AN1311: Integrating Crypto Functionality Using PSA Crypto Compared to Mbed TLS

This application note describes how to integrate crypto functionality into applications using PSA Crypto compared to Mbed TLS. It includes a guide to migrating existing Mbed TLS implementations to PSA Crypto.

This document focuses on the Silicon Labs PSA Crypto implementations that support the RNG, symmetric and asymmetric keys, message digests, MAC, unauthenticated ciphers, AEAD, KDF, DSA, and ECDH.

This document assumes familiarity with the crypto algorithms discussed.

KEY POINTS

- Overview of Mbed TLS and PSA Crypto
- Key management in PSA Crypto
- Migration guide
- PSA Crypto platform examples

1. Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs' security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys and to execute cryptographic functions and secure services.

On Series 1 devices, the security features are implemented by the TRNG (if available) and CRYPTO peripherals.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based, or virtual (software-based). Throughout this document, the following abbreviations are used:

- HSE - Hardware Secure Engine
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Level (1)	SE Support	Part (2)
Secure Vault High (SVH)	HSE only (HSE-SVH)	EFR32xG2yB (3)
Secure Vault Mid (SVM)	HSE (HSE-SVM)	EFR32xG2yA (3)
"	VSE (VSE-SVM)	EFR32xG2y, EFM32PG2y (4)
Secure Vault Base (SVB)	N/A	MCU Series 1 and Wireless SoC Series 1

Note:

1. The features of different Secure Vault levels can be found in https://www.silabs.com/security.
2. The x is a letter (B, F, M, or Z).
3. At the time of this writing, the y is a digit (1 or 3).
4. At the time of this writing, the y is a digit (2).

Secure Vault Mid consists of two core security functions:

- Secure Boot: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- Secure Debug access control: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- Secure Key Storage: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- Anti-Tamper protection: A configurable module to protect the device against tamper attacks.
- Device authentication: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

A Secure Engine Manager and other tools allow users to configure and control their devices both in-house during testing and manufacturing, and after the device is in the field.

1.1 User Assistance

In support of these products Silicon Labs offers whitepapers, webinars, and documentation. The following table summarizes the key security documents:

Document	Summary	Applicability
AN1190: Series 2 Secure Debug	How to lock and unlock Series 2 debug access, including background information about the SE	Secure Vault Mid and High
AN1218: Series 2 Secure Boot with RTSL	Describes the secure boot process on Series 2 devices using SE	Secure Vault Mid and High
AN1247: Anti-Tamper Protection Configuration and Use	How to program, provision, and configure the anti-tamper module	Secure Vault High
AN1268: Authenticating Silicon Labs Devices using Device Certificates	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Secure Vault High
AN1271: Secure Key Storage	How to securely “wrap” keys so they can be stored in non-volatile storage.	Secure Vault High
AN1222: Production Programming of Series 2 Devices	How to program, provision, and configure security information using SE during device production	Secure Vault Mid and High

1.2 Key Reference

Public/Private keypairs along with other keys are used throughout Silicon Labs security implementations. Because terminology can sometimes be confusing, the following table lists the key names, their applicability, and the documentation where they are used.

Key Name	Customer Programmed	Purpose	Used in
Public Sign key (Sign Key Public)	Yes	Secure Boot binary authentication and/or OTA upgrade payload authentication	AN1218 (primary), AN1222
Public Command key (Command Key Public)	Yes	Secure Debug Unlock or Disable Tamper command authentication	AN1190 (primary), AN1222, AN1247
OTA Decryption key (GBL Decryption key) aka AES-128 Key	Yes	Decrypting GBL payloads used for firmware upgrades	AN1222 (primary), UG266
Attestation key aka Private Device Key	No	Device authentication for secure identity	AN1268

2. Device Compatibility

This application note supports Series 1 and Series 2 device families, and some functionality is different depending on the device.

MCU Series 1 consists of:

- EFM32JG1/EFM32PG1
- EFM32JG12/EFM32PG12
- EFM32GG11/EFM32TG11
- EFM32GG12

Wireless SoC Series 1 consists of:

- EFR32BG1/EFR32FG1/EFR32MG1
- EFR32BG12/EFR32FG12/EFR32MG12
- EFR32BG13/EFR32FG13/EFR32MG13/EFR32ZG13
- EFR32FG14/EFR32ZG14

MCU Series 2 consists of:

- EFM32PG22 (VSE-SVM)

Wireless SoC Series 2 consists of:

- EFR32BG21A/EFR32MG21A (HSE-SVM)
- EFR32BG21B/EFR32MG21B (HSE-SVH)
- EFR32BG22/EFR32FG22/EFR32MG22 (VSE-SVM)
- EFR32FG23A/EFR32ZG23A (HSE-SVM)
- EFR32FG23B/EFR32ZG23B (HSE-SVH)

3. Device Capability

The following table lists the hardware related to cryptography hardware acceleration features on Series 1 and Series 2 devices (MCU and Wireless SoC).

Table 3.1. Cryptography Hardware Acceleration Features on Series 1 and Series 2 Devices

Feature	Series 1	Series 2 - VSE	Series 2 - HSE
TRNG	TRNG peripheral (1)	CRYPTOACC peripheral	HSE
Crypto Engine (2)	CRYPTO peripheral	CRYPTOACC peripheral	HSE
Advanced Crypto (3)	—	—	HSE-SVH
Secure Key Storage (4)	—	—	HSE-SVH

Note:

1. See Table 7.2 Entropy Source on Series 1 and Series 2 Devices on page 21 for details of TRNG (True Random Number Generator) on Series 1 devices.
2. Crypto engine supports up to 256-bit ciphers and elliptic curves.
3. Advanced crypto supports up to 512-bit ciphers and 521-bit elliptic curves.
4. See Table 5.3 PSA Crypto Key Lifetime Support on Series 1 and Series 2 Devices on page 10 for details of Secure Key Storage support on HSE-SVH devices.

4. Overview

4.1 Mbed TLS

Mbed TLS is a C library that implements cryptographic primitives, X.509 certificate manipulation, and the SSL/TLS and DTLS protocols.

ARM developed Mbed TLS, which was formerly known as PolarSSL. Mbed TLS has been handed over to Trusted Firmware under open governance since March 2020.

For the time being, Trusted Firmware Mbed TLS is the project containing a reference implementation of the PSA Crypto API and the TLS portion of Mbed TLS. The following table lists different Mbed TLS versions supported in Simplicity Studio and Gecko SDK suites.

Table 4.1. Mbed TLS Versions

Mbed TLS	Gecko SDK Suite	PSA Crypto	Location in Windows
v2.26.0	3.2.x (Simplicity Studio 5)	Y	C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite \v3.2\util\third_party\crypto\mbedtls
v2.24.0	3.1.x (Simplicity Studio 5)	Y	C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite \v3.1\util\third_party\crypto\mbedtls
v2.16.6	3.0.x (Simplicity Studio 5)	—	C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite \v3.0\util\third_party\mbedtls
v2.7.12	2.7.8 (Simplicity Studio 4)	—	C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite \v2.7\util\third_party\mbedtls

4.2 PSA Crypto

Platform Security Architecture (PSA)

The Platform Security Architecture (PSA) is made up of four key stages:

- Threat modeling
- Predefined architectural choices
- Standardized implementation
- Certification

The PSA Crypto API is one of the standardized implementation features and is discussed in the following sections.

PSA Root of Trust (PSA-RoT)

For an IoT product to achieve its security goals, it must meet the requirements of one of the pillars known as Root of Trust. The following figure shows the four PSA-Certified key elements that make up the Root of Trust.

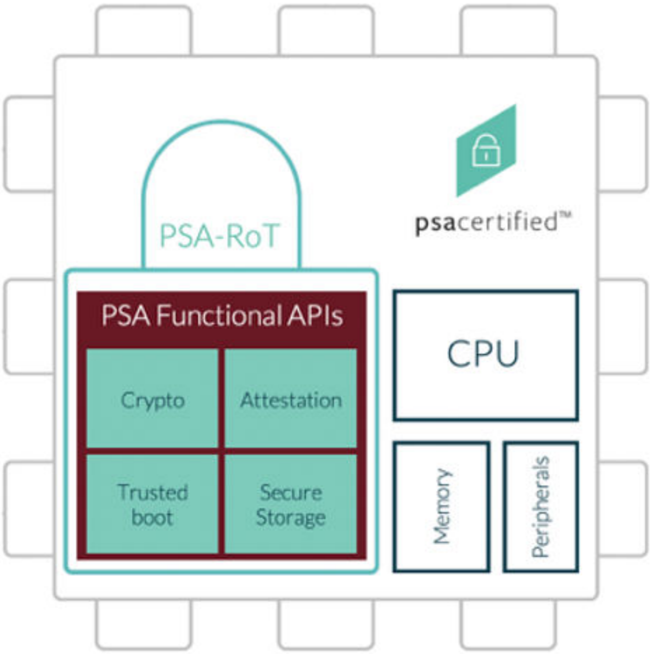


Figure 4.1. Key Requirements of PSA-RoT

The PSA Root of Trust (PSA-RoT) is a source of confidentiality (for example, crypto keys) and integrity. The PSA-RoT defines what it takes for a hardware or software system to be trusted.

The Trusted Firmware-M (TF-M) offers an open-source firmware reference implementation and APIs. These resources provide developers with a trusted code base that complies with PSA specifications and APIs that create a consistent interface to underlying Root of Trust hardware.

PSA Functional APIs

PSA Certified defines a set of PSA Functional APIs (which are implemented as part of TF-M) to access the Root of Trust features. The PSA Functional APIs provide a standardized set of vetted APIs to ensure portability and promote adherence to best practices.

- PSA Crypto APIs
- PSA Attestation APIs
- PSA Secure Storage (Internal Trusted Storage and Protected Storage) APIs

Since the Trusted boot (aka Secure boot) shown in Figure 4.1 Key Requirements of PSA-RoT on page 7 is used when booting up the device and is not used after the system is up and running, there is no need for a Trusted boot API.

The three APIs provide software developers with access to security functions to ensure interoperability across different hardware implementations of the Root of Trust. It means another hardware platform can reuse the applications in the following figure, because these APIs are standardized across various security hardware.

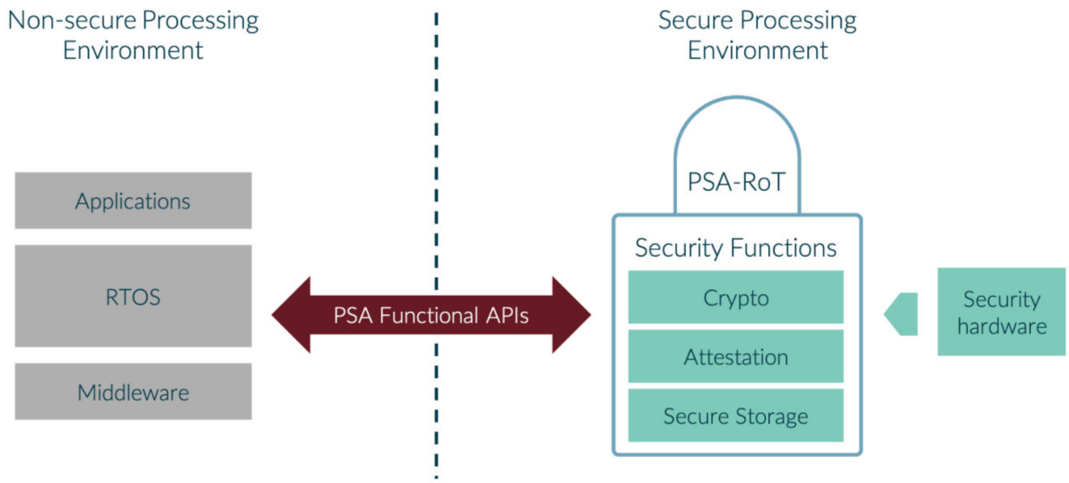


Figure 4.2. PSA Functional APIs

PSA Functional API Certification is part of PSA Certified, and demonstrates that software is compatible with the PSA Functional API specification. PSA Functional API Certified does not imply that a device has a security capability or is robust. Only PSA Certified Levels 1–3 can achieve this.

This application note only focuses on the PSA Crypto API.

PSA Crypto API

The PSA Crypto API is a low-level cryptographic API optimized for MCU and Wireless SoC. It provides APIs related to Random Number Generation (RNG), cryptographic algorithm usage, and key handling (symmetric and asymmetric).

The PSA Crypto API provides developers with an easy-to-use and easy-to-learn interface to crypto primitives. It is designed for usability and flexibility and is based on the idea of a key store. The store can isolate the keys from the rest of the applications, which means keys remain opaque in storage and only accessible for usage through crypto primitives.

5. Key Management in PSA Crypto

Key attributes are managed in a `psa_key_attributes_t` object. These are used when a key is created, after which the key attributes are impossible to change.

The actual key material is not considered an attribute of a key. Key attributes do not contain information that is generally considered highly confidential. The individual attributes (Key Types, Key Lifetimes, Key Identifiers, and Key Policies) are described in the following sections.

The `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\v3.2\util\third_party\crypto\mbedtls\include\psa\crypto_values.h` (Windows) includes the defines for macros of key attributes.

5.1 Key Types

This attribute consists of information about the key: the type, and the size used by this type. The key type and size are encoded in `psa_key_type_t` and `psa_key_bits_t` objects. The following table describes the type and size in symmetric and asymmetric keys.

Table 5.1. PSA Crypto Key Type and Size

Category	Key Type	Size in Bits
Symmetric Keys	HMAC key • `PSA_KEY_TYPE_HMAC`	Non-zero multiple of 8
	Key derivation • `PSA_KEY_TYPE_DERIVE`	Non-zero multiple of 8
	Cipher/AEAD/MAC key • `PSA_KEY_TYPE_AES`	• 128 (16-byte) • 192 (24-byte) • 256 (32-byte)
	ChaCha20/ChaCha20-Poly1305 AEAD key • `PSA_KEY_TYPE_CHACHA20`	256 (32-byte)
Elliptic Curve Cryptography (ECC) Keys	SEC random curves over prime fields • `PSA_ECC_FAMILY_SECP_R1`	• secp192r1 : 192 • secp224r1 : 224 • secp256r1 : 256 • secp384r1 : 384 • secp521r1 : 521
	SEC Koblitz curve over prime fields • `PSA_ECC_FAMILY_SECP_K1`	secp256k1 : 256
	Montgomery curves • `PSA_ECC_FAMILY_MONTGOMERY`	• Curve25519 : 255 • Curve448 : 448
	Twisted Edwards curve • `PSA_ECC_FAMILY_TWISTED_EDWARDS`	Ed25519: 255

5.2 Key Lifetimes

The lifetime is encoded in the `psa_key_lifetime_t` object (`[31:0]`). This object consists of a persistence level (`psa_key_persistence_t`) and a location indicator (`psa_key_location_t`). The persistent level indicates whether the key is volatile, persistent, or read-only. The location indicator indicates where the key is stored and where operations on the key are performed.

Table 5.2. PSA Crypto Key Lifetime

Type	Persistence Level [7:0]	Location Indicator [31:8]	Storage
Volatile Plain Key	PSA_KEY_PERSISTENCE_VOLATILE	Local (0x0)	RAM
Persistent Plain Key	PSA_KEY_PERSISTENCE_DEFAULT	Local (0x0)	Flash (2)
Volatile Wrapped Key	PSA_KEY_PERSISTENCE_VOLATILE	Secure (0x1) (1)	RAM
Persistent Wrapped Key	PSA_KEY_PERSISTENCE_DEFAULT	Secure (0x1) (1)	Flash (2)
Public Sign Key	PSA_KEY_PERSISTENCE_READ_ONLY	Secure (0x1)	SE OTP
Public Command Key	PSA_KEY_PERSISTENCE_READ_ONLY	Secure (0x1)	SE OTP
AES-128 Key	PSA_KEY_PERSISTENCE_READ_ONLY	Secure (0x1)	SE OTP
Private Device Key	PSA_KEY_PERSISTENCE_READ_ONLY	Secure (0x1)	SE OTP

Note:

1. If the key cannot be stored persistently inside the SE, it must be stored in a wrapped form in RAM or flash such that only the SE can access the key material in plaintext.
2. Persistent storage in flash memory is implemented by the NVM3 driver.

Table 5.3. PSA Crypto Key Lifetime Support on Series 1 and Series 2 Devices

Type	Series 1	Series 2 - VSE	Series 2 - HSE
Volatile Plain Key	Y	Y	Y
Persistent Plain Key	Y	Y	Y
Volatile Wrapped Key	—	—	HSE-SVH
Persistent Wrapped Key	—	—	HSE-SVH
Public Sign Key	—	— (1)	Y (2)
Public Command Key	—	— (1)	Y (2)
AES-128 Key	—	—	Y (3)
Private Device Key	—	—	HSE-SVH (2)

Note:

1. The PSA Crypto cannot access the Public Sign Key and Public Command Key in the VSE-SVM OTP.
2. These keys can only be used for ECDSA (SECP256R1) precomputed hash operations.
3. This key can only be used for AES cipher operations. The `SL_SE_BUILTIN_KEY_AES128_ALG` in `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\v3.2\util\third_party\crypto\sl_component\sl_psa_driver\inc\sli_se_opaque_types.h` (Windows) defines the cipher algorithm for this key (default is AES CTR).

5.3 Key Identifiers

A key identifier can be a permanent name for a persistent key, or a transient reference to a volatile key. Key identifiers are encoded in a `psa_key_id_t` object. The identifier and lifetime of a key indicate the location of the key in storage.

The `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\v3.2\util\third_party\crypto\sl_component\sl_psa_driver\inc\sli_se_opaque_types.h` (Windows) includes the defines for SE key identifier macros.

Table 5.4. PSA Crypto Key Identifier

Type	Key Identifier (Key ID)	SE Key Identifier
Volatile Plain Key	0 (Assigned by the PSA Crypto)	—
Persistent Plain Key	PSA_KEY_ID_USER_MIN to PSA_KEY_ID_USER_MAX	—
Volatile Wrapped Key	0 (Assigned by the PSA Crypto)	—
Persistent Wrapped Key	PSA_KEY_ID_USER_MIN to PSA_KEY_ID_USER_MAX	—
Public Sign Key	PSA_KEY_ID_VENDOR_MIN to PSA_KEY_ID_VENDOR_MAX	SL_SE_BUILTIN_KEY_SECUREBOOT_ID
Public Command Key	PSA_KEY_ID_VENDOR_MIN to PSA_KEY_ID_VENDOR_MAX	SL_SE_BUILTIN_KEY_SECUREDEBUG_ID
AES-128 Key	PSA_KEY_ID_VENDOR_MIN to PSA_KEY_ID_VENDOR_MAX	SL_SE_BUILTIN_KEY_AES128_ID
Private Device Key	PSA_KEY_ID_VENDOR_MIN to PSA_KEY_ID_VENDOR_MAX	SL_SE_BUILTIN_KEY_SYSTEM_ATTESTATION_ID

If users are about to use the PSA Crypto for persistent key storage in their application, make sure to adhere to the identifier (Key ID) allocation in https://docs.silabs.com/mbed-tls/latest/group-sl-psa-usage. The value 0 is reserved as an invalid key identifier.

```
|Key ID Range             |Scope                                           |
|-------------------------|------------------------------------------------|
|0x00000000 - 0x00004000  |Available for application use                   |
|0x00004000 - 0x0000FFFF  |Silicon Labs Layer 5 - 7 SDK components         |
|0x00010000 - 0x00FFFFFF  |Silicon Labs Layer 1 - 4 SDK components         |
|0x01000000 - 0x2FFFFFFF  |Reserved for future use by SDK components       |
|0x30000000 - 0x3FFFFFFF  |Available for application use                   |
|0x40000000 - 0x7FFEFFFF  |Range allocated by mbed TLS for internal usage  |
|0x7FFF0000 - 0x7FFFEFFF  |'Builtin' key range in PSA. Used for e.g.       |
|                         | accessing attestation keys on Vault devices.   |
|0x80000000 - 0xFFFFFFFF  |Range reserved by the PSA Crypto specification  |
|-------------------------|------------------------------------------------|
```

5.4 Key Policies

This attribute consists of usage flags and a specification of the permitted algorithm. The `psa_key_usage_t` encodes the usage flags in a bit-mask. The following table describes three kinds of usage flag in the PSA Crypto.

Table 5.5. PSA Crypto Key Usage Flags

Flag	Bit-mask	Description
Extractable	PSA_KEY_USAGE_EXPORT	Permission to export the key.
Copyable	PSA_KEY_USAGE_COPY	Permission to copy the key.
Other usage	PSA_KEY_USAGE_ENCRYPT	Permission for a symmetric encryption operation, for an AEAD encryption-and-authentication operation, or for an asymmetric encryption operation.
"	PSA_KEY_USAGE_DECRYPT	Permission for a symmetric decryption operation, for an AEAD decryption-and-verification operation, or for an asymmetric decryption operation
"	PSA_KEY_USAGE_SIGN_MESSAGE	Permission for a MAC calculation operation or for an asymmetric message signature operation.
"	PSA_KEY_USAGE_VERIFY_MESSAGE	Permission for a MAC verification operation or for an asymmetric message signature verification operation.
"	PSA_KEY_USAGE_SIGN_HASH	Permission to sign a message hash as part of an asymmetric signature operation.
"	PSA_KEY_USAGE_VERIFY_HASH	Permission to verify a message hash as part of an asymmetric signature verification operation.
"	PSA_KEY_USAGE_DERIVE	Permission to derive other keys from this key.

