PSA Certified Crypto Driver Interface 1.0

Document number:	111106
Release Quality:	Alpha
Issue Number:	1
Confidentiality:	Non-confidential
Date of Issue:	30/09/2025

ALPHA

This specification is work in progress and should be considered to be in at Alpha quality.

See *Current status and anticipated changes* on page vii for details.

Abstract

This document describes an interface for cryptoprocessor drivers within an implementation of the PSA Certified Crypto API. This interface complements *PSA Certified Crypto API* [PSA-CRYPT], which describes the interface between a Crypto API implementation and an application.

Contents

About this document

Release information

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

Table 1 Document revision history

Date	Version	Confidentiality	Change
2020-2024	N/A	Non-confidential	Developed as part of Mbed TLS.
September 2025	1.0 Alpha-1	Non-confidential	Republished as part of the PSA Certified APIs.

The detailed changes in each release are described in *Document change history* on page 50.

PSA Certified Crypto Driver Interface

License

Text and illustrations

About the license

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

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

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

Source code

References

This document refers to the following documents.

Table 2 Documents referenced by this document

Ref	Document Number	Title
[PSA-CRYPT]	IHI 0086	*PSA Certified Crypto API*. arm-software.github.io/psa-api/crypto

Potential for change

This document is in active, collaborative development.

Issues and changes are discussed in the project on Github, alongside the PSA Certified API specifications. See github.com/arm-software/psa-api.

A list of open questions can be found in *Open questions* on page 47.

.._conventions:

Current status and anticipated changes

This document is at Alpha quality. It is shared to support ongoing development and discussion of the architecture and interfaces for Crypto Drivers.

The current planned scope for v1.0 is found here at github.com/ARM-software/psa-api/discussions/306.

There are several projects currently implementing the Crypto Driver Interface, see github.com/arm-software/psa-api/blob/main/related-projects.md.

For a practical guide, with a description of the current state of drivers in Mbed TLS, see the examples in github.com/Mbed-TLS/TF-PSA-Crypto/blob/development/docs/psa-driver-example-and-guide.md.

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 Driver Interface).
- The number and issue (111106 1.0 Alpha (Issue 1)).
- The location in the document to which your comments apply.
- A concise explanation of your comments.

We also welcome general suggestions for additions and improvements.

1 Introduction

1.1 Purpose of the driver interface

The Crypto API defines an interface that allows applications to perform cryptographic operations in a uniform way regardless of how the operations are performed. Under the hood, different keys may be stored and used in different hardware or in different logical partitions, and different algorithms may involve different hardware or software components.

The driver interface allows implementations of the Crypto API to be built compositionally. An implementation of the Crypto API is composed of a **core** and zero or more **drivers**. The core handles key management, enforces key usage policies, and dispatches cryptographic operations either to the applicable driver or to built-in code.

Functions in the Crypto API implementation invoke functions in the core. Code from the core calls drivers as described in the present document.

1.2 Types of drivers

The PSA Cryptoprocessor driver interface supports two types of cryptoprocessors, and accordingly two types of drivers.

- **Transparent** drivers implement cryptographic operations on keys that are provided in cleartext at the beginning of each operation. They are typically used for hardware **accelerators**. When a transparent driver is available for a particular combination of parameters (cryptographic algorithm, key type and size, etc.), it is used instead of the default software implementation. Transparent drivers can also be pure software implementations that are distributed as plug-ins to a Crypto API implementation (for example, an alternative implementation with different performance characteristics, or a certified implementation).
- **Opaque** drivers implement cryptographic operations on keys that can only be used inside a protected environment such as a **secure element**, a hardware security module, a smartcard, a secure enclave, etc. An opaque driver is invoked for the specific key location that the driver is registered for: the dispatch is based on the key's lifetime.

1.3 Requirements

The present specification was designed to fulfill the following high-level requirements.

[Req.plugins] It is possible to combine multiple drivers from different providers into the same implementation, without any prior arrangement other than choosing certain names and values from disjoint namespaces.

[Req.compile] It is possible to compile the code of each driver and of the core separately, and link them together. A small amount of glue code may need to be compiled once the list of drivers is available.

[Req.types] Support drivers for the following types of hardware: accelerators that operate on keys in cleartext; cryptoprocessors that can wrap keys with a built-in keys but not store user keys; and cryptoprocessors that store key material.

[Req.portable]
: The interface between drivers and the core does not involve any platform-specific consideration. Driver calls are simple C function calls. Interactions with platform-specific hardware happen only inside the driver (and in fact a driver need not involve any hardware at all).

[Req.location]
: Applications can tell which location values correspond to which secure element drivers.

[Req.fallback]
: Accelerator drivers can specify that they do not fully support a cryptographic mechanism and that a fallback to core code may be necessary. Conversely, if an accelerator fully supports cryptographic mechanism, the core must be able to omit code for this mechanism.

[Req.mechanisms]
: Drivers can specify which mechanisms they support. A driver's code will not be invoked for cryptographic mechanisms that it does not support.

2 Overview of drivers

2.1 Deliverables for a driver

To write a driver, you need to implement some functions with C linkage, and to declare these functions in a **driver description file**. The driver description file declares which functions the driver implements and what cryptographic mechanisms they support. If the driver description references custom types, macros or constants, you also need to provide C header files defining those elements.

The concrete syntax for a driver description file is JSON. The structure of this JSON file is specified in the section *Driver description syntax* on page 10.

A driver therefore consists of:

- A driver description file (in JSON format).
- C header files defining the types required by the driver description. The names of these header files are declared in the driver description file.
- An object file compiled for the target platform defining the entry point functions specified by the driver description. Implementations may allow drivers to be provided as source files and compiled with the core instead of being pre-compiled.

How to provide the driver description file, the C header files and the object code is implementation-dependent.

2.2 Driver description list

Crypto API core implementations should support multiple drivers. The driver description files are passed to the implementation as an ordered list in an unspecified manner. This may be, for example, a list of file names passed on a command line, or a JSON list whose elements are individual driver descriptions.

3 Driver description

3.1 Driver description syntax

The concrete syntax for a driver description file is JSON.

In addition to the properties described here, any JSON object may have a property called `"_comment"` of type string, which will be ignored.

Crypto API core implementations may support additional properties. Such properties must use names consisting of the implementation's name, a slash, and additional characters. For example, the Yoyodyne implementation may use property names such as `"yoyodyne/foo"` and `"yoyodyne/widgets/girth"`.

3.2 Driver description top-level element

A driver description is a JSON object containing the following properties:

- `"prefix"` (mandatory, string). This must be a valid, non-empty prefix for a C identifier. All the types and functions provided by the driver have a name that starts with this prefix unless overridden with a `"name"` element in the applicable capability as described below.
- `"type"` (mandatory, string). One of `"transparent"` or `"opaque"`.
- `"headers"` (optional, array of strings). A list of header files. These header files must define the types, macros and constants referenced by the driver description. They may declare the entry point functions, but this is not required. They may include other PSA headers and standard headers of the platform. Whether they may include other headers is implementation-specific. If omitted, the list of headers is empty. The header files must be present at the specified location relative to a directory on the compiler's include path when compiling glue code between the core and the drivers.
- `"capabilities"` (mandatory, array of capabilities). A list of **capabilities**. Each capability describes a family of functions that the driver implements for a certain class of cryptographic mechanisms.
- `"key_context"` (not permitted for transparent drivers, mandatory for opaque drivers): information about the representation of keys.
- `"persistent_state_size"` (not permitted for transparent drivers, optional for opaque drivers, integer or string). The size in bytes of the persistent state of the driver. This may be either a non-negative integer or a C constant expression of type `size_t`.
- `"location"` (not permitted for transparent drivers, optional for opaque drivers, integer or string). The location value for which this driver is invoked. In other words, this determines the lifetimes for which the driver is invoked. This may be either a non-negative integer or a C constant expression of type `psa_key_location_t`.

3.3 Driver description capability

3.3.1 Capability syntax

A capability declares a family of functions that the driver implements for a certain class of cryptographic mechanisms. The capability specifies which key types and algorithms are covered and the names of the types and functions that implement it.

A capability is a JSON object containing the following properties:

- `"entry_points"` (mandatory, list of strings). Each element is the name of a driver entry point or driver entry point family. An entry point is a function defined by the driver. If specified, the core will invoke this capability of the driver only when performing one of the specified operations. The driver must implement all the specified entry points, as well as the types if applicable.
- `"algorithms"` (optional, list of strings). Each element is an algorithm specification. If specified, the core will invoke this capability of the driver only when performing one of the specified algorithms. If omitted, the core will invoke this capability for all applicable algorithms.
- `"key_types"` (optional, list of strings). Each element is a key type specification. If specified, the core will invoke this capability of the driver only for operations involving a key with one of the specified key types. If omitted, the core will invoke this capability of the driver for all applicable key types.
- `"key_sizes"` (optional, list of integers). If specified, the core will invoke this capability of the driver only for operations involving a key with one of the specified key sizes. If omitted, the core will invoke this capability of the driver for all applicable key sizes. Key sizes are expressed in bits.
- `"names"` (optional, object). A mapping from entry point names described by the `"entry_points"` property, to the name of the C function in the driver that implements the corresponding function. If a function is not listed here, name of the driver function that implements it is the driver's prefix followed by an underscore (_) followed by the function name. If this property is omitted, it is equivalent to an empty object (so each entry point *suffix* is implemented by a function called *prefix_suffix*).
- `"fallback"` (optional for transparent drivers, not permitted for opaque drivers, boolean). If present and true, the driver may return `PSA_ERROR_NOT_SUPPORTED`, in which case the core should call another driver or use built-in code to perform this operation. If absent or false, the driver is expected to fully support the mechanisms described by this capability. See the section *Fallback* on page 38 for more information.

3.3.2 Capability semantics

When the Crypto API implementation performs a cryptographic mechanism, it invokes available driver entry points as described in the section *Driver entry points* on page 13.

A driver is considered available for a cryptographic mechanism that invokes a given entry point if all of the following conditions are met:

- The driver specification includes a capability whose `"entry_points"` list either includes the entry point or includes an entry point family that includes the entry point.
- If the mechanism involves an algorithm:
  - either the capability does not have an `"algorithms"` property;
  - or the value of the capability's `"algorithms"` property includes an algorithm specification that matches this algorithm.
- If the mechanism involves a key:
  - either the key is transparent (its location is `PSA_KEY_LOCATION_LOCAL_STORAGE`) and the driver is transparent;
  - or the key is opaque (its location is not `PSA_KEY_LOCATION_LOCAL_STORAGE`) and the driver is an opaque driver whose location is the key's location.
- If the mechanism involves a key:
  - either the capability does not have a `"key_types"` property;
  - or the value of the capability's `"key_types"` property includes a key type specification that matches this algorithm.

- If the mechanism involves a key:
  - either the capability does not have a `"key_sizes"` property;
  - or the value of the capability's `"key_sizes"` property includes the key's size.

If a driver includes multiple applicable capabilities for a given combination of entry point, algorithm, key type and key size, and all the capabilities map the entry point to the same function name, the driver is considered available for this cryptographic mechanism. If a driver includes multiple applicable capabilities for a given combination of entry point, algorithm, key type and key size, and at least two of these capabilities map the entry point to the different function names, the driver specification is invalid.

If multiple transparent drivers have applicable capabilities for a given combination of entry point, algorithm, key type and key size, the first matching driver in the specification list is invoked. If the capability has fallback enabled and the first driver returns `PSA_ERROR_NOT_SUPPORTED`, the next matching driver is invoked, and so on.

If multiple opaque drivers have the same location, the list of driver specifications is invalid.

3.3.3 Capability examples

Example 1: the following capability declares that the driver can perform deterministic ECDSA signatures (but not signature verification) using any hash algorithm and any curve that the core supports. If the prefix of this driver is `"acme"`, the function that performs the signature is called `acme_sign_hash`.

```
{
    "entry_points": ["sign_hash"],
    "algorithms": ["PSA_ALG_DETERMINISTIC_ECDSA(PSA_ALG_ANY_HASH)"],
}
```

Example 2: the following capability declares that the driver can perform deterministic ECDSA signatures using SHA-256 or SHA-384 with a SECP256R1 or SECP384R1 private key (with either hash being possible in combination with either curve). If the prefix of this driver is `"acme"`, the function that performs the signature is called `acme_sign_hash`.

```
{
    "entry_points": ["sign_hash"],
    "algorithms": ["PSA_ALG_DETERMINISTIC_ECDSA(PSA_ALG_SHA_256)",
                   "PSA_ALG_DETERMINISTIC_ECDSA(PSA_ALG_SHA_384)"],
    "key_types": ["PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_SECP_R1)"],
    "key_sizes": [256, 384]
}
```

3.4 Algorithm and key specifications

3.4.1 Algorithm specifications

An algorithm specification is a string consisting of a `PSA_ALG_xxx` macro that specifies a cryptographic algorithm or an algorithm wildcard policy defined by the Crypto API. If the macro takes arguments, the string must have the syntax of a C macro call and each argument must be an algorithm specification or a decimal or hexadecimal literal with no suffix, depending on the expected type of argument.

