



PSA Certified Crypto API 1.2 PAKE Extension

Document number: AES 0058
Release Quality: Final
Issue Number: 1
Confidentiality: Non-confidential
Date of Issue: 28/03/2024

Copyright © 2022-2024 Arm Limited and/or its affiliates

FINAL RELEASE

This is an extension to the *PSA Certified Crypto API [PSA-CRYPT]* specification.

This is a FINAL release: the proposed changes and interfaces are complete and finalized, and suitable for product development.

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	iv
References	v
Terms and abbreviations	v
Potential for change	viii
Conventions	viii
Typographical conventions	viii
Numbers	viii
Current status and anticipated changes	ix
Feedback	ix
1 Introduction	10
1.1 About Platform Security Architecture	10
1.2 About the Crypto API PAKE Extension	10
1.3 Objectives for the PAKE Extension	10
1.3.1 Scheme review	10
1.3.2 Scope of the PAKE Extension	12
2 Password-authenticated key exchange (PAKE)	14
2.1 Common API for PAKE	14
2.1.1 PAKE algorithms	15
2.1.2 PAKE primitives	15
2.1.3 PAKE cipher suites	19
2.1.4 PAKE roles	24
2.1.5 PAKE step types	26
2.1.6 Multi-part PAKE operations	27
2.1.7 PAKE support macros	39
2.2 The J-PAKE protocol	41
2.2.1 J-PAKE cipher suites	41
2.2.2 J-PAKE password processing	42
2.2.3 J-PAKE operation	42
2.2.4 J-PAKE algorithms	45

2.3	The SPAKE2+ protocol	46
2.3.1	SPAKE2+ cipher suites	47
2.3.2	SPAKE2+ registration	48
2.3.3	SPAKE2+ operation	50
2.3.4	SPAKE2+ keys	53
2.3.5	SPAKE2+ algorithms	57
3	Algorithm and key type encoding	61
3.1	Algorithm encoding	61
3.1.1	PAKE algorithm encoding	61
3.2	Key encoding	62
3.2.1	SPAKE2+ key encoding	62
A	Example header file	63
A.1	psa/crypto.h	63
B	Example macro implementations	66
C	Changes to the API	68
C.1	Document change history	68
C.1.1	Changes between <i>Final</i> and <i>Final.1</i>	68
C.1.2	Changes between <i>Beta 1</i> and <i>Final</i>	68
C.1.3	Changes between <i>Beta 0</i> and <i>Beta 1</i>	69
	Index of API elements	70

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.
February 2024	Final	Non-confidential	Add support for the SPAKE2+ protocol. Rework the API to support augmented PAKE protocols, improve ease of use and implementation. API status is now Final/Release.
March 2024	Final 1	Non-confidential	Clarifications and fixes

The detailed changes in each release are described in [Document change history on page 68](#).

PSA Certified Crypto API

Copyright © 2022-2024 Arm Limited and/or its affiliates. The copyright statement reflects the fact that some draft issues of this document have been released, to a limited circulation.

License

Text and illustrations

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

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

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

About the license

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

1. Changes are made related to the defined terms, to align those defined terms with the terminology in CC BY-SA 4.0 rather than Apache 2.0 (for example, changing "Work" to "Licensed Material").
2. The scope of the defensive termination clause is changed from "any patent licenses granted to You" to "any licenses granted to You". This change is intended to help maintain a healthy ecosystem by providing additional protection to the community against patent litigation claims.

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

Source code

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

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

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

References

This document refers to the following documents.

Table 2 Documents referenced by this document

Ref	Document Number	Title
[PSA-CRYPT] [MATTER]	IHI 0086	PSA Certified Crypto API. arm-software.github.io/psa-api/crypto CSA, Matter Specification, Version 1.2, October 2023. csa-iot.org/all-solutions/matter/
[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
[SEC1]		Standards for Efficient Cryptography, SEC 1: Elliptic Curve Cryptography, May 2009. www.secg.org/sec1-v2.pdf
[SPAKE2P-2]		IETF, SPAKE2+, an Augmented PAKE (Draft 02), December 2020. datatracker.ietf.org/doc/draft-bar-cfrg-spake2plus-02
[THREAD]		Thread Group, Thread Specification 1.3.0, July 2022. www.threadgroup.org/ThreadSpec

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 <i>cipher</i> 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 <i>symmetric</i> 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 <i>symmetric</i> key.
Cryptoprocessor	The component that performs cryptographic operations. A cryptoprocessor might contain a <i>keystore</i> 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 <i>MAC</i> that uses a cryptographic key with a <i>hash</i> 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 <i>Initialization vector</i> .
KDF	See <i>Key Derivation Function</i> .
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 <i>Message Authentication Code</i> .

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 <i>symmetric</i> key.
Message digest	A <i>hash</i> of a message. Used to determine if a message has been tampered.
Multi-part operation	An <i>API</i> which splits a single cryptographic operation into a sequence of separate steps.
Non-extractable key	A key with a <i>key policy</i> that prevents it from being read by ordinary means.
Nonce	Used as an input for certain <i>AEAD</i> algorithms. Nonces must not be reused with the same key because this can break a cryptographic protocol.
PAKE	See <i>Password-authenticated key exchange</i> .
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 <i>asymmetric</i> keypair. Used to establish who produced a message.
Single-part function	An <i>API</i> 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.

Potential for change

The contents of this specification are stable for version 1.2 PAKE Extension.

The following may change in updates to the version 1.2 PAKE Extension specification:

- Small optional feature additions.
- Clarifications.

Significant additions, or any changes that affect the compatibility of the interfaces defined in this specification will only be included in a new major or minor version of the specification.

Conventions

Typographical conventions

The typographical conventions are:

<i>italic</i>	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 <i>Terms and abbreviations</i> .
Red text	Indicates an open issue.
Blue text	Indicates a link. This can be <ul style="list-style-type: none">• 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 Release/Final quality status.

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.1).
- The location in the document to which your comments apply.
- A concise explanation of your comments.

We also welcome general suggestions for additions and improvements.

1 Introduction

1.1 About Platform Security Architecture

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

1.2 About the Crypto API PAKE Extension

This document defines an extension to the *PSA Certified Crypto API* [PSA-CRYPT] specification, to provide support for *Password-authenticated key exchange* (PAKE) protocols, and specifically for the J-PAKE and SPAKE2+ protocols.

This extension API is now classed as Final, and 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 version of the document includes *Rationale* commentary that provides background information relating to the API design. 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+ – used in *Matter Specification, Version 1.2* [[MATTER](#)]
- J-PAKE (in particular the Elliptic Curve based variant, sometimes known as ECJPAKE) – used in *Thread Specification 1.3.0* [[THREAD](#)].

