SafeZone™
Cryptographic Abstraction Layer v4.1

Operations Manual

Document Revision: E
Document Date: 2015-02-11
Document Number: 007-912410-400
Document Status: Accepted

INSIDE Secure B.V.
Boxtelseweg 26A
5261 NE Vught
The Netherlands
Phone: +31-73-6581900
Fax: +31-73-6581999
http://www.insidesecure.com/

For further information contact: ESSEmbeddedHW-Support@insidesecure.com

Revision History

Doc Rev	Page(s) Section(s)	Date	Author	Purpose of Revision
A	All	2011-03-25	RWI JBO KLA	• Created new document derived from SafeZone CAL v4.0, Rev A. • Updated Figure 1 with CAL-PK. • Added new CAL API CPRM header file. • Updates based on the full review.
B	All	2011-05-27	KLA RWI	• Added Multipart hash example to 4.2.1 • Added Camellia example to 4.4 • Added MULTI2 example to 4.5 • Added CPRM (C2 key derivation) and C2 cipher examples (including C-CBC continuation) in new Chapter 6
C	All	2011-09-02	MHO KLA	• Template update and small editorial changes. • Updated DH example: 256 -> 512 bits & removed unneeded initialization of private keys.
D	All	2013-02-14	FvdM	• Update template
E	Ch 6	2015-02-11	MHO	• Updated for Secure Debug functionality example

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.1 Purpose

This Operations Manual covers the SafeZone Cryptographic Abstraction Layer API, from here on referred to as the CAL API.

This manual targets developers of software that use the CAL API. This document serves the needs where the CAL API Reference Manual might be too comprehensive. SafeZone CAL abstracts underlying cryptographic hardware or software and provides a consistent API to make use of it. Both symmetric and asymmetric cryptographic algorithms are being abstracted. Notice that most software should aim to use a higher-level cryptographic API, such as PKCS#11.

1.2 Scope

This document assumes that the reader understands the basics of cryptographic operations. This guide describes how to do typical cryptographic operations using CAL but does not explain what each operation is intended for.

This document covers CAL from the developers' perspective, mostly by example. The full Application Programming Interface (API) of CAL is described in *SafeZone CAL Reference Manual* [1].

For porting or adapting CAL for a new platform or operating system, read the *SafeZone CM-SDK Porting Guide Addendum* [2] for directions.

1.3 Related Documents

The following documents are part of the documentation set.

Ref	Document	Document Numbers
[1]	SafeZone CAL Reference Manual	007-912410-305
[2]	SafeZone CM SDK Porting Guide Addendum	007-910630-304
[3]	SafeZone CAL Operations Manual (this document)	007-912410-400

This information is correct at the time of document release. INSIDE Secure reserves the right to update the related documents without updating this document. Please contact INSIDE Secure for the latest document revisions.

For more information or support, please go to https://essoemsupport.insidesecure.com/ for our online support system. In case you do not have an account for this system, please ask one of your colleagues who already has an account to create one for you or send an e-mail to ESSEmbeddedHW-Support@insidesecure.com.

1.4 Target Audience

This document is intended for application developers.

1.5 Conventions

Documentation conventions and terminology are described in Appendix A.

2 Synopsis of the CAL module

CAL is the Cryptographic Abstraction Layer for SafeZone. Its purpose is to provide a well defined set of crypto APIs to upper layers regardless of where and how the features/schemes are implemented below. For example, the applications above CAL will always see the same CAL without needing to care if things are finally getting done in software, hardware or perhaps in a hybrid manner.

This document provides a brief introduction on how to use CAL APIs for some common cryptographic operations. The goal of this document is to bootstrap a developer quickly to a stage where he or she is able to use the majority of CAL APIs for typical cryptographic tasks. For more advanced use cases, the developer will need to refer to the CAL API Reference Manual. When reading this document, it is advisable to keep the API Reference Manual close at hand to help understand the examples better.

The following illustration shows simplified view of the various SafeZone/CAL software modules and their portability layers.