Spaces are optional after commas. Whether other whitespace is permitted is implementation-specific.

Valid examples:

```
PSA_ALG_SHA_256
PSA_ALG_HMAC(PSA_ALG_SHA_256)
PSA_ALG_KEY_AGREEMENT(PSA_ALG_ECDH, PSA_ALG_HKDF(PSA_ALG_SHA_256))
PSA_ALG_RSA_PSS(PSA_ALG_ANY_HASH)
```

3.4.2 Key type specifications

An algorithm specification is a string consisting of a `PSA_KEY_TYPE_xxx` macro that specifies a key type defined by the Crypto API. If the macro takes an argument, the string must have the syntax of a C macro call and each argument must be the name of a constant of suitable type (curve or group).

The name _ may be used instead of a curve or group to indicate that the capability concerns all curves or groups.

Valid examples:

```
PSA_KEY_TYPE_AES
PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_SECP_R1)
PSA_KEY_TYPE_ECC_KEY_PAIR(_)
```

4 Driver entry points

4.1 Overview of driver entry points

Drivers define functions, each of which implements an aspect of a capability of a driver, such as a cryptographic operation, a part of a cryptographic operation, or a key management action. These functions are called the **entry points** of the driver. Most driver entry points correspond to a particular function in the Crypto API. For example, if a call to `psa_sign_hash()` is dispatched to a driver, it invokes the driver's `sign_hash` function.

All driver entry points return a status of type `psa_status_t` which should use the status codes documented for PSA services in general and for the Crypto API. In particular: `PSA_SUCCESS` indicates that the function succeeded, and `PSA_ERROR_xxx` values indicate that an error occurred.

The signature of a driver entry point generally looks like the signature of the Crypto API that it implements, with some modifications. This section gives an overview of modifications that apply to whole classes of entry points. Refer to the reference section for each entry point or entry point family for details.

- For entry points that operate on an existing key, the `psa_key_id_t` parameter is replaced by a sequence of three parameters that describe the key:
  1. `const psa_key_attributes_t *attributes`: the key attributes.
  2. `const uint8_t *key_buffer`: a key material or key context buffer.
  3. `size_t key_buffer_size`: the size of the key buffer in bytes.

  For transparent drivers, the key buffer contains the key material, in the same format as defined for `psa_export_key()` and `psa_export_public_key()` in the Crypto API. For opaque drivers, the content of the key buffer is entirely up to the driver.

- For entry points that involve a multi-part operation, the operation state type (`psa_XXX_operation_t`) is replaced by a driver-specific operation state type (*prefix*`_XXX_operation_t`).
- For entry points that are involved in key creation, the `psa_key_id_t *` output parameter is replaced by a sequence of parameters that convey the key context:
    1. `uint8_t *key_buffer`: a buffer for the key material or key context.
    2. `size_t key_buffer_size`: the size of the key buffer in bytes.
    3. `size_t *key_buffer_length`: the length of the data written to the key buffer in bytes.

Some entry points are grouped in families that must be implemented as a whole. If a driver supports an entry point family, it must provide all the entry points in the family.

Drivers can also have entry points related to random generation. A transparent driver can provide a random generation interface. Separately, transparent and opaque drivers can have entropy collection entry points.

4.1.1 General considerations on driver entry point parameters

Buffer parameters for driver entry points obey the following conventions:

- An input buffer has the type `const uint8_t *` and is immediately followed by a parameter of type `size_t` that indicates the buffer size.
- An output buffer has the type `uint8_t *` and is immediately followed by a parameter of type `size_t` that indicates the buffer size. A third parameter of type `size_t *` is provided to report the actual length of the data written in the buffer if the function succeeds.
- An in-out buffer has the type `uint8_t *` and is immediately followed by a parameter of type `size_t` that indicates the buffer size. In-out buffers are only used when the input and the output have the same length.

Buffers of size 0 may be represented with either a null pointer or a non-null pointer.

Input buffers and other input-only parameters (`const` pointers) may be in read-only memory. Overlap is possible between input buffers, and between an input buffer and an output buffer, but not between two output buffers or between a non-buffer parameter and another parameter.

4.2 Driver entry points for single-part cryptographic operations

The following driver entry points perform a cryptographic operation in one shot (single-part operation):

- `"hash_compute"` (transparent drivers only): calculation of a hash. Called by `psa_hash_compute()` and `psa_hash_compare()`. To verify a hash with `psa_hash_compare()`, the core calls the driver's `"hash_compute"` entry point and compares the result with the reference hash value.
- `"mac_compute"`: calculation of a MAC. Called by `psa_mac_compute()` and possibly `psa_mac_verify()`. To verify a mac with `psa_mac_verify()`, the core calls an applicable driver's `"mac_verify"` entry point if there is one, otherwise the core calls an applicable driver's `"mac_compute"` entry point and compares the result with the reference MAC value.
- `"mac_verify"`: verification of a MAC. Called by `psa_mac_verify()`. This entry point is mainly useful for drivers of secure elements that verify a MAC without revealing the correct MAC. Although transparent drivers may implement this entry point in addition to `"mac_compute"`, it is generally not useful because the core can call the `"mac_compute"` entry point and compare with the expected MAC value.

- `"cipher_encrypt"`: unauthenticated symmetric cipher encryption. Called by `psa_cipher_encrypt()`.
- `"cipher_decrypt"`: unauthenticated symmetric cipher decryption. Called by `psa_cipher_decrypt()`.
- `"aead_encrypt"`: authenticated encryption with associated data. Called by `psa_aead_encrypt()`.
- `"aead_decrypt"`: authenticated decryption with associated data. Called by `psa_aead_decrypt()`.
- `"asymmetric_encrypt"`: asymmetric encryption. Called by `psa_asymmetric_encrypt()`.
- `"asymmetric_decrypt"`: asymmetric decryption. Called by `psa_asymmetric_decrypt()`.
- `"sign_hash"`: signature of an already calculated hash. Called by `psa_sign_hash()` and possibly `psa_sign_message()`. To sign a message with `psa_sign_message()`, the core calls an applicable driver's `"sign_message"` entry point if there is one, otherwise the core calls an applicable driver's `"hash_compute"` entry point followed by an applicable driver's `"sign_hash"` entry point.
- `"verify_hash"`: verification of an already calculated hash. Called by `psa_verify_hash()` and possibly `psa_verify_message()`. To verify a message with `psa_verify_message()`, the core calls an applicable driver's `"verify_message"` entry point if there is one, otherwise the core calls an applicable driver's `"hash_compute"` entry point followed by an applicable driver's `"verify_hash"` entry point.
- `"sign_message"`: signature of a message. Called by `psa_sign_message()`.
- `"verify_message"`: verification of a message. Called by `psa_verify_message()`.
- `"key_agreement"`: key agreement without a subsequent key derivation. Called by `psa_raw_key_agreement()` and possibly `psa_key_derivation_key_agreement()`.

4.3 Driver entry points for multi-part operations

4.3.1 General considerations on multi-part operations

The entry points that implement each step of a multi-part operation are grouped into a family. A driver that implements a multi-part operation must define all of the entry points in this family as well as a type that represents the operation context. The lifecycle of a driver operation context is similar to the lifecycle of an API operation context:

1. The core initializes operation context objects to either all-bits-zero or to logical zero (`{0}`), at its discretion.
2. The core calls the `xxx_setup` entry point for this operation family. If this fails, the core destroys the operation context object without calling any other driver entry point on it.
3. The core calls other entry points that manipulate the operation context object, respecting the constraints.
4. If any entry point fails, the core calls the driver's `xxx_abort` entry point for this operation family, then destroys the operation context object without calling any other driver entry point on it.
5. If a "finish" entry point fails, the core destroys the operation context object without calling any other driver entry point on it. The finish entry points are: *prefix*`_mac_sign_finish`, *prefix*`_mac_verify_finish`, *prefix*`_cipher_finish`, *prefix*`_aead_finish`, *prefix*`_aead_verify`.

If a driver implements a multi-part operation but not the corresponding single-part operation, the core calls the driver's multipart operation entry points to perform the single-part operation.

4.3.2 Multi-part operation entry point family `"hash_multipart"`

This family corresponds to the calculation of a hash in multiple steps.

This family applies to transparent drivers only.

This family requires the following type and entry points:

- Type `"hash_operation_t"`: the type of a hash operation context. It must be possible to copy a hash operation context byte by byte, therefore hash operation contexts must not contain any embedded pointers (except pointers to global data that do not change after the setup step).
- `"hash_setup"`: called by `psa_hash_setup()`.
- `"hash_update"`: called by `psa_hash_update()`.
- `"hash_finish"`: called by `psa_hash_finish()` and `psa_hash_verify()`.
- `"hash_abort"`: called by all multi-part hash functions of the Crypto API.

To verify a hash with `psa_hash_verify()`, the core calls the driver's *prefix*`_hash_finish` entry point and compares the result with the reference hash value.

For example, a driver with the prefix `"acme"` that implements the `"hash_multipart"` entry point family must define the following type and entry points (assuming that the capability does not use the `"names"` property to declare different type and entry point names):

```
typedef ... acme_hash_operation_t;
psa_status_t acme_hash_setup(acme_hash_operation_t *operation,
                             psa_algorithm_t alg);
psa_status_t acme_hash_update(acme_hash_operation_t *operation,
                              const uint8_t *input,
                              size_t input_length);
psa_status_t acme_hash_finish(acme_hash_operation_t *operation,
                              uint8_t *hash,
                              size_t hash_size,
                              size_t *hash_length);
psa_status_t acme_hash_abort(acme_hash_operation_t *operation);
```

4.3.3 Operation family `"mac_multipart"`

TODO

4.3.4 Operation family `"mac_verify_multipart"`

TODO

4.3.5 Operation family `"cipher_encrypt_multipart"`

TODO

4.3.6 Operation family `"cipher_decrypt_multipart"`

TODO

4.3.7 Operation family `"aead_encrypt_multipart"`

TODO

4.3.8 Operation family `"aead_decrypt_multipart"`

TODO

4.4 Driver entry points for key derivation

Key derivation is more complex than other multipart operations for several reasons:

- There are multiple inputs and outputs.
- Multiple drivers can be involved. This happens when an operation combines a key agreement and a subsequent symmetric key derivation, each of which can have independent drivers. This also happens when deriving an asymmetric key, where processing the secret input and generating the key output might involve different drivers.
- When multiple drivers are involved, they are not always independent: if the secret input is managed by an opaque driver, it might not allow the core to retrieve the intermediate output and pass it to another driver.
- The involvement of an opaque driver cannot be determined as soon as the operation is set up (since `psa_key_derivation_setup()` does not determine the key input).

4.4.1 Key derivation driver dispatch logic

The core decides whether to dispatch a key derivation operation to a driver based on the location associated with the input step `PSA_KEY_DERIVATION_INPUT_SECRET`.

1. If this step is passed via `psa_key_derivation_input_key()` for a key in a secure element:
    - If the driver for this secure element implements the `"key_derivation"` family for the specified algorithm, the core calls that driver's `"key_derivation_setup"` and subsequent entry points. Note that for all currently specified algorithms, the key type for the secret input does not matter.
    - Otherwise the core calls the secure element driver's "export_key" entry point.
2. Otherwise (or on fallback?), if there is a transparent driver for the specified algorithm, the core calls that driver's `"key_derivation_setup"` and subsequent entry points.
3. Otherwise, or on fallback, the core uses its built-in implementation.

4.4.2 Summary of entry points for the operation family `"key_derivation"`

A key derivation driver has the following entry points:

- `"key_derivation_setup"` (mandatory): always the first entry point to be called. This entry point provides the initial inputs. See *Key derivation driver setup* on page 20.
- `"key_derivation_input_step"` (mandatory if the driver supports a key derivation algorithm with long inputs, otherwise ignored): provide an extra input for the key derivation. This entry point is only mandatory in drivers that support algorithms that have extra inputs. See *Key derivation driver long inputs* on page 21.
- `"key_derivation_output_bytes"` (mandatory): derive cryptographic material and output it. See *Key derivation driver outputs* on page 21.
- `"key_derivation_output_key"`, `"key_derivation_verify_bytes"`, `"key_derivation_verify_key"` (optional, opaque drivers only): derive key material which remains inside the same secure element. See *Key derivation driver outputs* on page 21.
- `"key_derivation_set_capacity"` (mandatory for opaque drivers that implement `"key_derivation_output_key"` for "cooked", i.e. non-raw-data key types; ignored for other opaque drivers; not permitted for transparent drivers): update the capacity policy on the operation. See *Key derivation driver operation capacity* on page 21.
- `"key_derivation_abort"` (mandatory): always the last entry point to be called.

