` function.

This is the XMSS stateful hash-based signature algorithm, defined by *XMSS: eXtended Merkle Signature Scheme* [RFC8391]. XMSS requires an XMSS key. The key and the signature must both encode the same XMSS parameter set, which is used for the verification procedure.

> **Note:**
>
> XMSS signature calculation is not supported.

Compatible key types

`PSA_KEY_TYPE_XMSS_PUBLIC_KEY` (signature verification only)

`PSA_ALG_XMSS_MT` (macro)

Multi-tree eXtended Merkle Signature Scheme (XMSS^MT) signature algorithm.

Added in version 1.3.

```
#define PSA_ALG_XMSS_MT ((psa_algorithm_t) 0x06004B00)
```

This message-signature algorithm can only be used with the `psa_verify_message()` function.

This is the XMSS^{MT} stateful hash-based signature algorithm, defined by *XMSS: eXtended Merkle Signature Scheme* [RFC8391]. XMSS^{MT} requires an XMSS^{MT} key. The key and the signature must both encode the same XMSS^{MT} parameter set, which is used for the verification procedure.

Note:

XMSS^{MT} signature calculation is not supported.

Compatible key types

`PSA_KEY_TYPE_XMSS_MT_PUBLIC_KEY` (signature verification only)

See *Algorithm and key type encoding* on page 47 for the encoding of the key types and algorithm identifiers added by this extension.

Appendix A: Example header file

The API elements in this specification, once finalized, will be defined in `psa/crypto.h`.

This is an example of the header file definition of the PAKE API elements. This can be used as a starting point or reference for an implementation.

Note:

Not all of the API elements are fully defined. An implementation must provide the full definition.

The header will not compile without these missing definitions, and might require reordering to satisfy C compilation rules.

A.1 psa/crypto.h

```
/* This file contains reference definitions for implementation of the
 * PSA Certified Crypto API v1.3 PQC Extension beta
 *
 * These definitions must be embedded in, or included by, psa/crypto.h
 */

#define PSA_ALG_SHA_256_192 ((psa_algorithm_t)0x0200000E)
#define PSA_ALG_SHAKE128_256 ((psa_algorithm_t)0x02000016)
#define PSA_ALG_SHAKE256_192 ((psa_algorithm_t)0x02000017)
#define PSA_ALG_SHAKE256_256 ((psa_algorithm_t)0x02000018)
#define PSA_KEY_TYPE_ML_KEM_KEY_PAIR ((psa_key_type_t)0x7004)
#define PSA_KEY_TYPE_ML_KEM_PUBLIC_KEY ((psa_key_type_t)0x4004)
#define PSA_KEY_TYPE_IS_ML_KEM(type) /* specification-defined value */
#define PSA_ALG_ML_KEM ((psa_algorithm_t)0x0c000200)
#define PSA_KEY_TYPE_ML_DSA_KEY_PAIR ((psa_key_type_t)0x7002)
#define PSA_KEY_TYPE_ML_DSA_PUBLIC_KEY ((psa_key_type_t)0x4002)
#define PSA_KEY_TYPE_IS_ML_DSA(type) /* specification-defined value */
#define PSA_ALG_ML_DSA ((psa_algorithm_t) 0x06004400)
#define PSA_ALG_DETERMINISTIC_ML_DSA ((psa_algorithm_t) 0x06004500)
#define PSA_ALG_HASH_ML_DSA(hash_alg) /* specification-defined value */
#define PSA_ALG_DETERMINISTIC_HASH_ML_DSA(hash_alg) \
    /* specification-defined value */
#define PSA_ALG_IS_ML_DSA(alg) /* specification-defined value */
#define PSA_ALG_IS_HASH_ML_DSA(alg) /* specification-defined value */
#define PSA_ALG_IS_DETERMINISTIC_HASH_ML_DSA(alg) \
    /* specification-defined value */
#define PSA_ALG_IS_HEDGED_HASH_ML_DSA(alg) /* specification-defined value */
```

```
typedef uint8_t psa_slh_dsa_family_t;
#define PSA_KEY_TYPE_SLH_DSA_KEY_PAIR(set) /* specification-defined value */
#define PSA_KEY_TYPE_SLH_DSA_PUBLIC_KEY(set) /* specification-defined value */
#define PSA_SLH_DSA_FAMILY_SHA2_S ((psa_slh_dsa_family_t) 0x02)
#define PSA_SLH_DSA_FAMILY_SHA2_F ((psa_slh_dsa_family_t) 0x04)
#define PSA_SLH_DSA_FAMILY_SHAKE_S ((psa_slh_dsa_family_t) 0x0b)
#define PSA_SLH_DSA_FAMILY_SHAKE_F ((psa_slh_dsa_family_t) 0x0d)
#define PSA_KEY_TYPE_IS_SLH_DSA(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_SLH_DSA_KEY_PAIR(type) \
    /* specification-defined value */
#define PSA_KEY_TYPE_IS_SLH_DSA_PUBLIC_KEY(type) \
    /* specification-defined value */
#define PSA_KEY_TYPE_SLH_DSA_GET_FAMILY(type) /* specification-defined value */
#define PSA_ALG_SLH_DSA ((psa_algorithm_t) 0x06004000)
#define PSA_ALG_DETERMINISTIC_SLH_DSA ((psa_algorithm_t) 0x06004100)
#define PSA_ALG_HASH_SLH_DSA(hash_alg) /* specification-defined value */
#define PSA_ALG_DETERMINISTIC_HASH_SLH_DSA(hash_alg) \
    /* specification-defined value */
#define PSA_ALG_IS_SLH_DSA(alg) /* specification-defined value */
#define PSA_ALG_IS_HASH_SLH_DSA(alg) /* specification-defined value */
#define PSA_ALG_IS_DETERMINISTIC_HASH_SLH_DSA(alg) \
    /* specification-defined value */
#define PSA_ALG_IS_HEDGED_HASH_SLH_DSA(alg) /* specification-defined value */
#define PSA_KEY_TYPE_LMS_PUBLIC_KEY ((psa_key_type_t)0x4007)
#define PSA_KEY_TYPE_HSS_PUBLIC_KEY ((psa_key_type_t)0x4008)
#define PSA_ALG_LMS ((psa_algorithm_t) 0x06004800)
#define PSA_ALG_HSS ((psa_algorithm_t) 0x06004900)
#define PSA_KEY_TYPE_XMSS_PUBLIC_KEY ((psa_key_type_t)0x400B)
#define PSA_KEY_TYPE_XMSS_MT_PUBLIC_KEY ((psa_key_type_t)0x400D)
#define PSA_ALG_XMSS ((psa_algorithm_t) 0x06004A00)
#define PSA_ALG_XMSS_MT ((psa_algorithm_t) 0x06004B00)
```

