whiteCryption®

Secure Key Box 4.28

User Guide

Copyright Information

Disclaimer

Notice to U.S. Government End Users

Contact Information

whiteCryption Corporation, 920 Stewart Drive, Suite 100, Sunnyvale, California 94085, USA

contact@whitecryption.com

www.whitecryption.com

Table of Contents

1 Introduction . . . 9
- 1.1 General Concepts . . . 9
  - 1.1.1 What Is SKB? . . . 9
  - 1.1.2 Nomenclature . . . 9
  - 1.1.3 Purpose of SKB . . . 10
  - 1.1.4 White-Box Cryptography . . . 10
  - 1.1.5 Secure Data Objects . . . 10
  - 1.1.6 Export Key . . . 11
  - 1.1.7 Loading the First Key . . . 11
  - 1.1.8 Diversification . . . 13
  - 1.1.9 Tamper Resistance . . . 13
  - 1.1.10 Evaluation and Production Packages . . . 15
- 1.2 Supported Algorithms . . . 16
- 1.3 Supported ECC Curves . . . 18
- 1.4 Supported Target Platforms . . . 19
- 1.5 Package Structure and Contents . . . 19
- 1.6 Limitations and Known Problems . . . 21

2 Building Applications Protected by SKB . . . 22
- 2.1 Building a Protected Application . . . 22
- 2.2 Distributing a Protected Application . . . 23
  - 2.2.1 Safe Components . . . 23
  - 2.2.2 Unsafe Components . . . 23
- 2.3 Improving Execution Speed . . . 24
  - 2.3.1 Optimizing the Use of Cipher and Transform Objects . . . 24
  - 2.3.2 Key Caching . . . 24
  - 2.3.3 Using the Cross-Engine Export Format . . . 24
- 2.4 Building Examples, Tests, and the Platform-Specific Library . . . 25
  - 2.4.1 Building for Windows API . . . 25
  - 2.4.2 Building for Windows Runtime and Windows Phone . . . 26
  - 2.4.3 Building for Linux . . . 26
  - 2.4.4 Building for Android . . . 28
  - 2.4.5 Building for OS X and iOS . . . 29
  - 2.4.6 Building for Google Native Client (NaCl) . . . 29
  - 2.4.7 Building for PlayStation 3 . . . 30
  - 2.4.8 Building for MinGW . . . 30

3 Cryptographic Operations . . . 32
- 3.1 Loading Plain Keys . . . 32
- 3.2 Unwrapping Keys . . . 32
  - 3.2.1 Unwrapping Keys Wrapped with the ElGamal ECC Algorithm . . . 33
- 3.3 Wrapping Plain Data . . . 34
- 3.4 Wrapping Keys . . . 34
- 3.5 Exporting Keys . . . 35
- 3.6 Importing Keys . . . 36
- 3.7 Upgrading Exported Keys . . . 36

3.8 Generating Keys . . . 38
3.9 Deriving a Public Key from a Private Key . . . 38
3.10 Encrypting and Decrypting Data . . . 38
3.10.1 Encrypting Data . . . 39
3.10.2 Decrypting Data . . . 39
3.10.3 Using the High-Speed AES . . . 39
3.11 Calculating a Digest . . . 39
3.12 Creating a Signature . . . 40
3.13 Verifying a Signature . . . 41
3.14 Executing the Key Agreement Algorithm . . . 41
3.15 Deriving Keys . . . 42
3.15.1 Deriving a Key as a Substring of Bytes of Another Key . . . 42
3.15.2 Deriving a Key as Odd or Even Bytes of Another Key . . . 43
3.15.3 Deriving a Key by Encrypting or Decrypting an Existing Key . . . 43
3.15.4 Deriving a Key as a Protected Hash Value of Another Key . . . 43
3.15.5 Reversing the Order of Bytes of a Key . . . 44
3.15.6 Using the NIST 800-108 Key Derivation Function . . . 45
3.15.7 Using KDF2 of the RSAES-KEM-KWS Scheme Defined in the OMA DRM Specification 45
3.15.8 Deriving a Key as Raw Bytes from a Private ECC Key . . . 45
3.15.9 Deriving a Key By Encrypting Data Using 128-bit AES With a Concatenated Key . . . 46
3.15.10 Deriving a Key Using the CMLA Key Derivation Function . . . 47
3.15.11 Deriving a Key By XOR-ing It with Plain Data or Another Key . . . 47
3.16 Binding Keys to a Specific Device . . . 48
3.17 Decrypting Encrypted PDF Documents . . . 48
4 Supporting Libraries . . . 49
4.1 Sensitive Operations Library . . . 49
4.1.1 Overview . . . 49
4.1.2 Library Functions . . . 49
4.2 Platform-Specific Library . . . 55
4.2.1 Overview . . . 56
4.2.2 Library Functions . . . 56
4.2.3 Enabling Key Caching . . . 58
5 Utilities . . . 61
5.1 Key Export Tool . . . 61
5.1.1 Key Export Tool Overview . . . 61
5.1.2 Running the Key Export Tool . . . 62
5.2 Binary Update Tool . . . 63
5.2.1 Binary Update Tool Overview . . . 64
5.2.2 Running the Binary Update Tool . . . 64
5.3 Custom ECC Tool . . . 65
5.3.1 Custom ECC Tool Overview . . . 65
5.3.2 Parameter Size and Value Restrictions . . . 65
5.3.3 Running the Custom ECC Tool . . . 66
5.4 Diffie-Hellman Tool . . . 66
5.4.1 Diffie-Hellman Tool Overview . . . 66
5.4.2 Running the Diffie-Hellman Tool . . . 67
6 Decrypting PDF Files . . . 69
6.1 PDF Encryption Overview . . . 69

6.2 PDF Requirements . . . 69
6.3 Decrypting a PDF Document Using SKB . . . 69
6.3.1 SKB_Pdf_AuthenticateUserPassword . . . 70
6.3.2 SKB_Pdf_ComputeEncryptionKey . . . 71
6.3.3 SKB_Pdf_CreateDecryptionContext . . . 72
6.3.4 SKB_Pdf_DecryptionContext_ProcessBuffer . . . 73
6.3.5 SKB_Pdf_DecryptionContext_Release . . . 74
7 API Reference . . . 75
7.1 API Overview . . . 75
7.2 Obtaining Class Instances . . . 75
7.3 Making Method Calls . . . 75
7.4 Method Return Values . . . 76
7.5 Object Lifecycle . . . 77
7.6 Restrictions of Multi-Threading . . . 77
7.7 Overriding Memory Allocation Operators . . . 78
7.8 Classes . . . 78
7.8.1 SKB_Engine . . . 79
7.8.2 SKB_SecureData . . . 79
7.8.3 SKB_Cipher . . . 79
7.8.4 SKB_Transform . . . 79
7.8.5 SKB_KeyAgreement . . . 79
7.9 Methods . . . 79
7.9.1 SKB_Engine_GetInstance . . . 79
7.9.2 SKB_Engine_Release . . . 80
7.9.3 SKB_Engine_SetDeviceId . . . 80
7.9.4 SKB_Engine_GetInfo . . . 81
7.9.5 SKB_Engine_CreateDataFromWrapped . . . 81
7.9.6 SKB_Engine_CreateDataFromExported . . . 83
7.9.7 SKB_Engine_WrapDataFromPlain . . . 84
7.9.8 SKB_Engine_GenerateSecureData . . . 85
7.9.9 SKB_Engine_CreateCipher . . . 86
7.9.10 SKB_Engine_CreateTransform . . . 87
7.9.11 SKB_Engine_CreateKeyAgreement . . . 88
7.9.12 SKB_Engine_UpgradeExportedData . . . 89
7.9.13 SKB_SecureData_GetInfo . . . 90
7.9.14 SKB_SecureData_Export . . . 90
7.9.15 SKB_SecureData_Wrap . . . 91
7.9.16 SKB_SecureData_Derive . . . 93
7.9.17 SKB_SecureData_GetPublicKey . . . 94
7.9.18 SKB_SecureData_Release . . . 95
7.9.19 SKB_Cipher_ProcessBuffer . . . 95
7.9.20 SKB_Cipher_Release . . . 97
7.9.21 SKB_Transform_AddBytes . . . 97
7.9.22 SKB_Transform_AddSecureData . . . 97
7.9.23 SKB_Transform_GetOutput . . . 98
7.9.24 SKB_Transform_Release . . . 99
7.9.25 SKB_KeyAgreement_GetPublicKey . . . 99
7.9.26 SKB_KeyAgreement_ComputeSecret . . . 100

7.9.27 SKB_KeyAgreement_Release . . . 101
7.10 Supporting Structures . . . 101
7.10.1 SKB_EngineInfo . . . 101
7.10.2 SKB_EngineProperty . . . 102
7.10.3 SKB_DataInfo . . . 102
7.10.4 SKB_CtrModeCipherParameters . . . 103
7.10.5 SKB_DigestTransformParameters . . . 103
7.10.6 SKB_SignTransformParameters . . . 103
7.10.7 SKB_SignTransformParametersEx . . . 104
7.10.8 SKB_VerifyTransformParameters . . . 105
7.10.9 SKB_SliceDerivationParameters . . . 106
7.10.10 SKB_SelectBytesDerivationParameters . . . 106
7.10.11 SKB_CipherDerivationParameters . . . 107
7.10.12 SKB_Sha1DerivationParameters . . . 108
7.10.13 SKB_Sha256DerivationParameters . . . 108
7.10.14 SKB_Nist800108CounterCmacAes128Parameters . . . 109
7.10.15 SKB_OmaDrmKdf2DerivationParameters . . . 110
7.10.16 SKB_RawBytesFromEccPrivateDerivationParameters . . . 110
7.10.17 SKB_ShaAesDerivationParameters . . . 111
7.10.18 SKB_GenericDerivationParameters . . . 111
7.10.19 SKB_EccDomainParameters . . . 112
7.10.20 SKB_AesWrapParameters . . . 113
7.10.21 SKB_AesUnwrapParameters . . . 113
7.10.22 SKB_RsaPssParameters . . . 113
7.10.23 SKB_EccParameters . . . 114
7.10.24 SKB_PrimeDhParameters . . . 115
7.10.25 SKB_EcdhParameters . . . 116
7.10.26 SKB_RawBytesParameters . . . 116
7.11 Enumerations . . . 116
7.11.1 SKB_DataType . . . 117
7.11.2 SKB_DataFormat . . . 117
7.11.3 SKB_CipherAlgorithm . . . 118
7.11.4 SKB_CipherDirection . . . 120
7.11.5 SKB_SignatureAlgorithm . . . 121
7.11.6 SKB_DigestAlgorithm . . . 124
7.11.7 SKB_DerivationAlgorithm . . . 124
7.11.8 SKB_TransformType . . . 126
7.11.9 SKB_ExportTarget . . . 127
7.11.10 SKB_EccCurve . . . 127
7.11.11 SKB_KeyAgreementAlgorithm . . . 128
7.11.12 SKB_PrimeDhLength . . . 129
7.11.13 SKB_CbcPadding . . . 129
7.11.14 SKB_SelectBytesDerivationVariant . . . 129
8 Data Formats . . . 131
8.1 Data Encrypted with an Export Key . . . 131
8.2 AES-Wrapped Data Buffer . . . 132
8.2.1 ECB Mode . . . 132
8.2.2 CTR Mode . . . 132

8.2.3 CBC Mode 132
8.3 Key Format for the Triple DES Cipher 133
8.4 Input Buffer for the ElGamal ECC Cipher 134
8.5 Public ECC Key 135
8.6 Private ECC Key 135
8.7 AES-Wrapped Private ECC Key 136
8.8 ECDSA Output 136

1 Introduction

This chapter provides a general overview of the Secure Key Box (SKB) technology.

1.1 General Concepts

This section describes the general concepts that you should understand before working with SKB.

1.1.1 What Is SKB?

SKB is a library that provides a set of high-level classes and methods for working with common cryptographic algorithms. The library's white-box implementation hides and protects cryptographic keys. In SKB, keys are encrypted and cryptographic algorithms operate directly with encrypted keys.

SKB exposes an API that provides access to a set of functions, which the calling application uses to implement various cryptographic operations (see Figure 1.1).

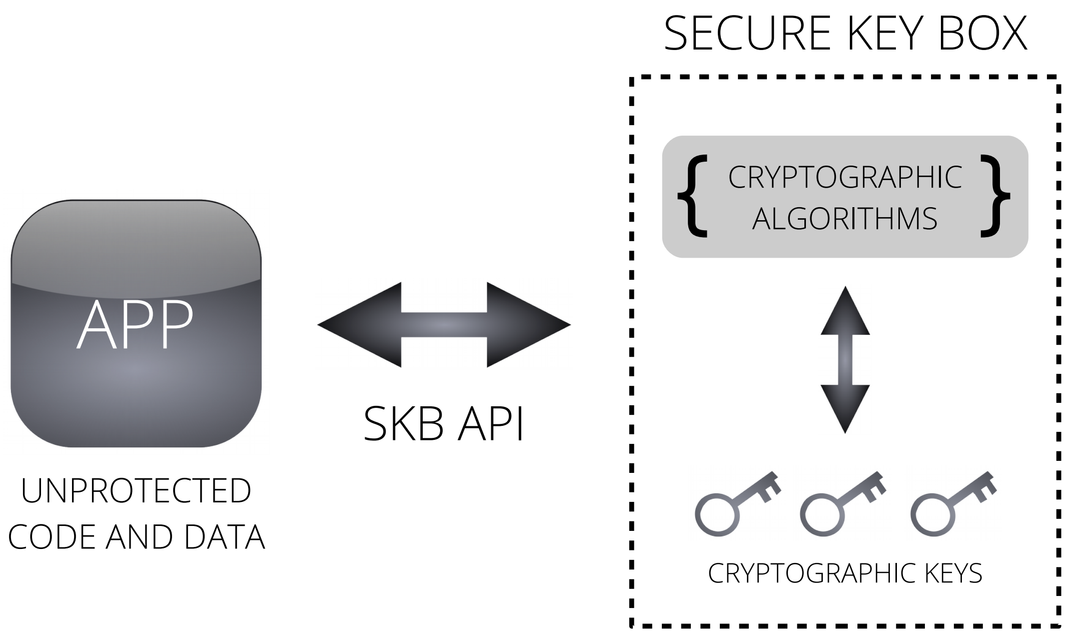

Figure 1.1: SKB overview

1.1.2 Nomenclature

This User Guide uses the terms “secure”, “protect”, “white-box protected”, “safe”, “tamper resistance” and variations of each to convey very specific concepts — indeed, concepts that are far more specific and limited in their meanings than many meanings often associated with such terms in everyday usage. At the risk of stating the obvious, as used herein, none of these terms describe an absolute condition. Use of SKB in compliance with this User Guide will not render any application or data absolutely secure, absolutely protected or absolutely safe from unauthorized accessing, use or manipulation. Nor will it render any application or data absolutely tamper resistant. In addition, use of these terms is not intended to convey a promise or warranty that SKB will never contain a bug or error, or that SKB will always operate without error.

As used in this User Guide:

- “secure” and variations of “secure” refer to data objects, the values of which reside in a cryptographic container and are white-box protected, and that can be operated on by SKB functions despite the fact that they are not revealed in plain form.
- “protected”, “white-box protected” and variations of these terms mean that a value has been subjected to some cryptographic processing that has resulted in it being placed in a container that seeks to render the value inaccessible in plain form to the outside world.
- “safe”, “safely”, “safer” and variations of these terms refer to actions or objects that when processed in accordance with this User Guide will not compromise the security protections provided by SKB.
- “tamper resistance” and variations of this term refer to the application of whiteCryption’s Code Protection product to render it more difficult for unauthorized parties to engage in reverse engineering and code modification.

1.1.3 Purpose of SKB

Cryptographic algorithms and keys are used to protect sensitive data, authenticate communication partners, verify signatures, and implement various other security schemes. A common weak point of cryptographic algorithms in today’s open architectures, such as smartphones, tablets, and desktops, is that the cryptographic keys are revealed in the code or memory at some point. Hackers can monitor such devices with special tools and extract the secret cryptographic keys. Without an efficient protection of cryptographic keys, security features may be compromised.

SKB is designed to prevent such attacks by encrypting and hiding cryptographic keys in the code and memory.

1.1.4 White-Box Cryptography

The term “white-box cryptography” is used to describe a secure implementation of cryptographic algorithms in an execution environment that is fully observable and modifiable by an attacker, such as a desktop computer or a mobile device. It is different from black-box cryptography where the algorithm’s internal processing data is unavailable to the attacker. The white-box environment puts certain restrictions on implementations of the cryptographic algorithms. For instance, an encryption key may never appear in plain text; otherwise it can be retrieved by an attacker.

1.1.5 Secure Data Objects

A secure data object (represented by the `SKB_SecureData` class in the API) is one of the basic concepts in the SKB protection scheme. It is a container of any sensitive data whose value is white-box protected and hidden from the outside world. Secure data objects can be operated on by SKB functions even though the contents of secure data objects are not revealed in plain form.

Because secure data objects usually hold cryptographic keys, in this document the terms “secure data object” and “cryptographic key” are used interchangeably.

1.1.6 Export Key

Secure data objects (cryptographic keys) are operated on in the device memory. Because the memory is not persistent, there needs to be a mechanism for safely storing secure data objects.

A special key called the export key is used for this purpose. This secret key is embedded into each SKB package and is used for encrypting other cryptographic keys exported from the protected SKB domain into the unprotected outside world. To import the exported data back into SKB, the same export key must be used (see Figure 1.2).

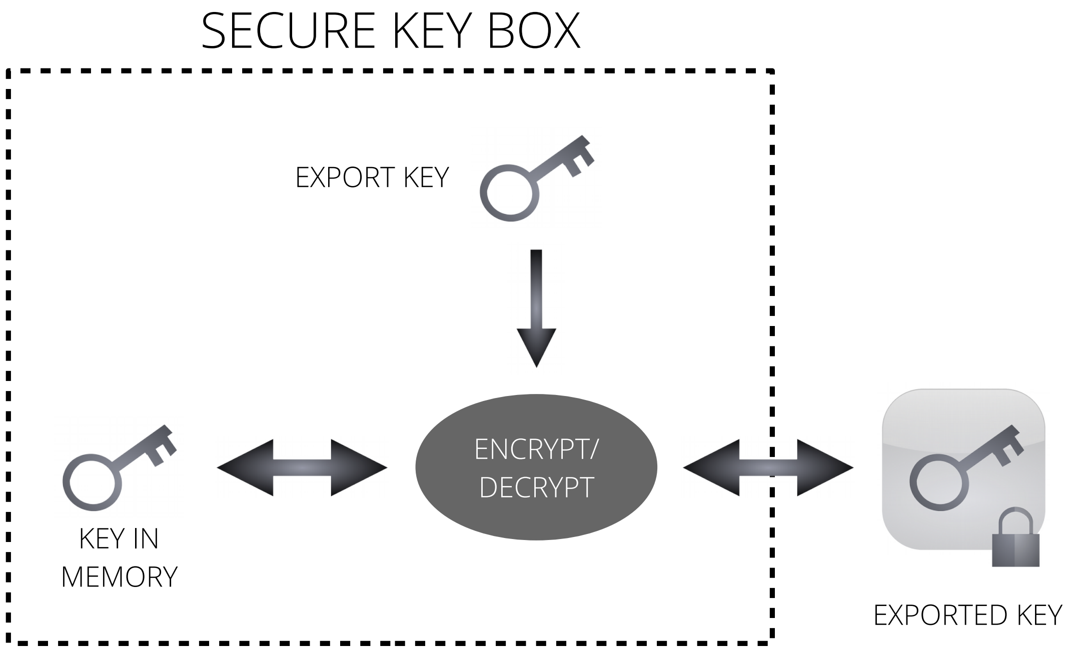

Figure 1.2: Using the export key

The export key is white-box protected, which means that it is not revealed in plain form in the SKB code or device memory. If different SKB packages share the same export key, data exported from one SKB package can be imported into the other SKB package.

1.1.7 Loading the First Key

Typically, if you need to load a new key, you encrypt it with another key, pass it to SKB, and then internally decrypt it as a new secure data object. However, at some point the first key needs to be loaded into SKB. Figure 1.3 shows four ways how the first key can be obtained.

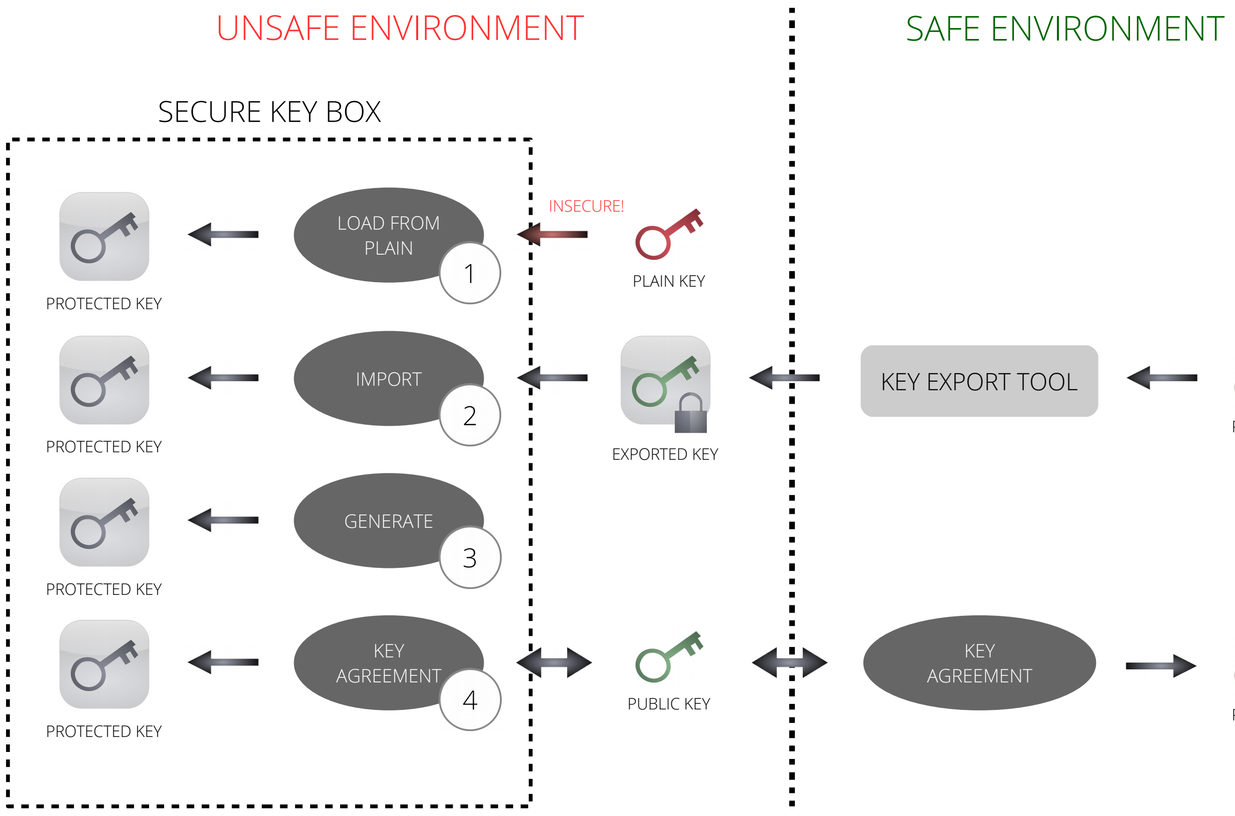

Figure 1.3: Different ways of loading the first cryptographic key into SKB

Figure 1.3 features two environments. The unsafe environment (on the left) is where potentially anyone can gain access to the hardware, code, and memory of the device (such as a desktop computer or mobile device). This is the environment where SKB is primarily intended to be used.

⚠ As a general rule, secret cryptographic keys should never appear in unsafe environments in plain form.

The safe environment (on the right) is a place where cryptographic keys are allowed to be exposed in plain form. You must maintain this environment as safe as possible in terms of potential risks of breaking in, reverse engineering, and exposure to unwelcome parties. Examples of environments that might be considered safe are closed-off facilities, encrypted servers with controlled access, and so on.

The following are the four methods of loading the first key into SKB, as shown in Figure 1.3:

1. Load from plain (see §3.1)

SKB directly loads a plain key.

⚠ This is an insecure approach for loading keys and should be avoided if possible. In normal circumstances, loading of plain keys is disabled in SKB.

2. Import (see §3.6)

SKB imports a key that is presented in a protected form, which is obtained using the Key Export Tool (see §5.1).

⚠ The Key Export Tool must never be delivered with the final protected application (see §2.2).

3. Key generation (see §3.8)

SKB internally generates a new key.

4. Key agreement (see §3.14)

SKB exchanges public keys with another party, and then internally generates a secret key.

1.1.8 Diversification

Diversification is a significant feature of SKB. It means that each customer receives one or several packages whose binary code differs from other packages. SKB achieves this by generating unique representations of white-box algorithms individually for each customer. Although the API provided by each SKB instance is the same, the way the operations are physically implemented in the program code varies.

This feature improves security. For example, if an adversary manages to compromise a particular system that uses SKB, systems of other customers would not be directly affected.

1.1.9 Tamper Resistance

Tamper resistance is an optional feature that you can request for the SKB library delivered to you. Tamper resistance guards the library code against analysis and modification. Although this feature slightly reduces performance of the protected application, it significantly increases security against hacker attacks.

SKB tamper resistance is implemented using Code Protection, a comprehensive tool for hardening software applications on multiple platforms. For information on Code Protection, see www.whitecryption.com/code-protection.

⚠ If your SKB package has tamper resistance applied, you have to run the Binary Update Tool on your final application executable every time it is built. Otherwise, the application will crash at run time with a segmentation fault. For more information on running the Binary Update Tool, see §5.2.

1.1.9.1 Security Features

SKB's implementation of tamper resistance consists of a combination of the following security features:

Integrity protection

Hundreds of embedded overlapping checksums can prevent modifications of the binary code of the entire executable (not just SKB).

Code obfuscation

Library code is transformed to make it difficult to analyze and reverse engineer.

Anti-debug protection

Platform-specific anti-debug code adds protection against main-stream debuggers providing another barrier to code analysis.

iOS jailbreak detection

Normally, a cracked or modified iOS application can be run only on jailbroken iOS devices. iOS jailbreak detection protects the application from being executed on a jailbroken device.

Android rooting detection

Rooting is a security risk to Android applications that deal with sensitive data or enforce certain usage restrictions. Rooting detection will protect the application if a rooted device is detected.

Inlining of static void functions

Static void functions with simple declarations are inlined into the calling functions. Such operation increases the obfuscation level of the final protected code and makes it more difficult to trace.

String literal obfuscation

Large portion of string literals, or string constants, are encrypted in the code and are decrypted only before they are actually used. The purpose of this feature is to hide useful and sensitive information from potential attackers.

Customizable defense action

Optionally, you can request a tamper resistance SKB library that is configured to execute specific callback functions depending on the type of attack it detects. Additionally, when requesting the SKB library, you can choose whether the program state should be corrupted or the application should be left running after a callback function is invoked. For more information on this feature, see §1.1.9.3.

1.1.9.2 Supported Platforms

Tamper resistant SKB libraries are currently available only for the following target platforms:

- Windows for Visual Studio 2010, 2012, and 2013 (x86 and x86_64)
- GNU/Linux (x86 and x86_64)
- OS X (x86 and x86_64)
- iOS (ARMv7 and ARM64)
- Android (armeabi, armeabi-v7a, arm64-v8a, and x86)

1.1.9.3 Callback Functions

When you request a tamper resistant edition of SKB, you may optionally specify whether you want SKB to invoke specific callback functions when threats are detected.

If you do not choose to use callback functions, SKB will corrupt the application state (typically, resulting in a crash) whenever it detects a threat. Callback functions allow you to implement custom responses to attacks. Additionally, when requesting an SKB library that uses callback functions, you may also specify if you want the attacked application to continue execution after a callback function is invoked.

If the SKB library that you received is configured to use callbacks, you have to provide implementation for the callback functions in the source code. The following are the attack types for which callback functions are supported:

Debugger

The following function is called by SKB when it detects that the application is run under a debugger.

```
void SKB_Callback_AntiDebug()
```

Android rooting

The following function is called by SKB when it detects that the application is run on a rooted Android device.

```
Rooting void SKB_Callback_Root()
```

iOS jailbreak

The following function is called by SKB when it detects that the application is run on a jailbroken iOS device.

```
void SKB_Callback_Jailbreak()
```

Please note that you have to provide implementations for the above functions only if you specifically requested an SKB library that uses them.

1.1.10 Evaluation and Production Packages

Two editions of the SKB package are available to customers:

Evaluation

The evaluation package is free and is typically given to new customers who want to try out SKB before purchasing a license. You should not use an evaluation edition in a production environment for two reasons. Firstly, you do not have the right to do so. Secondly, all evaluation packages have the same export key (see §1.1.6). This means that all encrypted data that you export from SKB can be decrypted by other evaluation packages.

Additionally, evaluation packages have an expiration date set. Once the date is reached, SKB will no longer be usable.

Production

The production package is given to customers who have licensed SKB. Each production package has a unique export key and no expiration date.

1.2 Supported Algorithms

This section lists the main cryptographic algorithms supported by SKB. Note that your SKB distribution includes only those algorithms that you specifically requested from whiteCryption.

Encryption (see §3.10.1)

- DES in ECB mode (no padding) and CBC mode (no padding)
- Triple DES in ECB mode (no padding) and CBC mode (no padding) with two keying options:
  - All three keys are distinct.
  - Key 1 is the same as key 3.
- 128-bit, 196-bit, and 256-bit AES in ECB mode (no padding), CBC mode (no padding), and CTR mode

Decryption (see §3.10.2)

- DES in ECB mode (no padding) and CBC mode (no padding)
- Triple DES in ECB mode (no padding) and CBC mode (no padding) with two keying options:
  - All three keys are distinct.
  - Key 1 is the same as key 3.
- 128-bit, 196-bit, and 256-bit AES in ECB mode (no padding), CBC mode (no padding), and CTR mode
- 1024-bit to 2048-bit RSA (no padding, PKCS#1 version 1.5 padding, or OAEP padding with MD5, SHA-1, SHA-224, SHA-256, SHA-384, or SHA-512)

  The size of the private RSA keys must be a multiple of 128 bits.
- ElGamal ECC (for supported curve types, see §1.3)

Signing (see §3.12)

- 128-bit AES-CMAC (based on OMAC1)
- HMAC using MD5, SHA-1, SHA-224, SHA-256, SHA-384, or SHA-512 as the hash function
- 1024-bit to 2048-bit RSA signature algorithms standardized in version 1.5 of PKCS#1 using MD5, SHA-1, SHA-224, SHA-256, SHA-384, or SHA-512 as the hash function

  The size of the private RSA keys must be a multiple of 128 bits.

  You may also use this algorithm without a hash function. In that case, SKB will expect that you provide input that is already hashed.
- 1024-bit to 2048-bit RSA signature algorithms based on the Probabilistic Signature Scheme using MD5, SHA-1, SHA-224, SHA-256, SHA-384, or SHA-512 as the hash function

  The size of the private RSA keys must be a multiple of 128 bits.
- ECDSA using MD5, SHA-1, SHA-224, SHA-256, SHA-384, or SHA-512 as the hash function (for supported curve types, see §1.3)

  You may also use this algorithm without a hash function. In that case, SKB will expect that you provide input that is already hashed.