For naming purposes, here and in the following subsection, this specification takes the example of a driver with the prefix `"acme"` that implements the `"key_derivation"` entry point family with a capability that does not use the `"names"` property to declare different type and entry point names. Such a driver must implement the following type and functions, as well as the entry points listed above and described in the following subsections:

```
typedef ... acme_key_derivation_operation_t;
psa_status_t acme_key_derivation_abort(acme_key_derivation_operation_t *operation);
```

4.4.3 Key derivation driver initial inputs

The core conveys the initial inputs for a key derivation via an opaque data structure of type `psa_crypto_driver_key_derivation_inputs_t`.

```
typedef ... psa_crypto_driver_key_derivation_inputs_t; // implementation-specific type
```

A driver receiving an argument that points to a `psa_crypto_driver_key_derivation_inputs_t` can retrieve its contents by calling one of the type-specific functions below. To determine the correct function, the driver can call `psa_crypto_driver_key_derivation_get_input_type()`.

```
enum psa_crypto_driver_key_derivation_input_type_t {
    PSA_KEY_DERIVATION_INPUT_TYPE_INVALID = 0,
    PSA_KEY_DERIVATION_INPUT_TYPE_OMITTED,
    PSA_KEY_DERIVATION_INPUT_TYPE_BYTES,
    PSA_KEY_DERIVATION_INPUT_TYPE_KEY,
    PSA_KEY_DERIVATION_INPUT_TYPE_INTEGER,
    // Implementations may add other values, and may freely choose the
    // numerical values for each identifer except as explicitly specified
    // above.
};
```

(continues on next page)

(continued from previous page)

```
psa_crypto_driver_key_derivation_input_type_t psa_crypto_driver_key_derivation_get_input_type(
    const psa_crypto_driver_key_derivation_inputs_t *inputs,
    psa_key_derivation_step_t step);
```

The function `psa_crypto_driver_key_derivation_get_input_type()` determines whether a given step is present and how to access its value:

- `PSA_KEY_DERIVATION_INPUT_TYPE_INVALID`: the step is invalid for the algorithm of the operation that the inputs are for.
- `PSA_KEY_DERIVATION_INPUT_TYPE_OMITTED`: the step is optional for the algorithm of the operation that the inputs are for, and has been omitted.
- `PSA_KEY_DERIVATION_INPUT_TYPE_BYTES`: the step is valid and present and is a transparent byte string. Call `psa_crypto_driver_key_derivation_get_input_size()` to obtain the size of the input data. Call `psa_crypto_driver_key_derivation_get_input_bytes()` to make a copy of the input data (design note: why a copy?).
- `PSA_KEY_DERIVATION_INPUT_TYPE_KEY`: the step is valid and present and is a byte string passed via a key object. Call `psa_crypto_driver_key_derivation_get_input_key()` to obtain a pointer to the key context.
- `PSA_KEY_DERIVATION_INPUT_TYPE_INTEGER`: the step is valid and present and is an integer. Call `psa_crypto_driver_key_derivation_get_input_integer()` to retrieve the integer value.

```
psa_status_t psa_crypto_driver_key_derivation_get_input_size(
    const psa_crypto_driver_key_derivation_inputs_t *inputs,
    psa_key_derivation_step_t step,
    size_t *size);
psa_status_t psa_crypto_driver_key_derivation_get_input_bytes(
    const psa_crypto_driver_key_derivation_inputs_t *inputs,
    psa_key_derivation_step_t step,
    uint8_t *buffer, size_t buffer_size, size_t *buffer_length);
psa_status_t psa_crypto_driver_key_derivation_get_input_key(
    const psa_crypto_driver_key_derivation_inputs_t *inputs,
    psa_key_derivation_step_t step,
    const psa_key_attributes_t *attributes,
    uint8_t** p_key_buffer, size_t *key_buffer_size);
psa_status_t psa_crypto_driver_key_derivation_get_input_integer(
    const psa_crypto_driver_key_derivation_inputs_t *inputs,
    psa_key_derivation_step_t step,
    uint64_t *value);
```

The get-data functions take the following parameters:

- The first parameter `inputs` must be a pointer passed by the core to a key derivation driver setup entry point which has not returned yet.
- The `step` parameter indicates the input step whose content the driver wants to retrieve.
- On a successful invocation of `psa_crypto_driver_key_derivation_get_input_size`, the core sets `*size` to the size of the specified input in bytes.
- On a successful invocation of `psa_crypto_driver_key_derivation_get_input_bytes`, the core fills the first *N* bytes of `buffer` with the specified input and sets `*buffer_length` to *N*, where *N* is the length of

the input in bytes. The value of `buffer_size` must be at least *N*, otherwise this function fails with the status `PSA_ERROR_BUFFER_TOO_SMALL`.

- On a successful invocation of `psa_crypto_driver_key_derivation_get_input_key`, the core sets `*key_buffer` to a pointer to a buffer containing the key context and `*key_buffer_size` to the size of the key context in bytes. The key context buffer remains valid for the duration of the driver entry point. If the driver needs to access the key context after the current entry point returns, it must make a copy of the key context.
- On a successful invocation of `psa_crypto_driver_key_derivation_get_input_integer`, the core sets `*value` to the value of the specified input.

These functions can return the following statuses:

- `PSA_SUCCESS`: the call succeeded and the requested value has been copied to the output parameter (`size`, `buffer`, `value` or `p_key_buffer`) and if applicable the size of the value has been written to the applicable parameter (`buffer_length`, `key_buffer_size`).
- `PSA_ERROR_DOES_NOT_EXIST`: the input step is valid for this particular algorithm, but it is not part of the initial inputs. This is not a fatal error. The driver will receive the input later as a long input.
- `PSA_ERROR_INVALID_ARGUMENT`: the input type is not compatible with this function or was omitted. Call `psa_crypto_driver_key_derivation_get_input_type()` to find out the actual type of this input step. This is not a fatal error and the driver can, for example, subsequently call the appropriate function on the same step.
- `PSA_ERROR_BUFFER_TOO_SMALL` (`psa_crypto_driver_key_derivation_get_input_bytes` only): the output buffer is too small. This is not a fatal error and the driver can, for example, subsequently call the same function again with a larger buffer. Call `psa_crypto_driver_key_derivation_get_input_size` to obtain the required size.
- The core may return other errors such as `PSA_ERROR_CORRUPTION_DETECTED` or `PSA_ERROR_COMMUNICATION_FAILURE` to convey implementation-specific error conditions. Portable drivers should treat such conditions as fatal errors.

4.4.4 Key derivation driver setup

A key derivation driver must implement the following entry point:

```
psa_status_t acme_key_derivation_setup(
    acme_key_derivation_operation_t *operation,
    psa_algorithm_t alg,
    const psa_crypto_driver_key_derivation_inputs_t *inputs);
```

- `operation` is a zero-initialized operation object.
- `alg` is the algorithm for the key derivation operation. It does not include a key agreement component.
- `inputs` is an opaque pointer to the initial inputs for the key derivation.

4.4.5 Key derivation driver long inputs

Some key derivation algorithms take long inputs which it would not be practical to pass in the initial inputs. A driver that implements a key derivation algorithm that takes such inputs must provide a `"key_derivation_input_step"` entry point. The core calls this entry point for all the long inputs after calling `"acme_key_derivation_setup"`. A long input step may be fragmented into multiple calls of `psa_key_derivation_input_bytes()`, and the core may reassemble or refragment those fragments before passing them to the driver. Calls to this entry point for different step values occur in an unspecified order and may be interspersed.

```
psa_status_t acme_key_derivation_input_step(
    acme_key_derivation_operation_t *operation,
    psa_key_derivation_step_t step,
    const uint8_t *input, size_t input_length);
```

At the time of writing, no standard key derivation algorithm has long inputs. It is likely that such algorithms will be added in the future.

4.4.6 Key derivation driver operation capacity

The core keeps track of an operation's capacity and enforces it. The core guarantees that it will not request output beyond the capacity of the operation, with one exception: opaque drivers that support "key_derivation_output_key", i.e. for key types where the derived key material is not a direct copy of the key derivation's output stream.

Such drivers must enforce the capacity limitation and must return `PSA_ERROR_INSUFFICIENT_CAPACITY` from any output request that exceeds the operation's capacity. Such drivers must provide the following entry point:

```
psa_status_t acme_key_derivation_set_capacity(
    acme_key_derivation_operation_t *operation,
    size_t capacity);
```

`capacity` is guaranteed to be less or equal to any value previously set through this entry point, and is guaranteed not to be `PSA_KEY_DERIVATION_UNLIMITED_CAPACITY`.

If this entry point has not been called, the operation has an unlimited capacity.

4.4.7 Key derivation driver outputs

A key derivation driver must provide the following entry point:

```
psa_status_t acme_key_derivation_output_bytes(
    acme_key_derivation_operation_t *operation,
    uint8_t *output, size_t length);
```

An opaque key derivation driver may provide the following entry points:

```
psa_status_t acme_key_derivation_output_key(
    const psa_key_attributes_t *attributes,
    acme_key_derivation_operation_t *operation,
```

(continues on next page)

(continued from previous page)

```
    uint8_t *key_buffer, size_t key_buffer_size, size_t *key_buffer_length);
psa_status_t acme_key_derivation_verify_bytes(
    acme_key_derivation_operation_t *operation,
    const uint8_t *expected output, size_t length);
psa_status_t acme_key_derivation_verify_key(
    acme_key_derivation_operation_t *operation,
    uint8_t *key_buffer, size_t key_buffer_size);
```

The core calls a key derivation driver's output entry point when the application calls `psa_key_derivation_output_bytes()`, `psa_key_derivation_output_key()`, `psa_key_derivation_verify_bytes()` or `psa_key_derivation_verify_key()`.

If the key derivation's `PSA_KEY_DERIVATION_INPUT_SECRET` input is in a secure element and the derivation operation is handled by that secure element, the core performs the following steps:

- For a call to `psa_key_derivation_output_key()`:
    1. If the derived key is in the same secure element, if the driver has an `"key_derivation_output_key"` entry point, call that entry point. If the driver has no such entry point, or if that entry point returns `PSA_ERROR_NOT_SUPPORTED`, continue with the following steps, otherwise stop.
    2. If the driver's capabilities indicate that its `"import_key"` entry point does not support the derived key, stop and return `PSA_ERROR_NOT_SUPPORTED`.
    3. Otherwise proceed as for `psa_key_derivation_output_bytes()`, then import the resulting key material.
- For a call to `psa_key_derivation_verify_key()`:
    1. If the driver has a `"key_derivation_verify_key"` entry point, call it and stop.
    2. Call the driver's `"export_key"` entry point on the key object that contains the expected value, then proceed as for `psa_key_derivation_verify_bytes()`.
- For a call to `psa_key_derivation_verify_bytes()`:
    1. If the driver has a `"key_derivation_verify_bytes"` entry point, call that entry point on the expected output, then stop.
    2. Otherwise, proceed as for `psa_key_derivation_output_bytes()`, and compare the resulting output to the expected output inside the core.
- For a call to `psa_key_derivation_output_bytes()`:
    1. Call the `"key_derivation_output_bytes"` entry point. The core may call this entry point multiple times to implement a single call from the application when deriving a cooked (non-raw) key as described below, or if the output size exceeds some implementation limit.

If the key derivation operation is not handled by an opaque driver as described above, the core calls the `"key_derivation_output_bytes"` from the applicable transparent driver (or multiple drivers in succession if fallback applies). In some cases, the core then calls additional entry points in the same or another driver:

- For a call to `psa_key_derivation_output_key()` for some key types, the core calls a transparent driver's `"derive_key"` entry point. See *Transparent cooked key derivation* on page 23.
- For a call to `psa_key_derivation_output_key()` where the derived key is in a secure element, call that secure element driver's `"import_key"` entry point.

4.4.8 Transparent cooked key derivation

Key derivation is said to be *raw* for some key types, where the key material of a derived (8*n*)-bit key consists of the next *n* bytes of output from the key derivation, and *cooked* otherwise. When deriving a raw key, the core only calls the driver's `"output_bytes"` entry point, except when deriving a key entirely inside a secure element as described in *Key derivation driver outputs* on page 21. When deriving a cooked key, the core calls a transparent driver's `"derive_key"` entry point if available.

