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, ECDSA, 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. 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/EFM32JG12
- EFM32PG1/EFM32PG12
- EFM32GG11/EFM32GG12
- EFM32TG11

Wireless SoC Series 1 consists of:

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

MCU Series 2 consists of:

- EFM32PG22

Wireless SoC Series 2 consists of:

- EFR32BG21A/EFR32BG21B/EFR32BG22
- EFR32FG22
- EFR32MG21A/EFR32MG21B/EFR32MG22

2. Device Capability

On Series 2 devices, the security features are implemented by the Secure Engine. The Secure Engine may be hardware-based, or virtual (software). If hardware-based, the implementation may be either with or without Secure Vault. Throughout this document, the following conventions will be used.

- HSE - Hardware Secure Engine, either with or without Secure Vault if not specified
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE, in general)

The following table lists the SE implemented on Series 2 devices (MCU and Wireless SoC).

Table 2.1. SE on Series 2 Devices

Series 2 Devices	SE
EFR32xG21A	HSE without Secure Vault
EFR32xG21B	HSE with Secure Vault
EFM32PG22, EFR32xG22	VSE

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

Table 2.2. 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 with Secure Vault
Secure Key (4)	—	—	HSE with Secure Vault

Note:

(1) See Table 5.3 Entropy Source on Series 1 and Series 2 Devices on page 12 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 4.3 PSA Crypto Key Lifetime Support on Series 1 and Series 2 Devices on page 8 for details of Secure Key support on HSE with Secure Vault devices.

3. Overview

3.1 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 3.1. Mbed TLS Versions

Mbed TLS	Simplicity Studio	Gecko SDK Suite	PSA Crypto API	Location in Windows
v2.24.0	v5	3.1.0	Y	C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\v3.1\util\third_party\crypto\mbedtls
v2.16.6	v5	3.0.0	—	C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\v3.0\util\third_party\mbedtls
v2.7.12	v4	2.7.8	—	C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\v2.7\util\third_party\mbedtls

3.2 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.

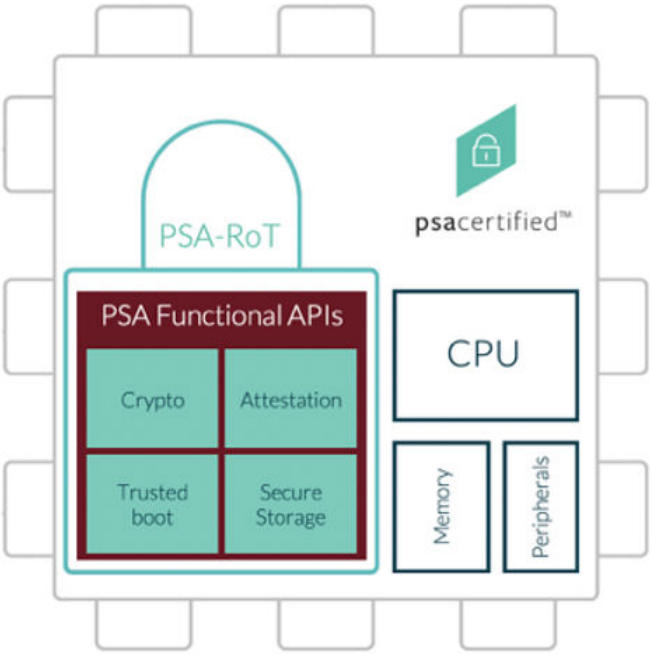


Figure 3.1. Key Requirements of PSA-RoT

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 3.1 Key Requirements of PSA-RoT on page 5 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.

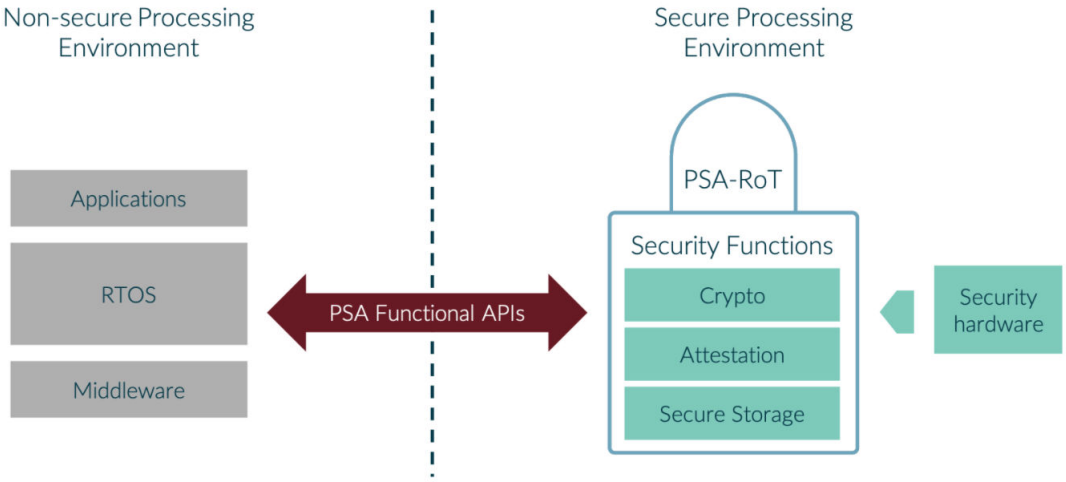


Figure 3.2. PSA Functional APIs

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.

