AN6205

Application note

Introduction to the use of PKA key wrapping with coupling and chaining bridge on STM32 MCUs

Introduction

This application note introduces the use of PKA key wrapping with the coupling and chaining bridge (CCB) peripheral available in a selection of STM32 microcontrollers (MCUs). The MCU is listed in Table 1.

The document is structured as follows:

- Why do we need PKA key wrapping? See Section 1.
- What are the cryptographic functions involved in the wrapping of PKA keys? See Section 2.
- What is the set of functions STM32CubeMX proposes to run those updated cryptographic functions? See Section 3.

Table 1. Applicable product

Type	Product line
Microcontroller	STM32U385

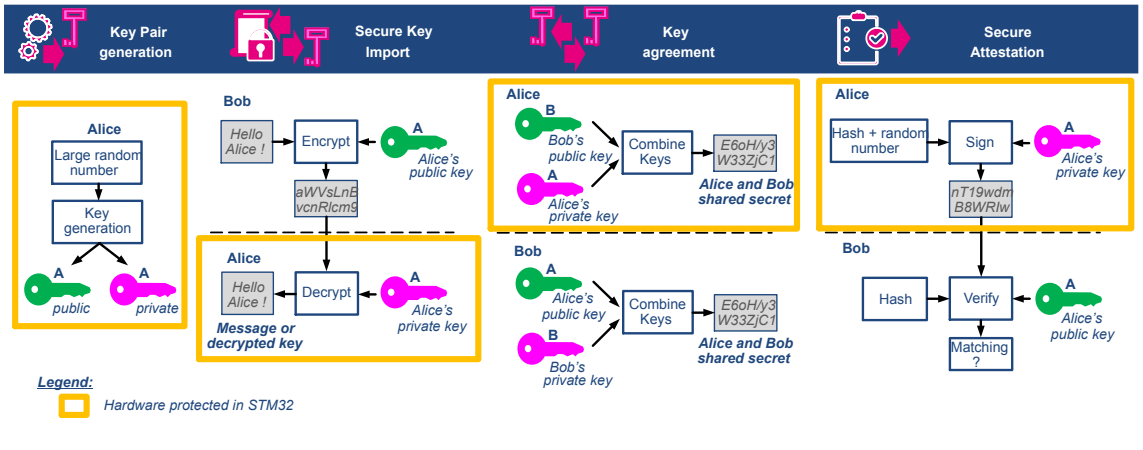
1 Why do we need PKA key wrapping?

This section concerns the use of private keys and why it is helpful to wrap them.

1.1 Private key usage

Figure 1 summarizes the possible usages of private keys in the PKA peripheral. This usage is highlighted in yellow.

Figure 1. Cryptographic usages for private keys



Private keys are used in secure key import, a crucial process in cryptographic systems that ensures that cryptographic keys are transferred securely from one entity to another without being exposed to unauthorized parties. Figure 1 shows a secure transfer of data from Bob to Alice using the RSA method. The only prerequisite for this transfer is for Bob to hold the genuine public key of Alice.

A more complex way of sharing an encryption key between two parties is called cryptographic key agreements. Those methods are protocols that allow two or more parties to securely establish a shared secret key over an insecure communication channel. This shared secret key can then be used for subsequent encryption and decryption of messages between the parties. Figure 1 shows a key agreement between Bob and Alice, with both parties holding the genuine public key of the other party.

Finally, private keys are also used for secure attestation. It is a process by which a device or system proves its integrity and authenticity to another party. At the end of the secure attestation process the other party accepts (or not) to interact with this device or system, because it got the proof that the device is running trusted software and has not been tampered with. This method is crucial in various applications, including secure boot processes, trusted computing, and IoT security. Figure 1 shows the principle of a secure attestation using the ECC method.

Note: For all those use case elliptic curves or RSA/DSA variants exists.

1.2 Private key protection

Private keys are created during a process called digital key pair generation, shown in Figure 1. It is a fundamental process in cryptography, during which two mathematically related keys are created: a public key and a private key.

The security of asymmetric cryptography relies heavily on the protection of the private key. For example, as the private key is used to decrypt messages that were encrypted with the user's public key, if someone gains access to a private key, it is then possible to decrypt and read confidential messages.

Also, as the private key is used to create digital signatures that verify the user's identity, if someone other than the user has access to the private key, the user's digital signature can be forged, making it appear as though the user has approved or sent messages that were not written by the user.

Lastly, when the private key ensures that the messages sent by the user are not tampered with, if someone other than the user has the private key, the messages can be altered and still make them appear as if they are from the user.

1.3 Private key wrapping in STM32

As described before, the private key usage can be critical for device security. Hence it is important to make it as secure as possible. On the STM32 microcontroller listed in Table 1. Applicable product, wrapped private keys can be stored anywhere, because they can only be used on this device. Also, the software that manipulates wrapped private keys can never access its value in the clear, making the overall application more robust against attacks.

This optional hardware protection relies on a CCB peripheral that enforces the encryption (or wrapping) of PKA private keys using a 256-bit AES key loaded in side-channel protected SAES engine.