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+	MATTER
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 and SPAKE2+.

Out of scope

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

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)

Note:

The API defined in this specification will be integrated into a future version of [\[PSA-CRYPT\]](#).

This chapter is divided into the following sections:

- [Common API for PAKE](#) – the common interface elements, including the PAKE operation.
- [The J-PAKE protocol on page 41](#) – the J-PAKE protocol, and the associated interface elements.
- [The SPAKE2+ protocol on page 46](#) – the SPAKE2+ protocols, and the associated interface elements.

PAKE also introduces additional algorithm identifiers and key types to the Crypto API. See [Algorithm and key type encoding on page 61](#) for the encoding of these values.

Rationale

PAKE protocols are more complex operations, when compared with the other types of cryptographic operation in the Crypto API. PAKE protocols can also have multiple phases, some of which are carried out prior to the PAKE operation itself, using other parts of the Crypto API.

To improve the understanding and correct use of PAKE protocols, it helps to show the protocol flow, and to demonstrate how to implement this with this API.

2.1 Common API for PAKE

This section defines all of the common interfaces used to carry out a PAKE protocol:

- [PAKE algorithms on page 15](#)
- [PAKE primitives on page 15](#)
- [PAKE cipher suites on page 19](#)
- [PAKE roles on page 24](#)
- [PAKE step types on page 26](#)
- [Multi-part PAKE operations on page 27](#)
- [PAKE support macros on page 39](#)

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 [PSA_PAKE_PRIMITIVE_TYPE_ECC](#) and [PSA_PAKE_PRIMITIVE_TYPE_DH](#).

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 on the family values, see [PSA_PAKE_PRIMITIVE_TYPE_ECC](#) and [PSA_PAKE_PRIMITIVE_TYPE_DH](#).

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, see PSA_PAKE_PRIMITIVE_TYPE_ECC and PSA_PAKE_PRIMITIVE_TYPE_DH .
pake_bits	The bit-size of the primitive: a value of type size_t . The interpretation of this parameter depends on pake_type and family. For more information, see PSA_PAKE_PRIMITIVE_TYPE_ECC and PSA_PAKE_PRIMITIVE_TYPE_DH .

Returns: [psa_pake_primitive_t](#)

The constructed primitive value. Return 0 if the requested primitive can't be encoded as [psa_pake_primitive_t](#).

Description

A PAKE primitive value is used to specify a PAKE operation, as part of a PAKE cipher suite.

PSA_PAKE_PRIMITIVE_GET_TYPE (macro)

Extract the PAKE primitive type from a PAKE primitive.

```
#define PSA_PAKE_PRIMITIVE_GET_TYPE(pake_primitive) \  
    /* specification-defined value */
```

Parameters

pake_primitive	A PAKE primitive: a value of type psa_pake_primitive_t .
----------------	--

2.1.3 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 protocol 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 15](#).
- 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.
key confirmation	PSA_PAKE_CONFIRMED_KEY — requesting that the secret key is confirmed before it can be returned.

Valid algorithm, primitive, and key confirmation values must be set when using a PAKE cipher suite.