4. 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.

4.1 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 key type and size supported in this application note.

Table 4.1. PSA Crypto Key Type and Size

Category	Key Type (1)	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 • secp256r1 : 256 • secp384r1 : 384 • secp521r1 : 521
	Montgomery curves • `PSA_ECC_FAMILY_MONTGOMERY`	• Curve25519 : 255 • Curve448 : 448

Note:

(1) The `crypto_values.h` includes the defines for key type macros.

4.2 Key Lifetimes

The lifetime is encoded in the `psa_key_lifetime_t` object. This object consists of a persistence level (bits [7:0]) and a location indicator (bits [31:8]). The persistent level indicates whether the key is volatile or persistent, and the location indicator indicates where the key is stored.

Table 4.2. PSA Crypto Key Lifetime

Type	Persistence Level (1)	Location Indicator	Storage
Volatile Plain Key	PSA_KEY_PERSISTENCE_VOLATILE	Local (0x0)	RAM
Persistent Plain Key	PSA_KEY_PERSISTENCE_DEFAULT	Local (0x1)	Flash (3)
Volatile Wrapped Key	PSA_KEY_PERSISTENCE_VOLATILE	Secure (0x1) (2)	RAM
Persistent Wrapped Key	PSA_KEY_PERSISTENCE_DEFAULT	Secure (0x1) (2)	Flash (3)
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) The `crypto_values.h` includes the defines for persistence level macros.

(2) 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.

(3) Persistent storage in flash memory is implemented by the NVM3 driver.

Table 4.3. 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 with Secure Vault
Persistent Wrapped Key	—	—	HSE with Secure Vault
Public Sign Key	—	— (1)	Y (2)
Public Command Key	—	— (1)	Y (2)
AES-128 Key	—	—	Y (3)
Private Device Key	—	—	HSE with Secure Vault (2)

Note:

(1) The PSA Crypto cannot access the Public Sign Key and Public Command Key in the VSE.

(2) These keys can only perform ECDSA operations.

(3) This key can only perform AES cipher operations. The default cipher is AES CTR (`SL_SE_BUILTIN_KEY_AES128_ALG` is defined in `sli_se_opaque_types.h`).

4.3 Key Identifiers

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

Table 4.4. PSA Crypto Key Identifier

Type	Key Identifier (1)	SE Key Identifier (2)
Volatile Plain Key	0	—
Persistent Plain Key	PSA_KEY_ID_USER_MIN to PSA_KEY_ID_USER_MAX	—
Volatile Wrapped Key	0	—
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

Note:

(1) The `crypto_values.h` includes the defines for key identifier macros.

(2) The `sli_se_opaque_types.h` includes the defines for SE key identifier macros.

4.4 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 four kinds of usage flag in the PSA Crypto.

Table 4.5. PSA Crypto Key Usage Flags

Flag	Bit-mask (1)	Description
Extractable	PSA_KEY_USAGE_EXPORT	Permission to export the key. (2)
Copyable	PSA_KEY_USAGE_COPY	Permission to copy the key.
Cacheable	PSA_KEY_USAGE_CACHE	Permission for the implementation to cache 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_MES-SAGE	Permission for a MAC calculation operation or for an asymmetric message signature operation.
"	PSA_KEY_USAGE_VERIFY_MES-SAGE	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:

(1) The `crypto_values.h` includes the defines for bit-mask macros.

(2) A public key or the public part of a key pair can always be exported regardless of the value of this permission 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.

4.5 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 set to allow the key to export in binary format.

5. Migration Guide

System Requirements and Document

Migration requires the following:

1. Simplicity Studio v5
2. Gecko SDK Suite 3.1.1 (Mbed TLS v2.24.0) or later
3. Legacy Mbed TLS API document:
   - ARM
4. PSA Crypto API (aka PSA Cryptography API) document:
   - ARM
   - Silicon Labs

Mbed TLS Versus PSA Crypto API

Table 5.1. Mbed TLS Versus PSA Crypto API in General

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. • An 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 5.2 Key Handling, 5.3 Symmetric Cryptographic Operation, and 5.4 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. The APIs (functions) for the Mbed TLS and PSA Crypto. For each type of symmetric cryptographic operation, the APIs (functions) include:
   - A pair of single-part (one-shot) functions
   - A series of functions that implement multi-part (streaming) operations

Platform Examples

Simplicity Studio 5 includes the PSA Crypto platform examples for key handling, symmetric and asymmetric cryptographic operations.

- 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.
- 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.

5.1 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 5.2. 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 TRNG (True Random Number Generator) 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 or ADC as the entropy source.