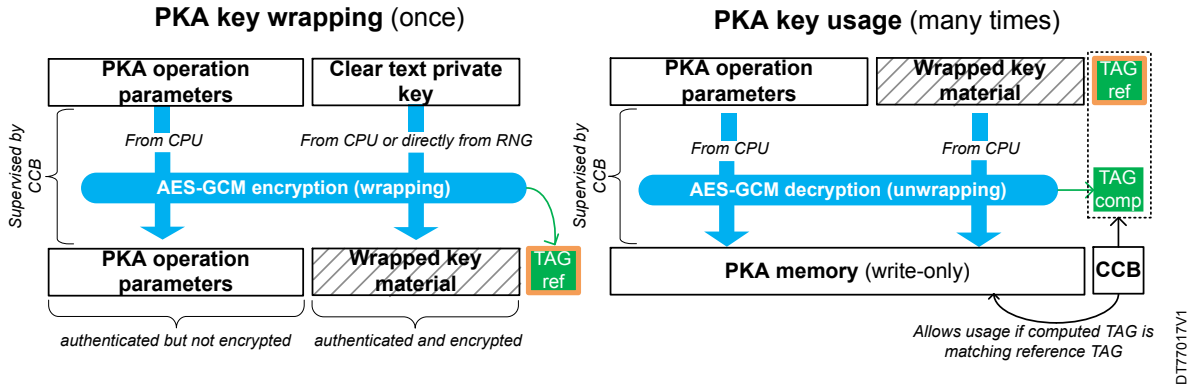
2 Cryptographic primitives with PKA key wrapping

This section describes the principle of PKA private keys wrapping using AES. It also summarizes the PKA operations involved in the use cases described in Section 1.1: Private key usage.

2.1 AES: AES-GCM authenticated encryption with associated data (AEAD)

AES-GCM (Advanced Encryption Standard - Galois/Counter Mode) is a widely used cryptographic method that provides both confidentiality and data integrity. It combines the AES encryption algorithm with the Galois/Counter Mode of operation to achieve authenticated encryption. The PKA key wrapping with coupling and chaining bridge (CCB) uses the AES-GCM algorithm, as shown in Figure 2. The key size used is 256-bit.

Figure 2. AES-GCM usage for PKA key wrapping



Under the supervision of the CCB, the *PKA operation parameters* are only authenticated while the *key material* is authenticated and encrypted using AES-GCM. At the end of the wrapping (or encryption) process, the application must store the *reference tag* together with the operation parameters and the wrapped key material. Indeed, once the PKA wrapped key is decrypted in PKA memory, PKA can use it with the verified operation parameters only if the reference and the computed AES-GCM tags match.

2.2 AES: Key wrapping in SAES peripheral

For PKA key wrapping or unwrapping the application must load a key in SAES peripheral before performing the required AES-GCM operation shown in Figure 2. To ensure the wrapped key material is never seen in the clear, the application must select the DHUK (with optional XOR with BHK) or unwrap in the SAES an existing PKA key wrapping encryption key that has been encrypted in wrapped-key mode using the DHUK (with optional XOR with BHK). STM32 HAL exports API functions that help managing this last scenario, as described in sections Section 3.1.1, Section 3.2.1, and Section 3.3.1.

2.3 PKA: Scalar multiplication

STM32CubeMX proposes the scalar multiplication method to implement the following use cases, listed in Section 1.1: ECC key generation, ECDH key agreement and ECC secure key import.

When an application needs a given ECC public key, it can use the related private key with the STM32CubeMX scalar multiplication API, adding the ECC curve base point as a parameter.

Also, ECDH key agreement requires the application to use the scalar multiplication API with the related private key and the other party public key as point on the ECC curve. The agreed AES key is usually the result of a KDF2 using the X coordinate of the previously calculated scalar multiplication result. This last part is out of scope of the application note.

Lastly, ECC secure key import requires the application to use the scalar multiplication API with the other party public key as a point on the ECC curve. Application then compute a KDF with the X coordinate of the point previously computed, using the result as an AES key to decrypt the key to import. This last part is out of scope of this application note.

In a legacy application, STM32 HAL provides the scalar multiplication API function below:

```
HAL_StatusTypeDef HAL_PKA_ECCMul(PKA_HandleTypeDef *hpka, PKA_ECCMulInTypeDef *in, uint32_t T
imeout);
```

```
typedef struct
{
  uint32_t scalarMulSize;           /*!< Number of element in scalarMul array */
  uint32_t modulusSize;             /*!< Number of element in modulus, coefA, pointX and poin
tY arrays */
  uint32_t coefSign;                /*!< Curve coefficient a sign */
  const uint8_t *coefA;             /*!< Pointer to curve coefficient |a| (Array of modulusSi
ze elements) */
  const uint8_t *coefB;             /*!< pointer to curve coefficient b */
  const uint8_t *modulus;           /*!< Pointer to curve modulus value p (Array of modulusSi
ze elements) */
  const uint8_t *pointX;            /*!< Pointer to point P coordinate xP (Array of modulusSi
ze elements) */
  const uint8_t *pointY;            /*!< Pointer to point P coordinate yP (Array of modulusSi
ze elements) */
  const uint8_t *scalarMul;         /*!< Pointer to scalar multiplier k   (Array of scalarMul
Size elements) */
  const uint8_t *primeOrder;        /*!< pointer to order of the curve */
} PKA_ECCMulInTypeDef;
```

STM32 HAL also provides the scalar multiplication API function with PKA wrapped keys, as described in Section 3.2.

2.4 PKA: ECDSA signature

STM32CubeMX proposes the ECDSA signature method to implement the ECC secure attestation, listed in Section 1.1. For this use case the application must do the following:

1. Select an ECC curve and generates a private key using the TRNG (size depends on the selected curves).
2. Compute the hash of the message for which a signature is needed.
3. Use the STM32 HAL ECDSA signature API to compute the signature (r, s). A random nonce must also be provided, unless the CCB is used (see Section 3.1).
4. When distributing the signature as an attestation, make sure the corresponding public key is available to the remote verifier. This public key can be computed using the scalar multiplication API and the ECC curve base point G (see Section 2.3).