Verification (see §3.13)

- 128-bit AES-CMAC (based on OMAC1)

- HMAC using MD5, SHA-1, SHA-224, SHA-256, SHA-384, or SHA-512 as the hash function

Unwrapping (see §3.2)

- unwrapping raw bytes using 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding), CBC mode (no padding or XML encryption padding), and CTR mode
- unwrapping private RSA keys using 128-bit, 192-bit, and 256-bit AES in CBC mode (XML encryption padding) and CTR mode
- unwrapping private ECC keys using 128-bit, 192-bit, and 256-bit AES in ECB mode (if the length of the private ECC key is a multiple of 128 bits), CBC mode (no padding or XML encryption padding) and CTR mode
- unwrapping raw bytes using 1024-bit to 2048-bit RSA (no padding, PKCS#1 version 1.5 padding, or OAEP padding with SHA-1)

  The size of the private RSA keys must be a multiple of 128 bits.
- unwrapping raw bytes using ElGamal ECC (for supported curve types, see §1.3)
- AES key unwrapping defined by NIST with 128-bit, 192-bit, and 256-bit AES keys
- CMLA AES unwrapping defined by the *CMLA Technical Specification*
- CMLA RSA unwrapping defined by the *CMLA Technical Specification*
- unwrapping using XOR

Wrapping (see §3.4)

- wrapping raw bytes using 128-bit, 192-bit, and 256-bit AES in CBC mode (XML encryption padding)
- wrapping raw bytes using 1024-bit to 2048-bit RSA (no padding, PKCS#1 version 1.5 padding, or OAEP padding with SHA-1)

  The size of the private RSA keys must be a multiple of 128 bits.
- wrapping private ECC keys using 128-bit, 192-bit, and 256-bit AES in CBC mode (XML encryption padding)
- wrapping plain data using 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding) and CBC mode (no padding)
- wrapping using XOR

Key generation (see §3.8)

- random buffer of bytes (for example, DES and AES keys)
- ECC key pairs (for supported curve types, see §1.3)

Key agreement (see §3.14)

- Classical Diffie-Hellman (DH) with up to 1024-bit prime P
- Elliptic curve Diffie-Hellman (ECDH) (for supported curve types, see §1.3)

Digests (see §3.11)

- MD5
- SHA-1

- SHA-224
- SHA-256
- SHA-384
- SHA-512

Key derivation (see §3.15)

- slicing (selecting a substring of bytes from another key)
- selecting odd or even bytes
- encrypting and decrypting raw bytes using the following algorithms:
  - 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding) and CBC mode (no padding)
  - DES in ECB mode (no padding) and CBC mode (no padding)
- iterated SHA-1
- SHA-256 with plain prefix and suffix
- SHA-384
- byte reversing
- NIST 800-108 key derivation with 128-bit AES-CMAC in counter mode
- KDF2 used in the RSAES-KEM-KWS scheme of the Open Mobile Alliance (OMA) DRM specification
- deriving raw bytes from a private ECC key
- CMLA key derivation defined by the *CMLA Technical Specification*
- 128-bit AES encryption in ECB mode (no padding) with a concatenated key and optional SHA-1 function
- XOR-ing a key with either plain data, or another key

1.3 Supported ECC Curves

SKB supports the following ECC curve types:

- 160-bit prime curve recommended by SECG, SECP R1
- 192-bit prime curve recommended by NIST (same as 192-bit SECG, SECP R1)
- 224-bit prime curve recommended by NIST (same as 224-bit SECG, SECP R1)
- 256-bit prime curve recommended by NIST (same as 256-bit SECG, SECP R1)
- 384-bit prime curve recommended by NIST (same as 384-bit SECG, SECP R1)
- 521-bit prime curve recommended by NIST (same as 521-bit SECG, SECP R1)
- 150-bit to 521-bit prime ECC curves with custom domain parameters

⚠ ElGamal ECC decryption and ElGamal ECC key unwrapping support only 160-bit, 192-bit, 224-bit, and 256-bit prime curves. ECC key generation, ECDSA, and ECDH support all the listed ECC curve types, including ECC curves with custom domain parameters.

1.4 Supported Target Platforms

Table 1.1 lists operating systems and architectures supported by SKB, and build systems used to build and test the SKB library. Use of other platforms may require additional support from us.

Platform	**Architectures**	**Build systems**
Windows Vista/7/8/8.1/10 (Windows API)	x86, x86_64	Visual Studio 2010, 2012, and 2013
Windows 8/8.1 (Windows Runtime)	x86, x86_64	Visual Studio 2012 and 2013
OS X	x86, x86_64	Xcode 7.1.1
GNU/Linux	x86, x86_64	GCC 4.9
Android	x86, x86_64, ARM (32-bit and 64-bit), MIPS (32-bit and 64-bit)	Android NDK r10d
iOS	ARM (32-bit and 64-bit)	Xcode 7.1.1
Windows Phone 8	ARM	Visual Studio 2012
Windows Phone 8.1	ARM	Visual Studio 2013
Google Native Client (NaCl)	X86, x86_64, ARM, PNaCl	Native Client SDK Pepper 39
PlayStation 3	Cell (PPU)	PlayStation 3 Programmer Tool Runtime Library 460.001
uClibc/Linux	ARM, MIPS, MIPSel	Buildroot 2015.02
MinGW	x86	MinGW-w64 i686-4.9.2-posix-dwarf-rt_v3-rev1
MinGW	x86_64	MinGW-w64 x86_64-4.9.2-posix-seh-rt_v3-rev1

Table 1.1: Supported target platforms

1.5 Package Structure and Contents

The SKB package delivered to you contains the following main folders and files that you should be aware of:

`/Build/Targets/`

Contains several subfolders for supported target operating systems and architectures. Each subfolder contains build files required for building SKB examples, tests, and the Platform-Specific Library for a particular target platform.

For information on building SKB examples, tests, and Platform-Specific Library, see §2.4.

`/Documents/`

Contains SKB documentation.

`/Examples/`

Contains SKB examples. For information on building the examples, see §2.4.

`/Include/`

Contains the `SkbSecureKeyBox.h` file, which is the entire public interface of SKB (see §7). This is the main API that you use with SKB.

`/Libraries/`

Contains the following binaries for different target platforms:

- main SKB library
- Sensitive Operations Library (see §4.1)
- Platform-Specific Library (see §4.2)
- Custom ECC Tool (see §5.3)
- Diffie-Hellman Tool (see §5.4)
- Key Export Tool (see §5.1)
- SQLite library (see §4.2.3)
- Binary Update Tool (see §5.2)

> ⚠ If you have ordered a tamper resistant SKB library (see §1.1.9), you always have to run the Binary Update Tool on the final protected application once it is built as described in §5.2. Otherwise, the protected application will crash at run time with a segmentation fault.

`/SpeedTests/`

Contains speed tests for measuring the performance of various cryptographic algorithms. For information on building tests, see §2.4.

`/Test/`

Contains unit tests. You can compile and run them to verify that SKB is running correctly. For information on building tests, see §2.4.

`/Tools/SkbPlatform/`

Contains source code for the Platform-Specific Library. This library is also available in a binary format, in the `Libraries` folder. For more information on Platform-Specific Library, see §4.2.

`/export.id`

Text file containing the identifier of the export key (see §1.1.6) included in this SKB instance, as well as identifiers of all export keys whose exported data this SKB instance is capable of upgrading (see §3.7).

The file structure is as follows:

```
Legacy key 0 ID: «export key identifier»
Legacy key 1 ID: «export key identifier»
Legacy key 2 ID: «export key identifier»
...
Current key version «N» ID: «export key identifier»
```

"`Legacy key`" specifies identifiers of export keys whose exported data can be upgraded by this SKB instance. Current key specifies the identifier of the export key that is used by this SKB instance to encrypt exported data.

To find out which export key was used to export particular data, you can compare export key identifiers in this file to the export key identifier in the header of exported data as described in §8.1.

`/SConstruct`

This file is used by SCons to build the source files delivered along with SKB. For information on using SCons for building the files, see §2.4.

1.6 Limitations and Known Problems

Please carefully review the following list of limitations and known problems before including SKB into your applications:

- Features utilizing RSA or ECC custom curve algorithms do not work on PlayStation 3.
- Before running SKB examples, speed tests, and unit tests on PlayStation 3, the `make_fself` tool included in the PS3 SDK has to be run on the executables.
- The Platform-Specific Library (see §4.2) and the main SKB library have a circular dependency. This means that if you are building your application using a build system other than Visual Studio and Xcode, you have to make sure the SKB library is linked twice using the linking sequence: `SecureKeyBox` > `SkbPlatform` > `SecureKeyBox`.

2 Building Applications Protected by SKB

This chapter describes the recommended manner to build and deploy an application that is integrated with SKB. Following these steps is important to achieve the maximum security provided by SKB. It also provides instructions for building SKB examples, tests, and the Platform-Specific Library (see §4.2) for different target platforms.

2.1 Building a Protected Application

SKB is delivered as a precompiled static library. The public interface to this library is described in the `SkbSecureKeyBox.h` file, which is located in the `Include` folder.

To build an application protected by SKB, you must perform the following main steps:

1. Optionally, to reduce the binary size of the application, disable those SKB modules that you do not use, as described in §4.2.1.2.
2. Link your application with the following libraries:
   - appropriate `SecureKeyBox` library from the `Libraries` folder, depending on the target platform
   - Platform-Specific Library (`SkbPlatform`) (see §4.2)

     ⚠ The Platform-Specific Library and the main SKB library have a circular dependency. This means that if you are building your application using a build system other than Visual Studio and Xcode, you have to make sure the SKB library is linked twice using the linking sequence: `SecureKeyBox` > `SkbPlatform` > `SecureKeyBox`.

     ⚠ A special note applies to the Android target. For Android applications, you have to specify the Platform-Specific Library using the `LOCAL_WHOLE_STATIC_LIBRARIES` option in the `android.mk` file, not using `LOCAL_STATIC_LIBRARIES`.
   - SQLite library if you are using SQLite-based key caching (see §4.2.3)

   ⚠ Make sure you are not linking or distributing any of the unsafe SKB components listed in §2.2.
3. Build your application and make sure you use the following compiler and linker settings depending on your build system:

   Visual Studio
   - enable references to remove unnecessary code (`/OPT:REF`)
   - enable COMDAT folding to remove duplicated code (`/OPT:ICF`)

   GCC
   - if compiling for OS X via the command line, use the `-Wl,-dead_strip` option to remove unnecessary code sections
   - if compiling for Linux, use the `-Wl,-gc-sections` option to remove unnecessary code sections

   Xcode

    - enable the **Deployment Postprocessing** option to remove information that can be used to reverse engineer the code
    - enable the **Strip Linked Product** option to remove information that can be used to reverse engineer the code

4. To ensure that symbol information is correctly stripped from the executable, open the executable in a binary editor and search for a string “whitebox”.

   The string should not be present in the code. If it is, ensure you have completed the items in step 3.

5. If the SKB package delivered to you has tamper resistance applied (see §1.1.9), run the Binary Update Tool on the final built application as described in §5.1.

2.2 Distributing a Protected Application

SKB consists of a number of binary libraries and supporting files. Some of these components are secure and can be safely included in the final protected application. However, some components expose sensitive operations that can lead to key exposure and therefore should be considered insecure. These components serve a specific purpose and are usually not required in the final deliverable that is delivered to end users.

2.2.1 Safe Components

The following components are self-sufficient and can be considered safe:

- SKB library (`SecureKeyBox`)
- Platform-Specific Library (`SkbPlatform`)
- examples
- SQLite library

Including these components in your application will not compromise the security provided by SKB.

2.2.2 Unsafe Components

The following components should be considered unsafe and must never be included in an application that is deployed in an open environment:

- Key Export Tool
- Custom ECC Tool
- Diffie-Hellman Tool
- unit tests and speed tests
- Sensitive Operations Library (`InternalHelpers`)
- utilities (`SkbUtils`)
- `LibTomCrypt` and `LibTomMath` libraries (only required by tests)

These components can only be used on a protected computer that is not accessible to end users.

2.3 Improving Execution Speed

This section provides recommendations for optimizing several SKB operations in order to achieve better execution speed for time-critical applications.

2.3.1 Optimizing the Use of Cipher and Transform Objects

A cipher is an object that performs encryption or decryption of data (see §7.8.3). A transform is an object that performs signing and verification (see §7.8.4). These objects must be created prior to executing the operations they provide. Because creation of cipher and transform objects is a time-consuming operation, consider the following recommendations:

- Create the cipher and transform objects at an appropriate moment ahead of time (for example, immediately after obtaining the keys they will require), not within time-critical functions.
- Release the cipher and transform objects only when you are absolutely sure they will not be used again.

2.3.2 Key Caching

In SKB, initialization of a secure data object that contains a private RSA key (for example, when creating a cipher object or a transform object) is a very time-consuming operation. To address this problem, SKB provides an optional feature called key caching, which significantly speeds up algorithms that deal with private RSA keys.

If enabled, key caching works as follows:

- When a secure data object containing a new private RSA key is initialized for the first time, an expanded form of the key is prepared and then stored in a special key cache.
- Whenever the same RSA key is required again, instead of going through the time-consuming key initialization phase again, the expanded form of the key is immediately retrieved from the key cache.
- When SKB exports a private RSA key that is stored in the key cache, the expanded form is also included in the exported data buffer. When such exported key is imported into SKB, its expanded form is immediately added to the key cache.
- When you create an exported form of a private RSA key using the Key Export Tool (see §5.1), the expanded form of the key will also be included in the exported data buffer. When such exported key is imported into SKB, its expanded form is immediately added to the key cache.

For information on enabling key caching, see §4.2.3.

Currently, key caching is implemented only for private RSA keys.

2.3.3 Using the Cross-Engine Export Format

Normally, the recommended approach for exporting and importing secure data objects (keys) is to encrypt and decrypt them with the export key (see §1.1.6). However, this approach involves the encryption and decryption operations, which may impact time-critical functions. In such cases, a faster alternative is to use the cross-engine export format when exporting or importing keys (see §7.11.9). When the cross-engine export format is used, the exported key is passed to the unprotected outside world in the exact form in which the key exists in the memory, and hence it is very simple to import such a key back into SKB. The performance gain lies in the fact that this approach does not require the encryption or decryption phase.

However, you should be aware of the following disadvantages of the cross-engine export format when compared to the normal export operation:

- Keys exported in this way can only be imported by the very same SKB package that exported it.
- The cross-engine export format does not include the expanded form of private RSA keys intended for the key cache. Therefore, if you use the cross-engine export format to export or import private RSA keys, you should also enable the persistent (SQLite-based) key caching mode (see §4.2.3).

2.4 Building Examples, Tests, and the Platform-Specific Library

A number of additional C++ source files are delivered together with SKB. These files include examples, tests, and the Platform-Specific Library (see §4.2). The Platform-Specific Library is a mandatory component that is necessary for the execution of SKB. Other components are optional.

The following subsections describe the recommended and supported way of building these files for different targets. You may also interpret instructions in these subsections as the recommended way of building your own applications that use the SKB library.

2.4.1 Building for Windows API

Visual Studio is used to build SKB examples, tests, and the Platform-Specific Library. This section describes how to set up the build environment and compile the source files.

2.4.1.1 Prerequisites

To compile the source files, you will need a computer with the Windows operating system that has Visual Studio installed. For information on supported Visual Studio versions, see §1.4.

2.4.1.2 Compiling

To compile the source files, proceed as follows:

1. Open the Visual Studio solution named `SecureKeyBox.sln` in one of the following folders, depending on your needs:

   `Build/Targets/all-microsoft-win32-vs2010`
   Visual Studio 2010 solution for Windows API.

   `Build/Targets/all-microsoft-win32-vs2012`
   Visual Studio 2012 solution for Windows API.

   `Build/Targets/all-microsoft-win32-vs2013`
   Visual Studio 2013 solution for Windows API.

   Each solution contains the following specific projects:

   - `SkbExamples`: runs SKB examples

- `SkbSpeedTests`: runs SKB speed tests
- `SkbTestSuite`: runs SKB unit tests

2. Compile the solution.

2.4.2 Building for Windows Runtime and Windows Phone

Visual Studio is used to build SKB examples, tests, and the Platform-Specific Library. This section describes how to set up the build environment and compile the source files.

2.4.2.1 Prerequisites

To compile the source files, you will need a computer with the Windows operating system that has Visual Studio installed. For information on supported Visual Studio versions, see §1.4.

2.4.2.2 Compiling

To compile the source files, proceed as follows:

1. Open the Visual Studio solution named `SecureKeyBox.sln` in one of the following folders, depending on your needs:

   `Build/Targets/all-microsoft-winrt-vs2012`
   This is the Visual Studio 2012 solution for Windows Runtime.

   `Build/Targets/all-microsoft-winrt-vs2013`
   This is the Visual Studio 2013 solution for Windows Runtime.

   `arm-windows-phone-vs2012`
   This is the Visual Studio 2012 solution for Windows Phone 8.0.

   `arm-windows-phone-vs2013`
   This is the Visual Studio 2013 solution for Windows Phone 8.1.

   Each solution contains the following specific projects:

   - `SkbExamplesApp`: runs SKB examples as a Microsoft design language app
   - `SkbExamplesUnitTest`: runs SKB examples as a Visual Studio unit test
   - `SkbSpeedTestsApp`: runs SKB speed tests as a Microsoft design language app
   - `SkbSpeedTestsUnitTest`: runs SKB speed tests as Visual Studio unit tests
   - `SkbTestSuiteUnitTest`: runs SKB unit tests as Visual Studio unit tests

2. Compile the solution.

2.4.3 Building for Linux

The SCons build tool is used to build SKB examples, tests, and the Platform-Specific Library. This section describes how to set up the build environment and compile the source code.

2.4.3.1 Prerequisites

The following prerequisites must be met before building the source files:

1. Download and install SCons 2.3.0.
2. Depending on your target architecture, download and install the necessary build system as listed in §1.4.

2.4.3.2 Compiling

To compile the source files, go to the root folder of the SKB package and execute the following command:

```
scons target=«your target» build_config=«release type»
```

The following arguments are used:

`target`

Specifies the target platform, corresponding to an appropriate subfolder in the **`Build/Targets`** folder:

- `x86-unknown-linux`: GNU/Linux edition for the x86 architecture
- `x86_64-unknown-linux`: GNU/Linux edition for the x86_64 architecture
- `arm-unknown-linux`: uClibc/Linux edition for the ARM architecture
  This target is built with the following compiler parameters:
  - `-march=armv7-a`
  - `-marm`
  - `-mthumb-interwork`
  - `-mfloat-abi=hard`
  - `-mfpu=neon`
  - `-mtune=cortex-a9`
- `mips-unknown-linux`: uClibc/Linux edition for the MIPS architecture
  This target is built with the "**`-mhard-float`**" compiler parameter.
- `mipsel-unknown-linux`: uClibc/Linux edition for the MIPSel architecture
  This target is built with the "**`-mhard-float`**" compiler parameter.

If the target is not specified, the source files will be built for the default target, which is your build machine.

`build_config`

Specifies whether the binaries should be compiled in release or debug mode. The following values can be set:

- **`Debug`**: Produces binary files in debug mode and places them in the **`Build/Targets/«your target»/Debug`** folder. This is the default value.
- **`Release`**: Produces binary files in release mode and places them in the **`Build/Targets/«your target»/Release`** folder.

2.4.4 Building for Android

Android NDK is used to build SKB examples, tests, and the Platform-Specific Library. This section describes how to set up the build environment, compile the source code, and run the compiled examples.

2.4.4.1 Prerequisites

To build the source files for Android, you will need a computer with Android NDK installed. For information on Android NDK requirements, see §1.4.

Make sure the Android NDK root folder is added to the system's `PATH` variable.

2.4.4.2 Compiling

To compile the source files, proceed as follows:

1. Go to one of the following folders, depending on the architecture used:
   - `Build/Targets/arm64-google-android`
   - `Build/Targets/arm-google-android`
   - `Build/Targets/mips64-google-android`
   - `Build/Targets/mips-google-android`
   - `Build/Targets/x86_64-google-android`
   - `Build/Targets/x86-google-android`
2. Execute the following command:

   ```
   ndk-build APP_OPTIM=«release type»
   ```

   `APP_OPTIM` specifies whether the binaries should be compiled in release or debug mode. The following values can be set:

   - `release` (default value)
   - `debug`

   A special note applies to the `arm-google-android`, `x86-google-android`, and `mips-google-android` targets. By default, SKB examples and tests intended for this target are built for the android-16 API version. If you want to build SKB examples and tests for an older API version, you must add the "`APP_PLATFORM=«API version»`" argument to the `ndk-build` command described above, for example as follows:

   ```
   ndk-build APP_OPTIM=release APP_PLATFORM=android-8
   ```

   The `APP_PLATFORM` argument is not necessary for targets other than `arm-google-android`, `x86-google-android`, and `mips-google-android`.

The compiled binary files will be placed in the `libs` folder.

2.4.4.3 Running

Once the files are compiled, you can transfer the files to the Android device via the Android Debug Bridge (ADB) tool, which is included in the Android SDK.

The following is an example ADB script that copies compiled SKB examples to the `/data/local` folder on the connected Android device (this folder always allows executing files), makes them executable, and runs them:

```
adb shell rm /data/local/SkbExamples
adb push SkbExamples /data/local
adb shell chmod 777 /data/local/SkbExamples
adb shell "cd /data/local/ && ./SkbExamples"
```

2.4.5 Building for OS X and iOS

Xcode is used to build SKB examples, tests, and the Platform-Specific Library. This section describes how to set up the build environment and compile the source code.

2.4.5.1 Prerequisites

To compile the source files, you will need a computer with the OS X system that has Xcode installed. For information on the supported Xcode version, see §1.4.

2.4.5.2 Compiling

To compile the source files, proceed as follows:

1. Go to the `Build/Targets/`*`«your target»`* folder.
2. Open the Xcode project.
3. Select the required scheme.
4. Compile the project.

   The location, where compiled files are placed, depends on the system:

   OS X

   One of the following folders depending on the compilation mode:
   - `Build/Targets/universal-apple-macosx/build/Debug`
   - `Build/Targets/universal-apple-macosx/build/Release`

   iOS

   One of the following folders depending on the compilation mode:
   - `Build/Targets/arm-apple-ios/build/Debug-iphoneos`
   - `Build/Targets/arm-apple-ios/build/Release-iphoneos`

2.4.6 Building for Google Native Client (NaCl)

The SCons build tool is used to build SKB examples, tests, and the Platform-Specific Library. This section describes how to set up the build environment and compile the source code.

2.4.6.1 Prerequisites

The following prerequisites must be met before building the source files:

1. Download and install SCons 2.3.0.
2. Download and install Native Client SDK.

2.4.6.2 Compiling

Compiling for Google Native Client is performed the same way as described in §2.4.3.2, with the `target` parameter set to one of the following, depending on the necessary architecture:

- `nacl/arm`
- `nacl/pnacl`
- `nacl/x86_32`
- `nacl/x86_64`

⚠ If your application is using SKB algorithms that depend on random generation, you must use the `SKB_InitRng` and `SKB_DestroyRng` functions of the Platform-Specific Library as described in §4.2.2.

2.4.7 Building for PlayStation 3

The SCons build tool is used to build SKB examples, tests, and the Platform-Specific Library. This section describes how to set up the build environment and compile the source code.

2.4.7.1 Prerequisites

The following prerequisites must be met before building the source files:

1. Download and install SCons 2.3.0.
2. Download and install the PlayStation 3 Programmer Tool Runtime Library version listed in §1.4.

2.4.7.2 Compiling

Compiling for PlayStation 3 is performed the same way as described in §2.4.3.2, with the `target` parameter set to `ppu-playstation3`.

2.4.8 Building for MinGW

The SCons build tool is used to build SKB examples, tests, and the Platform-Specific Library. This section describes how to set up the build environment and compile the source code.

2.4.8.1 Prerequisites

The following prerequisites must be met before building the source files:

1. Download and install SCons 2.3.0.
2. Download and install the MinGW-w64 version listed in §1.4.
3. Make sure the MinGW-w64 DLL files are available on the execution path using one of the following approaches:
   - Add the MinGW-w64 `bin` folder to the `PATH` environment variable.
   - Copy the `libgcc_s_dw2-1.dll`, `libstdc++-6.dll`, and `libwinpthread-1.dll` files next to the final executables.

2.4.8.2 Compiling

Compiling for MinGW is performed the same way as described in §2.4.3.2, with the target parameter set to one of the following:

- `x86_64-unknown-mingw`
- `x86-unknown-mingw`

3 Cryptographic Operations

This chapter provides high-level task-based information about the main cryptographic operations that can be performed with SKB.

3.1 Loading Plain Keys

SKB is designed to always work with keys in protected form. If a key needs to be loaded into SKB, normally you should deliver and load it in encrypted form. However, for very rare cases, SKB does support direct loading of plain keys.

⚠ Loading a plain key is a very insecure operation and should be avoided if possible. There are better alternatives for achieving this as described in §1.1.7. Normally, loading of plain keys is disabled in SKB.

To directly load a plain key as a secure data object, call the `SKB_Engine_CreateDataFromWrapped` method (see §7.9.5) and specify the unwrapping algorithm `SKB_CIPHER_ALGORITHM_NULL`. In this case, the unwrapping key and other parameters do not have to be provided.

This operation will work only if loading of plain keys is enabled in SKB. If loading of plain keys is disabled, you can use the Sensitive Operations Library to convert plain keys to secure data objects as described in §4.1.

3.2 Unwrapping Keys

A wrapped key is a cryptographic key encrypted with another key. Unwrapping is a process where SKB loads an encrypted key and internally decrypts it using a pre-loaded unwrapping key (see Figure 3.1). Unwrapping is a secure way of loading keys into SKB.

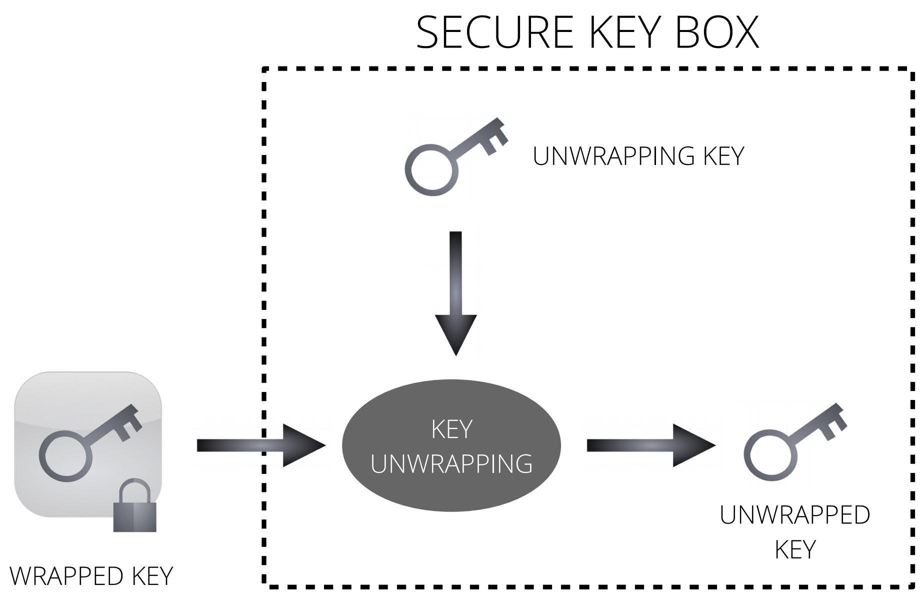

Figure 3.1: Unwrapping a key

Unwrapping is not the same operation as regular decryption, because regular decryption provides the output in plain form (see §3.10.2). In case of unwrapping, the unwrapped (decrypted) key is directly transformed into a secure data object and is never exposed in plain form.

To unwrap a key, call the `SKB_Engine_CreateDataFromWrapped` method (see §7.9.5) and provide the necessary parameters, such as the following:

- wrapped key
- type and format of the wrapped key
- algorithm for unwrapping the data (for the special case of using the ElGamal ECC unwrapping algorithm, see §3.2.1)
- additional parameters for the unwrapping algorithm
- unwrapping key

3.2.1 Unwrapping Keys Wrapped with the ElGamal ECC Algorithm

Since there are no widely accepted standards for storing the output of ElGamal ECC decryption, this section describes the format used by SKB. In connection with this, you may have to perform additional steps to extract the actual unwrapped key from the output as described below.

In the case of the ElGamal ECC unwrapping algorithm, the wrapped buffer of the `SKB_Engine_CreateDataFromWrapped` method (see §7.9.5) should contain two points on an ECC curve as described in §8.4.

After the unwrapping method is successfully executed, the data variable will point to a buffer that contains the X coordinate of the decrypted point on the ECC curve. The actual unwrapped key is stored within the X coordinate using the big-endian notation. You must then extract the unwrapped key bytes from the X coordinate using the `SKB_SecureData_Derive` method and the `SKB_DERIVATION_ALGORITHM_SLICE` algorithm (see §7.9.16) according to your ElGamal ECC encryption padding scheme used. With this approach, you can use any padding scheme for encryption.

For example, assume you use ElGamal ECC with the NIST-256 curve to wrap a secret 16-byte AES key by adding 4 bytes to its beginning to map it to a point on an ECC curve. Then the unwrapping code should resemble the following:

```
SKB_SecureData* secret_key; // This will contain the unwrapped AES key
SKB_SecureData* temp_data;
SKB_SecureData* ecc_key = ...; // Previously obtained private ECC key

SKB_Byte wrapped_buffer[256/8 * 4] = { ... };
SKB_Size wrapped_buffer_size = sizeof(wrapped_buffer);

// ECC parameters
SKB_EccParameters params = {};
params.curve = SKB_ECC_CURVE_NIST_256;
params.domain_parameters = NULL;
params.random_value = NULL;
```

```
SKB_Engine_CreateDataFromWrapped(engine,
                                 wrapped_buffer,
                                 wrapped_buffer_size,
                                 SKB_DATA_TYPE_BYTES,
                                 SKB_DATA_FORMAT_RAW,
                                 SKB_CIPHER_ALGORITHM_ECC_ELGAMAL,
                                 &params,
                                 ecc_key,
                                 &temp_data);

// Now temp_data contains 256/8 = 32 bytes. The secret AES key is stored in bytes with
// indices 12 to 27. Remember that data is in big-endian, so when you add 4 bytes before
// the 16-byte AES key in the encryption process, all 20 bytes go to bytes with indices
// 12 to 31 (the 4 added bytes are stored in bytes with indices 28 to 31).

// Extract the AES key from bytes 12 to 27
SKB_SliceDerivationParameters params = { 12, 16 }; // from, size
SKB_SecureData_Derive(temp_data,
                      SKB_DERIVATION_ALGORITHM_SLICE,
                      &params,
                      &secret_key);

// Release temporary data
SKB_SecureData_Release(temp_data);

// Now use secret_key that contains the 16-byte AES key
// ...

// Release the secret key when it is no longer needed
SKB_SecureData_Release(secret_key);
```

3.3 Wrapping Plain Data

In some cases, you may want to take a plain input buffer, encrypt it with a key, and store the output as a new secure data object. For example, this operation is suitable for deriving new keys from some input seed and a specific key.

To wrap plain data, call the `SKB_Engine_WrapDataFromPlain` method (see §7.9.7). The input is plain data, but the encryption key and the output of the method are secure data objects.

3.4 Wrapping Keys

A cryptographic key that is stored within a secure data object can be encrypted with another key. This process is called wrapping. The wrapped key can then be passed to any other cryptographic library (not necessarily SKB) where it can be unwrapped and used (see Figure 3.2).

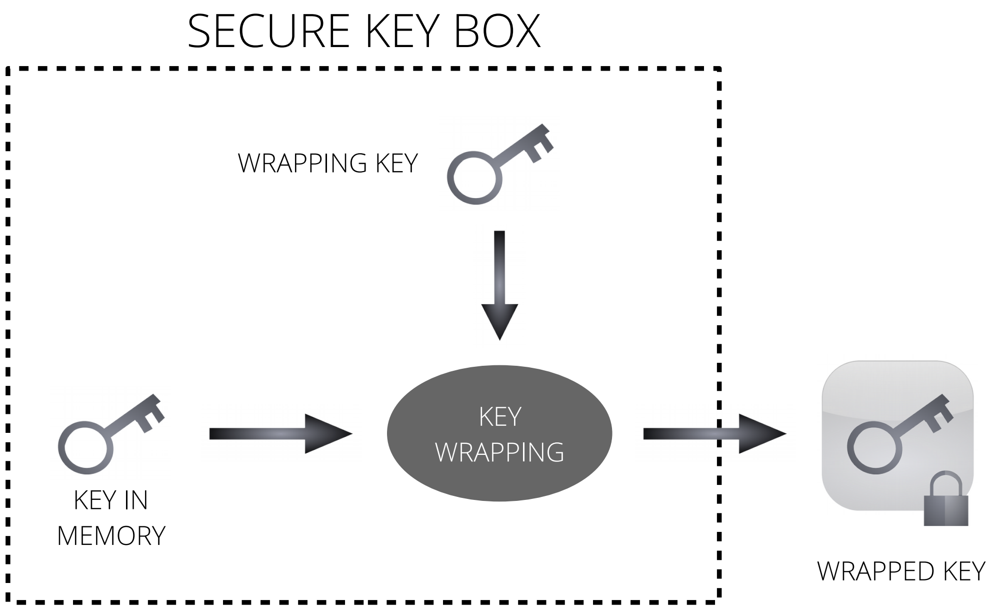

Figure 3.2: Wrapping a key

Wrapping is not the same operation as regular encryption, because regular encryption requires plain data as input (see §3.10.1). The wrapping operation takes a secure data object as an input, and therefore the wrapped key is never exposed in plain form.