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 implementation 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);
```


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.

2.1.4 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 assign an order to the two 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.

2.1.5 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.

2.1.6 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

<code>operation</code>	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.
<code>password_key</code>	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 15 . The key must permit the usage <code>PSA_KEY_USAGE_DERIVE</code> .
<code>cipher_suite</code>	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 <code>password_key</code> .

Returns: [psa_status_t](#)

<code>PSA_SUCCESS</code>	Success. The operation is now active.
<code>PSA_ERROR_BAD_STATE</code>	The following conditions can result in this error: <ul style="list-style-type: none">• The operation state is not valid: it must be inactive.• The library requires initializing by a call to <code>psa_crypto_init()</code>.
<code>PSA_ERROR_INVALID_HANDLE</code>	<code>password_key</code> is not a valid key identifier.
<code>PSA_ERROR_NOT_PERMITTED</code>	<code>password_key</code> does not have the <code>PSA_KEY_USAGE_DERIVE</code> flag, or it does not permit the algorithm in <code>cipher_suite</code> .
<code>PSA_ERROR_INVALID_ARGUMENT</code>	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 or key size of `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 15](#).

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

<code>operation</code>	Active PAKE operation.
<code>role</code>	A value of type <code>psa_pake_role_t</code> indicating the application role in the PAKE algorithm. See PAKE roles on page 24 .

Returns: `psa_status_t`

<code>PSA_SUCCESS</code>	Success.
<code>PSA_ERROR_BAD_STATE</code>	The following conditions can result in this error: <ul style="list-style-type: none">• The operation state is not valid: it must be active, and <code>psa_pake_set_role()</code>, <code>psa_pake_input()</code>, and <code>psa_pake_output()</code> must not have been called yet.• The library requires initializing by a call to <code>psa_crypto_init()</code>.
<code>PSA_ERROR_INVALID_ARGUMENT</code>	The following conditions can result in this error: <ul style="list-style-type: none">• <code>role</code> is not a valid PAKE role in the operation's algorithm.• <code>role</code> is not compatible with the operation's key type.
<code>PSA_ERROR_NOT_SUPPORTED</code>	The following conditions can result in this error: <ul style="list-style-type: none">• <code>role</code> is not a valid PAKE role, or is not supported for the operation's algorithm.• <code>role</code> is not supported with the operation's key type.
<code>PSA_ERROR_COMMUNICATION_FAILURE</code>	
<code>PSA_ERROR_CORRUPTION_DETECTED</code>	

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 15](#).

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

<code>operation</code>	Active PAKE operation.
<code>user_id</code>	The user ID to authenticate with.
<code>user_id_len</code>	Size of the <code>user_id</code> buffer in bytes.

Returns: `psa_status_t`

<code>PSA_SUCCESS</code>	Success.
<code>PSA_ERROR_BAD_STATE</code>	The following conditions can result in this error: <ul style="list-style-type: none">• The operation state is not valid: it must be active, and <code>psa_pake_set_user()</code>, <code>psa_pake_input()</code>, and <code>psa_pake_output()</code> must not have been called yet.• The library requires initializing by a call to <code>psa_crypto_init()</code>.
<code>PSA_ERROR_INVALID_ARGUMENT</code>	<code>user_id</code> is not valid for the operation's algorithm and cipher suite.
<code>PSA_ERROR_NOT_SUPPORTED</code>	The value of <code>user_id</code> is not supported by the implementation.
<code>PSA_ERROR_INSUFFICIENT_MEMORY</code>	
<code>PSA_ERROR_COMMUNICATION_FAILURE</code>	
<code>PSA_ERROR_CORRUPTION_DETECTED</code>	

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 15](#).

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: <ul style="list-style-type: none">• 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 15](#).

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: <ul style="list-style-type: none">• The operation state is not valid: it must be active, and <code>psa_pake_set_context()</code>, <code>psa_pake_input()</code>, and <code>psa_pake_output()</code> must not have been called yet.• Calling <code>psa_pake_set_context()</code> is invalid with the operation's algorithm.• The library requires initializing by a call to <code>psa_crypto_init()</code>.
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 15](#).

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

<code>operation</code>	Active PAKE operation.
<code>step</code>	The step of the algorithm for which the output is requested.
<code>output</code>	Buffer where the output is to be written. The format of the output depends on the <code>step</code> , see PAKE step types on page 26 .
<code>output_size</code>	Size of the output buffer in bytes. This must be appropriate for the cipher suite and output step: <ul style="list-style-type: none">• A sufficient output size is <code>PSA_PAKE_OUTPUT_SIZE(alg, primitive, step)</code> where <code>alg</code> and <code>primitive</code> are the PAKE algorithm and primitive in the operation's cipher suite, and <code>step</code> is the output step.• <code>PSA_PAKE_OUTPUT_MAX_SIZE</code> evaluates to the maximum output size of any supported PAKE algorithm, primitive and step.
<code>output_length</code>	On success, the number of bytes of the returned output.

Returns: `psa_status_t`

<code>PSA_SUCCESS</code>	Success. The first (<code>*output_length</code>) bytes of output contain the output.
<code>PSA_ERROR_BAD_STATE</code>	The following conditions can result in this error: <ul style="list-style-type: none">• 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 <code>psa_crypto_init()</code>.
<code>PSA_ERROR_BUFFER_TOO_SMALL</code>	The size of the output buffer is too small. <code>PSA_PAKE_OUTPUT_SIZE()</code> or <code>PSA_PAKE_OUTPUT_MAX_SIZE</code> can be used to determine a sufficient buffer size.
<code>PSA_ERROR_INVALID_ARGUMENT</code>	<code>step</code> is not compatible with the operation's algorithm.
<code>PSA_ERROR_NOT_SUPPORTED</code>	<code>step</code> is not supported with the operation's algorithm.
<code>PSA_ERROR_INSUFFICIENT_ENTROPY</code>	
<code>PSA_ERROR_INSUFFICIENT_MEMORY</code>	
<code>PSA_ERROR_COMMUNICATION_FAILURE</code>	
<code>PSA_ERROR_CORRUPTION_DETECTED</code>	
<code>PSA_ERROR_STORAGE_FAILURE</code>	
<code>PSA_ERROR_DATA_CORRUPT</code>	
<code>PSA_ERROR_DATA_INVALID</code>	

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 15](#).

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 26 .
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: <ul style="list-style-type: none">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: <ul style="list-style-type: none">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: <ul style="list-style-type: none">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 15](#).

[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: <ul style="list-style-type: none">• The key type is required. All PAKE algorithms can output a key of type <code>PSA_KEY_TYPE_DERIVE</code> or <code>PSA_KEY_TYPE_HMAC</code>. PAKE algorithms that produce a pseudo-random shared secret, can also output block-cipher key types, for example <code>PSA_KEY_TYPE_AES</code>. Refer to the documentation of individual PAKE algorithms for more information. See PAKE algorithms on page 15.• The key size in <code>attributes</code> 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`

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 15](#).

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 resources 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 resources 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 15](#).

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.

2.1.7 PAKE support macros

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

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

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\(\)](#).

2.2 The J-PAKE protocol

J-PAKE is the password-authenticated key exchange by juggling protocol, defined by *J-PAKE: Password-Authenticated Key Exchange by Juggling* [RFC8236]. This protocol uses the Schnorr Non-Interactive Zero-Knowledge Proof (NIZKP), as defined by *Schnorr Non-interactive Zero-Knowledge Proof* [RFC8235].

J-PAKE is a balanced PAKE, without key confirmation.

2.2.1 J-PAKE cipher suites

When setting up a PAKE cipher suite to use the J-PAKE protocol:

- Use the [PSA_ALG_JPAKE\(\)](#) algorithm, parameterized by the required hash algorithm.
- Use a PAKE primitive for the required elliptic curve, or finite field group.
- J-PAKE does not confirm the shared secret key that results from the key exchange.

For example, the following code creates a cipher suite to select J-PAKE using P-256 with the SHA-256 hash function:

```
psa_pake_cipher_suite_t cipher_suite = PSA_PAKE_CIPHER_SUITE_INIT;  
  
psa_pake_cs_set_algorithm(&cipher_suite, PSA_ALG_JPAKE(PSA_ALG_SHA_256));  
psa_pake_cs_set_primitive(&cipher_suite,  
                        PSA_PAKE_PRIMITIVE(PSA_PAKE_PRIMITIVE_TYPE_ECC,  
                                           PSA_ECC_FAMILY_SECP_R1, 256));  
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.

2.2.2 J-PAKE password processing

The PAKE operation for J-PAKE expects a key of 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 key input to the PAKE operation.

2.2.3 J-PAKE operation

The J-PAKE operation follows the protocol shown in [Figure 2 on page 43](#).

Setup

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 following steps demonstrate the application code for 'User' in [Figure 2 on page 43](#). The code flow for the 'Peer' is the same as for 'User', as J-PAKE is a balanced PAKE.