In a legacy application STM32 HAL provides the ECDSA signature function below.

```
HAL_StatusTypeDef HAL_PKA_ECDSASign(PKA_HandleTypeDef *hpka, PKA_ECDSASignInTypeDef *in, uint
32_t Timeout);
```

```
typedef struct
{
  uint32_t primeOrderSize;      /*!< Number of element in primeOrder array */
  uint32_t modulusSize;         /*!< Number of element in modulus array */
  uint32_t coefSign;              /*!< Curve coefficient a sign */
  const uint8_t *coef;            /*!< Pointer to curve coefficient |a|     (Array of mo
dulusSize elements) */
  const uint8_t *coefB;           /*!< Pointer to B coefficient             (Array of mo
dulusSize elements) */
  const uint8_t *modulus;         /*!< Pointer to curve modulus value p     (Array of mo
dulusSize elements) */
  const uint8_t *integer;         /*!< Pointer to random integer k          (Array of pr
imeOrderSize elements) */
  const uint8_t *basePointX;      /*!< Pointer to curve base point xG       (Array of mo
dulusSize elements) */
  const uint8_t *basePointY;      /*!< Pointer to curve base point yG       (Array of mo
dulusSize elements) */
  const uint8_t *hash;            /*!< Pointer to hash of the message       (Array of pr
imeOrderSize elements) */
  const uint8_t *privateKey;      /*!< Pointer to private key d             (Array of pr
imeOrderSize elements) */
  const uint8_t *primeOrder;      /*!< Pointer to order of the curve n      (Array of pr
```

```
imeOrderSize elements) */
} PKA_ECDSASignInTypeDef;
```

```
typedef struct
{
  uint8_t *RSign;                     /*!< Pointer to signature part r          (Array of mo
dulusSize elements) */
  uint8_t *SSign;                     /*!< Pointer to signature part s          (Array of mo
dulusSize elements) */
} PKA_ECDSASignOutTypeDef;
```

STM32 HAL also provides the ECDSA signature function with PKA wrapped keys, as described in Section 3.1.

2.5 PKA: Modular exponentiation

STM32CubeMx proposes the modular exponentiation method to implement the following use cases, listed in Section 1.1: Private key usage: RSA key agreement, RSA secure key import and RSA attestation. Details on this are out of scope of this application note.

In a legacy application, STM32 HAL provides the modular exponentiation function below.

```
HAL_StatusTypeDef HAL_PKA_ModExpProtectMode(PKA_HandleTypeDef *hpka, PKA_ModExpProtectModeInT
ypeDef *in, uint32_t Timeout);
```

```
typedef struct
{
  uint32_t expSize;                   /*!< Size of the operand in bytes */
  uint32_t OpSize;                    /*!< Size of the operand in bytes */
  const uint8_t *pOp1;                /*!< Pointer to Operand 1 */
  const uint8_t *pExp;                /*!< Pointer to Exponent */
  const uint8_t *pMod;                /*!< Pointer to modulus */
  const uint8_t *pPhi;                /*!< Pointer to Phi value */
} PKA_ModExpProtectModeInTypeDef;
```

STM32 HAL also provides the modular exponentiation function with PKA wrapped keys, as described in Section 3.3.

3 Wrapped key usage with the CCB

This section details how private keys in their wrapped form can be used in the STM32 microcontroller listed in Table 1.

3.1 ECDSA signature

STM32CubeMX proposes a set of functions to compute ECDSA signatures with PKA wrapped keys.

3.1.1 Key generation

ECC private key is generated with the RNG. In FIPS PUB 186-4 specification "*Digital Signature Standard (DSS)*" section B.4 NIST proposes two methods for the generation of the ECC private key (*extra random bits* or *testing candidates*). Application must select one of those two methods when computing private keys for ECDSA signature generation.

Public key computation is described in Step 3 of the next section.

