PSA Certified Crypto API 1.2 PAKE Extension

Document number:	AES 0058
Release Quality:	Beta
Issue Number:	2
Confidentiality:	Non-confidential
Date of Issue:	15/11/2023

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 Password-authenticated key exchange (PAKE) algorithms.

Contents

About this document iii

Release information iii

License vi

References vii

Terms and abbreviations vii

Conventions x
Typographical conventions x
Numbers x

Current status and anticipated changes x

Feedback x

1 Introduction 12

1.1 About Platform Security Architecture 12

1.2 About the Crypto API PAKE Extension 12

1.3 Objectives for the PAKE Extension 13
1.3.1 Scheme review 13
1.3.2 Scope of the PAKE Extension 14

2 Password-authenticated key exchange (PAKE) 16

2.1 Algorithm encoding 16
2.1.1 PAKE algorithm encoding 16

2.2 Key encoding 17
2.2.1 SPAKE2+ key encoding 17

2.3 Key formats 18

2.4 Changes and additions to the Programming API 19
2.4.1 SPAKE2+ keys 19
2.4.2 PAKE algorithms 23
2.4.3 PAKE primitives 33
2.4.4 PAKE cipher suites 35
2.4.5 PAKE roles 41
2.4.6 PAKE step types 42
2.4.7 Multi-part PAKE operations 44
2.4.8 Support macros 56

A Example header file 61

A.1 psa/crypto.h 61

B Example macro implementations 64

C Changes to the API 66

C.1 Document change history 66
C.1.1 Changes between *Beta 1* and *Beta 2* 66
C.1.2 Changes between *Beta 0* and *Beta 1* 67

Index of API elements 68

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
February 2022	Beta 0	Non-confidential	Initial release of the 1.1 PAKE Extension specification
October 2022	Beta 1	Non-confidential	Relicensed as open source under CC BY-SA 4.0.

The detailed changes in each release are described in *Document change history* on page 66.

TODO items

The following items are marked up as TODO in the document source:

Todo:

In this example, how does using a 'concatenation of elements' depiction compare to the 'bullet list of elements' approach used in the Weierstrass public key format in §9.6.4? For example, the above would be described as:

> For example, for SPAKE2+ over P-256 (secp256r1), the output from `psa_export_public_key()` would be the concatenation of:
>
> - w0, as a big-endian encoded, 32-byte string
> - The byte `0x04`
> - x_L (the x-coordinate of L), as a big-endian encoded, 32-byte string
> - y_L (the y-coordinate of L), as a big-endian encoded, 32-byte string

---

original entry

Todo:

In this example, how does the short-hand notation — [v]$_n$ — compare with the text description approach used in the Weierstrass public key format in §9.6.4, or the function-based (e.g. `I2OSP()`) approach used in texts such as SEC1? For example, the above would be described as:

> For example, for SPAKE2+ over P-256 (secp256r1), the output from `psa_export_public_key()` would be:
>
> I2OSP(w0, 32) || `0x04` || I2OSP(x_L, 32) || I2OSP(y_L, 32)

---

original entry

Todo:

Would it be better to provide an explicit definition for all of the elliptic curves over which SPAKE2+ is defined, rather than just provide a single example?

---

original entry

Todo:

It might also be time to decide on how to style/format pseudo-mathematical content of the specification. Presently there is a arbitrary mixture of `monospace code/LaTeX-source-style material a^b = 1, F_q` (as typical in IETF RFCs) and *emphasized* or regular font .rst material a^b = 1, F_q (seen in NIST publications, and some IETF RFCs). But we also have the ability to use the `:math:` role to render like LaTeX: $a^b = 1, \mathbb{F}_q$ (used in SECG and some NIST publications). For comparison:

Monospace	`I2OSP(w0, 32) || 0x04 || I2OSP(x_L, 32) || I2OSP(y_L, 32)`
Styled	[w0]$_{32}$ || `0x04` || [x_L]$_{32}$ || [y_L]$_{32}$
`:math:`	$[w0]_{32}$ || `0x04` || $[x_L]_{32}$ || $[y_L]_{32}$

original entry

Todo:

Do we need a new usage flag for augmented PAKEs? For example PSA_KEY_USAGE_PROVE/VERIFY. Or do we just use PSA_KEY_USAGE_DERIVE as specified by psa_pake_set_password_key()?`

original entry

Todo:

Would a table of required w0s/w1s lengths for each of the supported SPAKE2+ elliptic curve groups be useful here?

original entry

PSA Certified Crypto API

License

Text and illustrations

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

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
[MBED-TLS]		Arm Ltd, *Mbed TLS*. github.com/ARMmbed/mbedtls
[SEC1]		Standards for Efficient Cryptography, *SEC 1: Elliptic Curve Cryptography*, May 2009. www.secg.org/sec1-v2.pdf
[RFC8235]		IETF, *Schnorr Non-interactive Zero-Knowledge Proof*, September 2017. tools.ietf.org/html/rfc8235.html
[RFC8236]		IETF, *J-PAKE: Password-Authenticated Key Exchange by Juggling*, September 2017. tools.ietf.org/html/rfc8236.html
[RFC9383]		IETF, *SPAKE2+, an Augmented Password-Authenticated Key Exchange (PAKE) Protocol*, September 2023. tools.ietf.org/html/rfc9383.html
[SPAKE2P-2]		IETF, *SPAKE2+, an Augmented PAKE*, December 2020 (Draft). datatracker.ietf.org/doc/draft-bar-cfrg-spake2plus-02
[MATTER]		CSA, *Matter Specification*, *Version 1.2*, October 2023. csa-iot.org/all-solutions/matter/

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*.

continues on next page

Table 3 – continued from previous page

Term	Meaning
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.
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*.

continues on next page

Table 3 – continued from previous page

Term	Meaning
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.
PAKE	See *Password-authenticated key exchange*.
Password-authenticated key exchange (PAKE)	An interactive method for two or more parties to establish cryptographic keys based on knowledge of a low entropy secret, such as a password. This can provide strong security for communication from a weak password, because the password is not directly communicated as part of the key exchange.
Persistent key	A key that is stored in protected non-volatile memory.
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.
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 0058 1.2 PAKE Extension Beta (Issue 2) [DRAFT]).
- 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 PAKE Extension

This document introduces an extension to the *PSA Certified Crypto API* [PSA-CRYPT] specification, to provide support for *Password-authenticated key exchange* (PAKE) algorithms, and specifically for the J-PAKE algorithm.

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.

Note:

This extension has been developed in conjunction with the *Mbed TLS* [MBED-TLS] project, which is developing an implementation of the Crypto API.

Note

This version of the document includes *Rationale* commentary that provides background information relating to the design decisions that led to the current proposal. This enables the reader to understand the wider context and alternative approaches that have been considered.

1.3 Objectives for the PAKE Extension

1.3.1 Scheme review

There are a number of PAKE protocols in circulation, but none of them are used widely in practice, and they are very different in scope and mechanics. The API proposed for the Crypto API focuses on schemes that are most likely to be needed by users. A number of factors are used to identify important PAKE algorithms.

Wide deployment

Considering PAKE schemes with already wide deployment allows users with existing applications to migrate to the Crypto API. Currently there is only one scheme with non-negligible success in the industry: Secure Remote Password (SRP).

Requests

Some PAKE schemes have been requested by the community and need to be supported. Currently, these are SPAKE2+ and J-PAKE (in particular the Elliptic Curve based variant, sometimes known as ECJPAKE)

Standardization

There are PAKE schemes that are being standardized and will be recommended for use in future protocols. To ensure that the API is future proof, we need to consider these. The CFRG recommends CPace and OPAQUE for use in IETF protocols. These are also recommended for use in TLS and IKE in the future.

Applications

Some of these schemes are used in popular protocols. This information confirms the choices already made and can help to extend the list in future:

PAKE scheme	Protocols
J-PAKE	TLS, THREAD v1
SPAKE2+	CHIP
SRP	TLS
OPAQUE	TLS, IKE
CPace	TLS, IKE
Dragonfly	WPA3 (Before including the Dragonblood attack should be considered as well.)
SPAKE	Kerberos 5 v1.17
PACE	IKEv2
AugPAKE	IKEv2

1.3.2 Scope of the PAKE Extension

The following PAKE schemes are considered in the Crypto API design:

Balanced	Augmented
J-PAKE	SRP
SPAKE2	SPAKE2+
CPace	OPAQUE

Scope of this specification

The current API proposal provides the general interface for PAKE algorithms, and the specific interface for J-PAKE.

Out of scope

PAKE protocols that do not fit into any of the above categories are not taken into consideration in the proposed API. Some schemes like that are:

PAKE scheme	Specification
AMP	IEEE 1363.2, ISO/IEC 11770-4
BSPEKE2	IEEE 1363.2
PAKZ	IEEE 1363.2
PPK	IEEE 1363.2
SPEKE	IEEE 1363.2
WSPEKE	IEEE 1363.2
SPEKE	IEEE 1363.2
PAK	IEEE 1363.2, X.1035, RFC 5683
EAP-PWD	RFC 5931
EAP-EKE	RFC 6124
IKE-PSK	RFC 6617
PACE for IKEv2	RFC 6631
AugPAKE for IKEv2	RFC 6628
PAR	IEEE 1363.2
SESPAKE	RFC 8133
ITU-T	X.1035
SPAKE1	

continues on next page

Table 4 – continued from previous page

PAKE scheme	Specification
Dragonfly	
B-SPEKE	
PKEX	
EKE	
Augmented-EKE	
PAK-X	
PAKE	

The exception is SPAKE2, because of it is related to SPAKE2+.

2 Password-authenticated key exchange (PAKE)

This is a proposed PAKE interface for *PSA Certified Crypto API* [PSA-CRYPT]. It is not part of the official Crypto API yet.

> **Note:**
>
> The content of this specification is not part of the stable Crypto API and may change substantially from version to version.

2.1 Algorithm encoding

A new algorithm category is added for PAKE algorithms. The algorithm category table in [PSA-CRYPT] Appendix B is extended with the information in Table 5.

Table 5 New algorithm identifier categories

Algorithm category	CAT	Category details
PAKE	`0x0A`	See *PAKE algorithm encoding*

2.1.1 PAKE algorithm encoding

The algorithm identifier for PAKE algorithms defined in this specification are encoded as shown in Figure 1.

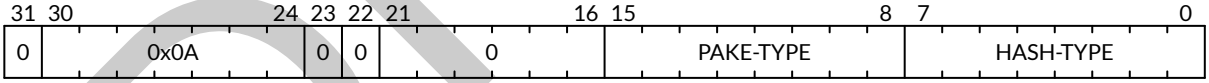

Figure 1 PAKE algorithm encoding

The defined values for PAKE-TYPE are shown in Table 6.

The permitted values of HASH-TYPE depend on the specific PAKE algorithm.

Table 6 PAKE algorithm sub-type values

PAKE algorithm	PAKE-TYPE	Algorithm identifier	Algorithm value
J-PAKE	`0x01`	`PSA_ALG_JPAKE(hash)`	`0x0A0001hh` [a]
SPAKE2+ wih HMAC	`0x04`	`PSA_ALG_SPAKE2P_HMAC(hash)`	`0x0A0004hh` [a]
SPAKE2+ wih CMAC	`0x05`	`PSA_ALG_SPAKE2P_CMAC(hash)`	`0x0A0005hh` [a]
SPAKE2+ for Matter	`0x06`	`PSA_ALG_SPAKE2P_MATTER`	`0x0A000609` [a]

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

2.2 Key encoding

A new type of asymmetric key is added for the SPAKE2+ algorithms. The Asymmetric key sub-type values table in [PSA-CRYPT] Appendix B is extended with the information in Table 7.

Table 7 New SPAKE2+ asymmetric key sub-type

Asymmetric key type	ASYM-TYPE	Details
SPAKE2+	4	See *SPAKE2+ key encoding*

Rationale

The ASYM-TYPE value 4 is selected as this has the same parity as the ECC sub-type, which have the value 1. The enables the same ECC-FAMILY and P values to be used when encoding a SPAKE2+ key type, as is used in the Elliptic Curve key types.

2.2.1 SPAKE2+ key encoding

The key type for SPAKE2+ keys defined in this specification are encoded as shown in Figure 2.

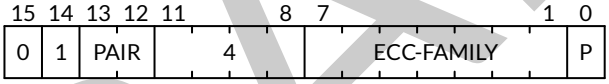

Figure 2 SPAKE2+ key encoding

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

The defined values for ECC-FAMILY and P are shown in Table 8.

Table 8 SPAKE2+ key family values

SPAKE2+ group	ECC-FAMILY	P	ECC family [a]	Public key value	Key pair value
SECP R1	0x09	0	`PSA_ECC_FAMILY_SECP_R1`	`0x4412`	`0x7412`
Twisted Edwards	0x21	0	`PSA_ECC_FAMILY_TWISTED_EDWARDS`	`0x4442`	`0x7442`

a. The key type value is constructed from the Elliptic Curve family using either `PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY(family)` or `PSA_KEY_TYPE_SPAKE2P_KEY_PAIR(family)` as required.

2.3 Key formats

A SPAKE2+ public key can be exported and imported, to enable use cases that require offline registration.

The public key consists of the two values w0 and L, which result from the SPAKE2+ registration phase. w0 is a scalar in the same range as a private Elliptic curve key from the group used as the SPAKE2+ primitive group. L is a point on the curve, similar to a public key from the same group.