Note: Users can always export a public key or the public part of a key pair regardless of the value of the `PSA_KEY_USAGE_EXPORT` flag.

The `psa_algorithm_t` encodes the permitted algorithm with the key. The Symmetric Cryptographic Operation and Asymmetric Cryptographic Operation describe which algorithms can apply to the corresponding cryptographic operations.

The application must supply the algorithm to use for the operation. This algorithm is checked against the permitted algorithm policy of the key.

5.5 Summary

The `psa_key_attributes_t` object specifies the attributes for the new key during the creation process. The attributes are immutable once the key has been created.

The key identifier and lifetime in the attributes determine the location of the key in storage. The application must set the key type and size, key algorithm policy, and the appropriate key usage flags in the attributes for the key to be used in any cryptographic operations.

The key material can be copied into a new key, which can have a different lifetime or a more restrictive usage policy.

If the key creation succeeds, the PSA Crypto will return an identifier for the newly created key. The PSA Crypto can destroy a key from both volatile memory and non-volatile storage (NVM3 object). The destroying process makes the key identifier invalid, and the key identifier must not be used again by the application.

If not necessary, the extractable usage flag (`PSA_KEY_USAGE_EXPORT`) should not be set to allow the key to export in binary format.

6. Key Attributes API

The following table lists the PSA Crypto API for the key attributes.

API	Description
`psa_key_attributes_init(…)`	Initialize the key attributes (`psa_key_attributes_t`) before calling any function.
`psa_get_key_attributes(…)`	Retrieve the key attributes (`psa_key_attributes_t`) of a key.
`psa_reset_key_attributes(…)`	Reset the key attributes (`psa_key_attributes_t`) to a initialized state.
`psa_set_key_type(…)`	Declare the key type (`psa_key_type_t`) of a key.
`psa_get_key_type(…)`	Retrieve the key type (`psa_key_type_t`) from key attributes.
`psa_set_key_bits(…)`	Declare the key size (`psa_key_bits_t`) of a key.
`psa_get_key_bits(…)`	Retrieve the key size (`psa_key_bits_t`) from key attributes.
`psa_set_key_usage_flags(…)`	Declare the usage flags (`psa_key_usage_t`) for a key.
`psa_get_key_usage_flags(…)`	Retrieve the usage flags (`psa_key_usage_t`) from key attributes.
`psa_set_key_algorithm(…)`	Declare the permitted algorithm policy (`psa_algorithm_t`) for a key.
`psa_get_key_algorithm(…)`	Retrieve the algorithm policy (`psa_algorithm_t`) from key attributes.
`psa_set_key_id(…)`	Declare a key as persistent and set its key identifier (`psa_key_id_t`).
`psa_get_key_id(…)`	Retrieve the key identifier (`psa_key_id_t`) from key attributes.
`psa_set_key_lifetime(…)`	Set the location (`psa_key_lifetime_t`) of a persistent key
`psa_get_key_lifetime(…)`	Retrieve the lifetime (`psa_key_lifetime_t`) from key attributes.

The following sections describe how to use the key attributes API to set up the storage for a key. Refer to the quick reference examples in 7.4.1 Symmetric Key and 7.4.2 Asymmetric Key for more details.

Volatile Plain Key

Key ID	Persistence Level	Location Indicator	API Flow
= 0	`PSA_KEY_PERSISTENCE_VOLATILE`	Local (0x0)	It is the default setting after calling `psa_key_attributes_init()`. No need to call `psa_set_key_id()` and `psa_set_key_lifetime()`.

Example:

```
psa_key_attributes_t key_attr;
key_attr = psa_key_attributes_init();

psa_set_key_type(&key_attr, PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_SECP_R1));
psa_set_key_bits(&key_attr, 256);
psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_SIGN_HASH | PSA_KEY_USAGE_VERIFY_HASH);
psa_set_key_algorithm(&key_attr, PSA_ALG_ECDSA_ANY);
```

Persistent Plain Key

Key ID	Persistence Level	Location Indicator	API Flow
> 0	`PSA_KEY_PERSISTENCE_DEFAULT`	Local (0x0)	A non-zero key ID in `psa_set_key_id()` will change the persistence level from `PSA_KEY_PERSISTENCE_VOLATILE` to `PSA_KEY_PERSISTENCE_DEFAULT`.

Example:

```
psa_key_attributes_t key_attr;
key_attr = psa_key_attributes_init();

psa_set_key_type(&key_attr, PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_SECP_R1));
psa_set_key_bits(&key_attr, 256);
psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_SIGN_HASH | PSA_KEY_USAGE_VERIFY_HASH);
psa_set_key_algorithm(&key_attr, PSA_ALG_ECDSA_ANY);

psa_set_key_id(&key_attr, 0x02);
```

Volatile Wrapped Key

Key ID	Persistence Level	Location Indicator	API Flow
= 0	`PSA_KEY_PERSISTENCE_VOLATILE`	Secure (0x1)	Use the `psa_set_key_lifetime()` to change the location indicator from Local to Secure (0x01).

Example:

```
psa_key_attributes_t key_attr;
key_attr = psa_key_attributes_init();

psa_set_key_type(&key_attr, PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_SECP_R1));
psa_set_key_bits(&key_attr, 256);
psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_SIGN_HASH | PSA_KEY_USAGE_VERIFY_HASH);
psa_set_key_algorithm(&key_attr, PSA_ALG_ECDSA_ANY);

psa_set_key_lifetime(&key_attr, PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(PSA_KEY_PERSISTENCE_VOLATILE, 0x01));
```

Persistent Wrapped Key

Key ID	Persistence Level	Location Indicator	API Flow
> 0	`PSA_KEY_PERSISTENCE_DEFAULT`	Local (0x0)	A non-zero key ID in `psa_set_key_id()` will change the persistence level from `PSA_KEY_PERSISTENCE_VOLATILE` to `PSA_KEY_PERSISTENCE_DEFAULT`.
> 0	`PSA_KEY_PERSISTENCE_DEFAULT`	Secure (0x1)	Use the `psa_set_key_lifetime()` to change the location indicator from Local to Secure (0x01).

Example:

```
psa_key_attributes_t key_attr;
key_attr = psa_key_attributes_init();

psa_set_key_type(&key_attr, PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_SECP_R1));
psa_set_key_bits(&key_attr, 256);
psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_SIGN_HASH | PSA_KEY_USAGE_VERIFY_HASH);
psa_set_key_algorithm(&key_attr, PSA_ALG_ECDSA_ANY);

psa_set_key_id(&key_attr, 0x02);
psa_set_key_lifetime(&key_attr, PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(PSA_KEY_PERSISTENCE_DEFAULT, 0x01));
```

Note:

- The `PSA_KEY_PERSISTENCE_DEFAULT` is equal to `PSA_KEY_LIFETIME_PERSISTENT`.
- Refer to 5.3 Key Identifiers for details about the Key ID.

7. Migration Guide

System Requirements and Document

1. Simplicity Studio 5
2. Gecko SDK Suite 3.1.1 (Mbed TLS v2.24.0) or later
3. The latest SE Firmware image and release note in `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\<version>\util\se_release\public` (Windows)
4. Legacy Mbed TLS API document: ARM and Silicon Labs
5. PSA Crypto API (aka PSA Cryptography API) document: ARM and Silicon Labs

Mbed TLS Versus PSA Crypto API

Item	Mbed TLS	PSA Crypto API
Key input	APIs take key input directly.	• APIs do not take key input directly. • Key (identifier) needs to be created or imported before use. • APIs take an identifier if a key is required.
Symmetric cryptographic operation	Individual API (one-shot and streaming) for algorithm-specific functions.	• APIs are grouped by algorithm category for one-shot and streaming modes. • The exact algorithm is a parameter (`psa_algorithm_t`) to the function, not an individual API.
	Except for AEAD (encrypt and decrypt), a one-shot function is not in a pair.	Single-part (one-shot) functions are in a pair. For example, compute and verify, or encrypt and decrypt.
	Initialization and free a context are required.	Initialization and abort an operation are only required in multi-part (streaming) operations.
Error code	APIs always return an integer.	APIs always return `psa_status_t`.

Migration

In 7.4 Key Handling, 7.5 Symmetric Cryptographic Operation, and 7.6 Asymmetric Cryptographic Operation the following items will be considered when migrating from Mbed TLS to PSA Crypto.

1. The algorithms that can be used in a cryptographic operation.
2. The key attributes (type and usage flags) for specific algorithms in the PSA Crypto.
3. Security Software Components.
4. The functions (APIs) for the Mbed TLS and PSA Crypto. For each type of symmetric cryptographic operation, the functions include:
    - A pair of single-part (one-shot) functions
    - A series of functions that implement multi-part (streaming) operations
5. Quick Reference Examples (without error handling).

Platform Examples

Simplicity Studio 5 includes the PSA Crypto platform examples to evaluate the performance on key handling, symmetric and asymmetric cryptographic operations, and X.509 certificate.

- Refer to the corresponding `readme.html` file for details about each PSA Crypto platform example. This file also includes the procedures to create the project and run the example.
- Unless specified in the example, the PSA Crypto platform examples will use the software fallback feature in Mbed TLS if the cryptography hardware accelerator of the selected device does not support the corresponding ECC key or algorithm.

This application note uses Gecko SDK Suite 3.2.2. The information in the following sections may not apply to the other GSDK versions.

7.1 Security Software Components

The `slcp` file for each PSA Crypto platform example defines the software components installed in the project. The following figure shows the installed security software components (under the **Platform** category) in the PSA Crypto asymmetric key example (`psa_crypto_asymmetric_key.slcp`) on an HSE-SVH device.

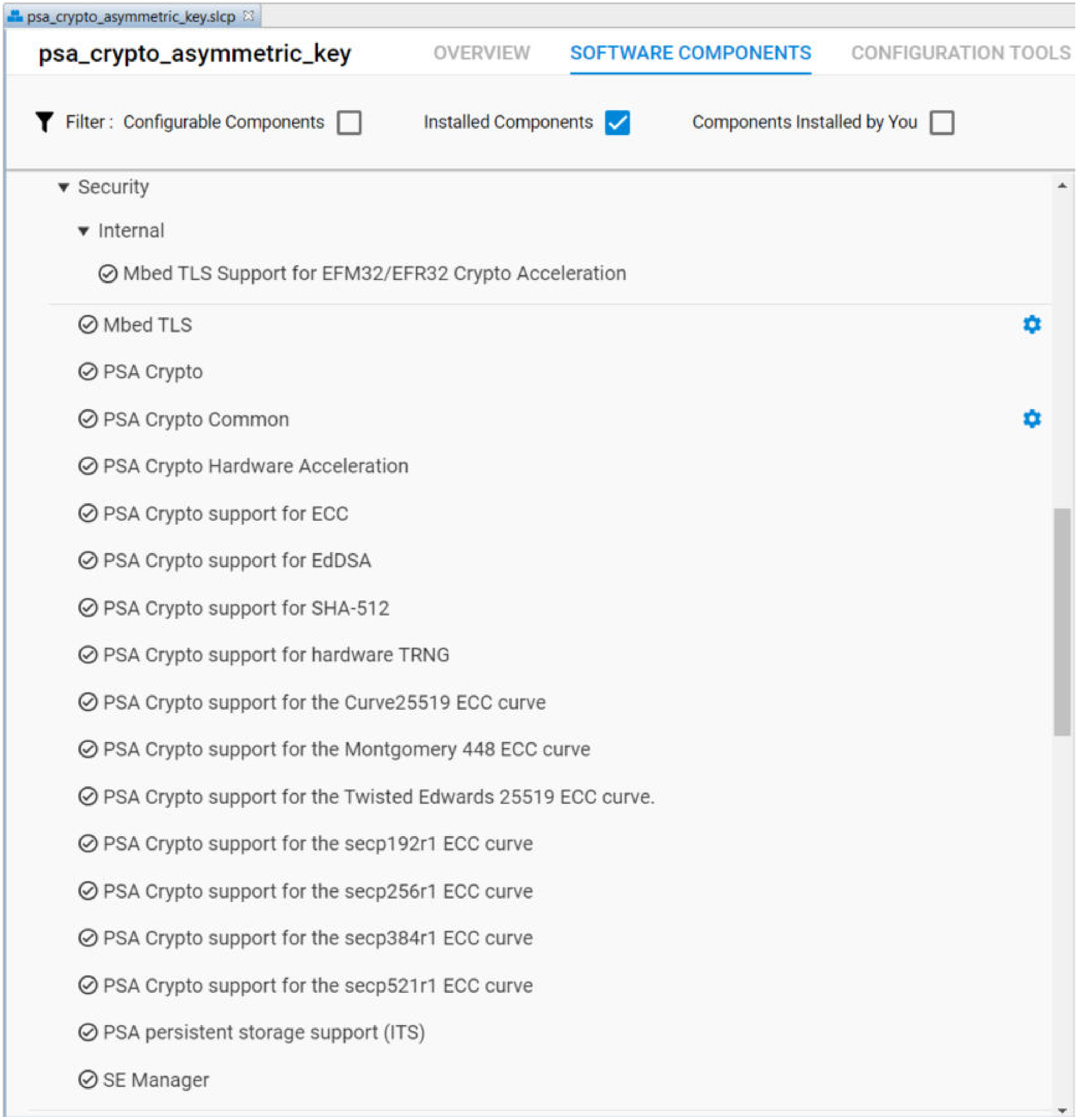


Figure 7.1. Installed Security Software Components

The Simplicity IDE uses the installed security software components to automatically generate the configuration files for Mbed TLS (`mbedtls_config_autogen.h`) and PSA Crypto (`psa_crypto_config_autogen.h`) in the `autogen` folder when creating the project.

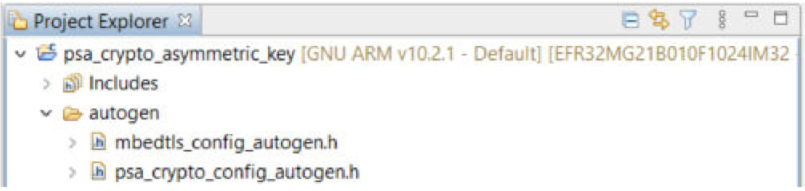


Figure 7.2. Mbed TLS and PSA Crypto Configuration Files

The user can browse the available security software components (under the **Platform** category) on the target MCU or Wireless SoC if the [**Installed Components**] checkbox is unchecked.

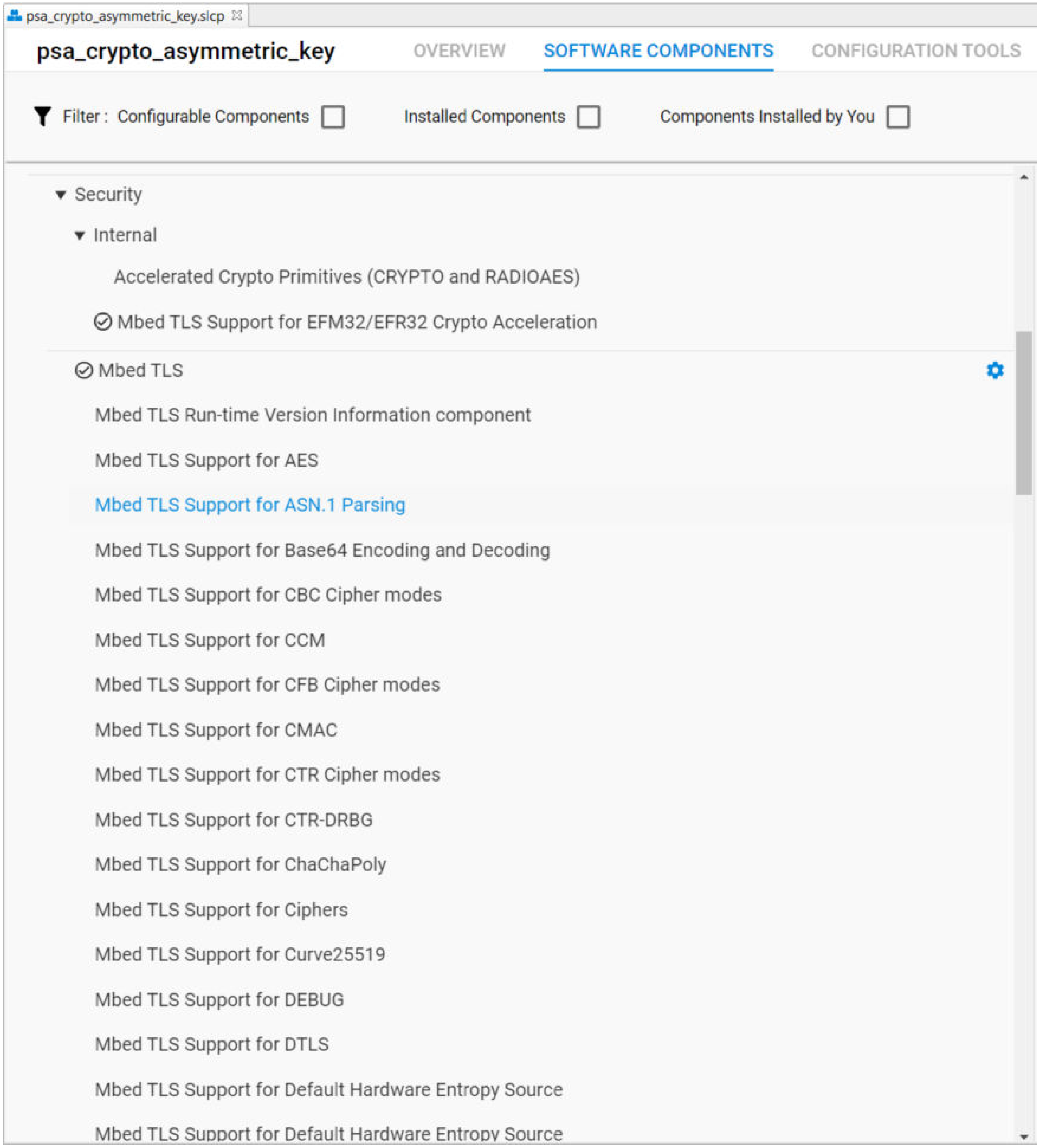


Figure 7.3. Available Security Software Components on the Target Device

The Mbed TLS and PSA Crypto configuration files automatically regenerates when the user installs or uninstalls a security software component in the project.