Wrapping is also not the same operation as exporting of secure data, because exporting uses an SKB-specific format that cannot be processed by other cryptographic libraries. Wrapping, on the other hand, produces output in a defined format that can be processed by such libraries.

To wrap a key contained within a secure data object, call the `SKB_SecureData_Wrap` method and specify the necessary parameters (see §7.9.15).

3.5 Exporting Keys

SKB operates on keys in memory. If you need a key to be persistent or if you want it to be available to multiple engines, you can request SKB to provide a protected form of the key, which can then be securely exported and stored.

When the exported key is needed again later, you can request SKB to import it as a new secure data object similar to the one whose data was initially exported. For information on importing exported keys, see §3.6.

When SKB is asked to export a key for persistent storage, it encrypts the actual contents of the referenced secure data object (not its binary representation) with the export key (see §1.1.6). The export key of the exporting instance must match the export key of the importing instance.

To export a key, call the `SKB_SecureData_Export` method, specify the secure data object to be exported, and provide a memory buffer where the exported data should be written (see §7.9.14).

3.6 Importing Keys

If you have a protected buffer of data containing a key previously exported from SKB (or obtained using the Key Export Tool), you can import it into SKB. If the data to be imported was encrypted with an export key (see §1.1.6), the export key of the importing SKB instance must match the export key that was used to export the data.

To import a key, call the `SKB_Engine_CreateDataFromExported` method and provide the exported data buffer (see §7.9.6). This method will create a new secure data object containing the imported key.

3.7 Upgrading Exported Keys

SKB supports one-way data upgrade deployments. This means that you have an option to request multiple SKB packages, where each package is assigned a version number, starting with version 1. These packages are configured so that an SKB instance with a greater version number can upgrade and import keys exported by all SKB instances with a smaller version number, but not the other way round.

The practical application of this is that older versions of your protected application will not be able to read data exported by newer versions of that application. For example, if someone successfully cracks your application, the attack will not be directly applicable to newer releases of that application.

The one-way data key upgrade mechanism is implemented by embedding into each SKB package all export keys of its previous versions as shown in Figure 3.3.

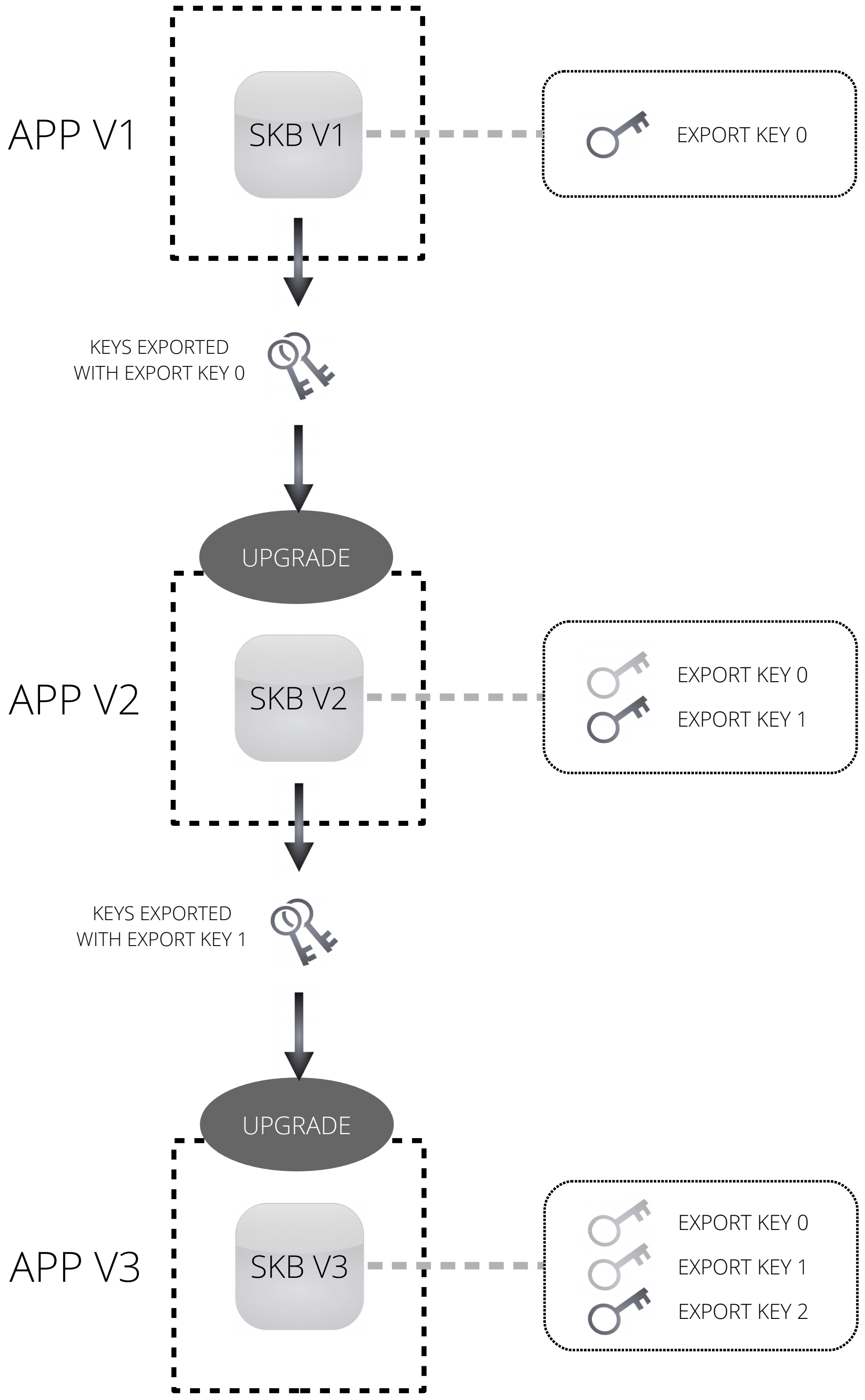

Figure 3.3: One-way data upgrading

To implement the one-way data upgrading process, proceed as follows:

1. Order multiple versioned SKB packages.
2. Integrate SKB package with version 1 into your application.
3. When creating an updated version of your application, execute the following steps:

- In the application code, replace the SKB library with the subsequent version.
- In the process of upgrading your application, execute the `SKB_Engine_UpgradeExportedData` method on each key exported by the previous SKB version as described in §7.9.12.

  This function will upgrade the exported keys so that they are no longer readable by the previous SKB version. With this approach, you can even upgrade keys exported by older SKB releases.

  Alternatively, you can upgrade exported keys with Key Export Tool as described in §5.1.

⚠ Key upgrading should be performed only once. To avoid security risks, do not perform key upgrading and importing every time SKB is run. After the upgrade, make sure all keys of previous versions are permanently deleted.

3.8 Generating Keys

SKB provides a way for generating new symmetric and private keys. The generated keys will contain random content based on the native system's random generator as follows:

- For Windows, CryptoAPI is used.
- For other systems, the `/dev/urandom` device is used, if available; otherwise, the `/dev/random` device is used.

If necessary, you can create a custom implementation for the random generator function as described in §4.2.

To generate a new random key, call the `SKB_Engine_GenerateSecureData` method, specify what type of key you want to generate, and provide the necessary parameters (see §7.9.8).

With the help of this method, you can generate:

- random buffer of bytes (for example, a DES or AES key)
- private ECC keys

Currently, SKB does not support generating private and public RSA keys.

3.9 Deriving a Public Key from a Private Key

As described in §3.8, SKB can generate new random private ECC keys. In connection with this, it may be necessary to get the corresponding public keys as well. A public key can be derived from a private key.

To derive a public key from a private key, call the `SKB_SecureData_GetPublicKey` method, provide the secure data object containing the private key, and supply the necessary parameters (see §7.9.17). This method will return a buffer of bytes containing the corresponding public key.

Currently, this operation is supported only for ECC keys, but not for RSA keys.

3.10 Encrypting and Decrypting Data

This section describes operations related to encrypting and decrypting data.

3.10.1 Encrypting Data

Encryption is a process where a cryptographic cipher and a cryptographic key are applied to plain data to produce encrypted data.

DES, Triple DES, and AES are the only supported encryption ciphers. The main reason for this is that encryption for asymmetric key ciphers (RSA and ElGamal ECC) require a public key, which is usually known and therefore does not require protection.

To encrypt data, proceed as follows:

1. Create a cipher object using the `SKB_Engine_CreateCipher` method, specify the direction `SKB_CIPHER_DIRECTION_ENCRYPT`, and provide all the necessary parameters as described in §7.9.9.
2. Encrypt the input buffer by calling the `SKB_Cipher_ProcessBuffer` method as described in §7.9.19.

   This method returns a byte buffer containing the encrypted data.
3. When no longer needed, release the cipher object by calling the `SKB_Cipher_Release` method as described in §7.9.20.

3.10.2 Decrypting Data

Decryption is a process where a cryptographic cipher and a cryptographic key are applied to encrypted data to produce plain data.

To decrypt data, proceed as follows:

1. Create a cipher object by calling the `SKB_Engine_CreateCipher` method, specify the direction `SKB_CIPHER_DIRECTION_DECRYPT`, and provide all the necessary parameters as described in §7.9.9.
2. Decrypt the input buffer by calling the `SKB_Cipher_ProcessBuffer` method as described in §7.9.19.

   This method returns a byte buffer containing the decrypted data.
3. When no longer needed, release the cipher object by calling the `SKB_Cipher_Release` method as described in §7.9.20.

3.10.3 Using the High-Speed AES

SKB provides an alternative high-speed implementation of AES, which is intended for encrypting and decrypting high-volume data, such as video streams. High-speed AES performance is very close to the performance of unprotected AES.

To use high-speed AES, specify the `SKB_CIPHER_FLAG_HIGH_SPEED` flag when creating the cipher object as described in §7.9.9.

3.11 Calculating a Digest

Calculating a digest involves taking a buffer of plain or secure data and calculating the hash value. The output is a plain buffer of bytes.

To calculate a digest, proceed as follows:

1. Create a transform object by calling the `SKB_Engine_CreateTransform` method, select the `SKB_TRANSFORM_TYPE_DIGEST` type, and specify all the necessary parameters as described in §7.9.10.

2. Supply a buffer of plain or secure data as an input to the transform object by calling the `SKB_Transform_AddBytes` method (see §7.9.21) and the `SKB_Transform_AddSecureData` method (see §7.9.22).

   You can call these methods more than once to pass a large buffer of input data consisting of several smaller data chunks.

3. To calculate the digest, call the `SKB_Transform_GetOutput` function as described in §7.9.23.

4. When no longer needed, release the transform object by calling the `SKB_Transform_Release` method as described in §7.9.24.

3.12 Creating a Signature

Calculating a signature involves executing the signing algorithm on a buffer of plain or secure data using a particular signing key. The output is a plain buffer of bytes containing the signature.

To calculate a signature, proceed as follows:

1. Obtain a secure data object containing the signing key.

2. Create a transform object by calling the `SKB_Engine_CreateTransform` method, select the `SKB_TRANSFORM_TYPE_SIGN` type, and specify all the necessary parameters as described in §7.9.10.

3. Supply a buffer of data as input to the transform object by calling the following methods:

   `SKB_Transform_AddBytes` (see §7.9.21)
   This method appends a buffer of plain data to the input.

   `SKB_Transform_AddSecureData` (see §7.9.22)
   This method appends contents of a secure data object to the input.

   ⚠ For the ECDSA and RSA signing algorithms, the `SKB_Transform_AddSecureData` method is available only if the corresponding digest algorithm (see §7.11.6) is also included in the SKB library. For instance, if you want to calculate a signature of a cryptographic key using the ECDSA signing algorithm with SHA-384 as the hash function, the SHA-384 digest algorithm must also be included in the SKB library. Otherwise, you will be allowed to provide only plain data to the input buffer.

   You can call these methods more than once to pass a large buffer of input data consisting of several smaller data chunks. An exception is those signing algorithms that do not have their own hash functions (`SKB_SIGNATURE_ALGORITHM_RSA` and `SKB_SIGNATURE_ALGORITHM_ECDSA`). These algorithms assume that the input is already a message digest calculated using an arbitrary hash function. Therefore, these algorithms will accept only one data buffer of plain data as an input. This means that only the `SKB_Transform_AddBytes` method can be used (not `SKB_Transform_AddSecureData`), and only once. Since these signing algorithms operate only on plain data, they are significantly faster than other algorithms that employ a hash function.

4. To calculate the signature, call the `SKB_Transform_GetOutput` function as described in §7.9.23.

5. When no longer needed, release the transform object by calling the `SKB_Transform_Release` method as described in §7.9.24.

3.13 Verifying a Signature

Verifying a signature involves executing the verification algorithm on a signature buffer using a particular verification key.

To verify a signature, proceed as follows:

1. Obtain a secure data object containing the verification key.
2. Create a transform object by calling the `SKB_Engine_CreateTransform` method, select the `SKB_TRANSFORM_TYPE_VERIFY` type, and specify all the necessary parameters including the verification key, as described in §7.9.10.
3. Supply a buffer of plain or secure data as an input to the transform object by calling the `SKB_Transform_AddBytes` method (see §7.9.21) and the `SKB_Transform_AddSecureData` method (see §7.9.22).

   You can call these methods more than once to pass a large buffer of input data consisting of several smaller data chunks.
4. To verify the signature against the supplied data buffer, call the `SKB_Transform_GetOutput` function as described in §7.9.23.

   The output is 1 if the signature is correct, and 0 if it is not.
5. When no longer needed, release the transform object by calling the `SKB_Transform_Release` method as described in §7.9.24.

3.14 Executing the Key Agreement Algorithm

The key agreement algorithm involves two parties that want to obtain a shared secret (usually a cryptographic key) that should be known only to them.

First, both parties each generate a public value or key that is given to the other party. Then each party takes the other party's public key and generates a shared secret. The shared secret is identical to both parties. This algorithm is shown in Figure 3.4.

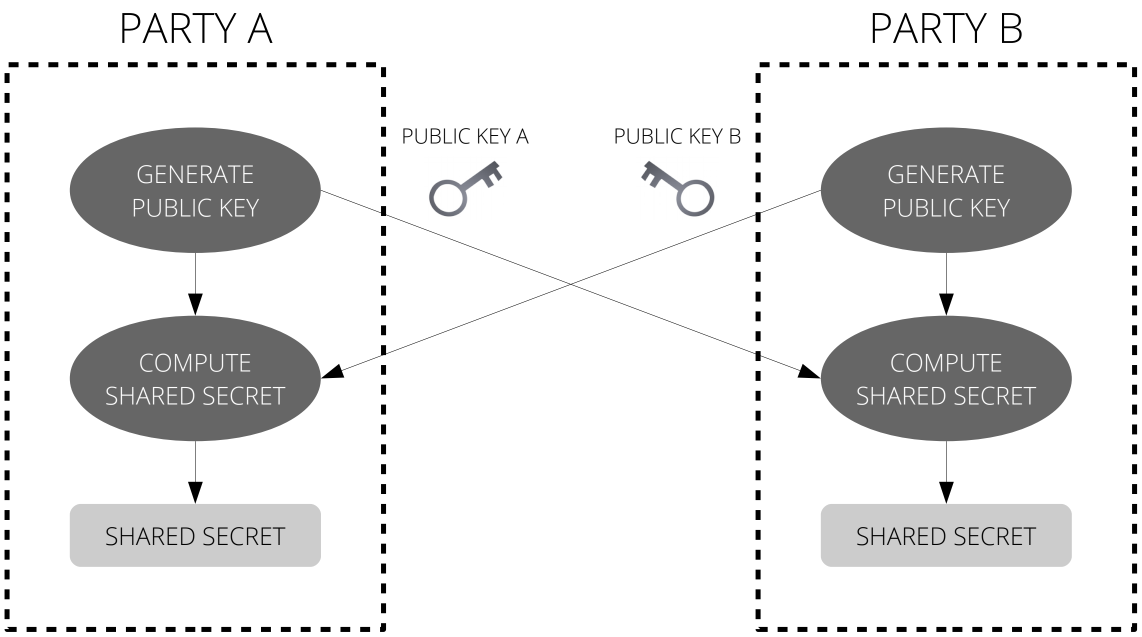

Figure 3.4: Key agreement algorithm

To perform the key agreement algorithm using SKB, proceed as follows:

1. Create a key agreement object by calling the `SKB_Engine_CreateKeyAgreement` method and specify the necessary parameters as described in §7.9.11.
2. Generate a public key by calling the `SKB_KeyAgreement_GetPublicKey` method as described in §7.9.25.
3. Exchange the public keys with the other party.
4. With the other party's public key on hand, compute the shared secret by calling the `SKB_KeyAgreement_ComputeSecret` method as described in §7.9.26.
5. When no longer needed, release the key agreement object by calling the `SKB_KeyAgreement_Release` method as described in §7.9.27.

3.15 Deriving Keys

This section describes several operations that can be used to derive one cryptographic key from another.

3.15.1 Deriving a Key as a Substring of Bytes of Another Key

In some cases, it is necessary to securely derive a new key as a substring of bytes of another key. To do this, call the `SKB_SecureData_Derive` method, select either the `SKB_DERIVATION_ALGORITHM_SLICE` or `SKB_DERIVATION_ALGORITHM_BLOCK_SLICE` algorithm, and specify the range of bytes to be derived as a new key (see §7.9.16).

The only difference between the `SKB_DERIVATION_ALGORITHM_SLICE` and `SKB_DERIVATION_ALGORITHM_BLOCK_SLICE` algorithms is that the latter requires the index of the first byte and the number of bytes in the substring to be multiples of 16.

You can use the `SKB_DERIVATION_ALGORITHM_SLICE` algorithm to extract the unwrapped key from the output of the ElGamal ECC unwrapping algorithm, as described in §3.2.1.

The `SKB_DERIVATION_ALGORITHM_SLICE` and `SKB_DERIVATION_ALGORITHM_BLOCK_SLICE` algorithms can only be performed on a secure data object that contains raw bytes (for example, a DES or AES key), but not an RSA or ECC key.

3.15.2 Deriving a Key as Odd or Even Bytes of Another Key

SKB allows you to derive new keys from an existing key by selecting a number of its odd or even bytes. For example, if you have a 256-byte key, you can derive two 128-byte keys from it (the size of the derived keys can be smaller). One key would have the bytes of the input key with indices 0, 2, 4, 6, and so on (odd bytes). The other key would have the bytes of the input key with indices 1, 3, 5, 7, and so on (even bytes).

To use this algorithm, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_SELECT_BYTES` algorithm, and specify the necessary parameters (see §7.9.16).

This operation can only be performed on a secure data object that contains raw bytes (for example, a DES or AES key), but not an RSA or ECC key.

3.15.3 Deriving a Key by Encrypting or Decrypting an Existing Key

One way of obtaining a new key is by taking an existing key and encrypting or decrypting it with another key. Since keys cannot appear in plain form, the input key, the encrypting or decrypting key, and the output key have to be secure data objects. SKB supplies a special derivation algorithm for this purpose.

To use this algorithm, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_CIPHER` algorithm, and specify the necessary parameters (see §7.9.16).

This operation can only be performed on a secure data object that contains raw bytes (for example, a DES or AES key), but not an RSA or ECC key.

3.15.4 Deriving a Key as a Protected Hash Value of Another Key

SKB provides the following special key derivation algorithms that allow obtaining a new key from a hash value calculated from another key:

- iterated SHA-1 derivation (see §3.15.4.1)
- SHA-256 derivation with plain prefix and suffix (see §3.15.4.2)
- SHA-384 derivation (see §3.15.4.3)

The main difference from the standard SHA operations (provided by the `SKB_Transform` class) is that the output of these special algorithms is a secure data object, whereas the `SKB_Transform` class provides the hash value in plain form. This feature makes these algorithms suitable for deriving new keys.

3.15.4.1 Iterated SHA-1 Derivation

The iterated SHA-1 derivation algorithm creates a new key as a substring of bytes from a SHA-1 hash value obtained from another key.

This algorithm functions as follows:

1. The SHA-1 hash value is calculated from the contents of the provided secure data object (key).

   The result is 20 bytes containing the hash value.

2. Optionally, if requested by the caller (number of rounds is greater than 1), the specified number of bytes is taken from the beginning of the 20-byte hash value and passed to the SHA-1 algorithm again one or several times.

   Each time, the result again is 20 bytes containing the hash value.

3. Finally, the specified number of bytes is taken from the beginning of the 20-byte hash value and returned as a new secure data object.

To use this algorithm, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_SHA_1` algorithm, and specify the necessary parameters (see §7.9.16).

This operation can only be performed on a secure data object that contains raw bytes (for example, a DES or AES key), but not an RSA or ECC key.

3.15.4.2 SHA-256 Derivation with Plain Prefix and Suffix

This derivation algorithm creates a hash value of a buffer that contains three parts in the following sequence:

1. plain data of arbitrary size
2. secure data object (key)
3. plain data of arbitrary size

The output is stored as a new secure data object, which can serve as a new key.

To use this algorithm, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_SHA_256` algorithm, and specify the plain prefix and suffix buffers (see §7.9.16).

This operation can only be performed on a secure data object that contains raw bytes (for example, a DES or AES key), but not an RSA or ECC key.

3.15.4.3 SHA-384 Derivation

The SHA-384 derivation algorithm applies SHA-384 to the input secure data object (key) and stores the output as a new secure data object (key). Unlike the SHA-1 derivation algorithm, this operation is executed only once, and the entire 48-byte hash value is returned as an output.

To use this algorithm, call the `SKB_SecureData_Derive` method and select the `SKB_DERIVATION_ALGORITHM_SHA_384` algorithm (see §7.9.16).

This operation can only be performed on a secure data object that contains raw bytes (for example, a DES or AES key), but not an RSA or ECC key.

3.15.5 Reversing the Order of Bytes of a Key

SKB provides a simple derivation algorithm that allows you to reverse the order of bytes within a secure data object. With this method, you can not only derive new keys but also convert a little-endian data buffer to big-endian and vice versa.

To use this algorithm, call the `SKB_SecureData_Derive` method and select the `SKB_DERIVATION_ALGORITHM_REVERSE_BYTES` algorithm (see §7.9.16).

This operation can only be performed on a secure data object that contains raw bytes (for example, a DES or AES key), but not an RSA or ECC key.

3.15.6 Using the NIST 800-108 Key Derivation Function

SKB provides a derivation algorithm that is based on the *NIST Special Publication 800-108*, which is available here:

http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf

The following special notes apply to the SKB implementation:

- 128-bit AES-CMAC is used as the pseudorandom function.
- The key derivation function works in counter mode.
- The size of the iteration counter and its binary representation (parameters “i” and “r”) is 8 bits.
- The size of the integer specifying the length of the derived key (parameter “L”) is either 16 bits or 32 bits (depending on the algorithm you choose) and is encoded using the big-endian notation.

To use this algorithm, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128` algorithm (for using the 32-bit “L” parameter) or `SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128_L16BIT` algorithm (for using the 16-bit “L” parameter), and specify the necessary parameters (see §7.9.16).

This operation can only be performed on a secure data object that contains raw bytes (for example, a DES or AES key), but not an RSA or ECC key.

3.15.7 Using KDF2 of the RSAES-KEM-KWS Scheme Defined in the OMA DRM Specification

SKB provides a derivation algorithm that is based on KDF2 used in the RSAES-KEM-KWS scheme of the OMA DRM specification.

To use this algorithm, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_OMA_DRM_KDF2` algorithm, and specify the necessary parameters (see §7.9.16).

This operation can only be performed on a secure data object that contains raw bytes (for example, a DES or AES key), but not an RSA or ECC key.

3.15.8 Deriving a Key as Raw Bytes from a Private ECC Key

In some scenarios, you may want to derive a new key as raw bytes (for example, a DES or AES key) from a private ECC key.

To use this algorithm, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_RAW_BYTES_FROM_ECC_PRIVATE` algorithm, and specify the necessary parameters (see §7.9.16).

The derived data buffer will contain the private ECC key in little-endian or big-endian encoding (depending on the selected parameters), and its size will be the same as the size of the private ECC key rounded up to whole bytes. You can then use other derivation algorithms to obtain new keys.

This operation can only be performed on a secure data object that contains a private ECC key.

3.15.9 Deriving a Key By Encrypting Data Using 128-bit AES With a Concatenated Key

This derivation algorithm consists of several steps executed one after another as shown in Figure 3.5.

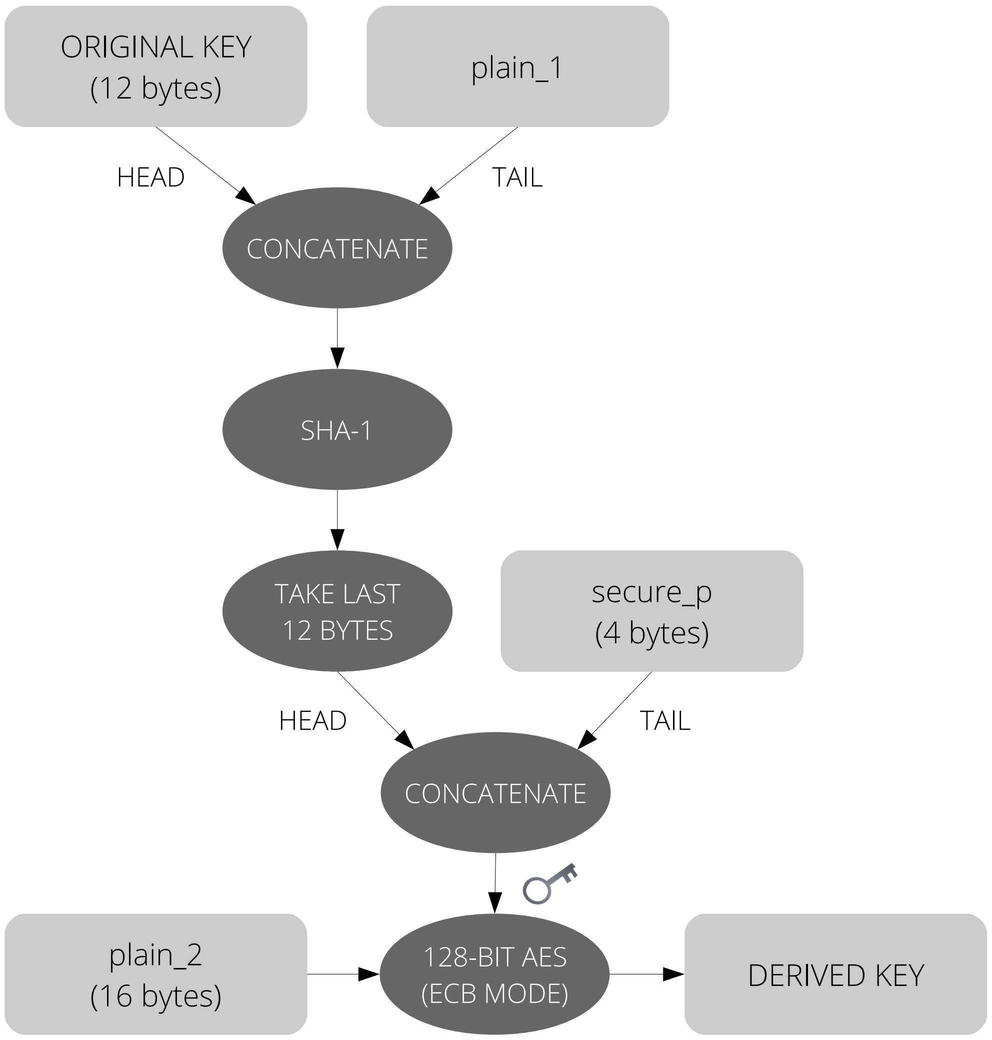

Figure 3.5: Key derivation based on 128-bit AES encryption with a concatenated key and SHA-1

Figure 3.5 contains the following elements:

- “ORIGINAL KEY” is the secure data object used as an input of the derivation algorithm (must be 12 bytes long).
- “plain_1” and “plain_2” are plain data buffers provided as input parameters to the algorithm.

  “plain_2” must be 16 bytes long.
- “secure_p” is a secure data object provided as an input parameter to the algorithm (must be 4 bytes

long).

- "DERIVED KEY" is a new secure data object obtained in the output.

Additionally, this derivation algorithm supports a simplified mode of operation when "plain_1" is not provided (is `NULL`). Then the algorithm is executed as shown in Figure 3.6.

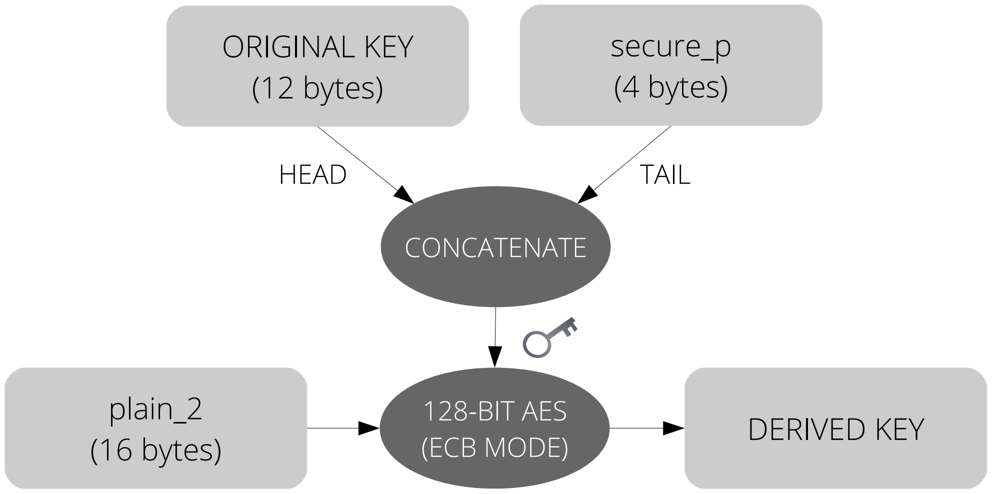

Figure 3.6: Key derivation based on 128-bit AES encryption with a concatenated key without SHA-1

As can be seen, this algorithm is similar to the first one, except the SHA-1 step involving "plain_1" parameter is omitted.

To execute this derivation algorithm, call the `SKB_SecureData_Derive` method (see §7.9.16), select the `SKB_DERIVATION_ALGORITHM_SHA_AES` algorithm, and supply `SKB_ShaAesDerivationParameters` as the parameters structure (see §7.10.17).

This operation can only be performed on a secure data object that contains raw bytes and is 12 bytes long.

3.15.10 Deriving a Key Using the CMLA Key Derivation Function

SKB provides a derivation algorithm that is based on the key derivation function specified in the *CMLA Technical Specification*.

To use this algorithm, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_CMLA_KDF` algorithm, and specify the necessary parameters (see §7.9.16).

This operation can only be performed on a secure data object that contains raw bytes (for example, a DES or AES key), but not an RSA or ECC key.

3.15.11 Deriving a Key By XOR-ing It with Plain Data or Another Key

SKB provides a derivation algorithm that obtains a new key by taking an existing key as input and executing the XOR operation on it using plain data or another key. The input key and the output key are secure data objects.

To use this algorithm, call the `SKB_SecureData_Derive` method, select the

`SKB_DERIVATION_ALGORITHM_XOR` algorithm, and specify the necessary parameters (see §7.9.16).

This operation can only be performed on a secure data object that contains raw bytes (for example, a DES or AES key), but not an RSA or ECC key.

3.16 Binding Keys to a Specific Device

Normally, keys exported by SKB can be imported by any other SKB instance that has the same export key (see §1.1.6), regardless of the device it is run on. In some cases, you may want to bind exported keys to a specific device, so that they cannot be imported on any other device.

SKB provides device binding via the method **`SKB_Engine_SetDeviceId`** (see §7.9.3), which can be called after initializing the engine. By calling this method, you set the device ID, which is a byte array of arbitrary length, typically derived from the hardware details or other environment-specific parameters. This ID is combined with the SKB export key to create a unique format for exported keys. The SKB instance that imports keys must have the same export key and same device ID set as the instance that exports the keys.