The default format for the SPAKE2+ public key is the concatenation of the formatted values for w0 and L, using the the standard formats for Elliptic curve keys. For example, for SPAKE2+ over P-256 (secp256r1), the output from `psa_export_public_key()` would be:

$[\ w0\ ]_{32}\ ||\ \texttt{0x04}\ ||\ [\ x_L\ ]_{32}\ ||\ [\ y_L\ ]_{32}$

Where $[\ v\ ]_n$ is an n-byte, big-endian encoding of the integer value v.

Todo:

In this example, how does using a 'concatenation of elements' depiction compare to the 'bullet list of elements' approach used in the Weierstrass public key format in §9.6.4? For example, the above would be described as:

> For example, for SPAKE2+ over P-256 (secp256r1), the output from `psa_export_public_key()` would be the concatenation of:
>
> - w0, as a big-endian encoded, 32-byte string
> - The byte `0x04`
> - x_L (the x-coordinate of L), as a big-endian encoded, 32-byte string
> - y_L (the y-coordinate of L), as a big-endian encoded, 32-byte string

---

Todo:

In this example, how does the short-hand notation — $[\ v\ ]_n$ — compare with the text description approach used in the Weierstrass public key format in §9.6.4, or the function-based (e.g. `I2OSP()`) approach used in texts such as SEC1? For example, the above would be described as:

> For example, for SPAKE2+ over P-256 (secp256r1), the output from `psa_export_public_key()` would be:
>
> $\text{I2OSP}(w0, 32)\ ||\ \texttt{0x04}\ ||\ \text{I2OSP}(x_L, 32)\ ||\ \text{I2OSP}(y_L, 32)$

Todo:

Would it be better to provide an explicit definition for all of the elliptic curves over which SPAKE2+ is defined, rather than just provide a single example?

Todo:

It might also be time to decide on how to style/format pseudo-mathematical content of the specification. Presently there is a arbitrary mixture of `monospace code/LaTeX-source-style material`

`a^b = 1, F_q` (as typical in IETF RFCs) and *emphasized* or regular font .rst material $a^b = 1$, F_q (seen in NIST publications, and some IETF RFCs). But we also have the ability to use the `:math:` role to render like LaTeX: $a^b = 1, \mathbb{F}_q$ (used in SECG and some NIST publications). For comparison:

Monospace	`I2OSP(w0, 32) || 0x04 || I2OSP(x_L, 32) || I2OSP(y_L, 32)`
Styled	[w0]$_{32}$ || `0x04` || [x_L]$_{32}$ || [y_L]$_{32}$
`:math:`	$[w0]_{32}$ || `0x04` || $[x_L]_{32}$ || $[y_L]_{32}$

2.4 Changes and additions to the Programming API

2.4.1 SPAKE2+ keys

The SPAKE2+ protocol consists of three phases:

1. Registration
2. Authenticated key exchange
3. Key confirmation

The registration phase can be carried out immediately prior to the other phases, or can be carried out offline, and the result of the registration phase transferred to the participants in the protocol for later online authentication.

The Crypto API uses an asymmetric key-pair, and public-key, to store the output of the registration, for input to the authentication protocol. The registration is carried out using a key derivation operation, and the key exchange and confirmation is carried out using a PAKE operation. For a SPAKE2+ PAKE operation, the prover, or client, role requires a SPAKE2+ key-pair, while the verifier, or server, role can use either a SPAKE2+ key-pair or SPAKE2+ public key.

The SPAKE2+ algorithms are based on Elliptic curve groups, and a SPAKE2+ key is parameterized by a specific Elliptic curve. The Elliptic curve families are used to parameterize the key type, and the key size selects the specific curve. Is this overkill? - RFC9383 only specifies cipher-suites that use the SECP R1 curves and the Edwards curves, we could have a custom set of families

PSA_KEY_TYPE_SPAKE2P_KEY_PAIR (macro)

SPAKE2+ key pair: both the prover and verifier key.

```
#define PSA_KEY_TYPE_SPAKE2P_KEY_PAIR(curve) /* specification-defined value */
```

Parameters

`curve`	A value of type `psa_ecc_family_t` that identifies the Elliptic curve family to be used.

Description

The size of a SPAKE2+ key is the size associated with the Elliptic curve group, that is, ceil(log2(q)) for a curve over a field F_q. See the documentation of each Elliptic curve family for details.

Compatible algorithms

`PSA_ALG_SPAKE2P_HMAC`

`PSA_ALG_SPAKE2P_CMAC`

`PSA_ALG_SPAKE2P_MATTER`

PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY (macro)

SPAKE2+ public key: the verifier key.

```
#define PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY(curve) \
    /* specification-defined value */
```

Parameters

`curve`	A value of type `psa_ecc_family_t` that identifies the Elliptic curve family to be used.

Description

The size of an SPAKE2+ public key is the same as the corresponding private key. See `PSA_KEY_TYPE_SPAKE2P_KEY_PAIR()` and the documentation of each Elliptic curve family for details.

Compatible algorithms

`PSA_ALG_SPAKE2P_HMAC` (verification only)

`PSA_ALG_SPAKE2P_CMAC` (verification only)

`PSA_ALG_SPAKE2P_MATTER` (verification only)

PSA_KEY_TYPE_IS_SPAKE2P (macro)

Whether a key type is a SPAKE2+ key, either a key pair or a public key.

```
#define PSA_KEY_TYPE_IS_SPAKE2P(type) /* specification-defined value */
```

Parameters

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

`PSA_KEY_TYPE_IS_SPAKE2P_KEY_PAIR` (macro)

Whether a key type is a SPAKE2+ key pair.

```
#define PSA_KEY_TYPE_IS_SPAKE2P_KEY_PAIR(type) \
    /* specification-defined value */
```

Parameters

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

`PSA_KEY_TYPE_IS_SPAKE2P_PUBLIC_KEY` (macro)

Whether a key type is a SPAKE2+ public key.

```
#define PSA_KEY_TYPE_IS_SPAKE2P_PUBLIC_KEY(type) \
    /* specification-defined value */
```

Parameters

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

`PSA_KEY_TYPE_SPAKE2P_GET_FAMILY` (macro)

Extract the curve family from a SPAKE2+ key type.

```
#define PSA_KEY_TYPE_SPAKE2P_GET_FAMILY(type) /* specification-defined value */
```

Parameters

`type`	A SPAKE2+ key type: a value of type `psa_key_type_t` such that `PSA_KEY_TYPE_IS_SPAKE2P(type)` is true.

Returns: `psa_ecc_family_t`

The elliptic curve family id, if `type` is a supported SPAKE2+ key. Unspecified if `type` is not a supported SPAKE2+ key.

Derivation of SPAKE2+ keys

The SPAKE2+ key types can be output from a key derivation using `psa_key_derivation_output_key()`. The SPAKE2+ protocol recommends that a key-stretching kdf, such as PBKDF2, is used to hash the SPAKE2+ password. See RFC 9383 for details.

For example, after setting up the PBKDF2 operation, the following process will derive the SPAKE2+ key pair for use with the P-256 Elliptic curve group (This example may be more than necessary in the specification?):

1. Allocate and initialize a key attributes object:

```
psa_key_attributes_t att = PSA_KEY_ATTRIBUTES_INIT;
```

2. Set the key type and size:

```
psa_set_key_type(&att, PSA_KEY_TYPE_SPAKE2P_KEY_PAIR(PSA_ECC_FAMILY_SECP_R1));
psa_set_key_bits(&att, 256);            // for P-256
```

3. Set the key policy:

```
psa_set_key_usage_flags(&att, PSA_KEY_USAGE_????);
psa_set_key_algorithm(&att, PSA_ALG_SPAKE2P);
```

> **Todo:**
>
> Do we need a new usage flag for augmented PAKEs? For example PSA_KEY_USAGE_PROVE/VERIFY. Or do we just use PSA_KEY_USAGE_DERIVE as specified by psa_pake_set_password_key()?`

4. Derive the key:

```
psa_key_id_t sp2_key;
psa_key_derivation_output_key(&att, &kdf_op, &sp2_key);
```

The key derivation process in `psa_key_derivation_output_key()` follows the recommendations for the registration process in RFC 9383, and matches the specification of this process in the Matter specification.

For the Crypto API:

- The derivation of SPAKE2+ keys extracts ceil(log2(p)/8) + 8 bytes from the PBKDF for each of w0s and w1s, where p is the prime factor of the order of the elliptic curve group.
- The calculation of w0, w1, and L then proceeds as described in the RFC.
- A SPAKE2+ key-pair is the pair (w0, w1).
- A SPAKE2+ public key is the pair (w0, L).

> **Todo:**
>
> Would a table of required w0s/w1s lengths for each of the supported SPAKE2+ elliptic curve groups be useful here?

2.4.2 PAKE algorithms

PSA_ALG_IS_PAKE (macro)

Whether the specified algorithm is a password-authenticated key exchange.

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

Parameters

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

Returns

`1` if `alg` is a password-authenticated key exchange (PAKE) algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

PSA_ALG_JPAKE (macro)

Macro to build the Password-authenticated key exchange by juggling (J-PAKE) algorithm.

```
#define PSA_ALG_JPAKE(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.

Returns

A J-PAKE algorithm, parameterized by a specific hash.

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

Description

This is J-PAKE as defined by *J-PAKE: Password-Authenticated Key Exchange by Juggling* [RFC8236], instantiated with the following parameters:

- The group can be either an elliptic curve or defined over a finite field.
- Schnorr Non-Interactive Zero-Knowledge Proof (NIZKP) as defined by *Schnorr Non-interactive Zero-Knowledge Proof* [RFC8235], using the same group as the J-PAKE algorithm.
- A cryptographic hash function, `hash_alg`.

J-PAKE does not confirm the shared secret key that results from the key exchange.

To select these parameters and set up the cipher suite, initialize a `psa_pake_cipher_suite_t` object, and call the following functions in any order:

```
psa_pake_cipher_suite_t cipher_suite = PSA_PAKE_CIPHER_SUITE_INIT;

psa_pake_cs_set_algorithm(&cipher_suite, PSA_ALG_JPAKE(hash));
psa_pake_cs_set_primitive(&cipher_suite,
                          PSA_PAKE_PRIMITIVE(type, family, bits));
psa_pake_cs_set_key_confirmation(&cipher_suite, PSA_PAKE_UNCONFIRMED_KEY);
```

More information on selecting a specific Elliptic curve or Diffie-Hellman field is provided with the PSA_PAKE_PRIMITIVE_TYPE_ECC and PSA_PAKE_PRIMITIVE_TYPE_DH constants.

The PAKE operation for J-PAKE requires a key of type type PSA_KEY_TYPE_PASSWORD or PSA_KEY_TYPE_PASSWORD_HASH. The same key value must be provided to the PAKE operation in both participants.

The key can be the password text itself, in an agreed character encoding, or some value derived from the password as required by a higher level protocol. For low-entropy passwords, it is recommended that a key-stretching derivation algorithm, such as PBKDF2, is used, and the resulting password hash is used as the PAKE operation key.

The J-PAKE operation follows the protocol shown in Figure 3 on page 25.

J-PAKE does not assign roles to the participants, so it is not necessary to call psa_pake_set_role().

J-PAKE requires both an application and a peer identity. If the peer identity provided to psa_pake_set_peer() does not match the data received from the peer, then the call to psa_pake_input() for the PSA_PAKE_STEP_ZK_PROOF step will fail with PSA_ERROR_INVALID_SIGNATURE.

The shared secret that is produced by J-PAKE is not suitable for use as an encryption key. It must be used as an input to a key derivation operation to produce additional cryptographic keys.

The following steps demonstrate the application code for 'User' in Figure 3 on page 25. The input and output steps must be carried out in exactly the same sequence as shown.

1. To prepare a J-PAKE operation, initialize and set up a psa_pake_operation_t object by calling the following functions:

```
psa_pake_operation_t jpake = PSA_PAKE_OPERATION_INIT;

psa_pake_setup(&jpake, pake_key, &cipher_suite);
psa_pake_set_user(&jpake, ...);
psa_pake_set_peer(&jpake, ...);
```

   The password is provided as key pake_key, with type PSA_KEY_TYPE_PASSWORD or PSA_KEY_TYPE_PASSWORD_HASH. This can be the password text itself, in an agreed character encoding, or some value derived from the password as required by a higher level protocol.

   The key material is used as an array of bytes, which is converted to an integer as described in *SEC 1: Elliptic Curve Cryptography* [SEC1] §2.3.8, before reducing it modulo *q*. Here, *q* is the order of the group defined by the cipher-suite primitive. psa_pake_setup() will return an error if the result of the conversion and reduction is 0.

After setup, the key exchange flow for J-PAKE is as follows:

1. To get the first round data that needs to be sent to the peer, call:

```
// Get g1
psa_pake_output(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
// Get V1, the ZKP public key for x1
psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
// Get r1, the ZKP proof for x1
psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
```

(continues on next page)

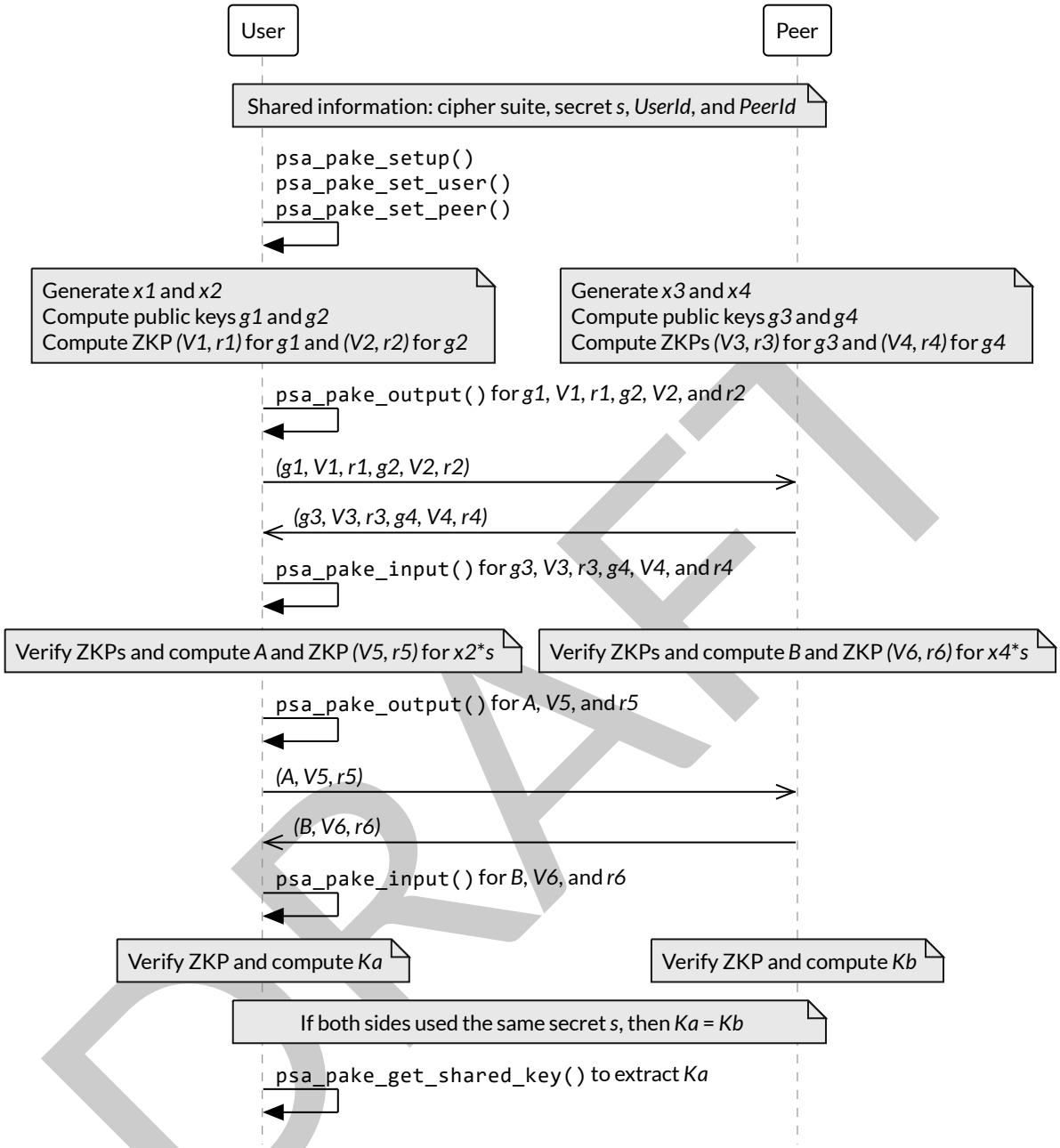

Figure 3 The J-PAKE protocol

The variable names *x1*, *g1*, and so on, are taken from the finite field implementation of J-PAKE in [RFC8236] §2. Details of the computation for the key shares and zero-knowledge proofs are in [RFC8236] and [RFC8235].

(continued from previous page)

```
// Get g2
psa_pake_output(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
// Get V2, the ZKP public key for x2
psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
// Get r2, the ZKP proof for x2
psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
```

2. To provide the first round data received from the peer to the operation, call:

```
// Set g3
psa_pake_input(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
// Set V3, the ZKP public key for x3
psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
// Set r3, the ZKP proof for x3
psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
// Set g4
psa_pake_input(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
// Set V4, the ZKP public key for x4
psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
// Set r4, the ZKP proof for x4
psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
```

3. To get the second round data that needs to be sent to the peer, call:

```
// Get A
psa_pake_output(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
// Get V5, the ZKP public key for x2*s
psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
// Get r5, the ZKP proof for x2*s
psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
```

4. To provide the second round data received from the peer to the operation call:

```
// Set B
psa_pake_input(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
// Set V6, the ZKP public key for x4*s
psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
// Set r6, the ZKP proof for x4*s
psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
```

5. To use the shared secret, extract it as a key-derivation key. For example, to extract a derivation key for HKDF-SHA-256:

```
// Set up the key attributes
psa_key_attributes_t att = PSA_KEY_ATTRIBUTES_INIT;
psa_key_set_type(&att, PSA_KEY_TYPE_DERIVE);
psa_key_set_usage_flags(&att, PSA_KEY_USAGE_DERIVE);
psa_key_set_algorithm(&att, PSA_ALG_HKDF(PSA_ALG_SHA256));

// Get Ka=Kb=K
psa_key_id_t shared_key;
psa_pake_get_shared_key(&jpake, &att, &shared_key);
```

For more information about the format of the values which are passed for each step, see *PAKE step types* on page 42.

If the verification of a Zero-knowledge proof provided by the peer fails, then the corresponding call to

psa_pake_input() for the PSA_PAKE_STEP_ZK_PROOF step will return PSA_ERROR_INVALID_SIGNATURE.

Warning: At the end of this sequence there is a cryptographic guarantee that only a peer that used the same password is able to compute the same key. But there is no guarantee that the peer is the participant it claims to be, or that the peer used the same password during the exchange.

At this point, authentication is implicit — material encrypted or authenticated using the computed key can only be decrypted or verified by someone with the same key. The peer is not authenticated at this point, and no action should be taken by the application which assumes that the peer is authenticated, for example, by accessing restricted files.

To make the authentication explicit, there are various methods to confirm that both parties have the same key. See [RFC8236] §5 for two examples.

Compatible key types

PSA_KEY_TYPE_PASSWORD

PSA_KEY_TYPE_PASSWORD_HASH

PSA_ALG_SPAKE2P_HMAC (macro)

Macro to build the SPAKE2+ algorithm, using HMAC-based key confirmation.

```
#define PSA_ALG_SPAKE2P_HMAC(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.

Returns

A SPAKE2+ algorithm, using HMAC for key confirmation, parameterized by a specific hash.

Unspecified if hash_alg is not a supported hash algorithm.

Description

This is SPAKE2+, as defined by *SPAKE2+, an Augmented Password-Authenticated Key Exchange (PAKE) Protocol* [RFC9383], instantiated with the following parameters:

- An elliptic curve group.
- A cryptographic hash function, hash_alg.
- Key derivation function HKDF, using the same hash function, hash_alg.
- Keyed MAC function HMAC, using the same hash function, hash_alg.

For SPAKE2+, valid combinations of elliptic curve PAKE primitives and hash algorithms are described in [RFC9383] §4.

SPAKE2+ includes confirmation of the shared secret key that results from the key exchange.

To select these parameters and set up the cipher suite, initialize a psa_pake_cipher_suite_t object, and call the following functions in any order:

```
psa_pake_cipher_suite_t cipher_suite = PSA_PAKE_CIPHER_SUITE_INIT;

psa_pake_cs_set_algorithm(&cipher_suite, PSA_ALG_SPAKE2P_HMAC(hash));
psa_pake_cs_set_primitive(&cipher_suite,
                          PSA_PAKE_PRIMITIVE(PSA_PAKE_PRIMITIVE_TYPE_ECC, family, bits));
```

For more information on selecting a specific Elliptic curve, see PSA_PAKE_PRIMITIVE_TYPE_ECC.

SPAKE2+ protocol

There are two particpants in the SPAKE2+ protocol:

- The *Prover* takes the role of client. It uses the protocol to prove that it knows the secret password, and produce a shared secret.
- The *Verifier* takes the role of server. It uses the protocol to verify the client's proof, and produce a shared secret.

The PAKE operation for SPAKE2+ only accepts a SPAKE2+ key type:

- The Prover requires a PSA_KEY_TYPE_SPAKE2P_KEY_PAIR(), on the same elliptic curve specified in the PAKE cipher suite.
- The Verifier requires either a PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY(), or a PSA_KEY_TYPE_SPAKE2P_KEY_PAIR(), on the same elliptic curve specified in the PAKE cipher suite.

These keys are derived from the initial SPAKE2+ password prior to starting the PAKE operation. It is recommended to use a key-stretching derivation algorithm, for example PBKDF2. This process can take place immediately before the PAKE operation, or derived at some earlier point and persisted in the key store. Alternatively, the Verifier can be provisioned with the PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY() for the protocol, by the Prover, or some other agent. Figure 4 on page 29 illustrates some example SPAKE2+ key derivation flows.

It is recommended that the Verifier stores only the public key, because disclosure of the public key does not enable an attacker to impersonate the Prover.

Both participants in SPAKE2+ have an optional identity. If no identity value is provided, then a zero-length string is used for that identity in the protocol. If the participants do not supply the same identity values to the protocol, the computed secrets will be different, and key confirmation will fail.

The SPAKE2+ operation follows the protocol shown in Figure 5 on page 60.

The shared secret that is produced by SPAKE2+ is pseudorandom. Although it can be used directly as an encryption key, it is recommended to use the shared secret as an input to a key derivation operation to produce additional cryptographic keys.

The following steps demonstrate the application code for both 'Prover' and 'Verifier' in Figure 5 on page 60.

Prover To prepare a SPAKE2+ operation for the Prover, initialize and set up a psa_pake_operation_t object by calling the following functions:

```
psa_pake_operation_t spake2p_p = PSA_PAKE_OPERATION_INIT;

psa_pake_setup(&spake2p_p, pake_key_p, &cipher_suite);
psa_pake_set_role(&spake2p_p, PSA_PAKE_ROLE_CLIENT);
```

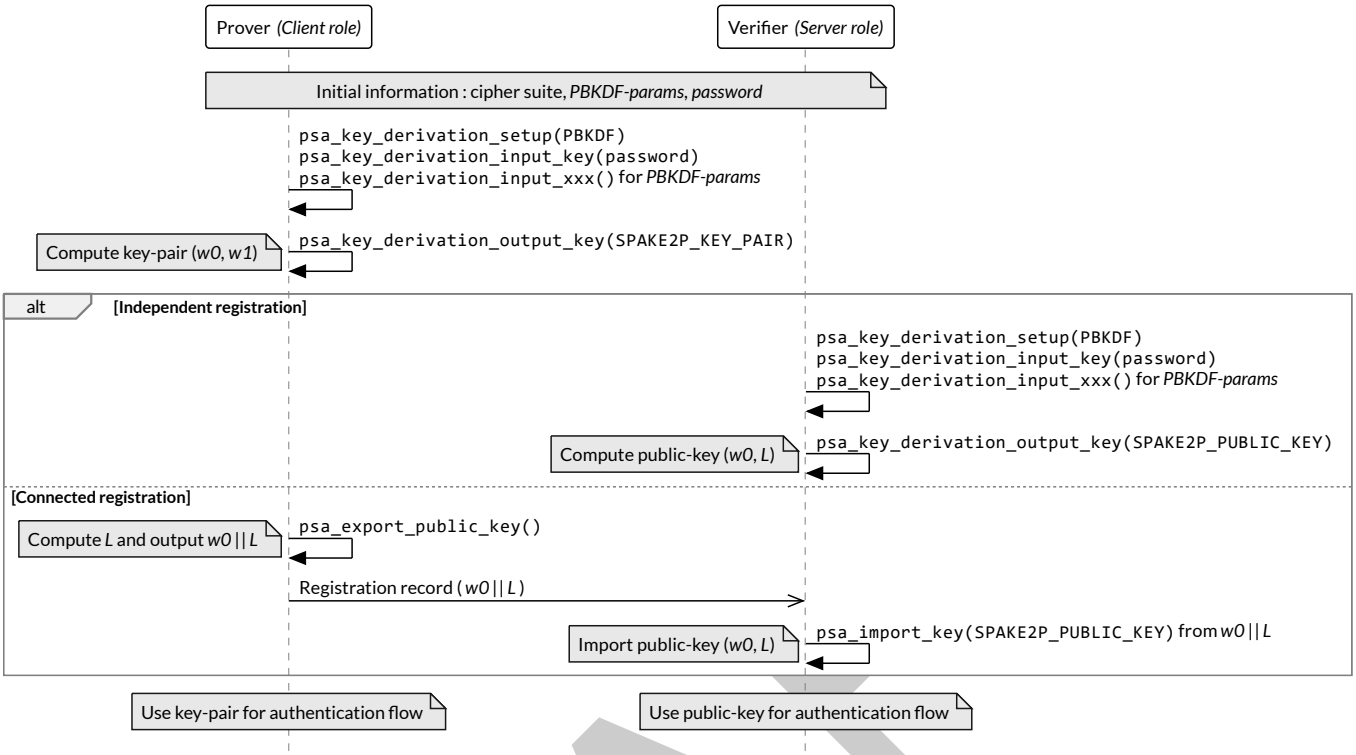

Figure 4 Examples of SPAKE2+ key derivation procedures

The variable names *w0*, *w1*, *L* are taken from the description of SPAKE2+ in [RFC9383].
Details of the computation for the key derivation values are in [RFC9383] §3.2.

The key `pake_key_p` is a SPAKE2+ key pair, `PSA_KEY_TYPE_SPAKE2P_KEY_PAIR()`.