Table 5.3. Entropy Source on Series 1 and Series 2 Devices

Device	Entropy Source
MCU Series 1 - EFM32JG1, EFM32PG1	ADC
MCU Series 1 - EFM32JG12, EFM32PG12, EFM32GG11, EFM32GG12, EFM32TG11	TRNG
Wireless SoC Series 1 - EFR32xG1, EFR32xG14	RAIL
Wireless SoC Series 1 - EFR32xG12, EFR32xG13 (1)	TRNG
Series 2 - VSE	TRNG
Series 2 - HSE	TRNG

Note:

(1) The TRNG is available in EFR32xG13 revision D or later devices.

5.2 Key Handling

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

Table 5.4. 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

5.2.1 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.

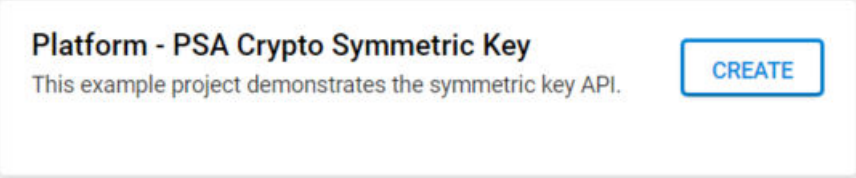
Functions

Table 5.5. 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(…)`

PSA Crypto Platform Example

The location of the `readme.html` file in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\v3.1\app\common\example\psa_crypto_symmetric_key`.



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

Table 5.6. 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 with Secure Vault devices.
128-bit	Y	Y	Y	—
192-bit	Y	Y	Y	—
256-bit	Y	Y	Y	—

5.2.2 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.

Functions

Table 5.7. 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 5.8. Asymmetric Key Size for Import and Export in PSA Crypto

ECC Key	Private Key Size (Import and Export)	Public Key Size (1) (Import (2) and Export)
SECP192R1	24-byte	49-byte
SECP256R1	32-byte	65-byte
SECP384R1	48-byte	97-byte
SECP521R1	66-byte	133-byte
CURVE25519	32-byte	32-byte
CURVE448	56-byte	56-byte

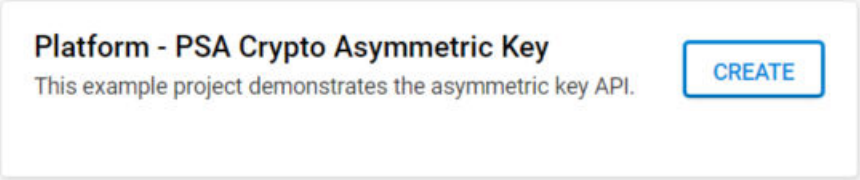
Note:

(1) The public key of the SECP curve is stored in an uncompressed format (prefix `0x04` with the X and Y coordinates).

(2) The public key cannot be stored in wrapped form.

PSA Crypto Platform Example

The location of the `readme.html` file in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\v3.1\app\common\example\psa_crypto_asymmetric_key`.



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

Table 5.9. 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 with Secure Vault device.
SECP192R1	Y	Y	Y	—
SECP256R1	Y	Y	Y	—
SECP384R1	Y	Y	Y	Hardware acceleration only on HSE with Secure Vault devices.
SECP521R1	Y	Y	Y	Hardware acceleration only on HSE with Secure Vault devices.
CURVE25519	Y	Y	Y	Hardware acceleration only on HSE with Secure Vault devices.
CURVE448 (not yet implemented)	—	—	—	Hardware acceleration only on HSE with Secure Vault devices.

Note: The VSE devices support SECP224R1, but this example does not handle this key.

5.3 Symmetric Cryptographic Operation

5.3.1 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 5.10. Hash Algorithms

Item	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`

Single-part Functions

Table 5.11. 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(…)`

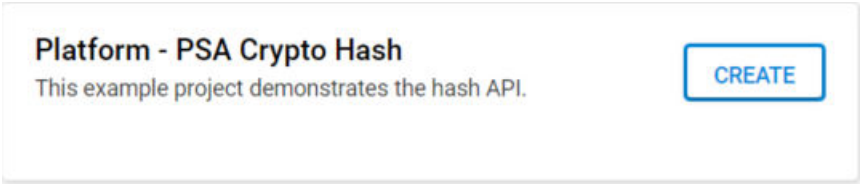
Multi-part Operations

Table 5.12. 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(…)`

PSA Crypto Platform Example

The location of the `readme.html` file in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\v3.1\app\common\example\psa_crypto_hash`.



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

Table 5.13. 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 with Secure Vault device.
SHA-512	Y	Y	Y	Hardware acceleration only on HSE with Secure Vault device.

5.3.2 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 5.14. 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 5.15. 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)	

Single-part Functions

Table 5.16. 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(…)

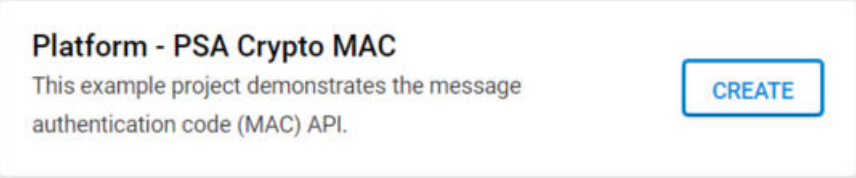
Multi-part Operations

Table 5.17. 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(…)`

PSA Crypto Platform Example

The location of the `readme.html` file in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\v3.1\app\common\example\psa_crypto_mac`.