When the device ID is no longer needed, you can restore the default export format that depends only on the export key.

Device binding can also be performed via Key Export Tool as described in §5.1.

3.17 Decrypting Encrypted PDF Documents

SKB provides several functions for safely decrypting contents of encrypted PDF files without revealing the user password and the derived encryption key. A typical PDF decryption process involves the following steps:

1. Obtain the user password.
2. Authenticate the user password to verify that the password is valid.
3. Derive the encryption key from the user password, which is then used in the actual decryption process.
4. Decrypt parts of encrypted PDF objects with the derived encryption key.

For detailed instructions on how to perform PDF decryption using SKB, see §6.

4 Supporting Libraries

The core of SKB is delivered as a single binary library. However, for several reasons certain functions are externalized as separate libraries that are delivered together with SKB.

The following supporting libraries are available:

Sensitive Operations Library (see §4.1)

Contains functions for loading plain keys into SKB.

Platform-Specific Library (see §4.2)

Contains functions that may be implemented differently on the same architecture.

4.1 Sensitive Operations Library

This section describes the Sensitive Operations Library, internally called `SkbInternalHelpers`.

4.1.1 Overview

The Sensitive Operations Library is used to perform the following operations:

- load plain keys as secure data objects
- save secure data objects as plain keys

Since loading and saving plain keys are very insecure operations, this library is separated from the main API and there is no dependency from one to another. For example, you may want to use the Sensitive Operations Library on a secure server that operates with plain keys, but you will definitely want to exclude this library from a client application that is exposed to attacks (see §1.1.7).

If loading of plain data is disabled in SKB, the `SKB_Engine_CreateDataFromWrapped` method will not allow loading plain keys (see §7.9.5), but the Sensitive Operations Library will still work.

Note that the Sensitive Operations Library is required to run SKB unit and speed tests.

4.1.2 Library Functions

The Sensitive Operations Library has its own interface, defined in the `SkbInternalHelpers.h` file, which is located in the `Tools/SkbInternalHelpers` folder. This section describes the functions declared in this interface.

4.1.2.1 SKB_CreateRawBytesFromPlain

This function creates an `SKB_SecureData` object from a plain data buffer. The type of the created `SKB_SecureData` object will be `SKB_DATA_TYPE_BYTES` (see §7.11.1).

The function is declared as follows:

```
SKB_Result SKB_CreateRawBytesFromPlain(const SKB_Engine* engine,
                                       const SKB_Byte*   plain,
                                       SKB_Size          plain_size,
                                       SKB_SecureData**  data);
```

The following are the parameters used:

engine

Pointer to the pre-initialized engine.

plain

Pointer to the data buffer containing the plain key.

plain_size

Size of the `plain` buffer in bytes.

data

Address of a pointer to the `SKB_SecureData` object that will contain the created key after this function is executed.

4.1.2.2 SKB_CreatePlainFromRawBytes

This function returns a plain data buffer from an `SKB_SecureData` object. The type of the provided `SKB_SecureData` object must be `SKB_DATA_TYPE_BYTES` (see §7.11.1).

The function is declared as follows:

```
SKB_Result SKB_CreatePlainFromRawBytes(const SKB_SecureData* data,
                                       SKB_Byte*             plain,
                                       SKB_Size*             plain_size);
```

The following are the parameters used:

data

Pointer to the `SKB_SecureData` object from which the plain data buffer must be created.

plain

This parameter is either `NULL` or a pointer to the memory buffer where the plain key is to be written.

If this parameter is `NULL`, the method simply returns, in `plain_size`, the number of bytes that would be sufficient to hold the plain key, and returns `SKB_SUCCESS`.

If this parameter points to a memory buffer (it is not `NULL`), and the buffer size is large enough to hold the plain key, the method stores the plain key there, sets `plain_size` to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets `plain_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

plain_size

Pointer to the size of the `plain` buffer in bytes.

4.1.2.3 SKB_CreateEccPrivateFromPlain

This function creates an `SKB_SecureData` object from a plain private ECC key. The type of the created `SKB_SecureData` object will be `SKB_DATA_TYPE_ECC_PRIVATE_KEY` (see §7.11.1).

The function is declared as follows:

```
SKB_Result SKB_CreateEccPrivateFromPlain(const SKB_Engine* engine,
                                         const SKB_Byte*   plain,
                                         SKB_Size          plain_size,
                                         SKB_SecureData**  data);
```

The following are the parameters used:

engine

Pointer to the pre-initialized engine.

plain

Pointer to the data buffer containing the private ECC key. For information on the ECC key format, see §8.6.

plain_size

Size of the `plain` buffer in bytes.

data

Address of a pointer to the `SKB_SecureData` that will contain the created key after this function is executed.

4.1.2.4 SKB_CreatePlainFromEccPrivate

This function derives a plain private ECC key from an `SKB_SecureData` object. The type of the provided `SKB_SecureData` object must be `SKB_DATA_TYPE_ECC_PRIVATE_KEY` (see §7.11.1).

The function is declared as follows:

```
SKB_Result SKB_CreatePlainFromEccPrivate(const SKB_SecureData* data,
                                         SKB_Byte*             plain,
                                         SKB_Size*             plain_size);
```

The following are the parameters used:

data

Pointer to the `SKB_SecureData` from which the plain private ECC key must be derived.

plain

This parameter is either `NULL` or a pointer to the memory buffer where the plain key is to be written.

If this parameter is `NULL`, the method simply returns, in `plain_size`, the number of bytes that would be sufficient to hold the plain key, and returns `SKB_SUCCESS`.

If this parameter points to a memory buffer (it is not `NULL`), and the buffer size is large enough to hold the plain key, the method stores the plain key there, sets `plain_size` to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets `plain_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

The data will be provided using the big-endian encoding.

`plain_size`

Pointer to the size of the `plain` buffer in bytes.

4.1.2.5 SKB_CreateRsaPrivateFromPlainPKCS8

This function creates an `SKB_SecureData` object from a plain private RSA key stored according to the PKCS#8 standard. The type of the created `SKB_SecureData` object will be `SKB_DATA_TYPE_RSA_PRIVATE_KEY` (see §7.11.1).

The function is declared as follows:

```
SKB_Result SKB_CreateRsaPrivateFromPlainPKCS8(const SKB_Engine* engine,
                                              const SKB_Byte*   plain,
                                              SKB_Size          plain_size,
                                              SKB_SecureData**  data);
```

The following are the parameters used:

`engine`

Pointer to the pre-initialized engine.

`plain`

Pointer to the data buffer containing the private RSA key stored according to the PKCS#8 standard.

`plain_size`

Size of the `plain` buffer in bytes.

`data`

Address of a pointer to the `SKB_SecureData` that will contain the created key after this function is executed.

4.1.2.6 SKB_CreateRsaPrivateFromPlain

This function creates an `SKB_SecureData` object from a plain private RSA key defined as a set of key components. The type of the created `SKB_SecureData` object will be `SKB_DATA_TYPE_RSA_PRIVATE_KEY` (see §7.11.1).

⚠ The input parameters must be provided in big-endian encoding.

The function is declared as follows:

```
SKB_Result SKB_CreateRsaPrivateFromPlain(const SKB_Engine* engine,
                                         void*             plain_p,
                                         void*             plain_q,
                                         void*             plain_d,
                                         void*             plain_n,
                                         SKB_Size          key_size,
                                         SKB_SecureData**  data);
```

The following are the parameters used:

`engine`

Pointer to the pre-initialized engine.

`plain_p`

Pointer to the prime number “p”.

`plain_q`

Pointer to the prime number “q”.

`plain_d`

Pointer to the decryption exponent “d”.

`plain_n`

Pointer to the modulus “n”.

`key_size`

Size of the key in bytes.

`data`

Address of a pointer to the `SKB_SecureData` that will contain the created key after this function is executed.

4.1.2.7 SKB_CreatePlainFromRsaPrivate

This function derives plain private RSA key components from an `SKB_SecureData` object. The type of the provided `SKB_SecureData` object must be `SKB_DATA_TYPE_RSA_PRIVATE_KEY` (see §7.11.1).

⚠ The output data buffers will be provided in big-endian encoding.

The function is declared as follows:

```
SKB_Result SKB_CreatePlainFromRsaPrivate(const SKB_SecureData* data,
                                         SKB_Byte*             p,
                                         SKB_Byte*             q,
                                         SKB_Byte*             d,
                                         SKB_Byte*             n,
                                         SKB_Size*             key_size);
```

The following are the parameters used:

`data`

Pointer to the `SKB_SecureData` from which the plain private RSA key components must be derived.

`plain_p`

This parameter is either `NULL` or a pointer to the memory buffer where the prime number “p” is to be written.

If this parameter is `NULL`, the method simply returns, in `key_size`, the number of bytes that would be sufficient to hold the prime number “p”, and returns `SKB_SUCCESS`.

If this parameter points to a memory buffer (it is not `NULL`), and the buffer size is large enough to hold the prime number “p”, the method stores the value there, sets `key_size` to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets `key_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

`plain_q`

Pointer to the prime number “q”. This parameter works similar to `plain_p` and will have the same size.

`plain_d`

Pointer to the decryption exponent “d”. This parameter works similar to `plain_p` and will have the same size.

`plain_n`

Pointer to the modulus “n”. This parameter works similar to `plain_p` and will have the same size.

`key_size`

Pointer to the size of the prime number “p”, prime number “q”, decryption exponent “d”, and modulus “n”.

4.1.2.8 SKB_CreateRsaPublicFromPlainPKCS1

This function creates an `SKB_SecureData` object from a plain public RSA key stored according to the PKCS#1 standard. The type of the created `SKB_SecureData` object will be `SKB_DATA_TYPE_RSA_PUBLIC_KEY_CONTEXT` (see §7.11.1).

The function is declared as follows:

```
SKB_Result SKB_CreateRsaPublicFromPlainPKCS1(const SKB_Engine* engine,
                                             const SKB_Byte*   plain,
                                             SKB_Size          plain_size,
                                             SKB_SecureData**  data);
```

The following are the parameters used:

`engine`

Pointer to the pre-initialized engine.

`plain`

Pointer to the data buffer containing the public RSA key stored according to the PKCS#1 standard.

`plain_size`

Size of the `plain` buffer in bytes.

`data`

Address of a pointer to the `SKB_SecureData` that will contain the created key after this function is executed.

4.1.2.9 SKB_CreateRsaPublicFromPlain

This function creates an `SKB_SecureData` object from a plain public RSA key defined as a set of key components. The type of the created `SKB_SecureData` object will be `SKB_DATA_TYPE_RSA_PUBLIC_KEY_CONTEXT` (see §7.11.1).

> ⚠ The input parameters must be provided in big-endian encoding.

The function is declared as follows:

```
SKB_Result SKB_CreateRsaPublicFromPlain(const SKB_Engine*     engine,
                                        const SKB_Byte*       plain_e,
                                        const SKB_Byte*       plain_n,
                                        const SKB_Size        key_size,
                                        SKB_SecureData**      data);
```

The following are the parameters used:

`engine`

Pointer to the pre-initialized engine.

`plain_e`

Pointer to the public exponent “e”.

`plain_n`

Pointer to the modulus “n”.

`key_size`

Size of the key in bytes.

`data`

Address of a pointer to the `SKB_SecureData` that will contain the created key after this function is executed.

4.2 Platform-Specific Library

This section describes the purpose and details of the Platform-Specific Library delivered together with SKB.

4.2.1 Overview

The following subsections describe the primary features of the Platform-Specific Library.

4.2.1.1 Externalization of Platform-Specific Functions

Although the largest part of SKB is delivered as a single library, a small subset of functions used by SKB depends on the target operating system and may be implemented differently on the same architecture. Therefore, these functions are externalized as a separate module called the Platform-Specific Library. This library is available as source code in the `Tools/SkbPlatform` folder, and as a precompiled binary in the `Libraries` folder.

The Platform-Specific Library has its own interface defined in the `SkbPlatform.h` file, which is located in the `Tools/SkbPlatform` folder. You can use the provided implementation as is or create your own custom implementation of library functions to suit your specific needs, for example to run SKB on an operating system that is not directly supported. All the necessary implementation information is provided in the comments of the `SkbPlatform.h` file.

For information on compiling the Platform-Specific Library, see §2.4.

4.2.1.2 Reducing the Number of SKB Modules

When you request an SKB package from whiteCryption, you receive a customized SKB archive that, by default, has all the requested algorithms enabled. In some cases (for example, if you build different editions or parts of your application), you may want to remove some of the included SKB algorithms, thus reducing the binary size of your application. The Platform-Specific Library allows you to do this.

The Platform-Specific Library has a dependency on the `SkbModules.h` file, which is located in the `Tools/SkbPlatform` folder. This file contains a list of enabled SKB modules (algorithms). You can reduce the number of SKB modules included in the final executable by commenting out or deleting individual lines in the `SkbModules.h` file. The file contains comments that will help you identify the algorithms and features. If you make any changes in the `SkbModules.h`, we recommend you make similar changes in the `SkbConfiguration.h` file as well (located in the same folder), which also contains a list of SKB algorithms. This is necessary to ensure SKB tests and examples work correctly.

4.2.2 Library Functions

The functions in the Platform-Specific Library can be grouped according to their logical purpose as follows:

Key caching

Key caching speeds up operations with private RSA keys. For details on key caching, see §2.3.2.

The following functions are related to key caching:

- `SKB_KeyCache_Create`
- `SKB_KeyCache_Destroy`
- `SKB_KeyCache_GetInfo`
- `SKB_KeyCache_SetGUID`
- `SKB_KeyCache_GetGUID`

- `SKB_KeyCache_ClearData`
- `SKB_KeyCache_SetData`
- `SKB_KeyCache_GetData`

Random generation

The function `SKB_GetRandomBytes` is used to generate a buffer of random bytes of a specific size.

There are two additional random generator related functions that are intended for the Google Native Client (NaCl) target only:

- `SKB_InitRng`
- `SKB_DestroyRng`

If you are building an application for the Google Native Client target, and this application uses SKB algorithms that depend on random generation (including but not limited to key generation, key exporting, ECDSA, and ECDH), you must call the `SKB_InitRng` function before the first instance of random generation. Otherwise, SKB will return the `SKB_ERROR_INVALID_STATE` (-80008) error code. The `SKB_DestroyRng` function must be called after the last instance of random generation. We recommend calling the `SKB_InitRng` function before the first invocation of the `SKB_Engine_GetInstance` function (see §7.9.1). In a similar manner, we recommend calling the `SKB_DestroyRng` function after the last invocation of the `SKB_Engine_Release` function (see §7.9.2).

Mutex handling

The purpose of mutexes is to avoid the simultaneous use of common resources. SKB uses mutex functions to ensure the correct use of threads.

The following functions are related to mutex handling:

- `SKB_Mutex_Create`
- `SKB_Mutex_LockAutoCreate`
- `SKB_Mutex_Lock`
- `SKB_Mutex_Unlock`
- `SKB_Mutex_Destroy`

Logging

SKB uses the function `SKB_LogMessage` to write log messages to a particular output. Logging is only used in the debug mode.

Debugging

SKB calls the function `SKB_StopInDebugger` when an exception occurs at run time. It is only used in the debug mode.

For details on individual functions of the Platform-Specific Library, see the comments in the `SkbPlatform.h` file.

4.2.3 Enabling Key Caching

Key caching is an optional feature that significantly speeds up operations that deal with private RSA keys (see §2.3.2).

Before compiling the Platform-Specific Library, you may enable one of the following key caching modes:

SQLite mode

This mode uses an SQLite-based implementation of key caching. If the SKB library delivered to you includes any RSA features, this is the default mode for all targets, except Google Native Client (NaCl) and PlayStation 3.

In this mode, the key caching data is stored in an SQLite database named `skb.db` in protected form. The implementation of this key caching mode is defined in the `SkbProtectedKeyCacheSQLite.cpp` file, which is located in the `Tools/SkbPlatform/KeyCacheImpl` folder.

You can either use this implementation in your application as is or treat it as an example implementation for key caching. If you use this implementation without modification, make sure that different applications are not accessing the same `skb.db` file. The path to this file varies for different operating systems. You can adjust the path by modifying the `Skb«`*`target`*`»KeyCacheFilePath.cpp` file, located in the `Tools/SkbPlatform/«`*`target`*`»` folder. For instance, if you are protecting an Android application, the file name is `SkbAndroidKeyCacheFilePath.cpp`, and it is located in the `Tools/SkbPlatform/Android` folder.

To use SQLite-based key caching, you will need an SQLite static library (version 3.7.14 or later) to be included in your project. You can obtain the library in the `Libraries` folder.

In-memory mode

This mode uses an internal in-memory map-like data structure for caching keys. If the SKB library delivered to you includes any RSA features, this is the default mode for the Google Native Client and PlayStation 3 targets.

The implementation of this key caching mode is defined in the `SkbProtectedKeyCacheInMemory.cpp` file, which is located in the `Tools/SkbPlatform/KeyCacheImpl` folder.

By default, only the last 10 keys are cached. If you want to change the number of cached keys, define the `SKB_KEY_CACHE_MAX_IN_MEMORY_ITEMS` preprocessor definition as the required number, for example as follows:

```
#define SKB_KEY_CACHE_MAX_IN_MEMORY_ITEMS 20
```

Custom mode

This mode tells SKB to use your own custom implementation of key caching. For information on creating the custom implementation, see §4.2.3.5.

None

In this mode, key caching is not used at all. This is the default mode if there are no RSA features included in the SKB library you requested.

4.2.3.1 Configuring Key Caching Using Visual Studio

In Visual Studio, the key caching mode to be used is set using a specific preprocessor definition in the `SkbPlatform` project properties. The following preprocessor definitions can be set, each corresponding to a particular key caching mode:

- `SKB_USE_KEY_CACHE_SQLITE`
- `SKB_USE_KEY_CACHE_IN_MEMORY`
- `SKB_USE_KEY_CACHE_CUSTOM`
- `SKB_USE_KEY_CACHE_NONE`

4.2.3.2 Configuring Key Caching Using SCons

For SCons, the key caching mode is set by passing the input parameter `skb_key_cache` to the SCons script. The input parameter can have the following values, each corresponding to a particular key caching mode:

- `sqlite`
- `inmem`
- `custom`
- `none`

For more information on running the SCons build script, see §2.4.3.2.

4.2.3.3 Configuring Key Caching Using Android NDK

For Android NDK, the key caching mode is set by passing the input parameter `SKB_KEY_CACHE` to the `ndk-build` command. The input parameter can have the following values, each corresponding to a particular key caching mode:

- `SKB_USE_KEY_CACHE_SQLITE`
- `SKB_USE_KEY_CACHE_IN_MEMORY`
- `SKB_USE_KEY_CACHE_CUSTOM`
- `SKB_USE_KEY_CACHE_NONE`

4.2.3.4 Configuring Key Caching Using Xcode

In Xcode, the key caching mode to be used is set using a specific preprocessor macro:

- For OS X, the macro is set in the `SecureKeyBox` project properties.
- For iOS, the macro is set in the `SkbPlatform` target properties.

The following preprocessor macros can be set, each corresponding to a particular key caching mode:

- `SKB_USE_KEY_CACHE_SQLITE`

- `SKB_USE_KEY_CACHE_IN_MEMORY`
- `SKB_USE_KEY_CACHE_CUSTOM`
- `SKB_USE_KEY_CACHE_NONE`

4.2.3.5 Creating a Custom Key Caching Implementation

In some cases, you might want to create your own implementation of key caching, for example to avoid including the additional SQLite code in your application. In such cases, the key cache API must be reimplemented according to the API description in the `SkbPlatform.h` file.

5 Utilities

This chapter describes the command-line utilities provided together with the SKB package.

The following utilities are available:

Key Export Tool (see §5.1)

Creates a protected exported form of an `SKB_SecureData` object from plain input data, and upgrades previously exported data.

Binary Update Tool (see §5.2)

Adjusts the final application executable if the tamper-resistant SKB library is used.

Custom ECC Tool (see §5.3)

Generates protected forms of ECC domain parameters, which are used for defining custom curves.

Diffie-Hellman Tool (see §5.4)

Generates protected forms of parameters for the Diffie-Hellman key agreement algorithm.

5.1 Key Export Tool

The Key Export Tool is used for the following purposes:

- creating a protected exported form of an `SKB_SecureData` object from plain input data

  The input can be raw bytes (for example, a DES or AES key), a private RSA key, a public RSA key, or a private ECC key.

- upgrading previously exported data to the current version in the one-way data upgrade scheme (see §3.7)

⚠ The Key Export Tool must always be used in a safe environment (see §1.1.7).

5.1.1 Key Export Tool Overview

The Key Export Tool performs the following actions:

1. Depending on the input format, do one of the following:
   - If the input is in plain form, load it as an `SKB_SecureData` object.
   - If the input is previously exported data containing an old key version, upgrade the data.
2. Save the output to a file or to the standard output in a protected format.

   The output format can be binary data, a hexadecimal string, or C code, in which the exported data is defined as an array of bytes.

Once the output is created, you can import it into SKB using the `SKB_Engine_CreateDataFromExported` method (see §7.9.6).

5.1.2 Running the Key Export Tool

The Key Export Tool is located in the `Libraries` folder along with the precompiled SKB library. To run the utility, execute it at the command line and pass several parameters to it as follows:

```
KeyExportTool
    --input-format «input format»
    --output-format «output format»
    (--input «input file» | --input-hex «hexadecimal string»)
    ([--output «output file»] [--output-stdout])
    [--cross-engine]
    [--device-id «file» | --device-id-string «string literal» |
    --device-id-hex «hexadecimal string»]
```

Please note the following special rules:

- Exactly one of the parameters `--input` and `--input-hex` must be provided.
- At least one or both of the parameters `--output` and `--output-stdout` must be provided.
- Either exactly one, or none of the parameters `--device-id`, `--device-id-string`, and `--device-id-hex` must be provided.

The following are the input parameters:

`--input-format`

Specifies the format of the input file. Possible values are the following:

- "`bytes`": raw bytes in plain (for example, a DES or AES key)
  If you are loading a key for the Triple DES algorithm, make sure the input corresponds to the format described in §8.3.
- "`rsa`": plain private RSA key in the PKCS#8 format
- "`rsa-public`": plain public RSA key in the PKCS#1 format

  > ⚠ Public RSA keys can only be exported using the cross-engine export format (see §7.11.9). This means that you must pass the `--cross-engine` parameter to the Key Export Tool.

- "`ecc`": plain private ECC key in the format that corresponds to the format described in §8.6
- "`upgrade`": previously exported data that needs to be upgraded to the current version (see §3.7)

`--output-format`

Specifies the format of the output. Possible values are the following:

- "`binary`": binary buffer of bytes (cannot be used if the `--output-stdout` parameter is specified)
- "`source`": definition of a C array, which you can then copy directly into your source code
- "`hex`": hexadecimal string with each byte represented as two symbols

`--input`

Name of the file to be used as the input.

`--input-hex`

Specifies the input as a command-line argument, rather than a file.

This parameter must be followed by an even number of hexadecimal characters. For example, to specify a key "ABCDEFGHIJKLMNOP", you would write this parameter as follows:

```
--input-hex 4142434445464748494A4B4C4D4E4F50
```

`--output`

File name of the output file generated by the Key Export Tool.

`--output-stdout`

Tells the Key Export Tool to print the output to the standard output.

Using this parameter will cause an error if the `--output-format` parameter value is set to "`binary`".

`--cross-engine`

Tells the Key Export Tool to generate the output using the `SKB_EXPORT_TARGET_CROSS_ENGINE` export format (see §7.11.9).

If the `--cross-engine` parameter is omitted, the `SKB_EXPORT_TARGET_PERSISTENT` export format is used.

`--device-id`, `--device-id-string`, `--device-id-hex`

Optional parameters that allow you to set the device ID. Device ID is combined with the export key to create a unique format for exported keys as described in §3.16.

Only one of the parameters may be provided. The value to be provided depends on the parameter as follows:

- `device-id`: path to a file that contains the device ID in binary form (for example, "`device_id.bin`")
- `device-id-string`: string literal (for example, "Device ABCD")
- `device-id-hex`: case-insensitive hexadecimal string (for example, "3Eadb54fC0A1Bab9")

Note that these parameters are available only if the SKB package you requested has the device binding feature enabled.

By passing only the `--version` parameter to the Key Export Tool you can find out its version number.

You can see a brief description of all available parameters by running the Key Export Tool with the `--help` parameter.

5.2 Binary Update Tool

If the SKB library that you link with your application has tamper resistance applied (see §1.1.9), you have to run the final built application executable through a binary update process to correctly adjust the embedded integrity protection checksums. Adjustment of the binary code is done using a command-line utility called the Binary Update Tool, which is included in the SKB package.

If the binary update process is not executed on a tamper resistant SKB library, the built application will crash at run time with a segmentation fault.

5.2.1 Binary Update Tool Overview

The binary code must be adjusted using the Binary Update Tool after every build of the final application. As an input, the Binary Update Tool requires the binary executable of the protected application and the `*.nwdb` file that is delivered together with SKB (see Figure 5.1).

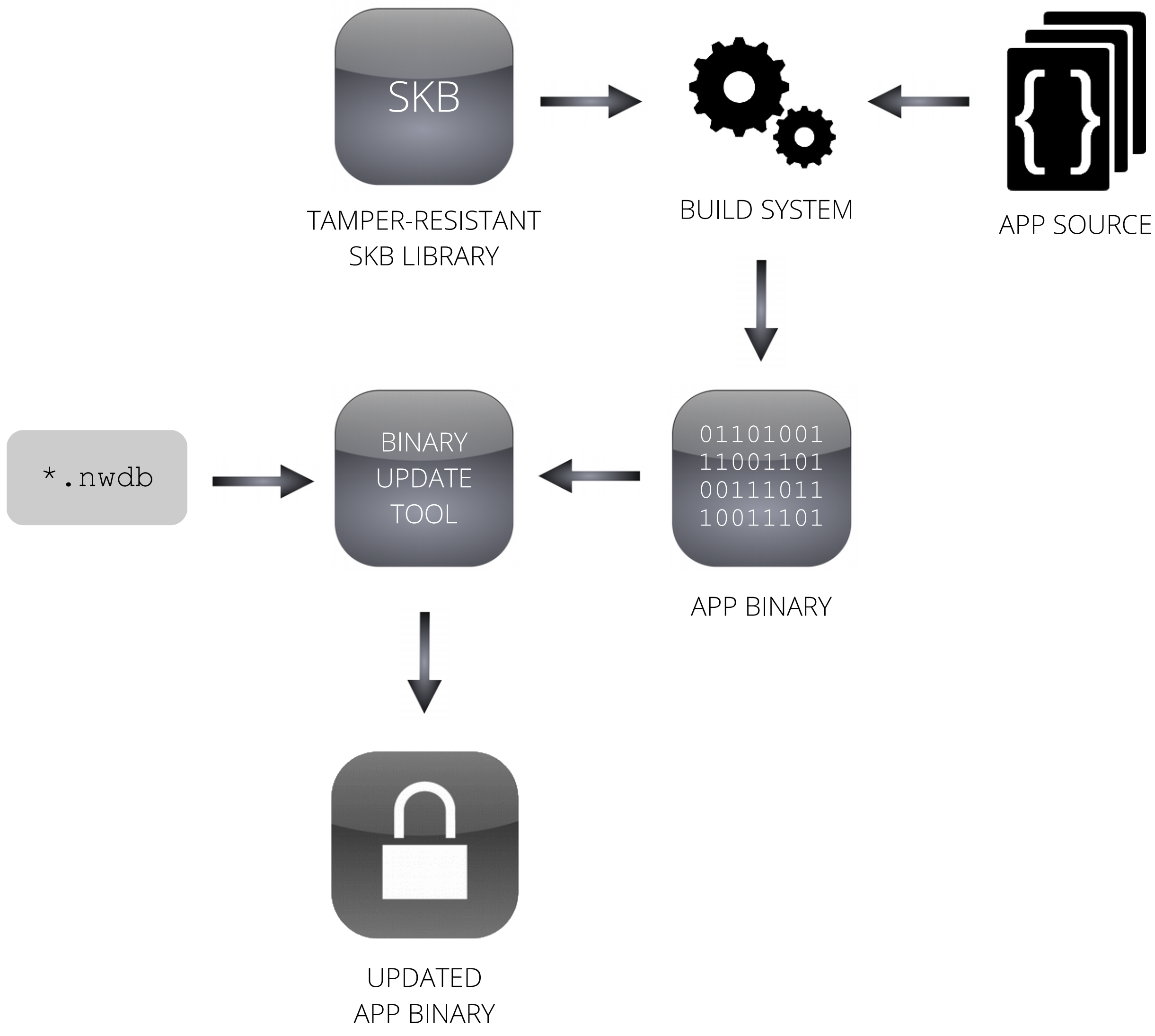

Figure 5.1: Building an application that uses a tamper-resistant SKB library

⚠ OS X and iOS applications must be re-signed after running the Binary Update Tool, because the binary footprint will be modified. You can perform signing using the **`codesign`** tool.

5.2.2 Running the Binary Update Tool

To process an executable with the Binary Update Tool, execute the following command:

```
scp-update-binary --binary=«compiled executable» «*.nwdb file»
```

The **`scp-update-binary`** file and the **`*.nwdb`** file are located in the **`Libraries`** folder. Each target platform, for which tamper resistance is supported, has a separate `*.nwdb` file.

The `--binary` parameter specifies the path to the application executable that contains the SKB library.

As the final parameter, you must provide the `*.nwdb` file of the particular target platform.

Note that you can see a brief description of the available parameters by running the Binary Update Tool with the `--help` parameter.

After the application executable is successfully processed by the Binary Update Tool, you can safely distribute the application to your customers.

5.3 Custom ECC Tool

The Custom ECC Tool generates protected forms of ECC domain parameters, which are used for defining custom curves. The generated protected forms of custom ECC domain parameters must then be specified in the `SKB_EccDomainParameters` structure (see §7.10.19) when you use custom ECC curves.

5.3.1 Custom ECC Tool Overview

The Custom ECC Tool can generate protected forms for the following ECC domain parameters:

- constant “a” in the curve equation
- prime “p” of the elliptic curve
- order “n” of the base point
- X coordinate of the base point
- Y coordinate of the base point
- fixed random value to be passed to the ECDSA algorithm (see §7.10.23)

To generate a protected form of any of these parameters, run the Custom ECC Tool at command prompt and specify the type of the parameter and the input value. The utility will write the protected binary form of the input parameter to the standard output.

The Custom ECC Tool generates only one parameter at a time. To generate multiple parameters, run the utility multiple times, specifying a different parameter each time.

5.3.2 Parameter Size and Value Restrictions

The size of all input parameters must be between 150 and 521 bits, and none of the parameters should have an equal or greater value than the order of the base point.

Note that SKB contains two run-time instances of ECC. One instance corresponds to 150 to 256 bit curves, and the other corresponds to 257 to 521 bit curves. The 150 to 256 bit ECC instance is faster than the 257 to 521 bit ECC instance. If the size of the order of the base point is greater than 256 bits, at run time, SKB will use the 257 to 521 bit ECC instance, which is slower.

5.3.3 Running the Custom ECC Tool

The Custom ECC Tool is located in the `Libraries` folder along with the precompiled SKB library. You can run the utility by executing it at the command line and passing several parameters to it.

The following is the pattern to be used to run Custom ECC Tool:

```
CustomEccTool «parameter type» «parameter value»
```

The following are the input parameters:

«parameter type»

Type of the ECC domain parameter for which the protected form must be generated. The following types are available:

- "`-a`": constant "a" in the curve equation
- "`-p`": prime "p" of the elliptic curve
- "`-n`": order "n" of the base point
- "`-x`": X coordinate of the base point
- "`-y`": Y coordinate of the base point
- "`-r`": fixed random value to be passed to the ECDSA algorithm

«parameter value»

Plain parameter value, which must be specified as an unsigned integer.

If you are passing the parameter "a" and it is a negative number, it must be provided as "p-a" where "p" is the prime of the elliptic curve.

You can see a brief description of all available parameters by running the Custom ECC Tool with the `--help` parameter.

The `SkbEccCustomDomainParameters.h` file, located in the `Examples` folder, contains examples of protected ECC domain parameters for different curve types.