Appendix B: Algorithm and key type encoding

These are encodings for PQC algorithms and keys defined in this extension. This information should be read in conjunction with [PSA-CRYPT] Appendix B.

Note:

These encodings will be integrated into a future version of [PSA-CRYPT].

B.1 Algorithm encoding

B.1.1 Hash algorithm encoding

Additional hash algorithms defined by this extension are shown in Table 4. See also *Hash algorithm encoding* in [PSA-CRYPT] Appendix B.

Table 4 Hash algorithm sub-type values

Hash algorithm	HASH-TYPE	Algorithm identifier	Algorithm value
SHA-256/192	0x0E	PSA_ALG_SHA_256_192	0x0200000E
SHAKE128/256	0x16	PSA_ALG_SHAKE128_256	0x02000016
SHAKE256/192	0x17	PSA_ALG_SHAKE256_192	0x02000017
SHAKE256/256	0x18	PSA_ALG_SHAKE256_256	0x02000018

B.1.2 Asymmetric signature algorithm encoding

Additional signature algorithms defined by this extension are shown in Table 5 on page 48. See also *Asymmetric signature algorithm encoding* in [PSA-CRYPT] Appendix B.

Table 5 Asymmetric signature algorithm sub-type values

Signature algorithm	SIGN-TYPE	Algorithm identifier	Algorithm value
Hedged SLH-DSA	`0x40`	`PSA_ALG_SLH_DSA`	`0x06004000`
Deterministic SLH-DSA	`0x41`	`PSA_ALG_DETERMINISTIC_SLH_DSA`	`0x06004100`
Hedged HashSLH-DSA	`0x42`	`PSA_ALG_HASH_SLH_DSA(hash)`	`0x060042hh` [a]
Deterministic HashSLH-DSA	`0x43`	`PSA_ALG_DETERMINISTIC_HASH_SLH_DSA(hash)`	`0x060043hh` [a]
Hedged ML-DSA	`0x44`	`PSA_ALG_ML_DSA`	`0x06004400`
Deterministic ML-DSA	`0x45`	`PSA_ALG_DETERMINISTIC_ML_DSA`	`0x06004500`
Hedged HashML-DSA	`0x46`	`PSA_ALG_HASH_ML_DSA(hash)`	`0x060046hh` [a]
Deterministic HashML-DSA	`0x47`	`PSA_ALG_DETERMINISTIC_HASH_ML_DSA(hash)`	`0x060047hh` [a]
LMS	`0x48`	`PSA_ALG_LMS`	`0x06004800`
HSS	`0x49`	`PSA_ALG_HSS`	`0x06004900`
XMSS	`0x4A`	`PSA_ALG_XMSS`	`0x06004A00`
XMSS^{MT}	`0x4B`	`PSA_ALG_XMSS_MT`	`0x06004B00`

a. `hh` is the HASH-TYPE for the hash algorithm, `hash`, used to construct the signature algorithm.