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

Table 5.18. 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 on the HSE device is not implemented yet. • HMAC on the 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.

5.3.3 Unauthenticated Ciphers

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

Algorithms

Table 5.19. 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_CHACHA20`

Key Attributes in PSA Crypto

Table 5.20. 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_CHACHA20	PSA_KEY_TYPE_CHACHA20	256 (32-byte)	

Single-part Functions

Table 5.21. 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(…)

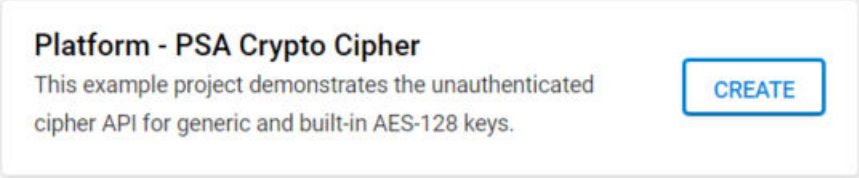
Multi-part Operations

Table 5.22. 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(…)`

PSA Crypto Platform Example

The location of the `readme.html` file in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\v3.1\app\common\example\psa_crypto_cipher`.



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

Table 5.23. 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 with Secure Vault device (not implemented yet). • CHACHA20 on the wrapped key is not supported yet.
AES-128 Key	—	—	Y	—

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

5.3.4 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 5.24. 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`
AES CCM	• `MBEDTLS_CIPHER_AES_128_CCM` • `MBEDTLS_CIPHER_AES_192_CCM` • `MBEDTLS_CIPHER_AES_256_CCM`	`PSA_ALG_CCM`
CHACHA20_POLY1305	`MBEDTLS_CIPHER_CHACHA20_POLY1305`	`PSA_ALG_CHACHA20_POLY1305`

Key Attributes in PSA Crypto

Table 5.25. 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)	

Single-part Functions

Table 5.26. 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

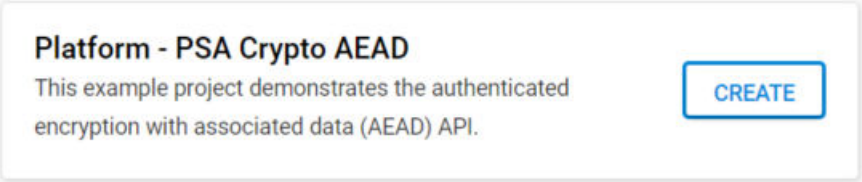
Table 5.27. Multi-part AEAD Operations

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(…)`

PSA Crypto Platform Example

The location of the `readme.html` file in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\v3.1\app\common\example\psa_crypto_aead`.



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

Table 5.28. 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 with Secure Vault device.

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

5.3.5 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 5.29. Key Derivation Algorithms

Algorithm	Mbed TLS	PSA Crypto
SHA1	`MBEDTLS_MD_SHA1`	`PSA_ALG_HKDF(PSA_ALG_SHA_1)`
SHA2	• `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)`

Key Attributes in PSA Crypto

Table 5.30. 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`

Single-part Functions

Table 5.31. Single-part Key Derivation Functions

Mbed TLS	PSA Crypto
`int mbedtls_hkdf(…)`	—

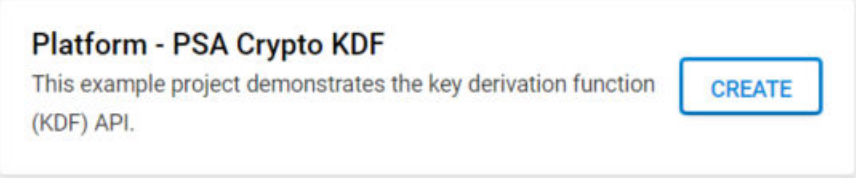
Multi-part Operations

Table 5.32. 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(…)`

PSA Crypto Platform Example

The location of the `readme.html` file in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\v3.1\app\common\example\psa_crypto_kdf`.



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

Table 5.33. 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 with Secure Vault device.
ECDH + HKDF	Y	Y	Y	Hardware acceleration (HKDF) only on HSE with Secure Vault device.

Note: The `PSA_ALG_KEY_AGREEMENT(PSA_ALG_ECDH, PSA_ALG_HKDF(hash_alg))` algorithm (ECDH + HKDF) does not apply to the wrapped key.

5.4 Asymmetric Cryptographic Operation

5.4.1 Asymmetric Signature (ECDSA)

Elliptic Curve Digital Signature Algorithm (ECDSA) is a Digital Signature Algorithm (DSA) that uses keys derived from Elliptic Curve Cryptography (ECC). ECDSA is used to create a digital signature of data to allow you to verify its authenticity without compromising its security. ECDSA is based on the more efficient ECC, so ECDSA requires smaller keys to provide equivalent security.