3.1.2 Blob creation, blob usage

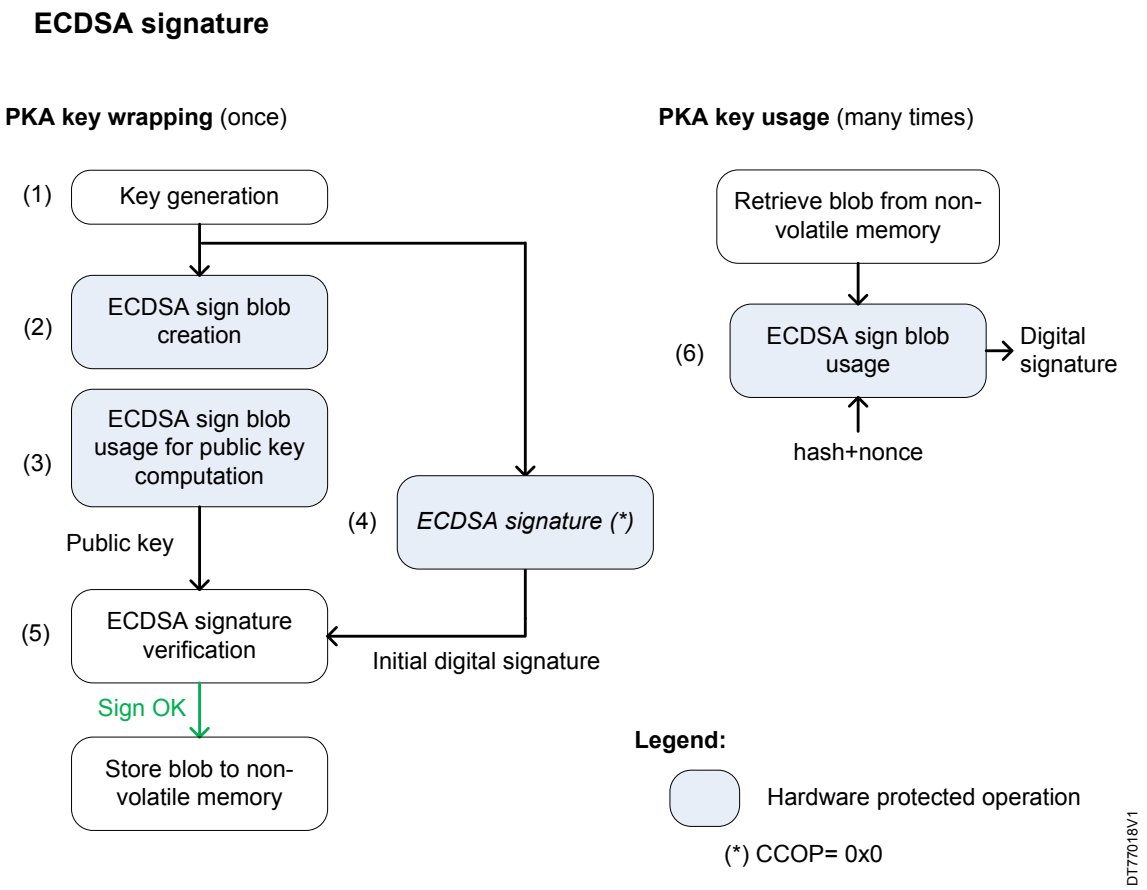
Wrapping a private key for ECDSA signatures corresponds to creating a ECDSA signature key blob. Its usage is summarized in Table 2.

Table 2. ECC signature key blob usage

Attestation	Key import	Key agreement	Key generation	Usage in PKA
X	-	-	X	ECDSA signature Scalar multiplication

ECDSA signature blob creation and usage methods are summarized in Figure 3.

Figure 3. ECDSA signature blob creation and usage



STM32CubeMX supports the following methods to implement the flow in Figure 3:

1. Generate keys.
2. Protected ECDSA signature blob creation, with a selected ECC curve and the generated private key as input.
3. Protected scalar multiplication, with ECC curve base point G and previously computed blob as input. Result is the computed ECDSA signature public key.
4. ECDSA signature using the generated private key as input. Any hash information can be used. Call the result of this signature (r,s).
5. ECDSA signature verification, using the reference hash information, the signature (r,s) and the computed ECDSA signature public key. If the API returns that the signature is correct you can save the ECDSA signature blob created in step 1.
6. ECDSA key blob can be used at any time to compute ECDSA signature using a wrapped key.

Step 1 code:

Generate private key `ClearPrivateKey` with the TRNG, as defined in Section 3.1.1.

Step 2 code:

If the user wants to wrap `ClearPrivateKey` with DHUK (or DHUK XOR BHK), select `WrappingKeyType = HAL_CCB_USER_KEY_HW` (or `HAL_CCB_USER_KEY_HSW`).

If the user wants to wrap `ClearPrivateKey` with a known `ClearAESKey` this user-defined wrapping key must first be encrypted by calling the HAL_CCB function below, with `WrappingKeyType = HAL_CCB_USER_KEY_WRAPPED`. The recommended algorithm is AES-CBC with 256-bit DHUK, selecting `AES_Algorithm = CCB_AES_CBC` and initializing `pInitVect`.

```
HAL_StatusTypeDef HAL_CCB_ECDSA_WrapSymmetricKey(CCB_HandleTypeDef *hccb, const uint32_t *pCl
earAESKey,CCB_WrappingKeyTypeDef *pWrappingKey);
```

```
typedef struct
{
uint32_t WrappingKeyType;    /*!< This parameter can be value of
@ref CCB_WrappingKeyTypeDef */
uint32_t AES_Algorithm;    /*!< Used only when wrappingkey type is
HAL_CCB_USER_KEY_WRAPPED, AES Algorithm. This parameter can be a value of @ref CCB_AES_Algori
thm_Mode */
uint32_t *pInitVect;    /*!< Used only when wrappingkey type is
HAL_CCB_USER_KEY_WRAPPED, Pointer to the
initialization vector, counter with CBC Algo */
uint32_t *pUserWrappedWrappingKey;    /*!< Used only when wrappingkey type is
HAL_CCB_USER_KEY_WRAPPED, Pointer to the wrapped wrapping Key */
} CCB_WrappingKeyTypeDef;
```

Wrap the `ClearPrivateKey` with the HAL_CCB function below. Keep the `WrappedPrivateKeyBlob` result in volatile memory.

```
HAL_StatusTypeDef HAL_CCB_ECDSA_WrapPrivateKey(CCB_HandleTypeDef *hccb, CCB_ECDSACurveParamTy
peDef *pCurveParam,
uint8_t *pClearPrivateKey, CCB_WrappingKeyTypeDef *pWrappingKey,
const CCB_ECDSAKeyBlobTypeDef *pWrappedPrivateKeyBlob);
```

Step 3 code:

Generate the public key with this HAL_CCB function and the input `WrappedPrivateKeyBlob`. Parameter `PublicKey` is the ECC curve base point.

```
HAL_StatusTypeDef HAL_CCB_ECDSA_ComputePublicKey(CCB_HandleTypeDef *hccb,
CCB_ECDSACurveParamTypeDef *pCurveParam,
CCB_WrappingKeyTypeDef *pWrappingKey,
CCB_ECDSAKeyBlobTypeDef *pWrappedPrivateKeyBlob,
CCB_ECCMulPointTypeDef *pPublicKey);
```

This Step 3 code is included in blob creation API function `HAL_CCB_ECDSA_WrapPrivateKey` described in Step 2.