B.1.3 Key-encapsulation algorithm encoding

Additional key-encapsulation algorithms defined by this extension are shown in Table 6.

Table 6 Encapsulation algorithm sub-type values

Encapsulation algorithm	ENCAPS-TYPE	Algorithm identifier	Algorithm value
ML-KEM	`0x02`	`PSA_ALG_ML_KEM`	`0x0C000200`

B.2 Key encoding

Additional asymmetric key types defined by this extension are shown in Table 7. See also *Asymmetric key encoding* in [PSA-CRYPT] Appendix B.

Table 7 Asymmetric key sub-type values

Asymmetric key type	ASYM-TYPE	Details
SLH-DSA	3	See *SLH-DSA key encoding* on page 49

B.2.1 Non-parameterized asymmetric key encoding

Additional non-parameterized asymmetric key types defined by this extension are shown in Table 8. See also *Non-parameterized asymmetric key encoding* in [PSA-CRYPT] Appendix B.

Table 8 Non-parameterized asymmetric key family values

Key family	Public/pair	PAIR	NP-FAMILY	P	Key type	Key value
ML-DSA	Public key	0	1	0	PSA_KEY_TYPE_ML_DSA_PUBLIC_KEY	0x4002
	Key pair	3	1	0	PSA_KEY_TYPE_ML_DSA_KEY_PAIR	0x7002
ML-KEM	Public key	0	2	0	PSA_KEY_TYPE_ML_KEM_PUBLIC_KEY	0x4004
	Key pair	3	2	0	PSA_KEY_TYPE_ML_KEM_KEY_PAIR	0x7004
LMS	Public key	0	3	1	PSA_KEY_TYPE_LMS_PUBLIC_KEY	0x4007
HSS	Public key	0	4	0	PSA_KEY_TYPE_HSS_PUBLIC_KEY	0x4008
XMSS	Public key	0	5	1	PSA_KEY_TYPE_XMSS_PUBLIC_KEY	0x400B
XMSS^{MT}	Public key	0	6	1	PSA_KEY_TYPE_XMSS_MT_PUBLIC_KEY	0x400D

B.2.2 SLH-DSA key encoding

The key type for SLH-DSA keys defined in this specification are encoded as shown in Figure 1.

15	14	13 12	11 7	6 1	0
0	1	PAIR	3	FAMILY	P
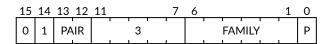
Figure 1 SLH-DSA key encoding

PAIR is either 0 for a public key, or 3 for a key pair.

The defined values for FAMILY and P are shown in Table 9.

Table 9 SLH-DSA key family values

SLH-DSA key family	FAMILY	P	SLH-DSA family [a]	Public-key value	Key-pair value
SLH-DSA-SHA2-*N*s	0x01	0	PSA_SLH_DSA_FAMILY_SHA2_S	0x4182	0x7182
SLH-DSA-SHA2-*N*f	0x02	0	PSA_SLH_DSA_FAMILY_SHA2_F	0x4184	0x7184
SLH-DSA-SHAKE-*N*s	0x05	1	PSA_SLH_DSA_FAMILY_SHAKE_S	0x418B	0x718B
SLH-DSA-SHAKE-*N*f	0x06	1	PSA_SLH_DSA_FAMILY_SHAKE_F	0x418D	0x718D

a. The SLH-DSA family values defined in the API also include the parity bit. The key type value is constructed from the SLH-DSA family using either PSA_KEY_TYPE_SLH_DSA_PUBLIC_KEY(family) or PSA_KEY_TYPE_SLH_DSA_KEY_PAIR(family) as required.

Appendix C: Example macro implementations

This section provides example implementations of the function-like macros that have specification-defined values.

Note:

In a future version of this specification, these example implementations will be replaced with a pseudo-code representation of the macro's computation in the macro description.