Algorithms

Table 5.34. Asymmetric Signature Algorithms

Algorithm	Mbed TLS (1)	PSA Crypto (2)
SHA-1	—	`PSA_ALG_ECDSA(PSA_ALG_SHA_1)`
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)`
Any hash algorithm	—	`PSA_ALG_ECDSA(PSA_ALG_ANY_HASH)`
No hashing	—	`PSA_ALG_ECDSA_ANY`

Note:

(1) Mbed TLS requires precomputed hash for ECDSA.

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

Key Attributes in PSA Crypto

Table 5.35. Key Attributes for Asymmetric Signature Algorithms

Algorithm	Key Type	Key Size in Bits	Key Usage Flag
• 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_ALG_ECDSA_ANY	PSA_ECC_FAMILY_SECP_R1	• secp192r1 : 192 • secp256r1 : 256 • secp384r1 : 384 • secp521r1 : 521	• PSA_KEY_USAGE_SIGN_HASH • PSA_KEY_USAGE_VERIFY_HASH

Functions

Table 5.36. Asymmetric Signature Functions

Mbed TLS	PSA Crypto
—	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(…)

PSA Crypto Platform Example

The location of the `readme.html` file in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\v3.1\app\common\example\psa_crypto_ecdsa`.

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

Table 5.37. PSA Crypto ECDSA 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 with Secure Vault device.
SECP521R1	Y	Y	Y	Hardware acceleration only on HSE with Secure Vault device.
Public Sign Key	—	—	Y	—
Public Command Key	—	—	Y	—
Private Device Key	—	—	Y	Only on HSE with Secure Vault device.

Note: The ECDSA hash-and-sign functions are not implemented yet.

5.4.2 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 5.38. Key Agreement Algorithms

Algorithm	Mbed TLS	PSA Crypto
Key agreement	—	`PSA_ALG_ECDH`
Key agreement with a key derivation	—	`PSA_ALG_KEY_AGREEMENT(PSA_ALG_ECDH, PSA_ALG_HKDF(hash_alg))`

Key Attributes in PSA Crypto

Table 5.39. 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 • secp256r1 : 256 • secp384r1 : 384 • secp521r1 : 521	PSA_KEY_USAGE_DERIVE
	PSA_ECC_FAMILY_MONTGOMERY	• Curve25519 : 255 • Curve448 : 448	

Functions

Table 5.40. Key Agreement Functions

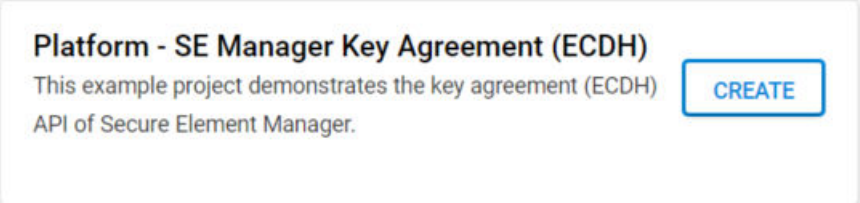
Mbed TLS	PSA Crypto
int mbedtls_ecdh_compute_shared(…)	psa_status_t psa_raw_key_agreement(…)
—	psa_status_t psa_key_derivation_key_agreement(…) (1)

Note:

(1) Refer to the PSA Crypto KDF platform example for details.

PSA Crypto Platform Example

The location of the readme.html file in Windows is C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\v3.1\app\common\example\psa_crypto_ecdh.



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

Table 5.41. 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 with Secure Vault devices.
SECP521R1	Y	Y	Y	Hardware acceleration only on HSE with Secure Vault devices.
CURVE25519	Y	Y	Y	Hardware acceleration only on HSE with Secure Vault devices.
CURVE448 (not yet implemented)	—	—	—	Hardware acceleration only on HSE with Secure Vault devices.

5.5 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 with Secure Vault device.