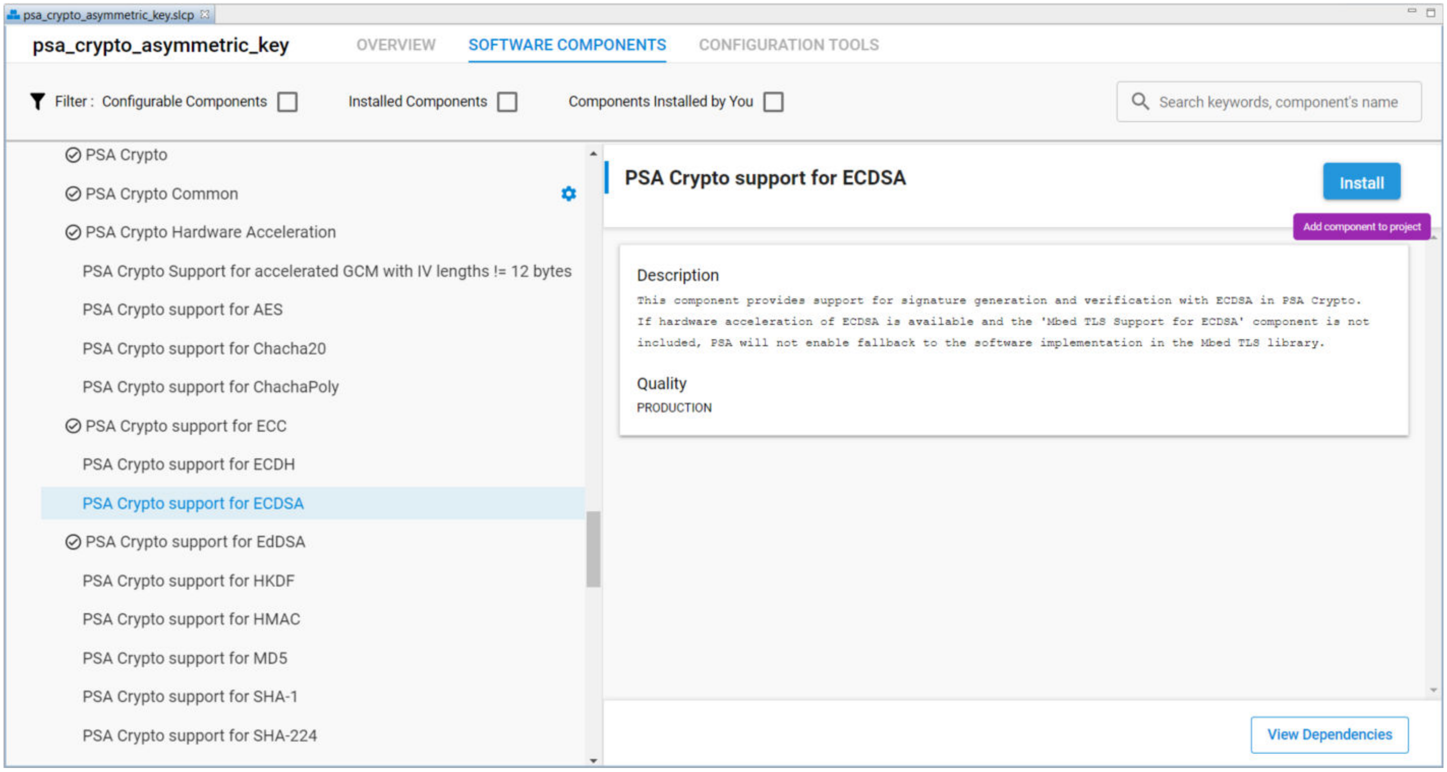


Figure 7.4. Install a Security Software Component

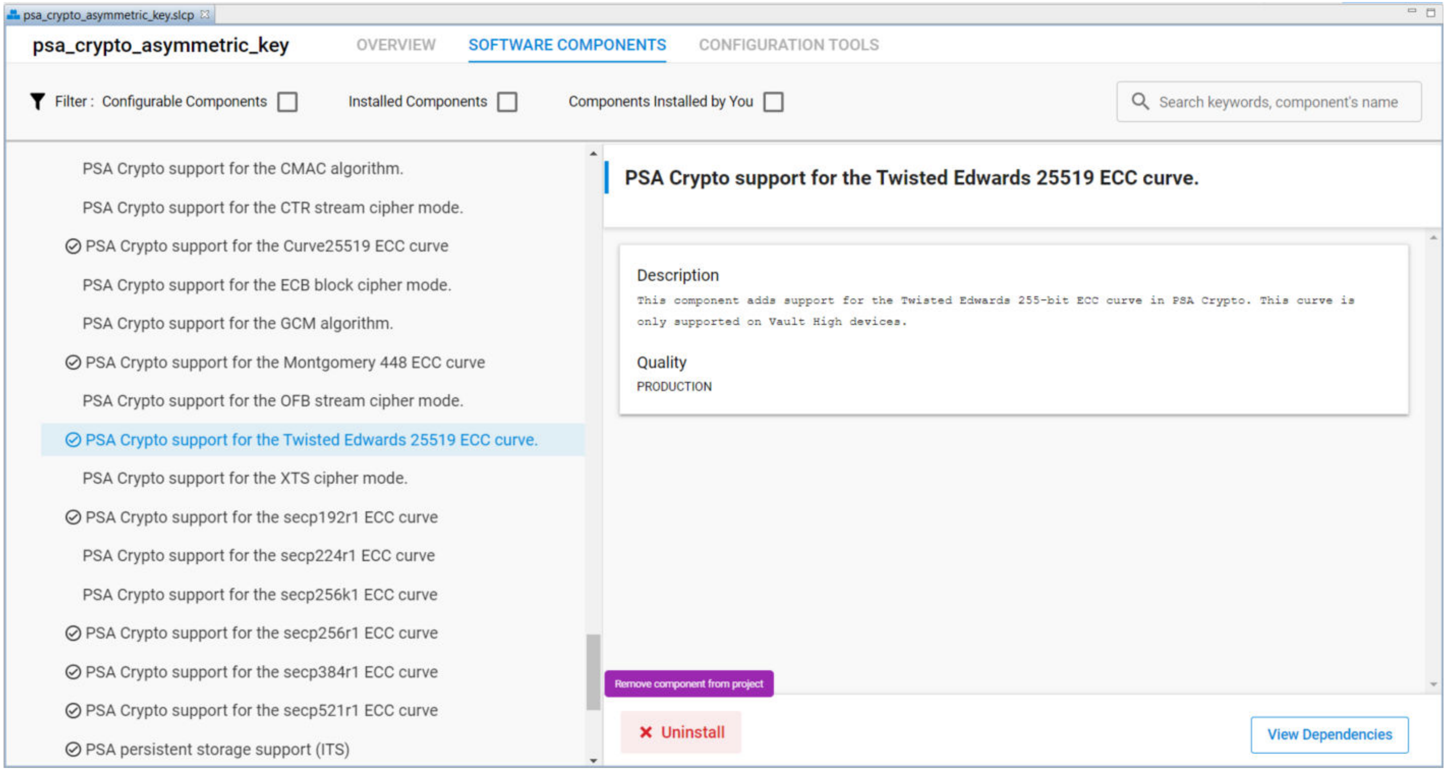


Figure 7.5. Uninstall a Security Software Component

The following security software components will be automatically added to the project after installing any cryptographic operation in PSA Crypto (like `PSA Crypto support for SHA-256`) from the user. The `SE Manager` component is only for the Series 2 devices.

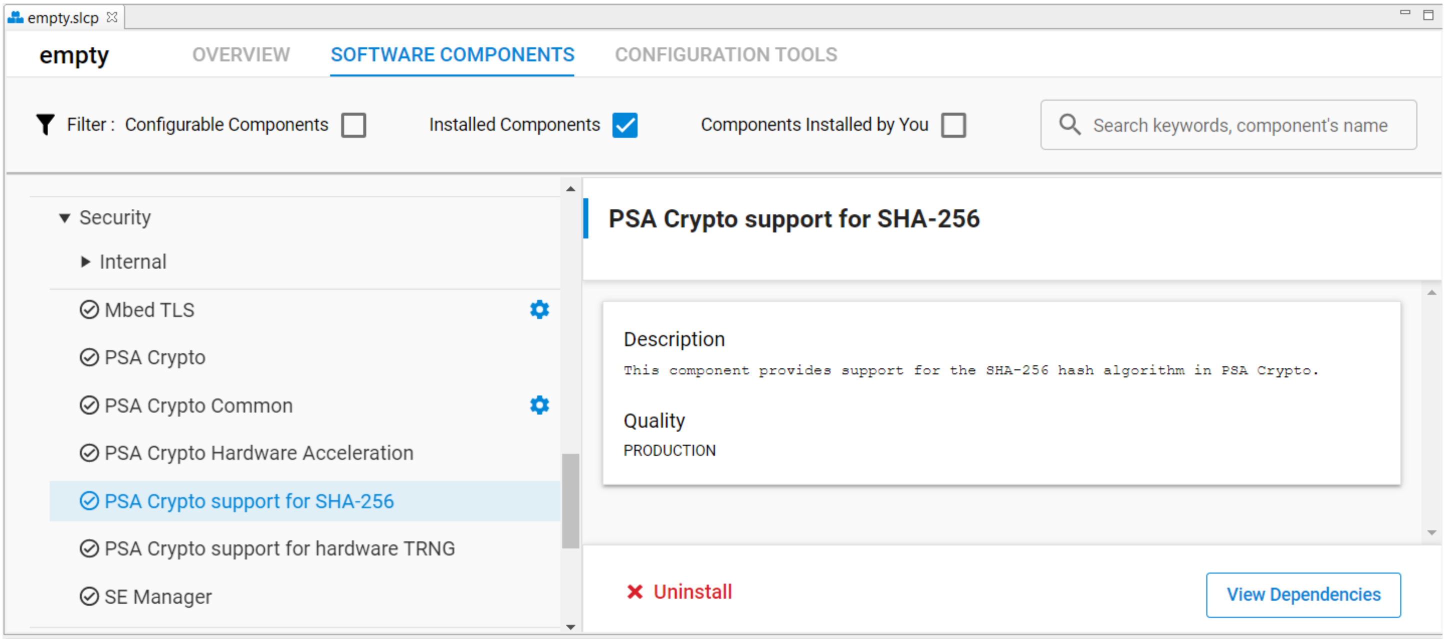


Figure 7.6. Security Software Components for PSA Crypto

If users are about to use the PSA Crypto for persistent key storage (either plain or wrapped) in their application, make sure to add the `PSA persistent storage support (ITS)` component to the project.

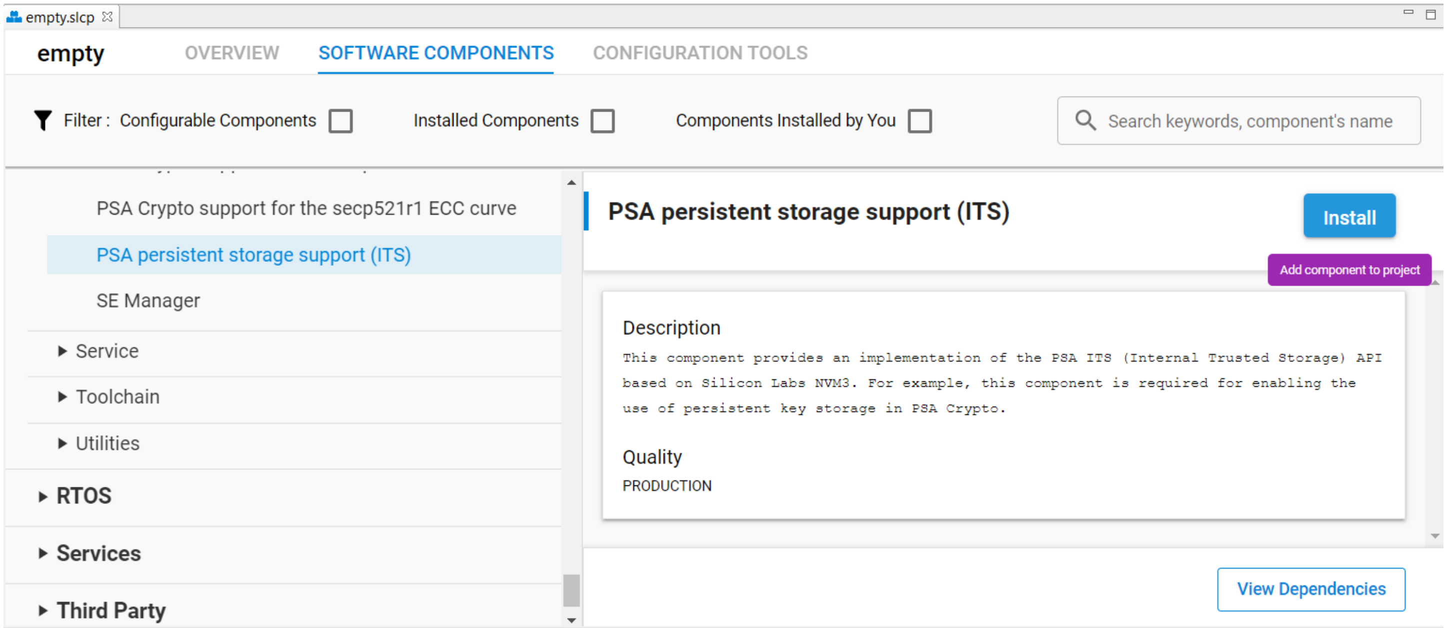
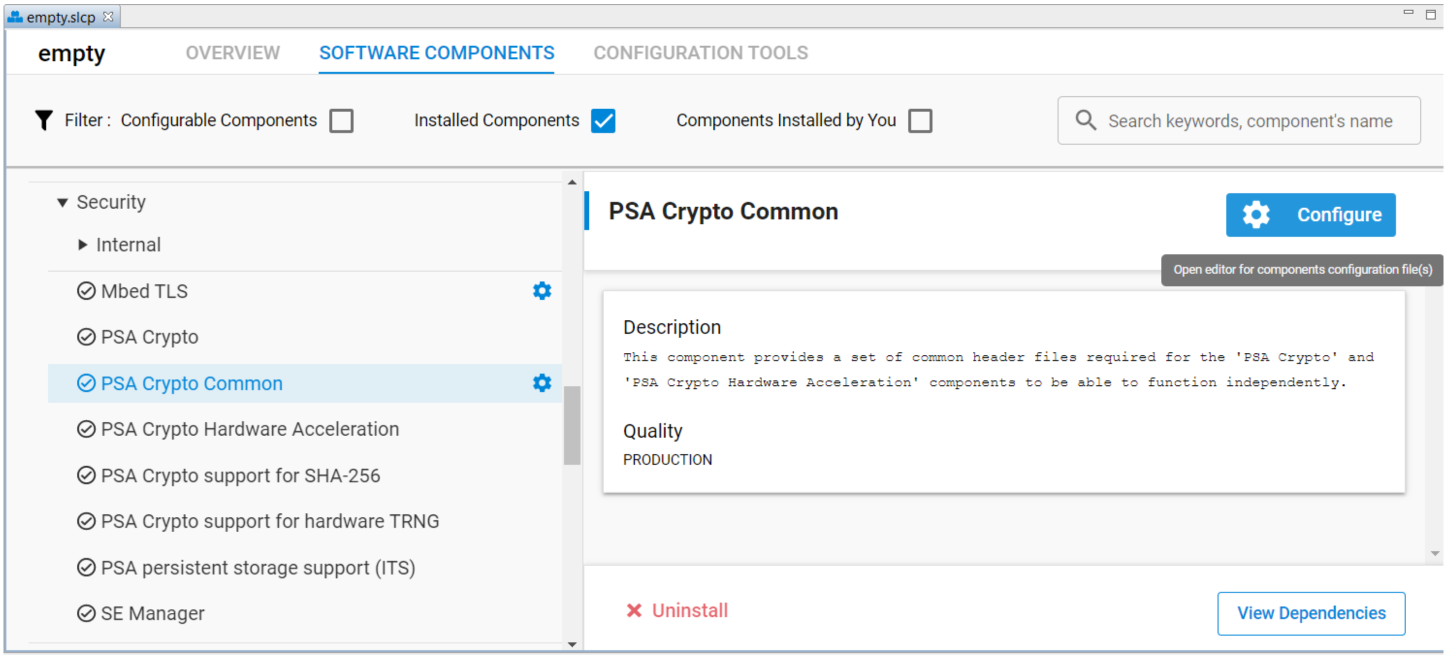


Figure 7.7. Security Software Component for Persistent Key Storage

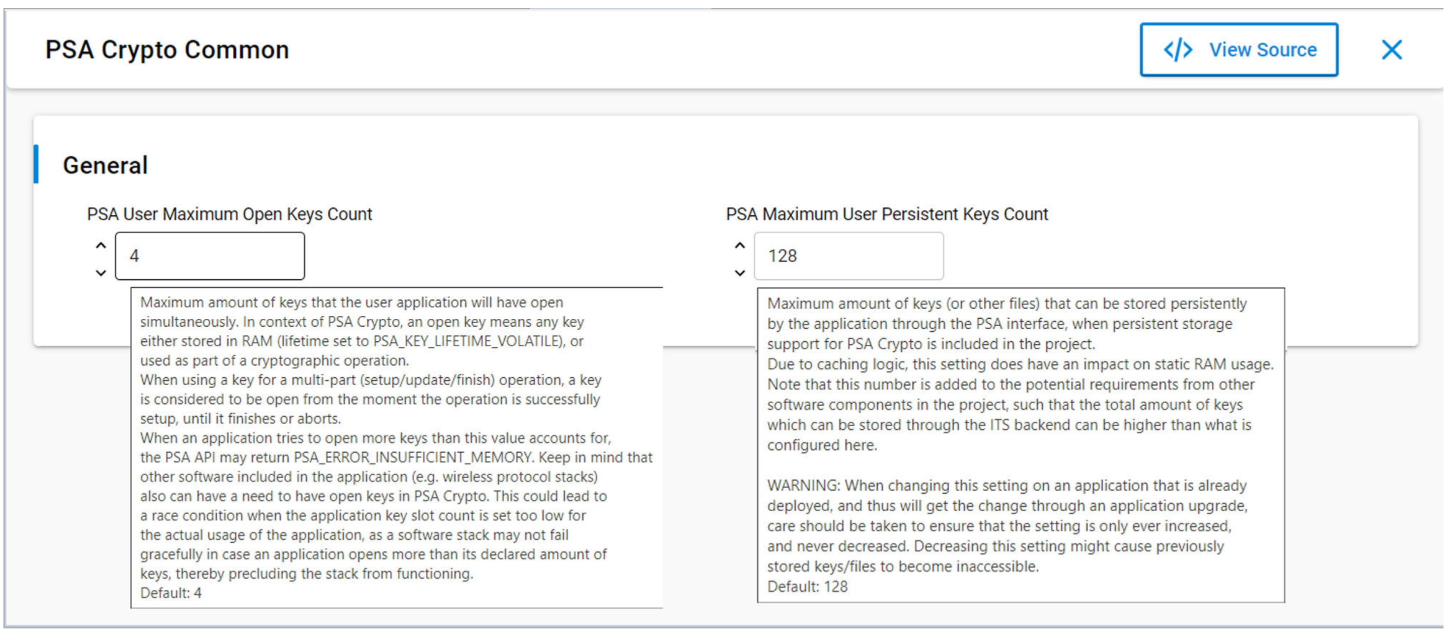
7.2 PSA Crypto Configuration

Click **PSA Crypto Common** in **Installed Components**. Click [**Configure**] to open the Configuration Wizard in Context Menu.



Enter the desired values in **PSA User Maximum Open Keys Count** (`SL_PSA_KEY_USER_SLOT_COUNT`) and **PSA Maximum User Persistent Keys Count** (`SL_PSA_ITS_USER_MAX_FILES`) to replace the default values. Click [**X**] to exit.

The default value of **PSA User Maximum Open Keys Count** is equal to 0 if the project installed any Wireless Stack before.



The PSA Crypto configuration file (`psa_crypto_config_autogen.h`) in the `autogen` folder (Figure 7.2 Mbed TLS and PSA Crypto Configuration Files on page 17) includes the definitions for PSA key slot count and maximum PSA ITS files.

```
#define MBEDTLS_PSA_KEY_SLOT_COUNT (2 + 1 + SL_PSA_KEY_USER_SLOT_COUNT)
#define SL_PSA_ITS_MAX_FILES (1 + SL_PSA_ITS_USER_MAX_FILES)
```

The first digit in `MBEDTLS_PSA_KEY_SLOT_COUNT` is Wireless Stack (if installed) dependent. The second digit should be 1 for the PSA Crypto.

The first digit in `SL_PSA_ITS_MAX_FILES` is equal to 1 if the project installed the `PSA persistent storage support (ITS)` component (Figure 7.7 Security Software Component for Persistent Key Storage on page 19) before.

7.3 Initialization and Random Number Generation (RNG)

In PSA Crypto, applications must call `psa_crypto_init()` to initialize the library before using any other function. The PSA Crypto initialization includes seeding the pseudo-random generator (CTR-DRBG) with a hardware entropy source during the execution of `psa_crypto_init()`.

Table 7.1. Initialization and RNG Functions

Item	Mbed TLS	PSA Crypto
Initialization	Initialize CTR-DRBG • `void mbedtls_entropy_init(…)` • `void mbedtls_ctr_drbg_init(…)` • `int mbedtls_entropy_add_source(…)` Seed and set up the CTR-DRBG entropy • `int mbedtls_ctr_drbg_seed(…)`	Initialize PSA Crypto • `psa_status_t psa_crypto_init(void)`
Generate random bytes	`int mbedtls_ctr_drbg_random(…)`	`psa_status_t psa_generate_random(…)`
Free resources	• `void mbedtls_ctr_drbg_free(…)` • `void mbedtls_entropy_free(…)`	`void mbedtls_psa_crypto_free(void)`

If a device includes a True Random Number Generator (TRNG) hardware module, the example will use the TRNG as an entropy source to seed the CTR-DRBG. If the device does not incorporate a TRNG, the example will use RAIL, Non-volatile (NV) seed (requires NVM3 driver), or ADC as the entropy source.