Step 4 code:

Generate the ECDSA signature (`RSign, SSign`) using the HAL_CCB function below, with the `ClearPrivateKey` key used as `privateKey`. A random number integer and a hash `hash` are also required.

```
HAL_StatusTypeDef HAL_PKA_ECDSASign(PKA_HandleTypeDef *hpka, PKA_ECDSASignInTypeDef *in, uint
32_t Timeout);
```

```
typedef struct
{
  uint32_t primeOrderSize;            /*!< Number of element in primeOrder array */
  uint32_t modulusSize;               /*!< Number of element in modulus array */
  uint32_t coefSign;                  /*!< Curve coefficient a sign */
  const uint8_t *coef;                /*!< Pointer to curve coefficient |a|     (Array of mo
dulusSize elements) */
  const uint8_t *coefB;               /*!< Pointer to B coefficient             (Array of mo
dulusSize elements) */
  const uint8_t *modulus;             /*!< Pointer to curve modulus value p     (Array of mo
dulusSize elements) */
  const uint8_t *integer;             /*!< Pointer to random integer k          (Array of pr
imeOrderSize elements) */
  const uint8_t *basePointX;          /*!< Pointer to curve base point xG       (Array of mo
dulusSize elements) */
  const uint8_t *basePointY;          /*!< Pointer to curve base point yG       (Array of mo
dulusSize elements) */
```

```
  const uint8_t *hash;                /*!< Pointer to hash of the message       (Array of pr
imeOrderSize elements) */
  const uint8_t *privateKey;          /*!< Pointer to private key d             (Array of pr
imeOrderSize elements) */
  const uint8_t *primeOrder;          /*!< Pointer to order of the curve n      (Array of pr
imeOrderSize elements) */
} PKA_ECDSASignInTypeDef;
```

```
typedef struct
{
  uint8_t *RSign;                     /*!< Pointer to signature part r          (Array of mo
dulusSize elements) */
  uint8_t *SSign;                     /*!< Pointer to signature part s          (Array of mo
dulusSize elements) */
} PKA_ECDSASignOutTypeDef;
```

This Step 4 code is included in blob creation API function `HAL_CCB_ECDSA_WrapPrivateKey` described in Step 2.

Step 5 code:

Verify `Signature` using `PublicKey` computed in Step 3 and the HAL_PKA function below.

```
HAL_StatusTypeDef HAL_PKA_ECDSAVerif(PKA_HandleTypeDef *hpka, PKA_ECDSAVerifInTypeDef *in, ui
nt32_t Timeout);
```

If the signature verification returns true `WrappedPrivateKeyBlob` can be saved anywhere in nonvolatile memory.

This Step 5 code is included in blob creation API function `HAL_CCB_ECDSA_WrapPrivateKey` described in Step 2

Step 6 code:

At any time, a signature can be computed using `WrappedPrivateKeyBlob` and the HAL_CCB function:

```
HAL_StatusTypeDef HAL_CCB_ECDSA_Sign(CCB_HandleTypeDef *hccb, CCB_ECDSACurveParamTypeDef *pCu
rveParam,
CCB_WrappingKeyTypeDef *pWrappingKey,
CCB_ECDSAKeyBlobTypeDef *pWrappedPrivateKeyBlob, uint8_t *pHash,
CCB_ECDSASignTypeDef *pSignature);
```

3.2 Scalar multiplication

STM32CubeMX proposes a set of functions to run scalar multiplication with PKA wrapped keys.

3.2.1 Key generation

ECC private key is generated with the RNG. In FIPS PUB 186-4 specification "*Digital Signature Standard (DSS)*" section B.4 NIST proposes two methods for the generation of the ECC private key (*extra random bits* or *testing candidates*). Application must select one of those two methods when computing ECC private keys.

Public key computation is described in Step 4 of the next section.