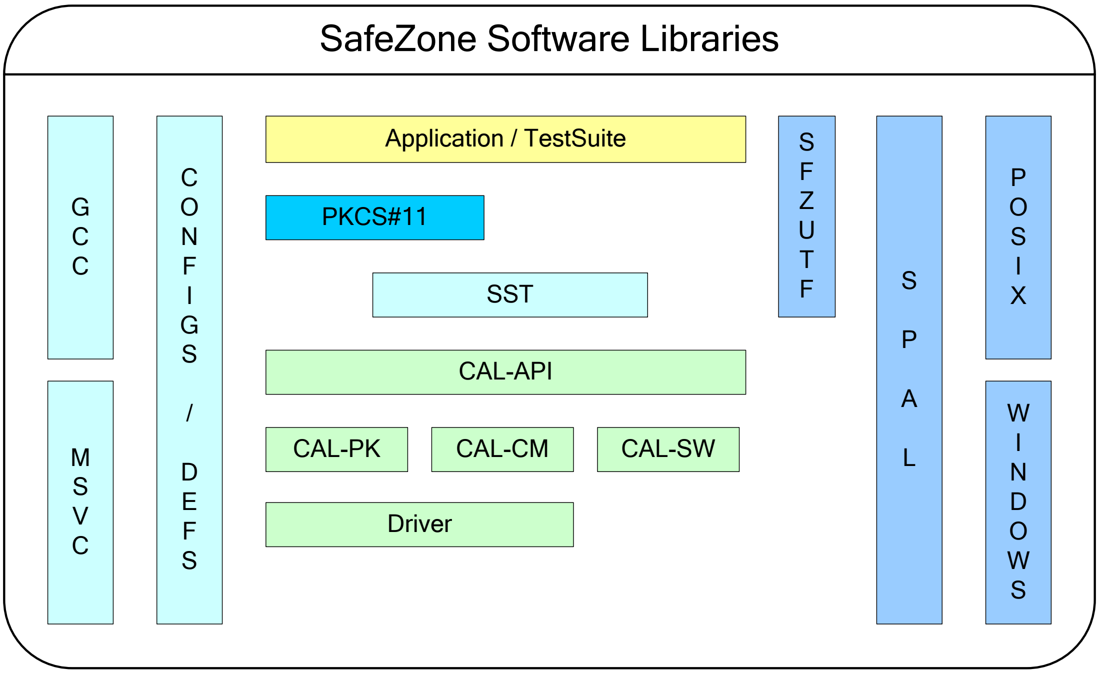

Figure 1 SafeZone/CAL Software Overview

This figure suggests a "hybrid" (see first paragraph of this section) configuration of CAL that implements some functions using underlying hardware (via CAL-PK, CAL-CM and the Driver module) and other functions in just software (CAL-SW).

CAL is currently provided as a static library. CAL can be built under both Linux (gcc) and Windows (MSVC).

CAL is declared in the following header files:

CAL_API/incl/sfzcryptoapi.h

This file `#include`‘s all header files described below (except `sfzcrypto_context.h`) so that the complete CAL API may be used.

CAL_API/incl/sfzcryptoapi_asset.h

This file provides all definitions and declarations associated with the Asset Store API.

CAL_API/incl/sfzcryptoapi_asym.h

This file provides the CAL APIs for doing asymmetric operations which include encrypt/decrypt, sign/verify, key generation, parameters generation etc.

CAL_API/incl/sfzcryptoapi_buffers.h

This file provides the data buffer type definitions used in the CAL API.

CAL_CONTEXT/incl/sfzcrypto_context.h

This file declares the `sfzcrypto_context_get()` function.

CAL_API/incl/sfzcryptoapi_enum.h

This file provides enumerated type definitions used by the CAL API.

CAL_API/incl/sfzcryptoapi_init.h

This file declares the `sfzcrypto_init()` function and the `SfzCryptoContext` type.

CAL_API/incl/sfzcryptoapi_misc.h

This file declares miscellaneous CAL APIs, like the ones for data copying, version info and feature reporting.

CAL_API/incl/sfzcryptoapi_rand.h

This file declares the CAL API related to random data generation.

CAL_API/incl/sfzcryptoapi_result.h

This file declares the `SfzCryptoStatus` type. Every function of the CAL API returns a value of this type.

CAL_API/incl/sfzcryptoapi_sym.h

This file declares CAL APIs related to hash, HMAC and symmetric key en/decrypt operations.

CAL_API/incl/sfzcryptoapi_cprm.h

This file declares CAL APIs related to CPRM.

CAL needs the header files in `Framework/PUBDEFS/incl` (part of the framework) to provide definitions of standard ISO C99 basic C types.