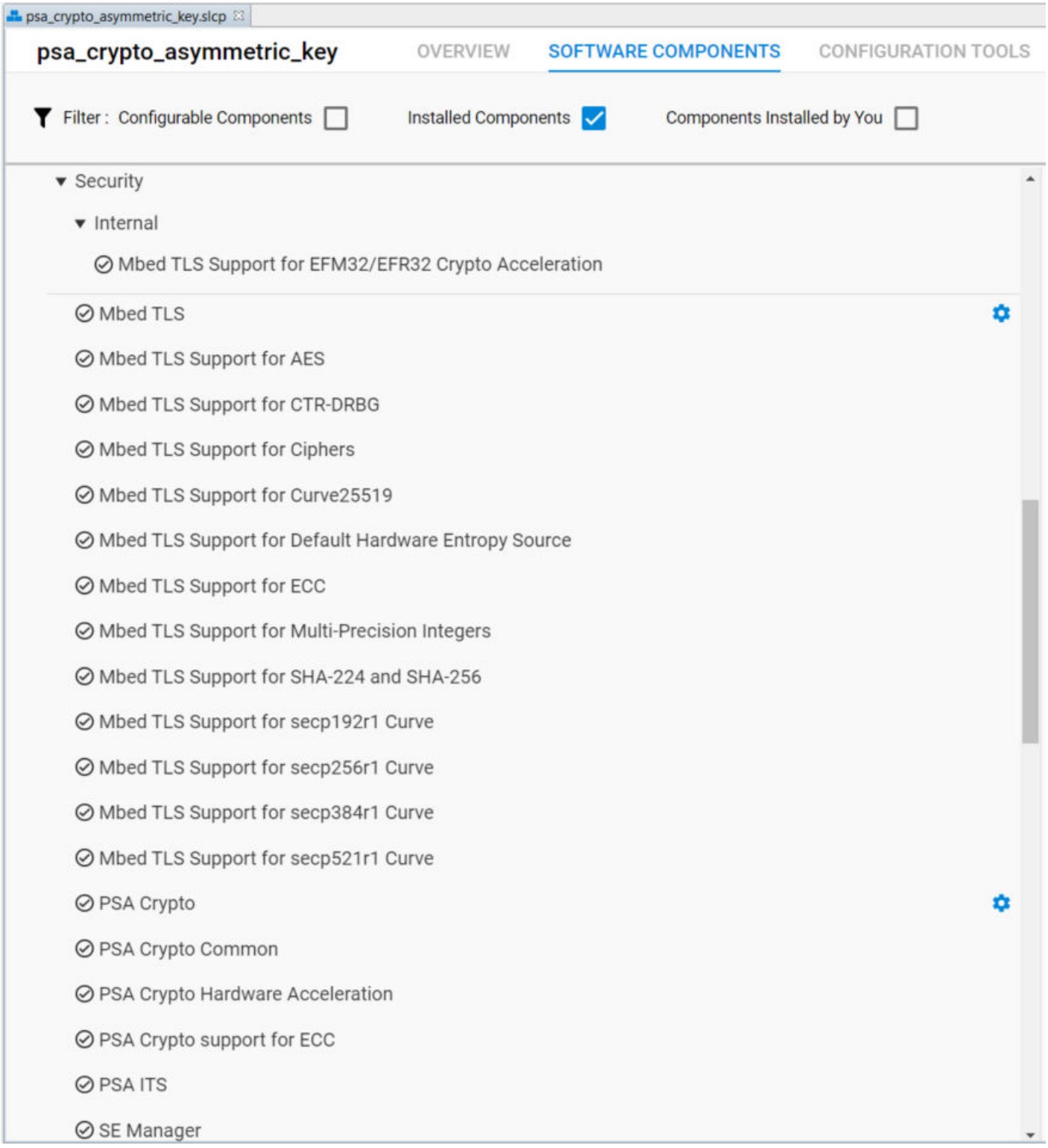


Figure 5.1. Installed Security Software Components

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.