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, ...);
```

See [J-PAKE cipher suites on page 41](#) and [J-PAKE password processing](#) for details on the requirements for the cipher suite and key.

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 \emptyset .

Key exchange

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

2. Round one.

The application can either extract the round one output values first, and then provide the round one inputs that are received from the Peer; or provide the peer inputs first, and then extract the outputs.

- To get the first round data that needs to be sent to the peer, make the following calls to `psa_pake_output()`, in the order shown:

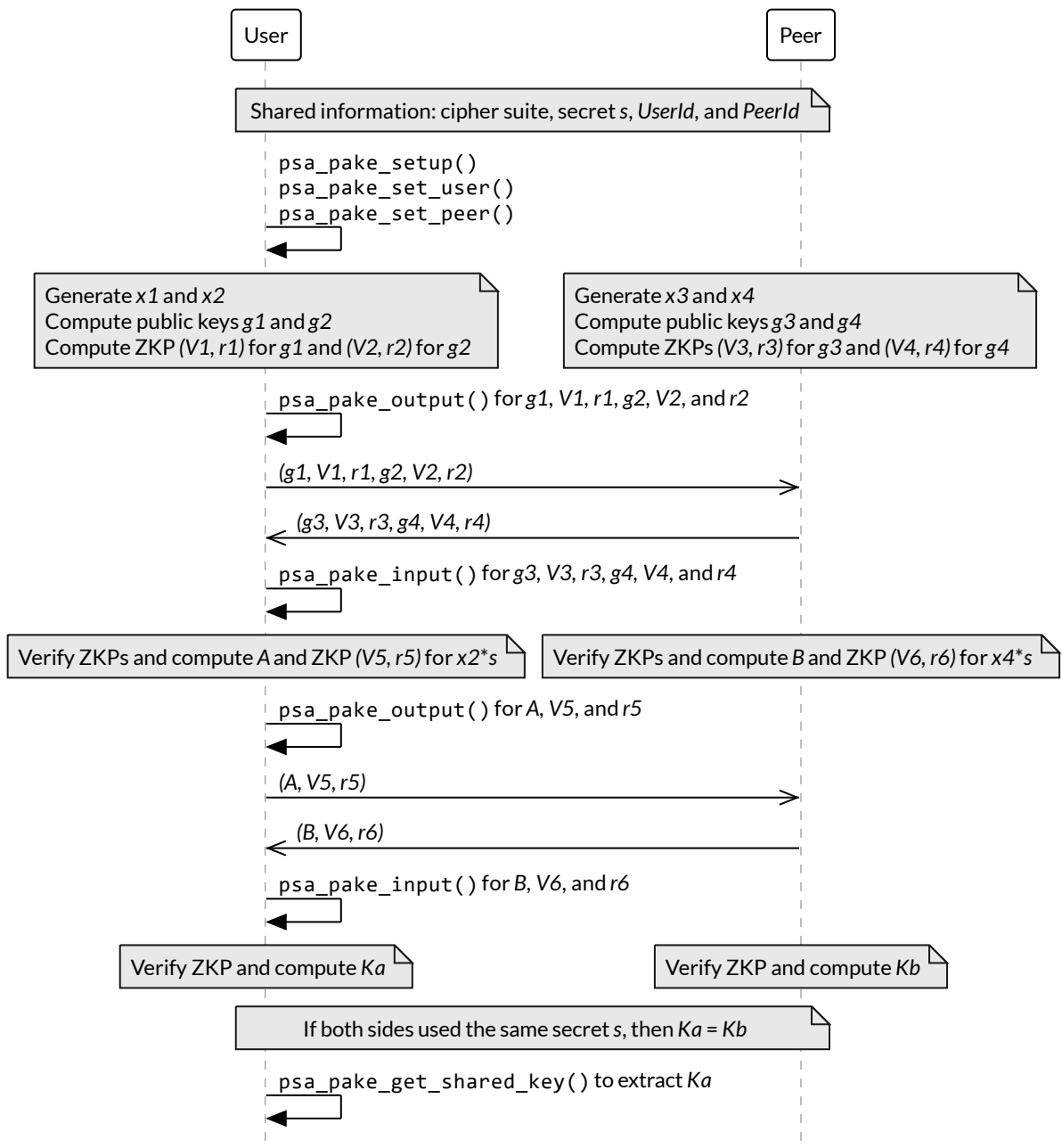


Figure 2 The J-PAKE protocol

The variable names x_1 , g_1 , 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].

```

// 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)

(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, ...);
```

- To provide the first round data received from the peer to the operation, make the following calls to `psa_pake_input()`, in the order shown:

```
// 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. Round two.

The application can either extract the round two output values first, and then provide the round two inputs that are received from the Peer; or provide the peer inputs first, and then extract the outputs.

- To get the second round data that needs to be sent to the peer, make the following calls to `psa_pake_output()`, in the order shown:

```
// 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, ...);
```

- To provide the second round data received from the peer to the operation, make the following calls to `psa_pake_input()`, in the order shown:

```
// 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, ...);
```

4. 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_set_key_type(&att, PSA_KEY_TYPE_DERIVE);
psa_set_key_usage_flags(&att, PSA_KEY_USAGE_DERIVE);
psa_set_key_algorithm(&att, PSA_ALG_HKDF(PSA_ALG_SHA_256));

// 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 26](#).

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`.

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.

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 resources.

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

2.2.4 J-PAKE algorithms

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 <code>psa_algorithm_t</code> such that <code>PSA_ALG_IS_HASH(hash_alg)</code> 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 [\[RFC8236\]](#), instantiated with the following parameters:

- The primitive group can be either an elliptic curve or defined over a finite field.
- The Schnorr NIZKP, using the same group as the J-PAKE algorithm.
- The cryptographic hash function, `hash_alg`.

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

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.

See [The J-PAKE protocol on page 41](#) for the J-PAKE protocol flow and how to implement it with the Crypto API.

Compatible key types

PSA_KEY_TYPE_PASSWORD

PSA_KEY_TYPE_PASSWORD_HASH

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\(hash_alg\)](#).

2.3 The SPAKE2+ protocol

SPAKE2+ is the augmented password-authenticated key exchange protocol, defined by SPAKE2+, an *Augmented Password-Authenticated Key Exchange (PAKE) Protocol* [\[RFC9383\]](#). SPAKE2+ includes confirmation of the shared secret key that results from the key exchange.

SPAKE2+ is required by *Matter Specification, Version 1.2* [\[MATTER\]](#), as `MATTER_PAKE`. [\[MATTER\]](#) uses an earlier draft of the SPAKE2+ protocol, *SPAKE2+, an Augmented PAKE (Draft 02)* [\[SPAKE2P-2\]](#).

Although the operation of the PAKE is similar for both of these variants, they have different key schedules for the derivation of the shared secret.

2.3.1 SPAKE2+ cipher suites

SPAKE2+ is instantiated with the following parameters:

- An elliptic curve group.
- A cryptographic hash function.
- A key derivation function.
- A keyed MAC function.

Valid combinations of these parameters are defined in the table of cipher suites in [\[RFC9383\] §4](#).

When setting up a PAKE cipher suite to use the SPAKE2+ protocol defined in [\[RFC9383\]](#):

- For cipher-suites that use HMAC for key confirmation, use the `PSA_ALG_SPAKE2P_HMAC()` algorithm, parameterized by the required hash algorithm.
- For cipher-suites that use CMAC-AES-128 for key confirmation, use the `PSA_ALG_SPAKE2P_CMAC()` algorithm, parameterized by the required hash algorithm.
- Use a PAKE primitive for the required elliptic curve.