5.4 Diffie-Hellman Tool

The SKB implementation of Diffie-Hellman key agreement algorithm operates on encrypted parameters. The Diffie-Hellman Tool is used to generate the protected forms of these parameters, which must then be provided to the `SKB_PrimeDhParameters` structure (see §7.10.24) when you use the Diffie-Hellman key agreement algorithm.

5.4.1 Diffie-Hellman Tool Overview

The Diffie-Hellman algorithm requires the following basic input parameters:

- prime "p"
- generator "g"
- random value "x"

By design, these parameters are used in plain form, but for increased security, the SKB implementation requires that they are operated on in protected form.

To generate a protected form of any of these parameters, run the Diffie-Hellman Tool at command prompt and specify the type of the parameter and the input value. The utility will write the protected form of the input parameter either to the standard output or to a binary file, depending on your choice.

The Diffie-Hellman Tool generates only one output value at a time. To generate multiple values, run the utility multiple times, specifying a different parameter each time.

5.4.2 Running the Diffie-Hellman Tool

The Diffie-Hellman Tool is located in the `Libraries` folder along with the precompiled SKB library. You can run the utility by executing it at the command line and passing several parameters to it.

The following is the pattern to be used to run the Diffie-Hellman Tool:

```
PrimeDHTool «arguments»
```

The following are the input arguments:

`-s` *«value»*

Maximum bit-length of “p”. This argument is mandatory.

`-p` *«value»*

Prime “p” as an unsigned integer.

If this parameter is specified, the generator “g” (argument `-g`) must also be provided. The output will be a single protected buffer containing both the “p” and “g” parameters.

The greatest common divisor of “p” and “g” must be 1.

`-g` *«value»*

Generator “g” as an unsigned integer.

If this parameter is specified, the prime “p” (argument `-p`) must also be provided. The output will be a single protected buffer containing both the “p” and “g” parameters.

The value of this parameter must be less than “p”, and the greatest common divisor of “p” and “g” must be 1.

`-x` *«value»*

Random value “x” as an unsigned integer.

The output of this parameter can be used to provide a fixed random value to the Diffie-Hellman algorithm as described in §7.10.24.

If this parameter is provided, the parameters `-p` and `-g` must not be supplied.

The output will be a buffer containing the protected random value.

`--output_format` *«value»*

Specifies the format of the output. Possible values are the following:

- “`binary`”: binary file containing a buffer of bytes

- "`source`": definition of a C array, which you can then copy directly into your source code
  Optionally, you can use the command-line argument "`–i` ***«value»***" to specify how many elements should be displayed on each line. The default value is 8.
- "`hex`": string containing the output in hexadecimal format

`--output` ***«value»***

File name of the output file generated by the Diffie-Hellman Tool.

If this argument is not provided, the Diffie-Hellman Tool writes the result to the standard output.

You can see a brief description of all available parameters by running the Diffie-Hellman Tool with the `--help` parameter.

6 Decrypting PDF Files

SKB provides functions specifically dedicated to decrypting encrypted PDF files without revealing the user password and the derived encryption key. This chapter describes how to use these functions.

6.1 PDF Encryption Overview

A PDF document can be encrypted to protect its contents from unauthorized access. Encryption applies only to string and stream objects in the PDF file. Other objects, such as integers and Boolean values, which are used primarily to convey information about the document's structure rather than its content, are left unencrypted. In an encrypted PDF file, every string and stream object is encrypted with a different key. All these keys are derived from one primary encryption key, which in turn is derived from the user password.

Encryption-related information is stored in the document's encryption dictionary, which itself is stored in the `Encrypt` entry of the document's trailer dictionary. The trailer dictionary is a collection of key and value pairs at the very end of the PDF file. The absence of the `Encrypt` entry means that the document is not encrypted.

The encryption dictionary is also a collection of key and value pairs describing all necessary parameters of the particular encryption used.

6.2 PDF Requirements

SKB supports only encrypted PDF files that match the following criteria:

- PDF version is either 1.6 or 1.7.

  PDF versions 1.5 and earlier use RC4 for content decryption. PDF versions above 1.7 use AES-256. These encryption algorithms are currently not supported by SKB for PDF decryption.

- The following key and value pairs are set in the encryption dictionary:
  - "`Filter`"="`Standard`"
  - "`Length`"="`128`"
  - "`R`"="`3`"
  - "`V`"="`2`"

  These restrictions ensure that 128-bit AES in CBC mode is used for content encryption and that the proper key handling algorithms are chosen. Optionally, the values of `R` and `V` keys can both be set to 4, but then the following additional rules must be met:

  - All crypt filters in the crypt filter dictionary must use "AESV2" as the value for the `CFM` parameter, and "16" for the `Length` parameter.
  - The `EncryptMetadata` parameter in the encryption dictionary must be absent or set to "true".

6.3 Decrypting a PDF Document Using SKB

SKB provides the necessary algorithms to perform the following typical PDF decryption process:

1. Obtain the user password in secure format.

   To ensure security, SKB requires that the user password is delivered as an `SKB_SecureData` object containing the password as raw bytes (data type must be `SKB_DATA_TYPE_BYTES`). It is up to you to decide how to load the user password as an `SKB_SecureData` object. For information on `SKB_SecureData` objects and how they can be obtained, see §7.8.2.

2. Authenticate the user password using the `SKB_Pdf_AuthenticateUserPassword` function (see §6.3.1).

   This function verifies that the user password can actually decrypt the document. Authentication is done only once, typically when the PDF document is opened. The user password is always handled as a secure data object in protected form.

3. Derive the encryption key from the user password using the `SKB_Pdf_ComputeEncryptionKey` function (see §6.3.2).

   Since the user password is not directly used for data decryption, an encryption key needs to be derived. This is done only once before any decryption is performed.

4. Prepare a PDF decryption context (represented by the `SKB_Pdf_DecryptionContext` object) using the `SKB_Pdf_CreateDecryptionContext`function (see §6.3.3).

   A PDF decryption context must be created for every PDF object whose data you want to decrypt. The context is passed to the decryption function. The context holds a PDF object decryption key and optionally the initialization vector.

5. Decrypt a buffer from a PDF object using the `SKB_Pdf_DecryptionContext_ProcessBuffer` function (see §6.3.4).

   You can call this function as many times as necessary to decrypt the required parts of a PDF object for which the PDF decryption context was created.

6. When the PDF decryption context is no longer needed, release it from the memory using the `SKB_Pdf_DecryptionContext_Release` function (see §6.3.5).

The functions mentioned above are not considered part of the main SKB API. Therefore, they are defined in a separate header file `SkbExtensions.h`, which is located in the `Source` folder.

6.3.1 SKB_Pdf_AuthenticateUserPassword

This function verifies if the provided user password is valid.

The function is declared as follows:

```
SKB_Result
SKB_Pdf_AuthenticateUserPassword(const SKB_SecureData* password,
                                 const SKB_Byte*       o,
                                 SKB_Size              o_size,
                                 int                   p,
                                 const SKB_Byte*       file_id,
                                 SKB_Size              file_id_size,
                                 const SKB_Byte*       u,
                                 SKB_Size              u_size,
                                 SKB_Byte*             is_user_password_valid);
```

The following are the parameters used:

password
: Pointer to the `SKB_SecureData` object containing the user password.

o
: Pointer to the value of parameter **O** in the encryption dictionary of the PDF file.

o_size
: Size of the `o` value.

p
: Value of parameter **P** in the encryption dictionary of the PDF file.

file_id
: Pointer to the first element of the file identifier array. This array is the value of the **ID** entry in the document's trailer dictionary.

file_id_size
: Size of the `file_id` value.

u
: Pointer to the value of parameter **U** in the encryption dictionary of the PDF file.

u_size
: Size of the `u` value.

is_user_password_valid
: Pointer to an `SKB_Byte` variable that will be set to 1 if the user password is correct, and 0 otherwise.

6.3.2 SKB_Pdf_ComputeEncryptionKey

This function derives the encryption key from the user password.

The function is declared as follows:

```
SKB_Result
SKB_Pdf_ComputeEncryptionKey(const SKB_SecureData* password,
                             const SKB_Byte*       o,
                             SKB_Size              o_size,
                             int                   p,
                             const SKB_Byte*       file_id,
                             SKB_Size              file_id_size,
                             SKB_SecureData**      encryption_key);
```

The following are the parameters used:

password
: Pointer to the `SKB_SecureData` object containing the user password.

o

Pointer to the value of parameter `O` in the encryption dictionary of the PDF file.

o_size

Size of the `o` value.

p

Value of the `P` parameter in the encryption dictionary of the PDF file.

file_id

Pointer to the first element of the file identifier array. This array is the value of the `ID` entry in the document's trailer dictionary.

file_id_size

Size of the `file_id` value.

encryption_key

Address of a pointer to the `SKB_SecureData` that will contain the derived encryption key after this function is executed.

6.3.3 SKB_Pdf_CreateDecryptionContext

This function prepares a PDF decryption context object that is later used in the `SKB_Pdf_DecryptionContext_ProcessBuffer` function (see §6.3.4).

The function is declared as follows:

```
SKB_Result
SKB_Pdf_CreateDecryptionContext(const SKB_SecureData*       encryption_key,
                                int                         object_number,
                                int                         generation_number,
                                const                       SKB_Byte* iv
                                SKB_Pdf_DecryptionContext** ctx);
```

The following are the parameters used:

encryption_key

Pointer to the `SKB_SecureData` object containing the encryption key, which you can obtain using the `SKB_Pdf_ComputeEncryptionKey` function (see §6.3.2).

The encryption key is combined with metadata of the particular PDF object to calculate a decryption key, which is then stored in the PDF decryption context object.

object_number

Object number in the PDF file.

generation_number

Generation number of the object.

iv

Pointer to an initialization vector to be stored in the PDF decryption context.

You can set this parameter to `NULL`, in which case the initialization vector must be passed in the first call of the `SKB_Pdf_DecryptionContext_ProcessBuffer` function (see §6.3.4).

ctx

Address of a pointer to the `SKB_Pdf_DecryptionContext` object that will contain the PDF decryption context after this function is executed.

6.3.4 SKB_Pdf_DecryptionContext_ProcessBuffer

This function decrypts a part of a particular encrypted object in the PDF file.

The function is declared as follows:

```
SKB_Result
SKB_Pdf_DecryptionContext_ProcessBuffer(
   SKB_Pdf_DecryptionContext* ctx,
   const SKB_Byte*            in_buffer,
   SKB_Size                   in_buffer_size,
   const SKB_Byte*            iv,
   SKB_Byte                   is_last_chunk,
   SKB_Byte*                  out_buffer,
   SKB_Size*                  out_buffer_size);
```

The following are the parameters used:

ctx

Pointer to the `SKB_Pdf_DecryptionContext` object, which you prepared before using the `SKB_Pdf_CreateDecryptionContext` function (see §6.3.3).

in_buffer

Pointer to an input buffer containing the part of the encrypted PDF object data to be decrypted.

in_buffer_size

Size of the `in_buffer` value.

iv

Pointer to the initialization vector.

This parameter may be `NULL`, in which case the initialization vector will be taken from the PDF decryption context. The last block of the processed buffer will be stored as the initialization vector in the PDF decryption context after this function is executed.

is_last_chunk

Parameter that should be set to 1 if this is the last part of the encrypted object data, and 0 otherwise.

This information is used to process the CBC mode padding in the encrypted data and calculate the precise decrypted content length.

If this parameter is 1 then the PDF decryption context will no longer contain an initialization vector after the function is executed, and the next call of the `SKB_Pdf_DecryptionContext_ProcessBuffer` function using the same PDF decryption context has to provide an initialization vector.

`out_buffer`

This parameter is either `NULL` or a pointer to the memory buffer where the decrypted content is to be written.

If this parameter is `NULL`, the function simply returns, in `out_buffer_size`, the number of bytes that would be sufficient to hold the output, and returns `SKB_SUCCESS`.

If this parameter points to a memory buffer (it is not `NULL`), and the buffer size is large enough to hold the output, the method stores the output there, sets `out_buffer_size` to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets `out_buffer_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

`out_buffer_size`

Pointer to the variable that holds the size of the memory buffer in bytes where the output is to be stored. For more details, see the description of the `out_buffer` parameter.

You can actually point the `in_buffer` and `out_buffer` to the same memory location, in which case the encrypted input data will be overwritten with the decrypted content.

6.3.5 SKB_Pdf_DecryptionContext_Release

This function releases a PDF decryption context object from the memory.

⚠ You must always call this function when you have completed decrypting PDF object data and no longer need the PDF decryption context.

The function is declared as follows:

```
SKB_Result
SKB_Pdf_DecryptionContext_Release(SKB_Pdf_DecryptionContext* ctx);
```

`ctx` is a pointer to the `SKB_Pdf_DecryptionContext` object to be released.

7 API Reference

This chapter provides full reference information about the SKB API.

7.1 API Overview

The SKB API is a C interface, composed of a number of object classes. Even though the interface is an ANSI C interface, it adopts an object-oriented style. The header file declares a set of classes and class methods. Each method of a class interface is a function whose first argument is a reference to an instance of the same class. The data type that represents references to object instances is a pointer to an opaque C structure. It may be considered as analogous to a pointer to a C++ object.

A concrete example is that for the class named `SKB_Cipher`, the data type `SKB_Cipher` is the name of a C structure. The function name for one of the methods of `SKB_Cipher` is `SKB_Cipher_ProcessBuffer`, and the function takes `SKB_Cipher*` as its first parameter.

7.2 Obtaining Class Instances

An instance of a class is obtained by declaring a pointer to an object for the class and passing the address of that pointer to a particular method. The method creates the instance and sets the pointer to refer to it.

For example, the first object you need to create is `SKB_Engine`, which represents an instance of an engine that can initialize other API objects. `SKB_Engine` is obtained by calling the method `SKB_Engine_GetInstance`, which is declared as follows:

```
SKB_Result SKB_Engine_GetInstance(SKB_Engine** engine);
```

The parameter `engine` is the address of a pointer to an `SKB_Engine` object. This method creates an `SKB_Engine` instance and sets the pointer to refer to the new instance. Here is a sample call:

```
SKB_Engine* engine = NULL;
SKB_Result result;
result = SKB_Engine_GetInstance(&engine);
```

7.3 Making Method Calls

A call to a method of a particular instance is done by calling a function and passing a pointer to the instance as the first parameter.

For example, once an `SKB_Engine` object is created, as shown in the previous section, all the `SKB_Engine` methods can be called to operate on that instance. One such method is `SKB_Engine_GetInfo`, which is used to obtain information about the engine (version number, properties, and so on). This method is declared as follows:

```
SKB_Result SKB_Engine_GetInfo(const SKB_Engine* self, SKB_EngineInfo* info);
```

It stores the engine information in the `SKB_EngineInfo` structure pointed to by the `info` parameter. Assuming `engine` is a pointer previously set by `SKB_Engine_GetInstance` to refer to the `SKB_Engine` instance it created, `SKB_Engine_GetInfo` can be invoked as follows:

```
SKB_Result result;
SKB_EngineInfo engineInfo;
result = SKB_Engine_GetInfo(engine, &engineInfo);
```

7.4 Method Return Values

All methods return an integer value of type `SKB_Result`. When a method call succeeds, the return value is `SKB_SUCCESS`. Otherwise, it is a negative number, as defined by the following constants in the header file:

`SKB_SUCCESS (0)`

The called method was successfully executed.

`SKB_ERROR_INTERNAL (-80001)`

An internal SKB error occurred. Please consult with whiteCryption for assistance.

`SKB_ERROR_INVALID_PARAMETERS (-80002)`

Invalid parameters were supplied to the method.

`SKB_ERROR_NOT_SUPPORTED (-80003)`

The configuration provided to the method is not supported by SKB. It may also mean that you tried to execute an algorithm that is not included in the SKB package you requested.

`SKB_ERROR_OUT_OF_RESOURCES (-80004)`

The method failed to allocate the required amount of memory on the heap.

`SKB_ERROR_BUFFER_TOO_SMALL (-80005)`

The provided memory buffer was not large enough to contain the output.

`SKB_ERROR_INVALID_FORMAT (-80006)`

The format of the input buffer was invalid.

`SKB_ERROR_ILLEGAL_OPERATION (-80007)`

This error code is not used by SKB.

`SKB_ERROR_INVALID_STATE (-80008)`

You attempted to perform an invalid operation on the `SKB_Transform` object, such as the following:

- You tried to add an input buffer to an `SKB_Transform` object after its `SKB_Transform_GetOutput` method was called (see §7.9.23).
- You tried to call the `SKB_Transform_GetOutput` method again after it was already executed.
- You executed the `SKB_Transform_AddBytes` method more than once on an `SKB_Transform` object that is associated with a signing algorithm that does not have its own hash function (`SKB_SIGNATURE_ALGORITHM_RSA` or `SKB_SIGNATURE_ALGORITHM_ECDSA`). For these algorithms, the `SKB_Transform_AddBytes` method may be called only once (see §3.12).

`SKB_ERROR_OUT_OF_RANGE (-80009)`

The specified offset or index of the input buffer was out of range.

`SKB_ERROR_EVALUATION_EXPIRED (-80101)`

The evaluation period of the current SKB package has expired.

`SKB_ERROR_KEY_CACHE_FAILED (-80102)`

A key cache operation failed. Typically this occurs when the key cache database is not available or is write-protected.

`SKB_ERROR_INVALID_EXPORT_KEY_VERSION (-80103)`

Either you were trying to upgrade a key whose version number is equal to or greater than that of the current SKB instance (see §3.7), or you were trying to import a key whose version is not equal to that of the current SKB instance.

`SKB_ERROR_INVALID_EXPORT_KEY (-80104)`

The export key of the current SKB instance did not match the export key that was used for exporting the data that you were trying to import or upgrade.

7.5 Object Lifecycle

To avoid exceptions and correctly release memory, you have to follow certain rules regarding the lifecycle of SKB objects:

- All SKB objects must be released when they are no longer needed by calling the corresponding release methods.
- `SKB_Engine` (see §7.8.1) is the first SKB object to be created and the last one to be released. All other objects created via the `SKB_Engine` object must be released before it.
- `SKB_SecureData` (see §7.8.2) must not be released while it is being used as a key in one the following objects:
  - `SKB_KeyAgreement` object
  - `SKB_Transform` object if this object is associated with one of the ECDSA algorithms

7.6 Restrictions of Multi-Threading

As a general rule, SKB methods and objects are not synchronized and therefore they should not be shared between multiple threads. However, there are two exceptions to this rule:

- The `SKB_Engine` object is thread-safe and can be shared between multiple threads using the `SKB_Engine_GetInstance` method (see §7.9.1). This method will always return the same `SKB_Engine` instance.
- Since the `SKB_SecureData` object is immutable, it can also be shared between multiple threads.

7.7 Overriding Memory Allocation Operators

You may want to override the `new` and `delete` operators to implement custom memory allocation for your application. To successfully achieve this, you must take into account that SKB uses the non-throwing `new` operator for memory allocation.

Let's assume you have the following code for overriding the `new` and `delete` operators:

```
void* operator new (size_t size) {
  // your implementation
}

void* operator new[] (size_t size) {
  // your implementation
}

void operator delete (void* ptr) {
 // your implementation
}

void operator delete[] (void* ptr) {
 // your implementation
}
```

SKB requires that you also provide the following implementations for the non-throwing operators:

```
void* operator new (size_t size, const std::nothrow_t&) {
   return operator new (size);
}

void* operator new[] (size_t size, const std::nothrow_t&) {
   return operator new[] (size);
}

void operator delete (void* ptr, const std::nothrow_t&) {
   return operator delete (ptr);
}

void operator delete[] (void* ptr, const std::nothrow_t&) {
   return operator delete[] (ptr);
}
```

7.8 Classes

This section describes the classes of the API. Most operations are performed via these classes and their related methods.

7.8.1 SKB_Engine

`SKB_Engine` is the first object that you create before using the API. It is used to initialize other API objects.

7.8.2 SKB_SecureData

`SKB_SecureData` contains any data whose value is white-box protected and hidden from the outside world but can be internally operated on by SKB. Usually, the `SKB_SecureData` object is the container for cryptographic keys protected by SKB. Secure data objects can be operated by SKB cryptographic functions but their contents cannot be accessed.

There are several ways how `SKB_SecureData` objects are obtained:

- loading plain keys
- unwrapping encrypted keys
- importing previously exported keys
- obtaining as a shared secret via a key agreement algorithm
- generating a new random `SKB_SecureData` object to be used as a cryptographic key
- deriving an `SKB_SecureData` object from another `SKB_SecureData` object
- wrapping plain keys

7.8.3 SKB_Cipher

`SKB_Cipher` is an object that can encrypt or decrypt data. It encapsulates the attributes and parameters necessary to perform cryptographic operations on data buffers. For more information on encryption and decryption, see §3.10.

7.8.4 SKB_Transform

`SKB_Transform` is an object that can calculate a digest, sign data, or verify a signature. This object can operate both on plain data and secure data. The output is always plain data.

7.8.5 SKB_KeyAgreement

`SKB_KeyAgreement` is an object used to perform the key agreement algorithm. For more information on this algorithm, see §3.14.

7.9 Methods

This section describes all the methods provided by the SKB API.

7.9.1 SKB_Engine_GetInstance

This method creates an `SKB_Engine` instance (see §7.8.1). This instance is the first object that you must obtain before using the API.

⚠ Make sure that every `SKB_Engine_GetInstance` call has a corresponding `SKB_Engine_Release` call to correctly release the memory.

The method is declared as follows:

```
SKB_Result
SKB_Engine_GetInstance(SKB_Engine** engine);
```

The parameter `engine` is an address of a pointer to the `SKB_Engine` object. After execution, this method creates an `SKB_Engine` instance and sets the pointer to refer to the new instance. Every subsequent call of the `SKB_Engine_GetInstance` method will return the same `SKB_Engine` object until this object is released.

7.9.2 SKB_Engine_Release

This method releases an `SKB_Engine` instance from the memory when it is no longer needed.

⚠ Make sure that every `SKB_Engine_GetInstance` call has a corresponding `SKB_Engine_Release` call to correctly release the memory. Also, all other SKB objects created via the `SKB_Engine` object must be released before you call the `SKB_Engine_Release` method.

The method is declared as follows:

```
SKB_Result
SKB_Engine_Release(SKB_Engine* self)
```

The parameter `self` is a pointer to the engine instance that should be released.

7.9.3 SKB_Engine_SetDeviceId

This method sets the device ID, which is a byte array of arbitrary length that will be combined with the export key to form a unique format for exported keys. This method enables you to bind exported keys to a specific device (see §3.16). By default, when an engine is initialized, there is no device ID set and the export format depends only on the export key.

The method is declared as follows:

```
SKB_Result
SKB_Engine_SetDeviceId(SKB_Engine*     self,
                       const SKB_Byte* id,
                       SKB_Size        size);
```

The following are the parameters used:

`self`

Pointer to the pre-initialized engine.

`id`

Pointer to the byte array containing the device ID.

You have to generate this byte array yourself based on some hardware details or other environment-specific parameters.

size

Number of bytes in the **id** parameter. The device ID can be of arbitrary length.

If the size is 0, SKB will remove the previously set device ID. This can be useful when the device ID is no longer needed and the default export format (based only on the export key) needs to be restored.

7.9.4 SKB_Engine_GetInfo

This method populates an `SKB_EngineInfo` structure (see §7.10.1) with the generic information about an initialized engine.

⚠ The contents of the populated `SKB_EngineInfo` structure will not be valid after the corresponding `SKB_Engine` object is released from memory. During examination of the `SKB_EngineInfo` object, the `SKB_Engine` object must exist.

The method is declared as follows:

```
SKB_Result
SKB_Engine_GetInfo(const SKB_Engine* self,
                   SKB_EngineInfo*   info);
```

The following are the parameters used:

self

Pointer to the pre-initialized engine that you want to get the information about.

info

Pointer to the `SKB_EngineInfo` structure to be populated with the engine information (see §7.10.1).

7.9.5 SKB_Engine_CreateDataFromWrapped

This method creates a new `SKB_SecureData` object from a wrapped buffer of data (usually a cryptographic key) by unwrapping it with a previously loaded key. The unwrapped data is never exposed in plain form. For more information on using this method, see §3.2.

As a special case of calling this method, you can also load a plain buffer of data as an `SKB_SecureData` object (see §3.1). In this case, the unwrapping algorithm and the decryption key are not specified. This operation should be used with extreme care because you are providing the key in plain form. Use this approach only in a highly protected environment. The loading of plain keys can be executed only if loading of plain data is enabled in SKB.

The `SKB_Engine_CreateDataFromWrapped` method is declared as follows:

```
SKB_Result
SKB_Engine_CreateDataFromWrapped(SKB_Engine*          self,
                                 const SKB_Byte*      wrapped,
                                 SKB_Size             wrapped_size,
                                 SKB_DataType         wrapped_type,
                                 SKB_DataFormat       wrapped_format,
                                 SKB_CipherAlgorithm   wrapping_algorithm,
                                 const void*           wrapping_parameters,
                                 const SKB_SecureData* unwrapping_key,
                                 SKB_SecureData**      data);
```

The following are the parameters used:

self

Pointer to the pre-initialized engine.

wrapped

Pointer to the buffer of encrypted data (cryptographic key) to be unwrapped.

If you are unwrapping a key for the Triple DES algorithm, make sure the input corresponds to the format described in §8.3.

If you are unwrapping an AES-wrapped private ECC key, for information on how the input buffer must be formatted, see §8.7.

For other cases of AES-wrapped data, see §8.2.

wrapped_size

Size of the `wrapped` buffer in bytes.

wrapped_type

Type of the wrapped key. The available types are defined in the `SKB_DataType` enumeration (see §7.11.1).

wrapped_format

Format how the wrapped key is stored in the input data buffer. The available formats are defined in the `SKB_DataFormat` enumeration (see §7.11.2).

wrapping_algorithm

Cryptographic algorithm to be used for decrypting the data. The available algorithms are defined in the `SKB_CipherAlgorithm` enumeration (see §7.11.3). For information on algorithms that support key unwrapping, see §1.2.

The following algorithms only support unwrapping of raw bytes, meaning that the `wrapped_type` parameter should always be `SKB_DATA_TYPE_BYTES`, and `wrapped_format` should always be `SKB_DATA_FORMAT_RAW`:

- `SKB_CIPHER_ALGORITHM_ECC_ELGAMAL`
- `SKB_CIPHER_ALGORITHM_RSA`
- `SKB_CIPHER_ALGORITHM_RSA_1_5`
- `SKB_CIPHER_ALGORITHM_RSA_OAEP`
- `SKB_CIPHER_ALGORITHM_NIST_AES`
- `SKB_CIPHER_ALGORITHM_AES_CMLA`
- `SKB_CIPHER_ALGORITHM_RSA_CMLA`
- `SKB_CIPHER_ALGORITHM_XOR`

If the `SKB_CIPHER_ALGORITHM_NIST_AES` algorithm is used, in the case of integrity check failure this method will return the `SKB_ERROR_INVALID_FORMAT` error.

If the `SKB_CIPHER_ALGORITHM_NULL` algorithm is used, the method assumes that the key in the input buffer is in plain form. Then you do not have to provide the unwrapping key or unwrapping parameters.

If the `SKB_CIPHER_ALGORITHM_ECC_ELGAMAL` algorithm is used, see the special instructions described in §3.2.1.

`wrapping_parameters`

Additional parameters for the unwrapping algorithm.

If you are using one of the following algorithms, you can optionally point this parameter to the `SKB_AesUnwrapParameters` structure (see §7.10.21) to specify the CBC padding type:

- `SKB_CIPHER_ALGORITHM_AES_128_CBC`
- `SKB_CIPHER_ALGORITHM_AES_192_CBC`
- `SKB_CIPHER_ALGORITHM_AES_256_CBC`

If you use any of the algorithms above and set the `wrapping_parameters` value to `NULL`, CBC mode with the XML encryption padding will be used by default, which is the equivalent of the `SKB_CBC_PADDING_TYPE_XMLENC` value of the `SKB_CbcPadding` enumeration (see §7.11.13).

If you are using the `SKB_CIPHER_ALGORITHM_ECC_ELGAMAL` algorithm, this parameter must be a pointer to the `SKB_EccParameters` structure (see §7.10.23). For special instructions for using the ElGamal ECC unwrapping algorithm, see §3.2.1.

For all other cases, set this parameter to `NULL`.

`unwrapping_key`

`SKB_SecureData` object containing the key needed to decrypt the data.

If the `SKB_CIPHER_ALGORITHM_NULL` algorithm is used, this parameter should be set to `NULL`.

`data`

Address of a pointer to the `SKB_SecureData` that will contain the unwrapped key after this method is executed.

7.9.6 SKB_Engine_CreateDataFromExported

This method imports data that was previously exported using the `SKB_SecureData_Export` method (see §7.9.14).

The method is declared as follows:

```
SKB_Result
SKB_Engine_CreateDataFromExported(SKB_Engine*      self,
                                  const SKB_Byte*  exported,
                                  SKB_Size         exported_size,
                                  SKB_SecureData** data);
```

The following are the parameters used:

self

Pointer to the pre-initialized engine.

exported

Pointer to the memory buffer containing the exported data.

exported_size

Size of the `exported` buffer.

data

Address of a pointer to the `SKB_SecureData` object that will be created by this method. This object will contain the imported data.

7.9.7 SKB_Engine_WrapDataFromPlain

This method takes a plain data buffer, encrypts it with a key stored in an `SKB_SecureData` object, and stores the output as a new `SKB_SecureData` object. For more information on this method, see §3.3.

The method is declared as follows:

```
SKB_Result
SKB_Engine_WrapDataFromPlain(SKB_Engine*           self,
                             const SKB_Byte*       plain,
                             SKB_Size*             plain_size,
                             SKB_DataType          data_type,
                             SKB_DataFormat        plain_format,
                             SKB_CipherAlgorithm   algorithm,
                             const void*           encryption_parameters,
                             const SKB_SecureData* encryption_key,
                             const SKB_Byte*       iv,
                             SKB_Size              iv_size,
                             SKB_SecureData**      data);
```

The following are the parameters used:

self

Pointer to the pre-initialized engine.

plain

Pointer to the memory buffer where the plain input data is stored.

plain_size

Pointer to a variable that holds the size of the input data in bytes.

data_type

Type of data stored in the input buffer. The available types are defined in the `SKB_DataType` enumeration (see §7.11.1).

Currently, this method supports only the `SKB_DATA_TYPE_BYTES` data type.

plain_format

Format how the plain data is stored in the input buffer. The available formats are defined in the `SKB_DataFormat` enumeration (see §7.11.2).

Currently, this method supports only the `SKB_DATA_FORMAT_RAW` data type.

algorithm

Algorithm to be used for encrypting the input data. Available algorithms are defined in the `SKB_CipherAlgorithm` enumeration (see §7.11.3).

Currently, this method supports only the following algorithms:

- `SKB_CIPHER_ALGORITHM_AES_128_ECB`
- `SKB_CIPHER_ALGORITHM_AES_128_CBC`
- `SKB_CIPHER_ALGORITHM_AES_192_ECB`
- `SKB_CIPHER_ALGORITHM_AES_192_CBC`
- `SKB_CIPHER_ALGORITHM_AES_256_ECB`
- `SKB_CIPHER_ALGORITHM_AES_256_CBC`

encryption_parameters

Pointer to a structure that provides additional parameters for the cipher.

Currently, this parameter must always be `NULL`.

encryption_key

Pointer to the `SKB_SecureData` object containing the encryption key.

iv

Pointer to the initialization vector if the encryption algorithm used is AES in CBC mode.

If the initialization vector is not used or if it is all zeros, the value of this parameter should be `NULL`.

iv_size

Size of the initialization vector in bytes.

If the value of the `iv` parameter is `NULL`, this parameter should be 0.

data

Address of a pointer to the `SKB_SecureData` object that will contain the output when this method is executed.

7.9.8 SKB_Engine_GenerateSecureData

This method creates a new random `SKB_SecureData` object based on the provided parameters. This operation is typically used for generating new random keys.

The method is declared as follows:

```
SKB_Result
SKB_Engine_GenerateSecureData(SKB_Engine*      self,
                              SKB_DataType     data_type,
                              const void*      generate_parameters,
                               SKB_SecureData** data);
```

The following are the parameters used:

self

Pointer to the pre-initialized engine.

data_type

Type of data to be generated. The available types are defined in the `SKB_DataType` enumeration (see §7.11.1).