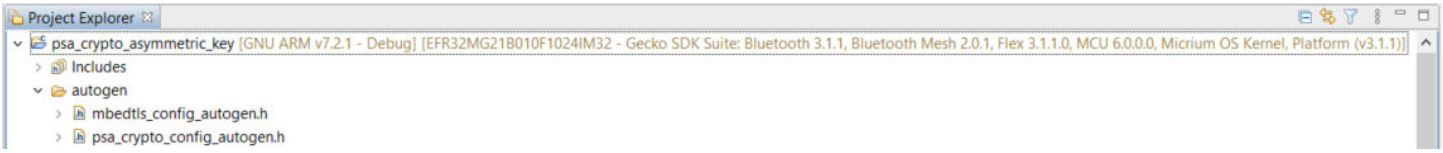


Figure 5.2. Mbed TLS and PSA Crypto Configuration Files

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

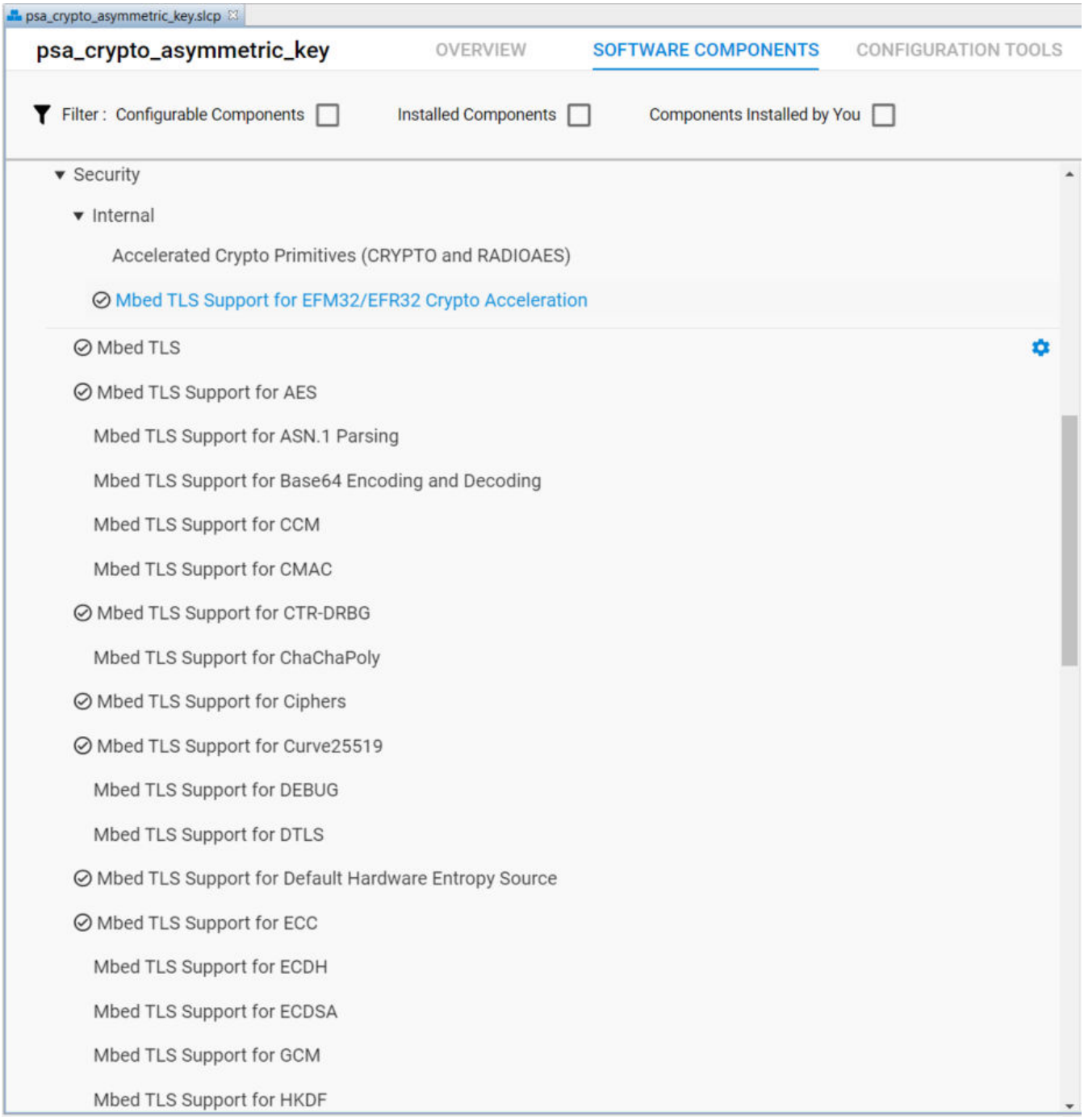


Figure 5.3. Available Security Software Components on the Target Device