A capability for cooked key derivation contains the following properties (this is not a subset of the usual entry point properties):

- `"entry_points"` (mandatory, list of strings). Must be `["derive_key"]`.
- `"derived_types"` (mandatory, list of strings). Each element is a key type specification. This capability only applies when deriving a key of the specified type.
- `"derived_sizes"` (optional, list of integers). Each element is a size for the derived key, in bits. This capability only applies when deriving a key of the specified sizes. If absent, this capability applies to all sizes for the specified types.
- `"memory"` (optional, boolean). If present and true, the driver must define a type `"derive_key_memory_t"` and the core will allocate an object of that type as specified below.
- `"names"` (optional, object). A mapping from entry point names to C function and type names, as usual.
- `"fallback"` (optional, boolean). If present and true, the driver may return `PSA_ERROR_NOT_SUPPORTED` if it only partially supports the specified mechanism, as usual.

A transparent driver with the prefix `"acme"` that implements cooked key derivation must provide the following type and function:

```
typedef ... acme_derive_key_memory_t; // only if the "memory" property is true
psa_status_t acme_derive_key(
    const psa_key_attributes_t *attributes,
    const uint8_t *input, size_t input_length,
    acme_derive_key_memory_t *memory, // if the "memory" property is false: void*
    uint8_t *key_buffer, size_t key_buffer_size, size_t *key_buffer_length);
```

- `attributes` contains the attributes of the specified key. Note that only the key type and the bit-size are guaranteed to be set.
- `input` is a buffer of `input_length` bytes which contains the raw key stream, i.e. the data that `psa_key_derivation_output_bytes()` would return.
- If `"memory"` property in the driver capability is true, `memory` is a data structure that the driver may use to store data between successive calls of the `"derive_key"` entry point to derive the same key. If the `"memory"` property is false or absent, the `memory` parameter is a null pointer.
- `key_buffer` is a buffer for the output material, in the appropriate export format for the key type. Its size is `key_buffer_size` bytes.
- On success, `*key_buffer_length` must contain the number of bytes written to `key_buffer`.

This entry point may return the following statuses:

- `PSA_SUCCESS`: a key was derived successfully. The driver has placed the representation of the key in `key_buffer`.

- `PSA_ERROR_NOT_SUPPORTED` (for the first call only) (only if fallback is enabled): the driver cannot fulfill this request, but a fallback driver might.
- `PSA_ERROR_INSUFFICIENT_DATA`: the core must call the `"derive_key"` entry point again with the same `memory` object and with subsequent data from the key stream.
- Any other error is a fatal error.

The core calls the `"derive_key"` entry point in a loop until it returns a status other than `PSA_ERROR_INSUFFICIENT_DATA`. Each call has a successive fragment of the key stream. The `memory` object is guaranteed to be the same for successive calls, but note that its address may change between calls. Before the first call, `*memory` is initialized to all-bits-zero.

For standard key types, the `"derive_key"` entry point is called with a certain input length as follows:

- `PSA_KEY_TYPE_DES`: the length of the key.
- `PSA_KEY_TYPE_ECC_KEY_PAIR(...)`, `PSA_KEY_TYPE_DH_KEY_PAIR(...)`: m bytes, where the bit-size of the key n satisfies $8(m-1) < n <= 8m$.
- `PSA_KEY_TYPE_RSA_KEY_PAIR`: an implementation-defined length. A future version of this specification may specify a length.
- Other key types: not applicable.

See *Open questions around cooked key derivation* on page 49 for some points that may not be fully settled.

4.4.9 Key agreement

The core always decouples key agreement from symmetric key derivation.

To implement a call to `psa_key_derivation_key_agreement()` where the private key is in a secure element that has a `"key_agreement_to_key"` entry point which is applicable for the given key type and algorithm, the core calls the secure element driver as follows:

1. Call the `"key_agreement_to_key"` entry point to create a key object containing the shared secret. The key object is volatile and has the type `PSA_KEY_TYPE_DERIVE`.
2. Call the `"key_derivation_setup"` entry point, passing the resulting key object .
3. Perform the rest of the key derivation, up to and including the call to the `"key_derivation_abort"` entry point.
4. Call the `"destroy_key"` entry point to destroy the key containing the key object.

In other cases, the core treats `psa_key_derivation_key_agreement()` as if it was a call to `psa_raw_key_agreement()` followed by a call to `psa_key_derivation_input_bytes()` on the shared secret.

The entry points related to key agreement have the following prototypes for a driver with the prefix `"acme"`:

```
psa_status_t acme_key_agreement(psa_algorithm_t alg,
                                const psa_key_attributes_t *our_attributes,
                                const uint8_t *our_key_buffer,
                                size_t our_key_buffer_length,
                                const uint8_t *peer_key,
                                size_t peer_key_length,
```

(continues on next page)

(continued from previous page)

```
                                uint8_t *output,
                                size_t output_size,
                                size_t *output_length);
psa_status_t acme_key_agreement_to_key(psa_algorithm_t alg,
                                       const psa_key_attributes_t *our_attributes,
                                       const uint8_t *our_key_buffer,
                                       size_t our_key_buffer_length,
                                       const uint8_t *peer_key,
                                       size_t peer_key_length,
                                       const psa_key_attributes_t *shared_secret_attributes,
                                       uint8_t *shared_secret_key_buffer,
                                       size_t shared_secret_key_buffer_size,
                                       size_t *shared_secret_key_buffer_length);
```

Note that unlike most other key creation entry points, in `"acme_key_agreement_to_key"`, the attributes for the shared secret are not placed near the beginning, but rather grouped with the other parameters related to the shared secret at the end of the parameter list. This is to avoid potential confusion with the attributes of the private key that is passed as an input.

4.5 Driver entry points for PAKE

A PAKE operation is divided into two stages: collecting inputs and computation. Core side is responsible for keeping inputs and core set-data functions do not have driver entry points. Collected inputs are available for drivers via get-data functions for `password`, `role` and `cipher_suite`.

4.5.1 PAKE driver dispatch logic

The core decides whether to dispatch a PAKE operation to a driver based on the location of the provided password. When all inputs are collected and `"psa_pake_output"` or `"psa_pake_input"` is called for the first time `"pake_setup"` driver entry point is invoked.

1. If the location of the `password` is the local storage
   - if there is a transparent driver for the specified ciphersuite, the core calls that driver's `"pake_setup"` and subsequent entry points.
   - otherwise, or on fallback, the core uses its built-in implementation.
2. If the location of the `password` is the location of a secure element - the core calls the `"pake_setup"` entry point of the secure element driver and subsequent entry points.

4.5.2 Summary of entry points for PAKE

A PAKE driver has the following entry points:

- `"pake_setup"` (mandatory): always the first entry point to be called. It is called when all inputs are collected and the computation stage starts.
- `"pake_output"` (mandatory): derive cryptographic material for the specified step and output it.

- `"pake_input"` (mandatory): provides cryptographic material in the format appropriate for the specified step.
- `"pake_get_implicit_key"` (mandatory): returns implicitly confirmed shared secret from a PAKE.
- `"pake_abort"` (mandatory): always the last entry point to be called.

For naming purposes, here and in the following subsection, this specification takes the example of a driver with the prefix `"acme"` that implements the PAKE entry point family with a capability that does not use the `"names"` property to declare different type and entry point names. Such a driver must implement the following type and functions, as well as the entry points listed above and described in the following subsections:

```
typedef ... acme_pake_operation_t;
psa_status_t acme_pake_abort( acme_pake_operation_t *operation );
```

4.5.3 PAKE driver inputs

The core conveys the initial inputs for a PAKE operation via an opaque data structure of type `psa_crypto_driver_pake_inputs_t`.

```
typedef ... psa_crypto_driver_pake_inputs_t; // implementation-specific type
```

A driver receiving an argument that points to a `psa_crypto_driver_pake_inputs_t` can retrieve its contents by calling one of the get-data functions below.

```
psa_status_t psa_crypto_driver_pake_get_password_len(
    const psa_crypto_driver_pake_inputs_t *inputs,
    size_t *password_len);

psa_status_t psa_crypto_driver_pake_get_password_bytes(
    const psa_crypto_driver_pake_inputs_t *inputs,
    uint8_t *buffer, size_t buffer_size, size_t *buffer_length);

psa_status_t psa_crypto_driver_pake_get_password_key(
    const psa_crypto_driver_pake_inputs_t *inputs,
    uint8_t** p_key_buffer, size_t *key_buffer_size,
    const psa_key_attributes_t *attributes);

psa_status_t psa_crypto_driver_pake_get_user_len(
    const psa_crypto_driver_pake_inputs_t *inputs,
    size_t *user_len);

psa_status_t psa_crypto_driver_pake_get_user(
    const psa_crypto_driver_pake_inputs_t *inputs,
    uint8_t *user_id, size_t user_id_size, size_t *user_id_len);

psa_status_t psa_crypto_driver_pake_get_peer_len(
    const psa_crypto_driver_pake_inputs_t *inputs,
    size_t *peer_len);
```

(continues on next page)

(continued from previous page)

```
psa_status_t psa_crypto_driver_pake_get_peer(
    const psa_crypto_driver_pake_inputs_t *inputs,
    uint8_t *peer_id, size_t peer_id_size, size_t *peer_id_length);

psa_status_t psa_crypto_driver_pake_get_cipher_suite(
    const psa_crypto_driver_pake_inputs_t *inputs,
    psa_pake_cipher_suite_t *cipher_suite);
```

The get-data functions take the following parameters:

The first parameter `inputs` must be a pointer passed by the core to a PAKE driver setup entry point. Next parameters are return buffers (must not be null pointers).

These functions can return the following statuses:

- `PSA_SUCCESS`: value has been successfully obtained
- `PSA_ERROR_BAD_STATE`: the inputs are not ready
- `PSA_ERROR_BUFFER_TOO_SMALL` (`psa_crypto_driver_pake_get_password_bytes` and `psa_crypto_driver_pake_get_password_key` only): the output buffer is too small. This is not a fatal error and the driver can, for example, subsequently call the same function again with a larger buffer. Call `psa_crypto_driver_pake_get_password_len` to obtain the required size.

4.5.4 PAKE driver setup

```
psa_status_t acme_pake_setup( acme_pake_operation_t *operation,
                              const psa_crypto_driver_pake_inputs_t *inputs );
```

- `operation` is a zero-initialized operation object.
- `inputs` is an opaque pointer to the inputs for the PAKE operation.

The setup driver function should preserve the inputs using get-data functions.

The pointer output by `psa_crypto_driver_pake_get_password_key` is only valid until the “pake_setup” entry point returns. Opaque drivers must copy all relevant data from the key buffer during the “pake_setup” entry point and must not store the pointer itself.

4.5.5 PAKE driver output

```
psa_status_t acme_pake_output(acme_pake_operation_t *operation,
                              psa_crypto_driver_pake_step_t step,
                              uint8_t *output,
                              size_t output_size,
                              size_t *output_length);
```

- `operation` is an operation object.
- `step` computation step based on which driver should perform an action.

- `output` buffer where the output is to be written.
- `output_size` size of the output buffer in bytes.
- `output_length` the number of bytes of the returned output.

For `PSA_ALG_JPAKE` the following steps are available for output operation: `step` can be one of the following values:

- `PSA_JPAKE_X1_STEP_KEY_SHARE` Round 1: output our key share (for ephemeral private key X1)
- `PSA_JPAKE_X1_STEP_ZK_PUBLIC` Round 1: output Schnorr NIZKP public key for the X1 key
- `PSA_JPAKE_X1_STEP_ZK_PROOF` Round 1: output Schnorr NIZKP proof for the X1 key
- `PSA_JPAKE_X2_STEP_KEY_SHARE` Round 1: output our key share (for ephemeral private key X2)
- `PSA_JPAKE_X2_STEP_ZK_PUBLIC` Round 1: output Schnorr NIZKP public key for the X2 key
- `PSA_JPAKE_X2_STEP_ZK_PROOF` Round 1: output Schnorr NIZKP proof for the X2 key
- `PSA_JPAKE_X2S_STEP_KEY_SHARE` Round 2: output our X2S key
- `PSA_JPAKE_X2S_STEP_ZK_PUBLIC` Round 2: output Schnorr NIZKP public key for the X2S key
- `PSA_JPAKE_X2S_STEP_ZK_PROOF` Round 2: output Schnorr NIZKP proof for the X2S key

4.5.6 PAKE driver input

```
psa_status_t acme_pake_input(acme_pake_operation_t *operation,
                             psa_crypto_driver_pake_step_t step,
                             uint8_t *input,
                             size_t input_size);
```

- `operation` is an operation object.
- `step` computation step based on which driver should perform an action.
- `input` buffer containing the input.
- `input_length` length of the input in bytes.