The key must have the `PSA_KEY_USAGE_??????` usage flag.

Prover Provide any additional, optional, parameters:

```
psa_pake_set_user(&spake2p_p, ...);       // Prover identity
psa_pake_set_peer(&spake2p_p, ...);       // Verifier identity
psa_pake_set_context(&spake2p_p, ...);
```

Verifier To prepare a SPAKE2+ operation for the Verifier, initialize and set up a `psa_pake_operation_t` object by calling the following functions:

```
psa_pake_operation_t spake2p_v = PSA_PAKE_OPERATION_INIT;

psa_pake_setup(&spake2p_v, pake_key_v, &cipher_suite);
psa_pake_set_role(&spake2p_v, PSA_PAKE_ROLE_SERVER);
```

The key `pake_key_v` is a SPAKE2+ key pair, `PSA_KEY_TYPE_SPAKE2P_KEY_PAIR()`, or public key, `PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY()`.

The key must have the `PSA_KEY_USAGE_??????` usage flag.

Verifier Provide any additional, optional, parameters:

```
psa_pake_set_user(&spake2p_v, ...);       // Verifier identity
psa_pake_set_peer(&spake2p_v, ...);       // Prover identity
psa_pake_set_context(&spake2p_v, ...);
```

After setup, the key exchange flow for SPAKE2+ is as follows:

Prover To get the key share to send to the Verifier, call:

```
// Get shareP
psa_pake_output(&spake2p_p, PSA_PAKE_STEP_KEY_SHARE, ...);
```

Verifier To provide and validate the Prover key share, call:

```
// Set shareP
psa_pake_input(&spake2p_v, PSA_PAKE_STEP_KEY_SHARE, ...);
```

Verifier To get the Verifier key share and confirmation value to send to the Prover, call:

```
// Get shareV
psa_pake_output(&spake2p_v, PSA_PAKE_STEP_KEY_SHARE, ...);
// Get confirmV
psa_pake_output(&spake2p_v, PSA_PAKE_STEP_CONFIRM, ...);
```

Prover To provide and validate the Verifier key share, and confirm the Verifier key, call:

```
// Set shareV
psa_pake_input(&spake2p_p, PSA_PAKE_STEP_KEY_SHARE, ...);
// Set confirmV
psa_pake_input(&spake2p_p, PSA_PAKE_STEP_KEY_CONFIRM, ...);
```

Prover To get the Prover key confirmation value to send to the Verifier, call:

```
// Get confirmV
psa_pake_output(&spake2p_p, PSA_PAKE_STEP_CONFIRM, ...);
```

Verifier To confirm the Prover key, call:

```
// Set shareP
psa_pake_input(&spake2p_v, PSA_PAKE_STEP_CONFIRM, ...);
```

Prover To use the shared secret, extract it as a key-derivation key. For example, to extract a derivation key for HKDF-SHA-256:

```
// Set up the key attributes
psa_key_attributes_t att = PSA_KEY_ATTRIBUTES_INIT;
psa_key_set_type(&att, PSA_KEY_TYPE_DERIVE);
psa_key_set_usage_flags(&att, PSA_KEY_USAGE_DERIVE);
psa_key_set_algorithm(&att, PSA_ALG_HKDF(PSA_ALG_SHA256));

// Get K_shared
```

(continues on next page)

(continued from previous page)

```
psa_key_id_t shared_key;
psa_pake_get_shared_key(&spake2p_p, &att, &shared_key);
```

Verifier To use the shared secret, extract it as a key-derivation key. The same key attributes can be used as the Prover:

```
// Get K_shared
psa_key_id_t shared_key;
psa_pake_get_shared_key(&spake2p_v, &att, &shared_key);
```

For more information about the format of the values which are passed for each step, see *PAKE step types on page 42*.

If the validation of a key share fails, then the corresponding call to `psa_pake_input()` for the `PSA_PAKE_STEP_KEY_SHARE` step will return `PSA_ERROR_INVALID_ARGUMENT`. If the verification of a key confirmation value fails, then the corresponding call to `psa_pake_input()` for the `PSA_PAKE_STEP_CONFIRM` step will return `PSA_ERROR_INVALID_SIGNATURE`.

Compatible key types

`PSA_KEY_TYPE_SPAKE2P_KEY_PAIR`

`PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY` (verification only)

PSA_ALG_SPAKE2P_CMAC (macro)

Macro to build the SPAKE2+ algorithm, using CMAC-based key confirmation.

```
#define PSA_ALG_SPAKE2P_CMAC(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.

Returns

A SPAKE2+ algorithm, using CMAC for key confirmation, parameterized by a specific hash.

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

Description

This is SPAKE2+, as defined by *SPAKE2+, an Augmented Password-Authenticated Key Exchange (PAKE) Protocol* [RFC9383], instantiated with the following parameters:

- An elliptic curve group.
- A cryptographic hash function, `hash_alg`.
- Key derivation function HKDF, using the same hash function, `hash_alg`.
- Keyed MAC function CMAC-AES-128.

For SPAKE2+, valid combinations of elliptic curve PAKE primitives and hash algorithms for use with CMAC-AES-128 are described in [RFC9383] §4.

SPAKE2+ includes confirmation of the shared secret key that results from the key exchange.

To select these parameters and set up the cipher suite, initialize a `psa_pake_cipher_suite_t` object, and call the following functions in any order:

```
psa_pake_cipher_suite_t cipher_suite = PSA_PAKE_CIPHER_SUITE_INIT;

psa_pake_cs_set_algorithm(&cipher_suite, PSA_ALG_SPAKE2P_CMAC(hash));
psa_pake_cs_set_primitive(&cipher_suite,
                          PSA_PAKE_PRIMITIVE(PSA_PAKE_PRIMITIVE_TYPE_ECC, family, bits));
```

For more information on selecting a specific Elliptic curve, see `PSA_PAKE_PRIMITIVE_TYPE_ECC`.

The SPAKE2+ protocol flow and usage for `PSA_ALG_SPAKE2P_CMAC()` is the same as for `PSA_ALG_SPAKE2P_HMAC()`. See *SPAKE2+ protocol* on page 28.

Compatible key types

`PSA_KEY_TYPE_SPAKE2P_KEY_PAIR`

`PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY` (verification only)

PSA_ALG_SPAKE2P_MATTER (macro)

The SPAKE2+ algorithm, as used by the Matter v1 specification.

```
#define PSA_ALG_SPAKE2P_MATTER ((psa_algoirithm_t)0x0A000609)
```

This is the PAKE algorithm specified as MATTER_PAKE in *Matter Specification*, *Version 1.2* [MATTER]. This is based on draft-02 of the SPAKE2+ protocol, *SPAKE2+, an Augmented PAKE* [SPAKE2P-2].

[MATTER] specifies a single cipher suite, as follows:

- The NIST P-256 elliptic curve (secp256r1).
- The SHA256 hash function.
- Key derivation function HKDF-SHA256.
- Keyed MAC function HMAC-SHA256.

SPAKE2+ includes confirmation of the shared secret key that results from the key exchange.

To set up the cipher suite for `PSA_ALG_SPAKE2P_MATTER`, initialize a `psa_pake_cipher_suite_t` object, and call the following functions in any order:

```
psa_pake_cipher_suite_t cipher_suite = PSA_PAKE_CIPHER_SUITE_INIT;

psa_pake_cs_set_algorithm(&cipher_suite, PSA_ALG_SPAKE2P_MATTER);
psa_pake_cs_set_primitive(&cipher_suite,
                          PSA_PAKE_PRIMITIVE(PSA_PAKE_PRIMITIVE_TYPE_ECC,
                                             PSA_ECC_FAMILY_SECP_R1, 256));
```

This algorithm is compatible with the SPAKE2+ key types, key derivation, protocol flow, and the API usage described in *SPAKE2+ keys* on page 19 and *SPAKE2+ protocol* on page 28. However, the following aspects are different:

- The key schedule is different. This affects the computation of the shared secret and key confirmation values.
- The protocol inputs and outputs have been renamed between draft-02 and the final RFC, as follows:

RFC 9383	Draft-02
shareP	pA
shareV	pB
confirmP	cA
confirmV	cB
K_shared	Ke

Compatible key types

`PSA_KEY_TYPE_SPAKE2P_KEY_PAIR`

`PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY` (verification only)

2.4.3 PAKE primitives

A PAKE algorithm specifies a sequence of interactions between the participants. Many PAKE algorithms are designed to allow different cryptographic primitives to be used for the key establishment operation, so long as all the participants are using the same underlying cryptography.

The cryptographic primitive for a PAKE operation is specified using a `psa_pake_primitive_t` value, which can be constructed using the `PSA_PAKE_PRIMITIVE()` macro, or can be provided as a numerical constant value.

A PAKE primitive is required when constructing a PAKE cipher-suite object, `psa_pake_cipher_suite_t`, which fully specifies the PAKE operation to be carried out.

`psa_pake_primitive_type_t` (typedef)

Encoding of the type of the PAKE's primitive.

```
typedef uint8_t psa_pake_primitive_type_t;
```

The range of PAKE primitive type values is divided as follows:

`0x00` Reserved as an invalid primitive type.

`0x01 – 0x7f` Specification-defined primitive type. Primitive types defined by this standard always have bit 7 clear. Unallocated primitive type values in this range are reserved for future use.

`0x80 – 0xff` Implementation-defined primitive type. Implementations that define additional primitive types must use an encoding with bit 7 set.

For specification-defined primitive types, see the documentation of individual `PSA_PAKE_PRIMITIVE_TYPE_XXX` constants.

PSA_PAKE_PRIMITIVE_TYPE_ECC (macro)

The PAKE primitive type indicating the use of elliptic curves.

```
#define PSA_PAKE_PRIMITIVE_TYPE_ECC ((psa_pake_primitive_type_t)0x01)
```

The values of the `family` and `bits` components of the PAKE primitive identify a specific elliptic curve, using the same mapping that is used for ECC keys. See the definition of `psa_ecc_family_t`. Here `family` and `bits` refer to the values used to construct the PAKE primitive using `PSA_PAKE_PRIMITIVE()`.

Input and output during the operation can involve group elements and scalar values:

- The format for group elements is the same as that for public keys on the specific Elliptic curve. For more information, consult the documentation of key formats in [PSA-CRYPT].
- The format for scalars is the same as that for private keys on the specific Elliptic curve. For more information, consult the documentation of key formats in [PSA-CRYPT].

PSA_PAKE_PRIMITIVE_TYPE_DH (macro)

The PAKE primitive type indicating the use of Diffie-Hellman groups.

```
#define PSA_PAKE_PRIMITIVE_TYPE_DH ((psa_pake_primitive_type_t)0x02)
```

The values of the `family` and `bits` components of the PAKE primitive identify a specific Diffie-Hellman group, using the same mapping that is used for Diffie-Hellman keys. See the definition of `psa_dh_family_t`. Here `family` and `bits` refer to the values used to construct the PAKE primitive using `PSA_PAKE_PRIMITIVE()`.

Input and output during the operation can involve group elements and scalar values:

- The format for group elements is the same as that for public keys in the specific Diffie-Hellman group. For more information, consult the documentation of key formats in [PSA-CRYPT].
- The format for scalars is the same as that for private keys in the specific Diffie-Hellman group. For more information, consult the documentation of key formats in [PSA-CRYPT].

psa_pake_family_t (typedef)

Encoding of the family of the primitive associated with the PAKE.

```
typedef uint8_t psa_pake_family_t;
```

For more information see the documentation of individual `PSA_PAKE_PRIMITIVE_TYPE_XXX` constants.

`psa_pake_primitive_t` (typedef)

Encoding of the primitive associated with the PAKE.

```
typedef uint32_t psa_pake_primitive_t;
```

PAKE primitive values are constructed using `PSA_PAKE_PRIMITIVE()`.

> **Rationale**
>
> An integral type is required for `psa_pake_primitive_t` to enable values of this type to be compile-time-constants. This allows them to be used in `case` statements, and used to calculate static buffer sizes with `PSA_PAKE_OUTPUT_SIZE()` and `PSA_PAKE_INPUT_SIZE()`.

`PSA_PAKE_PRIMITIVE` (macro)

Construct a PAKE primitive from type, family and bit-size.

```
#define PSA_PAKE_PRIMITIVE(pake_type, pake_family, pake_bits) \
    /* specification-defined value */
```

Parameters

`pake_type`	The type of the primitive: a value of type `psa_pake_primitive_type_t`.
`pake_family`	The family of the primitive. The type and interpretation of this parameter depends on `pake_type`. For more information, consult the documentation of individual `psa_pake_primitive_type_t` constants.
`pake_bits`	The bit-size of the primitive: a value of type `size_t`. The interpretation of this parameter depends on `family`. For more information, consult the documentation of individual `psa_pake_primitive_type_t` constants.

Returns: `psa_pake_primitive_t`

The constructed primitive value. Return `0` if the requested primitive can't be encoded as `psa_pake_primitive_t`.

2.4.4 PAKE cipher suites

Most PAKE algorithms have parameters that must be specified by the application. These parameters include the following:

- The cryptographic primitive used for key establishment, specified using a PAKE primitive.
- A cryptographic hash algorithm.
- Whether the application requires the shared secret before, or after, it is confirmed.

The hash algorithm is encoded into the PAKE algorithm identifier. The `psa_pake_cipher_suite_t` object is used to fully specify a PAKE operation, combining the PAKE algorithm with all of the above parameters.

A PAKE cipher suite is required when setting up a PAKE operation in `psa_pake_setup()`.

`psa_pake_cipher_suite_t` (typedef)

The type of an object describing a PAKE cipher suite.