Table 7.2. Entropy Source on Series 1 and Series 2 Devices

Device	Entropy Source
MCU Series 1 - EFM32JG1, EFM32PG1	NV seed (default) or ADC
MCU Series 1 - EFM32JG12, EFM32PG12, EFM32GG11, EFM32GG12, EFM32TG11	TRNG
Wireless SoC Series 1 - EFR32xG1, EFR32xG14	RAIL
Wireless SoC Series 1 - EFR32xG12, EFR32xG13 (Revision D or later)	TRNG
All MCU Series 2 and Wireless SoC Series 2 devices	TRNG

Quick Reference Examples

PSA Crypto Initialization and Random Number Generation

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t rand_buf[32];
  psa_status_t ret;

  // Initialize the PSA Crypto and generate random numbers
  ret = psa_crypto_init();
  ret = psa_generate_random(rand_buf, sizeof(rand_buf));
}
```

7.4 Key Handling

The following table describes the main differences in key handling between Mbed TLS and PSA Crypto.

Table 7.3. Mbed TLS Versus PSA Crypto in Key Handling

Item	Mbed TLS	PSA Crypto
Random Number Generation (RNG)	It requires application code to keep track of RNG.	The core keeps track of RNG.
Buffer	It requires dedicated key buffers.	The core manages the key.
Key export	The key is exportable.	The usage flag manages this option.
Lifetime	It is volatile.	It can be volatile or persistent.
Location	Local	Local or Secure

7.4.1 Symmetric Key

A symmetric key can be used with a block cipher or a stream cipher.

Algorithms

Refer to the Symmetric Cryptographic Operation section.

Key Attributes in PSA Crypto

Refer to the Symmetric Cryptographic Operation section.

Security Software Components

Refer to the Symmetric Cryptographic Operation section.

Functions

Table 7.4. Symmetric Key Handling Functions

Item	Mbed TLS	PSA Crypto
Create a random key	Generate random numbers to a buffer • `int mbedtls_ctr_drbg_random(…)` Set up a key from a buffer (API is algorithm dependent) • `int mbedtls_aes_setkey_enc(…)` • `int mbedtls_aes_setkey_dec(…)` • `int mbedtls_cipher_setkey(…)` • `int mbedtls_ccm_setkey(…)` • `int mbedtls_gcm_setkey(…)` • `int mbedtls_chacha20_setkey(…)` • `int mbedtls_chachapoly_setkey(…)`	Create a key from randomly generated data • `psa_status_t psa_generate_key(…)`
Import a key from a buffer	API is algorithm dependent • `int mbedtls_aes_setkey_enc(…)` • `int mbedtls_aes_setkey_dec(…)` • `int mbedtls_cipher_setkey(…)` • `int mbedtls_ccm_setkey(…)` • `int mbedtls_gcm_setkey(…)` • `int mbedtls_chacha20_setkey(…)` • `int mbedtls_chachapoly_setkey(…)`	`psa_status_t psa_import_key(…)`
Copy a key	—	`psa_status_t psa_copy_key(…)`
Export a key to a buffer	The key is always in a buffer.	`psa_status_t psa_export_key(…)`
Destroy a key	Zero the key buffer.	`psa_status_t psa_destroy_key(…)`

Quick Reference Examples

Symmetric Plain Key Creation and Import

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t aes_ecb_key[16] = {0};

  psa_status_t ret;
  psa_key_id_t generate_key_id;
  psa_key_id_t import_key_id;
  psa_key_attributes_t key_attr;

  ret = psa_crypto_init();

  // Set up attributes for a AES ECB key
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_AES);
  psa_set_key_bits(&key_attr, 128);
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_ENCRYPT | PSA_KEY_USAGE_DECRYPT);
  psa_set_key_algorithm(&key_attr, PSA_ALG_ECB_NO_PADDING);

  // Generate a random volatile plain key for AES ECB
  ret = psa_generate_key(&key_attr, &generate_key_id);

  // Import a volatile plain key for AES ECB
  ret = psa_import_key(&key_attr, aes_ecb_key, sizeof(aes_ecb_key), &import_key_id);

  // Destroy the volatile plain keys for AES ECB
  ret = psa_destroy_key(generate_key_id);
  ret = psa_destroy_key(import_key_id);

  // Generate a random persistent plain key for AES ECB (ID = 0x02)
  psa_set_key_id(&key_attr, 0x02);
  ret = psa_generate_key(&key_attr, &generate_key_id);

  // Import a persistent plain key for AES ECB (ID = 0x03)
  psa_set_key_id(&key_attr, 0x03);
  ret = psa_import_key(&key_attr, aes_ecb_key, sizeof(aes_ecb_key), &import_key_id);

  // Destroy the persistent plain keys for AES ECB
  ret = psa_destroy_key(generate_key_id);
  ret = psa_destroy_key(import_key_id);
}
```

Symmetric Wrapped Key Creation and Import

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t aes_ecb_key[16] = {0};

  psa_status_t ret;
  psa_key_id_t generate_key_id;
  psa_key_id_t import_key_id;
  psa_key_attributes_t key_attr;

  ret = psa_crypto_init();

  // Set up attributes for a AES ECB key
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_AES);
  psa_set_key_bits(&key_attr, 128);
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_ENCRYPT | PSA_KEY_USAGE_DECRYPT);
  psa_set_key_algorithm(&key_attr, PSA_ALG_ECB_NO_PADDING);
  psa_set_key_lifetime(&key_attr, PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(PSA_KEY_PERSISTENCE_VOLATILE, 0x01));

  // Generate a random volatile wrapped key for AES ECB
  ret = psa_generate_key(&key_attr, &generate_key_id);

  // Import a volatile wrapped key for AES ECB
  ret = psa_import_key(&key_attr, aes_ecb_key, sizeof(aes_ecb_key), &import_key_id);

  // Destroy the volatile wrapped keys for AES ECB
  ret = psa_destroy_key(generate_key_id);
  ret = psa_destroy_key(import_key_id);

  // Generate a random persistent wrapped key for AES ECB (ID = 0x02)
  psa_set_key_id(&key_attr, 0x02);
  psa_set_key_lifetime(&key_attr, PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(PSA_KEY_PERSISTENCE_DEFAULT, 0x01));
  ret = psa_generate_key(&key_attr, &generate_key_id);

  // Import a persistent wrapped key for AES ECB (ID = 0x03)
  psa_set_key_id(&key_attr, 0x03);
  psa_set_key_lifetime(&key_attr, PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(PSA_KEY_PERSISTENCE_DEFAULT, 0x01));
  ret = psa_import_key(&key_attr, aes_ecb_key, sizeof(aes_ecb_key), &import_key_id);

  // Destroy the persistent wrapped keys for AES ECB
  ret = psa_destroy_key(generate_key_id);
  ret = psa_destroy_key(import_key_id);
}
```

PSA Crypto Platform Example

Click the `View Project Documentation` link to open the `readme.html` file.

The following table describes the implementation status of the PSA Crypto symmetric key platform example.

Table 7.5. PSA Crypto Symmetric Key Platform Example on Series 1 and Series 2 Devices

Key	Series 1	Series 2 - VSE	Series 2 - HSE	Remark
Extractable	Y	Y	Y	—
Copyable	Y	Y	Y	The PSA_KEY_USAGE_COPY usage flag does not apply to the wrapped key.
Wrapped	—	—	Y	Only on HSE-SVH devices.
128-bit	Y	Y	Y	—
192-bit	Y	Y	Y	—
256-bit	Y	Y	Y	—

7.4.2 Asymmetric Key

An asymmetric key pair consists of a (secret) private key and a public key (not secret). A public key cryptographic algorithm can be used for key distribution and digital signatures.

Algorithms

Refer to the Asymmetric Cryptographic Operation section.

Key Attributes in PSA Crypto

Refer to the Asymmetric Cryptographic Operation section.

Security Software Components

Refer to the Asymmetric Cryptographic Operation section.

Functions

Table 7.6. Asymmetric Key Handling Functions

Item	Mbed TLS	PSA Crypto
Create a random key	ECDH • `void mbedtls_ecdh_init(…)` • `int mbedtls_ecp_group_load(…)` • `int mbedtls_ecdh_gen_public(…)` ECDSA • `void mbedtls_ecdsa_init(…)` • `int mbedtls_ecdsa_genkey(…)`	Create a key from randomly generated data • `psa_status_t psa_generate_key(…)`
Import a private or public key from a buffer	• `int mbedtls_ecp_point_read_binary(…)` • `int mbedtls_mpi_read_binary(…)`	`psa_status_t psa_import_key(…)`
Copy a key	—	`psa_status_t psa_copy_key(…)`
Export a private key to a buffer	• `int mbedtls_ecp_point_write_binary(…)` • `int mbedtls_mpi_write_binary(…)`	`psa_status_t psa_export_key(…)`
Export a public key to a buffer	• `int mbedtls_ecp_point_write_binary(…)` • `int mbedtls_mpi_write_binary(…)`	`psa_status_t psa_export_public_key(…)`
Destroy a key	ECDH • `void mbedtls_ecdh_free(…)` ECDSA • `void mbedtls_ecdsa_free(…)`	`psa_status_t psa_destroy_key(…)`

Table 7.7. Asymmetric Key Size for Import and Export in PSA Crypto

ECC Key	Private Key Size (Import and Export)	Public Key Size (Import and Export)
SECP192R1	24-byte	49-byte
SECP224R1	28-byte	57-byte
SECP256R1	32-byte	65-byte
SECP384R1	48-byte	97-byte
SECP521R1	66-byte	133-byte
SECP256K1	32-byte	65-byte
CURVE25519	32-byte	32-byte
CURVE448	56-byte	56-byte
Ed25519	32-byte	32-byte

Note:

- The public key of the SECP curve is stored in an uncompressed format (prefix `0x04` with the X and Y coordinates).
- It cannot import a public key in wrapped form.
- EFR32xG21A/B devices do not support hardware acceleration on the SECP224R1 curve.
- Only the VSE-SVM devices support hardware acceleration on the SECP256K1 curve.
- SECP224R1 and SECP256K1 with wrapped keys are not supported yet.

Quick Reference Examples

Asymmetric Key Creation and Import

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t public_key[65];       // Uncompressed point format
  size_t pubkey_len;

  psa_status_t ret;
  psa_key_id_t key_id;
  psa_key_attributes_t key_attr;

  ret = psa_crypto_init();

  // Check if there is already a persistent key with the given identifier (ID = 0x02)
  key_attr = psa_key_attributes_init();
  ret = psa_get_key_attributes(0x02, &key_attr);

  if (ret == PSA_ERROR_INVALID_HANDLE) {
    // Key identifier does not exist, set up attributes for a persistent private wrapped key (SECP256R1)
    key_attr = psa_key_attributes_init();
    psa_set_key_type(&key_attr, PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_SECP_R1));
    psa_set_key_bits(&key_attr, 256);
    psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_SIGN_HASH | PSA_KEY_USAGE_VERIFY_HASH);
    psa_set_key_algorithm(&key_attr, PSA_ALG_ECDSA_ANY);
    psa_set_key_id(&key_attr, 0x02);
    psa_set_key_lifetime(&key_attr, PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(PSA_KEY_PERSISTENCE_DEFAULT, 0x01));

    // Generate a random persistent private wrapped key (ID = 0x02)
    ret = psa_generate_key(&key_attr, &key_id);

    // Export a public key from a persistent private wrapped key (ID = 0x02
    ret = psa_export_public_key(0x02, public_key, sizeof(public_key), &pubkey_len);

    // Set up attributes for a public key (SECP256R1)
    key_attr = psa_key_attributes_init();
    psa_set_key_type(&key_attr, PSA_KEY_TYPE_ECC_PUBLIC_KEY(PSA_ECC_FAMILY_SECP_R1));
    psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_VERIFY_HASH);
    psa_set_key_algorithm(&key_attr, PSA_ALG_ECDSA_ANY);

    // Import a public key
    ret = psa_import_key(&key_attr, public_key, sizeof(public_key), &key_id);

    // Destroy a persistent private wrapped key (ID = 0x02) and public key
    ret = psa_destroy_key(0x02);
    ret = psa_destroy_key(key_id);
  } else if (ret == PSA_SUCCESS) {
    // Key identifier already exists
    return;
  } else {
    // Unexpected error
    return;
  }
}
```

PSA Crypto Platform Example

Click the `View Project Documentation` link to open the `readme.html` file.

The following table describes the implementation status of the PSA Crypto asymmetric key platform example.

Table 7.8. PSA Crypto Asymmetric Key Platform Example on Series 1 and Series 2 Devices

ECC Key	Series 1	Series 2 - VSE	Series 2 - HSE	Remark
Extractable	Y	Y	Y	—
Copyable	Y	Y	Y	The `PSA_KEY_USAGE_COPY` usage flag does not apply to the wrapped key.
Wrapped	—	—	Y	Only on HSE-SVH devices.
SECP192R1	Y	Y	Y	—
SECP256R1	Y	Y	Y	—
SECP384R1	Y	Y	Y	Hardware acceleration only on HSE-SVH devices.
SECP521R1	Y	Y	Y	Hardware acceleration only on HSE-SVH devices.
CURVE25519	Y	Y	Y	Hardware acceleration only on HSE-SVH devices.
CURVE448	—	—	Y	Only on HSE-SVH devices with hardware acceleration.
Ed25519	—	—	Y	Only on HSE-SVH devices with hardware acceleration.

Note:
- This example does not include SECP224R1 and SECP256K1.
- The PSA Crypto does not yet support software fallback on the CURVE448 and Ed25519.

7.5 Symmetric Cryptographic Operation

7.5.1 Message Digests

Message digests are designed to protect the integrity of a piece of data or media to detect changes to any part of a message. They are a type of cryptography utilizing hash values that can warn the receiver of any modifications applied to a message transmitted over an insecure channel.

Algorithms

Table 7.9. Hash Algorithms

Algorithm	Mbed TLS	PSA Crypto
SHA-1	`MBEDTLS_MD_SHA1`	`PSA_ALG_SHA_1`
SHA-2	• `MBEDTLS_MD_SHA224` • `MBEDTLS_MD_SHA256` • `MBEDTLS_MD_SHA384` • `MBEDTLS_MD_SHA512`	• `PSA_ALG_SHA_224` • `PSA_ALG_SHA_256` • `PSA_ALG_SHA_384` • `PSA_ALG_SHA_512`

Security Software Components

Table 7.10. Security Software Components for Hash Algorithms

Algorithm	Security Software Components
`PSA_ALG_SHA_1`	PSA Crypto support for SHA-1
`PSA_ALG_SHA_224`	PSA Crypto support for SHA-224
`PSA_ALG_SHA_256`	PSA Crypto support for SHA-256
`PSA_ALG_SHA_384`	PSA Crypto support for SHA-384
`PSA_ALG_SHA_512`	PSA Crypto support for SHA-512

Single-Part Functions

Table 7.11. Single-Part Hashing Functions

Mbed TLS	PSA Crypto
Generic • `int mbedtls_md(…)` Algorithm specific • `int mbedtls_sha1_ret(…)` • `int mbedtls_sha256_ret(…)` • `int mbedtls_sha512_ret(…)`	`psa_status_t psa_hash_compute(…)`
—	`psa_status_t psa_hash_compare(…)`

Multi-Part Operations

Table 7.12. Multi-Part Hashing Operations

Mbed TLS	PSA Crypto
Generic • `void mbedtls_md_init(…)` Algorithm specific • `void mbedtls_sha1_init(…)` • `void mbedtls_sha256_init(…)` • `void mbedtls_sha512_init(…)`	`psa_hash_operation_t psa_hash_operation_init(void)`
Generic • `int mbedtls_md_setup(…)` • `int mbedtls_md_starts(…)` Algorithm specific • `int mbedtls_sha1_starts_ret(…)` • `int mbedtls_sha256_starts_ret(…)` • `int mbedtls_sha512_starts_ret(…)`	`psa_status_t psa_hash_setup(…)`
Generic • `int mbedtls_md_update(…)` Algorithm specific • `int mbedtls_sha1_update_ret(…)` • `int mbedtls_sha256_update_ret(…)` • `int mbedtls_sha512_update_ret(…)`	`psa_status_t psa_hash_update(…)`
Generic • `int mbedtls_md_finish(…)` Algorithm specific • `int mbedtls_sha1_finish_ret(…)` • `int mbedtls_sha256_finish_ret(…)` • `int mbedtls_sha512_finish_ret(…)`	`psa_status_t psa_hash_finish(…)`
—	`psa_status_t psa_hash_verify(…)`
Generic • `void mbedtls_md_free(…)` Algorithm specific • `void mbedtls_sha1_free(…)` • `void mbedtls_sha256_free(…)` • `void mbedtls_sha512_free(…)`	`psa_status_t psa_hash_abort(…)`
Generic • `int mbedtls_md_clone(…)` Algorithm specific • `int mbedtls_sha1_clone(…)` • `int mbedtls_sha256_clone(…)` • `int mbedtls_sha512_clone(…)`	`psa_status_t psa_hash_clone(…)`

Quick Reference Examples

SHA-256

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  char test_msg[] = {"abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"};
  uint8_t expect_sha256_hash[] = {
    0x24, 0x8d, 0x6a, 0x61, 0xd2, 0x06, 0x38, 0xb8, 0xe5, 0xc0, 0x26, 0x93, 0x0c, 0x3e, 0x60, 0x39,
    0xa3, 0x3c, 0xe4, 0x59, 0x64, 0xff, 0x21, 0x67, 0xf6, 0xec, 0xed, 0xd4, 0x19, 0xdb, 0x06, 0xc1
  };

  uint8_t hash_buf[32];
  size_t hash_len;
  psa_status_t ret;

  ret = psa_crypto_init();

  // Calculate the hash of a message
  ret = psa_hash_compute(PSA_ALG_SHA_256,
                         (uint8_t *)test_msg,
                         sizeof(test_msg) - 1,
                         hash_buf,
                         sizeof(hash_buf),
                         &hash_len);

  // Calculate the hash of a message and compare it with a reference value
  ret = psa_hash_compare(PSA_ALG_SHA_256,
                         (uint8_t *)test_msg,
                         sizeof(test_msg) - 1,
                         expect_sha256_hash,
                         sizeof(expect_sha256_hash));
}
```

PSA Crypto Platform Example

Click the `View Project Documentation` link to open the `readme.html` file.

The following table describes the implementation status of the PSA Crypto hash platform example.

Table 7.13. PSA Crypto Hash Platform Example on Series 1 and Series 2 Devices

Algorithm	Series 1	Series 2 - VSE	Series 2 - HSE	Remark
SHA-1	Y	Y	Y	—
SHA-224	Y	Y	Y	—
SHA-256	Y	Y	Y	—
SHA-384	Y	Y	Y	Hardware acceleration only on HSE-SVH devices.
SHA-512	Y	Y	Y	Hardware acceleration only on HSE-SVH devices.

7.5.2 Message Authentication Codes (MAC)

A Message Authentication Code (MAC), sometimes known as a tag, is a short piece of information used to confirm that the message came from the stated sender (its authenticity) and has not been changed.

Algorithms

Table 7.14. MAC Algorithms

Algorithm	Mbed TLS	PSA Crypto
HMAC	• MBEDTLS_MD_SHA1 • MBEDTLS_MD_SHA224 • MBEDTLS_MD_SHA256 • MBEDTLS_MD_SHA384 • MBEDTLS_MD_SHA512	• PSA_ALG_HMAC(PSA_ALG_SHA_1) • PSA_ALG_HMAC(PSA_ALG_SHA_224) • PSA_ALG_HMAC(PSA_ALG_SHA_256) • PSA_ALG_HMAC(PSA_ALG_SHA_384) • PSA_ALG_HMAC(PSA_ALG_SHA_512)
CMAC	• MBEDTLS_CIPHER_AES_128_ECB • MBEDTLS_CIPHER_AES_192_ECB • MBEDTLS_CIPHER_AES_256_ECB	PSA_ALG_CMAC

Key Attributes in PSA Crypto

Table 7.15. Key Attributes for MAC Algorithms

Algorithm	Key Type	Key Size in Bits	Key Usage Flag
• PSA_ALG_HMAC(PSA_ALG_SHA_1) • PSA_ALG_HMAC(PSA_ALG_SHA_224) • PSA_ALG_HMAC(PSA_ALG_SHA_256) • PSA_ALG_HMAC(PSA_ALG_SHA_384) • PSA_ALG_HMAC(PSA_ALG_SHA_512)	PSA_KEY_TYPE_HMAC	Multiple of 8	• PSA_KEY_USAGE_SIGN_HASH • PSA_KEY_USAGE_VERIFY_HASH
PSA_ALG_CMAC	PSA_KEY_TYPE_AES	• 128 (16-byte) • 192 (24-byte) • 256 (32-byte)	

Security Software Components

Table 7.16. Security Software Components for MAC Algorithms

Algorithm	Security Software Components
PSA_ALG_HMAC(PSA_ALG_SHA_1)	PSA Crypto support for HMAC & PSA Crypto support for SHA-1
PSA_ALG_HMAC(PSA_ALG_SHA_224)	PSA Crypto support for HMAC & PSA Crypto support for SHA-224
PSA_ALG_HMAC(PSA_ALG_SHA_256)	PSA Crypto support for HMAC & PSA Crypto support for SHA-256
PSA_ALG_HMAC(PSA_ALG_SHA_384)	PSA Crypto support for HMAC & PSA Crypto support for SHA-384
PSA_ALG_HMAC(PSA_ALG_SHA_512)	PSA Crypto support for HMAC & PSA Crypto support for SHA-512
PSA_ALG_CMAC	PSA Crypto support for the CMAC algorithm

Single-Part Functions

Table 7.17. Single-Part MAC Functions

Mbed TLS	PSA Crypto
HMAC • `int mbedtls_md_hmac(…)` CMAC • `int mbedtls_cipher_cmac(…)`	`psa_status_t psa_mac_compute(…)`
—	`psa_status_t psa_mac_verify(…)`

Multi-Part Operations

Table 7.18. Multi-Part MAC Operations

Mbed TLS	PSA Crypto
HMAC • `void mbedtls_md_init(…)` • `int mbedtls_md_setup(…)` CMAC • `void mbedtls_cipher_init(…)`	`psa_mac_operation_t psa_mac_operation_init(void)`
HMAC • `int mbedtls_md_hmac_starts(…)` CMAC • `int mbedtls_cipher_cmac_starts(…)`	`psa_status_t psa_mac_sign_setup(…)`
—	`psa_status_t psa_mac_verify_setup(…)`
HMAC • `int mbedtls_md_hmac_update(…)` CMAC • `int mbedtls_cipher_cmac_update(…)`	`psa_status_t psa_mac_update(…)`
HMAC • `int mbedtls_md_hmac_finish(…)` CMAC • `int mbedtls_cipher_cmac_finish(…)`	`psa_status_t psa_mac_sign_finish(…)`
—	`psa_status_t psa_mac_verify_finish(…)`
HMAC • `void mbedtls_md_free(…)` CMAC • `void mbedtls_cipher_free(…)`	`psa_status_t psa_mac_abort(…)`

Quick Reference Examples

CMAC and HMAC (SHA-256)

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t hmac_key[] = {
    0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66
  };
  uint8_t hmac_msg[] = {
    0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66
  };
  uint8_t cmac_key[] = {
    0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe, 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
    0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7, 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4
  };
  uint8_t cmac_msg[] = {
    0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
    0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
    0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11
  };
  uint8_t mac_buf[32];
  size_t mac_len;

  psa_status_t ret;
  psa_key_id_t key_id;
  psa_key_attributes_t key_attr;
  psa_mac_operation_t mac_op;

  ret = psa_crypto_init();

  // Set up attributes for a CMAC key
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_AES);
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_SIGN_HASH | PSA_KEY_USAGE_VERIFY_HASH);
  psa_set_key_algorithm(&key_attr, PSA_ALG_CMAC);

  // Import a volatile plain key for CMAC
  ret = psa_import_key(&key_attr, cmac_key, sizeof(cmac_key), &key_id);

  // Calculate the CMAC MAC of a message
  // Expected CMAC MAC: aa f3 d8 f1 de 56 40 c2 32 f5 b1 69 b9 c9 11 e6
  mac_op = psa_mac_operation_init();
  ret = psa_mac_sign_setup(&mac_op, key_id, PSA_ALG_CMAC);
  ret = psa_mac_update(&mac_op, cmac_msg, sizeof(cmac_msg));
  ret = psa_mac_sign_finish(&mac_op, mac_buf, sizeof(mac_buf), &mac_len);

  // Destroy a volatile plain key for CMAC
  ret = psa_destroy_key(key_id);

  // Set up attributes for a HMAC key
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_HMAC);
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_SIGN_HASH | PSA_KEY_USAGE_VERIFY_HASH);
  psa_set_key_algorithm(&key_attr, PSA_ALG_HMAC(PSA_ALG_SHA_256));

  // Import a volatile plain key for HMAC
  ret = psa_import_key(&key_attr, hmac_key, sizeof(hmac_key), &key_id);

  // Calculate the HMAC MAC of a message
  // Expected HMAC MAC:
  // fb 5b 26 22 9c 20 b7 ed 86 67 06 a2 fb fa e6 7e 3f 40 4b b6 ab e7 7f f4 50 63 a4 59 a4 29 24 a4
  mac_op = psa_mac_operation_init();
  ret = psa_mac_sign_setup(&mac_op, key_id, PSA_ALG_HMAC(PSA_ALG_SHA_256));
  ret = psa_mac_update(&mac_op, hmac_msg, sizeof(hmac_msg));
  ret = psa_mac_sign_finish(&mac_op, mac_buf, sizeof(mac_buf), &mac_len);

  // Destroy a volatile plain key for HMAC
  ret = psa_destroy_key(key_id);
}
```