For `PSA_ALG_JPAKE` the following steps are available for input operation:

- `PSA_JPAKE_X1_STEP_KEY_SHARE` Round 1: input key share from peer (for ephemeral private key X1)
- `PSA_JPAKE_X1_STEP_ZK_PUBLIC` Round 1: input Schnorr NIZKP public key for the X1 key
- `PSA_JPAKE_X1_STEP_ZK_PROOF` Round 1: input Schnorr NIZKP proof for the X1 key
- `PSA_JPAKE_X2_STEP_KEY_SHARE` Round 1: input key share from peer (for ephemeral private key X2)
- `PSA_JPAKE_X2_STEP_ZK_PUBLIC` Round 1: input Schnorr NIZKP public key for the X2 key
- `PSA_JPAKE_X2_STEP_ZK_PROOF` Round 1: input Schnorr NIZKP proof for the X2 key
- `PSA_JPAKE_X4S_STEP_KEY_SHARE` Round 2: input X4S key from peer
- `PSA_JPAKE_X4S_STEP_ZK_PUBLIC` Round 2: input Schnorr NIZKP public key for the X4S key
- `PSA_JPAKE_X4S_STEP_ZK_PROOF` Round 2: input Schnorr NIZKP proof for the X4S key

The core checks that `input_length` is not greater than `PSA_PAKE_INPUT_SIZE(alg, prim, step)` and the driver can rely on that.

4.5.7 PAKE driver get implicit key

```
psa_status_t acme_pake_get_implicit_key(
                            acme_pake_operation_t *operation,
                            uint8_t *output, size_t output_size,
                            size_t *output_length );
```

- `operation` The driver PAKE operation object to use.
- `output` Buffer where the implicit key is to be written.
- `output_size` Size of the output buffer in bytes.
- `output_length` On success, the number of bytes of the implicit key.

4.6 Driver entry points for key management

The driver entry points for key management differ significantly between transparent drivers and opaque drivers. This section describes common elements. Refer to the applicable section for each driver type for more information.

The entry points that create or format key data have the following prototypes for a driver with the prefix `"acme"`:

```
psa_status_t acme_import_key(const psa_key_attributes_t *attributes,
                             const uint8_t *data,
                             size_t data_length,
                             uint8_t *key_buffer,
                             size_t key_buffer_size,
                             size_t *key_buffer_length,
                             size_t *bits); // additional parameter, see below
psa_status_t acme_generate_key(const psa_key_attributes_t *attributes,
                               uint8_t *key_buffer,
                               size_t key_buffer_size,
                               size_t *key_buffer_length);
```

Additionally, opaque drivers can create keys through their "key_derivation_output_key" and "key_agreement_key" entry points. Transparent drivers can create key material through their "derive_key" entry point.

TODO: copy

- The key attributes (`attributes`) have the same semantics as in the Crypto API.
- For the `"import_key"` entry point, the input in the `data` buffer is either the export format or an implementation-specific format that the core documents as an acceptable input format for `psa_import_key()`.
- The size of the key data buffer `key_buffer` is sufficient for the internal representation of the key. For a transparent driver, this is the key's export format. For an opaque driver, this is the size determined from the driver description and the key attributes, as specified in the section *Key format for opaque drivers* on page 38.

- For an opaque driver with an `"allocate_key"` entry point, the content of the key data buffer on entry is the output of that entry point.
- The `"import_key"` entry point must determine or validate the key size and set `*bits` as described in *Key size determination on import*.

All key creation entry points must ensure that the resulting key is valid as specified in *Key validation*. This is primarily important for import entry points since the key data comes from the application.

4.6.1 Key size determination on import

The `"import_key"` entry point must determine or validate the key size. The Crypto API exposes the key size as part of the key attributes. When importing a key, the key size recorded in the key attributes can be either a size specified by the caller of the API (who may not be trusted), or `0` which indicates that the size must be calculated from the data.

When the core calls the `"import_key"` entry point to process a call to `psa_import_key`, it passes an `attributes` structure such that `psa_get_key_bits(attributes)` is the size passed by the caller of `psa_import_key`. If this size is `0`, the `"import_key"` entry point must set the `bits` input-output parameter to the correct key size. The semantics of `bits` is as follows:

- The core sets `*bits` to `psa_get_key_bits(attributes)` before calling the `"import_key"` entry point.
- If `*bits == 0`, the driver must determine the key size from the data and set `*bits` to this size. If the key size cannot be determined from the data, the driver must return `PSA_ERROR_INVALID_ARGUMENT` (as of version 1.0 of the Crypto API specification, it is possible to determine the key size for all standard key types).
- If `*bits != 0`, the driver must check the value of `*bits` against the data and return `PSA_ERROR_INVALID_ARGUMENT` if it does not match. If the driver entry point changes `*bits` to a different value but returns `PSA_SUCCESS`, the core will consider the key as invalid and the import will fail.

4.6.2 Key validation

Key creation entry points must produce valid key data. Key data is *valid* if operations involving the key are guaranteed to work functionally and not to cause indirect security loss. Operation functions are supposed to receive valid keys, and should not have to check and report invalid keys. For example:

- If a cryptographic mechanism is defined as having keying material of a certain size, or if the keying material involves integers that have to be in a certain range, key creation must ensure that the keying material has an appropriate size and falls within an appropriate range.
- If a cryptographic operation involves a division by an integer which is provided as part of a key, key creation must ensure that this integer is nonzero.
- If a cryptographic operation involves two keys A and B (or more), then the creation of A must ensure that using it does not risk compromising B. This applies even if A's policy does not explicitly allow a problematic operation, but A is exportable. In particular, public keys that can potentially be used for key agreement are considered invalid and must not be created if they risk compromising the private key.
- On the other hand, it is acceptable for import to accept a key that cannot be verified as valid if using this key would at most compromise the key itself and material that is secured with this key. For example, RSA key import does not need to verify that the primes are actually prime. Key import may

accept an insecure key if the consequences of the insecurity are no worse than a leak of the key prior to its import.

With opaque drivers, the key context can only be used by code from the same driver, so key validity is primarily intended to report key creation errors at creation time rather than during an operation. With transparent drivers, the key context can potentially be used by code from a different provider, so key validity is critical for interoperability.

This section describes some minimal validity requirements for standard key types.

- For symmetric key types, check that the key size is suitable for the type.
- For DES (`PSA_KEY_TYPE_DES`), additionally verify the parity bits.
- For RSA (`PSA_KEY_TYPE_RSA_PUBLIC_KEY`, `PSA_KEY_TYPE_RSA_KEY_PAIR`), check the syntax of the key and make sanity checks on its components. TODO: what sanity checks? Value ranges (e.g. p < n), sanity checks such as parity, minimum and maximum size, what else?
- For elliptic curve private keys (`PSA_KEY_TYPE_ECC_KEY_PAIR`), check the size and range. TODO: what else?
- For elliptic curve public keys (`PSA_KEY_TYPE_ECC_PUBLIC_KEY`), check the size and range, and that the point is on the curve. TODO: what else?

4.7 Entropy collection entry point

A driver can declare an entropy source by providing a `"get_entropy"` entry point. This entry point has the following prototype for a driver with the prefix `"acme"`:

```
typedef uint32_t psa_driver_get_entropy_flags_t;

psa_status_t acme_get_entropy(psa_driver_get_entropy_flags_t flags,
                              size_t *estimate_bits,
                              uint8_t *output,
                              size_t output_size);
```

The semantics of the parameters is as follows:

- `flags`: a bit-mask of entropy collection flags.
- `estimate_bits`: on success, an estimate of the amount of entropy that is present in the `output` buffer, in bits. This must be at least `1` on success. The value is ignored on failure. Drivers should return a conservative estimate, even in circumstances where the quality of the entropy source is degraded due to environmental conditions (e.g. undervolting, low temperature, etc.).
- `output`: on success, this buffer contains non-deterministic data with an estimated entropy of at least `*estimate_bits` bits. When the entropy is coming from a hardware peripheral, this should preferably be raw or lightly conditioned measurements from a physical process, such that statistical tests run over a sufficiently large amount of output can confirm the entropy estimates. But this specification also permits entropy sources that are fully conditioned, for example when the Crypto API implementation is running within an application in an operating system and `"get_entropy"` returns data from the random generator in the operating system's kernel.
- `output_size`: the size of the `output` buffer in bytes. This size should be large enough to allow a driver to pass unconditioned data with a low density of entropy; for example a peripheral that returns eight

bytes of data with an estimated one bit of entropy cannot provide meaningful output in less than 8 bytes.

Note that there is no output parameter indicating how many bytes the driver wrote to the buffer. Such an output length indication is not necessary because the entropy may be located anywhere in the buffer, so the driver may write less than `output_size` bytes but the core does not need to know this. The output parameter `estimate_bits` contains the amount of entropy, expressed in bits, which may be significantly less than `output_size * 8`.

The entry point may return the following statuses:

- `PSA_SUCCESS`: success. The output buffer contains some entropy.
- `PSA_ERROR_INSUFFICIENT_ENTROPY`: no entropy is available without blocking. This is only permitted if the `PSA_DRIVER_GET_ENTROPY_NONBLOCK` flag is set. The core may call `get_entropy` again later, giving time for entropy to be gathered or for adverse environmental conditions to be rectified.
- `PSA_ERROR_NOT_SUPPORTED`: a flag is not recognized. The core may try again with different flags.
- Other error codes indicate a transient or permanent failure of the entropy source.

Unlike most other entry points, if multiple transparent drivers include a `"get_entropy"` point, the core will call all of them (as well as the entry points from opaque drivers). Fallback is not applicable to `"get_entropy"`.

4.7.1 Entropy collection flags

- `PSA_DRIVER_GET_ENTROPY_NONBLOCK`: If this flag is clean, the driver should block until it has at least one bit of entropy. If this flag is set, the driver should avoid blocking if no entropy is readily available.
- `PSA_DRIVER_GET_ENTROPY_KEEPALIVE`: This flag is intended to help with energy management for entropy-generating peripherals. If this flag is set, the driver should expect another call to `acme_get_entropy` after a short time. If this flag is clear, the core is not expecting to call the `"get_entropy"` entry point again within a short amount of time (but it may do so nonetheless).

A very simple core can just pass `flags=0`. All entropy drivers should support this case.

If the entry point returns `PSA_ERROR_NOT_SUPPORTED`, the core may try calling the entry point again with fewer flags. Drivers should be consistent from one call to the next with respect to which flags they support. The core may cache an acceptable flag mask on its first call to an entry point.

4.7.2 Entropy collection and blocking

The intent of the `NONBLOCK` and `KEEPALIVE` flags is to support drivers for TRNG (True Random Number Generator, i.e. an entropy source peripheral) that have a long ramp-up time, especially on platforms with multiple entropy sources.

Here is a suggested call sequence for entropy collection that leverages these flags:

1. The core makes a first round of calls to `"get_entropy"` on every source with the `NONBLOCK` flag set and the `KEEPALIVE` flag set, so that drivers can prepare the TRNG peripheral.
2. The core makes a second round of calls with the `NONBLOCK` flag clear and the `KEEPALIVE` flag clear to gather needed entropy.
3. If the second round does not collect enough entropy, the core makes more similar rounds, until the total amount of collected entropy is sufficient.

4.8 Miscellaneous driver entry points

4.8.1 Driver initialization

A driver may declare an `"init"` entry point in a capability with no algorithm, key type or key size. If so, the core calls this entry point once during the initialization of the Crypto API implementation. If the init entry point of any driver fails, the initialization of the Crypto API implementation fails.

When multiple drivers have an init entry point, the order in which they are called is unspecified. It is also unspecified whether other drivers' `"init"` entry points are called if one or more init entry point fails.

On platforms where the Crypto API implementation is a subsystem of a single application, the initialization of the Crypto API implementation takes place during the call to `psa_crypto_init()`. On platforms where the Crypto API implementation is separate from the application or applications, the initialization of the Crypto API implementation takes place before or during the first time an application calls `psa_crypto_init()`.

The init entry point does not take any parameter.

4.9 Combining multiple drivers

To declare a cryptoprocessor can handle both cleartext and wrapped keys, you need to provide two driver descriptions, one for a transparent driver and one for an opaque driver. You can use the mapping in capabilities' `"names"` property to arrange for multiple driver entry points to map to the same C function.

5 Transparent drivers

5.1 Key format for transparent drivers

The format of a key for transparent drivers is the same as in applications. Refer to the documentation in the *Key format* sub-section of each key type in §9.2 Key types in the Crypto API specification. For custom key types defined by an implementation, refer to the documentation of that implementation.