The examples here provide correct results for the valid inputs defined by each API, for an implementation that supports all of the defined algorithms and key types. An implementation can provide alternative definitions of these macros:

C.1 Algorithm macros

C.1.1 Updated macros

```
#define PSA_ALG_IS_HASH_AND_SIGN(alg) \
    (PSA_ALG_IS_RSA_PSS(alg) || PSA_ALG_IS_RSA_PKCS1V15_SIGN(alg) || \
     PSA_ALG_IS_ECDSA(alg) || PSA_ALG_IS_HASH_EDDSA(alg) || \
     PSA_ALG_IS_HASH_ML_DSA(alg) || PSA_ALG_IS_HASH_SLH_DSA(alg))

#define PSA_ALG_IS_SIGN_HASH(alg) \
    (PSA_ALG_IS_HASH_AND_SIGN(alg) ||
    (alg) == PSA_ALG_RSA_PKCS1V15_SIGN_RAW ||
    (alg) == PSA_ALG_ECDSA_ANY
    )
```

C.1.2 New macros

```
#define PSA_ALG_DETERMINISTIC_HASH_ML_DSA(hash_alg) \
    ((psa_algorithm_t) (0x06004700 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_DETERMINISTIC_HASH_SLH_DSA(hash_alg) \
    ((psa_algorithm_t) (0x06004300 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_HASH_ML_DSA(hash_alg) \
    ((psa_algorithm_t) (0x06004600 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_HASH_SLH_DSA(hash_alg) \
    ((psa_algorithm_t) (0x06004200 | ((hash_alg) & 0x000000ff)))
```

```
#define PSA_ALG_IS_DETERMINISTIC_HASH_ML_DSA(alg) \
    (((alg) & ~0x000000ff) == 0x06004700)

#define PSA_ALG_IS_DETERMINISTIC_HASH_SLH_DSA(alg) \
    (((alg) & ~0x000000ff) == 0x06004300)

#define PSA_ALG_IS_HASH_ML_DSA(alg) \
    (((alg) & ~0x000001ff) == 0x06004600)

#define PSA_ALG_IS_HASH_SLH_DSA(alg) \
    (((alg) & ~0x000001ff) == 0x06004200)

#define PSA_ALG_IS_HEDGED_HASH_ML_DSA(alg) \
    (((alg) & ~0x000000ff) == 0x06004600)

#define PSA_ALG_IS_HEDGED_HASH_SLH_DSA(alg) \
    (((alg) & ~0x000000ff) == 0x06004200)

#define PSA_ALG_IS_ML_DSA(alg) \
    (((alg) & ~0x00000100) == 0x06004400)

#define PSA_ALG_IS_SLH_DSA(alg) \
    (((alg) & ~0x00000100) == 0x06004000)
```

C.2 Key type macros

```
#define PSA_KEY_TYPE_IS_ML_DSA(type) \
    (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) == 0x4002)

#define PSA_KEY_TYPE_IS_ML_KEM(type) \
    (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) == 0x4004)

#define PSA_KEY_TYPE_IS_SLH_DSA(type) \
    ((PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) & 0xff80) == 0x4180)

#define PSA_KEY_TYPE_IS_SLH_DSA_KEY_PAIR(type) \
    (((type) & 0xff80) == 0x7180)

#define PSA_KEY_TYPE_IS_SLH_DSA_PUBLIC_KEY(type) \
    (((type) & 0xff80) == 0x4180)

#define PSA_KEY_TYPE_SLH_DSA_GET_FAMILY(type) \
    ((psa_slh_dsa_family_t) ((type) & 0x007f))

#define PSA_KEY_TYPE_SLH_DSA_KEY_PAIR(set) \
```

```
    ((psa_key_type_t) (0x7180 | ((set) & 0x007f)))

#define PSA_KEY_TYPE_SLH_DSA_PUBLIC_KEY(set) \
    ((psa_key_type_t) (0x4180 | ((set) & 0x007f)))
```

Appendix D: Changes to the API

D.1 Document change history

D.1.1 Changes between *Beta 0* and *Beta 1*

Clarifications and fixes

- Added references from each section to the relevant APIs in *PSA Certified Crypto API* [PSA-CRYPT].

D.1.2 Beta release

First release of the PQC Extension.

- Added support for FIPS 203 ML-KEM key-encapsulation algorithm and keys. See *Module Lattice-based key encapsulation* on page 14.

- Added support for FIPS 204 ML-DSA signature algorithm and keys. See *Module Lattice-based signatures* on page 18.

- Added support for FIPS 205 SLH-DSA signature algorithm and keys. See *Stateless Hash-based signatures* on page 27.

- Added support for LMS and HSS stateful hash-based signature verification and public keys. See *Leighton-Micali Signatures* on page 39.

- Added support for XMSS and XMSS^{MT} stateful hash-based signature verification and public keys. See *eXtended Merkle Signature Scheme* on page 41.

Index of API elements