```
typedef /* implementation-defined type */ psa_pake_cipher_suite_t;
```

This is the object that represents the cipher suite used for a PAKE algorithm. The PAKE cipher suite specifies the PAKE algorithm, and the options selected for that algorithm. The cipher suite includes the following attributes:

- The PAKE algorithm itself.
- The hash algorithm, encoded within the PAKE algorithm.
- The PAKE primitive, which identifies the prime order group used for the key exchange operation. See *PAKE primitives* on page 33.
- Whether to confirm the shared secret.

This is an implementation-defined type. Applications that make assumptions about the content of this object will result in implementation-specific behavior, and are non-portable.

Before calling any function on a PAKE cipher suite object, the application must initialize it by any of the following means:

- Set the object to all-bits-zero, for example:

  ```
  psa_pake_cipher_suite_t cipher_suite;
  memset(&cipher_suite, 0, sizeof(cipher_suite));
  ```

- Initialize the object to logical zero values by declaring the object as static or global without an explicit initializer, for example:

  ```
  static psa_pake_cipher_suite_t cipher_suite;
  ```

- Initialize the object to the initializer `PSA_PAKE_CIPHER_SUITE_INIT`, for example:

  ```
  psa_pake_cipher_suite_t cipher_suite = PSA_PAKE_CIPHER_SUITE_INIT;
  ```

- Assign the result of the function `psa_pake_cipher_suite_init()` to the object, for example:

  ```
  psa_pake_cipher_suite_t cipher_suite;
  cipher_suite = psa_pake_cipher_suite_init();
  ```

Following initialization, the cipher-suite object contains the following values:

Attribute	Value
algorithm	`PSA_ALG_NONE` — an invalid algorithm identifier.
primitive	`0` — an invalid PAKE primitive.
hash	`PSA_ALG_NONE` — an invalid algorithm identifier.
key confirmation	`PSA_PAKE_CONFIRMED_KEY` — requesting that the secret key is confirmed before it can be returned.

The algorithm and primitive values must be set for all PAKE algorithms, the hash and key confirmation values are required for some PAKE algorithms.

> **Implementation note**
>
> Implementations are recommended to define the cipher-suite object as a simple data structure, with fields corresponding to the individual cipher suite attributes. In such an implementation, each function `psa_pake_cs_set_xxx()` sets a field and the corresponding function `psa_pake_cs_get_xxx()` retrieves the value of the field.
>
> An implementations can report attribute values that are equivalent to the original one, but have a different encoding. For example, an implementation can use a more compact representation for attributes where many bit-patterns are invalid or not supported, and store all values that it does not support as a special marker value. In such an implementation, after setting an invalid value, the corresponding get function returns an invalid value which might not be the one that was originally stored.

`PSA_PAKE_CIPHER_SUITE_INIT` (macro)

This macro returns a suitable initializer for a PAKE cipher suite object of type `psa_pake_cipher_suite_t`.

```
#define PSA_PAKE_CIPHER_SUITE_INIT /* implementation-defined value */
```

`psa_pake_cipher_suite_init` (function)

Return an initial value for a PAKE cipher suite object.

```
psa_pake_cipher_suite_t psa_pake_cipher_suite_init(void);
```

Returns: `psa_pake_cipher_suite_t`

`psa_pake_cs_get_algorithm` (function)

Retrieve the PAKE algorithm from a PAKE cipher suite.

```
psa_algorithm_t psa_pake_cs_get_algorithm(const psa_pake_cipher_suite_t* cipher_suite);
```

Parameters

`cipher_suite`	The cipher suite object to query.

Returns: `psa_algorithm_t`

The PAKE algorithm stored in the cipher suite object.

Description

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

`psa_pake_cs_set_algorithm` (function)

Declare the PAKE algorithm for the cipher suite.

```
void psa_pake_cs_set_algorithm(psa_pake_cipher_suite_t* cipher_suite,
                               psa_algorithm_t alg);
```

Parameters

`cipher_suite`	The cipher suite object to write to.
`alg`	The PAKE algorithm to write: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_PAKE(alg)` is true.

Returns: `void`

Description

This function overwrites any PAKE algorithm previously set in `cipher_suite`.

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

`psa_pake_cs_get_primitive` (function)

Retrieve the primitive from a PAKE cipher suite.

```
psa_pake_primitive_t psa_pake_cs_get_primitive(const psa_pake_cipher_suite_t* cipher_suite);
```

Parameters

`cipher_suite`	The cipher suite object to query.

Returns: `psa_pake_primitive_t`

The primitive stored in the cipher suite object.

Description

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

`psa_pake_cs_set_primitive` (function)

Declare the primitive for a PAKE cipher suite.

```
void psa_pake_cs_set_primitive(psa_pake_cipher_suite_t* cipher_suite,
                               psa_pake_primitive_t primitive);
```

Parameters

`cipher_suite`	The cipher suite object to write to.
`primitive`	The PAKE primitive to write: a value of type `psa_pake_primitive_t`. If this is `0`, the primitive type in `cipher_suite` becomes unspecified.

Returns: `void`

Description

This function overwrites any primitive previously set in `cipher_suite`.

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

`PSA_PAKE_CONFIRMED_KEY` (macro)

A key confirmation value that indicates an confirmed key in a PAKE cipher suite.

```
#define PSA_PAKE_CONFIRMED_KEY 0
```

This key confirmation value will result in the PAKE algorithm exchanging data to verify that the shared key is identical for both parties. This is the default key confirmation value in an initialized PAKE cipher suite object.

Some algorithms do not include confirmation of the shared key.

PSA_PAKE_UNCONFIRMED_KEY (macro)

A key confirmation value that indicates an unconfirmed key in a PAKE cipher suite.

```
#define PSA_PAKE_UNCONFIRMED_KEY 1
```

This key confirmation value will result in the PAKE algorithm terminating prior to confirming that the resulting shared key is identical for both parties.

Some algorithms do not support returning an unconfirmed shared key.

> **Warning:** When the shared key is not confirmed as part of the PAKE operation, the application is responsible for mitigating risks that arise from the possible mismatch in the output keys.

psa_pake_cs_get_key_confirmation (function)

Retrieve the key confirmation from a PAKE cipher suite.

```
uint32_t psa_pake_cs_get_key_confirmation(const psa_pake_cipher_suite_t* cipher_suite);
```

Parameters

`cipher_suite`	The cipher suite object to query.

Returns: `uint32_t`

A key confirmation value: either `PSA_PAKE_CONFIRMED_KEY` or `PSA_PAKE_UNCONFIRMED_KEY`.

Description

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

psa_pake_cs_set_key_confirmation (function)

Declare the key confirmation from a PAKE cipher suite.

```
void psa_pake_cs_set_key_confirmation(psa_pake_cipher_suite_t* cipher_suite,
                                      uint32_t key_confirmation);
```

Parameters

`cipher_suite`	The cipher suite object to write to.
`key_confirmation`	The key confirmation value to write: either `PSA_PAKE_CONFIRMED_KEY` or `PSA_PAKE_UNCONFIRMED_KEY`.

Returns: `void`

Description

This function overwrites any key confirmation previously set in `cipher_suite`.

The documentation of individual PAKE algorithms specifies which key confirmation values are valid for the algorithm.

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

2.4.5 PAKE roles

Some PAKE algorithms need to know which role each participant is taking in the algorithm. For example:

- Augmented PAKE algorithms typically have a client and a server participant.
- Some symmetric PAKE algorithms need to assign an order to the participants.

`psa_pake_role_t` (typedef)

Encoding of the application role in a PAKE algorithm.

```
typedef uint8_t psa_pake_role_t;
```

This type is used to encode the application's role in the algorithm being executed. For more information see the documentation of individual PAKE role constants.

`PSA_PAKE_ROLE_NONE` (macro)

A value to indicate no role in a PAKE algorithm.

```
#define PSA_PAKE_ROLE_NONE ((psa_pake_role_t)0x00)
```

This value can be used in a call to `psa_pake_set_role()` for symmetric PAKE algorithms which do not assign roles.

`PSA_PAKE_ROLE_FIRST` (macro)

The first peer in a balanced PAKE.

```
#define PSA_PAKE_ROLE_FIRST ((psa_pake_role_t)0x01)
```

Although balanced PAKE algorithms are symmetric, some of them need the peers to be ordered for the transcript calculations. If the algorithm does not need a specific ordering, then either do not call `psa_pake_set_role()`, or use `PSA_PAKE_ROLE_NONE` as the role parameter.

PSA_PAKE_ROLE_SECOND (macro)

The second peer in a balanced PAKE.

```
#define PSA_PAKE_ROLE_SECOND ((psa_pake_role_t)0x02)
```

Although balanced PAKE algorithms are symmetric, some of them need the peers to be ordered for the transcript calculations. If the algorithm does not need a specific ordering, then either do not call psa_pake_set_role(), or use PSA_PAKE_ROLE_NONE as the role parameter.

PSA_PAKE_ROLE_CLIENT (macro)

The client in an augmented PAKE.

```
#define PSA_PAKE_ROLE_CLIENT ((psa_pake_role_t)0x11)
```

Augmented PAKE algorithms need to differentiate between client and server.

PSA_PAKE_ROLE_SERVER (macro)

The server in an augmented PAKE.

```
#define PSA_PAKE_ROLE_SERVER ((psa_pake_role_t)0x12)
```

Augmented PAKE algorithms need to differentiate between client and server.

2.4.6 PAKE step types

psa_pake_step_t (typedef)

Encoding of input and output steps for a PAKE algorithm.

```
typedef uint8_t psa_pake_step_t;
```

Some PAKE algorithms need to exchange more data than a single key share. This type encodes additional input and output steps for such algorithms.

PSA_PAKE_STEP_KEY_SHARE (macro)

The key share being sent to or received from the peer.

```
#define PSA_PAKE_STEP_KEY_SHARE ((psa_pake_step_t)0x01)
```

The format for both input and output using this step is the same as the format for public keys on the group specified by the PAKE operation's primitive.

The public key formats are defined in the documentation for psa_export_public_key().

For information regarding how the group is determined, consult the documentation PSA_PAKE_PRIMITIVE().

PSA_PAKE_STEP_ZK_PUBLIC (macro)

A Schnorr NIZKP public key.

```
#define PSA_PAKE_STEP_ZK_PUBLIC ((psa_pake_step_t)0x02)
```

This is the ephemeral public key in the Schnorr Non-Interactive Zero-Knowledge Proof, this is the value denoted by *V* in [RFC8235].

The format for both input and output at this step is the same as that for public keys on the group specified by the PAKE operation's primitive.

For more information on the format, consult the documentation of `psa_export_public_key()`.

For information regarding how the group is determined, consult the documentation `PSA_PAKE_PRIMITIVE()`.

PSA_PAKE_STEP_ZK_PROOF (macro)

A Schnorr NIZKP proof.

```
#define PSA_PAKE_STEP_ZK_PROOF ((psa_pake_step_t)0x03)
```

This is the proof in the Schnorr Non-Interactive Zero-Knowledge Proof, this is the value denoted by *r* in [RFC8235].

Both for input and output, the value at this step is an integer less than the order of the group specified by the PAKE operation's primitive. The format depends on the group as well:

- For Montgomery curves, the encoding is little endian.
- For other Elliptic curves, and for Diffie-Hellman groups, the encoding is big endian. See [SEC1] §2.3.8.

In both cases leading zeroes are permitted as long as the length in bytes does not exceed the byte length of the group order.

For information regarding how the group is determined, consult the documentation `PSA_PAKE_PRIMITIVE()`.

PSA_PAKE_STEP_CONFIRM (macro)

The key confirmation value.

```
#define PSA_PAKE_STEP_CONFIRM ((psa_pake_step_t)0x04)
```

This value is used during the key confirmation phase of a PAKE protocol. The format of the value depends on the algorithm and cipher suite:

- For `PSA_ALG_SPAKE2P`, the format for both input and output at this step is the same as the output of the MAC algorithm specified in the cipher suite.

2.4.7 Multi-part PAKE operations

`psa_pake_operation_t` (typedef)

The type of the state object for PAKE operations.

```
typedef /* implementation-defined type */ psa_pake_operation_t;
```

Before calling any function on a PAKE operation object, the application must initialize it by any of the following means:

- Set the object to all-bits-zero, for example:

  ```
  psa_pake_operation_t operation;
  memset(&operation, 0, sizeof(operation));
  ```

- Initialize the object to logical zero values by declaring the object as static or global without an explicit initializer, for example:

  ```
  static psa_pake_operation_t operation;
  ```

- Initialize the object to the initializer `PSA_PAKE_OPERATION_INIT`, for example:

  ```
  psa_pake_operation_t operation = PSA_PAKE_OPERATION_INIT;
  ```

- Assign the result of the function `psa_pake_cipher_suite_init()` to the object, for example:

  ```
  psa_pake_operation_t operation;
  operation = psa_pake_operation_init();
  ```

This is an implementation-defined type. Applications that make assumptions about the content of this object will result in implementation-specific behavior, and are non-portable.

`PSA_PAKE_OPERATION_INIT` (macro)

This macro returns a suitable initializer for a PAKE operation object of type `psa_pake_operation_t`.

```
#define PSA_PAKE_OPERATION_INIT /* implementation-defined value */
```

`psa_pake_operation_init` (function)

Return an initial value for a PAKE operation object.