5.2 Key management with transparent drivers

Transparent drivers may provide the following key management entry points:

- "import_key": called by `psa_import_key()`, only when importing a key pair or a public key (key such that `PSA_KEY_TYPE_IS_ASYMMETRIC` is true).
- `"generate_key"`: called by `psa_generate_key()`, only when generating a key pair (key such that `PSA_KEY_TYPE_IS_KEY_PAIR` is true).
- `"key_derivation_output_key"`: called by `psa_key_derivation_output_key()`, only when deriving a key pair (key such that `PSA_KEY_TYPE_IS_KEY_PAIR` is true).
- `"export_public_key"`: called by the core to obtain the public key of a key pair. The core may call this function at any time to obtain the public key, which can be for `psa_export_public_key()` but also at other times, including during a cryptographic operation that requires the public key such as a call to `psa_verify_message()` on a key pair object.

Transparent drivers are not involved when exporting, copying or destroying keys, or when importing, generating or deriving symmetric keys.

5.2.1 Key import with transparent drivers

As discussed in the general section about key management entry points, the key import entry points has the following prototype for a driver with the prefix `"acme"`:

```
psa_status_t acme_import_key(const psa_key_attributes_t *attributes,
                             const uint8_t *data,
                             size_t data_length,
                             uint8_t *key_buffer,
                             size_t key_buffer_size,
                             size_t *key_buffer_length,
                             size_t *bits);
```

This entry point has several roles:

1. Parse the key data in the input buffer `data`. The driver must support the export format for the key types that the entry point is declared for. It may support additional formats as specified in the description of psa_import_key() in the Crypto API specification.
2. Validate the key data. The necessary validation is described in *Key validation* on page 30.
3. Determine the key size and output it through `*bits`.
4. Copy the validated key data from `data` to `key_buffer`. The output must be in the canonical format documented for the key type: see the *Key format* sub-section of the key type in §9.2 Key types, so if the input is not in this format, the entry point must convert it.

5.3 Random generation entry points

A transparent driver may provide an operation family that can be used as a cryptographic random number generator. The random generation mechanism must obey the following requirements:

- The random output must be of cryptographic quality, with a uniform distribution. Therefore, if the random generator includes an entropy source, this entropy source must be fed through a CSPRNG (cryptographically secure pseudo-random number generator).
- Random generation is expected to be fast. (If a device can provide entropy but is slow at generating random data, declare it as an entropy driver instead.)
- The random generator should be able to incorporate entropy provided by an outside source. If it isn't, the random generator can only be used if it's the only entropy source on the platform. (A random generator peripheral can be declared as an entropy source instead of a random generator; this way the core will combine it with other entropy sources.)
- The random generator may either be deterministic (in the sense that it always returns the same data when given the same entropy inputs) or non-deterministic (including its own entropy source). In other words, this interface is suitable both for PRNG (pseudo-random number generator, also known as DRBG (deterministic random bit generator)) and for NRBG (non-deterministic random bit generator).

If no driver implements the random generation entry point family, the core provides an unspecified random generation mechanism.

This operation family requires the following type, entry points and parameters (TODO: where exactly are the parameters in the JSON structure?):

- Type `"random_context_t"`: the type of a random generation context.
- `"init_random"` (entry point, optional): if this function is present, the core calls it once after allocating a `"random_context_t"` object.
- `"add_entropy"` (entry point, optional): the core calls this function to inject entropy. This entry point is optional if the driver is for a peripheral that includes an entropy source of its own, however random generator drivers without entropy injection have limited portability since they can only be used on platforms with no other entropy source. This entry point is mandatory if `"initial_entropy_size"` is nonzero.
- `"get_random"` (entry point, mandatory): the core calls this function whenever it needs to obtain random data.
- `"initial_entropy_size"` (integer, mandatory): the minimum number of bytes of entropy that the core must supply before the driver can output random data. This can be `0` if the driver is for a peripheral that includes an entropy source of its own.
- `"reseed_entropy_size"` (integer, optional): the minimum number of bytes of entropy that the core should supply via "add_entropy" when the driver runs out of entropy. This value is also a hint for the size to supply if the core makes additional calls to `"add_entropy"`, for example to enforce prediction resistance. If omitted, the core should pass an amount of entropy corresponding to the expected security strength of the device (for example, pass 32 bytes of entropy when reseeding to achieve a security strength of 256 bits). If specified, the core should pass the larger of `"reseed_entropy_size"` and the amount corresponding to the security strength.

Random generation is not parametrized by an algorithm. The choice of algorithm is up to the driver.

5.3.1 Random generator initialization

The `"init_random"` entry point has the following prototype for a driver with the prefix `"acme"`:

```
psa_status_t acme_init_random(acme_random_context_t *context);
```

The core calls this entry point once after allocating a random generation context. Initially, the context object is all-bits-zero.

If a driver does not have an `"init_random"` entry point, the context object passed to the first call to `"add_entropy"` or `"get_random"` will be all-bits-zero.

5.3.2 Entropy injection

The `"add_entropy"` entry point has the following prototype for a driver with the prefix `"acme"`:

```
psa_status_t acme_add_entropy(acme_random_context_t *context,
                              const uint8_t *entropy,
                              size_t entropy_size);
```

The semantics of the parameters is as follows:

- `context`: a random generation context. On the first call to `"add_entropy"`, this object has been initialized by a call to the driver's `"init_random"` entry point if one is present, and to all-bits-zero otherwise.
- `entropy`: a buffer containing full-entropy data to seed the random generator. "Full-entropy" means that the data is uniformly distributed and independent of any other observable quantity.
- `entropy_size`: the size of the `entropy` buffer in bytes. It is guaranteed to be at least `1`, but it may be smaller than the amount of entropy that the driver needs to deliver random data, in which case the core will call the `"add_entropy"` entry point again to supply more entropy.

The core calls this function to supply entropy to the driver. The driver must mix this entropy into its internal state. The driver must mix the whole supplied entropy, even if there is more than what the driver requires, to ensure that all entropy sources are mixed into the random generator state. The driver may mix additional entropy of its own.

The core may call this function at any time. For example, to enforce prediction resistance, the core can call `"add_entropy"` immediately after each call to `"get_random"`. The core must call this function in two circumstances:

- Before the first call to the `"get_random"` entry point, to supply `"initial_entropy_size"` bytes of entropy.
- After a call to the `"get_random"` entry point returns less than the required amount of random data, to supply at least `"reseed_entropy_size"` bytes of entropy.

When the driver requires entropy, the core can supply it with one or more successive calls to the `"add_entropy"` entry point. If the required entropy size is zero, the core does not need to call `"add_entropy"`.

5.3.3 Combining entropy sources with a random generation driver

This section provides guidance on combining one or more entropy sources (each having a `"get_entropy"` entry point) with a random generation driver (with an `"add_entropy"` entry point).

Note that `"get_entropy"` returns data with an estimated amount of entropy that is in general less than the buffer size. The core must apply a mixing algorithm to the output of `"get_entropy"` to obtain full-entropy data.

For example, the core may use a simple mixing scheme based on a pseudorandom function family (F_k) with an E-bit output where $E = 8\ \text{entropy}_{\text{size}}$ and $\text{entropy}_{\text{size}}$ is the desired amount of entropy in bytes (typically the random driver's `"initial_entropy_size"` property for the initial seeding and the `"reseed_entropy_size"` property for subsequent reseeding). The core calls the `"get_entropy"` points of the available entropy drivers, outputting a string s_i and an entropy estimate e_i on the ith call. It does so until the total entropy estimate $e_1 + e_2 + \ldots + e_n$ is at least E. The core then calculates $F_k(0)$ where $k = s_1 \,||\, s_2 \,||\, \ldots \,||\, s_n$. This value is a string of $\text{entropy}_{\text{size}}$, and since (F_k) is a pseudorandom function family, $F_k(0)$ is uniformly distributed over strings of $\text{entropy}_{\text{size}}$ bytes. Therefore $F_k(0)$ is a suitable value to pass to `"add_entropy"`.

Note that the mechanism above is only given as an example. Implementations may choose a different mechanism, for example involving multiple pools or intermediate compression functions.

5.3.4 Random generator drivers without entropy injection

Random generator drivers should have the capability to inject additional entropy through the `"add_entropy"` entry point. This ensures that the random generator depends on all the entropy sources that are available on the platform. A driver where a call to `"add_entropy"` does not affect the state of the random generator is not compliant with this specification.

However, a driver may omit the `"add_entropy"` entry point. This limits the driver's portability: implementations of the Crypto API specification may reject drivers without an `"add_entropy"` entry point, or only accept such drivers in certain configurations. In particular, the `"add_entropy"` entry point is required if:

- the implementation of the Crypto API includes an entropy source that is outside the driver; or
- the core saves random data in persistent storage to be preserved across platform resets.

5.3.5 The `"get_random"` entry point

The `"get_random"` entry point has the following prototype for a driver with the prefix `"acme"`:

```
psa_status_t acme_get_random(acme_random_context_t *context,
                             uint8_t *output,
                             size_t output_size,
                             size_t *output_length);
```

The semantics of the parameters is as follows:

- `context`: a random generation context. If the driver's `"initial_entropy_size"` property is nonzero, the core must have called `"add_entropy"` at least once with a total of at least `"initial_entropy_size"` bytes of entropy before it calls `"get_random"`. Alternatively, if the driver's `"initial_entropy_size"` property is zero and the core did not call `"add_entropy"`, or if the driver has no `"add_entropy"` entry point, the core must have called `"init_random"` if present, and otherwise the context is all-bits zero.
- `output`: on success (including partial success), the first `*output_length` bytes of this buffer contain cryptographic-quality random data. The output is not used on error.
- `output_size`: the size of the `output` buffer in bytes.
- `*output_length`: on success (including partial success), the number of bytes of random data that the driver has written to the `output` buffer. This is preferably `output_size`, but the driver is allowed to return less data if it runs out of entropy as described below. The core sets this value to 0 on entry. The value is not used on error.

The driver may return the following status codes:

- `PSA_SUCCESS`: the `output` buffer contains `*output_length` bytes of cryptographic-quality random data. Note that this may be less than `output_size`; in this case the core should call the driver's `"add_entropy"` method to supply at least `"reseed_entropy_size"` bytes of entropy before calling `"get_random"` again.
- `PSA_ERROR_INSUFFICIENT_ENTROPY`: the core must supply additional entropy by calling the `"add_entropy"` entry point with at least `"reseed_entropy_size"` bytes.
- `PSA_ERROR_NOT_SUPPORTED`: the random generator is not available. This is only permitted if the driver specification for random generation has the fallback property enabled.
- Other error codes such as `PSA_ERROR_COMMUNICATION_FAILURE` or `PSA_ERROR_HARDWARE_FAILURE` indicate a transient or permanent error.

5.4 Fallback

Sometimes cryptographic accelerators only support certain cryptographic mechanisms partially. The capability description language allows specifying some restrictions, including restrictions on key sizes, but it cannot cover all the possibilities that may arise in practice. Furthermore, it may be desirable to deploy the same binary image on different devices, only some of which have a cryptographic accelerators. For these purposes, a transparent driver can declare that it only supports a capability partially, by setting the capability's `"fallback"` property to true.

If a transparent driver entry point is part of a capability which has a true `"fallback"` property and returns `PSA_ERROR_NOT_SUPPORTED`, the core will call the next transparent driver that supports the mechanism, if there is one. The core considers drivers in the order given by the driver description list.

If all the available drivers have fallback enabled and return `PSA_ERROR_NOT_SUPPORTED`, the core will perform the operation using built-in code. As soon as a driver returns any value other than `PSA_ERROR_NOT_SUPPORTED` (`PSA_SUCCESS` or a different error code), this value is returned to the application, without attempting to call any other driver or built-in code.

If a transparent driver entry point is part of a capability where the `"fallback"` property is false or omitted, the core should not include any other code for this capability, whether built in or in another transparent driver.

6 Opaque drivers

Opaque drivers allow a Crypto API implementation to delegate cryptographic operations to a separate environment that might not allow exporting key material in cleartext. The opaque driver interface is designed so that the core never inspects the representation of a key. The opaque driver interface is designed to support two subtypes of cryptoprocessors:

- Some cryptoprocessors do not have persistent storage for individual keys. The representation of a key is the key material wrapped with a master key which is located in the cryptoprocessor and never exported from it. The core stores this wrapped key material on behalf of the cryptoprocessor.
- Some cryptoprocessors have persistent storage for individual keys. The representation of a key is an identifier such as label or slot number. The core stores this identifier.