For example, the following code creates a cipher suite to select SPAKE2+ using edwards25519 with the SHA-256 hash function:

```
psa_pake_cipher_suite_t cipher_suite = PSA_PAKE_CIPHER_SUITE_INIT;  
  
psa_pake_cs_set_algorithm(&cipher_suite, PSA_ALG_SPAKE2P_HMAC(PSA_ALG_SHA_256));  
psa_pake_cs_set_primitive(&cipher_suite,  
                        PSA_PAKE_PRIMITIVE(PSA_PAKE_PRIMITIVE_TYPE_ECC,  
                                           PSA_ECC_FAMILY_TWISTED_EDWARDS, 255));
```

When setting up a PAKE cipher suite to use the SPAKE2+ protocol used by [\[MATTER\]](#):

- Use the `PSA_ALG_SPAKE2P_MATTER` algorithm.
- Use the `PSA_PAKE_PRIMITIVE(PSA_PAKE_PRIMITIVE_TYPE_ECC, PSA_ECC_FAMILY_SECP_R1, 256)` PAKE primitive.

The following code creates a cipher suite to select the [\[MATTER\]](#) variant of SPAKE2+:

```
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));
```


2.3.2 SPAKE2+ registration

The SPAKE2+ protocol has distinct roles for the two participants:

- 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 registration phase of SPAKE2+ provides the initial password processing, described in [RFC9383] §3.2. The result of registration is two pairs of values – (w_0, w_1) and (w_0, L) – that need to be provided during the authentication phase to the Prover and Verifier, respectively. The design of SPAKE2+ ensures that knowledge of (w_0, L) does not enable an attacker to determine the password, or to compute w_1 .

In the Crypto API, the registration output values are managed as an asymmetric key-pair:

- The Prover values, (w_0, w_1) , are stored in a key of type `PSA_KEY_TYPE_SPAKE2P_KEY_PAIR()`.
- The Verifier values, (w_0, L) , are stored in a key of type `PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY()`, or derived from the matching `PSA_KEY_TYPE_SPAKE2P_KEY_PAIR()`.

The SPAKE2+ key types are parameterized by the same elliptic curve as the SPAKE2+ cipher suite.

The key pair is 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 stored by the participant. 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 3 on page 49 illustrates some example SPAKE2+ key derivation flows.

The resulting SPAKE2+ key-pair must be protected at least as well as the password. The public key, exported from the key pair, does not need to be kept confidential. 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.

The following steps demonstrate the derivation of a SPAKE2+ key pair using PBKDF2-HMAC-SHA256, for use with a SPAKE2+ cipher suite, `cipher_suite`. See [SPAKE2+ cipher suites on page 47](#) for an example of how to construct the cipher suite object.

1. Allocate and initialize a key derivation object:

```
psa_key_derivation_operation_t pbkdf = PSA_KEY_DERIVATION_OPERATION_INIT;
```

2. Setup the key derivation from the SPAKE2+ password, `password_key`, and parameters `pbkdf2_params`:

```
psa_key_derivation_setup(&pbkdf, PSA_ALG_PBKDF2_HMAC(PSA_ALG_SHA_256));  
psa_key_derivation_input_key(&pbkdf, PSA_KEY_DERIVATION_INPUT_PASSWORD, password_key);  
psa_key_derivation_input_integer(&pbkdf, PSA_KEY_DERIVATION_INPUT_COST, pbkdf2_params.cost);  
psa_key_derivation_input_bytes(&pbkdf, PSA_KEY_DERIVATION_INPUT_SALT,  
                               &pbkdf2_params.salt, pbkdf2_params.salt_len);
```

3. Allocate and initialize a key attributes object:

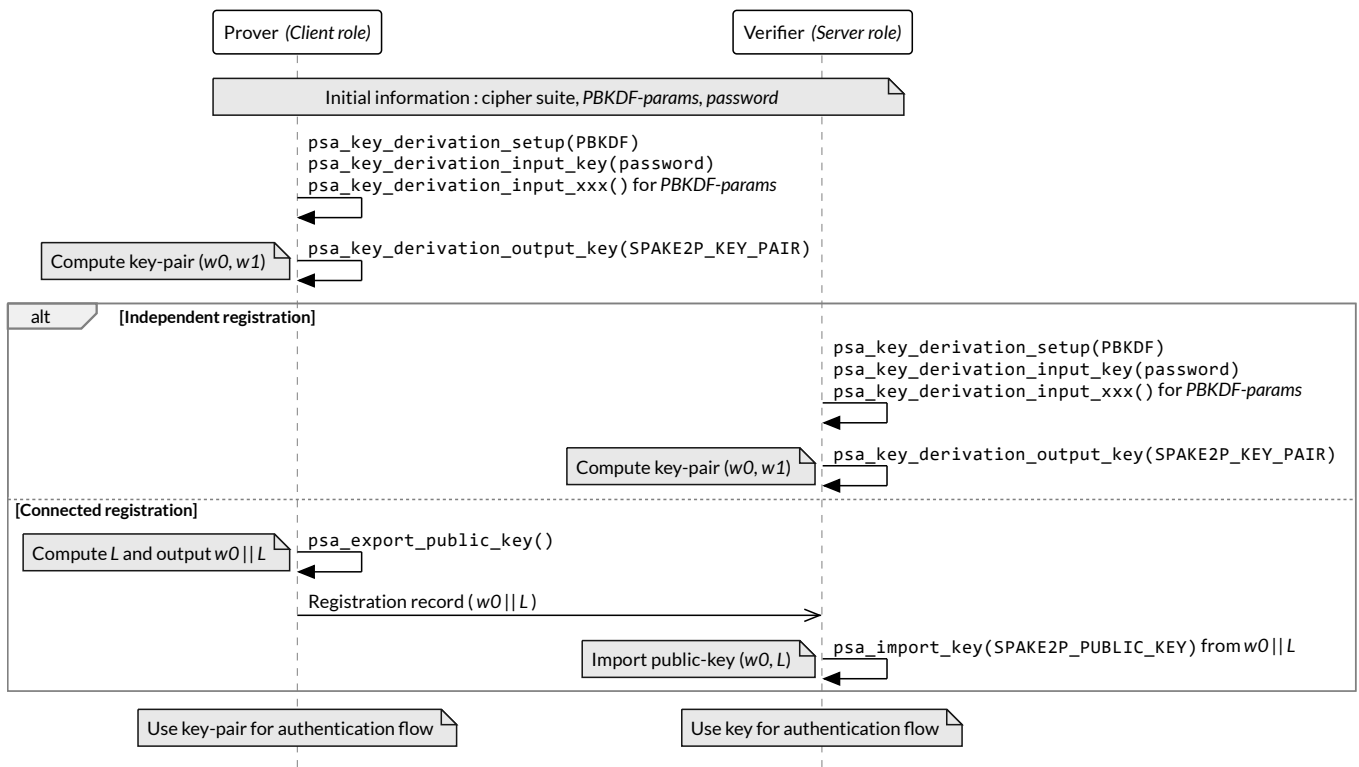


Figure 3 Examples of SPAKE2+ key derivation procedures

The variable names w_0 , w_1 , and L are taken from the description of SPAKE2+ in [RFC9383].
 Details of the computation for the key derivation values are in [RFC9383] §3.2.

```
psa_key_attributes_t att = PSA_KEY_ATTRIBUTES_INIT;
```