```
psa_pake_operation_t psa_pake_operation_init(void);
```

Returns: `psa_pake_operation_t`

psa_pake_setup (function)

Setup a password-authenticated key exchange.

```
psa_status_t psa_pake_setup(psa_pake_operation_t *operation,
                            psa_key_id_t password_key,
                            const psa_pake_cipher_suite_t *cipher_suite);
```

Parameters

`operation`	The operation object to set up. It must have been initialized as per the documentation for `psa_pake_operation_t` and not yet in use.
`password_key`	Identifier of the key holding the password or a value derived from the password. It must remain valid until the operation terminates. The valid key types depend on the PAKE algorithm, and participant role. Refer to the documentation of individual PAKE algorithms for more information, see *PAKE algorithms* on page 23. The key must permit the usage `PSA_KEY_USAGE_DERIVE`. Is this still the appropriate usage flag for SPAKE2+ key-pairs and public keys?
`cipher_suite`	The cipher suite to use. A PAKE cipher suite fully characterizes a PAKE algorithm, including the PAKE algorithm. The cipher suite must be compatible with the key type of `password_key`.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The operation is now active.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be inactive. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`password_key` is not a valid key identifier.
`PSA_ERROR_NOT_PERMITTED`	`psssword_key` does not have the `PSA_KEY_USAGE_DERIVE` flag, or it does not permit the algorithm in `cipher_suite`.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • The algorithm in `cipher_suite` is not a PAKE algorithm, or encodes an invalid hash algorithm. • The PAKE primitive in `cipher_suite` is not compatible with the PAKE algorithm. • The key confirmation value in `cipher_suite` is not compatible with the PAKE algorithm and primitive. • The key type for `password_key` is not `PSA_KEY_TYPE_PASSWORD` or `PSA_KEY_TYPE_PASSWORD_HASH`. • `password_key` is not compatible with `cipher_suite`.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error:

- The algorithm in `cipher_suite` is not a supported PAKE algorithm, or encodes an unsupported hash algorithm..
- The PAKE primitive in `cipher_suite` is not supported or not compatible with the PAKE algorithm.
- The key confirmation value in `cipher_suite` is not supported, or not compatible, with the PAKE algorithm and primitive.
- The key type or key size of `password_key` is not supported with `cipher suite`.

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

The sequence of operations to set up a password-authenticated key exchange operation is as follows:

1. Allocate a PAKE operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for `psa_pake_operation_t`. For example, using `PSA_PAKE_OPERATION_INIT`.
3. Call `psa_pake_setup()` to specify the cipher suite.
4. Call `psa_pake_set_xxx()` functions on the operation to complete the setup. The exact sequence of `psa_pake_set_xxx()` functions that needs to be called depends on the algorithm in use.

A typical sequence of calls to perform a password-authenticated key exchange:

1. Call `psa_pake_output(operation, PSA_PAKE_STEP_KEY_SHARE, ...)` to get the key share that needs to be sent to the peer.
2. Call `psa_pake_input(operation, PSA_PAKE_STEP_KEY_SHARE, ...)` to provide the key share that was received from the peer.
3. Depending on the algorithm additional calls to `psa_pake_output()` and `psa_pake_input()` might be necessary.
4. Call `psa_pake_get_shared_key()` to access the shared secret.

Refer to the documentation of individual PAKE algorithms for details on the required set up and operation for each algorithm, and for constraints on the format and content of valid passwords. See *PAKE algorithms on page 23*.

After a successful call to `psa_pake_setup()`, the operation is active, and the application must eventually terminate the operation. The following events terminate an operation:

- A successful call to `psa_pake_get_shared_key()`.
- A call to `psa_pake_abort()`.

If `psa_pake_setup()` returns an error, the operation object is unchanged. If a subsequent function call with an active operation returns an error, the operation enters an error state.

To abandon an active operation, or reset an operation in an error state, call `psa_pake_abort()`.

psa_pake_set_role (function)

Set the application role for a password-authenticated key exchange.

```
psa_status_t psa_pake_set_role(psa_pake_operation_t *operation,
                               psa_pake_role_t role);
```

Parameters

`operation`	Active PAKE operation.
`role`	A value of type `psa_pake_role_t` indicating the application role in the PAKE algorithm. See *PAKE roles* on page 41.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active, and `psa_pake_set_role()`, `psa_pake_input()`, and `psa_pake_output()` must not have been called yet. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	`role` is not a valid PAKE role in the operation's algorithm.
`PSA_ERROR_NOT_SUPPORTED`	`role` is not a valid PAKE role, or is not supported for the operation's algorithm.
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

Description

Not all PAKE algorithms need to differentiate the communicating participants. For PAKE algorithms that do not require a role to be specified, the application can do either of the following:

- Not call `psa_pake_set_role()` on the PAKE operation.
- Call `psa_pake_set_role()` with the `PSA_PAKE_ROLE_NONE` role.

Refer to the documentation of individual PAKE algorithms for more information. See *PAKE algorithms* on page 23.

psa_pake_set_user (function)

Set the user ID for a password-authenticated key exchange.

```
psa_status_t psa_pake_set_user(psa_pake_operation_t *operation,
                               const uint8_t *user_id,
                               size_t user_id_len);
```

Parameters

`operation`	Active PAKE operation.
`user_id`	The user ID to authenticate with.
`user_id_len`	Size of the `user_id` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active, and `psa_pake_set_user()`, `psa_pake_input()`, and `psa_pake_output()` must not have been called yet. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	`user_id` is not valid for the operation's algorithm and cipher suite.
`PSA_ERROR_NOT_SUPPORTED`	The value of `user_id` is not supported by the implementation.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

Description

Call this function to set the user ID. For PAKE algorithms that associate a user identifier with both participants in the session, also call `psa_pake_set_peer()` with the peer ID. For PAKE algorithms that associate a single user identifier with the session, call `psa_pake_set_user()` only.

Refer to the documentation of individual PAKE algorithms for more information. See *PAKE algorithms* on page 23.

psa_pake_set_peer (function)

Set the peer ID for a password-authenticated key exchange.

```
psa_status_t psa_pake_set_peer(psa_pake_operation_t *operation,
                               const uint8_t *peer_id,
                               size_t peer_id_len);
```

Parameters

`operation`	Active PAKE operation.
`peer_id`	The peer's ID to authenticate.
`peer_id_len`	Size of the `peer_id` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active, and `psa_pake_set_peer()`, `psa_pake_input()`, and `psa_pake_output()` must not have been called yet. • Calling `psa_pake_set_peer()` is invalid with the operation's algorithm. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	`peer_id` is not valid for the operation's algorithm and cipher suite.
`PSA_ERROR_NOT_SUPPORTED`	The value of `peer_id` is not supported by the implementation.
`PSA_ERROR_NOT_SUPPORTED`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

Description

Call this function in addition to `psa_pake_set_user()` for PAKE algorithms that associate a user identifier with both participants in the session. For PAKE algorithms that associate a single user identifier with the session, call `psa_pake_set_user()` only.

Refer to the documentation of individual PAKE algorithms for more information. See *PAKE algorithms* on page 23.

`psa_pake_set_context` (function)

Set the context data for a password-authenticated key exchange.

```
psa_status_t psa_pake_set_context(psa_pake_operation_t *operation,
                                  const uint8_t *context,
                                  size_t context_len);
```

Parameters

`operation`	Active PAKE operation.
`context`	The peer's ID to authenticate.
`context_len`	Size of the `context` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active, and `psa_pake_input()`, and `psa_pake_output()` must not have been called yet. • Calling `psa_pake_set_context()` is invalid with the operation's algorithm. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	`context` is not valid for the operation's algorithm and cipher suite.
`PSA_ERROR_NOT_SUPPORTED`	The value of `context` is not supported by the implementation.
`PSA_ERROR_NOT_SUPPORTED`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

Description

Call this function for PAKE algorithms that accept additional context data as part of the protocol setup.

Refer to the documentation of individual PAKE algorithms for more information. See *PAKE algorithms* on page 23.

psa_pake_output (function)

Get output for a step of a password-authenticated key exchange.

```
psa_status_t psa_pake_output(psa_pake_operation_t *operation,
                             psa_pake_step_t step,
                             uint8_t *output,
                             size_t output_size,
                             size_t *output_length);
```

Parameters

`operation`	Active PAKE operation.
`step`	The step of the algorithm for which the output is requested.
`output`	Buffer where the output is to be written. The format of the output depends on the `step`, see *PAKE step types* on page 42.
`output_size`	Size of the `output` buffer in bytes. This must be appropriate for the cipher suite and output step: • A sufficient output size is `PSA_PAKE_OUTPUT_SIZE(alg, primitive, step)` where `alg` and `primitive` are the PAKE algorithm and primitive in the operation's cipher suite, and `step` is the output step. • `PSA_PAKE_OUTPUT_MAX_SIZE` evaluates to the maximum output size of any supported PAKE algorithm, primitive and step.
`output_length`	On success, the number of bytes of the returned output.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The first `(*output_length)` bytes of `output` contain the output.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active and fully set up, and this call must conform to the algorithm's requirements for ordering of input and output steps. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_BUFFER_TOO_SMALL`	The size of the `output` buffer is too small. `PSA_PAKE_OUTPUT_SIZE()` or `PSA_PAKE_OUTPUT_MAX_SIZE` can be used to determine a sufficient buffer size.
`PSA_ERROR_INVALID_ARGUMENT`	`step` is not compatible with the operation's algorithm.
`PSA_ERROR_NOT_SUPPORTED`	`step` is not supported with the operation's algorithm.
`PSA_ERROR_INSUFFICIENT_ENTROPY`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

Depending on the algorithm being executed, you might need to call this function several times or you might not need to call this at all.

The exact sequence of calls to perform a password-authenticated key exchange depends on the algorithm in use. Refer to the documentation of individual PAKE algorithms for more information. See *PAKE algorithms* on page 23.

If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_pake_abort()`.

`psa_pake_input` (function)

Provide input for a step of a password-authenticated key exchange.

```
psa_status_t psa_pake_input(psa_pake_operation_t *operation,
                            psa_pake_step_t step,
                            const uint8_t *input,
                            size_t input_length);
```

Parameters

`operation`	Active PAKE operation.
`step`	The step for which the input is provided.
`input`	Buffer containing the input. The format of the input depends on the `step`, see *PAKE step types* on page 42.
`input_length`	Size of the `input` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active and fully set up, and this call must conform to the algorithm's requirements for ordering of input and output steps. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_SIGNATURE`	The verification fails for a `PSA_PAKE_STEP_ZK_PROOF` or `PSA_PAKE_STEP_CONFIRM` input step.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `step` is not compatible with the operation's algorithm. • The input is not valid for the operation's algorithm, cipher suite or `step`.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `step` is not supported with the operation's algorithm. • The input is not supported for the operation's algorithm, cipher suite or `step`.
`PSA_ERROR_INSUFFICIENT_MEMORY`	

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_CORRUPTION_DETECTED

PSA_ERROR_STORAGE_FAILURE

PSA_ERROR_DATA_CORRUPT

PSA_ERROR_DATA_INVALID

Description

Depending on the algorithm being executed, you might need to call this function several times or you might not need to call this at all.

The exact sequence of calls to perform a password-authenticated key exchange depends on the algorithm in use. Refer to the documentation of individual PAKE algorithms for more information. See *PAKE algorithms* on page 23.

`PSA_PAKE_INPUT_SIZE()` or `PSA_PAKE_INPUT_MAX_SIZE` can be used to allocate buffers of sufficient size to transfer inputs that are received from the peer into the operation.

If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_pake_abort()`.

psa_pake_get_shared_key (function)

Extract the shared secret from the PAKE as a key.