6.1 Key format for opaque drivers

The format of a key for opaque drivers is an opaque blob. The content of this blob is fully up to the driver. The core merely stores this blob.

Note that since the core stores the key context blob as it is in memory, it must only contain data that is meaningful after a reboot. In particular, it must not contain any pointers or transient handles.

The `"key_context"` property in the driver description specifies how to calculate the size of the key context as a function of the key type and size. This is an object with the following properties:

- `"base_size"` (integer or string, optional): this many bytes are included in every key context. If omitted, this value defaults to 0.
- `"key_pair_size"` (integer or string, optional): this many bytes are included in every key context for a key pair. If omitted, this value defaults to 0.

- `"public_key_size"` (integer or string, optional): this many bytes are included in every key context for a public key. If omitted, this value defaults to 0.
- `"symmetric_factor"` (integer or string, optional): every key context for a symmetric key includes this many times the key size. If omitted, this value defaults to 0.
- `"store_public_key"` (boolean, optional): If specified and true, for a key pair, the key context includes space for the public key. If omitted or false, no additional space is added for the public key.
- `"size_function"` (string, optional): the name of a function that returns the number of bytes that the driver needs in a key context for a key. This may be a pointer to function. This must be a C identifier; more complex expressions are not permitted. If the core uses this function, it supersedes all the other properties except for `"builtin_key_size"` (where applicable, if present).
- `"builtin_key_size"` (integer or string, optional): If specified, this overrides all other methods (including the `"size_function"` entry point) to determine the size of the key context for built-in keys. This allows drivers to efficiently represent application keys as wrapped key material, but built-in keys by an internal identifier that takes up less space.

The integer properties must be C language constants. A typical value for `"base_size"` is `sizeof(acme_key_context_t)` where `acme_key_context_t` is a type defined in a driver header file.

6.1.1 Size of a dynamically allocated key context

If the core supports dynamic allocation for the key context and chooses to use it, and the driver specification includes the `"size_function"` property, the size of the key context is at least

```
size_function(key_type, key_bits)
```

where `size_function` is the function named in the `"size_function"` property, `key_type` is the key type and `key_bits` is the key size in bits. The prototype of the size function is

. code-block:

```
size_t size_function(psa_key_type_t key_type, size_t key_bits);
```

6.1.2 Size of a statically allocated key context

If the core does not support dynamic allocation for the key context or chooses not to use it, or if the driver specification does not include the `"size_function"` property, the size of the key context for a key of type `key_type` and of size `key_bits` bits is:

- For a key pair (`PSA_KEY_TYPE_IS_KEY_PAIR(key_type)` is true):

  ```
  base_size + key_pair_size + public_key_overhead
  ```

  where `public_key_overhead = PSA_EXPORT_PUBLIC_KEY_MAX_SIZE(key_type, key_bits)` if the `"store_public_key"` property is true and `public_key_overhead = 0` otherwise.
- For a public key (`PSA_KEY_TYPE_IS_PUBLIC_KEY(key_type)` is true):

  ```
  base_size + public_key_size
  ```
- For a symmetric key (not a key pair or public key):

```
base_size + symmetric_factor * key_bytes
```

where `key_bytes = ((key_bits + 7) / 8)` is the key size in bytes.

6.1.3 Key context size for a secure element with storage

If the key is stored in the secure element and the driver only needs to store a label for the key, use `"base_size"` as the size of the label plus any other metadata that the driver needs to store, and omit the other properties.

If the key is stored in the secure element, but the secure element does not store the public part of a key pair and cannot recompute it on demand, additionally use the `"store_public_key"` property with the value `true`. Note that this only influences the size of the key context: the driver code must copy the public key to the key context and retrieve it on demand in its `export_public_key` entry point.

6.1.4 Key context size for a secure element without storage

If the key is stored in wrapped form outside the secure element, and the wrapped form of the key plus any metadata has up to *N* bytes of overhead, use *N* as the value of the `"base_size"` property and set the `"symmetric_factor"` property to 1. Set the `"key_pair_size"` and `"public_key_size"` properties appropriately for the largest supported key pair and the largest supported public key respectively.

6.2 Key management with opaque drivers

Opaque drivers may provide the following key management entry points:

- `"export_key"`: called by `psa_export_key()`, or by `psa_copy_key()` when copying a key from or to a different location, or as a fallback for key derivation.
- `"export_public_key"`: called by the core to obtain the public key of a key pair. The core may call this entry point at any time to obtain the public key, which can be for `psa_export_public_key()` but also at other times, including during a cryptographic operation that requires the public key such as a call to `psa_verify_message()` on a key pair object.
- `"import_key"`: called by `psa_import_key()`, or by `psa_copy_key()` when copying a key from another location.
- `"generate_key"`: called by `psa_generate_key()`.
- `"key_derivation_output_key"`: called by `psa_key_derivation_output_key()`.
- `"copy_key"`: called by `psa_copy_key()` when copying a key within the same location.
- `"get_builtin_key"`: called by functions that access a key to retrieve information about a built-in key.

In addition, secure elements that store the key material internally must provide the following two entry points:

- `"allocate_key"`: called by `psa_import_key()`, `psa_generate_key()`, `psa_key_derivation_output_key()` or `psa_copy_key()` before creating a key in the location of this driver.
- `"destroy_key"`: called by `psa_destroy_key()`.

6.2.1 Key creation in a secure element without storage

This section describes the key creation process for secure elements that do not store the key material. The driver must obtain a wrapped form of the key material which the core will store. A driver for such a secure element has no `"allocate_key"` or `"destroy_key"` entry point.

When creating a key with an opaque driver which does not have an `"allocate_key"` or `"destroy_key"` entry point:

1. The core allocates memory for the key context.
2. The core calls the driver's import, generate, derive or copy entry point.
3. The core saves the resulting wrapped key material and any other data that the key context may contain.

To destroy a key, the core simply destroys the wrapped key material, without invoking driver code.

6.2.2 Key management in a secure element with storage

This section describes the key creation and key destruction processes for secure elements that have persistent storage for the key material. A driver for such a secure element has two mandatory entry points:

- `"allocate_key"`: this function obtains an internal identifier for the key. This may be, for example, a unique label or a slot number.
- `"destroy_key"`: this function invalidates the internal identifier and destroys the associated key material.

These functions have the following prototypes for a driver with the prefix `"acme"`:

```
psa_status_t acme_allocate_key(const psa_key_attributes_t *attributes,
                               uint8_t *key_buffer,
                               size_t key_buffer_size);
psa_status_t acme_destroy_key(const psa_key_attributes_t *attributes,
                              const uint8_t *key_buffer,
                              size_t key_buffer_size);
```

When creating a persistent key with an opaque driver which has an `"allocate_key"` entry point:

1. The core calls the driver's `"allocate_key"` entry point. This function typically allocates an internal identifier for the key without modifying the state of the secure element and stores the identifier in the key context. This function should not modify the state of the secure element. It may modify the copy of the persistent state of the driver in memory.
2. The core saves the key context to persistent storage.
3. The core calls the driver's key creation entry point.
4. The core saves the updated key context to persistent storage.

If a failure occurs after the `"allocate_key"` step but before the call to the second driver entry point, the core will do one of the following:

- Fail the creation of the key without indicating this to the driver. This can happen, in particular, if the device loses power immediately after the key allocation entry point returns.
- Call the driver's `"destroy_key"` entry point.

To destroy a key, the core calls the driver's `"destroy_key"` entry point.

Note that the key allocation and destruction entry points must not rely solely on the key identifier in the key attributes to identify a key. Some implementations of the Crypto API store keys on behalf of multiple clients, and different clients may use the same key identifier to designate different keys. The manner in which the core distinguishes keys that have the same identifier but are part of the key namespace for different clients is implementation-dependent and is not accessible to drivers. Some typical strategies to allocate an internal key identifier are:

- Maintain a set of free slot numbers which is stored either in the secure element or in the driver's persistent storage. To allocate a key slot, find a free slot number, mark it as occupied and store the number in the key context. When the key is destroyed, mark the slot number as free.
- Maintain a monotonic counter with a practically unbounded range in the secure element or in the driver's persistent storage. To allocate a key slot, increment the counter and store the current value in the key context. Destroying a key does not change the counter.

TODO: explain constraints on how the driver updates its persistent state for resilience

TODO: some of the above doesn't apply to volatile keys

6.2.3 Key creation entry points in opaque drivers

The key creation entry points have the following prototypes for a driver with the prefix `"acme"`:

```
psa_status_t acme_import_key(const psa_key_attributes_t *attributes,
                             const uint8_t *data,
                             size_t data_length,
                             uint8_t *key_buffer,
                             size_t key_buffer_size,
                             size_t *key_buffer_length,
                             size_t *bits);
psa_status_t acme_generate_key(const psa_key_attributes_t *attributes,
                               uint8_t *key_buffer,
                               size_t key_buffer_size,
                               size_t *key_buffer_length);
```

If the driver has an "allocate_key" entry point, the core calls the `"allocate_key"` entry point with the same attributes on the same key buffer before calling the key creation entry point.

TODO: derivation, copy

6.2.4 Key export entry points in opaque drivers

The key export entry points have the following prototypes for a driver with the prefix `"acme"`:

```
psa_status_t acme_export_key(const psa_key_attributes_t *attributes,
                             const uint8_t *key_buffer,
                             size_t key_buffer_size,
                             uint8_t *data,
                             size_t data_size,
```

(continues on next page)

(continued from previous page)

```
                                size_t *data_length);
psa_status_t acme_export_public_key(const psa_key_attributes_t *attributes,
                                    const uint8_t *key_buffer,
                                    size_t key_buffer_size,
                                    uint8_t *data,
                                    size_t data_size,
                                    size_t *data_length);
```

The core will only call `acme_export_public_key` on a private key. Drivers implementers may choose to store the public key in the key context buffer or to recalculate it on demand. If the key context includes the public key, it needs to have an adequate size; see *Key format for opaque drivers* on page 38.

The core guarantees that the size of the output buffer (`data_size`) is sufficient to export any key with the given attributes. The driver must set `*data_length` to the exact size of the exported key.

6.3 Opaque driver persistent state

The core maintains persistent state on behalf of an opaque driver. This persistent state consists of a single byte array whose size is given by the `"persistent_state_size"` property in the driver description.

The core loads the persistent state in memory before it calls the driver's init entry point. It is adjusted to match the size declared by the driver, in case a driver upgrade changes the size:

- The first time the driver is loaded on a system, the persistent state is all-bits-zero.
- If the stored persistent state is smaller than the declared size, the core pads the persistent state with all-bits-zero at the end.
- If the stored persistent state is larger than the declared size, the core truncates the persistent state to the declared size.

The core provides the following callback functions, which an opaque driver may call while it is processing a call from the driver:

```
psa_status_t psa_crypto_driver_get_persistent_state(uint_8_t **persistent_state_ptr);
psa_status_t psa_crypto_driver_commit_persistent_state(size_t from, size_t length);
```

`psa_crypto_driver_get_persistent_state` sets `*persistent_state_ptr` to a pointer to the first byte of the persistent state. This pointer remains valid during a call to a driver entry point. Once the entry point returns, the pointer is no longer valid. The core guarantees that calls to `psa_crypto_driver_get_persistent_state` within the same entry point return the same address for the persistent state, but this address may change between calls to an entry point.

`psa_crypto_driver_commit_persistent_state` updates the persistent state in persistent storage. Only the portion at byte offsets `from` inclusive to `from + length` exclusive is guaranteed to be updated; it is unspecified whether changes made to other parts of the state are taken into account. The driver must call this function after updating the persistent state in memory and before returning from the entry point, otherwise it is unspecified whether the persistent state is updated.

The core will not update the persistent state in storage while an entry point is running except when the entry point calls `psa_crypto_driver_commit_persistent_state`. It may update the persistent state in storage after an entry point returns.

In a multithreaded environment, the driver may only call these two functions from the thread that is executing the entry point.

6.3.1 Built-in keys

Opaque drivers may declare built-in keys. Built-in keys can be accessed, but not created, through the Crypto API.

A built-in key is identified by its location and its **slot number**. Drivers that support built-in keys must provide a `"get_builtin_key"` entry point to retrieve the key data and metadata. The core calls this entry point when it needs to access the key, typically because the application requested an operation on the key. The core may keep information about the key in cache, and successive calls to access the same slot number should return the same data. This entry point has the following prototype:

```
psa_status_t acme_get_builtin_key(psa_drv_slot_number_t slot_number,
                                  psa_key_attributes_t *attributes,
                                  uint8_t *key_buffer,
                                  size_t key_buffer_size,
                                  size_t *key_buffer_length);
```

If this function returns `PSA_SUCCESS` or `PSA_ERROR_BUFFER_TOO_SMALL`, it must fill `attributes` with the attributes of the key (except for the key identifier). On success, this function must also fill `key_buffer` with the key context.