3 Initializing CAL

Before the CAL API can be used, the following API function must be used to initialize CAL:

```
SfzCryptoStatus
sfzcrypto_init(
    SfzCryptoContext * sfzcryptoctx_p);
```

The single argument is a pointer to the context for this application/thread and can be allocated through `sfzcrypto_context_get()` as shown in the code snippet below, before passing it to the `sfzcrypto_init()` API.

```
/* Initialize CAL */
status = sfzcrypto_init(sfzcrypto_context_get());
assert(status == SFZCRYPTO_SUCCESS);
```

In the example code in subsequent chapters, the `sfzcrypto_context_get()` function is frequently used to retrieve the context that was used to initialize CAL.

4 Hashing and Symmetric Key Cryptography with CAL

4.1 Key and Context structures, Algorithms and Modes

When performing a hash, CAL requires that a `SfzCryptoHashContext` structure is set up. This structure is used to hold the state of the hash operation, in particular when multiple calls to the hash API are used to hash a single message. See 4.2.1 for example code.

When performing symmetric cryptography, typically two CAL structures need to be initialized. The first is a `SfzCryptoCipherKey` structure that holds the symmetric key, either explicitly or by reference. How to obtain and use key values by reference is postponed until the section on Asset Store (see 4.3).

The second structure to initialize is a context structure like the `SfzCryptoCipherContext` structure for encrypt or decrypt operations, the `SfzCryptoHmacContext` structure for an HMAC operation or a `SfzCryptoCipherMacContext` for a CMAC or CBCMAC operation.

Depending on the selected mode (`SfzCryptoCipherContext.fbmode`) and algorithm (`SfzCryptoCipherKey.type`), some additional fields of the `SfzCryptoCipherContext` respectively `SfzCryptoCipherKey` structure need explicit initialization as shown in the following table.

Table 1 Symmetric Cipher Feedback Modes

Mode	Algorithm	Description	Extra fields to initialize
ECB	AES, DES, 3DES	Electronic Code Book	-
CBC	AES, DES, 3DES	Cipher Block Chaining	`iv`[1] field must be initialized with IV to use.
CTR	AES	32-bit Counter Mode	`iv`[1] field must be initialized with IV to use.
ICM	AES	16-bit Counter Mode	`iv`[1] field must be initialized with IV to use.
F8	AES-f8	AES-f8	`iv`[1] field must be initialized with AES IV to use. `f8_iv`[1] must be initialized with f8 IV to use. `f8_salt_key`[2] and `f8_salt_keyLen`[2] must initialized with a salt key to use.
CMAC	AES	CMAC Mode	-
CBCMAC	AES	CBCMAC Mode	-
S2V_CMAC	AES	S2V CMAC Mode	-
ARC4_STATELESS	ARCFOUR	Start until final.	-
ARC4_FINAL	ARCFOUR	Continue until final.	-
ARC4_INITIAL	ARCFOUR	Start not final.	-
ARC4_STATEFUL	ARCFOUR	Continue.	-

[1] The `iv` and `f8_iv` fields are part of the `SfzCryptoCipherContext` structure.

[2] The `f8_salt_key` and `f8_salt_keyLen` fields are part of the `SfzCryptoCipherKey` structure.

4.2 Basic Examples

The following paragraphs show some basic examples of how to use the CAL API not using functionality related to *Asset Store*. Most examples are based on test vectors available from NIST at [Toolkit Examples].

4.2.1 Hashing

For doing hash operations, CAL provides the following API.

```
SfzCryptoStatus
sfzcrypto_hash_data(
    SfzCryptoContext * const     sfzcryptoctx_p,
    SfzCryptoHashContext * const ctxt_p,
    uint8_t *                    data_p,
    uint32_t                     length,
    bool                         init,
    bool                         final);
```

When the `init` argument is true, a new hash is started, i.e. the starting digest value is set to the value specified by the applicable hash standard; otherwise, an ongoing hash operation is continued, i.e. `ctxt_p->digest` is used as the starting digest value. When the `final` argument is true, the hash is closed after processing the given data and the final hash value is returned via `ctxt_p->digest`; otherwise, the hash is left unfinished and `ctxt_p->digest` holds an intermediate hash result.