PSA Crypto Platform Example

Click the `View Project Documentation` link to open the `readme.html` file.

The following table describes the implementation status of the PSA Crypto MAC platform example.

Table 7.19. PSA Crypto MAC Platform Example on Series 1 and Series 2 Devices

Algorithm	Series 1	Series 2 - VSE	Series 2 - HSE	Remark
HMAC	Y	Y	Y	• Hardware acceleration only on Series 2 devices. • HMAC streaming with wrapped key is not supported yet.
CMAC	Y	Y	Y	Series 1 devices do not support a 192-bit key.

Note: The single-part MAC functions are not supported yet.

7.5.3 Unauthenticated Ciphers

The unauthenticated cipher API is for use cases where the data integrity and authenticity is guaranteed by non-cryptographic means.

Algorithms

Table 7.20. Unauthenticated Cipher Algorithms

Algorithm	Mbed TLS	PSA Crypto
AES ECB	• `MBEDTLS_CIPHER_AES_128_ECB` • `MBEDTLS_CIPHER_AES_192_ECB` • `MBEDTLS_CIPHER_AES_256_ECB`	`PSA_ALG_ECB_NO_PADDING`
AES CBC	• `MBEDTLS_CIPHER_AES_128_CBC` • `MBEDTLS_CIPHER_AES_192_CBC` • `MBEDTLS_CIPHER_AES_256_CBC`	`PSA_ALG_CBC_NO_PADDING`
AES CFB	• `MBEDTLS_CIPHER_AES_128_CFB128` • `MBEDTLS_CIPHER_AES_192_CFB128` • `MBEDTLS_CIPHER_AES_256_CFB128`	`PSA_ALG_CFB`
AES CTR	• `MBEDTLS_CIPHER_AES_128_CTR` • `MBEDTLS_CIPHER_AES_192_CTR` • `MBEDTLS_CIPHER_AES_256_CTR`	`PSA_ALG_CTR`
CHACHA20	`MBEDTLS_CIPHER_CHACHA20`	`PSA_ALG_STREAM_CIPHER`

Key Attributes in PSA Crypto

Table 7.21. Key Attributes for Unauthenticated Cipher Algorithms

Algorithm	Key Type	Key Size in Bits	Key Usage Flag
PSA_ALG_ECB_NO_PADDING	PSA_KEY_TYPE_AES	• 128 (16-byte) • 192 (24-byte) • 256 (32-byte)	• PSA_KEY_USAGE_ENCRYPT • PSA_KEY_USAGE_DECRYPT
PSA_ALG_CBC_NO_PADDING			
PSA_ALG_CFB			
PSA_ALG_CTR			
PSA_ALG_STREAM_CIPHER	PSA_KEY_TYPE_CHACHA20	256 (32-byte)	

Security Software Components

Table 7.22. Security Software Components for Unauthenticated Cipher Algorithms

Algorithm	Security Software Components
PSA_ALG_ECB_NO_PADDING	PSA Crypto support for the ECB block cipher mode
PSA_ALG_CBC_NO_PADDING	PSA Crypto support for the CBC block cipher mode
PSA_ALG_CFB	PSA Crypto support for the CFB stream cipher mode
PSA_ALG_CTR	PSA Crypto support for the CTR stream cipher mode
PSA_ALG_STREAM_CIPHER	PSA Crypto support for Chacha20 & Mbed TLS Support for ChaChaPoly

Single-Part Functions

Table 7.23. Single-Part Unauthenticated Cipher Functions

Mbed TLS	PSA Crypto
Generic • int mbedtls_cipher_crypt(…) Algorithm specific • int mbedtls_aes_crypt_ecb(…) • int mbedtls_aes_crypt_cbc(…) • int mbedtls_aes_crypt_cfb128(…) • int mbedtls_aes_crypt_ctr(…) • int mbedtls_chacha20_crypt(…)	psa_status_t psa_cipher_encrypt(…)
Generic • int mbedtls_cipher_crypt(…) Algorithm specific • int mbedtls_aes_crypt_ecb(…) • int mbedtls_aes_crypt_cbc(…) • int mbedtls_aes_crypt_cfb128(…) • int mbedtls_aes_crypt_ctr(…) • int mbedtls_chacha20_crypt(…)	psa_status_t psa_cipher_decrypt(…)

Multi-Part Operations

Table 7.24. Multi-Part Unauthenticated Cipher Operations

Mbed TLS	PSA Crypto
Generic • `void mbedtls_cipher_init(…)` Algorithm specific • `void mbedtls_chacha20_init(…)`	`psa_cipher_operation_t psa_cipher_operation_init(void)`
Generic • `int mbedtls_cipher_setup(…)` • `int mbedtls_cipher_setkey(…)` Algorithm specific • `int mbedtls_chacha20_setkey(…)`	`psa_status_t psa_cipher_encrypt_setup(…)`
Generic • `int mbedtls_cipher_set_iv(…)` Algorithm specific • `int mbedtls_chacha20_starts(…)`	`psa_status_t psa_cipher_generate_iv(…)`
Generic • `int mbedtls_cipher_setup(…)` • `int mbedtls_cipher_setkey(…)` Algorithm specific • `int mbedtls_chacha20_setkey(…)`	`psa_status_t psa_cipher_decrypt_setup(…)`
Generic • `int mbedtls_cipher_set_iv(…)` Algorithm specific • `int mbedtls_chacha20_starts(…)`	`psa_status_t psa_cipher_set_iv(…)`
Generic • `int mbedtls_cipher_update(…)` Algorithm specific • `int mbedtls_chacha20_update(…)`	`psa_status_t psa_cipher_update(…)`
Generic • `int mbedtls_cipher_finish(…)`	`psa_status_t psa_cipher_finish(…)`
Generic • `void mbedtls_cipher_free(…)` Algorithm specific • `int mbedtls_chacha20_free(…)`	`psa_status_t psa_cipher_abort(…)`

Quick Reference Examples

AES ECB

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t aes_ecb_key[] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f
  };
  uint8_t plain_msg_buf[] = {
    0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff
  };
  uint8_t cipher_buf[16];
  size_t out_len;

  psa_status_t ret;
  psa_key_id_t key_id;
  psa_key_attributes_t key_attr;
  psa_cipher_operation_t cipher_op;

  ret = psa_crypto_init();

  // Set up attributes for a AES ECB key
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_AES);
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_ENCRYPT | PSA_KEY_USAGE_DECRYPT);
  psa_set_key_algorithm(&key_attr, PSA_ALG_ECB_NO_PADDING);

  // Import a volatile plain key for AES ECB
  ret = psa_import_key(&key_attr, aes_ecb_key, sizeof(aes_ecb_key), &key_id);

  // AES ECB encryption
  // Expected ciphertext: 69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a
  cipher_op = psa_cipher_operation_init();
  ret = psa_cipher_encrypt_setup(&cipher_op, key_id, PSA_ALG_ECB_NO_PADDING);
  ret = psa_cipher_update(&cipher_op,
                          plain_msg_buf, sizeof(plain_msg_buf),
                          cipher_buf, sizeof(cipher_buf),
                          &out_len);
  ret = psa_cipher_finish(&cipher_op,
                          cipher_buf - out_len,
                          sizeof(cipher_buf) - out_len,
                          &out_len);

  // AES ECB decryption
  cipher_op = psa_cipher_operation_init();
  ret = psa_cipher_decrypt_setup(&cipher_op, key_id, PSA_ALG_ECB_NO_PADDING);
  ret = psa_cipher_update(&cipher_op,
                          cipher_buf, sizeof(cipher_buf),
                          plain_msg_buf, sizeof(plain_msg_buf),
                          &out_len);
  ret = psa_cipher_finish(&cipher_op,
                          plain_msg_buf - out_len,
                          sizeof(plain_msg_buf) - out_len,
                          &out_len);

  // Destroy a volatile plain key for AES ECB
  ret = psa_destroy_key(key_id);
}
```

AES CBC

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t aes_cbc_key[] = {
    0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c
  };
  uint8_t iv_buf[] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f
  };
  uint8_t plain_msg_buf[] = {
    0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a
  };
  uint8_t cipher_buf[16];
  size_t out_len;

  psa_status_t ret;
  psa_key_id_t key_id;
  psa_key_attributes_t key_attr;
  psa_cipher_operation_t cipher_op;

  ret = psa_crypto_init();

  // Set up attributes for a AES CBC key
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_AES);
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_ENCRYPT | PSA_KEY_USAGE_DECRYPT);
  psa_set_key_algorithm(&key_attr, PSA_ALG_CBC_NO_PADDING);

  // Import a volatile plain key for AES CBC
  ret = psa_import_key(&key_attr, aes_cbc_key, sizeof(aes_cbc_key), &key_id);

  // AES CBC encryption
  // Expected ciphertext: 76 49 ab ac 81 19 b2 46 ce e9 8e 9b 12 e9 19 7d
  cipher_op = psa_cipher_operation_init();
  ret = psa_cipher_encrypt_setup(&cipher_op, key_id, PSA_ALG_CBC_NO_PADDING);
  ret = psa_cipher_set_iv(&cipher_op, iv_buf, sizeof(iv_buf));
  ret = psa_cipher_update(&cipher_op,
                          plain_msg_buf, sizeof(plain_msg_buf),
                          cipher_buf, sizeof(cipher_buf),
                          &out_len);
  ret = psa_cipher_finish(&cipher_op,
                          cipher_buf - out_len,
                          sizeof(cipher_buf) - out_len,
                          &out_len);

  // AES CBC decryption
  cipher_op = psa_cipher_operation_init();
  ret = psa_cipher_decrypt_setup(&cipher_op, key_id, PSA_ALG_CBC_NO_PADDING);
  ret = psa_cipher_set_iv(&cipher_op, iv_buf, sizeof(iv_buf));
  ret = psa_cipher_update(&cipher_op,
                          cipher_buf, sizeof(cipher_buf),
                          plain_msg_buf, sizeof(plain_msg_buf),
                          &out_len);
  ret = psa_cipher_finish(&cipher_op,
                          plain_msg_buf - out_len,
                          sizeof(plain_msg_buf) - out_len,
                          &out_len);

  // Destroy a volatile plain key for AES ECB
  ret = psa_destroy_key(key_id);
}
```

AES CFB

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t aes_cfb_key[] = {
    0xf0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
  };
  uint8_t iv_buf[16] = {0};
  uint8_t plain_msg_buf[16] = {0};
  uint8_t cipher_buf[16];
  size_t out_len;

  psa_status_t ret;
  psa_key_id_t key_id;
  psa_key_attributes_t key_attr;
  psa_cipher_operation_t cipher_op;

  ret = psa_crypto_init();

  // Set up attributes for a AES CFB key
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_AES);
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_ENCRYPT | PSA_KEY_USAGE_DECRYPT);
  psa_set_key_algorithm(&key_attr, PSA_ALG_CFB);

  // Import a volatile plain key for AES CFB
  ret = psa_import_key(&key_attr, aes_cfb_key, sizeof(aes_cfb_key), &key_id);

  // AES CFB encryption
  // Expected ciphertext: 97 00 14 d6 34 e2 b7 65 07 77 e8 e8 4d 03 cc d8
  cipher_op = psa_cipher_operation_init();
  ret = psa_cipher_encrypt_setup(&cipher_op, key_id, PSA_ALG_CFB);
  ret = psa_cipher_set_iv(&cipher_op, iv_buf, sizeof(iv_buf));
  ret = psa_cipher_update(&cipher_op,
                          plain_msg_buf, sizeof(plain_msg_buf),
                          cipher_buf, sizeof(cipher_buf),
                          &out_len);
  ret = psa_cipher_finish(&cipher_op,
                          cipher_buf - out_len,
                          sizeof(cipher_buf) - out_len,
                          &out_len);

  // AES CFB decryption
  cipher_op = psa_cipher_operation_init();
  ret = psa_cipher_decrypt_setup(&cipher_op, key_id, PSA_ALG_CFB);
  ret = psa_cipher_set_iv(&cipher_op, iv_buf, sizeof(iv_buf));
  ret = psa_cipher_update(&cipher_op,
                          cipher_buf, sizeof(cipher_buf),
                          plain_msg_buf, sizeof(plain_msg_buf),
                          &out_len);
  ret = psa_cipher_finish(&cipher_op,
                          plain_msg_buf - out_len,
                          sizeof(plain_msg_buf) - out_len,
                          &out_len);

  // Destroy a volatile plain key for AES CFB
  ret = psa_destroy_key(key_id);
}
```

AES CTR

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t aes_ctr_key[] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
  };
  uint8_t iv_buf[] = {
    0x22, 0x22, 0x1a, 0x70, 0x22, 0x22, 0x1a, 0x70, 0x22, 0x22, 0x1a, 0x70, 0x22, 0x22, 0x1a, 0x70
  };
  uint8_t plain_msg_buf[] = {
    0xd8, 0x65, 0xc9, 0xcd, 0xea, 0x33, 0x56, 0xc5, 0x48, 0x8e, 0x7b, 0xa1, 0x5e, 0x84, 0xf4, 0xeb,
    0xa3, 0xb8, 0x25, 0x9c, 0x05, 0x3f, 0x24, 0xce, 0x29, 0x67, 0x22, 0x1c, 0x00, 0x38, 0x84, 0xd7,
    0x9d, 0x4c, 0xa4, 0x87, 0x7f, 0xfa, 0x4b, 0xc6, 0x87, 0xc6, 0x67, 0xe5, 0x49, 0x5b, 0xcf, 0xec,
    0x12, 0xf4, 0x87, 0x17, 0x32, 0xaa, 0xe4, 0x5a, 0x11, 0x06, 0x76, 0x11, 0x3d, 0xf9, 0xe7, 0xda
  };
  uint8_t cipher_buf[64];
  size_t out_len;

  psa_status_t ret;
  psa_key_id_t key_id;
  psa_key_attributes_t key_attr;
  psa_cipher_operation_t cipher_op;

  ret = psa_crypto_init();

  // Set up attributes for a AES CTR key
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_AES);
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_ENCRYPT | PSA_KEY_USAGE_DECRYPT);
  psa_set_key_algorithm(&key_attr, PSA_ALG_CTR);

  // Import a volatile plain key for AES CTR
  ret = psa_import_key(&key_attr, aes_ctr_key, sizeof(aes_ctr_key), &key_id);

  // AES CTR encryption
  // Expected ciphertext:
  // b6 72 f2 af 6a cc 20 ae ee 1a d8 14 12 8c 31 8b 95 5b be 80 5b 38 92 49 89 76 00 f5 20 74 54 32
  // 7d 6d 0f b4 ac 0a 94 f3 7c a0 9e 45 05 33 98 fe a8 9c 20 0a d3 58 12 6d 9e 89 a4 05 26 5c 96 e7
  cipher_op = psa_cipher_operation_init();
  ret = psa_cipher_encrypt_setup(&cipher_op, key_id, PSA_ALG_CTR);
  ret = psa_cipher_set_iv(&cipher_op, iv_buf, sizeof(iv_buf));
  ret = psa_cipher_update(&cipher_op,
                          plain_msg_buf, sizeof(plain_msg_buf),
                          cipher_buf, sizeof(cipher_buf),
                          &out_len);
  ret = psa_cipher_finish(&cipher_op,
                          cipher_buf - out_len,
                          sizeof(cipher_buf) - out_len,
                          &out_len);

  // AES CTR decryption
  cipher_op = psa_cipher_operation_init();
  ret = psa_cipher_decrypt_setup(&cipher_op, key_id, PSA_ALG_CTR);
  ret = psa_cipher_set_iv(&cipher_op, iv_buf, sizeof(iv_buf));
  ret = psa_cipher_update(&cipher_op,
                          cipher_buf, sizeof(cipher_buf),
                          plain_msg_buf, sizeof(plain_msg_buf),
                          &out_len);
  ret = psa_cipher_finish(&cipher_op,
                          plain_msg_buf - out_len,
                          sizeof(plain_msg_buf) - out_len,
                          &out_len);

  // Destroy a volatile plain key for AES CTR
  ret = psa_destroy_key(key_id);
}
```

CHACHA20

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t chacha20_key[32] = {0};
  uint8_t iv_buf[12] = {0};
  uint8_t plain_msg_buf[64] = {0};
  uint8_t cipher_buf[64];
  size_t out_len;

  psa_status_t ret;
  psa_key_id_t key_id;
  psa_key_attributes_t key_attr;
  psa_cipher_operation_t cipher_op;

  ret = psa_crypto_init();

  // Set up attributes for a CHACHA20 key
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_CHACHA20);
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_ENCRYPT | PSA_KEY_USAGE_DECRYPT);
  psa_set_key_algorithm(&key_attr, PSA_ALG_STREAM_CIPHER);

  // Import a volatile plain key for CHACHA20
  ret = psa_import_key(&key_attr, chacha20_key, sizeof(chacha20_key), &key_id);