```
psa_status_t psa_pake_get_shared_key(psa_pake_operation_t *operation,
                                     const psa_key_attributes_t * attributes,
                                     psa_key_id_t * key);
```

Parameters

`operation`
: Active PAKE operation.

`attributes`
: The attributes for the new key. This function uses the attributes as follows:

- The key type is required. All PAKE algorithms can output a key of type `PSA_KEY_TYPE_DERIVE` or `PSA_KEY_TYPE_HMAC`. PAKE algorithms that produce a pseudo-random shared secret, can also output block-cipher key types, for example `PSA_KEY_TYPE_AES`.
  Refer to the documentation of individual PAKE algorithms for more information. See *PAKE algorithms* on page 23.
- The key size in `attributes` must be zero. The returned key size is always determined from the PAKE shared secret.
- The key permitted-algorithm policy is required for keys that will be used for a cryptographic operation.
- The key usage flags define what operations are permitted with the key.
- The key lifetime and identifier are required for a persistent key.

Note:

This is an input parameter: it is not updated with the final key attributes. The final attributes of the new key can be queried by calling `psa_get_key_attributes()` with the key's identifier.

`key` — On success, an identifier for the newly created key. `PSA_KEY_ID_NULL` on failure.

Returns: `psa_status_t`

`PSA_SUCCESS` — Success. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.

`PSA_ERROR_BAD_STATE` — The following conditions can result in this error:

- The state of PAKE operation `operation` is not valid: it must be ready to return the shared secret.
  For an unconfirmed key, this will be when the key-exchange output and input steps are complete, but prior to any key-confirmation output and input steps.
  For a confirmed key, this will be when all key-exchange and key-confirmation output and input steps are complete.
- The library requires initializing by a call to `psa_crypto_init()`.

`PSA_ERROR_NOT_PERMITTED` — The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.

`PSA_ERROR_ALREADY_EXISTS` — This is an attempt to create a persistent key, and there is already a persistent key with the given identifier.

`PSA_ERROR_INVALID_ARGUMENT` — The following conditions can result in this error:

- The key type is not valid for output from this operation's algorithm.
- The key size is nonzero.
- The key lifetime is invalid.
- The key identifier is not valid for the key lifetime.
- The key usage flags include invalid values.
- The key's permitted-usage algorithm is invalid.
- The key attributes, as a whole, are invalid.

`PSA_ERROR_NOT_SUPPORTED` — The key attributes, as a whole, are not supported for creation from a PAKE secret, either by the implementation in general or in the specified storage location.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

This is the final call in a PAKE operation, which retrieves the shared secret as a key. It is recommended that this key is used as an input to a key derivation operation to produce additional cryptographic keys. For some PAKE algorithms, the shared secret is also suitable for use as a key in cryptographic operations such as encryption. Refer to the documentation of individual PAKE algorithms for more information, see *PAKE algorithms* on page 23.

Depending on the key confirmation requested in the cipher suite, `psa_pake_get_shared_key()` must be called either before or after the key-confirmation output and input steps for the PAKE algorithm. The key confirmation affects the guarantees that can be made about the shared key:

Unconfirmed key

If the cipher suite used to set up the operation requested an unconfirmed key, the application must call `psa_pake_get_shared_key()` after the key-exchange output and input steps are completed. The PAKE algorithm provides a cryptographic guarantee that only a peer who used the same password, and identity inputs, is able to compute the same key. However, there is no guarantee that the peer is the participant it claims to be, and was able to compute the same key.

Since the peer is not authenticated, no action should be taken that assumes that the peer is who it claims to be. For example, do not access restricted files on the peer's behalf until an explicit authentication has succeeded.

Note:

Some PAKE algorithms do not enable the output of the shared secret until it has been confirmed.

Confirmed key

If the cipher suite used to set up the operation requested a confirmed key, the application must call `psa_pake_get_shared_key()` after the key-exchange and key-confirmation output and input steps are completed.

Following key confirmation, the PAKE algorithm provides a cryptographic guarantee that the peer used the same password and identity inputs, and has computed the identical shared secret key.

Since the peer is not authenticated, no action should be taken that assumes that the peer is who it claims to be. For example, do not access restricted files on the peer's behalf until an explicit authentication has succeeded.

Note:

Some PAKE algorithms do not include any key-confirmation steps.

The exact sequence of calls to perform a password-authenticated key exchange depends on the algorithm in use. Refer to the documentation of individual PAKE algorithms for more information. See *PAKE algorithms* on page 23.

When this function returns successfully, `operation` becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_pake_abort()`.

psa_pake_abort (function)

Abort a PAKE operation.

```
psa_status_t psa_pake_abort(psa_pake_operation_t * operation);
```

Parameters

`operation`	Initialized PAKE operation.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The operation object can now be discarded or reused.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

Description

Aborting an operation frees all associated resources except for the `operation` object itself. Once aborted, the operation object can be reused for another operation by calling `psa_pake_setup()` again.

This function can be called any time after the operation object has been initialized as described in `psa_pake_operation_t`.

In particular, calling `psa_pake_abort()` after the operation has been terminated by a call to `psa_pake_abort()` or `psa_pake_get_shared_key()` is safe and has no effect.

2.4.8 Support macros

PSA_ALG_IS_JPAKE (macro)

Whether the specified algorithm is a J-PAKE algorithm.

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

Parameters

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

Returns

`1` if `alg` is a J-PAKE algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported PAKE algorithm identifier.

Description

J-PAKE algorithms are constructed using `PSA_ALG_JPAKE(hash_alg)`.

PSA_ALG_IS_SPAKE2P (macro)

Whether the specified algorithm is a SPAKE2+ algorithm.

```
#define PSA_ALG_IS_SPAKE2P(alg) /* specification-defined value */
```

Parameters

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

Returns

`1` if `alg` is a SPAKE2+ algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported PAKE algorithm identifier.

Description

SPAKE2+ algorithms are constructed using `PSA_ALG_SPAKE2P_HMAC(hash_alg)`, `PSA_ALG_SPAKE2P_CMAC(hash_alg)`, or `PSA_ALG_SPAKE2P_MATTER`.

PSA_ALG_IS_SPAKE2P_HMAC (macro)

Whether the specified algorithm is a SPAKE2+ algorithm that uses a HMAC-based key confirmation.

```
#define PSA_ALG_IS_SPAKE2P_HMAC(alg) /* specification-defined value */
```

Parameters

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

Returns

`1` if `alg` is a SPAKE2+ algorithm that uses a HMAC-based key confirmation, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported PAKE algorithm identifier.

Description

SPAKE2+ algorithms, using HMAC-based key confirmation, are constructed using `PSA_ALG_SPAKE2P_HMAC(hash_alg)`.

PSA_ALG_IS_SPAKE2P_CMAC (macro)

Whether the specified algorithm is a SPAKE2+ algorithm that uses a CMAC-based key confirmation.

```
#define PSA_ALG_IS_SPAKE2P_CMAC(alg) /* specification-defined value */
```

Parameters

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

Returns

`1` if `alg` is a SPAKE2+ algorithm that uses a CMAC-based key confirmation, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported PAKE algorithm identifier.

Description

SPAKE2+ algorithms, using CMAC-based key confirmation, are constructed using `PSA_ALG_SPAKE2P_CMAC(hash_alg)`.

PSA_PAKE_OUTPUT_SIZE (macro)

Sufficient output buffer size for `psa_pake_output()`, in bytes.

```
#define PSA_PAKE_OUTPUT_SIZE(alg, primitive, output_step) \
    /* implementation-defined value */
```

Parameters

`alg`	A PAKE algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_PAKE(alg)` is true.
`primitive`	A primitive of type `psa_pake_primitive_t` that is compatible with algorithm `alg`.
`output_step`	A value of type `psa_pake_step_t` that is valid for the algorithm `alg`.

Returns

A sufficient output buffer size for the specified PAKE algorithm, primitive, and output step. An implementation can return either `0` or a correct size for a PAKE algorithm, primitive, and output step that it recognizes, but does not support. If the parameters are not valid, the return value is unspecified.

Description

If the size of the output buffer is at least this large, it is guaranteed that `psa_pake_output()` will not fail due to an insufficient buffer size. The actual size of the output might be smaller in any given call.

See also `PSA_PAKE_OUTPUT_MAX_SIZE`

PSA_PAKE_OUTPUT_MAX_SIZE (macro)

Sufficient output buffer size for `psa_pake_output()` for any of the supported PAKE algorithms, primitives and output steps.

```
#define PSA_PAKE_OUTPUT_MAX_SIZE /* implementation-defined value */
```

If the size of the output buffer is at least this large, it is guaranteed that `psa_pake_output()` will not fail due to an insufficient buffer size.

See also `PSA_PAKE_OUTPUT_SIZE()`.

PSA_PAKE_INPUT_SIZE (macro)

Sufficient buffer size for inputs to psa_pake_input().

```
#define PSA_PAKE_INPUT_SIZE(alg, primitive, input_step) \
    /* implementation-defined value */
```

Parameters

alg	A PAKE algorithm: a value of type psa_algorithm_t such that PSA_ALG_IS_PAKE(alg) is true.
primitive	A primitive of type psa_pake_primitive_t that is compatible with algorithm alg.
input_step	A value of type psa_pake_step_t that is valid for the algorithm alg.

Returns

A sufficient buffer size for the specified PAKE algorithm, primitive, and input step. An implementation can return either 0 or a correct size for a PAKE algorithm, primitive, and output step that it recognizes, but does not support. If the parameters are not valid, the return value is unspecified.

Description

The value returned by this macro is guaranteed to be large enough for any valid input to psa_pake_input() in an operation with the specified parameters.

This macro can be useful when transferring inputs from the peer into the PAKE operation.

See also PSA_PAKE_INPUT_MAX_SIZE

PSA_PAKE_INPUT_MAX_SIZE (macro)

Sufficient buffer size for inputs to psa_pake_input() for any of the supported PAKE algorithms, primitives and input steps.

```
#define PSA_PAKE_INPUT_MAX_SIZE /* implementation-defined value */
```

This macro can be useful when transferring inputs from the peer into the PAKE operation.

See also PSA_PAKE_INPUT_SIZE().

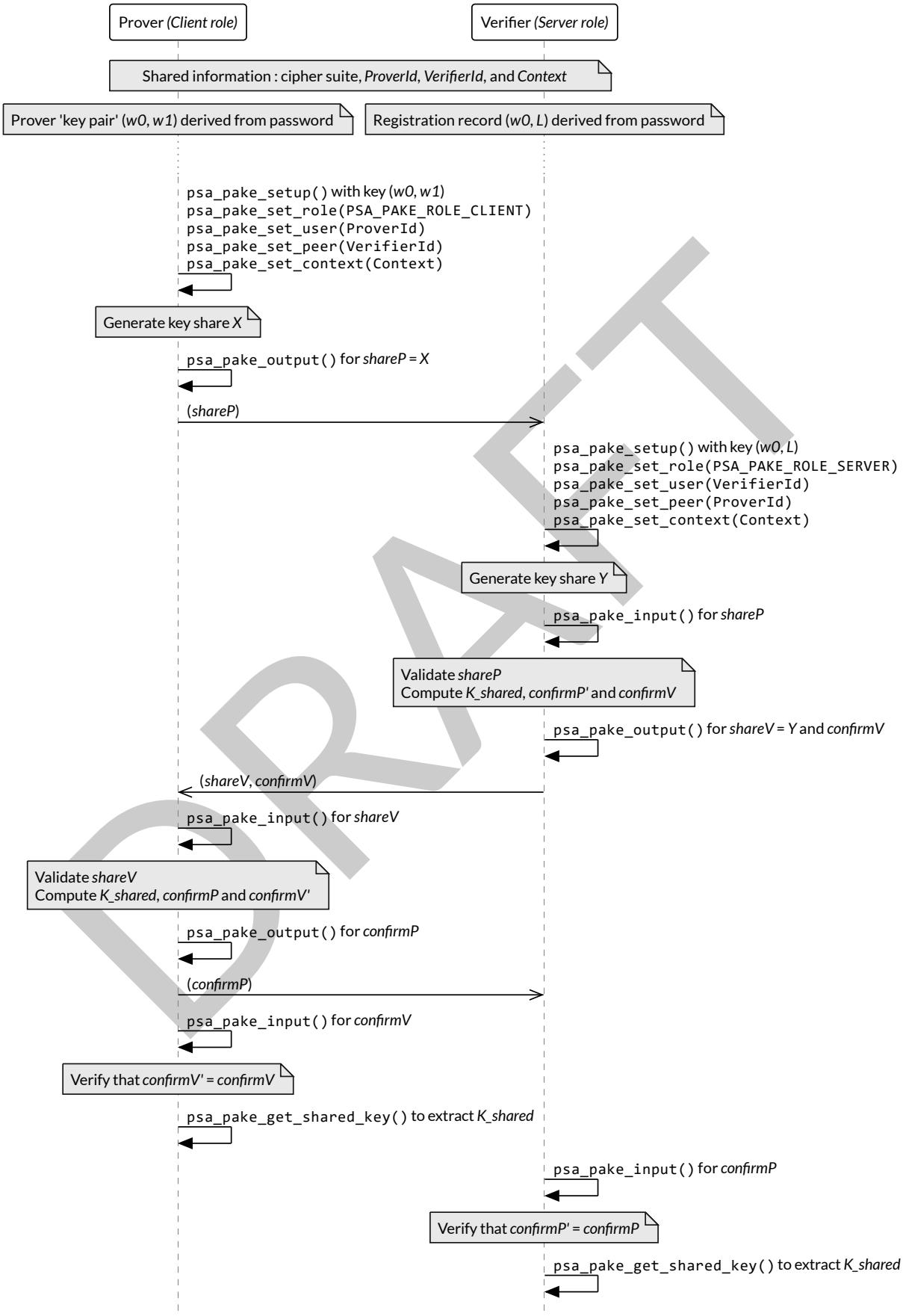

Figure 5 The SPAKE2+ authentication and key confirmation protocol

The variable names w0, w1, L, and so on, are taken from the description of SPAKE2+ in [RFC9383].
Details of the computation for the key shares is in [RFC9383] §3.3 and confirmation values in [RFC9383] §3.4.

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 v1.2 PAKE Extension beta.2
 *
 * These definitions must be embedded in, or included by, psa/crypto.h
 */

#define PSA_KEY_TYPE_SPAKE2P_KEY_PAIR(curve) /* specification-defined value */
#define PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY(curve) \
    /* specification-defined value */
#define PSA_KEY_TYPE_IS_SPAKE2P(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_SPAKE2P_KEY_PAIR(type) \
    /* specification-defined value */
#define PSA_KEY_TYPE_IS_SPAKE2P_PUBLIC_KEY(type) \
    /* specification-defined value */
#define PSA_KEY_TYPE_SPAKE2P_GET_FAMILY(type) /* specification-defined value */
#define PSA_ALG_IS_PAKE(alg) /* specification-defined value */
#define PSA_ALG_JPAKE(hash_alg) /* specification-defined value */
#define PSA_ALG_SPAKE2P_HMAC(hash_alg) /* specification-defined value */
#define PSA_ALG_SPAKE2P_CMAC(hash_alg) /* specification-defined value */
#define PSA_ALG_SPAKE2P_MATTER ((psa_algoirithm_t)0x0A000609)
typedef uint8_t psa_pake_primitive_type_t;
#define PSA_PAKE_PRIMITIVE_TYPE_ECC ((psa_pake_primitive_type_t)0x01)
#define PSA_PAKE_PRIMITIVE_TYPE_DH ((psa_pake_primitive_type_t)0x02)
typedef uint8_t psa_pake_family_t;
typedef uint32_t psa_pake_primitive_t;
#define PSA_PAKE_PRIMITIVE(pake_type, pake_family, pake_bits) \
    /* specification-defined value */
typedef /* implementation-defined type */ psa_pake_cipher_suite_t;
```