The following code illustrates how to use the CAL API to verify the first example given in [Toolkit Examples, SHA1].

```
static uint8_t msg[] = {'a', 'b', 'c'};
static uint8_t expected_sha1_digest[] = {
  0xA9,0x99,0x3E,0x36, 0x47,0x06,0x81,0x6A,
  0xBA,0x3E,0x25,0x71, 0x78,0x50,0xC2,0x6C,
  0x9C,0xD0,0xD8,0x9D };

SfzCryptoContext *sfzcryptoctx_p = sfzcrypto_context_get();
SfzCryptoHashContext hc;
SfzCryptoStatus ret;

/* Initialize the hash context. */
memset(&hc, 0x00, sizeof(SfzCryptoHashContext));
hc.algo = SFZCRYPTO_ALGO_HASH_SHA160;

/* Now call the hash API. */
ret = sfzcrypto_hash_data(sfzcryptoctx_p, &hc,
                          msg, sizeof(msg),
                          true, true);

assert(ret == SFZCRYPTO_SUCCESS);
assert(0 == memcmp(hc.digest, expected_sha1_digest,
                   sizeof(expected_sha1_digest));
```

The following example shows how to digest a sequence of one million 'a' characters in multiple chunks:

```
#define ONE_MILLION 1000000u
#ifndef MIN
#define MIN(a, b)  ((a) < (b) ? (a) : (b))
#endif
static uint8_t HashBuffer[1024];
static void
MultipartHashExample(void)
{
static uint8_t ExpectedDigest [] = {
  /* Expected SHA256 digest for 1 million 'a' characters. */
  0xcd, 0xc7, 0x6e, 0x5c, 0x99, 0x14, 0xfb, 0x92,
  0x81, 0xa1, 0xc7, 0xe2, 0x84, 0xd7, 0x3e, 0x67,
  0xf1, 0x80, 0x9a, 0x48, 0xa4, 0x97, 0x20, 0x0e,
  0x04, 0x6d, 0x39, 0xcc, 0xc7, 0x11, 0x2c, 0xd0 };
SfzCryptoContext *sfzcryptoctx_p = sfzcrypto_context_get();
SfzCryptoHashContext HashContext;
SfzCryptoStatus ret;
size_t nbytes_done, chunk_size = sizeof(HashBuffer);

c_memset(HashBuffer, 'a', sizeof(HashBuffer));
c_memset(&HashContext, 0, sizeof(HashContext));
HashContext.algo = SFZCRYPTO_ALGO_HASH_SHA256;
for (nbytes_done = 0;
     nbytes_done < ONE_MILLION;
     nbytes_done += chunk_size)
{
  bool init, finish;

  chunk_size = MIN(chunk_size, ONE_MILLION - nbytes_done);
  init = (nbytes_done == 0);
  finish = (nbytes_done + chunk_size == ONE_MILLION);
  ret = sfzcrypto_hash_data(sfzcryptoctx_p, &HashContext,
                            HashBuffer, chunk_size,
                            init, finish);
  assert (ret == SFZCRYPTO_SUCCESS);
}
assert (c_memcmp(HashContext.digest, ExpectedDigest,
                 sizeof(ExpectedDigest)) == 0);
}
```

4.2.2 Random Key Generation

CAL provides no dedicated API to generate random symmetric keys. The following example shows how to use the `sfzcrypto_rand_data()` API for this.

```
SfzCryptoStatus
sfzcrypto_rand_data(
    SfzCryptoContext * const sfzcryptoctx_p,
    uint32_t                 rand_num_size_bytes,
    uint8_t *                rand_num_p);
```

The following code shows how to generate a random 256-bit AES key.

```
SfzCryptoContext *sfzcryptoctx_p = sfzcrypto_context_get();
SfzCryptoCipherKey sKey;

/* Zero out everything */
memset(&sKey, 0x0, sizeof (SfzCryptoCipherKey));

/* Setup key length and type */
sKey.type = SFZCRYPTO_KEY_AES;
sKey.length = 32;
sKey.asset_id = SFZCRYPTO_ASSETID_INVALID;

/* Setup the key bits with random data */
status = sfzcrypto_rand_data(sfzcryptoctx_p,
                             sKey.length,
                             sKey.key);

assert(status == SFZCRYPTO_SUCCESS);
```

4.2.3 Symmetric Encryption

For doing encryption or decryption with symmetric keys, CAL provides the following API.

```
SfzCryptoStatus
sfzcrypto_symm_crypt(
    SfzCryptoContext * const       sfzcryptoctx_p,
    SfzCryptoCipherContext * const ctxt_p,
    SfzCryptoCipherKey * const     key_p,
    uint8_t *                      src_p,
    uint32_t                       src_len,
    uint8_t *                      dst_p,
    uint32_t * const               dst_len_p);
```