Currently, the `SKB_DATA_TYPE_RSA_PRIVATE_KEY` and `SKB_DATA_TYPE_RSA_PUBLIC_KEY_CONTEXT` types are not supported for generating secure data, meaning that SKB cannot generate private and public RSA keys.

generate_parameters

Pointer to a structure that specifies the necessary parameters for generating the secure data object.

For different secure data types, different structures must be provided as follows:

- For `SKB_DATA_TYPE_BYTES`, this parameter must point to the `SKB_RawBytesParameters` structure (see §7.10.26), which specifies the number of bytes to be generated.
- For `SKB_DATA_TYPE_ECC_PRIVATE_KEY`, this parameter must point to the `SKB_EccParameters` structure (see §7.10.23), which specifies the ECC curve type to be used.

data

Address of a pointer to the `SKB_SecureData` object that will be created by this method. This object will contain the generated data.

7.9.9 SKB_Engine_CreateCipher

This method creates a new `SKB_Cipher` object based on the provided parameters. The `SKB_Cipher` object is used to encrypt or decrypt data.

The method is declared as follows:

```
SKB_Result
SKB_Engine_CreateCipher(SKB_Engine*           self,
                        SKB_CipherAlgorithm   cipher_algorithm,
                        SKB_CipherDirection   cipher_direction,
                        unsigned int          cipher_flags,
                        const void*           cipher_parameters,
                        const SKB_SecureData* cipher_key,
                        SKB_Cipher**          cipher);
```

The following are the parameters used:

self

Pointer to the pre-initialized engine.

`cipher_algorithm`

Algorithm to be used for encrypting or decrypting data. Available algorithms are defined in the `SKB_CipherAlgorithm` enumeration (see §7.11.3).

`cipher_direction`

Parameter that specifies whether the provided data should be encrypted or decrypted. Available directions are defined in the `SKB_CipherDirection` enumeration (see §7.11.4).

Encryption is supported only for the DES, Triple DES, and AES ciphers.

`cipher_flags`

Optional flags for the cipher.

Currently, the only defined flag is `SKB_CIPHER_FLAG_HIGH_SPEED`. This flag can be used only for the AES cipher when it is intended to be used with high throughput, for example media content decryption.

`cipher_parameters`

Pointer to a structure that provides additional parameters for the cipher.

For the `SKB_CIPHER_ALGORITHM_AES_128_CTR`, `SKB_CIPHER_ALGORITHM_AES_192_CTR`, and `SKB_CIPHER_ALGORITHM_AES_256_CTR` ciphers, it must point to the `SKB_CtrModeCipherParameters` structure (see §7.10.4), or `NULL` if the default counter size of 16 is to be used.

For the `SKB_CIPHER_ALGORITHM_ECC_ELGAMAL` cipher, it must point to the `SKB_EccParameters` structure, which specifies the curve type (see §7.10.23).

For all other ciphers, this parameter must be `NULL`.

`cipher_key`

Pointer to the `SKB_SecureData` object containing the encryption or decryption key.

`cipher`

Address of a pointer to the `SKB_Cipher` object which will be created by this method.

7.9.10 SKB_Engine_CreateTransform

This method creates a new `SKB_Transform` object based on the provided parameters. The `SKB_Transform` object is used to calculate a digest, sign data, or verify a signature.

The method is declared as follows:

```
SKB_Result
SKB_Engine_CreateTransform(SKB_Engine*       self,
                           SKB_TransformType transform_type,
                           const void*       transform_parameters,
                           SKB_Transform**   transform);
```

The following are the parameters used:

`self`

Pointer to the pre-initialized engine.

`transform_type`

Transform type to be created. Available transform types are defined in the `SKB_TransformType` enumeration (see §7.11.8).

`transform_parameters`

Pointer to a structure that provides the necessary parameters for the transform. For different transform types, a different structure must be provided.

For the `SKB_TRANSFORM_TYPE_DIGEST` transform, this parameter must point to the `SKB_DigestTransformParameters` structure (see §7.10.5).

For the `SKB_TRANSFORM_TYPE_SIGN` transform, this parameter must point to one of the following structures:

- If one of the following algorithms is to be used, this parameter must point to the `SKB_SignTransformParametersEx` structure (see §7.10.7):
  - `SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5_EX`
  - `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1_EX`
  - `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224_EX`
  - `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256_EX`
  - `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA384_EX`
  - `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA512_EX`
  - `SKB_SIGNATURE_ALGORITHM_ECDSA`
  - `SKB_SIGNATURE_ALGORITHM_ECDSA_MD5`
  - `SKB_SIGNATURE_ALGORITHM_ECDSA_SHA1`
  - `SKB_SIGNATURE_ALGORITHM_ECDSA_SHA224`
  - `SKB_SIGNATURE_ALGORITHM_ECDSA_SHA256`
  - `SKB_SIGNATURE_ALGORITHM_ECDSA_SHA384`
  - `SKB_SIGNATURE_ALGORITHM_ECDSA_SHA512`
- For all other algorithms, this parameter must point to the `SKB_SignTransformParameters` structure (see §7.10.6).

For the `SKB_TRANSFORM_TYPE_VERIFY` transform, this parameter must point to the `SKB_VerifyTransformParameters` structure (see §7.10.8).

`transform`

Address of a pointer to the `SKB_Transform` object that will be created by this method.

7.9.11 SKB_Engine_CreateKeyAgreement

This method creates a new `SKB_KeyAgreement` object based on the provided parameters. The `SKB_KeyAgreement` object is used to calculate a shared secret based on the key agreement algorithm.

The method is declared as follows:

```
SKB_Result
SKB_Engine_CreateKeyAgreement(SKB_Engine*                self,
                              SKB_KeyAgreementAlgorithm key_agreement_algorithm,
                              const void*               key_agreement_parameters,
                               SKB_KeyAgreement**        key_agreement);
```

The following are the parameters used:

self

Pointer to the pre-initialized engine.

key_agreement_algorithm

Key agreement algorithm to be used. Available algorithms are defined in the `SKB_KeyAgreementAlgorithm` enumeration (see §7.11.11).

key_agreement_parameters

Pointer to a structure providing the necessary parameters for the particular key agreement algorithm:

- For the `SKB_KEY_AGREEMENT_ALGORITHM_ECDH` algorithm, this parameter must point to the `SKB_EccParameters` structure (see §7.10.23).
- For the `SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH` algorithm, this parameter must point to the `SKB_PrimeDhParameters` structure (see §7.10.24).
- For the `SKB_KEY_AGREEMENT_ALGORITHM_ECDH_STATIC` algorithm, this parameter must point to the `SKB_EcdhParameters` structure (see §7.10.25).

key_agreement

Address of a pointer to the `SKB_KeyAgreement` object which will be created by this method.

7.9.12 SKB_Engine_UpgradeExportedData

This method upgrades an exported `SKB_SecureData` object to the latest version as described in §3.7.

The method is declared as follows:

```
SKB_Result
SKB_Engine_UpgradeExportedData(SKB_Engine*     engine,
                               const SKB_Byte* input,
                               SKB_Size        input_size,
                               SKB_Byte*       buffer,
                               SKB_Size*       buffer_size);
```

The following are the parameters used:

engine

Pointer to the pre-initialized engine.

input

Input data buffer containing the previously exported `SKB_SecureData` object that needs to be upgraded to the latest export format.

input_size

Size of the `input` buffer in bytes.

buffer

This parameter is either NULL or a pointer to the memory buffer where the upgraded data is to be written.

If this parameter is NULL, the method simply returns, in buffer_size, the number of bytes that would be sufficient to hold the output, and returns SKB_SUCCESS.

If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the output, the method stores the output there, sets buffer_size to the exact number of bytes stored, and returns SKB_SUCCESS. If the buffer is not large enough, then the method sets buffer_size to the number of bytes that would be sufficient, and returns SKB_ERROR_BUFFER_TOO_SMALL.

buffer_size

Pointer to a variable that holds the size of the memory buffer in bytes where the output is to be stored. For more details, see the description of the buffer parameter.

7.9.13 SKB_SecureData_GetInfo

This method provides information about the size and type of contents stored within a particular SKB_SecureData object.

The method is declared as follows:

```
SKB_Result
SKB_SecureData_GetInfo(const SKB_SecureData* self,
                       SKB_DataInfo*         info);
```

The following are the parameters used:

self

Pointer to the SKB_SecureData object whose size and type you want to know.

info

Pointer to the SKB_DataInfo structure, which will be populated by this method to return the characteristics of the SKB_SecureData object (see §7.10.3).

7.9.14 SKB_SecureData_Export

This method returns a protected form of the contents of a particular SKB_SecureData object. This protected data is intended for exporting keys to a persistent storage. Later the exported data can be imported back into SKB using the SKB_Engine_CreateDataFromExported method (see §7.9.6).

The method is declared as follows:

```
SKB_Result
SKB_SecureData_Export(const SKB_SecureData* self,
                      SKB_ExportTarget      target,
                      const void*           target_parameters,
                      SKB_Byte*             buffer,
                      SKB_Size*             buffer_size);
```

The following are the parameters used:

self

Pointer to the `SKB_SecureData` object to be exported.

target

Export type to be used. Available export types are defined in the `SKB_ExportTarget` enumeration (see §7.11.9).

target_parameters

Currently, this parameter is not used.

buffer

This parameter is either `NULL` or a pointer to the memory buffer where the exported data is to be written.

If this parameter is `NULL`, the method simply returns, in `buffer_size`, the number of bytes that would be sufficient to hold the exported data, and returns `SKB_SUCCESS`.

If this parameter points to a memory buffer (it is not `NULL`), and the buffer size is large enough to hold the exported data, the method stores the exported data there, sets `buffer_size` to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets `buffer_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

buffer_size

Pointer to a variable that holds the size of the memory buffer in bytes where the exported data is to be stored. For more details, see the description of the `buffer` parameter.

7.9.15 SKB_SecureData_Wrap

This method wraps (encrypts) the contents of a particular `SKB_SecureData` object using a specified cipher and wrapping key. For more information on wrapping secure data, see §3.2.

The method is declared as follows:

```
SKB_Result
SKB_SecureData_Wrap(const SKB_SecureData* self,
                    SKB_CipherAlgorithm   wrapping_algorithm,
                    const void*           wrapping_parameters,
                    const SKB_SecureData* wrapping_key,
                    SKB_Byte*             buffer,
                    SKB_Size*             buffer_size);
```

The following are the parameters used:

self

Pointer to the `SKB_SecureData` object whose contents need to be wrapped.

`wrapping_algorithm`

Wrapping algorithm to be used. The available algorithms are defined in the `SKB_CipherAlgorithm` enumeration (see §7.11.3).

Currently, only the following algorithms are supported for wrapping:

- `SKB_CIPHER_ALGORITHM_AES_128_CBC`
- `SKB_CIPHER_ALGORITHM_AES_192_CBC`
- `SKB_CIPHER_ALGORITHM_AES_256_CBC`
- `SKB_CIPHER_ALGORITHM_RSA`
- `SKB_CIPHER_ALGORITHM_RSA_1_5`
- `SKB_CIPHER_ALGORITHM_RSA_OAEP`
- `SKB_CIPHER_ALGORITHM_XOR`

The AES-based algorithms can only be used on `SKB_SecureData` objects whose data type is `SKB_DATA_TYPE_BYTES` or `SKB_DATA_TYPE_ECC_PRIVATE_KEY` (see §7.11.1).

The RSA-based algorithms and the `SKB_CIPHER_ALGORITHM_XOR` algorithm can only be used on `SKB_SecureData` objects whose data type is `SKB_DATA_TYPE_BYTES`.

`wrapping_parameters`

Pointer to a structure that provides additional parameters for the wrapping algorithm.

This parameter is applicable only if you use one of the following algorithms (for others, it should be `NULL`):

- `SKB_CIPHER_ALGORITHM_AES_128_CBC`
- `SKB_CIPHER_ALGORITHM_AES_192_CBC`
- `SKB_CIPHER_ALGORITHM_AES_256_CBC`

Then this parameter can be used to provide a specific initialization vector to the AES wrapping algorithm. In that case, you should point this parameter to the `SKB_AesWrapParameters` structure (see §7.10.20) where the initialization vector is specified. If this structure is not provided (`wrapping_parameters` is `NULL`), the AES algorithm generates a random initialization vector.

`wrapping_key`

Pointer to the `SKB_SecureData` object containing the wrapping key.

If one of the RSA-based algorithms is used, the data type of this `SKB_SecureData` object must be `SKB_DATA_TYPE_RSA_PUBLIC_KEY_CONTEXT` (see §7.11.1).

`buffer`

This parameter is either `NULL` or a pointer to the memory buffer where the output is to be stored.

If this parameter is `NULL`, the method simply returns, in `buffer_size`, the number of bytes that would be sufficient to hold the output, and returns `SKB_SUCCESS`.

If this parameter points to a memory buffer (it is not `NULL`), and the buffer size is large enough to hold the output, the method stores the output there, sets `buffer_size` to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets `buffer_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

For information on the way the output buffer is formatted in case you use the AES-based algorithms, see §8.2.3.

`buffer_size`

Pointer to a variable that holds the size of the memory buffer in bytes where the output data is to be stored. For more details, see the description of the `buffer` parameter.

7.9.16 SKB_SecureData_Derive

This method creates a new `SKB_SecureData` object from another `SKB_SecureData` object using a particular derivation algorithm. This method can only be used on `SKB_SecureData` objects whose data type is `SKB_DATA_TYPE_BYTES` (see §7.11.1).

The method is declared as follows:

```
SKB_Result
SKB_SecureData_Derive(const SKB_SecureData*   self,
                      SKB_DerivationAlgorithm algorithm,
                      const void*             parameters,
                      SKB_SecureData**        data);
```

The following are the parameters used:

`self`

Pointer to the `SKB_SecureData` object from which a new `SKB_SecureData` object needs to be derived.

`algorithm`

Derivation algorithm to be used. The available algorithms are defined in the `SKB_DerivationAlgorithm` enumeration (see §7.11.7).

`parameters`

Pointer to a structure containing parameters for the derivation algorithm. For different algorithms, a different structure must be provided:

- If the `SKB_DERIVATION_ALGORITHM_SLICE` or `SKB_DERIVATION_ALGORITHM_BLOCK_SLICE` algorithm is used, this parameter must point to the `SKB_SliceDerivationParameters` structure (see §7.10.9).
- If the `SKB_DERIVATION_ALGORITHM_SELECT_BYTES` algorithm is used, this parameter must point to the `SKB_SelectBytesDerivationParameters` structure (see §7.10.10).
- If the `SKB_DERIVATION_ALGORITHM_CIPHER` algorithm is used, this parameter must point to the `SKB_CipherDerivationParameters` structure (see §7.10.11).
- If the `SKB_DERIVATION_ALGORITHM_SHA_1` algorithm is used, this parameter may point to the `SKB_Sha1DerivationParameters` structure, which specifies how many times the SHA-1 algorithm should be executed and how many bytes from the result should be derived (see §7.10.12). If the parameter is `NULL`, the SHA-1 algorithm will be executed once and all 20 bytes of the output will be derived as a new `SKB_SecureData` object.

- If the `SKB_DERIVATION_ALGORITHM_SHA_256` algorithm is used, this parameter may point to the `SKB_Sha256DerivationParameters` structure, which provides the plain buffers that should be prepended and appended to the `SKB_SecureData` object processed (see §7.10.13). If the parameter is `NULL`, SKB will assume that there are no plain data buffers to be prepended or appended.
- If the `SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128` or `SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128_L16BIT` algorithm is used, this parameter must point to the `SKB_Nist800108CounterCmacAes128Parameters` structure (see §7.10.14).
- If the `SKB_DERIVATION_ALGORITHM_OMA_DRM_KDF2` algorithm is used, this parameter must point to the `SKB_OmaDrmKdf2DerivationParameters` structure (see §7.10.15).
- If the `SKB_DERIVATION_ALGORITHM_RAW_BYTES_FROM_ECC_PRIVATE` algorithm is used, this parameter may point to the `SKB_RawBytesFromEccPrivateDerivationParameters` structure, which specifies whether the output should be encoded in little-endian or big-endian (see §7.10.16). If the parameter is `NULL`, the output will be encoded in little-endian.
- If the `SKB_DERIVATION_ALGORITHM_SHA_AES` algorithm is used, this parameter must point to the `SKB_ShaAesDerivationParameters` structure (see §7.10.17).
- If the `SKB_DERIVATION_ALGORITHM_XOR` algorithm is used, this parameter must point to the `SKB_GenericDerivationParameters` structure (see §7.10.18).
- For all other key derivation algorithms, this parameter is not used and therefore should be `NULL`.

`data`

Address of a pointer that will point to the new derived `SKB_SecureData` object when this method is executed.

7.9.17 SKB_SecureData_GetPublicKey

This method returns a public key that corresponds to the supplied private key.

Currently, this method supports only ECC keys, but not RSA.

The method is declared as follows:

```
SKB_Result
SKB_SecureData_GetPublicKey(const SKB_SecureData* self,
                            SKB_DataFormat        format,
                            const void*           parameters,
                            SKB_Byte*             output,
                            SKB_Size*             output_size);
```

The following are the parameters used:

`self`

Pointer to the `SKB_SecureData` object containing the private key. From this key, the public key will be derived.

`format`

Format in which the derived public key should be stored in the returned buffer of bytes. The available formats are defined in the `SKB_DataFormat` enumeration (see §7.11.2).

Currently, the only valid value is `SKB_DATA_FORMAT_ECC_BINARY`.

`parameters`

Pointer to a structure containing parameters necessary for deriving the public key.

Since SKB supports only ECC key generation, this parameter should point to the `SKB_EccParameters` structure, which specifies the ECC curve type (see §7.10.23).

`output`

This parameter is either `NULL` or a pointer to the memory buffer where the output is to be stored.

If this parameter is `NULL`, the method simply returns, in **`output_size`**, the number of bytes that would be sufficient to hold the output, and returns `SKB_SUCCESS`.

If this parameter points to a memory buffer (it is not `NULL`), and the buffer size is large enough to hold the output, the method stores the output there, sets **`output_size`** to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets **`output_size`** to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

After successfully executing the method, the content of the **`output`** parameter will be a pointer to a buffer of bytes containing the public key. For information on the format used, see §8.5.

`output_size`

Pointer to a variable that holds the size of the memory buffer in bytes where the public key is to be stored. For more details, see the description of the **`output`** parameter.

7.9.18 SKB_SecureData_Release

This method releases the specified `SKB_SecureData` object from memory. It should always be called when the object is no longer needed.

The method is declared as follows:

```
SKB_Result
SKB_SecureData_Release(SKB_SecureData* self);
```

The parameter **`self`** is a pointer to the `SKB_SecureData` object that should be released.

7.9.19 SKB_Cipher_ProcessBuffer

This method performs either data encryption or decryption depending on the parameters of the previously created `SKB_Cipher` object (see §7.8.3).

The method is declared as follows:

```
SKB_Result
SKB_Cipher_ProcessBuffer(SKB_Cipher*     self,
                         const SKB_Byte* in_buffer,
                         SKB_Size        in_buffer_size,
                         SKB_Byte*       out_buffer,
                         SKB_Size*       out_buffer_size,
                         const SKB_Byte* iv,
                              SKB_Size         iv_size);
```

The following are the parameters used:

`self`

Pointer to the previously created `SKB_Cipher` object, which contains all the necessary parameters.

`in_buffer`

Pointer to a buffer of data to be encrypted or decrypted.

For block ciphers, this parameter must point to the beginning of a cipher block.

For the ElGamal ECC cipher, this parameter must point to a buffer of bytes described in §8.4.

`in_buffer_size`

Size in bytes of the data buffer to be encrypted or decrypted.

For the DES and Triple DES cipher, this parameter must be a multiple of 8.

For the AES cipher in the ECB or CBC mode, this parameter must be a multiple of 16.

For the RSA cipher, this parameter must be the size of the entire encrypted message, but not more than the length of the RSA key.

`out_buffer`

This parameter is either `NULL` or a pointer to the memory buffer where the output is to be stored.

If this parameter is `NULL`, the method simply returns, in `out_buffer_size`, the number of bytes that would be sufficient to hold the output, and returns `SKB_SUCCESS`.

If this parameter points to a memory buffer (it is not `NULL`), and the buffer size is large enough to hold the output, the method stores the output there, sets `out_buffer_size` to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets `out_buffer_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

For the ElGamal ECC cipher, the output buffer will contain the X coordinate of the decrypted point in big-endian notation. It is the caller's responsibility to extract the decrypted message from this output according to the way the message was encrypted.

SKB supports in-place encryption and decryption, which means that the `out_buffer` and `in_buffer` parameters may point to the same memory location. Then, the output of this method will overwrite the input.

`out_buffer_size`

Pointer to a variable that holds the size of the memory buffer in bytes where the output data is to be stored. For more details, see the description of the `out_buffer` parameter.

`iv`

Pointer to the initialization vector if you use the DES or Triple DES cipher in the CBC mode, or AES cipher in the CBC or CTR mode. For other cases, this parameter should contain `NULL`.

The initialization vector must be provided in the first call of this method. In subsequent calls, you may set the `iv` parameter to NULL, in which case, SKB will interpret the provided input buffer as continuation of the same message and will use the initialization vector that is internally preserved from the last method call (this approach is useful for processing very large data buffers that may not fit in the memory). In other words, if you provide the initialization vector, SKB will interpret the input buffer as a new message.

iv_size

Size in bytes of the initialization vector. It should be 0 if the `iv` parameter is `NULL`.

7.9.20 SKB_Cipher_Release

This method releases an `SKB_Cipher` object from memory. It should always be called when the object is no longer needed.

The method is declared as follows:

```
SKB_Result
SKB_Cipher_Release(SKB_Cipher* self);
```

The parameter `self` is a pointer to the `SKB_Cipher` object that should be released.

7.9.21 SKB_Transform_AddBytes

This method appends a plain buffer of bytes to a previously created `SKB_Transform` object. Data must be added to an `SKB_Transform` object before the actual transform algorithm (digest, signing, or verifying) can be executed.

The method is declared as follows:

```
SKB_Result
SKB_Transform_AddBytes(SKB_Transform*  self,
                       const SKB_Byte* data,
                       SKB_Size        data_size);
```

The following are the parameters used:

self

Pointer to the previously created `SKB_Transform` object.

data

Pointer to the buffer of data to be appended to the `SKB_Transform` object.

data_size

Size of the `data` buffer in bytes.

7.9.22 SKB_Transform_AddSecureData

This method appends the contents of an `SKB_SecureData` object to a previously created `SKB_Transform` object. Data must be added to an `SKB_Transform` object before the actual transform algorithm (digest, signing, or verifying) can be executed.

⚠ This method cannot be used for the `SKB_SIGNATURE_ALGORITHM_RSA` and `SKB_SIGNATURE_ALGORITHM_ECDSA` signing algorithms because they can operate only on plain input.

⚠ If you are using the ECDSA or RSA signing algorithms, this method is available only if the corresponding digest algorithm (see §7.11.6) is also included in the SKB library. For instance, if you want to calculate a signature of a cryptographic key using the ECDSA signing algorithm with SHA-384 as the hash function, the SHA-384 digest algorithm must also be included in the SKB library.

The method is declared as follows:

```
SKB_Result
SKB_Transform_AddSecureData(SKB_Transform*        self,
                            const SKB_SecureData* data);
```

The following are the parameters used:

`self`

Pointer to the previously created `SKB_Transform` object.

`data`

Pointer to the `SKB_SecureData` object whose contents must be appended to the `SKB_Transform` object.

7.9.23 SKB_Transform_GetOutput

This method executes a transform algorithm on a particular `SKB_Transform` object. The transform algorithm is specified during the creation of the `SKB_Transform` object, and the input data is provided using the `SKB_Transform_AddBytes` and `SKB_Transform_AddSecureData` methods.

⚠ After this method is called, you will no longer be allowed to execute the `SKB_Transform_AddBytes` and `SKB_Transform_AddSecureData` methods on the same `SKB_Transform` object. Also, you will not be allowed to call the `SKB_Transform_GetOutput` method again.

The method is declared as follows:

```
SKB_Result
SKB_Transform_GetOutput(SKB_Transform* self,
                        SKB_Byte*      output,
                        SKB_Size*      output_size);
```

The following are the parameters used:

`self`

Pointer to the `SKB_Transform` object on which the transform algorithm must be executed.

`output`

This parameter is either `NULL` or a pointer to the memory buffer where the transform output will be stored.

If this parameter is `NULL`, the method simply returns, in `output_size`, the number of bytes that would be sufficient to hold the output, and returns `SKB_SUCCESS`.

If this parameter points to a memory buffer (it is not `NULL`), and the buffer size is large enough to hold the output, the method stores the output there, sets `output_size` to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets `output_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

In the case of the `SKB_TRANSFORM_TYPE_VERIFY` transform, the output will be a single byte with the value 1 if the signature is verified, and 0 if it is not.

In the case of the ECDSA signature algorithm, the output will be a pointer to a buffer with the format described in §8.8.

`output_size`

Pointer to a variable that holds the size of the memory buffer in bytes where the transform output data is to be stored. For more details, see the description of the `output` parameter.

7.9.24 SKB_Transform_Release

This method releases the specified `SKB_Transform` object from memory.

The method is declared as follows:

```
SKB_Result
SKB_Transform_Release(SKB_Transform* self);
```

The parameter `self` is a pointer to the `SKB_Transform` object to be released.

7.9.25 SKB_KeyAgreement_GetPublicKey

This method creates a new public key that should be sent to the other party of the key agreement algorithm.

The method is declared as follows:

```
SKB_Result
SKB_KeyAgreement_GetPublicKey(SKB_KeyAgreement* self,
                              SKB_Byte*         public_key_buffer,
                              SKB_Size*         public_key_buffer_size);
```

The following are the parameters used:

`self`

Pointer to the previously created `SKB_KeyAgreement` object, which contains all the necessary parameters.

`public_key_buffer`

This parameter is either `NULL` or a pointer to the memory buffer where the public key will be stored.

If this parameter is `NULL`, the method simply returns, in `public_key_buffer_size`, the number of bytes that would be sufficient to hold the output, and returns `SKB_SUCCESS`.

If this parameter points to a memory buffer (it is not `NULL`), and the buffer size is large enough to hold the output, the method stores the output there, sets `public_key_buffer_size` to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets `public_key_buffer_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

For the `SKB_KEY_AGREEMENT_ALGORITHM_ECDH` and `SKB_KEY_AGREEMENT_ALGORITHM_ECDH_STATIC` algorithms, the public key is stored using the format described in §8.5.

For the `SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH` algorithm, the buffer size is 128 bytes, and it stores the public value encoded in big-endian.

`public_key_buffer_size`

Pointer to a variable that holds the size of the memory buffer in bytes where the public key is to be stored. For more details, see the description of the `public_key_buffer` parameter.

7.9.26 SKB_KeyAgreement_ComputeSecret

This method takes the public key received from the other party of the key agreement algorithm and computes the shared secret.

The method is declared as follows:

```
SKB_Result
SKB_KeyAgreement_ComputeSecret(SKB_KeyAgreement* self,
                               const SKB_Byte*   peer_public_key,
                               SKB_Size          peer_public_key_size,
                               SKB_Size          secret_size,
                               SKB_SecureData**  secret);
```

The following are the parameters used:

`self`

Pointer to the previously created `SKB_KeyAgreement` object, which contains all the necessary parameters.

`peer_public_key`

Pointer to the memory buffer where the public key received from the other party is stored.

For the `SKB_KEY_AGREEMENT_ALGORITHM_ECDH` and `SKB_KEY_AGREEMENT_ALGORITHM_ECDH_STATIC` algorithms, the public key is expected to be stored using the format described in §8.5.

For the `SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH` algorithm, the buffer has to be 128 bytes long, and it should store the public value encoded in big-endian.

`peer_public_key_size`

Size of the `peer_public_key` parameter in bytes. This size must be equal to the value returned by the `SKB_KeyAgreement_GetPublicKey` method used by the other key agreement party.

`secret_size`

Size of the desired shared secret data output.

To select the largest possible shared secret size, pass the value `SKB_KEY_AGREEMENT_MAXIMAL_SECRET_SIZE` as an input for this parameter.

`secret`

Address of a pointer to the `SKB_SecureData` object containing the shared secret data that will be created by this method. The bytes will be ordered using the big-endian notation.

7.9.27 SKB_KeyAgreement_Release

This method releases the specified `SKB_KeyAgreement` object from memory. It should always be called when the object is no longer needed.

The method is declared as follows:

```
SKB_Result
SKB_KeyAgreement_Release(SKB_KeyAgreement* self);
```

The parameter `self` is a pointer to the `SKB_KeyAgreement` object that should be released.

7.10 Supporting Structures

This section describes various supporting structures used by the SKB API.

7.10.1 SKB_EngineInfo

`SKB_EngineInfo` is a structure that is populated by the `SKB_Engine_GetInfo` method (see §7.9.4) to provide information about a particular `SKB_Engine` instance.

⚠ The contents of a populated `SKB_EngineInfo` structure will not be valid after the corresponding `SKB_Engine` object is released from the memory. During examination of the `SKB_EngineInfo` object, the `SKB_Engine` object must exist.

The `SKB_EngineInfo` structure is declared as follows:

```
typedef struct {
    struct {
        unsigned int major;
        unsigned int minor;
        unsigned int revision;
    } api_version;
    unsigned int        flags;
    unsigned int        property_count;
    SKB_EngineProperty* properties;
} SKB_EngineInfo;
```

The following are the properties used:

`major`, `minor`, `revision`
: Version numbers specified in the API header file.

`flags`
: Currently, this property is not used because there are no engine-specific flags defined.

`property_count`
: Number of elements in the `properties` array.

`properties`
: Array of engine properties with `property_count` elements, where each property is an `SKB_EngineProperty` structure (see §7.10.2).

The following properties are used:

- `implementation`: Cryptographic technique used by SKB. "`U`" identifies an implementation based on composite automata. "`P`" identifies an implementation based on polynomial encryption.
- `key_cache`: Key caching mechanism used by SKB. Available values are "`sqlite`", "`memory`", "`custom`", and "`none`". For information on key caching and its modes, see §4.2.3.
- `key_cache_max_items`: Maximum number of keys that can be cached in the memory. This property is available only if the memory key caching mechanism is used.
- `diversification_guid`: Unique diversification identifier consisting of 16 bytes in the hexadecimal format. SKB packages with the same binary implementation will have the same identifier. For information on diversification in SKB, see §1.1.8.
- `export_guid`: Export key identifier consisting of 16 bytes in the hexadecimal format. SKB packages with the same export key will have the same identifier.
- `export_key_version`: Current export key version in the one-way data upgrade scheme (see §3.7).

7.10.2 SKB_EngineProperty

`SKB_EngineProperty` is a name-value pair representing a particular `SKB_Engine` property in the `SKB_EngineInfo` structure (see §7.10.1).

The `SKB_EngineProperty` structure is declared as follows:

```
typedef struct {
    const char* name;
    const char* value;
} SKB_EngineProperty;
```

7.10.3 SKB_DataInfo

This structure is used by the `SKB_SecureData_GetInfo` method to return the size and type of a particular `SKB_SecureData` object (see §7.9.13).

The structure is declared as follows:

```
typedef struct {
    SKB_DataType type;
    SKB_Size     size;
} SKB_DataInfo;
```

The following are the properties used:

`type`

Type of the data stored within the `SKB_SecureData` object. Available types are defined in the `SKB_DataType` enumeration (see §7.11.1).

`size`

Size of the contents in bytes.

Value 0 means that the information is not available.

For the data type `SKB_DATA_TYPE_RSA_PRIVATE_KEY`, this value is the size of the modulus in bytes.

7.10.4 SKB_CtrModeCipherParameters

This structure provides an additional parameter for the `SKB_Engine_CreateCipher` method when the `SKB_CIPHER_ALGORITHM_AES_128_CTR`, `SKB_CIPHER_ALGORITHM_AES_192_CTR`, and `SKB_CIPHER_ALGORITHM_AES_256_CTR` algorithms are used (see §7.9.9).

The structure is declared as follows:

```
typedef struct {
    SKB_Size counter_size;
} SKB_CtrModeCipherParameters;
```

The property `counter_size` specifies the counter size in bytes.