(continues on next page)

(continued from previous page)

```
#define PSA_PAKE_CIPHER_SUITE_INIT /* implementation-defined value */
psa_pake_cipher_suite_t psa_pake_cipher_suite_init(void);
psa_algorithm_t psa_pake_cs_get_algorithm(const psa_pake_cipher_suite_t* cipher_suite);
void psa_pake_cs_set_algorithm(psa_pake_cipher_suite_t* cipher_suite,
                               psa_algorithm_t alg);
psa_pake_primitive_t psa_pake_cs_get_primitive(const psa_pake_cipher_suite_t* cipher_suite);
void psa_pake_cs_set_primitive(psa_pake_cipher_suite_t* cipher_suite,
                               psa_pake_primitive_t primitive);
#define PSA_PAKE_CONFIRMED_KEY 0
#define PSA_PAKE_UNCONFIRMED_KEY 1
uint32_t psa_pake_cs_get_key_confirmation(const psa_pake_cipher_suite_t* cipher_suite);
void psa_pake_cs_set_key_confirmation(psa_pake_cipher_suite_t* cipher_suite,
                                      uint32_t key_confirmation);
typedef uint8_t psa_pake_role_t;
#define PSA_PAKE_ROLE_NONE ((psa_pake_role_t)0x00)
#define PSA_PAKE_ROLE_FIRST ((psa_pake_role_t)0x01)
#define PSA_PAKE_ROLE_SECOND ((psa_pake_role_t)0x02)
#define PSA_PAKE_ROLE_CLIENT ((psa_pake_role_t)0x11)
#define PSA_PAKE_ROLE_SERVER ((psa_pake_role_t)0x12)
typedef uint8_t psa_pake_step_t;
#define PSA_PAKE_STEP_KEY_SHARE ((psa_pake_step_t)0x01)
#define PSA_PAKE_STEP_ZK_PUBLIC ((psa_pake_step_t)0x02)
#define PSA_PAKE_STEP_ZK_PROOF ((psa_pake_step_t)0x03)
#define PSA_PAKE_STEP_CONFIRM ((psa_pake_step_t)0x04)
typedef /* implementation-defined type */ psa_pake_operation_t;
#define PSA_PAKE_OPERATION_INIT /* implementation-defined value */
psa_pake_operation_t psa_pake_operation_init(void);
psa_status_t psa_pake_setup(psa_pake_operation_t *operation,
                            psa_key_id_t password_key,
                            const psa_pake_cipher_suite_t *cipher_suite);
psa_status_t psa_pake_set_role(psa_pake_operation_t *operation,
                               psa_pake_role_t role);
psa_status_t psa_pake_set_user(psa_pake_operation_t *operation,
                               const uint8_t *user_id,
                               size_t user_id_len);
psa_status_t psa_pake_set_peer(psa_pake_operation_t *operation,
                               const uint8_t *peer_id,
                               size_t peer_id_len);
psa_status_t psa_pake_set_context(psa_pake_operation_t *operation,
                                  const uint8_t *context,
                                  size_t context_len);
psa_status_t psa_pake_output(psa_pake_operation_t *operation,
                             psa_pake_step_t step,
                             uint8_t *output,
                             size_t output_size,
                             size_t *output_length);
```

(continues on next page)

(continued from previous page)

```
psa_status_t psa_pake_input(psa_pake_operation_t *operation,
                            psa_pake_step_t step,
                            const uint8_t *input,
                            size_t input_length);
psa_status_t psa_pake_get_shared_key(psa_pake_operation_t *operation,
                                     const psa_key_attributes_t * attributes,
                                     psa_key_id_t * key);
psa_status_t psa_pake_abort(psa_pake_operation_t * operation);
#define PSA_ALG_IS_JPAKE(alg) /* specification-defined value */
#define PSA_ALG_IS_SPAKE2P(alg) /* specification-defined value */
#define PSA_ALG_IS_SPAKE2P_HMAC(alg) /* specification-defined value */
#define PSA_ALG_IS_SPAKE2P_CMAC(alg) /* specification-defined value */
#define PSA_PAKE_OUTPUT_SIZE(alg, primitive, output_step) \
    /* implementation-defined value */
#define PSA_PAKE_OUTPUT_MAX_SIZE /* implementation-defined value */
#define PSA_PAKE_INPUT_SIZE(alg, primitive, input_step) \
    /* implementation-defined value */
#define PSA_PAKE_INPUT_MAX_SIZE /* implementation-defined value */
```

Appendix B: 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:

```
#define PSA_ALG_IS_JPAKE(alg) \
    (((alg) & ~0x000000ff) == 0x0a000100)

#define PSA_ALG_IS_PAKE(alg) \
    (((alg) & 0x7f000000) == 0x0a000000)

#define PSA_ALG_IS_SPAKE2P(alg) \
    (((alg) & ~0x000003ff) == 0x0a000400)

#define PSA_ALG_IS_SPAKE2P_CMAC(alg) \
    (((alg) & ~0x000000ff) == 0x0a000500)

#define PSA_ALG_IS_SPAKE2P_HMAC(alg) \
    (((alg) & ~0x000000ff) == 0x0a000400)

#define PSA_ALG_JPAKE(hash_alg) \
    ((psa_algorithm_t) (0x0a000100 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_SPAKE2P_CMAC(hash_alg) \
    ((psa_algorithm_t) (0x0a000500 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_SPAKE2P_HMAC(hash_alg) \
    ((psa_algorithm_t) (0x0a000400 | ((hash_alg) & 0x000000ff)))

#define PSA_PAKE_PRIMITIVE(pake_type, pake_family, pake_bits) \
    ((pake_bits & 0xFFFF) != pake_bits) ? 0 :                 \
    ((psa_pake_primitive_t) (((pake_type) << 24 |             \
            (pake_family) << 16) | (pake_bits)))

#define PSA_KEY_TYPE_SPAKE2P_GET_FAMILY(type) \
```

(continues on next page)

(continued from previous page)

```
    ((psa_ecc_family_t) ((type) & 0x00ff))

#define PSA_KEY_TYPE_SPAKE2P_KEY_PAIR(curve) \
    ((psa_key_type_t) (0x7400 | (curve)))

#define PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY(curve) \
    ((psa_key_type_t) (0x4400 | (curve)))

#define PSA_KEY_TYPE_IS_SPAKE2P(type) \
    ((PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) & 0xff00) == 0x4400)

#define PSA_KEY_TYPE_IS_SPAKE2P_KEY_PAIR(type) \
    (((type) & 0xff00) == 0x7400)

#define PSA_KEY_TYPE_IS_SPAKE2P_PUBLIC_KEY(type) \
    (((type) & 0xff00) == 0x4400)
```

Appendix C: Changes to the API

C.1 Document change history

This section provides the detailed changes made between published version of the document.

C.1.1 Changes between *Beta 1* and *Beta 2*

Changes to the API

- Combined `psa_pake_set_password_key()` with `psa_pake_setup()`. This aligns the API better with other multi-part operations, and also enables an implementation to identify the key location when setting up the operation. This affects the following APIs:
- Removed `psa_pake_set_password_key()`
- Changed `psa_pake_setup()`: it now takes an additional parameter
- Replaced `psa_pake_get_implicit_key()` with `psa_pake_get_shared_key()`. This returns a new key containing the shared secret, instead of injecting the shared secret into a key derivation operation.
- Added a key confirmation attribute to the PAKE cipher suite. This indicates whether the application wants to extract the shared secret before, or after, key confirmation. See *PAKE cipher suites* on page 35. This adds the APIs `PSA_PAKE_CONFIRMED_KEY`, `PSA_PAKE_UNCONFIRMED_KEY`, `psa_pake_cs_set_key_confirmation()`, and `psa_pake_cs_get_key_confirmation()`.
- Moved the hash algorithm parameter to the PAKE cipher suite into the PAKE algorithm identifier, instead of a separate attribute of the cipher suite. This also makes the hash algorithm value available to the `PSA_PAKE_OUTPUT_SIZE()` and `PSA_PAKE_INPUT_SIZE()` macros. This affects the following APIs:
- Removed `psa_pake_cs_get_hash()` and `psa_pake_cs_set_hash()`
- Changed `PSA_ALG_JPAKE()`: it now requires a `hash_alg` parameter
- Added `PSA_ALG_IS_JPAKE()`
- Add the `PSA_PAKE_STEP_CONFIRM` PAKE step for input and output of key confirmation values.
- Add `psa_pake_set_context()` to set context data for a PAKE operation.
- Added asymmetric key types for SPAKE2+ registration, `PSA_KEY_TYPE_SPAKE2P_KEY_PAIR()` and `PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY()`. Documented the import/export public key format and key derivation process for these keys.
- Added SPAKE2+ algorithms, supporting both *SPAKE2+, an Augmented Password-Authenticated Key Exchange (PAKE) Protocol* [RFC9383] and *Matter Specification, Version 1.2* [MATTER]. Added the following APIs:
  - `PSA_ALG_SPAKE2P_HMAC()`
  - `PSA_ALG_SPAKE2P_CMAC()`
  - `PSA_ALG_SPAKE2P_MATTER`

- `PSA_ALG_IS_SPAKE2P()`
- `PSA_ALG_IS_SPAKE2P_HMAC()`
- `PSA_ALG_IS_SPAKE2P_CMAC()`

Clarifications

- Clarified the behavior of the PAKE operation following a call to `psa_pake_setup()`.

C.1.2 Changes between *Beta 0* and *Beta 1*

Other changes

- Relicensed the document under Attribution-ShareAlike 4.0 International with a patent license derived from Apache License 2.0. See *License* on page vi.

Index of API elements

PSA_A

PSA_ALG_IS_JPAKE, 56
PSA_ALG_IS_PAKE, 23
PSA_ALG_IS_SPAKE2P, 57
PSA_ALG_IS_SPAKE2P_CMAC, 57
PSA_ALG_IS_SPAKE2P_HMAC, 57
PSA_ALG_JPAKE, 23
PSA_ALG_SPAKE2P_CMAC, 31
PSA_ALG_SPAKE2P_HMAC, 27
PSA_ALG_SPAKE2P_MATTER, 32

PSA_K

PSA_KEY_TYPE_IS_SPAKE2P, 20
PSA_KEY_TYPE_IS_SPAKE2P_KEY_PAIR, 21
PSA_KEY_TYPE_IS_SPAKE2P_PUBLIC_KEY, 21
PSA_KEY_TYPE_SPAKE2P_GET_FAMILY, 21
PSA_KEY_TYPE_SPAKE2P_KEY_PAIR, 19
PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY, 20

PSA_PAKE_A

psa_pake_abort, 56

PSA_PAKE_C

PSA_PAKE_CIPHER_SUITE_INIT, 37
PSA_PAKE_CONFIRMED_KEY, 39
psa_pake_cipher_suite_init, 37
psa_pake_cipher_suite_t, 36
psa_pake_cs_get_algorithm, 37
psa_pake_cs_get_key_confirmation, 40
psa_pake_cs_get_primitive, 38
psa_pake_cs_set_algorithm, 38
psa_pake_cs_set_key_confirmation, 40
psa_pake_cs_set_primitive, 39

PSA_PAKE_F

psa_pake_family_t, 34

PSA_PAKE_G

psa_pake_get_shared_key, 53

PSA_PAKE_I

PSA_PAKE_INPUT_MAX_SIZE, 59
PSA_PAKE_INPUT_SIZE, 59
psa_pake_input, 52

PSA_PAKE_O

PSA_PAKE_OPERATION_INIT, 44
PSA_PAKE_OUTPUT_MAX_SIZE, 58
PSA_PAKE_OUTPUT_SIZE, 58
psa_pake_operation_init, 44
psa_pake_operation_t, 44
psa_pake_output, 50

PSA_PAKE_P

PSA_PAKE_PRIMITIVE, 35
PSA_PAKE_PRIMITIVE_TYPE_DH, 34
PSA_PAKE_PRIMITIVE_TYPE_ECC, 34
psa_pake_primitive_t, 35
psa_pake_primitive_type_t, 33

PSA_PAKE_R

PSA_PAKE_ROLE_CLIENT, 42
PSA_PAKE_ROLE_FIRST, 41
PSA_PAKE_ROLE_NONE, 41
PSA_PAKE_ROLE_SECOND, 42
PSA_PAKE_ROLE_SERVER, 42
psa_pake_role_t, 41

PSA_PAKE_S

PSA_PAKE_STEP_CONFIRM, 43
PSA_PAKE_STEP_KEY_SHARE, 42
PSA_PAKE_STEP_ZK_PROOF, 43
PSA_PAKE_STEP_ZK_PUBLIC, 43
psa_pake_set_context, 49
psa_pake_set_peer, 48
psa_pake_set_role, 47
psa_pake_set_user, 47
psa_pake_setup, 45
psa_pake_step_t, 42

PSA_PAKE_U

PSA_PAKE_UNCONFIRMED_KEY, 40