4. Set the key type, size, and policy from the cipher_suite object:

```
const psa_pake_primitive_t primitive = psa_pake_cs_get_primitive(&cipher_suite);

psa_set_key_type(&att,
                PSA_KEY_TYPE_SPAKE2P_KEY_PAIR(PSA_PAKE_PRIMITIVE_GET_FAMILY(primitive)));
psa_set_key_bits(&att, PSA_PAKE_PRIMITIVE_GET_BITS(primitive));
psa_set_key_usage_flags(&att, PSA_KEY_USAGE_DERIVE);
psa_set_key_algorithm(&att, psa_pake_cs_get_algorithm(&cipher_suite));
```

5. Derive the key:

```
psa_key_id_t spake2p_key;
psa_key_derivation_output_key(&att, &pbkdf, &spake2p_key);
psa_key_derivation_abort(&pbkdf);
```

See [SPAKE2+ keys on page 53](#) for details of the key types, key pair derivation, and public key format.

2.3.3 SPAKE2+ operation

The SPAKE2+ operation follows the protocol shown in [Figure 4 on page 51](#).

Setup

In SPAKE2+, the Prover uses the `PSA_PAKE_ROLE_CLIENT` role, and the Verifier uses the `PSA_PAKE_ROLE_SERVER` role.

The key passed to the Prover must be a SPAKE2+ key-pair, which is derived as recommended in [SPAKE2+ registration on page 48](#). The key passed to the Verifier can either be a SPAKE2+ key-pair, or a SPAKE2+ public key. A SPAKE2+ public key is imported from data that is output by calling `psa_export_public_key()` on a SPAKE2+ key-pair.

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 following steps demonstrate the application code for both Prover and Verifier in [Figure 4 on page 51](#).

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);
```

The key `pake_key_p` is a SPAKE2+ key pair, `PSA_KEY_TYPE_SPAKE2P_KEY_PAIR()`. See [SPAKE2+ cipher suites on page 47](#) for details on constructing a suitable cipher suite.

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()`. See [SPAKE2+ cipher suites on page 47](#) for details on constructing a suitable cipher suite.

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, ...);
```

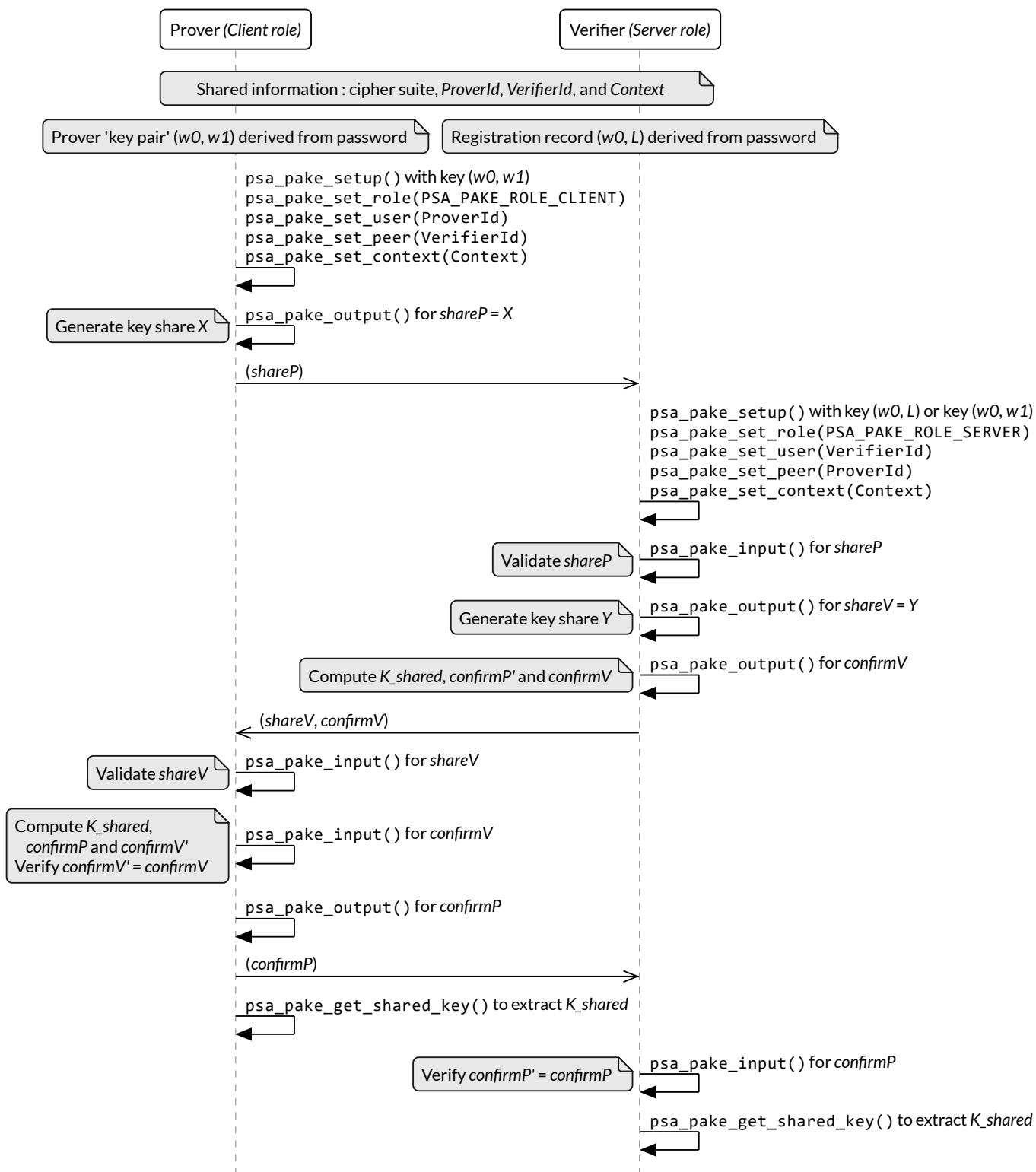


Figure 4 The SPAKE2+ authentication and key confirmation protocol

The variable names w_0 , w_1 , 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.