The following code illustrates how to implement a simple AES-CBC mode encryption, given as the first example in [Toolkit Examples, AES-CBC], using the CAL API:

```
SfzCryptoContext *sfzcryptoctx_p = sfzcrypto_context_get();
SfzCryptoCipherContext ctxt;
SfzCryptoCipherKey sKey;
SfzCryptoStatus ret;

static uint8_t plaintext[64] = {
  0x6B,0xC1,0xBE,0xE2, 0x2E,0x40,0x9F,0x96,
  0xE9,0x3D,0x7E,0x11, 0x73,0x93,0x17,0x2A,
  0xAE,0x2D,0x8A,0x57, 0x1E,0x03,0xAC,0x9C,
  0x9E,0xB7,0x6F,0xAC, 0x45,0xAF,0x8E,0x51,
  0x30,0xC8,0x1C,0x46, 0xA3,0x5C,0xE4,0x11,
  0xE5,0xFB,0xC1,0x19, 0x1A,0x0A,0x52,0xEF,
  0xF6,0x9F,0x24,0x45, 0xDF,0x4F,0x9B,0x17,
  0xAD,0x2B,0x41,0x7B, 0xE6,0x6C,0x37,0x10 };

static uint8_t iv[16] = {
  0x00,0x01,0x02,0x03, 0x04,0x05,0x06,0x07,
  0x08,0x09,0x0A,0x0B, 0x0C,0x0D,0x0E,0x0F };

static uint8_t key[16] = {
  0x2B,0x7E,0x15,0x16, 0x28,0xAE,0xD2,0xA6,
  0xAB,0xF7,0x15,0x88, 0x09,0xCF,0x4F,0x3C };

static uint8_t expected_ciphertext[64] = {
  0x76,0x49,0xAB,0xAC, 0x81,0x19,0xB2,0x46,
  0xCE,0xE9,0x8E,0x9B, 0x12,0xE9,0x19,0x7D,
  0x50,0x86,0xCB,0x9B, 0x50,0x72,0x19,0xEE,
  0x95,0xDB,0x11,0x3A, 0x91,0x76,0x78,0xB2,
  0x73,0xBE,0xD6,0xB8, 0xE3,0xC1,0x74,0x3B,
  0x71,0x16,0xE6,0x9E, 0x22,0x22,0x95,0x16,
  0x3F,0xF1,0xCA,0xA1, 0x68,0x1F,0xAC,0x09,
  0x12,0x0E,0xCA,0x30, 0x75,0x86,0xE1,0xA7 };

uint8_t ciphertext[64];
uint32_t ciphertext_len = sizeof(ciphertext);

/* Zero out everything. */
memset(&ctxt, 0x0, sizeof(SfzCryptoCipherContext));
memset(&sKey, 0x0, sizeof(SfzCryptoCipherKey));
```

```
/* Instantiate a key */
sKey.type = SFZCRYPTO_KEY_AES;
sKey.length = sizeof(key);
sKey.asset_id = SFZCRYPTO_ASSETID_INVALID;
memcpy(sKey.key, key, sizeof(key));

/* Set up the cipher context */
ctxt.fbmode = SFZCRYPTO_MODE_CBC;
c_memcpy(ctxt.iv, iv, sizeof (iv));
ctxt.iv_asset_id = SFZCRYPTO_ASSETID_INVALID;
ctxt.iv_loc = SFZ_IN_CONTEXT;

/* Request the encrypt operation. */
ret = sfzcrypto_symm_crypt(sfzcryptoctx_p,
                           &ctxt, &sKey,
                           plaintext, sizeof(plaintext),
                           ciphertext, &ciphertext_len,
                           SFZ_ENCRYPT);

assert(ret == SFZCRYPTO_SUCCESS);
assert(0 == memcmp(ciphertext, expected_ciphertext,
                   sizeof(expected_ciphertext)));
```

From the perspective of the CAL API, decryption works just like encryption, the only difference is that `SFZ_ENCRYPT` is replaced with `SFZ_DECRYPT`.

4.2.4 Calculating an HMAC

The CAL API for calculating an HMAC is as follows:

```
SfzCryptoStatus
sfzcrypto_hmac_data(
    SfzCryptoContext * const     sfzcryptoctx_p,
    SfzCryptoHmacContext * const ctxt_p,
    SfzCryptoCipherKey * const   key_p,
    uint8_t *                    data_p,
    uint32_t                     length,
    bool                         init,
    bool                         final);
```

The `SfzCryptoCipherKey` structure is filled with the HMAC key.