3.2.2 Blob creation, blob usage

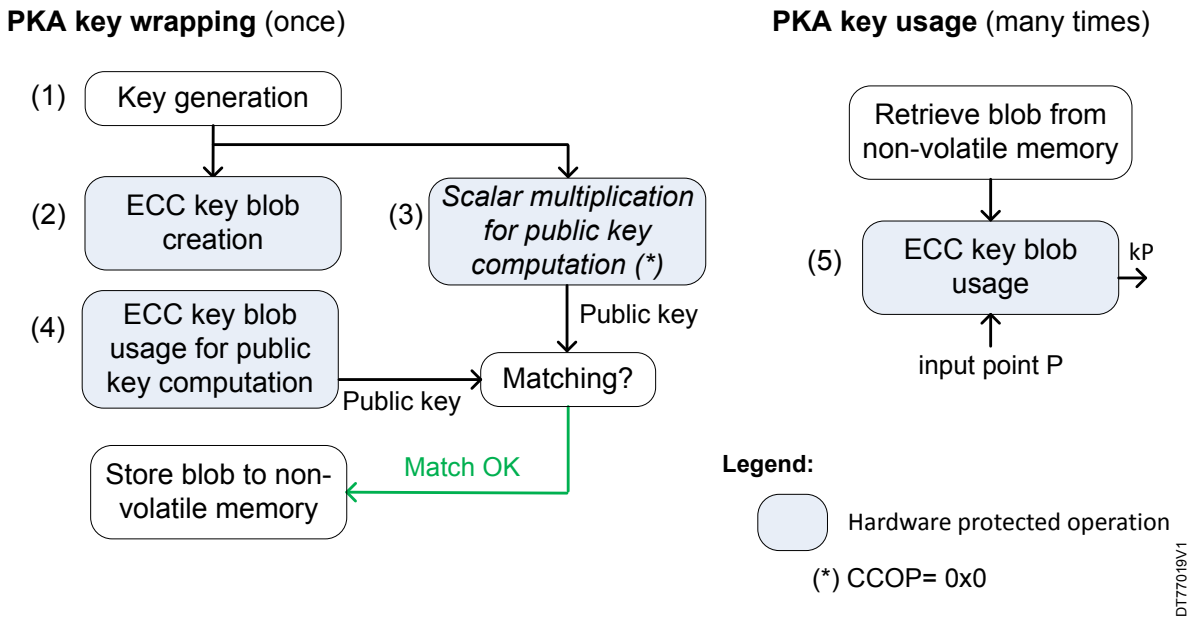
Wrapping an ECC private key corresponds to creating an ECC key blob. Its usage is summarized in Table 3.

Table 3. ECC key blob usage

Attestation	Key import	Key agreement	Key generation	Usage in PKA
-	X	X	X	Scalar multiplication

ECC key blob creation and usage methods are summarized in Figure 4.

Figure 4. ECC key blob creation and usage



STM32CubeMX supports the following methods to implement the flow on Figure 4:

1. Generate keys.
2. Protected ECC key blob creation, with selected ECC curve and the generated private key as input.
3. Protected scalar multiplication, with ECC curve base point G and previously generated private key as input. The result is the reference public key.
4. Protected scalar multiplication, with ECC curve base point G and previously computed ECC key blob as input. The result is the computed public key. If reference and computed points are identical, save the ECC key blob created in step 1.
5. ECC key blob can be used at any time to compute scalar multiplication using a wrapped key.

Step 1 code:

Generate private key `ClearPrivateKe`y with the TRNG, as defined in Section 3.2.1.

Step 2 code:

If the user wants to wrap `ClearPrivateKey` with DHUK (or DHUK XOR BHK) select `WrappingKeyType = HAL_CCB_USER_KEY_HW` (or `HAL_CCB_USER_KEY_HSW`).

If the user wants to wrap `ClearPrivateKey` with a known `ClearAESKey` this user-defined wrapping key must first be encrypted by calling the HAL_CCB function below, with `WrappingKeyType = HAL_CCB_USER_KEY_WRAPPED`. The recommended algorithm is AES-CBC with 256-bit DHUK, selecting `AES_Algorithm = CCB_AES_CBC` and initializing `pInitVect`.

```
HAL_StatusTypeDef HAL_CCB_ECC_WrapSymmetricKey(CCB_HandleTypeDef *hccb, const uint32_t *pClea
rAESKey,CCB_WrappingKeyTypeDef *pWrappingKey);
```

```
typedef struct
{
uint32_t WrappingKeyType;    /*!< This parameter can be value of
@ref CCB_WrappingKeyTypeDef */
uint32_t AES_Algorithm;    /*!< Used only when wrappingkey type is
HAL_CCB_USER_KEY_WRAPPED, AES Algorithm. This parameter can be a value of @ref CCB_AES_Algori
thm_Mode */
uint32_t *pInitVect;    /*!< Used only when wrappingkey type is
HAL_CCB_USER_KEY_WRAPPED, Pointer to the initialization vector, counter with CBC Algo */
uint32_t *pUserWrappedWrappingKey; /*!< Used only when wrappingkey type is
```

```
HAL_CCB_USER_KEY_WRAPPED, Pointer to the wrapped wrapping Key */
} CCB_WrappingKeyTypeDef;
```

Wrap the `ClearPrivateKey` with the HAL_CCB function below. Keep the `WrappedPrivateKeyBlob` result in volatile memory.

```
HAL_StatusTypeDef HAL_CCB_ECC_WrapPrivateKey(CCB_HandleTypeDef *hccb, CCB_ECCMulCurveParamTyp
eDef *pCurveParam,
uint8_t *pClearPrivateKey,
CCB_WrappingKeyTypeDef *pWrappingKey,
CCB_ECCMulKeyBlobTypeDef *pWrappedPrivateKeyBlob);
```

Step 3 code:

Generate the reference public key with this HAL_PKA function below. ECC curve base point G must be written as input (`pointX, pointY`). Private key is the `scalarMul`.

```
HAL_StatusTypeDef HAL_PKA_ECCMul(PKA_HandleTypeDef *hpka, PKA_ECCMulInTypeDef *in, uint32_t T
imeout);
```

```
typedef struct
{
  uint32_t scalarMulSize;              /*!< Number of element in scalarMul array */
  uint32_t modulusSize;                /*!< Number of element in modulus, coefA, pointX and p
ointY arrays */
  uint32_t coefSign;                   /*!< Curve coefficient a sign */
  const uint8_t *coefA;                /*!< Pointer to curve coefficient |a| (Array of modulu
sSize elements) */
  const uint8_t *coefB;                /*!< pointer to curve coefficient b */
  const uint8_t *modulus;              /*!< Pointer to curve modulus value p (Array of modulu
sSize elements) */
  const uint8_t *pointX;               /*!< Pointer to point P coordinate xP (Array of modulu
sSize elements) */
  const uint8_t *pointY;               /*!< Pointer to point P coordinate yP (Array of modulu
sSize elements) */
  const uint8_t *scalarMul;            /*!< Pointer to scalar multiplier k   (Array of scalar
MulSize elements) */
  const uint8_t *primeOrder;           /*!< pointer to order of the curve */
} PKA_ECCMulInTypeDef;
```

This Step 3 code is included in blob creation API function `HAL_CCB_ECC_WrapPrivateKey` described in Step 2.

Step 4 code:

Generate public key with this HAL_CCB function and the input `WrappedPrivateKeyBlob` computed in Step 2. ECC curve base point G must be written as `InputPoint`.

```
HAL_StatusTypeDef HAL_CCB_ECC_ComputeScalarMul(CCB_HandleTypeDef *hccb, CCB_ECCMulCurveParamT
ypeDef *pCurveParam,
CCB_WrappingKeyTypeDef *pWrappingKey,
CCB_ECCMulKeyBlobTypeDef *pWrappedPrivateKeyBlob,
CCB_ECCMulPointTypeDef *pInputPoint,
CCB_ECCMulPointTypeDef *pOutputPoint);
```

```
typedef struct
{
  uint8_t *pPointX;      /*!< Pointer to point P coordinate xP */
  uint8_t *pPointY;      /*!< Pointer to point P coordinate yP */
} CCB_ECCMulPointTypeDef;
```

Compare `OutputPoint` (`PointX, PointY`) with the output point computed in Step 3. If coordinate matches `WrappedPrivateKeyBlob` can be saved anywhere in nonvolatile memory.

This Step 4 code is included in blob creation API function `HAL_CCB_ECC_WrapPrivateKey` described in Step 2.

Step 5 code:

At any time, a scalar multiplication can be computed using `WrappedPrivateKeyBlob` and the HAL_CCB function used in Step 4, with a different `InputPoint`.

3.3 Modular exponentiation

STM32CubeMX proposes a set of functions to compute modular exponentiation with PKA wrapped keys.

3.3.1 Key generation

RSA key generation is out of scope of this document.

3.3.2 Blob creation, blob usage

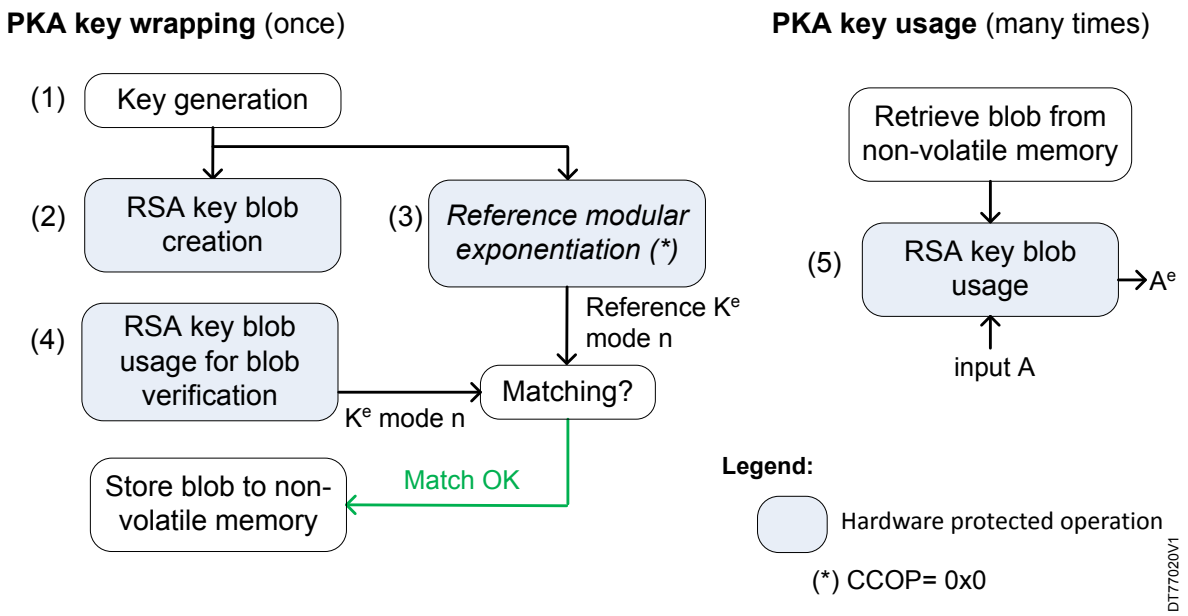
Wrapping an RSA private key corresponds to creating an RSA key blob. Its usage is summarized in Table 4.

Table 4. RSA key blob usage

Attestation	Key import	Key agreement	Key generation	Usage in PKA
X	X	X	-	Modular exponentiation

RSA key blob creation and usage methods are summarized in Figure 5.