  // CHACHA20 encryption
  // Expected ciphertext:
  // 76 b8 e0 ad a0 f1 3d 90 40 5d 6a e5 53 86 bd 28 bd d2 19 b8 a0 8d ed 1a a8 36 ef cc 8b 77 0d c7
  // da 41 59 7c 51 57 48 8d 77 24 e0 3f b8 d8 4a 37 6a 43 b8 f4 15 18 a1 1c c3 87 b6 69 b2 ee 65 86
  cipher_op = psa_cipher_operation_init();
  ret = psa_cipher_encrypt_setup(&cipher_op, key_id, PSA_ALG_STREAM_CIPHER);
  ret = psa_cipher_set_iv(&cipher_op, iv_buf, sizeof(iv_buf));
  ret = psa_cipher_update(&cipher_op,
                          plain_msg_buf, sizeof(plain_msg_buf),
                          cipher_buf, sizeof(cipher_buf),
                          &out_len);
  ret = psa_cipher_finish(&cipher_op,
                          cipher_buf - out_len,
                          sizeof(cipher_buf) - out_len,
                          &out_len);

  // CHACHA20 decryption
  cipher_op = psa_cipher_operation_init();
  ret = psa_cipher_decrypt_setup(&cipher_op, key_id, PSA_ALG_STREAM_CIPHER);
  ret = psa_cipher_set_iv(&cipher_op, iv_buf, sizeof(iv_buf));
  ret = psa_cipher_update(&cipher_op,
                          cipher_buf, sizeof(cipher_buf),
                          plain_msg_buf, sizeof(plain_msg_buf),
                          &out_len);
  ret = psa_cipher_finish(&cipher_op,
                          plain_msg_buf - out_len,
                          sizeof(plain_msg_buf) - out_len,
                          &out_len);

  // Destroy a volatile plain key for CHACHA20
  ret = psa_destroy_key(key_id);
}
```

AES CTR with Built-in AES-128 Key (HSE only)

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t iv_buf[16] = {0};
  uint8_t plain_msg_buf[16] = {0};
  uint8_t cipher_buf[16];
  size_t out_len;

  psa_status_t ret;
  psa_cipher_operation_t cipher_op;

  ret = psa_crypto_init();

  // AES CTR encryption with built-in AES-128 key
  // Built-in AES-128 key: 81 a5 e2 1f a1 52 86 f1 df 44 5c 2c c1 20 fa 3f
  // Expected ciphertext: 66 d2 0f 99 65 3e a8 d0 83 05 a6 39 d4 4e 98 a6
  cipher_op = psa_cipher_operation_init();
  ret = psa_cipher_encrypt_setup(&cipher_op, SL_SE_BUILTIN_KEY_AES128_ID, PSA_ALG_CTR);
  ret = psa_cipher_set_iv(&cipher_op, iv_buf, sizeof(iv_buf));
  ret = psa_cipher_update(&cipher_op,
                          plain_msg_buf, sizeof(plain_msg_buf),
                          cipher_buf, sizeof(cipher_buf),
                          &out_len);
  ret = psa_cipher_finish(&cipher_op,
                          cipher_buf - out_len,
                          sizeof(cipher_buf) - out_len,
                          &out_len);

  // AES CTR decryption with built-in AES-128 key
  cipher_op = psa_cipher_operation_init();
  ret = psa_cipher_decrypt_setup(&cipher_op, SL_SE_BUILTIN_KEY_AES128_ID, PSA_ALG_CTR);
  ret = psa_cipher_set_iv(&cipher_op, iv_buf, sizeof(iv_buf));
  ret = psa_cipher_update(&cipher_op,
                          cipher_buf, sizeof(cipher_buf),
                          plain_msg_buf, sizeof(plain_msg_buf),
                          &out_len);
  ret = psa_cipher_finish(&cipher_op,
                          plain_msg_buf - out_len,
                          sizeof(plain_msg_buf) - out_len,
                          &out_len);
}
```

PSA Crypto Platform Example

Click the `View Project Documentation` link to open the `readme.html` file.

The following table describes the implementation status of the PSA Crypto cipher platform example.

Table 7.25. PSA Crypto Cipher Platform Example on Series 1 and Series 2 Devices

Algorithm	Series 1	Series 2 - VSE	Series 2 - HSE	Remark
AES ECB	Y	Y	Y	Series 1 devices do not support a 192-bit key.
AES CBC	Y	Y	Y	Series 1 devices do not support a 192-bit key.
AES CFB	Y	Y	Y	Series 1 devices do not support a 192-bit key.
AES CTR	Y	Y	Y	Series 1 devices do not support a 192-bit key.
CHACHA20	Y	Y	Y	Hardware acceleration only on HSE-SVH devices.
AES-128 Key	—	—	Y	—

Note: The single-part unauthenticated cipher functions are not supported yet.

7.5.4 Authenticated Encryption with Associated Data (AEAD)

The authenticated encryption with associated data (AEAD) is a form of encryption that simultaneously assures the confidentiality and authenticity of data.

Algorithms

Table 7.26. AEAD Algorithms

Algorithm	Mbed TLS	PSA Crypto
AES GCM	• `MBEDTLS_CIPHER_AES_128_GCM` • `MBEDTLS_CIPHER_AES_192_GCM` • `MBEDTLS_CIPHER_AES_256_GCM`	• `PSA_ALG_GCM` • `PSA_ALG_AEAD_WITH_SHORTENED_TAG(PSA_ALG_GCM, tag_length)`
AES CCM	• `MBEDTLS_CIPHER_AES_128_CCM` • `MBEDTLS_CIPHER_AES_192_CCM` • `MBEDTLS_CIPHER_AES_256_CCM`	• `PSA_ALG_CCM` • `PSA_ALG_AEAD_WITH_SHORTENED_TAG(PSA_ALG_CCM, tag_length)`
CHACHA20_POLY1305	`MBEDTLS_CIPHER_CHACHA20_POLY1305`	• `PSA_ALG_CHACHA20_POLY1305` • `PSA_ALG_AEAD_WITH_SHORTENED_TAG(PSA_ALG_CHACHA20_POLY1305, tag_length)`

Key Attributes in PSA Crypto

Table 7.27. Key Attributes for AEAD Algorithms

Algorithm	Key Type	Key Size in Bits	Key Usage Flag
PSA_ALG_GCM	PSA_KEY_TYPE_AES	• 128 (16-byte) • 192 (24-byte) • 256 (32-byte)	• PSA_KEY_USAGE_ENCRYPT • PSA_KEY_USAGE_DECRYPT
PSA_ALG_CCM			
PSA_ALG_CHACHA20_POLY1305	PSA_KEY_TYPE_CHACHA20	256 (32-byte)	

Security Software Components

Table 7.28. Security Software Components for AEAD Algorithms

Algorithm	Security Software Components
PSA_ALG_GCM	PSA Crypto support for the CCM algorithm
PSA_ALG_GCM	PSA Crypto support for the GCM algorithm
PSA_ALG_CHACHA20_POLY1305	PSA Crypto support for ChachaPoly

Single-Part Functions

Table 7.29. Single-Part AEAD Functions

Mbed TLS	PSA Crypto
Generic • int mbedtls_cipher_auth_encrypt(…) Algorithm specific • int mbedtls_ccm_encrypt_and_tag(…) • int mbedtls_gcm_crypt_and_tag(…) • int mbedtls_chachapoly_encrypt_and_tag(…)	psa_status_t psa_aead_encrypt(…)
Generic • int mbedtls_cipher_auth_decrypt(…) Algorithm specific • int mbedtls_ccm_auth_decrypt(…) • int mbedtls_gcm_auth_decrypt(…) • int mbedtls_chachapoly_auth_decrypt(…)	psa_status_t psa_aead_decrypt(…)

Multi-Part Operations

Table 7.30. Multi-Part AEAD Operations

Mbed TLS	PSA Crypto
Generic • `void mbedtls_cipher_init(…)` Algorithm specific • `void mbedtls_gcm_init(…)` • `void mbedtls_chachapoly_init(…)`	`psa_aead_operation_t psa_aead_operation_init(void)`
Generic • `int mbedtls_cipher_setup(…)` • `int mbedtls_cipher_setkey(…)` Algorithm specific • `int mbedtls_gcm_setkey(…)` • `int mbedtls_chachapoly_setkey(…)`	`psa_status_t psa_aead_encrypt_setup(…)`
Generic • `int mbedtls_cipher_set_iv(…)` Algorithm specific • `int mbedtls_gcm_starts(…)` • `int mbedtls_chachapoly_starts(…)`	`psa_status_t psa_aead_generate_nonce(…)`
—	`psa_status_t psa_aead_set_lengths(…)`
Generic • `int mbedtls_cipher_setup(…)` • `int mbedtls_cipher_setkey(…)` Algorithm specific • `int mbedtls_gcm_setkey(…)` • `int mbedtls_chachapoly_setkey(…)`	`psa_status_t psa_aead_decrypt_setup(…)`
Generic • `int mbedtls_cipher_set_iv(…)` Algorithm specific • `int mbedtls_gcm_starts(…)` • `int mbedtls_chachapoly_starts(…)`	`psa_status_t psa_aead_set_nonce(…)`
Generic • `int mbedtls_cipher_update_ad(…)` Algorithm specific • `int mbedtls_gcm_starts(…)` • `int mbedtls_chachapoly_update_aad(…)`	`psa_status_t psa_aead_update_ad(…)`
Generic • `int mbedtls_cipher_update(…)` Algorithm specific • `int mbedtls_gcm_update(…)` • `int mbedtls_chachapoly_update(…)`	`psa_status_t psa_aead_update(…)`

Mbed TLS	PSA Crypto
Generic • `int mbedtls_cipher_finish(…)` Algorithm specific • `int mbedtls_gcm_finish(…)` • `int mbedtls_chachapoly_finish(…)`	`psa_status_t psa_aead_finish(…)`
—	`psa_status_t psa_aead_verify(…)`
Generic • `void mbedtls_cipher_free(…)` Algorithm specific • `int mbedtls_gcm_free(…)` • `int mbedtls_chachapoly_free(…)`	`psa_status_t psa_aead_abort(…)`

Quick Reference Examples

AES CCM and AES GCM

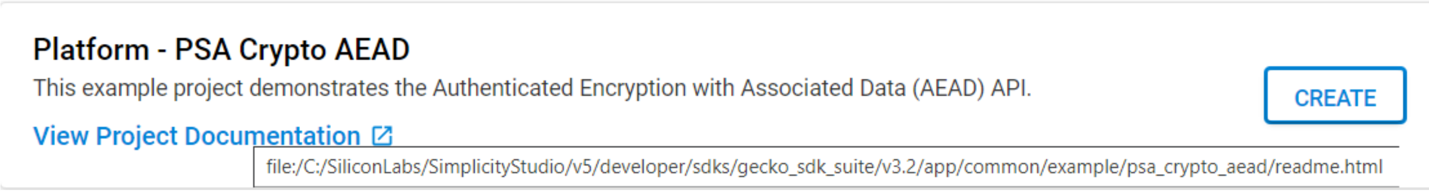
```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t key_buf[] = {
    0xea, 0x4f, 0x6f, 0x3c, 0x2f, 0xed, 0x2b, 0x9d, 0xd9, 0x70, 0x8c, 0x2e, 0x72, 0x1a, 0xe0, 0x0f
  };
  uint8_t nonce_buf[] = {0xf9, 0x75, 0x80, 0x9d, 0xdb, 0x51, 0x72, 0x38, 0x27, 0x45, 0x63, 0x4f};
  uint8_t ad_buf[] = {0x5c, 0x65, 0xd4, 0xf2, 0x61, 0xd2, 0xc5, 0x4f, 0xfe, 0x6a};
  uint8_t plain_msg_buf[] = {0x8d, 0x6c, 0x08, 0x44, 0x6c, 0xb1, 0x0d, 0x9a, 0x20, 0x75};
  uint8_t cipher_tag_buf[32];
  size_t out_len;

  psa_status_t ret;
  psa_key_id_t key_id;
  psa_key_attributes_t key_attr;

  ret = psa_crypto_init();

  // Set up attributes for a AES CCM key
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_AES);
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_ENCRYPT | PSA_KEY_USAGE_DECRYPT);
  psa_set_key_algorithm(&key_attr, PSA_ALG_CCM);

  // Import a volatile plain key for AES CCM
  ret = psa_import_key(&key_attr, key_buf, sizeof(key_buf), &key_id);

  // AES CCM encryption
  // Expected ciphertext: e2 2f 37 3b eb f6 4a 3e 9b 87 & Expected tag: 75 2b f9 db 34 dc 4d 43 3f 00 f5 5c 3f 53 0c 89
  ret = psa_aead_encrypt(key_id, PSA_ALG_CCM,
                         nonce_buf, sizeof(nonce_buf),
                         ad_buf, sizeof(ad_buf),
                         plain_msg_buf, sizeof(plain_msg_buf),
                         cipher_tag_buf, sizeof(cipher_tag_buf),
                         &out_len);

  // AES CCM decryption
  ret = psa_aead_decrypt(key_id, PSA_ALG_CCM,
                         nonce_buf, sizeof(nonce_buf),
                         ad_buf, sizeof(ad_buf),
                         cipher_tag_buf, out_len,
                         plain_msg_buf, sizeof(plain_msg_buf),
                         &out_len);
  // Destroy a volatile plain key for AES CCM
  ret = psa_destroy_key(key_id);

  // Set up attributes for a AES GCM key
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_AES);
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_ENCRYPT | PSA_KEY_USAGE_DECRYPT);
  psa_set_key_algorithm(&key_attr, PSA_ALG_GCM);

  // Import a volatile plain key for AES GCM
  ret = psa_import_key(&key_attr, key_buf, sizeof(key_buf), &key_id);

  // AES GCM encryption
  // Expected ciphertext: 0f 51 f7 a8 3c 5b 5a a7 96 b9 & Expected tag: 70 25 9c dd fe 8f 9a 15 a5 c5 eb 48 5a f5 78 fb
  ret = psa_aead_encrypt(key_id, PSA_ALG_GCM,
                         nonce_buf, sizeof(nonce_buf),
                         ad_buf, sizeof(ad_buf),
                         plain_msg_buf, sizeof(plain_msg_buf),
                         cipher_tag_buf, sizeof(cipher_tag_buf),
                         &out_len);

  // AES GCM decryption
  ret = psa_aead_decrypt(key_id, PSA_ALG_GCM,
                         nonce_buf, sizeof(nonce_buf),
                         ad_buf, sizeof(ad_buf),
                         cipher_tag_buf, out_len,
                         plain_msg_buf, sizeof(plain_msg_buf),
                         &out_len);
  // Destroy a volatile plain key for AES GCM
  ret = psa_destroy_key(key_id);
}
```

CHACHA20_POLY1305

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t key_buf[] = {
    0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e, 0x8f,
    0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98, 0x99, 0x9a, 0x9b, 0x9c, 0x9d, 0x9e, 0x9f
  };
  uint8_t nonce_buf[] = {0x07, 0x00, 0x00, 0x00, 0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47};
  uint8_t ad_buf[] = {0x50, 0x51, 0x52, 0x53, 0xc0, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5, 0xc6, 0xc7};
  uint8_t plain_msg_buf[] = {
    0x4c, 0x61, 0x64, 0x69, 0x65, 0x73, 0x20, 0x61, 0x6e, 0x64, 0x20, 0x47, 0x65, 0x6e, 0x74, 0x6c,
    0x65, 0x6d, 0x65, 0x6e, 0x20, 0x6f, 0x66, 0x20, 0x74, 0x68, 0x65, 0x20, 0x63, 0x6c, 0x61, 0x73,
    0x73, 0x20, 0x6f, 0x66, 0x20, 0x27, 0x39, 0x39, 0x3a, 0x20, 0x49, 0x66, 0x20, 0x49, 0x20, 0x63,
    0x6f, 0x75, 0x6c, 0x64, 0x20, 0x6f, 0x66, 0x66, 0x65, 0x72, 0x20, 0x79, 0x6f, 0x75, 0x20, 0x6f,
    0x6e, 0x6c, 0x79, 0x20, 0x6f, 0x6e, 0x65, 0x20, 0x74, 0x69, 0x70, 0x20, 0x66, 0x6f, 0x72, 0x20,
    0x74, 0x68, 0x65, 0x20, 0x66, 0x75, 0x74, 0x75, 0x72, 0x65, 0x2c, 0x20, 0x73, 0x75, 0x6e, 0x73,
    0x63, 0x72, 0x65, 0x65, 0x6e, 0x20, 0x77, 0x6f, 0x75, 0x6c, 0x64, 0x20, 0x62, 0x65, 0x20, 0x69,
    0x74, 0x2e
  };
  uint8_t cipher_tag_buf[130];
  size_t out_len;

  psa_status_t ret;
  psa_key_id_t key_id;
  psa_key_attributes_t key_attr;

  ret = psa_crypto_init();

  // Set up attributes for a CHACHA20_POLY1305 key
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_CHACHA20);
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_ENCRYPT | PSA_KEY_USAGE_DECRYPT);
  psa_set_key_algorithm(&key_attr, PSA_ALG_CHACHA20_POLY1305);

  // Import a volatile plain key for CHACHA20_POLY1305
  ret = psa_import_key(&key_attr, key_buf, sizeof(key_buf), &key_id);

  // CHACHA20_POLY1305 encryption
  // Expected ciphertext:
  // d3 1a 8d 34 64 8e 60 db 7b 86 af bc 53 ef 7e c2 a4 ad ed 51 29 6e 08 fe a9 e2 b5 a7 36 ee 62 d6
  // 3d be a4 5e 8c a9 67 12 82 fa fb 69 da 92 72 8b 1a 71 de 0a 9e 06 0b 29 05 d6 a5 b6 7e cd 3b 36
  // 92 dd bd 7f 2d 77 8b 8c 98 03 ae e3 28 09 1b 58 fa b3 24 e4 fa d6 75 94 55 85 80 8b 48 31 d7 bc
  // 3f f4 de f0 8e 4b 7a 9d e5 76 d2 65 86 ce c6 4b 61 16
  // Expected tag: 1a e1 0b 59 4f 09 e2 6a 7e 90 2e cb d0 60 06 91
  ret = psa_aead_encrypt(key_id,
                         PSA_ALG_CHACHA20_POLY1305,
                         nonce_buf,
                         sizeof(nonce_buf),
                         ad_buf,
                         sizeof(ad_buf),
                         plain_msg_buf,
                         sizeof(plain_msg_buf),
                         cipher_tag_buf,
                         sizeof(cipher_tag_buf),
                         &out_len);

  // CHACHA20_POLY1305 decryption
  ret = psa_aead_decrypt(key_id,
                         PSA_ALG_CHACHA20_POLY1305,
                         nonce_buf,
                         sizeof(nonce_buf),
                         ad_buf,
                         sizeof(ad_buf),
                         cipher_tag_buf,
                         out_len,
                         plain_msg_buf,
                         sizeof(plain_msg_buf),
                         &out_len);