Key exchange

After setup, the key exchange and confirmation flow for SPAKE2+ is as follows.

Note:

The sequence of calls for the Prover, and the sequence for the Verifier, must be in exactly this order.

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 key share received from the Prover, 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 key share and verify the confirmation value received from the Verifier, 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 confirmP
psa_pake_output(&spake2p_p, PSA_PAKE_STEP_CONFIRM, ...);
```

Verifier To verify the confirmation value received from the Prover, call:

```
// Set confirmP
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_set_key_type(&att, PSA_KEY_TYPE_DERIVE);
psa_set_key_usage_flags(&att, PSA_KEY_USAGE_DERIVE);
```

(continues on next page)

(continued from previous page)

```
psa_set_key_algorithm(&att, PSA_ALG_HKDF(PSA_ALG_SHA_256));

// Get K_shared
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);
```

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.

For more information about the format of the values which are passed for each step, see [PAKE step types on page 26](#).

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`.

2.3.4 SPAKE2+ keys

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 <code>psa_ecc_family_t</code> that identifies the Elliptic curve family to be used.
-------	---

Description

The bit size of a SPAKE2+ key is the size associated with the Elliptic curve group, that is, $\lceil \log_2(q) \rceil$ for a curve over a field \mathbb{F}_q . See the documentation of each Elliptic curve family for details.

To construct a SPAKE2+ key pair, it must either be output from a key derivation operation, or imported.

The corresponding public key can be exported using `psa_export_public_key()`. See also [PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY\(\)](#).

Key derivation

A SPAKE2+ key pair can be output from a key derivation using `psa_key_derivation_output_key()`. The SPAKE2+ protocol recommends that a key-stretching key-derivation function, such as PBKDF2, is used to hash the SPAKE2+ password. See [\[RFC9383\]](#) for details.

The key derivation process in `psa_key_derivation_output_key()` follows the recommendations for the registration process in [\[RFC9383\]](#), and matches the specification of this process in [\[MATTER\]](#).

For the Crypto API:

- The derivation of SPAKE2+ keys extracts $\lceil \log_2(p)/8 \rceil + 8$ bytes from the PBKDF for each of w_0s and w_1s , where p is the prime factor of the order of the elliptic curve group. The following sizes are used for extracting w_0s and w_1s , depending on the elliptic curve:

Elliptic curve	Size of w_0s and w_1s , in bytes
P-256	40
P-384	56
P-521	74
edwards25519	40
edwards448	64

- The calculation of w_0 , w_1 , and L then proceeds as described in the RFC.

Implementation note

The values of w_0 and w_1 are required as part of the SPAKE2+ key pair.

It is [IMPLEMENTATION DEFINED](#) whether L is computed during key derivation, and stored as part of the key pair; or only computed when required from the key pair.

Key-pair format

A SPAKE2+ key pair can be exported and imported.

Warning: To create a new SPAKE2+ key pair, use `psa_key_derivation_output_key()` as described in [SPAKE2+ registration on page 48](#). This follows the recommended process described in [\[RFC9383\]](#).

Do not call `psa_import_key()` with data extracted from a key derivation operation using `psa_key_derivation_output_bytes()`. If the data is not considered invalid by `psa_import_key()`, this will result in a different, insecure key pair.

The key consists of the two values w_0 and w_1 , which result from the SPAKE2+ registration phase. w_0 and w_1 are scalars in the same range as an Elliptic curve private key from the group used as the SPAKE2+ primitive group.

For the Crypto API, the default format for a SPAKE2+ key pair is the concatenation of the formatted values for w_0 and w_1 , using the standard formats for Elliptic curve keys used by the Crypto API. For example, for SPAKE2+ over P-256 (secp256r1), the output from `psa_export_key()` would be the concatenation of:

- The P-256 private key w_0 . This is a 32-byte big-endian encoding of the integer w_0 .
- The P-256 private key w_1 . This is a 32-byte big-endian encoding of the integer w_1 .

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 <code>psa_ecc_family_t</code> 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.

To construct a SPAKE2+ public key, it must be imported.

Public key format

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 w_0 and L , which result from the SPAKE2+ registration phase. w_0 is a scalar in the same range as a Elliptic curve private 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, corresponding to the w_1 value in the key pair.

For the Crypto API, the default format for a SPAKE2+ public key is the concatenation of the formatted values for w_0 and L , using the standard formats for Elliptic curve keys used by the Crypto API. For example, for SPAKE2+ over P-256 (secp256r1), the output from `psa_export_public_key()` would be the concatenation of:

- The P-256 private key w_0 . This is a 32-byte big-endian encoding of the integer w_0 .
- The P-256 public key L . This is a 65-byte concatenation of:
 - The byte `0x04`.
 - The 32-byte big-endian encoding of the x-coordinate of L .
 - The 32-byte big-endian encoding of the y-coordinate of L .

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.

2.3.5 SPAKE2+ algorithms

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], for cipher suites that use HMAC for key confirmation. SPAKE2+ cipher suites are specified in [RFC9383] §4. The cipher suite's hash algorithm is used as input to `PSA_ALG_SPAKE2P_HMAC()`.

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.

See [The SPAKE2+ protocol on page 46](#) for the SPAKE2+ protocol flow and how to implement it with the Crypto API.

Compatible key types

`PSA_KEY_TYPE_SPAKE2P_KEY_PAIR`

`PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY` (verification only)

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)`.

3 Algorithm and key type encoding

These are encodings for a PAKE interface for *PSA Certified Crypto API* [PSA-CRYPT].

Note:

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

3.1 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

3.1.1 PAKE algorithm encoding

The algorithm identifier for PAKE algorithms defined in this specification are encoded as shown in Figure 5.

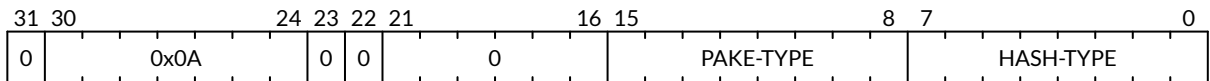


Figure 5 PAKE algorithm encoding

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+ with HMAC	0x04	PSA_ALG_SPAKE2P_HMAC (hash)	0x0A0004hh ^a
SPAKE2+ with CMAC	0x05	PSA_ALG_SPAKE2P_CMAC (hash)	0x0A0005hh ^a
SPAKE2+ for Matter	0x06	PSA_ALG_SPAKE2P_MATTER	0x0A000609

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

3.2 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. This 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.

3.2.1 SPAKE2+ key encoding

The key type for SPAKE2+ keys defined in this specification are encoded as shown in Figure 6.

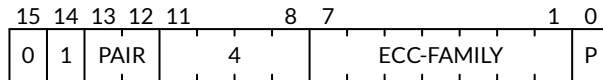


Figure 6 SPAKE2+ key encoding

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.

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
 *
 * These definitions must be embedded in, or included by, psa/crypto.h
 */

#define PSA_ALG_IS_PAKE(alg) /* specification-defined value */
typedef uint32_t psa_pake_primitive_t;
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;
#define PSA_PAKE_PRIMITIVE(pake_type, pake_family, pake_bits) \
    /* specification-defined value */
#define PSA_PAKE_PRIMITIVE_GET_TYPE(pake_primitive) \
    /* specification-defined value */
#define PSA_PAKE_PRIMITIVE_GET_FAMILY(pake_primitive) \
    /* specification-defined value */
#define PSA_PAKE_PRIMITIVE_GET_BITS(pake_primitive) \
    /* specification-defined value */
typedef /* implementation-defined type */ psa_pake_cipher_suite_t;
#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,
```