The next code illustrates how to use the CAL API to verify the first example given in [Toolkit Examples, HMAC-SHA256].

```
static uint8_t msg[] = { "Sample message for keylen=blocklen" };
static uint8_t hmac_key_bytes[] = {
  0x00,0x01,0x02,0x03, 0x04,0x05,0x06,0x07,
  0x08,0x09,0x0A,0x0B, 0x0C,0x0D,0x0E,0x0F,
  0x10,0x11,0x12,0x13, 0x14,0x15,0x16,0x17,
  0x18,0x19,0x1A,0x1B, 0x1C,0x1D,0x1E,0x1F,
  0x20,0x21,0x22,0x23, 0x24,0x25,0x26,0x27,
  0x28,0x29,0x2A,0x2B, 0x2C,0x2D,0x2E,0x2F,
  0x30,0x31,0x32,0x33, 0x34,0x35,0x36,0x37,
  0x38,0x39,0x3A,0x3B, 0x3C,0x3D,0x3E,0x3F };
static uint8_t expected_sha256_hmac[] = {
  0x8B,0xB9,0xA1,0xDB, 0x98,0x06,0xF2,0x0D,
  0xF7,0xF7,0x7B,0x82, 0x13,0x8C,0x79,0x14,
  0xD1,0x74,0xD5,0x9E, 0x13,0xDC,0x4D,0x01,
  0x69,0xC9,0x05,0x7B, 0x13,0x3E,0x1D,0x62 };

SfzCryptoContext *sfzcryptoctx_p = sfzcrypto_context_get();
SfzCryptoHmacContext hmacCtx;
SfzCryptoCipherKey hmacKey;
uint32_t msg_len = 34;
SfzCryptoStatus status;

/* Zeroize everything */
memset(&hmacCtx, 0x00, sizeof (SfzCryptoHmacContext));
memset(&hmacKey, 0x00, sizeof (SfzCryptoCipherKey));

hmacCtx.hashCtx.algo = SFZCRYPTO_ALGO_HASH_SHA256;
hmacCtx.mac_asset_id = SFZCRYPTO_ASSETID_INVALID;
hmacCtx.mac_loc = SFZ_IN_CONTEXT;

hmacKey.type = SFZCRYPTO_KEY_HMAC;
hmacKey.length = sizeof(hmac_key_bytes);
hmacKey.asset_id = SFZCRYPTO_ASSETID_INVALID;
memcpy(hmacKey.key, hmac_key_bytes, sizeof(hmac_key_bytes));
```

```
status = sfzcrypto_hmac_data(sfzcryptoctx_p,
                             &hmacCtx,
                             &hmacKey,
                             msg, msg_len,
                             true, true);

assert(status == SFZCRYPTO_SUCCESS);
assert(0 == memcmp(hmacCtx.hashCtx.digest, expected_sha256_hmac,
                   sizeof(expected_sha256_hmac)));
```

4.3 *Asset Store Examples*

This part of the manual shows how the *Asset Store* can be used. An introduction to the *Asset Store* can be found in the *CAL Reference Manual* [1].

4.3.1 Key Derivation

The CAL API for loading the content of an asset through key derivation is as follows:

```
SfzCryptoStatus
sfzcrypto_asset_derive(
    SfzCryptoContext * const sfzcryptoctx_p,
    SfzCryptoAssetId         TargetAssetId,
    SfzCryptoTrustedAssetId  KdkAssetId,
    SfzCryptoOctetsIn *      Label_p,
    SfzCryptoSize            LabelLen);
```

The following code shows how a KEK asset can be allocated and loaded through key derivation.

```
static uint8_t KekLabel[] = {
  'K', 'E', 'K', ' ', 'L', 'A', 'B', 'E', 'L' };

SfzCryptoContext *sfzcryptoctx_p = sfzcrypto_context_get();
SfzCryptoAssetId KekAssetId;
SfzCryptoStatus status;

/* Allocate a KEK key asset and setup its contents through
   derivation from the root key. */
status = sfzcrypto_asset_alloc(sfzcryptoctx_p,
                               SFZCRYPTO_POLICY_SECURE_WRAP |
                               SFZCRYPTO_POLICY_SECURE_UNWRAP,
                               64,
                               &KekAssetId);
assert(status == SFZCRYPTO_SUCCESS);

status = sfzcrypto_asset_derive(sfzcryptoctx_p,
                                KekAssetId,
                                sfzcrypto_asset_get_root_key(),
                                KekLabel, sizeof(KekLabel));
assert(status == SFZCRYPTO_SUCCESS);
```