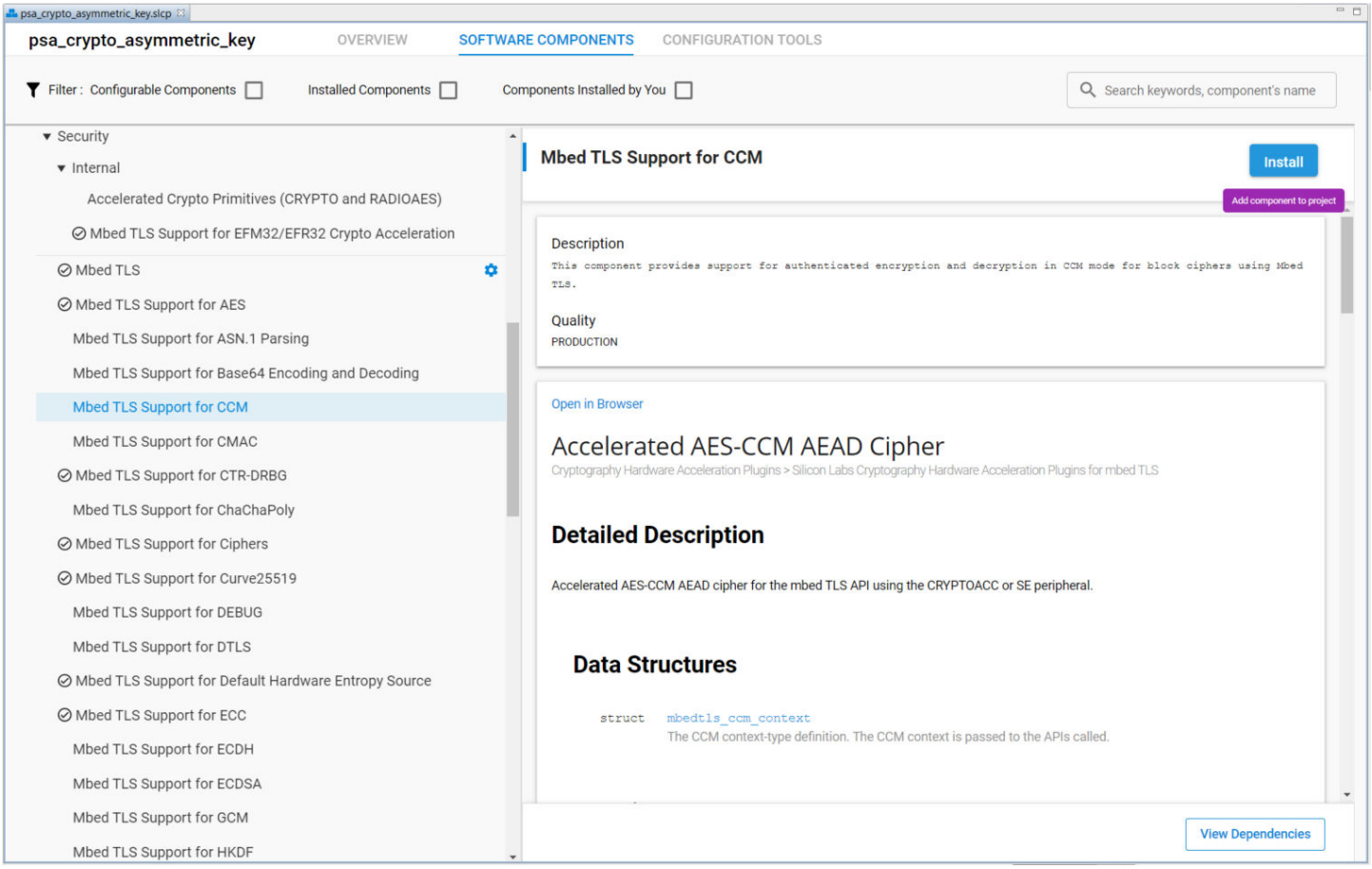


Figure 5.4. Install a Security Software Component

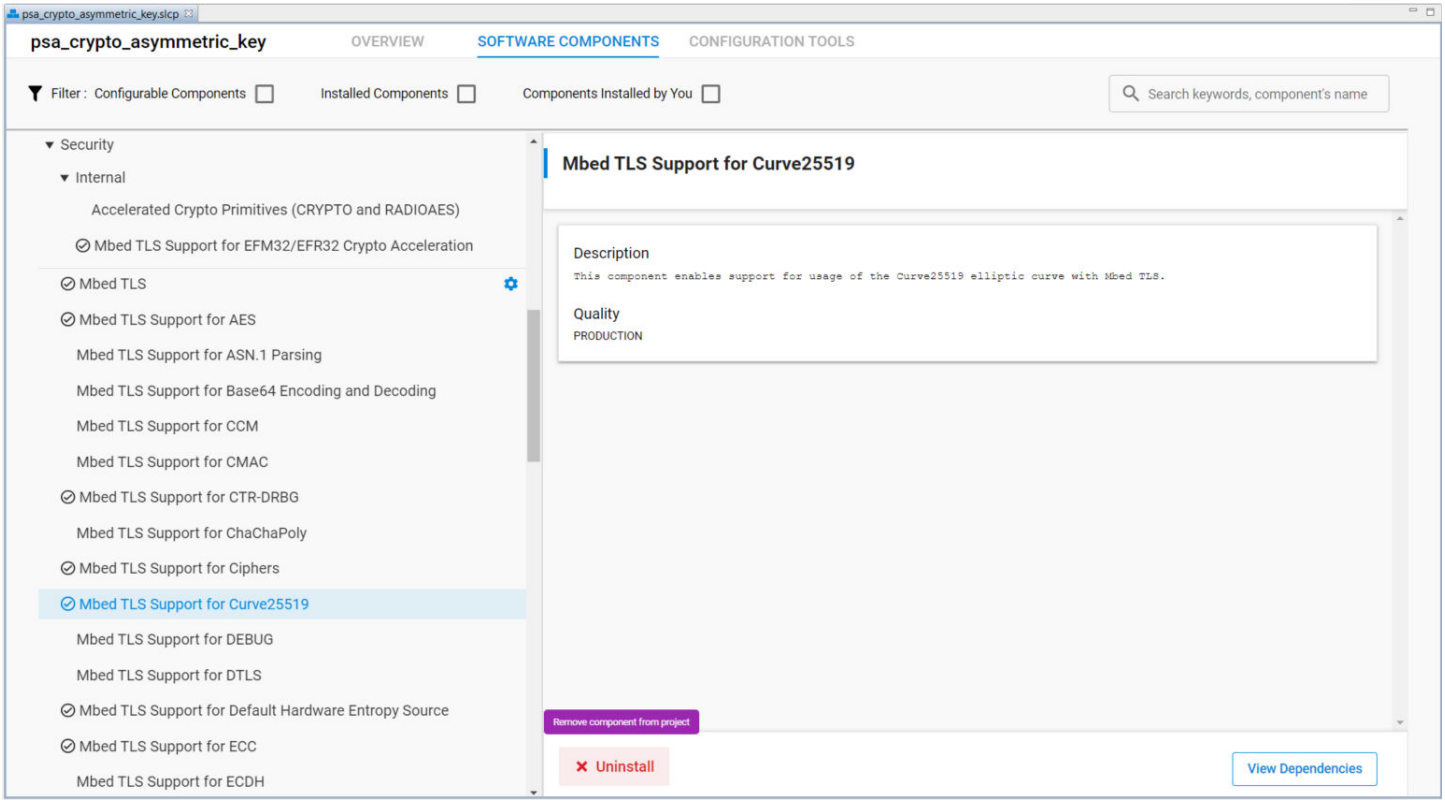


Figure 5.5. Uninstall a Security Software Component

6. Revision History

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