(continues on next page)

(continued from previous page)

```
        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);
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);
```

(continues on next page)

```
psa_status_t psa_pake_abort(psa_pake_operation_t * operation);
#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 */
#define PSA_ALG_JPAKE(hash_alg) /* specification-defined value */
#define PSA_ALG_IS_JPAKE(alg) /* specification-defined value */
#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_SPAKE2P_HMAC(hash_alg) /* specification-defined value */
#define PSA_ALG_SPAKE2P_CMAC(hash_alg) /* specification-defined value */
#define PSA_ALG_SPAKE2P_MATTER ((psa_algorithm_t)0x0A000609)
#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 */
```

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_PAKE_PRIMITIVE_GET_BITS(pake_primitive) \
```

(continues on next page)

```
((size_t)(pake_primitive & 0xFFFF))

#define PSA_PAKE_PRIMITIVE_GET_FAMILY(pake_primitive) \
    ((psa_pake_family_t)((pake_primitive >> 16) & 0xFF))

#define PSA_PAKE_PRIMITIVE_GET_TYPE(pake_primitive) \
    ((psa_pake_primitive_type_t)((pake_primitive >> 24) & 0xFF))

#define PSA_KEY_TYPE_SPAKE2P_GET_FAMILY(type) \
    ((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 *Final* and *Final.1*

Clarifications and fixes

- Fixed the sample code that demonstrates extracting the shared key for J-PAKE and SPAKE2+ operations.

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

API changes

- 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.
- Moved the hash algorithm parameter for 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.
- 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.
- 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 19](#).
- 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.1.3 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 iv](#).

Index of API elements

PSA_A

PSA_ALG_IS_JPAKE, [46](#)
PSA_ALG_IS_PAKE, [15](#)
PSA_ALG_IS_SPAKE2P, [59](#)
PSA_ALG_IS_SPAKE2P_CMAC, [60](#)
PSA_ALG_IS_SPAKE2P_HMAC, [59](#)
PSA_ALG_JPAKE, [45](#)
PSA_ALG_SPAKE2P_CMAC, [58](#)
PSA_ALG_SPAKE2P_HMAC, [57](#)
PSA_ALG_SPAKE2P_MATTER, [58](#)

PSA_K

PSA_KEY_TYPE_IS_SPAKE2P, [56](#)
PSA_KEY_TYPE_IS_SPAKE2P_KEY_PAIR, [56](#)
PSA_KEY_TYPE_IS_SPAKE2P_PUBLIC_KEY, [56](#)
PSA_KEY_TYPE_SPAKE2P_GET_FAMILY, [56](#)
PSA_KEY_TYPE_SPAKE2P_KEY_PAIR, [53](#)
PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY, [55](#)

PSA_PAKE_A

psa_pake_abort, [39](#)

PSA_PAKE_C

PSA_PAKE_CIPHER_SUITE_INIT, [20](#)
PSA_PAKE_CONFIRMED_KEY, [23](#)
psa_pake_cipher_suite_init, [20](#)
psa_pake_cipher_suite_t, [19](#)
psa_pake_cs_get_algorithm, [21](#)
psa_pake_cs_get_key_confirmation, [23](#)
psa_pake_cs_get_primitive, [22](#)
psa_pake_cs_set_algorithm, [21](#)
psa_pake_cs_set_key_confirmation, [24](#)
psa_pake_cs_set_primitive, [22](#)

PSA_PAKE_F

psa_pake_family_t, [17](#)

PSA_PAKE_G

psa_pake_get_shared_key, [36](#)

PSA_PAKE_I

PSA_PAKE_INPUT_MAX_SIZE, [41](#)
PSA_PAKE_INPUT_SIZE, [40](#)
psa_pake_input, [35](#)

PSA_PAKE_O

PSA_PAKE_OPERATION_INIT, [28](#)
PSA_PAKE_OUTPUT_MAX_SIZE, [40](#)
PSA_PAKE_OUTPUT_SIZE, [39](#)
psa_pake_operation_init, [28](#)
psa_pake_operation_t, [27](#)
psa_pake_output, [33](#)

PSA_PAKE_P

PSA_PAKE_PRIMITIVE, [17](#)
PSA_PAKE_PRIMITIVE_GET_BITS, [18](#)
PSA_PAKE_PRIMITIVE_GET_FAMILY, [18](#)
PSA_PAKE_PRIMITIVE_GET_TYPE, [17](#)
PSA_PAKE_PRIMITIVE_TYPE_DH, [16](#)
PSA_PAKE_PRIMITIVE_TYPE_ECC, [16](#)
psa_pake_primitive_t, [15](#)
psa_pake_primitive_type_t, [16](#)

PSA_PAKE_R

PSA_PAKE_ROLE_CLIENT, [25](#)
PSA_PAKE_ROLE_FIRST, [25](#)
PSA_PAKE_ROLE_NONE, [25](#)
PSA_PAKE_ROLE_SECOND, [25](#)
PSA_PAKE_ROLE_SERVER, [25](#)
psa_pake_role_t, [24](#)

PSA_PAKE_S

PSA_PAKE_STEP_CONFIRM, [27](#)
PSA_PAKE_STEP_KEY_SHARE, [26](#)
PSA_PAKE_STEP_ZK_PROOF, [26](#)
PSA_PAKE_STEP_ZK_PUBLIC, [26](#)
psa_pake_set_context, [33](#)
psa_pake_set_peer, [32](#)
psa_pake_set_role, [30](#)
psa_pake_set_user, [31](#)

psa_pake_setup, [28](#)
psa_pake_step_t, [26](#)

PSA_PAKE_U

PSA_PAKE_UNCONFIRMED_KEY, [23](#)