7.10.5 SKB_DigestTransformParameters

This structure is used by the `SKB_Engine_CreateTransform` method if the `SKB_TRANSFORM_TYPE_DIGEST` transform is used (see §7.9.10). The purpose of this structure is to specify the digest algorithm.

The structure is declared as follows:

```
typedef struct {
    SKB_DigestAlgorithm algorithm;
} SKB_DigestTransformParameters;
```

The property `algorithm` specifies the digest algorithm to be used. The available algorithms are defined in the `SKB_DigestAlgorithm` enumeration (see §7.11.6).

7.10.6 SKB_SignTransformParameters

This structure is used by the `SKB_Engine_CreateTransform` method if the `SKB_TRANSFORM_TYPE_SIGN` transform is used (see §7.9.10). The purpose of this structure is to specify the signing algorithm and the signing key.

The structure is declared as follows:

```
typedef struct {
    SKB_SignatureAlgorithm  algorithm;
    const SKB_SecureData*   key;
} SKB_SignTransformParameters;
```

The following are the properties used:

algorithm

Signing algorithm to be used. The available signing algorithms are defined in the `SKB_SignatureAlgorithm` enumeration (see §7.11.5).

key

Pointer to the `SKB_SecureData` object, which contains the signing key.

7.10.7 SKB_SignTransformParametersEx

This structure is an extension to the `SKB_SignTransformParameters` structure. It provides the additional ability to specify the ECC curve type in case the ECDSA signature algorithm is used, or salt and salt length in case the RSA signature algorithm based on the Probabilistic Signature Scheme is used.

The structure is declared as follows:

```
typedef struct {
    SKB_SignTransformParameters base;
    const void*                 extension;
} SKB_SignTransformParametersEx;
```

The following are the properties used:

base

`SKB_SignTransformParameters` structure that specifies the signature algorithm and the key to be used (see §7.10.6).

extension

If one of the following signature algorithms is used, this pointer must point to the `SKB_EccParameters` structure, which specifies the ECC curve type to be used (see §7.10.23):

- SKB_SIGNATURE_ALGORITHM_ECDSA
- SKB_SIGNATURE_ALGORITHM_ECDSA_MD5
- SKB_SIGNATURE_ALGORITHM_ECDSA_SHA1
- SKB_SIGNATURE_ALGORITHM_ECDSA_SHA224
- SKB_SIGNATURE_ALGORITHM_ECDSA_SHA256
- SKB_SIGNATURE_ALGORITHM_ECDSA_SHA384
- SKB_SIGNATURE_ALGORITHM_ECDSA_SHA512

If one of the following algorithms is used, this pointer must point to the `SKB_RsaPssParameters` structure, which specifies the salt and salt length (see §7.10.22):

- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5_EX`
- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1_EX`
- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224_EX`
- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256_EX`
- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA384_EX`
- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA512_EX`

7.10.8 SKB_VerifyTransformParameters

This structure is used by the `SKB_Engine_CreateTransform` method if the `SKB_TRANSFORM_TYPE_VERIFY` transform is used (see §7.9.10). The purpose of this structure is to specify the verification algorithm, verification key, and the signature.

The structure is declared as follows:

```
typedef struct {
    SKB_SignatureAlgorithm  algorithm;
    const SKB_SecureData*   key;
    const SKB_Byte*         signature;
    SKB_Size                signature_size;
} SKB_VerifyTransformParameters;
```

The following are the properties used:

`algorithm`

Verification algorithm to be used. The available verification algorithms are defined in the `SKB_SignatureAlgorithm` enumeration (see §7.11.5).

Only the following algorithms are supported for verification:

- `SKB_SIGNATURE_ALGORITHM_AES_128_CMAC`
- `SKB_SIGNATURE_ALGORITHM_HMAC_SHA1`
- `SKB_SIGNATURE_ALGORITHM_HMAC_SHA224`
- `SKB_SIGNATURE_ALGORITHM_HMAC_SHA256`
- `SKB_SIGNATURE_ALGORITHM_HMAC_SHA384`
- `SKB_SIGNATURE_ALGORITHM_HMAC_SHA512`
- `SKB_SIGNATURE_ALGORITHM_HMAC_MD5`

`key`

Pointer to the `SKB_SecureData` object, which contains the verification key.

`signature`

Pointer to the data buffer containing the signature to be verified.

`signature_size`

Size of the `signature` buffer in bytes.

7.10.9 SKB_SliceDerivationParameters

This structure is used by the `SKB_SecureData_Derive` method if the `SKB_DERIVATION_ALGORITHM_SLICE` or `SKB_DERIVATION_ALGORITHM_BLOCK_SLICE` derivation algorithm is used (see §7.9.16). The purpose of this structure is to specify the range of bytes (first byte and the number of bytes) that should be derived from one `SKB_SecureData` object into another `SKB_SecureData` object.

The structure is declared as follows:

```
typedef struct {
    unsigned int first;
    unsigned int size;
} SKB_SliceDerivationParameters;
```

The following are the properties used:

`first`

Index of the first byte of the source `SKB_SecureData` object where the derived range starts. Bytes are numbered starting with 0.

If you are using the `SKB_DERIVATION_ALGORITHM_BLOCK_SLICE` algorithm, the value must be a multiple of 16.

`size`

Number of bytes to derive starting with the byte specified in the **`first`** parameter.

If you are using the `SKB_DERIVATION_ALGORITHM_BLOCK_SLICE` algorithm, the value must be a multiple of 16.

7.10.10 SKB_SelectBytesDerivationParameters

This structure is used by the `SKB_SecureData_Derive` method if the `SKB_DERIVATION_ALGORITHM_SELECT_BYTES` algorithm is used (see §7.9.16). It specifies whether odd or even bytes should be copied from the input, and how many bytes should be copied.

The structure is declared as follows:

```
typedef struct {
    SKB_SelectBytesDerivationVariant variant;
    unsigned int                     output_size;
} SKB_SelectBytesDerivationParameters;
```

The following are the properties used:

`variant`

Reference to a value of the `SKB_SelectBytesDerivationVariant` enumeration (see §7.11.14), which tells whether odd or even bytes should be selected.

`output_size`

Size of the output in bytes, which is the number of bytes copied from the input.

7.10.11 SKB_CipherDerivationParameters

This structure is used by the `SKB_SecureData_Derive` method if the `SKB_DERIVATION_ALGORITHM_CIPHER` algorithm is used (see §7.9.16). The purpose of this structure is to specify all the necessary parameters to execute the derivation.

The structure is declared as follows:

```
typedef struct {
    SKB_CipherAlgorithm     cipher_algorithm;
    SKB_CipherDirection     cipher_direction;
    unsigned int            cipher_flags;
    const void*             cipher_parameters;
    const SKB_SecureData*   cipher_key;
    const SKB_Byte*         iv;
    SKB_Size                iv_size;
} SKB_CipherDerivationParameters;
```

The following are the properties used:

`cipher_algorithm`

Cipher algorithm to be executed on the input data. This is a reference to the `SKB_CipherAlgorithm` enumeration (see §7.11.3).

Currently, the `SKB_DERIVATION_ALGORITHM_CIPHER` algorithm supports only the following ciphers:

- `SKB_CIPHER_ALGORITHM_AES_128_ECB`
- `SKB_CIPHER_ALGORITHM_AES_128_CBC`
- `SKB_CIPHER_ALGORITHM_AES_192_ECB`
- `SKB_CIPHER_ALGORITHM_AES_192_CBC`
- `SKB_CIPHER_ALGORITHM_AES_256_ECB`
- `SKB_CIPHER_ALGORITHM_AES_256_CBC`
- `SKB_CIPHER_ALGORITHM_DES_ECB`
- `SKB_CIPHER_ALGORITHM_DES_CBC`

`cipher_direction`

Parameter that specifies whether the input data should be encrypted or decrypted. Available directions are defined in the `SKB_CipherDirection` enumeration (see §7.11.4).

`cipher_flags`

Optional flags for the cipher.

Currently, the only defined flag is `SKB_CIPHER_FLAG_HIGH_SPEED`. This flag can be set only for the AES cipher when it is intended to be used with high throughput, for example, media content decryption.

`cipher_parameters`

Pointer to a structure that provides additional parameters for the cipher.

Currently, this parameter must always be `NULL`.

`cipher_key`

Pointer to the `SKB_SecureData` object containing the encryption or decryption key.

`iv`

Pointer to the initialization vector if you use the AES or DES cipher in the CBC mode. For other cases, this parameter should contain `NULL`.

`iv_size`

Initialization vector size in bytes.

7.10.12 SKB_Sha1DerivationParameters

This structure is used by the `SKB_SecureData_Derive` method (see §7.9.16) if the `SKB_DERIVATION_ALGORITHM_SHA_1` algorithm is used (see §3.15.4.1). The purpose of this structure is to specify how many times the SHA-1 algorithm should be executed on the specified `SKB_SecureData` object and how many bytes should be derived from the final hash value.

The structure is declared as follows:

```
typedef struct {
    unsigned int round_count;
    unsigned int output_size;
} SKB_Sha1DerivationParameters;
```

The following are the properties used:

`round_count`

How many times the SHA-1 algorithm should be executed in a sequence.

0 is also a valid value. In this case, the SHA-1 value will not be calculated; the derived `SKB_SecureData` object will simply contain the first `output_size` bytes of the source `SKB_SecureData` object.

`output_size`

Number of bytes to be derived from the final output of the SHA-1 algorithm. For example, if `output_size` is 4, the first four bytes of the hash value will be derived as a new `SKB_SecureData` object.

The standard size of the SHA-1 output is 20 bytes. Hence, `output_size` cannot exceed 20.

7.10.13 SKB_Sha256DerivationParameters

This structure is used by the `SKB_SecureData_Derive` method (see §7.9.16) if the `SKB_DERIVATION_ALGORITHM_SHA_256` algorithm is used (see §3.15.4.2). The purpose of this structure is to provide the two plain data buffers that should be prepended and appended to the source `SKB_SecureData` object before the SHA-256 algorithm is executed.

This structure may be omitted (provided as `NULL`). In that case, SKB will assume that there are no plain data buffers prepended or appended to the source `SKB_SecureData` object.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte*         plain1;
    SKB_Size                plain1_size;
    const SKB_Byte*         plain2;
    SKB_Size                plain2_size;
} SKB_Sha256DerivationParameters;
```

The following are the properties used:

`plain1`

Pointer to a buffer of bytes that should be prepended to the source `SKB_SecureData` object before calculating the SHA-256 hash value.

This property can be `NULL`, in which case there will be no plain data prepended to the `SKB_SecureData` object.

`plain1_size`

Number of bytes in the `plain1` buffer.

`plain2`

Pointer to a buffer of bytes that should be appended to the source `SKB_SecureData` object before calculating the SHA-256 hash value.

This property can be `NULL`, in which case there will be no plain data appended to the `SKB_SecureData` object.

`plain2_size`

Number of bytes in the `plain2` buffer.

7.10.14 SKB_Nist800108CounterCmacAes128Parameters

This structure is used by the `SKB_SecureData_Derive` method if the `SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128` or `SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128_L16BIT` derivation algorithm is used (see §7.9.16). For more information on this derivation algorithm, see §3.15.6.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte*         label;
    SKB_Size                label_size;
    const SKB_Byte*         context;
    SKB_Size                context_size;
    SKB_Size                output_size;
} SKB_Nist800108CounterCmacAes128Parameters;
```

The following are the properties used:

`label`

Pointer to the label, a binary buffer that identifies the purpose for the derived key, as defined by the *NIST Special Publication 800-108*.

`label_size`

Size of the label in bytes.

`context`

Pointer to the context, a binary buffer containing the information related to the derived key, as defined by the *NIST Special Publication 800-108*.

`context_size`

Size of the context in bytes.

`output_size`

Size of the derivation output in bytes. It cannot exceed 4096 bytes and must be a multiple of 16.

7.10.15 SKB_OmaDrmKdf2DerivationParameters

This structure is used by the `SKB_SecureData_Derive` method if the `SKB_DERIVATION_ALGORITHM_OMA_DRM_KDF2` derivation algorithm is used (see §7.9.16). For more information on this derivation algorithm, see §3.15.7.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* label;
    SKB_Size        label_size;
    SKB_Size        output_size;
} SKB_OmaDrmKdf2DerivationParameters;
```

The following are the properties used:

`label`

Pointer to the buffer containing the “otherInfo” parameter as defined in the OMA DRM specification.

`label_size`

Size of the `label` buffer in bytes.

`output_size`

Size of the derivation output in bytes.

7.10.16 SKB_RawBytesFromEccPrivateDerivationParameters

This structure may be used by the `SKB_SecureData_Derive` method to specify the endianness of the output if the `SKB_DERIVATION_ALGORITHM_RAW_BYTES_FROM_ECC_PRIVATE` derivation algorithm is used (see §7.9.16). The purpose of this structure is to specify whether the output should be encoded in little-endian or big-endian. For more information on this derivation algorithm, see §3.15.8.

The structure is declared as follows:

```
typedef struct {
    unsigned int derivation_flags;
} SKB_RawBytesFromEccPrivateDerivationParameters;
```

If `derivation_flags` includes the `SKB_DERIVATION_FLAG_OUTPUT_IN_BIG_ENDIAN` flag, the output will be encoded in big-endian. Otherwise, the output will be encoded in little-endian.

7.10.17 SKB_ShaAesDerivationParameters

This structure is used by the `SKB_SecureData_Derive` method if the `SKB_DERIVATION_ALGORITHM_SHA_AES` derivation algorithm is used (see §7.9.16). For more information on this derivation algorithm, see §3.15.9.

The structure is declared as follows:

```
typedef struct {
    const SKB_SecureData* secure_p;
    const SKB_Byte*       plain_1;
    SKB_Size              plain_1_size;
    const SKB_Byte*       plain_2;
} SKB_ShaAesDerivationParameters;
```

The following are the properties used:

`secure_p`

Pointer to the `SKB_SecureData` object containing the “secure_p” value.

`plain_1`

Pointer to the “plain_1” buffer.

This property may be set to `NULL`. In that case, the simplified version of the derivation algorithm will be executed (see §3.15.9).

`plain_1_size`

Size of the “plain_1” buffer. It must be 0 if “plain_1” is set to `NULL`.

`plain_2`

Pointer to the “plain_2” buffer, which must be 16 bytes long.

7.10.18 SKB_GenericDerivationParameters

This structure is used by the `SKB_SecureData_Derive` method for those derivation algorithms that require secure or plain data input. Currently, this structure is used only by the `SKB_DERIVATION_ALGORITHM_XOR` derivation algorithm to specify either the plain data buffer or secure data buffer to be XOR-ed with the input key (see §3.15.11). You may use either the secure data input, or the plain data input, but not both at the same time.

The structure is declared as follows:

```
typedef struct {
    const SKB_SecureData* secure_input;
    const SKB_Byte*       plain_input;
    SKB_Size              plain_input_size;
} SKB_GenericDerivationParameters;
```

The following are the properties used:

`secure_input`

Pointer to the `SKB_SecureData` object containing the secure data input. If this parameter is provided, the `plain_input` parameter must be `NULL`.

`plain_input`

Pointer to the plain data input. If this parameter is provided, the `secure_input` parameter must be `NULL`.

`plain_input_size`

Size of the `plain_input` buffer. It must be 0 if `plain_input` is set to `NULL`.

7.10.19 SKB_EccDomainParameters

This structure defines domain parameters for a custom ECC curve, and therefore should be employed only when the `SKB_ECC_CURVE_CUSTOM` curve type of the `SKB_EccCurve` enumeration is used (see §7.11.10). Currently, custom ECC curves are supported only for the ECDSA, ECDH, and ECC key generation algorithms. For all other cases, this structure is not used.

The structure is declared as follows:

```
typedef struct {
    SKB_Size            prime_bit_length;
    SKB_Size            order_bit_length;
    const unsigned int* prime;
    const unsigned int* a;
    const unsigned int* gx;
    const unsigned int* gy;
    const unsigned int* order;
} SKB_EccDomainParameters;
```

The following are the properties used:

`prime_bit_length`

Bit-length of the `prime`, `a`, `gx`, and `gy` parameters.

`order_bit_length`

Bit-length of the order domain parameter.

`prime`

Pointer to the prime modulo of the field.

a

Pointer to the constant from the equation “$y^2 = x^3 + ax + b$”.

gx

Pointer to the X coordinate of the base point.

gy

Pointer to the Y coordinate of the base point.

order

Pointer to the order of the base point.

All domain parameters, except for `prime_bit_length` and `order_bit_length`, must be provided in protected form. To obtain the protected form of custom ECC domain parameters, use the Custom ECC Tool as described in §5.3.

7.10.20 SKB_AesWrapParameters

This structure provides a fixed initialization vector to the AES algorithm when the `SKB_SecureData_Wrap` method is used (see §7.9.15). If this structure is not provided, the AES wrapping algorithm generates a random initialization vector.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* iv;
} SKB_AesWrapParameters;
```

The `iv` property is a pointer to the byte buffer containing the initialization vector.

7.10.21 SKB_AesUnwrapParameters

A pointer to this structure can be passed to the `SKB_Engine_CreateDataFromWrapped` method (see §7.9.5) in case the CBC mode of the AES algorithm is used. This structure specifies the CBC padding type to be used. For information on available CBC padding types, see §8.2.3.

The structure is declared as follows:

```
typedef struct {
    SKB_CbcPadding padding;
} SKB_AesUnwrapParameters;
```

The `padding` property specifies the CBC padding type to be used. The available padding types are defined in the `SKB_CbcPadding` enumeration (see §7.11.13).

7.10.22 SKB_RsaPssParameters

This structure provides the ability to specify the salt and salt length when one of the following signature algorithms is used:

- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5_EX`
- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1_EX`
- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224_EX`
- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256_EX`
- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_384_EX`
- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_512_EX`

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* salt;
    SKB_Size        salt_length;
} SKB_RsaPssParameters;
```

The following are the properties used:

salt

Pointer to a byte buffer containing the salt value to be used. The value of this property cannot be `NULL`.

salt_length

Length of the salt value in bytes.

It must be equal or greater than 0, and must not exceed the hash function block size.

7.10.23 SKB_EccParameters

This structure provides additional parameters when the ECC functions are used.

The structure is declared as follows:

```
typedef struct {
    SKB_EccCurve             curve;
    SKB_EccDomainParameters* domain_parameters;
    const unsigned int*      random_value;
} SKB_EccParameters;
```

The following are the properties used:

curve

Specifies the ECC curve type to be used. The available curve types are defined in the `SKB_EccCurve` enumeration (see §7.11.10).

domain_parameters

Pointer to the `SKB_EccDomainParameters` structure, which provides domain parameters for a custom ECC curve (see §7.10.19).

This parameter should be set only when the `SKB_ECC_CURVE_CUSTOM` curve type is used.

Currently, custom ECC curves are supported only for the ECDSA, ECDH, and ECC key generation algorithms. For all other cases, there is no point setting this parameter.

`random_value`

Property that allows you to provide a fixed random value to the ECDSA algorithm.

Typically, the value of this property should be `NULL`, in which case SKB uses an internally generated random value.

However, if you want to pass a fixed number to be used as the random value, the number must be passed as an integer array containing the value in protected form. To obtain the protected form of the fixed random value, use the Custom ECC Tool as described in §5.3.

7.10.24 SKB_PrimeDhParameters

This structure is required by the `SKB_Engine_CreateKeyAgreement` method when the classical Diffie-Hellman algorithm (`SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH`) is selected.

The structure is declared as follows:

```
typedef struct {
    SKB_PrimeDhLength      length;
    const SKB_Byte*        data;
    const unsigned int*    random_value;
} SKB_PrimeDhParameters;
```

The following are the properties used:

`length`

Maximum bit-length of the prime “p”. The available values are defined in the `SKB_PrimeDhLength` enumeration (see §7.11.12).

`data`

Pointer to an integer array containing a combination of the prime “p” and generator “g” in protected form to be used by the Diffie-Hellman algorithm. To obtain this protected data buffer, use the Diffie-Hellman Tool as described in §5.4.

`random_value`

Property that allows you to provide a fixed random value to the Diffie-Hellman algorithm.

Typically, the value of this property should be `NULL`, in which case SKB uses an internally generated random value.

However, you can also pass a fixed number to be used as the random value. The fixed number must be passed as an integer array containing the value in protected form. To obtain the protected form of the fixed random value, use the Diffie-Hellman Tool as described in §5.4.

7.10.25 SKB_EcdhParameters

This structure is required by the `SKB_Engine_CreateKeyAgreement` method (see §7.9.11) when the `SKB_KEY_AGREEMENT_ALGORITHM_ECDH_STATIC` algorithm is used. This structure allows you to provide a fixed private ECC key to the ECDH algorithm.

The structure is declared as follows:

```
typedef struct {
    SKB_EccCurve                   curve;
    const SKB_EccDomainParameters* domain_parameters;
    const SKB_SecureData*          private_key;
} SKB_EcdhParameters;
```

The following are the properties used:

`curve`

Specifies the ECC curve type to be used. The available curve types are defined in the `SKB_EccCurve` enumeration (see §7.11.10).

`domain_parameters`

Pointer to the `SKB_EccDomainParameters` structure, which provides domain parameters of a custom ECC curve (see §7.10.19).

This parameter should be set only when the `SKB_ECC_CURVE_CUSTOM` curve type is used.

`private_key`

Pointer to the `SKB_SecureData` object that contains the fixed private ECC key that must be provided to the ECDH algorithm.

The type of the `SKB_SecureData` object must be `SKB_DATA_TYPE_ECC_PRIVATE_KEY` (see §7.11.1).

7.10.26 SKB_RawBytesParameters

This structure is required by the `SKB_Engine_GenerateSecureData` method (see §7.9.8) to generate an `SKB_SecureData` object containing a buffer of random raw bytes. The only purpose of this structure is to specify the number of bytes to generate.

The structure is declared as follows:

```
typedef struct {
    SKB_Size byte_count;
} SKB_RawBytesParameters;
```

The `byte_count` parameter specifies the number of bytes to be generated.

7.11 Enumerations

This section describes various enumerations defined in the SKB API.

7.11.1 SKB_DataType

This enumeration specifies the possible data types of the content encapsulated by an `SKB_SecureData` object.

The enumeration is defined as follows:

```
typedef enum {
    SKB_DATA_TYPE_BYTES,
    SKB_DATA_TYPE_RSA_PRIVATE_KEY,
    SKB_DATA_TYPE_ECC_PRIVATE_KEY,
    SKB_DATA_TYPE_RSA_PUBLIC_KEY_CONTEXT
} SKB_DataType;
```

The following are the values defined:

`SKB_DATA_TYPE_BYTES`

Raw bytes (for example, a DES or AES key).

`SKB_DATA_TYPE_RSA_PRIVATE_KEY`

Private RSA key.

`SKB_DATA_TYPE_ECC_PRIVATE_KEY`

Private ECC key.

`SKB_DATA_TYPE_RSA_PUBLIC_KEY_CONTEXT`

Public RSA key.

Currently, this data type is used only for wrapping raw bytes using the RSA algorithm (see §7.9.15), and loading plain public RSA keys using the `SKB_Engine_CreateDataFromWrapped` method (see §7.9.5).

7.11.2 SKB_DataFormat

This enumeration specifies the possible formats how a cryptographic key can be stored in a data buffer.

The enumeration is defined as follows:

```
typedef enum {
    SKB_DATA_FORMAT_RAW,
    SKB_DATA_FORMAT_PKCS8,
    SKB_DATA_FORMAT_PKCS1,
    SKB_DATA_FORMAT_ECC_BINARY
} SKB_DataFormat;
```

The following are the values defined:

`SKB_DATA_FORMAT_RAW`

Buffer of raw bytes (for example, a DES or AES key)

SKB_DATA_FORMAT_PKCS8

Private RSA key stored according to the PKCS#8 standard

SKB_DATA_FORMAT_PKCS1

Public RSA key stored according to the PKCS#1 standard

SKB_DATA_FORMAT_ECC_BINARY

Private ECC key stored using the format described in §8.6

7.11.3 SKB_CipherAlgorithm

This enumeration specifies cryptographic algorithms that are used for encrypting and decrypting data.

The enumeration is defined as follows:

```
typedef enum {
    SKB_CIPHER_ALGORITHM_NULL,
    SKB_CIPHER_ALGORITHM_AES_128_ECB,
    SKB_CIPHER_ALGORITHM_AES_128_CBC,
    SKB_CIPHER_ALGORITHM_AES_128_CTR,
    SKB_CIPHER_ALGORITHM_RSA,
    SKB_CIPHER_ALGORITHM_RSA_1_5,
    SKB_CIPHER_ALGORITHM_RSA_OAEP,
    SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA224,
    SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA256,
    SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA384,
    SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA512,
    SKB_CIPHER_ALGORITHM_RSA_OAEP_MD5,
    SKB_CIPHER_ALGORITHM_ECC_ELGAMAL,
    SKB_CIPHER_ALGORITHM_AES_192_ECB,
    SKB_CIPHER_ALGORITHM_AES_192_CBC,
    SKB_CIPHER_ALGORITHM_AES_192_CTR,
    SKB_CIPHER_ALGORITHM_AES_256_ECB,
    SKB_CIPHER_ALGORITHM_AES_256_CBC,
    SKB_CIPHER_ALGORITHM_AES_256_CTR,
    SKB_CIPHER_ALGORITHM_DES_ECB,
    SKB_CIPHER_ALGORITHM_DES_CBC,
    SKB_CIPHER_ALGORITHM_TRIPLE_DES_ECB,
    SKB_CIPHER_ALGORITHM_TRIPLE_DES_CBC,
    SKB_CIPHER_ALGORITHM_NIST_AES,
    SKB_CIPHER_ALGORITHM_AES_CMLA,
    SKB_CIPHER_ALGORITHM_RSA_CMLA,
    SKB_CIPHER_ALGORITHM_XOR,
} SKB_CipherAlgorithm;
```

The following are the values defined:

SKB_CIPHER_ALGORITHM_NULL

Value that identifies that no algorithm was used, meaning that the corresponding data is not encrypted.

This value is used by the `SKB_Engine_CreateDataFromWrapped` method to specify that the data to be loaded is in plain form (see §3.1).

`SKB_CIPHER_ALGORITHM_AES_128_ECB`

128-bit AES in the ECB mode

`SKB_CIPHER_ALGORITHM_AES_128_CBC`

128-bit AES in the CBC mode

`SKB_CIPHER_ALGORITHM_AES_128_CTR`

128-bit AES in the CTR mode

`SKB_CIPHER_ALGORITHM_RSA`

1024-bit to 2048-bit RSA with no padding

`SKB_CIPHER_ALGORITHM_RSA_1_5`

1024-bit to 2048-bit RSA with PKCS#1 version 1.5 padding

`SKB_CIPHER_ALGORITHM_RSA_OAEP`

1024-bit to 2048-bit RSA with OAEP padding that uses SHA-1

`SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA224`

1024-bit to 2048-bit RSA with OAEP padding that uses SHA-224

`SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA256`

1024-bit to 2048-bit RSA with OAEP padding that uses SHA-256

`SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA384`

1024-bit to 2048-bit RSA with OAEP padding that uses SHA-384

`SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA512`

1024-bit to 2048-bit RSA with OAEP padding that uses SHA-512

`SKB_CIPHER_ALGORITHM_RSA_OAEP_MD5`

1024-bit to 2048-bit RSA with OAEP padding that uses MD5

`SKB_CIPHER_ALGORITHM_ECC_ELGAMAL`

ElGamal ECC

`SKB_CIPHER_ALGORITHM_AES_192_ECB`

192-bit AES in the ECB mode

`SKB_CIPHER_ALGORITHM_AES_192_CBC`

192-bit AES in the CBC mode

`SKB_CIPHER_ALGORITHM_AES_192_CTR`

192-bit AES in the CTR mode

`SKB_CIPHER_ALGORITHM_AES_256_ECB`
: 256-bit AES in the ECB mode

`SKB_CIPHER_ALGORITHM_AES_256_CBC`
: 256-bit AES in the CBC mode

`SKB_CIPHER_ALGORITHM_AES_256_CTR`
: 256-bit AES in the CTR mode

`SKB_CIPHER_ALGORITHM_DES_ECB`
: DES in the ECB mode

`SKB_CIPHER_ALGORITHM_DES_CBC`
: DES in the CBC mode

`SKB_CIPHER_ALGORITHM_TRIPLE_DES_ECB`
: Triple DES in the ECB mode

`SKB_CIPHER_ALGORITHM_TRIPLE_DES_CBC`
: Triple DES in the CBC mode

`SKB_CIPHER_ALGORITHM_NIST_AES`
: AES key unwrapping algorithm defined by NIST. This cipher is supported only by the `SKB_Engine_CreateDataFromWrapped` method (see §7.9.5).

`SKB_CIPHER_ALGORITHM_AES_CMLA`
: CMLA AES unwrapping defined by the *CMLA Technical Specification*

`SKB_CIPHER_ALGORITHM_RSA_CMLA`
: CMLA RSA unwrapping defined by the *CMLA Technical Specification*

`SKB_CIPHER_ALGORITHM_XOR`
: Wrapping and unwrapping using XOR:

- If the `SKB_SecureData_Wrap` function is used (see §7.9.15), the key to be wrapped is XOR-ed with the wrapping key.
- If the `SKB_Engine_CreateDataFromWrapped` function is used (see §7.9.5), the wrapped buffer is XOR-ed with the unwrapping key.

In both cases, the two XOR-ed buffers must be of equal size.

7.11.4 SKB_CipherDirection

This enumeration specifies the possible directions (encryption or decryption) for the `SKB_Engine_CreateCipher` method (see §7.9.9) and the `SKB_DERIVATION_ALGORITHM_CIPHER` derivation algorithm (see §7.10.11).

The enumeration is defined as follows:

```
typedef enum {
    SKB_CIPHER_DIRECTION_ENCRYPT,
    SKB_CIPHER_DIRECTION_DECRYPT
} SKB_CipherDirection;
```

7.11.5 SKB_SignatureAlgorithm

This enumeration specifies the possible signing and verifying algorithms for the `SKB_Transform` object.

The enumeration is defined as follows:

```
typedef enum {
    SKB_SIGNATURE_ALGORITHM_AES_128_CMAC,
    SKB_SIGNATURE_ALGORITHM_HMAC_SHA1,
    SKB_SIGNATURE_ALGORITHM_HMAC_SHA224,
    SKB_SIGNATURE_ALGORITHM_HMAC_SHA256,
    SKB_SIGNATURE_ALGORITHM_HMAC_SHA384,
    SKB_SIGNATURE_ALGORITHM_HMAC_SHA512,
    SKB_SIGNATURE_ALGORITHM_HMAC_MD5,
    SKB_SIGNATURE_ALGORITHM_RSA,
    SKB_SIGNATURE_ALGORITHM_ECDSA,
    SKB_SIGNATURE_ALGORITHM_ECDSA_SHA1,
    SKB_SIGNATURE_ALGORITHM_ECDSA_SHA224,
    SKB_SIGNATURE_ALGORITHM_ECDSA_SHA256,
    SKB_SIGNATURE_ALGORITHM_ECDSA_SHA384,
    SKB_SIGNATURE_ALGORITHM_ECDSA_SHA512,
    SKB_SIGNATURE_ALGORITHM_ECDSA_MD5,
    SKB_SIGNATURE_ALGORITHM_RSA_MD5,
    SKB_SIGNATURE_ALGORITHM_RSA_SHA1,
    SKB_SIGNATURE_ALGORITHM_RSA_SHA224,
    SKB_SIGNATURE_ALGORITHM_RSA_SHA256,
    SKB_SIGNATURE_ALGORITHM_RSA_SHA384,
    SKB_SIGNATURE_ALGORITHM_RSA_SHA512,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA384,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA512,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5_EX,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1_EX,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224_EX,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256_EX,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA384_EX,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA512_EX,
} SKB_SignatureAlgorithm;
```

The following are the values defined:

`SKB_SIGNATURE_ALGORITHM_AES_128_CMAC`

128-bit AES-CMAC (based on OMAC1)

`SKB_SIGNATURE_ALGORITHM_HMAC_SHA1`

HMAC using SHA-1 as the hash function

`SKB_SIGNATURE_ALGORITHM_HMAC_SHA224`

HMAC using SHA-224 as the hash function

`SKB_SIGNATURE_ALGORITHM_HMAC_SHA256`

HMAC using SHA-256 as the hash function

`SKB_SIGNATURE_ALGORITHM_HMAC_SHA384`

HMAC using SHA-384 as the hash function

`SKB_SIGNATURE_ALGORITHM_HMAC_SHA512`

HMAC using SHA-512 as the hash function

`SKB_SIGNATURE_ALGORITHM_HMAC_MD5`

HMAC using MD5 as the hash function

`SKB_SIGNATURE_ALGORITHM_RSA`

1024-bit to 2048-bit RSA signature algorithms standardized in version 1.5 of PKCS#1 without a hash function (can only be executed on plain input, which is a digest of some hash function)

`SKB_SIGNATURE_ALGORITHM_ECDSA`

ECDSA with either standard or custom curves (can only be executed on plain input, which is a digest of some hash function)

`SKB_SIGNATURE_ALGORITHM_ECDSA_SHA1`

ECDSA with either standard or custom curves using SHA-1 as the hash function

`SKB_SIGNATURE_ALGORITHM_ECDSA_SHA224`

ECDSA with either standard or custom curves using SHA-224 as the hash function

`SKB_SIGNATURE_ALGORITHM_ECDSA_SHA256`

ECDSA with either standard or custom curves using SHA-256 as the hash function

`SKB_SIGNATURE_ALGORITHM_ECDSA_SHA384`

ECDSA with either standard or custom curves using SHA-384 as the hash function

`SKB_SIGNATURE_ALGORITHM_ECDSA_SHA512`

ECDSA with either standard or custom curves using SHA-512 as the hash function

`SKB_SIGNATURE_ALGORITHM_ECDSA_MD5`

ECDSA with either standard or custom curves using MD5 as the hash function

SKB_SIGNATURE_ALGORITHM_RSA_MD5

1024-bit to 2048-bit RSA signature algorithms standardized in version 1.5 of PKCS#1 using MD5 as the hash function

SKB_SIGNATURE_ALGORITHM_RSA_SHA1

1024-bit to 2048-bit RSA signature algorithms standardized in version 1.5 of PKCS#1 using SHA-1 as the hash function

SKB_SIGNATURE_ALGORITHM_RSA_SHA224

1024-bit to 2048-bit RSA signature algorithms standardized in version 1.5 of PKCS#1 using SHA-224 as the hash function

SKB_SIGNATURE_ALGORITHM_RSA_SHA256

1024-bit to 2048-bit RSA signature algorithms standardized in version 1.5 of PKCS#1 using SHA-256 as the hash function

SKB_SIGNATURE_ALGORITHM_RSA_SHA384

1024-bit to 2048-bit RSA signature algorithms standardized in version 1.5 of PKCS#1 using SHA-384 as the hash function

SKB_SIGNATURE_ALGORITHM_RSA_SHA512

1024-bit to 2048-bit RSA signature algorithms standardized in version 1.5 of PKCS#1 using SHA-512 as the hash function

SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5

1024-bit to 2048-bit RSA signature algorithms based on the Probabilistic Signature Scheme using MD5 as the hash function. A random salt value of 16 bytes will be generated.

SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1

1024-bit to 2048-bit RSA signature algorithms based on the Probabilistic Signature Scheme using SHA-1 as the hash function. A random salt value of 20 bytes will be generated.

SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224

1024-bit to 2048-bit RSA signature algorithms based on the Probabilistic Signature Scheme using SHA-224 as the hash function. A random salt value of 28 bytes will be generated.

SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256

1024-bit to 2048-bit RSA signature algorithms based on the Probabilistic Signature Scheme using SHA-256 as the hash function. A random salt value of 32 bytes will be generated.

SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA384

1024-bit to 2048-bit RSA signature algorithms based on the Probabilistic Signature Scheme using SHA-384 as the hash function. A random salt value of 48 bytes will be generated.

SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA512

1024-bit to 2048-bit RSA signature algorithms based on the Probabilistic Signature Scheme using SHA-512 as the hash function. A random salt value of 64 bytes will be generated.

SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5_EX

Same as `SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5` but allows specifying the salt value and length

SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1_EX

Same as `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1` but allows specifying the salt value and length

SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224_EX

Same as `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224` but allows specifying the salt value and length

SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256_EX

Same as `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256` but allows specifying the salt value and length

SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA384_EX

Same as `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA384` but allows specifying the salt value and length

SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA512_EX

Same as `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA512` but allows specifying the salt value and length

7.11.6 SKB_DigestAlgorithm

This enumeration specifies the available digest algorithms, and is defined as follows:

```
typedef enum {
    SKB_DIGEST_ALGORITHM_SHA1,
    SKB_DIGEST_ALGORITHM_SHA224,
    SKB_DIGEST_ALGORITHM_SHA256,
    SKB_DIGEST_ALGORITHM_SHA384,
    SKB_DIGEST_ALGORITHM_SHA512,
    SKB_DIGEST_ALGORITHM_MD5,
} SKB_DigestAlgorithm;
```

7.11.7 SKB_DerivationAlgorithm

This enumeration specifies the possible algorithms that can be used for deriving one `SKB_SecureData` object from another using the `SKB_SecureData_Derive` method (see §7.9.16).

The enumeration is defined as follows:

```
typedef enum {
    SKB_DERIVATION_ALGORITHM_SLICE,
    SKB_DERIVATION_ALGORITHM_BLOCK_SLICE,
    SKB_DERIVATION_ALGORITHM_SELECT_BYTES,
    SKB_DERIVATION_ALGORITHM_CIPHER,
    SKB_DERIVATION_ALGORITHM_SHA_1,
    SKB_DERIVATION_ALGORITHM_SHA_256,
    SKB_DERIVATION_ALGORITHM_SHA_384,
    SKB_DERIVATION_ALGORITHM_REVERSE_BYTES,
    SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128,
    SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128_L16BIT,
    SKB_DERIVATION_ALGORITHM_OMA_DRM_KDF2,
    SKB_DERIVATION_ALGORITHM_RAW_BYTES_FROM_ECC_PRIVATE,
    SKB_DERIVATION_ALGORITHM_CMLA_KDF,
    SKB_DERIVATION_ALGORITHM_SHA_AES,
    SKB_DERIVATION_ALGORITHM_XOR,
} SKB_DerivationAlgorithm;
```

The following are the values defined:

`SKB_DERIVATION_ALGORITHM_SLICE`

Derives a new `SKB_SecureData` object as a substring of bytes of another `SKB_SecureData` object (see §3.15.1)

`SKB_DERIVATION_ALGORITHM_BLOCK_SLICE`

Same as `SKB_DERIVATION_ALGORITHM_SLICE`, but it requires the index of the first byte and the number of bytes in the substring to be multiples of 16

`SKB_DERIVATION_ALGORITHM_SELECT_BYTES`

Derives a new `SKB_SecureData` object from the input `SKB_SecureData` object by copying only odd or even bytes from it (see §3.15.2)

`SKB_DERIVATION_ALGORITHM_CIPHER`

Derives a new `SKB_SecureData` object from the input `SKB_SecureData` object by encrypting or decrypting it with another key (see §3.15.3)

`SKB_DERIVATION_ALGORITHM_SHA_1`

Obtains a hash value from the input `SKB_SecureData` object by executing SHA-1 one or several times, and stores the specified substring of bytes from the output as a new `SKB_SecureData` object (see §3.15.4.1)

`SKB_DERIVATION_ALGORITHM_SHA_256`

Obtains a SHA-256 hash value from a buffer that contains a `SKB_SecureData` object, prefixed and suffixed with plain data, and stores the output as a new `SKB_SecureData` object (see §3.15.4.2)

`SKB_DERIVATION_ALGORITHM_SHA_384`

Obtains a hash value from the input `SKB_SecureData` object by executing SHA-384 and storing the entire 48-byte output as a new `SKB_SecureData` object (see §3.15.4.3)

`SKB_DERIVATION_ALGORITHM_REVERSE_BYTES`

Derives a new `SKB_SecureData` object where the order of bytes is reversed (see §3.15.5). You can use this derivation type to convert little-endian data buffers to big-endian and vice versa.

`SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128`

Derives a new `SKB_SecureData` object according to the key derivation function specified in the *NIST Special Publication 800-108*, using 128-bit AES-CMAC as the pseudorandom function in the counter mode (see §3.15.6). This algorithm version uses the 32-bit parameter "L".

`SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128_L16BIT`

Same as `SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128`, but it uses the 16-bit parameter "L".

`SKB_DERIVATION_ALGORITHM_OMA_DRM_KDF2`

Derives a new `SKB_SecureData` object according to KDF2 used in the RSAES-KEM-KWS scheme of the OMA DRM specification (see §3.15.7)

`SKB_DERIVATION_ALGORITHM_RAW_BYTES_FROM_ECC_PRIVATE`

Derives a new `SKB_SecureData` object with the type `SKB_DATA_TYPE_BYTES` from another `SKB_SecureData` object whose type is `SKB_DATA_TYPE_ECC_PRIVATE_KEY` (see §3.15.8)

`SKB_DERIVATION_ALGORITHM_CMLA_KDF`

Derives a new `SKB_SecureData` object according to the key derivation function defined in the *CMLA Technical Specification* (see §3.15.10)

`SKB_DERIVATION_ALGORITHM_SHA_AES`

Derives a new `SKB_SecureData` object using the algorithm described in §3.15.9

`SKB_DERIVATION_ALGORITHM_XOR`

Derives a new `SKB_SecureData` object by XOR-ing a key with plain data or another key as described in §3.15.11

7.11.8 SKB_TransformType

This enumeration specifies the available transform types used by the `SKB_Engine_CreateTransform` method to create `SKB_Transform` objects (see §7.9.10).

The enumeration is defined as follows:

```
typedef enum {
    SKB_TRANSFORM_TYPE_DIGEST,
    SKB_TRANSFORM_TYPE_SIGN,
    SKB_TRANSFORM_TYPE_VERIFY
} SKB_TransformType;
```

The following are the values defined:

`SKB_TRANSFORM_TYPE_DIGEST`

Transform for calculating a digest (hash value)

`SKB_TRANSFORM_TYPE_SIGN`

Transform for creating a signature

`SKB_TRANSFORM_TYPE_VERIFY`

Transform for verifying a signature.

7.11.9 SKB_ExportTarget

This enumeration specifies the various export types supported by the `SKB_SecureData_Export` method (see §7.9.14).

The enumeration is defined as follows:

```
typedef enum {
    SKB_EXPORT_TARGET_CLEARTEXT,
    SKB_EXPORT_TARGET_PERSISTENT,
    SKB_EXPORT_TARGET_CROSS_ENGINE,
    SKB_EXPORT_TARGET_CUSTOM
} SKB_ExportTarget;
```

The following are the values defined:

`SKB_EXPORT_TARGET_CLEARTEXT`

Not used by SKB.

`SKB_EXPORT_TARGET_PERSISTENT`

The `SKB_SecureData` object to be exported is encrypted with the export key (see §1.1.6) before it leaves the protected SKB domain.

Because exported data needs to be decrypted before it can be used by SKB, `SKB_SecureData` objects exported in this way can only be imported by SKB packages that have the same export key.

For details on the data format used by this method, see §8.1.

`SKB_EXPORT_TARGET_CROSS_ENGINE`

The `SKB_SecureData` object to be exported is passed to the unprotected outside world in the exact form the object exists in the memory.

This method is a faster but more limited approach of exporting and importing keys when compared to the `SKB_EXPORT_TARGET_PERSISTENT` method, as explained in §2.3.3.

`SKB_EXPORT_TARGET_CUSTOM`

Not used by SKB.

7.11.10 SKB_EccCurve

This enumeration specifies the available ECC curve types.

This enumeration is defined as follows:

```
typedef enum {
    SKB_ECC_CURVE_SECP_R1_160,
    SKB_ECC_CURVE_NIST_192,
    SKB_ECC_CURVE_NIST_224,
    SKB_ECC_CURVE_NIST_256,
    SKB_ECC_CURVE_NIST_384,
    SKB_ECC_CURVE_NIST_521,
    SKB_ECC_CURVE_CUSTOM
} SKB_EccCurve;
```

The following are the values defined:

`SKB_ECC_CURVE_SECP_R1_160`

160-bit prime curve recommended by SECG, SECP R1

`SKB_ECC_CURVE_NIST_192`

192-bit prime curve recommended by NIST (same as 192-bit SECG, SECP R1)

`SKB_ECC_CURVE_NIST_224`

224-bit prime curve recommended by NIST (same as 224-bit SECG, SECP R1)

`SKB_ECC_CURVE_NIST_256`

256-bit prime curve recommended by NIST (same as 256-bit SECG, SECP R1)

`SKB_ECC_CURVE_NIST_384`

384-bit prime curve recommended by NIST (same as 384-bit SECG, SECP R1).

Currently, this curve type is supported only for ECDSA, ECDH, and key generation, but not for decrypting and unwrapping.

`SKB_ECC_CURVE_NIST_521`

521-bit prime curve recommended by NIST (same as 521-bit SECG, SECP R1).

Currently, this curve type is supported only for ECDSA, ECDH, and key generation, but not for decrypting and unwrapping.

`SKB_ECC_CURVE_CUSTOM`

Prime ECC curve with custom domain parameters.

Currently, this curve type is supported only for ECDSA, ECDH, and key generation, but not for decrypting and unwrapping.

7.11.11 SKB_KeyAgreementAlgorithm

This enumeration specifies the available key agreement algorithms used by the `SKB_Engine_CreateKeyAgreement` method to create an `SKB_KeyAgreement` object (see §7.9.11).

The enumeration is defined as follows:

```
typedef enum {
    SKB_KEY_AGREEMENT_ALGORITHM_ECDH,
    SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH,
    SKB_KEY_AGREEMENT_ALGORITHM_ECDH_STATIC
} SKB_KeyAgreementAlgorithm;
```

The following are the values defined:

`SKB_KEY_AGREEMENT_ALGORITHM_ECDH`

Elliptic curve Diffie-Hellman with a randomly generated private ECC key

SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH

Classical Diffie-Hellman with the protected prime “p” and generator “g”

SKB_KEY_AGREEMENT_ALGORITHM_ECDH_STATIC

Elliptic curve Diffie-Hellman with a fixed private ECC key, which is provided as a secure data object

7.11.12 SKB_PrimeDhLength

This enumeration specifies the available maximum bit-lengths of the prime “p” for the classical Diffie-Hellman key agreement algorithm. The values of this enumeration are referenced by the `length` parameter of the `SKB_PrimeDhParameters` structure (see §7.10.24).

The enumeration is defined as follows:

```
typedef enum {
    SKB_PRIME_DH_LENGTH_1024
} SKB_PrimeDhLength;
```

The value `SKB_PRIME_DH_LENGTH_1024` specifies that the maximum bit-length of the prime “p” is 1024 bits.

7.11.13 SKB_CbcPadding

This enumeration specifies the CBC mode types that can be referenced by the `SKB_AesUnwrapParameters` structure (see §7.10.21).

The enumeration is defined as follows:

```
typedef enum {
    SKB_CBC_PADDING_TYPE_NONE,
    SKB_CBC_PADDING_TYPE_XMLENC
} SKB_CbcPadding;
```

The following are the values defined:

SKB_CBC_PADDING_TYPE_NONE

CBC mode with no padding (see §8.2.3.1)

SKB_CBC_PADDING_TYPE_XMLENC

CBC mode with the XML encryption padding (see §8.2.3.2)

7.11.14 SKB_SelectBytesDerivationVariant

This enumeration is used by the `SKB_SelectBytesDerivationParameters` structure (see §7.10.10) to specify whether odd or even bytes should be selected.

The enumeration is defined as follows:

```
typedef enum {
    SKB_SELECT_BYTES_DERIVATION_ODD_BYTES,
    SKB_SELECT_BYTES_DERIVATION_EVEN_BYTES,
} SKB_SelectBytesDerivationVariant;
```

The following are the values defined:

`SKB_SELECT_BYTES_DERIVATION_ODD_BYTES`

Odd bytes should be selected.

`SKB_SELECT_BYTES_DERIVATION_EVEN_BYTES`

Even bytes should be selected.

8 Data Formats

This is a reference chapter describing various data formats used in SKB.

8.1 Data Encrypted with an Export Key

Data that is encrypted with an export key (see §1.1.6) is a binary buffer that consists of a header and an encrypted content. The header can provide valuable information, especially if you are dealing with several SKB packages with different export keys, or if you are employing the one-way data upgrade deployment (see §3.7).

Figure 8.1 shows the format of data encrypted with an export key.

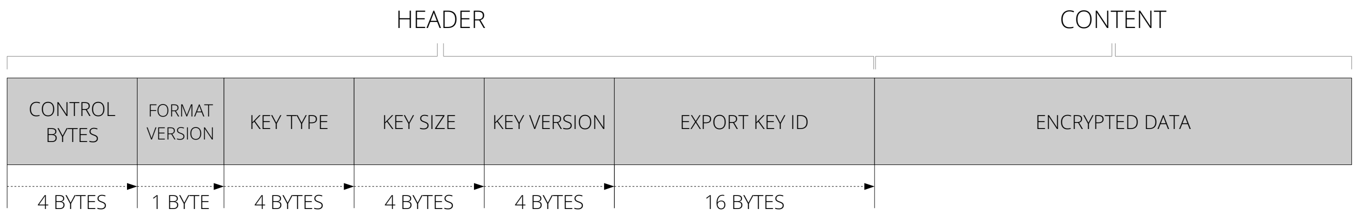

Figure 8.1: Export data format

The following are the components of the header:

Control bytes

Random bytes with specific properties that identify data exported by SKB.

Format version

Version of the export format. Currently, it is always "02".

Key type

Type of the exported key. The following values are used:

- "00" identifies raw bytes (for example, an AES or DES key)
- "01" identifies a private ECC key
- "02" identifies a private RSA key

Key size

Size of the exported key.

Key version

Key version in the one-way data upgrade scheme described in §3.7.

Export key ID

Identifier of the export key that was used in exporting the data. An SKB instance that needs to import this data has to have the same export key (with the same identifier).

You can find out the identifier of the export key of the current SKB instance using one of the following approaches:

- Look into the `export.id` file delivered with the SKB package (see §1.5).
- Call the `SKB_Engine_GetInfo` method and read the value of the `export_guid` property (see §7.9.4).

8.2 AES-Wrapped Data Buffer

This section describes the format of the encrypted data buffer, such as raw bytes, private RSA key, or the encrypted part of a wrapped private ECC key (see §8.7), that is either to be passed to the AES unwrapping algorithm (see §3.2), or is the output of the AES wrapping algorithm (see §3.4). Different modes of operation are described in separate subsections. In all modes, the big-endian encoding is used.

8.2.1 ECB Mode

In ECB mode, the size of the wrapped data buffer is an exact multiple of 16 bytes (block size for AES), as shown in Figure 8.2.

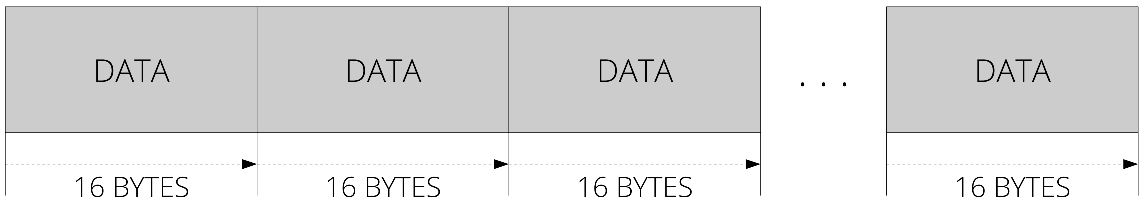

Figure 8.2: AES-wrapped buffer in the ECB mode

> ⚠ The ECB mode cannot be used to unwrap private RSA keys. Private ECC keys can be unwrapped only if the length of the wrapped key is a multiple of 128 bits.

8.2.2 CTR Mode

In CTR mode, the wrapped data buffer begins with the initialization vector, which is 16 bytes, followed by a data buffer of N bytes, as shown in Figure 8.3. N is an arbitrary number, not necessarily a multiple of 16 (block size for AES).

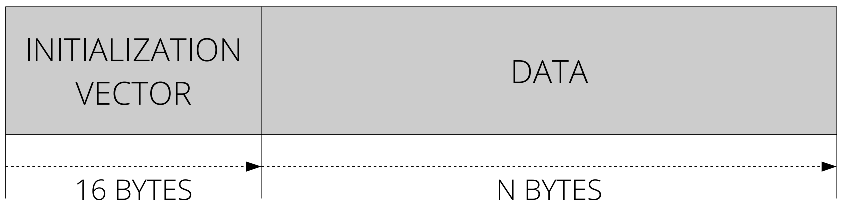

Figure 8.3: AES-wrapped buffer in the CTR mode

8.2.3 CBC Mode

In SKB, two CBC types are used — with no padding, and with XML encryption padding. In both cases, the wrapped data buffer begins with the initialization vector, which is 16 bytes, followed by a data buffer that is a multiple of 16 bytes (block size for AES).

The following subsections describe the two CBC mode types available.

8.2.3.1 No Padding

If no CBC padding is used, it is assumed that the size of the encrypted data buffer is an exact multiple of 16 bytes, and nothing is suffixed to the end of the buffer, as shown in Figure 8.4.

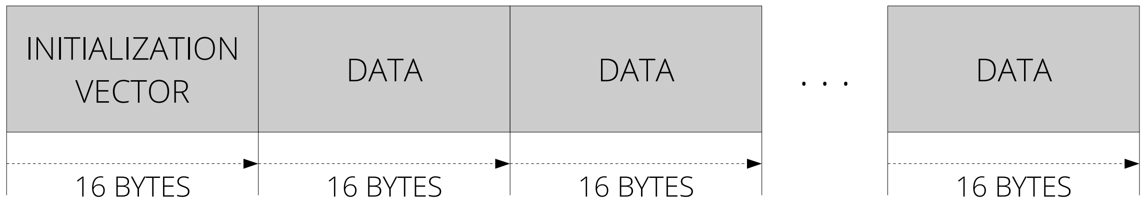

Figure 8.4: AES-wrapped buffer in the CBC mode with no padding

⚠ This CBC type cannot be used to unwrap RSA keys. If you are unwrapping ECC keys, this CBC type can only unwrap keys of the 256-bit and 384-bit curves recommended by NIST. Other ECC curve types are not supported.

8.2.3.2 XML Encryption Padding

SKB supports the CBC mode with padding conventions of the standard XML encryption, which is described in http://www.w3.org/TR/xmlenc-core. This means that if the size of the encrypted message within the data buffer is not an exact multiple of the block size, the last block must be padded by suffixing additional bytes to the data buffer to reach a multiple of the block size. The last byte in the last block must contain a number that specifies how many bytes must be stripped from the end of the decrypted data. Other added bytes are arbitrary. This approach is illustrated in Figure 8.5.

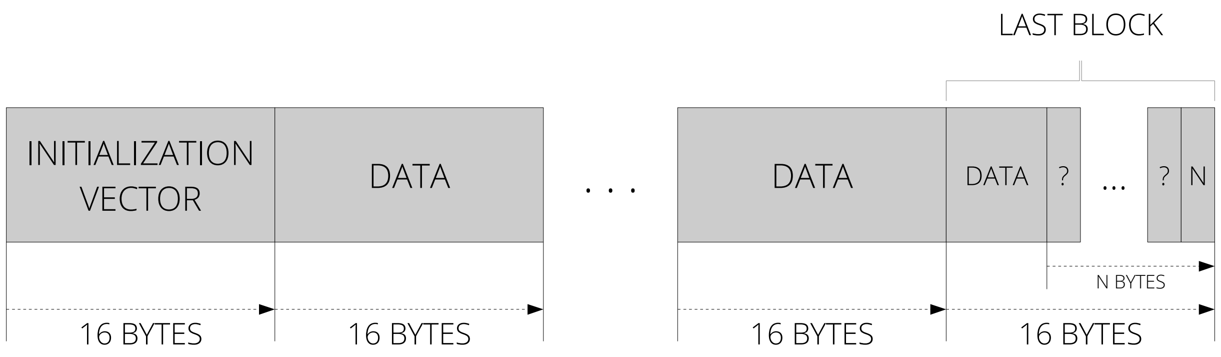

Figure 8.5: AES-wrapped buffer in the CBC mode with XML encryption padding

N is the number of bytes added to the last block. If the message size happens to be an exact multiple of 16 bytes, an additional block is added, in which the contents are arbitrary, but the last byte contains the number 16.

8.3 Key Format for the Triple DES Cipher

SKB supports two keying options for the Triple DES cipher:

- All three keys are distinct.

- Key 1 and key 2 are distinct, but key 3 is identical to key 1.

In both cases, the keys have to be provided as one buffer of bytes. SKB determines the keying option to be used based on the buffer size.

If the buffer is 192 bits long, SKB assumes the keys are provided in the format illustrated in Figure 8.6.

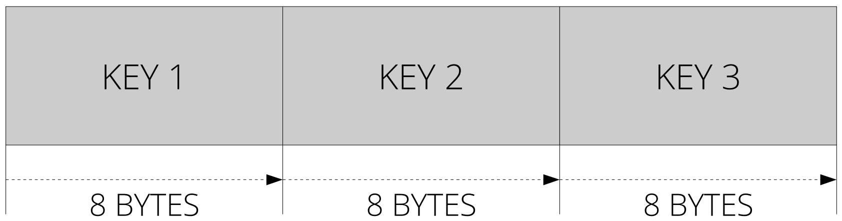

Figure 8.6: Triple DES key buffer containing three distinct keys

If the buffer is 128 bits long, SKB assumes the keys are provided in the format illustrated in Figure 8.7.

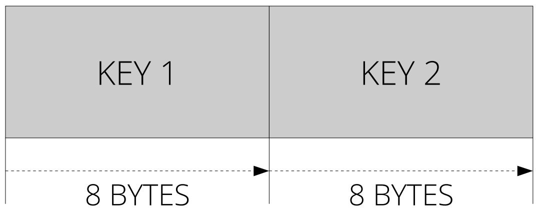

Figure 8.7: Triple DES key buffer containing only the first two keys

In the latter case, it is assumed that key 3 is identical to key 1.

8.4 Input Buffer for the ElGamal ECC Cipher

The buffer that is passed as an input to the ElGamal ECC decryption and unwrapping algorithms must contain coordinates of two points on an ECC curve. The two points correspond to the two arguments c_1 and c_2 of the ElGamal algorithm such that $(c_1, c_2) = (g^y, m' \cdot h^y)$, where g is the generator, y is the random value, m′ is the message, h is g^x, and x is the private key.

The input buffer must correspond to the format illustrated in Figure 8.8.

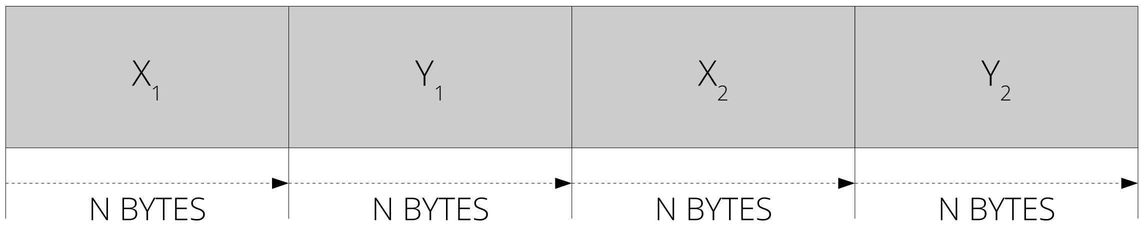

Figure 8.8: Input buffer for the ElGamal ECC algorithm

X_1 and Y_1 are the X and Y coordinates of c_1, and X_2 and Y_2 are the X and Y coordinates of c_2 encoded using the big-endian notation. N is the number of bytes used to store each coordinate, calculated as follows:

N = (L+7) / 8

where L is the length of the curve in bits.

8.5 Public ECC Key

SKB stores public ECC keys using the format illustrated in Figure 8.9.

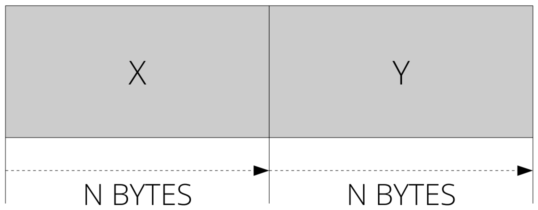

Figure 8.9: Public ECC key format

X and Y are the coordinates of the public key encoded using the big-endian notation, and N is the number of bytes used to store each coordinate. N depends on the ECC curve as shown in Table 8.1.

Curve type	**N (bytes)**
`SKB_ECC_CURVE_SECP_R1_160`	20
`SKB_ECC_CURVE_NIST_192`	24
`SKB_ECC_CURVE_NIST_224`	28
`SKB_ECC_CURVE_NIST_256`	32
`SKB_ECC_CURVE_NIST_384`	48
`SKB_ECC_CURVE_NIST_521`	66
`SKB_ECC_CURVE_CUSTOM`	(prime domain parameter bit-length) + 7 / 8

Table 8.1: Relationship between the curve type and N

8.6 Private ECC Key

SKB stores private ECC keys using the format illustrated in Figure 8.10 (this corresponds to the `SKB_DATA_FORMAT_ECC_BINARY` value of the `SKB_DataFormat` structure).

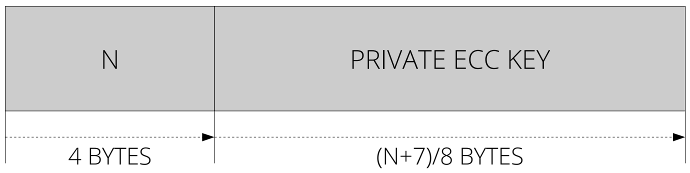

Figure 8.10: Private ECC key format

N, which is stored in the first four bytes, is the bit-length of the ECC curve. The rest of the buffer is the actual private ECC key. Both parameters are encoded using the big-endian notation.

8.7 AES-Wrapped Private ECC Key

If you are unwrapping an AES-wrapped private ECC key, the input buffer to the unwrapping algorithm must have the format illustrated in Figure 8.11 (all data must be encoded in big-endian).

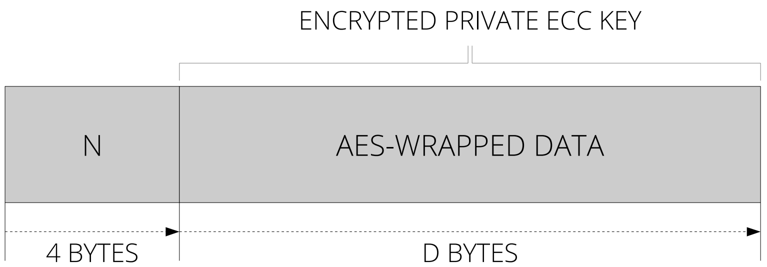

Figure 8.11: AES-wrapped private ECC key

The first 4 bytes must contain the number N in plain, which is the bit-length of the ECC curve used. The rest of the buffer is AES-wrapped data, containing the wrapped private ECC key and possibly some padding bytes. The AES-wrapped portion of the buffer must be formatted as described in §8.2.

Once the AES-wrapped content is decrypted, the first (N+7)/8 bytes are taken as the actual unwrapped private ECC key.

8.8 ECDSA Output

The format of the output of the ECDSA algorithm is illustrated in Figure 8.12.

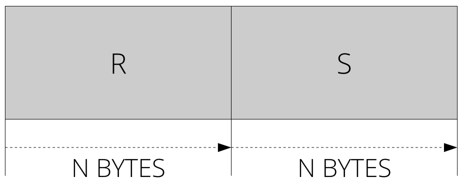

Figure 8.12: ECDSA output format

R and S are the two parameters of the signature used in the ECDSA algorithm, encoded using the big-endian notation. N depends on the ECC curve used as shown in Table 8.2.

Curve type	N (bytes)
`SKB_ECC_CURVE_SECP_R1_160`	21
`SKB_ECC_CURVE_NIST_192`	24
`SKB_ECC_CURVE_NIST_224`	28
`SKB_ECC_CURVE_NIST_256`	32
`SKB_ECC_CURVE_NIST_384`	48
`SKB_ECC_CURVE_NIST_521`	66
`SKB_ECC_CURVE_CUSTOM`	(order domain parameter bit-length) + 7 / 8

Table 8.2: Relationship between the curve type and N