On entry, `psa_get_key_lifetime(attributes)` is the location at which the driver was declared and a persistence level with which the platform is attempting to register the key. The driver entry point may choose to change the lifetime (`psa_set_key_lifetime(attributes, lifetime)`) of the reported key attributes to one with the same location but a different persistence level, in case the driver has more specific knowledge about the actual persistence level of the key which is being retrieved. For example, if a driver knows it cannot delete a key, it may override the persistence level in the lifetime to `PSA_KEY_PERSISTENCE_READ_ONLY`. The standard attributes other than the key identifier and lifetime have the value conveyed by `PSA_KEY_ATTRIBUTES_INIT`.

The output parameter `key_buffer` points to a writable buffer of `key_buffer_size` bytes. If the driver has a "builtin_key_size" property property, `key_buffer_size` has this value, otherwise `key_buffer_size` has the value determined from the key type and size.

Typically, for a built-in key, the key context is a reference to key material that is kept inside the secure element, similar to the format returned by "allocate_key". A driver may have built-in keys even if it doesn't have an `"allocate_key"` entry point.

This entry point may return the following status values:

- `PSA_SUCCESS`: the requested key exists, and the output parameters `attributes` and `key_buffer` contain the key metadata and key context respectively, and `*key_buffer_length` contains the length of the data written to `key_buffer`.
- `PSA_ERROR_BUFFER_TOO_SMALL`: `key_buffer_size` is insufficient. In this case, the driver must pass the key's attributes in `*attributes`. In particular, `get_builtin_key(slot_number, &attributes, NULL, 0)` is a way for the core to obtain the key's attributes.
- `PSA_ERROR_DOES_NOT_EXIST`: the requested key does not exist.
- Other error codes such as `PSA_ERROR_COMMUNICATION_FAILURE` or `PSA_ERROR_HARDWARE_FAILURE` indicate a transient or permanent error.

The core will pass authorized requests to destroy a built-in key to the "destroy_key" entry point if there is one. If built-in keys must not be destroyed, it is up to the driver to reject such requests.

7 Using drivers from an application

7.1 Using transparent drivers

Transparent drivers linked into the library are automatically used for the mechanisms that they implement.

7.2 Using opaque drivers

Each opaque driver is assigned a location. The driver is invoked for all actions that use a key in that location. A key's location is indicated by its lifetime. The application chooses the key's lifetime when it creates the key.

For example, the following snippet creates an AES-GCM key which is only accessible inside the secure element designated by the location `PSA_KEY_LOCATION_acme`.

```
psa_key_attributes_t attributes = PSA_KEY_ATTRIBUTES_INIT;
psa_set_key_lifetime(&attributes, PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(
        PSA_KEY_PERSISTENCE_DEFAULT, PSA_KEY_LOCATION_acme));
psa_set_key_identifier(&attributes, 42);
psa_set_key_type(&attributes, PSA_KEY_TYPE_AES);
psa_set_key_size(&attributes, 128);
psa_set_key_algorithm(&attributes, PSA_ALG_GCM);
psa_set_key_usage_flags(&attributes, PSA_KEY_USAGE_ENCRYPT | PSA_KEY_USAGE_DECRYPT);
psa_key_id_t key;
psa_generate_key(&attributes, &key);
```

7.2.1 Lifetimes and locations

The PSA Certified Crypto API defines lifetimes as an attribute of a key that indicates where the key is stored and which application and system actions will create and destroy it. The lifetime is expressed as a 32-bit value (`typedef uint32_t psa_key_lifetime_t`). An upcoming version of the Crypto API defines more structure for lifetime values to separate these two aspects of the lifetime:

- Bits 0-7 are a *persistence level*. This value indicates what device management actions can cause it to be destroyed. In particular, it indicates whether the key is volatile or persistent.
- Bits 8-31 are a *location indicator*. This value indicates where the key material is stored and where operations on the key are performed. Location values can be stored in a variable of type `psa_key_location_t`.

An opaque driver is attached to a specific location. Keys in the default location (`PSA_KEY_LOCATION_LOCAL_STORAGE = 0`) are transparent: the core has direct access to the key material. For keys in a location that is managed by an opaque driver, only the secure element has access to the key material and can perform operations on the key, while the core only manipulates a wrapped form of the key or an identifier of the key.

7.2.2 Creating a key in a secure element

The core defines a compile-time constant for each opaque driver indicating its location called `PSA_KEY_LOCATION_`*prefix* where *prefix* is the value of the `"prefix"` property in the driver description. For convenience, Mbed TLS also declares a compile-time constant for the corresponding lifetime with the default persistence called `PSA_KEY_LIFETIME_`*prefix*. Therefore, to declare an opaque key in the location with the prefix `foo` with the default persistence, call `psa_set_key_lifetime` during the key creation as follows:

```
psa_set_key_lifetime(&attributes, PSA_KEY_LIFETIME_foo);
```

To declare a volatile key:

```
psa_set_key_lifetime(&attributes, PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(
        PSA_KEY_LOCATION_foo,
        PSA_KEY_PERSISTENCE_VOLATILE));
```

Generally speaking, to declare a key with a specified persistence:

```
psa_set_key_lifetime(&attributes, PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(
        PSA_KEY_LOCATION_foo,
        persistence));
```

Appendix A: Open questions

A.1 Value representation

A.1.1 Integers

It would be better if there was a uniform requirement on integer values. Do they have to be JSON integers? C preprocessor integers (which could be e.g. a macro defined in some header file)? C compile-time constants (allowing `sizeof`)?

This choice is partly driven by the use of the values, so they might not be uniform. Note that if the value can be zero and it's plausible that the core would want to statically allocate an array of the given size, the core needs to know whether the value is 0 so that it could use code like

```
#if ACME_FOO_SIZE != 0
    uint8_t foo[ACME_FOO_SIZE];
#endif
```

A.2 Driver declarations

A.2.1 Declaring driver entry points

The core may want to provide declarations for the driver entry points so that it can compile code using them. At the time of writing this paragraph, the driver headers must define types but there is no obligation for them to declare functions. The core knows what the function names and argument types are, so it can generate prototypes.

It should be ok for driver functions to be function-like macros or function pointers.

A.2.2 Driver location values

How does a driver author decide which location values to use? It should be possible to combine drivers from different sources. Use the same vendor assignment as for PSA services?

Can the driver assembly process generate distinct location values as needed? This can be convenient, but it's also risky: if you upgrade a device, you need the location values to be the same between builds.

The current plan is for Arm to maintain a registry of vendors and assign a location namespace to each vendor. Parts of the namespace would be reserved for implementations and integrators.

A.2.3 Multiple transparent drivers

When multiple transparent drivers implement the same mechanism, which one is called? The first one? The last one? Unspecified? Or is this an error (excluding capabilities with fallback enabled)?

The current choice is that the first one is used, which allows having a preference order on drivers, but may mask integration errors.

A.3 Driver function interfaces

A.3.1 Driver function parameter conventions

Should 0-size buffers be guaranteed to have a non-null pointers?

Should drivers really have to cope with overlap?

Should the core guarantee that the output buffer size has the size indicated by the applicable buffer size macro (which may be an overestimation)?

A.3.2 Key derivation inputs and buffer ownership

Why is `psa_crypto_driver_key_derivation_get_input_bytes` a copy, rather than giving a pointer?

The main reason is to avoid complex buffer ownership. A driver entry point does not own memory after the entry point return. This is generally necessary because an API function does not own memory after the entry point returns. In the case of key derivation inputs, this could be relaxed because the driver entry point is making callbacks to the core: these functions could return a pointer that is valid until the driver entry point returns, which would allow the driver to process the data immediately (e.g. hash it rather than copy it).

A.4 Partial computations in drivers

A.4.1 Substitution points

Earlier drafts of the driver interface had a concept of *substitution points*: places in the calculation where a driver may be called. Some hardware doesn't do the whole calculation, but only the "main" part. This goes both for transparent and opaque drivers. Some common examples:

- A processor that performs the RSA exponentiation, but not the padding. The driver should be able to leverage the padding code in the core.
- A processor that performs a block cipher operation only for a single block, or only in ECB mode, or only in CTR mode. The core would perform the block mode (CBC, CTR, CCM, …).

This concept, or some other way to reuse portable code such as specifying inner functions like `psa_rsa_pad` in the core, should be added to the specification.

A.5 Key management

A.5.1 Mixing drivers in key derivation

How does `psa_key_derivation_output_key` work when the extraction part and the expansion part use different drivers?

A.5.2 Public key calculation

ECC key pairs are represented as the private key value only. The public key needs to be calculated from that. Both transparent drivers and opaque drivers provide a function to calculate the public key (`"export_public_key"`).

The specification doesn't mention when the public key might be calculated. The core may calculate it on creation, on demand, or anything in between. Opaque drivers have a choice of storing the public key in the key context or calculating it on demand and can convey whether the core should store the public key with the `"store_public_key"` property. Is this good enough or should the specification include non-functional requirements?

A.5.3 Symmetric key validation with transparent drivers

Should the entry point be called for symmetric keys as well?

A.5.4 Support for custom import formats

Driver entry points for key management on page 29 states that the input to `"import_key"` can be an implementation-defined format. Is this a good idea? It reduces driver portability, since a core that accepts a custom format would not work with a driver that doesn't accept this format. On the other hand, if a driver accepts a custom format, the core should let it through because the driver presumably handles it more efficiently (in terms of speed and code size) than the core could.

Allowing custom formats also causes a problem with import: the core can't know the size of the key representation until it knows the bit-size of the key, but determining the bit-size of the key is part of the job of the `"import_key"` entry point. For standard key types, this could plausibly be an issue for RSA private keys, where an implementation might accept a custom format that omits the CRT parameters (or that omits *d*).

A.6 Opaque drivers

A.6.1 Opaque driver persistent state

The driver is allowed to update the state at any time. Is this ok?

An example use case for updating the persistent state at arbitrary times is to renew a key that is used to encrypt communications between the application processor and the secure element.

`psa_crypto_driver_get_persistent_state` does not identify the calling driver, so the driver needs to remember which driver it's calling. This may require a thread-local variable in a multithreaded core. Is this ok?

A.6.2 Open questions around cooked key derivation

`"derive_key"` is not a clear name. Can we use a better one?

For the `"derive_key"` entry point, how does the core choose `input_length`? Doesn't the driver know better? Should there be a driver entry point to determine the length, or should there be a callback that allows the driver to retrieve the input? Note that for some key types, it's impossible to predict the amount of input in advance, because it depends on some complex calculation or even on random data, e.g. if doing a randomized pseudo-primality test. However, for all key types except RSA, the specification mandates how the key is derived, which practically dictates how the pseudorandom key stream is consumed. So it's probably ok.

A.6.3 Fallback for key derivation in opaque drivers

Should dispatch to an opaque driver allow fallback, so that if `"key_derivation_setup"` returns `PSA_ERROR_NOT_SUPPORTED` then the core exports the key from the secure element instead?

Should the "key_derivation_output_key" capability indicate which key types the driver can derive? How should fallback work? For example, consider a secure element that implements HMAC, HKDF and ECDSA, and that can derive an HMAC key from HKDF without exporting intermediate material but can only import or randomly generate ECC keys. How does this driver convey that it can't derive an ECC key with HKDF, but it can let the core do this and import the resulting key?

A.7 Randomness

A.7.1 Input to `"add_entropy"`

Should the input to the "add_entropy" entry point be a full-entropy buffer (with data from all entropy sources already mixed), raw entropy direct from the entropy sources, or give the core a choice?

- Raw data: drivers must implement entropy mixing. `"add_entropy"` needs an extra parameter to indicate the amount of entropy in the data. The core must not do any conditioning.
- Choice: drivers must implement entropy mixing. `"add_entropy"` needs an extra parameter to indicate the amount of entropy in the data. The core may do conditioning if it wants, but doesn't have to.
- Full entropy: drivers don't need to do entropy mixing.

A.7.2 Flags for `"get_entropy"`

Are the entropy collection flags well-chosen?

A.7.3 Random generator instantiations

May the core instantiate a random generation context more than once? In other words, can there be multiple objects of type `acme_random_context_t`?

Functionally, one RNG is as good as any. If the core wants some parts of the system to use a deterministic generator for reproducibility, it can't use this interface anyway, since the RNG is not necessarily deterministic. However, for performance on multiprocessor systems, a multithreaded core could prefer to use one RNG instance per thread.

Appendix B: Changes to the API

B.1 Document change history

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