  // Destroy a volatile plain key for CHACHA20_POLY1305
  ret = psa_destroy_key(key_id);
}
```

PSA Crypto Platform Example

Click the `View Project Documentation` link to open the `readme.html` file.



The following table describes the implementation status of the PSA Crypto AEAD platform example.

Table 7.31. PSA Crypto AEAD Platform Example on Series 1 and Series 2 Devices

Algorithm	Series 1	Series 2 - VSE	Series 2 - HSE	Remark
AES CCM	Y	Y	Y	Series 1 devices do not support a 192-bit key.
AES GCM	Y	Y	Y	Series 1 devices do not support a 192-bit key.
CHACHA20_POLY1305	Y	Y	Y	Hardware acceleration only on HSE-SVH devices.

Note: The multi-part AEAD operations are not supported yet.

7.5.5 Key Derivation

A Key Derivation Function (KDF) derives one or many secret keys from a secret value such as a master key, a password, or a passphrase using a pseudo-random function. The typical usage of a key derivation function is to use a secret, such as a password or an ECDH shared secret, and a salt to produce a symmetric key and initialization vector (IV) for use with AES.

Algorithms

Table 7.32. Key Derivation Algorithms

Algorithm	Mbed TLS	PSA Crypto
HKDF (SHA-1)	`MBEDTLS_MD_SHA1`	`PSA_ALG_HKDF(PSA_ALG_SHA_1)`
HKDF (SHA-2)	• `MBEDTLS_MD_SHA224` • `MBEDTLS_MD_SHA256` • `MBEDTLS_MD_SHA384` • `MBEDTLS_MD_SHA512`	• `PSA_ALG_HKDF(PSA_ALG_SHA_224)` • `PSA_ALG_HKDF(PSA_ALG_SHA_256)` • `PSA_ALG_HKDF(PSA_ALG_SHA_384)` • `PSA_ALG_HKDF(PSA_ALG_SHA_512)`
ECDH + HKDF	—	`PSA_ALG_KEY_AGREEMENT(PSA_ALG_ECDH, PSA_ALG_HKDF(hash_alg))`

Key Attributes in PSA Crypto

Table 7.33. Key Attributes for Key Derivation Algorithms

Algorithm	Key Type	Key Size in Bits	Key Usage Flag
• `PSA_ALG_HKDF(PSA_ALG_SHA_1)` • `PSA_ALG_HKDF(PSA_ALG_SHA_224)` • `PSA_ALG_HKDF(PSA_ALG_SHA_256)` • `PSA_ALG_HKDF(PSA_ALG_SHA_384)` • `PSA_ALG_HKDF(PSA_ALG_SHA_512)`	`PSA_KEY_TYPE_DERIVE`	Multiple of 8	`PSA_KEY_USAGE_DERIVE`

Security Software Components

Table 7.34. Security Software Components for Key Derivation Algorithms

Algorithm	Security Software Components
`PSA_ALG_HKDF(PSA_ALG_SHA_1)`	PSA Crypto support for HKDF & PSA Crypto support for SHA-1
`PSA_ALG_HKDF(PSA_ALG_SHA_224)`	PSA Crypto support for HKDF & PSA Crypto support for SHA-224
`PSA_ALG_HKDF(PSA_ALG_SHA_256)`	PSA Crypto support for HKDF & PSA Crypto support for SHA-256
`PSA_ALG_HKDF(PSA_ALG_SHA_384)`	PSA Crypto support for HKDF & PSA Crypto support for SHA-384
`PSA_ALG_HKDF(PSA_ALG_SHA_512)`	PSA Crypto support for HKDF & PSA Crypto support for SHA-512
`PSA_ALG_KEY_AGREEMENT(PSA_ALG_ECDH, PSA_ALG_HKDF(hash_alg))`	PSA Crypto support for ECDH & PSA Crypto support for HKDF & PSA Crypto support for SHA-X

Note:

- It should add the components for the derived key algorithm to implement the HKDF. For example, it requires the `PSA Crypto support for the CTR stream cipher mode` component if the derived key algorithm is `PSA_ALG_CTR`.
- Refer to 7.6.2 Key Agreement (ECDH) to add the ECDH components to derive the shared secret for the ECHD + HKDF algorithm (`PSA_ALG_KEY_AGREEMENT(PSA_ALG_ECDH, PSA_ALG_HKDF(hash_alg))`).

Single-Part Functions

Table 7.35. Single-Part Key Derivation Functions

Mbed TLS	PSA Crypto
`int mbedtls_hkdf(…)`	`psa_status_t sl_psa_key_derivation_single_shot(…)`

Note: The `sl_psa_key_derivation_single_shot(…)` is a **Silicon Labs custom API**. It can only use on HSE-SVH devices.

Multi-Part Operations

Table 7.36. Multi-Part Key Derivation Operations

Mbed TLS	PSA Crypto
—	`psa_key_derivation_operation_t psa_key_derivation_operation_init(void)`
—	`psa_status_t psa_key_derivation_setup(…)`
—	`psa_status_t psa_key_derivation_get_capacity(…)`
—	`psa_status_t psa_key_derivation_set_capacity(…)`
—	`psa_status_t psa_key_derivation_input_bytes(…)`
—	`psa_status_t psa_key_derivation_input_key(…)`
—	`psa_status_t psa_key_derivation_output_key(…)`
—	`psa_status_t psa_key_derivation_output_bytes(…)`
—	`psa_status_t psa_key_derivation_abort(…)`

Quick Reference Examples

HKDF (SHA-256)

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t hkdf_ikm[] = {
    0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b,
    0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b
  };
  uint8_t hkdf_salt[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c};
  uint8_t hkdf_info[] = {0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8, 0xf9};
  uint8_t key_buf[32];
  size_t key_len;

  psa_status_t ret;
  psa_key_id_t master_key_id;
  psa_key_id_t hkdf_key_id;
  psa_key_attributes_t key_attr;
  psa_key_derivation_operation_t kdf_op;

  ret = psa_crypto_init();

  // Set up attributes for a volatile master plain key
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_DERIVE);
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_DERIVE);
  psa_set_key_algorithm(&key_attr, PSA_ALG_HKDF(PSA_ALG_SHA_256));

  // Import a volatile master plain key
  ret = psa_import_key(&key_attr, hkdf_ikm, sizeof(hkdf_ikm), &master_key_id);

  // Set up attributes for a volatile derived plain key (exportable for verification)
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_AES);
  psa_set_key_bits(&key_attr, sizeof(key_buf) * 8);
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_EXPORT | PSA_KEY_USAGE_ENCRYPT | PSA_KEY_USAGE_DECRYPT);
  psa_set_key_algorithm(&key_attr, PSA_ALG_CTR);

  // Derive (HKDF SHA256) a volatile plain key for AES CTR
#if defined(SEMAILBOX_PRESENT) && (_SILICON_LABS_SECURITY_FEATURE == _SILICON_LABS_SECURITY_FEATURE_VAULT)
  // Silicon Labs custom API for Secure Vault High devices
  ret = sl_psa_key_derivation_single_shot(PSA_ALG_HKDF(PSA_ALG_SHA_256), master_key_id,
                                          hkdf_info, sizeof(hkdf_info),
                                          hkdf_salt, sizeof(hkdf_salt),
                                          0, &key_attr, &hkdf_key_id);
#else
  kdf_op = psa_key_derivation_operation_init();
  ret = psa_key_derivation_setup(&kdf_op, PSA_ALG_HKDF(PSA_ALG_SHA_256));
  ret = psa_key_derivation_set_capacity(&kdf_op, sizeof(key_buf));
  ret = psa_key_derivation_input_bytes(&kdf_op, PSA_KEY_DERIVATION_INPUT_SALT, hkdf_salt, sizeof(hkdf_salt));
  ret = psa_key_derivation_input_bytes(&kdf_op, PSA_KEY_DERIVATION_INPUT_INFO, hkdf_info, sizeof(hkdf_info));
  ret = psa_key_derivation_input_key(&kdf_op, PSA_KEY_DERIVATION_INPUT_SECRET, master_key_id);
  ret = psa_key_derivation_output_key(&key_attr, &kdf_op, &hkdf_key_id);
  ret = psa_key_derivation_abort(&kdf_op);
#endif

  // Export derived volatile plain key (expected value of HKDF SHA256):
  // 3c b2 5f 25 fa ac d5 7a 90 43 4f 64 d0 36 2f 2a 2d 2d 0a 90 cf 1a 5a 4c 5d b0 2d 56 ec c4 c5 bf
  ret = psa_export_key(hkdf_key_id, key_buf, sizeof(key_buf), &key_len);

  // Destroy the master and derived keys
  ret = psa_destroy_key(master_key_id);
  ret = psa_destroy_key(hkdf_key_id);
}
```

PSA Crypto Platform Example

Click the `View Project Documentation` link to open the `readme.html` file.

The following table describes the implementation status of the PSA Crypto KDF platform example.

Table 7.37. PSA Crypto KDF Platform Example on Series 1 and Series 2 Devices

Algorithm	Series 1	Series 2 - VSE	Series 2 - HSE	Remark
HKDF	Y	Y	Y	Hardware acceleration only on HSE-SVH devices with Silicon Labs custom API.
ECDH + HKDF	Y	Y	Y	Hardware acceleration (HKDF) only on HSE-SVH devices with Silicon Labs custom API.

Note: The ECDH + HKDF algorithm does not apply to the wrapped key.

7.6 Asymmetric Cryptographic Operation

7.6.1 Asymmetric Signature (ECDSA and EdDSA)

In modern cryptography, the Elliptic-Curve-based signatures (like ECDSA and EdDSA) are widely used because of shorter key lengths, shorter signature lengths, higher security levels (for the same key length), and better performance.

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a cryptographically secure digital signature scheme based on the Elliptic Curve Cryptography (ECC). The Edwards-curve Digital Signature Algorithm (EdDSA) is a fast digital signature algorithm, using elliptic curves in Edwards form (like Ed25519 and Ed448).

Algorithms

Table 7.38. Asymmetric Signature Algorithms

Algorithm	Mbed TLS	PSA Crypto
ECDSA (SHA-1)	—	`PSA_ALG_ECDSA(PSA_ALG_SHA_1)`
ECDSA (SHA-2)	—	• `PSA_ALG_ECDSA(PSA_ALG_SHA_224)` • `PSA_ALG_ECDSA(PSA_ALG_SHA_256)` • `PSA_ALG_ECDSA(PSA_ALG_SHA_384)` • `PSA_ALG_ECDSA(PSA_ALG_SHA_512)`
ECDSA (Any hash algorithm)	—	`PSA_ALG_ECDSA(PSA_ALG_ANY_HASH)`
ECDSA (No hashing)	—	`PSA_ALG_ECDSA_ANY`
EdDSA	—	`PSA_ALG_PURE_EDDSA`

Note: The hash-and-sign algorithms (`PSA_ALG_ECDSA(hash_alg)` and `PSA_ALG_ECDSA(PSA_ALG_ANY_HASH)`) include the hashing step for ECDSA.

Key Attributes in PSA Crypto

Table 7.39. Key Attributes for Asymmetric Signature Algorithms

Algorithm	Key Type	Key Size in Bits	Key Usage Flag
PSA_ALG_ECDSA_ANY	PSA_ECC_FAMILY_SECP_R1	• secp192r1 : 192 • secp224r1 : 224 • secp256r1 : 256 • secp384r1 : 384 • secp521r1 : 521	• PSA_KEY_USAGE_SIGN_HASH • PSA_KEY_USAGE_VERIFY_HASH
	PSA_ECC_FAMILY_SECP_K1	secp256k1 : 256	
• PSA_ALG_ECDSA(PSA_ALG_SHA_1) • PSA_ALG_ECDSA(PSA_ALG_SHA_224) • PSA_ALG_ECDSA(PSA_ALG_SHA_256) • PSA_ALG_ECDSA(PSA_ALG_SHA_384) • PSA_ALG_ECDSA(PSA_ALG_SHA_512) • PSA_ALG_ECDSA(PSA_ALG_ANY_HASH)	PSA_ECC_FAMILY_SECP_R1	• secp192r1 : 192 • secp224r1 : 224 • secp256r1 : 256 • secp384r1 : 384 • secp521r1 : 521	• PSA_KEY_USAGE_SIGN_MESSAGE • PSA_KEY_USAGE_VERIFY_MESSAGE
	PSA_ECC_FAMILY_SECP_K1	secp256k1 : 256	
PSA_ALG_PURE_EDDSA	PSA_ECC_FAMILY_TWISTED_EDWARDS	Ed25519 : 255	• PSA_KEY_USAGE_SIGN_MESSAGE • PSA_KEY_USAGE_VERIFY_MESSAGE

Security Software Components

Table 7.40. Security Software Components for Asymmetric Signature Algorithms

Algorithm	Security Software Components
PSA_ALG_ECDSA_ANY	PSA Crypto support for ECDSA
PSA_ALG_ECDSA(PSA_ALG_SHA_1)	PSA Crypto support for ECDSA & PSA Crypto support for SHA-1
PSA_ALG_ECDSA(PSA_ALG_SHA_224)	PSA Crypto support for ECDSA & PSA Crypto support for SHA-224
PSA_ALG_ECDSA(PSA_ALG_SHA_256)	PSA Crypto support for ECDSA & PSA Crypto support for SHA-256
PSA_ALG_ECDSA(PSA_ALG_SHA_384)	PSA Crypto support for ECDSA & PSA Crypto support for SHA-384
PSA_ALG_ECDSA(PSA_ALG_SHA_512)	PSA Crypto support for ECDSA & PSA Crypto support for SHA-512
PSA_ALG_ECDSA(PSA_ALG_ANY_HASH)	PSA Crypto support for ECDSA & PSA Crypto support for SHA-X
PSA_ALG_PURE_EDDSA	PSA Crypto support for EdDSA

Table 7.41. Security Software Components for ECDSA PSA_ECC_FAMILY_SECP_R1 Curves

PSA_ECC_FAMILY_SECP_R1	Security Software Components
SECP192R1	PSA Crypto support for the secp192r1 ECC curve
SECP224R1	PSA Crypto support for the secp224r1 ECC curve
SECP256R1	PSA Crypto support for the secp256r1 ECC curve
SECP384R1	PSA Crypto support for the secp384r1 ECC curve
SECP521R1	PSA Crypto support for the secp521r1 ECC curve

Table 7.42. Security Software Components for ECDSA PSA_ECC_FAMILY_SECP_K1 Curves

PSA_ECC_FAMILY_SECP_K1	Security Software Components
`SECP256K1`	PSA Crypto support for the secp256k1 ECC curve

Functions

Table 7.43. Asymmetric Signature Functions

Mbed TLS	PSA Crypto
Hash-and-sign —	Hash-and-sign • `psa_status_t psa_sign_message(…)` • `psa_status_t psa_verify_message(…)`
Precomputed hash • `int mbedtls_ecdsa_write_signature(…)` • `int mbedtls_ecdsa_read_signature(…)`	Precomputed hash • `psa_status_t psa_sign_hash(…)` • `psa_status_t psa_verify_hash(…)`

Quick Reference Examples

ECDSA on SECP256R1

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t hash_data[] = {
    0x24, 0x8d, 0x6a, 0x61, 0xd2, 0x06, 0x38, 0xb8, 0xe5, 0xc0, 0x26, 0x93, 0x0c, 0x3e, 0x60, 0x39,
    0xa3, 0x3c, 0xe4, 0x59, 0x64, 0xff, 0x21, 0x67, 0xf6, 0xec, 0xed, 0xd4, 0x19, 0xdb, 0x06, 0xc1
  };
  uint8_t public_key[65];       // Uncompressed point format
  size_t pubkey_len;
  uint8_t signature_buf[64];
  size_t signature_len;

  psa_status_t ret;
  psa_key_id_t key_id;
  psa_key_attributes_t key_attr;

  ret = psa_crypto_init();

  // Set up attributes for a volatile private plain key (SECP256R1)
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_SECP_R1));
  psa_set_key_bits(&key_attr, 256);
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_SIGN_HASH | PSA_KEY_USAGE_VERIFY_HASH);
  psa_set_key_algorithm(&key_attr, PSA_ALG_ECDSA_ANY);

  // Generate a random volatile private plain key
  ret = psa_generate_key(&key_attr, &key_id);

  // Sign a hash with a volatile private plain key
  ret = psa_sign_hash(key_id,
                      PSA_ALG_ECDSA_ANY,
                      hash_data,
                      sizeof(hash_data),
                      signature_buf,
                      sizeof(signature_buf),
                      &signature_len);

  // Verify a signature with a volatile private plain key
  ret = psa_verify_hash(key_id,
                        PSA_ALG_ECDSA_ANY,
                        hash_data,
                        sizeof(hash_data),
                        signature_buf,
                        signature_len);

  // Export a public key from a volatile private plain key
  ret = psa_export_public_key(key_id,
                              public_key,
                              sizeof(public_key),
                              &pubkey_len);

  // Destroy a volatile private plain key
  ret = psa_destroy_key(key_id);

  // Set up attributes for a public key (SECP256R1)
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_ECC_PUBLIC_KEY(PSA_ECC_FAMILY_SECP_R1));
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_VERIFY_HASH);
  psa_set_key_algorithm(&key_attr, PSA_ALG_ECDSA_ANY);

  // Import a public key
  ret = psa_import_key(&key_attr, public_key, sizeof(public_key), &key_id);

  // Verify a signature with a public key
  ret = psa_verify_hash(key_id,
                        PSA_ALG_ECDSA_ANY,
                        hash_data,
                        sizeof(hash_data),
                        signature_buf,
                        signature_len);

  // Destroy a public key
  ret = psa_destroy_key(key_id);
}
```

ECDSA with Built-in Private Device Key (HSE-SVH only)

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
#if (_SILICON_LABS_SECURITY_FEATURE == _SILICON_LABS_SECURITY_FEATURE_VAULT)
  uint8_t hash_data[] = {
    0x24, 0x8d, 0x6a, 0x61, 0xd2, 0x06, 0x38, 0xb8, 0xe5, 0xc0, 0x26, 0x93, 0x0c, 0x3e, 0x60, 0x39,
    0xa3, 0x3c, 0xe4, 0x59, 0x64, 0xff, 0x21, 0x67, 0xf6, 0xec, 0xed, 0xd4, 0x19, 0xdb, 0x06, 0xc1
  };
  uint8_t public_key[65];       // Uncompressed point format
  size_t pubkey_len;
  uint8_t signature_buf[64];
  size_t signature_len;

  psa_status_t ret;
  psa_key_id_t key_id;
  psa_key_attributes_t key_attr;

  ret = psa_crypto_init();

  // Sign a hash with a built-in private device key
  ret = psa_sign_hash(SL_SE_BUILTIN_KEY_APPLICATION_ATTESTATION_ID,
                      PSA_ALG_ECDSA_ANY,
                      hash_data,
                      sizeof(hash_data),
                      signature_buf,
                      sizeof(signature_buf),
                      &signature_len);

  // Verify a signature with a built-in private device key
  ret = psa_verify_hash(SL_SE_BUILTIN_KEY_APPLICATION_ATTESTATION_ID,
                        PSA_ALG_ECDSA_ANY,
                        hash_data,
                        sizeof(hash_data),
                        signature_buf,
                        signature_len);

  // Export a built-in public device key
  ret = psa_export_public_key(SL_SE_BUILTIN_KEY_APPLICATION_ATTESTATION_ID,
                              public_key,
                              sizeof(public_key),
                              &pubkey_len);

  // Set up attributes for a public device key
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_ECC_PUBLIC_KEY(PSA_ECC_FAMILY_SECP_R1));
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_VERIFY_HASH);
  psa_set_key_algorithm(&key_attr, PSA_ALG_ECDSA_ANY);

  // Import a public device key
  ret = psa_import_key(&key_attr, public_key, sizeof(public_key), &key_id);

  // Verify a signature with a public device key
  ret = psa_verify_hash(key_id,
                        PSA_ALG_ECDSA_ANY,
                        hash_data,
                        sizeof(hash_data),
                        signature_buf,
                        signature_len);

  // Destroy a public device key
  ret = psa_destroy_key(key_id);
#endif
}
```

EdDSA on Ed25519 (HSE-SVH only)

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
#if (_SILICON_LABS_SECURITY_FEATURE == _SILICON_LABS_SECURITY_FEATURE_VAULT)
  uint8_t eddsa_msg[] = {
    0xdd, 0xaf, 0x35, 0xa1, 0x93, 0x61, 0x7a, 0xba, 0xcc, 0x41, 0x73, 0x49, 0xae, 0x20, 0x41, 0x31,
    0x12, 0xe6, 0xfa, 0x4e, 0x89, 0xa9, 0x7e, 0xa2, 0x0a, 0x9e, 0xee, 0xe6, 0x4b, 0x55, 0xd3, 0x9a,
    0x21, 0x92, 0x99, 0x2a, 0x27, 0x4f, 0xc1, 0xa8, 0x36, 0xba, 0x3c, 0x23, 0xa3, 0xfe, 0xeb, 0xbd,
    0x45, 0x4d, 0x44, 0x23, 0x64, 0x3c, 0xe8, 0x0e, 0x2a, 0x9a, 0xc9, 0x4f, 0xa5, 0x4c, 0xa4, 0x9f
  };
  uint8_t ed25519_private[] = {
    0x83, 0x3f, 0xe6, 0x24, 0x09, 0x23, 0x7b, 0x9d, 0x62, 0xec, 0x77, 0x58, 0x75, 0x20, 0x91, 0x1e,
    0x9a, 0x75, 0x9c, 0xec, 0x1d, 0x19, 0x75, 0x5b, 0x7d, 0xa9, 0x01, 0xb9, 0x6d, 0xca, 0x3d, 0x42
  };
  uint8_t ed25519_public[] = {
    0xec, 0x17, 0x2b, 0x93, 0xad, 0x5e, 0x56, 0x3b, 0xf4, 0x93, 0x2c, 0x70, 0xe1, 0x24, 0x50, 0x34,
    0xc3, 0x54, 0x67, 0xef, 0x2e, 0xfd, 0x4d, 0x64, 0xeb, 0xf8, 0x19, 0x68, 0x34, 0x67, 0xe2, 0xbf
  };
  uint8_t signature_buf[64];
  size_t signature_len;

  psa_status_t ret;
  psa_key_id_t key_id;
  psa_key_attributes_t key_attr;

  ret = psa_crypto_init();

  // Set up attributes for a volatile private plain key (Ed25519)
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_TWISTED_EDWARDS));
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_SIGN_MESSAGE | PSA_KEY_USAGE_VERIFY_MESSAGE);
  psa_set_key_algorithm(&key_attr, PSA_ALG_PURE_EDDSA);

  // Import a volatile private plain key
  ret = psa_import_key(&key_attr, ed25519_private, sizeof(ed25519_private), &key_id);

  // Hash-and-Sign a message with a volatile private plain key (expected EdDSA signature):
  // dc 2a 44 59 e7 36 96 33 a5 2b 1b f2 77 83 9a 00 20 10 09 a3 ef bf 3e cb 69 be a2 18 6c 26 b5 89
  // 09 35 1f c9 ac 90 b3 ec fd fb c7 c6 64 31 e0 30 3d ca 17 9c 13 8a c1 7a d9 be f1 17 73 31 a7 04
  ret = psa_sign_message(key_id,
                         PSA_ALG_PURE_EDDSA,
                         eddsa_msg,
                         sizeof(eddsa_msg),
                         signature_buf,
                         sizeof(signature_buf),
                         &signature_len);

  // Destroy a volatile private plain key
  ret = psa_destroy_key(key_id);