Figure 5. RSA key blob creation and usage



STM32CubeMX supports the following methods to implement the flow in Figure 5:

1. Generate keys.
2. Protected RSA key blob creation, with selected RSA parameters and the generated private key as input.
3. Protected modular exponentiation, using a fixed value K and a previously generated private key as input. The result is the reference *K^e mode n*.
4. Protected modular exponentiation, using the fixed value K of step 3 and the previously computed RSA key blob as input. The result is the computed *K^e mode n*. If reference and computed values are identical, save the RSA key blob created in step 1.
5. RSA key blob can be used at any time to compute modular exponentiation using a wrapped key.

Step 1 code:

As the key is not generated on the microcontroller (see Section 3.3.1: Key generation) this part of the code retrieves the RSA key components from local key storage.

Step 2 code:

If the user wants to wrap `RSAClearPrivateKey` with DHUK (or DHUK XOR BHK) select `WrappingKeyType = HAL_CCB_USER_KEY_HW` (or `HAL_CCB_USER_KEY_HSW`).

If the user wants to wrap `RSAClearPrivateKey` with a known `ClearAESKey` this user-defined wrapping key must first be encrypted by calling the HAL_CCB function below, with `WrappingKeyType = HAL_CCB_USER_KEY_WRAPPED`. The recommended algorithm is AES-CBC with 256-bit DHUK, selecting `AES_Algorithm = CCB_AES_CBC` and initializing `pInitVect`.

```
HAL_StatusTypeDef HAL_CCB_RSA_WrapSymmetricKey(CCB_HandleTypeDef *hccb, const uint32_t *pClea
rAESKey,CCB_WrappingKeyTypeDef *pWrappingKey);
```

```
typedef struct
{
uint32_t WrappingKeyType;    /*!< This parameter can be value of
@ref CCB_WrappingKeyTypeDef */
uint32_t AES_Algorithm;    /*!< Used only when wrappingkey type is
HAL_CCB_USER_KEY_WRAPPED, AES Algorithm. This parameter can be a value of @ref CCB_AES_Algori
thm_Mode */
uint32_t *pInitVect;    /*!< Used only when wrappingkey type is
HAL_CCB_USER_KEY_WRAPPED, Pointer to the
initialization vector, counter with CBC Algo */
uint32_t *pUserWrappedWrappingKey;    /*!< Used only when wrappingkey type is
HAL_CCB_USER_KEY_WRAPPED, Pointer to the wrapped wrapping Key */
} CCB_WrappingKeyTypeDef;
```

Wrap the `RSAClearPrivateKey` with the HAL_CCB function below. Keep the `WrappedPrivateKeyBlob` result in volatile memory.

```
HAL_StatusTypeDef HAL_CCB_RSA_WrapPrivateKey(CCB_HandleTypeDef *hccb, CCB_RSAParamTypeDef *pP
aram,
CCB_RSAClearKeyTypeDef *pRSAClearPrivateKey,
CCB_WrappingKeyTypeDef *pWrappingKey,
CCB_RSAKeyBlobTypeDef *pWrappedPrivateKeyBlob);
```

Step 3 code

Generate the reference K^e *mode n* with this HAL_PKA function below. The fixed value K is `Op1`, while the private key is `Exp` and `Phi`.

```
HAL_StatusTypeDef HAL_PKA_ModExpProtectMode(PKA_HandleTypeDef *hpka, PKA_ModExpProtectModeInT
ypeDef *in, uint32_t Timeout);
```

```
typedef struct
{
  uint32_t expSize;                    /*!< Size of the operand in bytes */
  uint32_t OpSize;                     /*!< Size of the operand in bytes */
  const uint8_t *pOp1;                 /*!< Pointer to Operand 1 */
  const uint8_t *pExp;                 /*!< Pointer to Exponent */
  const uint8_t *pMod;                 /*!< Pointer to modulus */
  const uint8_t *pPhi;                 /*!< Pointer to Phi value */
} PKA_ModExpProtectModeInTypeDef;
```

This Step 3 code is included in blob creation API function `HAL_CCB_RSA_WrapPrivateKey` described in Step 2.

Step 4 code:

Generate K^e *mode n* with this HAL_CCB function and the input `WrappedPrivateKeyBlob` computed in Step 2. The fixed value K used in Step 3 must be written as `Operand`.

```
HAL_StatusTypeDef HAL_CCB_RSA_ComputeModularExp(CCB_HandleTypeDef *hccb, CCB_RSAParamTypeDef
*pParam,
CCB_WrappingKeyTypeDef *pWrappingKey,
CCB_RSAKeyBlobTypeDef *pWrappedPrivateKeyBlob, uint8_t *pOperand,
uint8_t *pModularExp);
```

Compare `ModularExp` with the reference computed in Step 3. If it matches `WrappedPrivateKeyBlob` can be saved anywhere in nonvolatile memory.

This Step 4 code is included in blob creation API function `HAL_CCB_RSA_WrapPrivateKey` described in Step 2.

Step 5 code:

At any time, a modular exponentiation can be computed using `WrappedPrivateKeyBlob` and the HAL_CCB function used in Step 4. Any value can be written as `Operand`.

4 Conclusion

In IoT devices, the usage of cryptographic private keys can be critical for device security. This STM32 MCU offers the PKA key wrapping mechanism to ensure the usability of those private keys without disclosing them in cleartext. STM32CubeMX has a set of functions that makes this feature easier to use for developers.

Revision history

Table 5. **Document revision history**

Date	Version	Changes
03-Mar-2025	1	Initial release.

Contents

List of tables

List of figures