4.3.2 Key Generation

The CAL API for loading the content of an asset with random data is as follows:

```
SfzCryptoStatus
sfzcrypto_asset_gen_key_and_wrap(
    SfzCryptoContext * const sfzcryptoctx_p,
    SfzCryptoAssetId         TargetAssetId,
    SfzCryptoAssetSize       AssetSize,
    SfzCryptoTrustedAssetId  KekAssetId,
    SfzCryptoOctetsIn *      AdditionalData_p,
    SfzCryptoSize            AdditionalDataSize,
    SfzCryptoOctetsOut *     KeyBlob_p,
    SfzCryptoSize * const    KeyBlobSize_p);
```

Note that CAL also provides a related API that loads the content with random data but does not export the key in the form of a key blob, see `sfzcrypto_asset_gen_key()`.

The following code shows how a random 3DES key asset can be allocated and loaded.

```
SfzCryptoContext *sfzcryptoctx_p = sfzcrypto_context_get();
SfzCryptoCipherKey tdesKey;
SfzCryptoStatus status;

memset(&tdesKey, 0x00, sizeof (SfzCryptoCipherKey));
tdesKey.type = SFZCRYPTO_KEY_TRIPLE_DES;
tdesKey.length = 3*8;

/* Allocate the 3DES key asset */
status = sfzcrypto_asset_alloc(
                       sfzcryptoctx_p,
                       SFZCRYPTO_POLICY_ALGO_CIPHER_TRIPLE_DES |
                       SFZCRYPTO_POLICY_FUNCTION_ENCRYPT |
                       SFZCRYPTO_POLICY_FUNCTION_DECRYPT,
                       tdesKey.length,
                       &tdesKey.asset_id);
assert(status == SFZCRYPTO_SUCCESS);

/* Assign a random value to the 3DES key */
status = sfzcrypto_asset_gen_key(sfzcryptoctx_p,
                                 tdesKey.asset_id,
                                 tdesKey.length);
assert(status == SFZCRYPTO_SUCCESS);
```

4.3.3 Key Import

The CAL API for loading the content of an asset through the import of a key blob is as follows:

```
SfzCryptoStatus
sfzcrypto_asset_import(
    SfzCryptoContext * const sfzcryptoctx_p,
    SfzCryptoAssetId         TargetAssetId,
    SfzCryptoTrustedAssetId  KekAssetId,
    SfzCryptoOctetsIn *      AdditionalData_p,
    SfzCryptoSize            AdditionalDataSize,
    SfzCryptoOctetsIn *      KeyBlob_p,
    SfzCryptoSize            KeyBlobSize);
```

The following code illustrates the use of this API. It is assumed here that a key blob is available that contains a 128-bit AES key intended for calculating a CMAC.

```
static uint8_t AssociatedData[] = {
  0xDA,0x7A,0x00,0x01, 0xDA,0x7A,0x00,0x02,
  0xDA,0x7A,0x00,0x03, 0xDA,0x7A,0x00,0x04 };

SfzCryptoContext *sfzcryptoctx_p = sfzcrypto_context_get();
SfzCryptoOctetsIn * WrappedAesCmacKey_p = ...; /* some ptr value */
SfzCryptoSize WrappedAesCmacKeySize = ...;       /* key blob size*/
SfzCryptoAssetId KekAssetId = ...;    /* reference to KEK to use */
SfzCryptoCipherKey cmacKey;
SfzCryptoStatus status;

/* Initialize key structure */
memset(&cmacKey, 0x00, sizeof (SfzCryptoCipherKey));
cmacKey.type = SFZCRYPTO_KEY_AES;
cmacKey.length = 128/8;

/* Allocate the 128-bit AES key asset */
status = sfzcrypto_asset_alloc(sfzcryptoctx_p,
                               SFZCRYPTO_POLICY_ALGO_CIPHER_AES |
                               SFZCRYPTO_POLICY_FUNCTION_MAC,
                               cmacKey.length,
                               &cmacKey.asset_id);
assert(status == SFZCRYPTO_SUCCESS);

/* Setup the content of the CMAC key asset from the given key blob,
   assuming that KekAssetId already refers to the KEK necessary for
   decrypting the key blob, as shown in 4.3.1. for example. */
status = sfzcrypto_asset_import(sfzcryptoctx_p,
                                cmacKey.asset_id,
                                KekAssetId,
                                AssociatedData, sizeof(AssociatedData),
                                WrappedAesCmacKey_p,
                                WrappedAesCmacKeySize);
assert(status == SFZCRYPTO_SUCCESS);
```