  // Set up attributes for a public key (Ed25519)
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_ECC_PUBLIC_KEY(PSA_ECC_FAMILY_TWISTED_EDWARDS));
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_VERIFY_MESSAGE);
  psa_set_key_algorithm(&key_attr, PSA_ALG_PURE_EDDSA);

  // Import a public key
  ret = psa_import_key(&key_attr, ed25519_public, sizeof(ed25519_public), &key_id);

  // Hash a message and verify the signature with a public key
  ret = psa_verify_message(key_id,
                           PSA_ALG_PURE_EDDSA,
                           eddsa_msg,
                           sizeof(eddsa_msg),
                           signature_buf,
                           signature_len);

  // Destroy a public key
  ret = psa_destroy_key(key_id);
#endif
}
```

PSA Crypto Platform Example

Click the `View Project Documentation` link to open the `readme.html` file.

The following table describes the implementation status of the PSA Crypto DSA platform example.

Table 7.44. PSA Crypto DSA Platform Example on Series 1 and Series 2 Devices

ECC Key	Series 1	Series 2 - VSE	Series 2 - HSE	Remark
SECP192R1	Y	Y	Y	—
SECP256R1	Y	Y	Y	—
SECP384R1	Y	Y	Y	Hardware acceleration only on HSE-SVH devices.
SECP521R1	Y	Y	Y	Hardware acceleration only on HSE-SVH devices.
Ed25519	—	—	Y	Only on HSE-SVH devices with hardware acceleration.
Public Sign Key	—	—	Y	—
Public Command Key	—	—	Y	—
Private Device Key	—	—	Y	Only on HSE-SVH devices.

Note:
- This example does not include SECP224R1 and SECP256K1.
- The PSA Crypto does not yet support software fallback on the Ed25519.

7.6.2 Key Agreement (ECDH)

The Elliptic Curve Diffie-Hellman (ECDH) is an anonymous key agreement protocol that allows two parties, each having an elliptic-curve private-public key pair, to establish a shared secret over an insecure channel.

Algorithms

Table 7.45. Key Agreement Algorithms

Algorithm	Mbed TLS	PSA Crypto
ECDH	—	`PSA_ALG_ECDH`
ECDH and HKDF	—	`PSA_ALG_KEY_AGREEMENT(PSA_ALG_ECDH, PSA_ALG_HKDF(hash_alg))`

Key Attributes in PSA Crypto

Table 7.46. Key Attributes for Key Agreement Algorithms

Algorithm	Key Type	Key Size in Bits	Key Usage Flag
PSA_ALG_ECDH	PSA_ECC_FAMILY_SECP_R1	• secp192r1 : 192 • secp224r1 : 224 • secp256r1 : 256 • secp384r1 : 384 • secp521r1 : 521	PSA_KEY_USAGE_DERIVE
	PSA_ECC_FAMILY_SECP_K1	secp256k1 : 256	
	PSA_ECC_FAMILY_MONTGOMERY	• Curve25519 : 255 • Curve448 : 448	

Security Software Components

Table 7.47. Security Software Components for Key Agreement Algorithms

Algorithm	Security Software Components
PSA_ALG_ECDH	PSA Crypto support for ECDH

Table 7.48. Security Software Components for ECDH PSA_ECC_FAMILY_SECP_R1 Curves

PSA_ECC_FAMILY_SECP_R1	Security Software Components
SECP192R1	PSA Crypto support for the secp192r1 ECC curve
SECP224R1	PSA Crypto support for the secp224r1 ECC curve
SECP256R1	PSA Crypto support for the secp256r1 ECC curve
SECP384R1	PSA Crypto support for the secp384r1 ECC curve
SECP521R1	PSA Crypto support for the secp521r1 ECC curve

Table 7.49. Security Software Components for ECDH PSA_ECC_FAMILY_SECP_K1 Curves

PSA_ECC_FAMILY_SECP_K1	Security Software Components
SECP256K1	PSA Crypto support for the secp256k1 ECC curve

Table 7.50. Security Software Components for ECDH PSA_ECC_FAMILY_MONTGOMERY Curves

PSA_ECC_FAMILY_MONTGOMERY	Security Software Components
CURVE25519	PSA Crypto support for the Curve25519 ECC curve
CURVE448	PSA Crypto support for the Montgomery 448 ECC curve

Functions

Table 7.51. Key Agreement Functions

Mbed TLS	PSA Crypto
`int mbedtls_ecdh_compute_shared(…)`	`psa_status_t psa_raw_key_agreement(…)`
—	`psa_status_t psa_key_derivation_key_agreement(…)`

Note: For the `psa_key_derivation_key_agreement(…)` function, refer to the PSA Crypto KDF platform example for details.

Quick Reference Examples

ECDH on SECP256R1

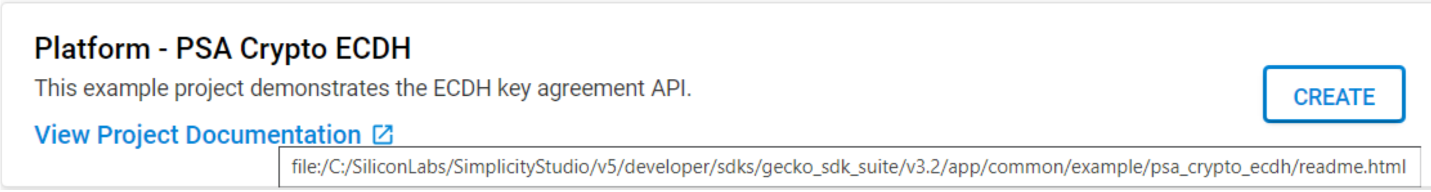
```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t client_private_key[] = {
    0xc8, 0x8f, 0x01, 0xf5, 0x10, 0xd9, 0xac, 0x3f, 0x70, 0xa2, 0x92, 0xda, 0xa2, 0x31, 0x6d, 0xe5,
    0x44, 0xe9, 0xaa, 0xb8, 0xaf, 0xe8, 0x40, 0x49, 0xc6, 0x2a, 0x9c, 0x57, 0x86, 0x2d, 0x14, 0x33
  };
  uint8_t client_public_key[] = { // Uncompressed point format
    0x04, 0xda, 0xd0, 0xb6, 0x53, 0x94, 0x22, 0x1c, 0xf9, 0xb0, 0x51, 0xe1, 0xfe, 0xca, 0x57, 0x87, 0xd0,
    0x98, 0xdf, 0xe6, 0x37, 0xfc, 0x90, 0xb9, 0xef, 0x94, 0x5d, 0x0c, 0x37, 0x72, 0x58, 0x11, 0x80,
    0x52, 0x71, 0xa0, 0x46, 0x1c, 0xdb, 0x82, 0x52, 0xd6, 0x1f, 0x1c, 0x45, 0x6f, 0xa3, 0xe5, 0x9a,
    0xb1, 0xf4, 0x5b, 0x33, 0xac, 0xcf, 0x5f, 0x58, 0x38, 0x9e, 0x05, 0x77, 0xb8, 0x99, 0x0b, 0xb3
  };
  uint8_t server_private_key[] = {
    0xc6, 0xef, 0x9c, 0x5d, 0x78, 0xae, 0x01, 0x2a, 0x01, 0x11, 0x64, 0xac, 0xb3, 0x97, 0xce, 0x20,
    0x88, 0x68, 0x5d, 0x8f, 0x06, 0xbf, 0x9b, 0xe0, 0xb2, 0x83, 0xab, 0x46, 0x47, 0x6b, 0xee, 0x53
  };
  uint8_t server_public_key[] = { // Uncompressed point format
    0x04, 0xd1, 0x2d, 0xfb, 0x52, 0x89, 0xc8, 0xd4, 0xf8, 0x12, 0x08, 0xb7, 0x02, 0x70, 0x39, 0x8c, 0x34,
    0x22, 0x96, 0x97, 0x0a, 0x0b, 0xcc, 0xb7, 0x4c, 0x73, 0x6f, 0xc7, 0x55, 0x44, 0x94, 0xbf, 0x63,
    0x56, 0xfb, 0xf3, 0xca, 0x36, 0x6c, 0xc2, 0x3e, 0x81, 0x57, 0x85, 0x4c, 0x13, 0xc5, 0x8d, 0x6a,
    0xac, 0x23, 0xf0, 0x46, 0xad, 0xa3, 0x0f, 0x83, 0x53, 0xe7, 0x4f, 0x33, 0x03, 0x98, 0x72, 0xab
  };
  // Expected shared secret:
  // d6 84 0f 6b 42 f6 ed af d1 31 16 e0 e1 25 65 20 2f ef 8e 9e ce 7d ce 03 81 24 64 d0 4b 94 42 de
  uint8_t shared_secret_buf[32];
  size_t shared_secret_len;

  psa_status_t ret;
  psa_key_id_t key_id;
  psa_key_attributes_t key_attr;

  ret = psa_crypto_init();

  // Set up attributes for a volatile client private plain key (SECP256R1)
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_SECP_R1));
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_DERIVE);
  psa_set_key_algorithm(&key_attr, PSA_ALG_ECDH);

  // Import a volatile client private plain key
  ret = psa_import_key(&key_attr, client_private_key, sizeof(client_private_key), &key_id);

  // Perform a key agreement with server public key
  ret = psa_raw_key_agreement(PSA_ALG_ECDH,
                              key_id,
                              server_public_key,
                              sizeof(server_public_key),
                              shared_secret_buf,
                              sizeof(shared_secret_buf),
                              &shared_secret_len);
  // Destroy the client private key
  ret = psa_destroy_key(key_id);

  // Set up attributes for a volatile server private plain key (SECP256R1)
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_SECP_R1));
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_DERIVE);
  psa_set_key_algorithm(&key_attr, PSA_ALG_ECDH);

  // Import a volatile server private plain key
  ret = psa_import_key(&key_attr, server_private_key, sizeof(server_private_key), &key_id);

  // Perform a key agreement with client public key
  ret = psa_raw_key_agreement(PSA_ALG_ECDH,
                              key_id,
                              client_public_key,
                              sizeof(client_public_key),
                              shared_secret_buf,
                              sizeof(shared_secret_buf),
                              &shared_secret_len);
  // Destroy the server private key
  ret = psa_destroy_key(key_id);
}
```

ECDH on CURVE25519

```
#include <stdbool.h>
#include "psa/crypto.h"

void app_process_action(void)
{
  uint8_t client_private_key[] = {
    0xB0, 0x76, 0x51, 0xEA, 0x20, 0xF0, 0x28, 0xA8,0x16, 0xEE, 0x01, 0xB0, 0xD1, 0x06, 0x2A, 0x7C,
    0x81, 0x58, 0xE8, 0x84, 0xE9, 0xBC, 0xC6, 0x1C, 0x5D, 0xAB, 0xDB, 0x4E, 0x38, 0x2F, 0x96, 0x69,
  };
  uint8_t client_public_key[] = {
    0x87, 0xD8, 0x6B, 0xDA, 0xAC, 0x38, 0x3C, 0x85, 0xA6, 0xBC, 0xF8, 0xFC, 0xC6, 0x26, 0xD6, 0x14,
    0x36, 0xE4, 0x8F, 0xDB, 0xFA, 0x5A, 0x45, 0xFE, 0x0C, 0x9E, 0xA8, 0x4B, 0x35, 0x3E, 0xF1, 0x37,
  };
  uint8_t server_private_key[] = {
    0x98, 0x2E, 0xB6, 0x7D, 0x0A, 0x01, 0x57, 0x90, 0xE1, 0x45, 0xF3, 0x67, 0xF6, 0xDA, 0xA6, 0x44,
    0x2C, 0x87, 0xC0, 0xED, 0x3C, 0x36, 0x71, 0xA6, 0x89, 0xC7, 0x49, 0xAC, 0x0D, 0xFE, 0x43, 0x6E,
  };
  uint8_t server_public_key[] = {
    0x0C, 0x04, 0x10, 0x5B, 0xE8, 0x7C, 0xAB, 0x37, 0x21, 0x15, 0x7A, 0x8D, 0x49, 0x85, 0x8C, 0x7A,
    0x9F, 0xC1, 0x46, 0xDA, 0xCC, 0x96, 0xEF, 0x6E, 0xD4, 0xDA, 0x71, 0xBF, 0xED, 0x32, 0x0D, 0x76,
  };
  // Expected shared secret:
  // F2 E6 0E 1C B7 64 BC 48 F2 9D BB 12 FB 12 17 31 32 1D 79 AF 0A 9F AB AD 34 05 A2 07 39 9C 5F 15
  uint8_t shared_secret_buf[32];
  size_t shared_secret_len;

  psa_status_t ret;
  psa_key_id_t key_id;
  psa_key_attributes_t key_attr;

  ret = psa_crypto_init();

  // Set up attributes for a volatile client private plain key (CURVE25519)
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_MONTGOMERY));
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_DERIVE);
  psa_set_key_algorithm(&key_attr, PSA_ALG_ECDH);

  // Import a volatile client private plain key
  ret = psa_import_key(&key_attr, client_private_key, sizeof(client_private_key), &key_id);

  // Perform a key agreement with server public key
  ret = psa_raw_key_agreement(PSA_ALG_ECDH,
                              key_id,
                              server_public_key,
                              sizeof(server_public_key),
                              shared_secret_buf,
                              sizeof(shared_secret_buf),
                              &shared_secret_len);

  // Destroy the client private key
  ret = psa_destroy_key(key_id);

  // Set up attributes for a volatile server private plain key (CURVE25519)
  key_attr = psa_key_attributes_init();
  psa_set_key_type(&key_attr, PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_MONTGOMERY));
  psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_DERIVE);
  psa_set_key_algorithm(&key_attr, PSA_ALG_ECDH);

  // Import a volatile server private plain key
  ret = psa_import_key(&key_attr, server_private_key, sizeof(server_private_key), &key_id);

  // Perform a key agreement with client public key
  ret = psa_raw_key_agreement(PSA_ALG_ECDH,
                              key_id,
                              client_public_key,
                              sizeof(client_public_key),
                              shared_secret_buf,
                              sizeof(shared_secret_buf),
                              &shared_secret_len);

  // Destroy the server private key
  ret = psa_destroy_key(key_id);
}
```

PSA Crypto Platform Example

Click the `View Project Documentation` link to open the `readme.html` file.



The following table describes the implementation status of the PSA Crypto ECDH platform example.

Table 7.52. PSA Crypto ECDH Platform Example on Series 1 and Series 2 Devices

ECC Key	Series 1	Series 2 - VSE	Series 2 - HSE	Remark
SECP192R1	Y	Y	Y	—
SECP256R1	Y	Y	Y	—
SECP384R1	Y	Y	Y	Hardware acceleration only on HSE-SVH devices.
SECP521R1	Y	Y	Y	Hardware acceleration only on HSE-SVH devices.
CURVE25519	Y	Y	Y	Hardware acceleration only on HSE-SVH devices.
CURVE448	—	—	Y	Only on HSE-SVH devices with hardware acceleration.

Note:
- This example does not include SECP224R1 and SECP256K1.
- The PSA Crypto does not yet support software fallback on the CURVE448.

7.6.3 X.509 Certificate

An X.509 certificate is a digital certificate that uses the widely accepted international X.509 public key infrastructure (PKI) standard to verify that a public key belongs to the user, computer, or service identity contained within the certificate.

An X.509 certificate contains a public key and an identity (a hostname, an organization, an individual). It is either signed by a certificate authority or self-signed. When a certificate is signed by a trusted certificate authority or validated by other means, someone holding that certificate can rely on the public key it contains to establish secure communications with another party or validate documents digitally signed by the corresponding private key.

Algorithms

Table 7.53. ECDSA Algorithms

Algorithm	Mbed TLS	PSA Crypto
ECDSA (SHA-2)	—	• `PSA_ALG_ECDSA(PSA_ALG_SHA_224)` • `PSA_ALG_ECDSA(PSA_ALG_SHA_256)` • `PSA_ALG_ECDSA(PSA_ALG_SHA_384)` • `PSA_ALG_ECDSA(PSA_ALG_SHA_512)`

Key Attributes in PSA Crypto

Table 7.54. Key Attributes for ECDSA Algorithms

Algorithm	Key Type	Key Size in Bits	Key Usage Flag
• `PSA_ALG_ECDSA(PSA_ALG_SHA_224)` • `PSA_ALG_ECDSA(PSA_ALG_SHA_256)` • `PSA_ALG_ECDSA(PSA_ALG_SHA_384)` • `PSA_ALG_ECDSA(PSA_ALG_SHA_512)`	`PSA_ECC_FAMILY_SECP_R1`	• secp192r1 : 192 • secp256r1 : 256 • secp384r1 : 384 • secp521r1 : 521	• `PSA_KEY_USAGE_SIGN_HASH` • `PSA_KEY_USAGE_VERIFY_HASH`

Note: The key usage flag must use `PSA_KEY_USAGE_SIGN_HASH` and `PSA_KEY_USAGE_VERIFY_HASH`.

Security Software Components

Table 7.55. Security Software Components for ECDSA Algorithms

Algorithm	Security Software Components
`PSA_ALG_ECDSA(PSA_ALG_SHA_224)`	PSA Crypto support for ECDSA & PSA Crypto support for SHA-224
`PSA_ALG_ECDSA(PSA_ALG_SHA_256)`	PSA Crypto support for ECDSA & PSA Crypto support for SHA-256
`PSA_ALG_ECDSA(PSA_ALG_SHA_384)`	PSA Crypto support for ECDSA & PSA Crypto support for SHA-384
`PSA_ALG_ECDSA(PSA_ALG_SHA_512)`	PSA Crypto support for ECDSA & PSA Crypto support for SHA-512

Table 7.56. Security Software Components for ECDSA PSA_ECC_FAMILY_SECP_R1 Curves

PSA_ECC_FAMILY_SECP_R1	Security Software Components
`SECP192R1`	PSA Crypto support for the secp192r1 ECC curve
`SECP256R1`	PSA Crypto support for the secp256r1 ECC curve
`SECP384R1`	PSA Crypto support for the secp384r1 ECC curve
`SECP521R1`	PSA Crypto support for the secp521r1 ECC curve

Table 7.57. Security Software Components for X.509 Certificate

Item	Security Software Components
SHA-1	PSA Crypto support for SHA-1
X.509	Mbed TLS Support for X.509 and PKI

Using Opaque ECDSA Key to Generate Certificate Signing Request (CSR)

1. Declare (and allocate) an object of type `mbedtls_pk_context` (PK context) and an object of type `psa_key_id_t` (key identifier).
2. Use the key identifier to generate an ECDSA key or load the built-in ECDSA key. Refer to the 7.4.2 Asymmetric Key for details.
3. Set up the PK context to wrap that PSA key by calling `mbedtls_pk_setup_opaque(mbedtls_pk_context *ctx, const psa_key_id_t key)`.
4. Configure the pending CSR object to use that key by calling `mbedtls_x509write_csr_set_key(mbedtls_x509write_csr *ctx, mbedtls_pk_context *key)` on that PK context.
5. Call any other function that needs to configure and generate the CSR.
6. After generating the CSR, free the PK context using `mbedtls_pk_free(mbedtls_pk_context *ctx)`. It only frees the PK context itself and leaves the key identifier untouched.
7. Either keep using the key identifier or call `psa_destroy_key()` on it, depending on the application flow.

PSA Crypto Platform Example

Click the `View Project Documentation` link to open the `readme.html` file.

The following table describes the implementation status of the PSA Crypto X.509 platform example.

Table 7.58. PSA Crypto X.509 Platform Example on Series 1 and Series 2 Devices

ECDSA Key	Series 1	Series 2 - VSE	Series 2 - HSE	Remark
SECP192R1	Y	Y	Y	—
SECP256R1	Y	Y	Y	—
SECP384R1	Y	Y	Y	Hardware acceleration only on HSE-SVH devices.
SECP521R1	Y	Y	Y	Hardware acceleration only on HSE-SVH devices.
Private Device Key	—	—	Y	Only on HSE-SVH devices.

Note: This example can select the Private Device Key (SECP256R1) to generate the root certificate CSR.

8. Revision History

Revision 0.2

September 2021

- Added 1. Series 2 Device Security Features and use the terminology defined in this section throughout the document.
- Updated 2. Device Compatibility.
- Removed terminology and Table 2.1 in 3. Device Capability.
- Updated Table 4.1 Mbed TLS Versions on page 6.
- Updated Table 5.1 PSA Crypto Key Type and Size on page 9.
- Added Key ID range to 5.3 Key Identifiers.
- Added 6. Key Attributes API.
- Updated 7. Migration Guide.
- Added Security Software Components and Quick Reference Examples to sections in 7. Migration Guide.
- Updated 7.1 Security Software Components.
- Added 7.2 PSA Crypto Configuration.
- Added NV seed to 7.3 Initialization and Random Number Generation (RNG).
- Updated Asymmetric Signature (ECDSA) to 7.6.1 Asymmetric Signature (ECDSA and EdDSA).
- Added 7.6.3 X.509 Certificate to 7. Migration Guide.

Revision 0.1

April 2021

- Initial Revision.

Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

www.silabs.com