4.3.4 Using a Key Asset

CAL provides the following API for calculating a CMAC:

```
SfzCryptoStatus
sfzcrypto_cipher_mac_data (
    SfzCryptoContext * const          sfzcryptoctx_p,
    SfzCryptoCipherMacContext * const ctxt_p,
    SfzCryptoCipherKey * const        key_p,
    uint8_t *                         data_p,
    uint32_t                          length,
    bool                              init,
    bool                              final);
```

The following code illustrates how to use the CMAC API in combination with the CMAC key asset as imported in the previous example. The CMAC key value is assumed to be equal to the one used in CMAC-AES128 Example #2 from [Toolkit Examples, AES-CMAC].

```
static uint8_t msg[] = {
  0x6B,0xC1,0xBE,0xE2, 0x2E,0x40,0x9F,0x96,
  0xE9,0x3D,0x7E,0x11, 0x73,0x93,0x17,0x2A };
static uint8_t expected_aes_cmac[] = {
  0x07,0x0A,0x16,0xB4, 0x6B,0x4D,0x41,0x44,
  0xF7,0x9B,0xDD,0x9D, 0xD0,0x4A,0x28,0x7C };

SfzCryptoContext *sfzcryptoctx_p = sfzcrypto_context_get();
SfzCryptoCipherMacContext cmacCtx;
SfzCryptoCipherKey * cmacKey_p;
SfzCryptoStatus status;

/* Setup CMAC context */
memset(&cmacCtx, 0x00, sizeof (SfzCryptoCipherMacContext));
cmacCtx.fbmode = SFZCRYPTO_MODE_CMAC;
cmacCtx.iv_asset_id = SFZCRYPTO_ASSETID_INVALID;
cmacCtx.iv_loc = SFZ_IN_CONTEXT;

/* Assume cmacKey_p points to the CMAC key that was setup
   as shown in the previous example... */

/* Calculate CMAC and verify result */
status = sfzcrypto_cipher_mac_data(sfzcryptoctx_p,
                                   &cmacCtx,
                                   cmacKey_p,
                                   msg, sizeof(msg),
                                   true, true);
assert(status == SFZCRYPTO_SUCCESS);
assert(0 == memcmp(cmacCtx.iv, expected_aes_cmac, 16));

/* Free CMAC key asset */
status = sfzcrypto_asset_free(sfzcryptoctx_p,
                              cmacKey_p->asset_id);
assert(status == SFZCRYPTO_SUCCESS);
```

4.4 Camellia Examples

The following example code shows how CAL is used to verify some publicly available Camellia test vectors. The test cases are taken from RFC-3713 and RFC-5528 (see [RFC-3713] respectively [RFC-5528].

Note that the RFC-5528 example shows a CTR mode encrypt of 36 bytes of data. Since CAL's `sfzcrypto_symm_crypt` API rejects non-block-sized data, even for CTR mode, 12 pad bytes are appended to the 36-byte input.

```
static void
CamelliaCipherExample(void)
{
    /* ECB mode test vectors, from "RFC-3713", first test case */
    static uint8_t CAM_ECB_KEY128[] = {
      0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef,
      0xfe, 0xdc, 0xba, 0x98, 0x76, 0x54, 0x32, 0x10 };
    /* This key value is also used as PTX data. */
    static uint8_t CAM_ECB_KEY128_CTX[] = {
      0x67, 0x67, 0x31, 0x38, 0x54, 0x96, 0x69, 0x73,
      0x08, 0x57, 0x06, 0x56, 0x48, 0xea, 0xbe, 0x43 };
    /* CTR mode test vectors, from "RFC-5528", last = TV#9 test case */
    static uint8_t CAM_CTR_KEY256[] = {
      0xFF, 0x7A, 0x61, 0x7C, 0xE6, 0x91, 0x48, 0xE4,
      0xF1, 0x72, 0x6E, 0x2F, 0x43, 0x58, 0x1D, 0xE2,
      0xAA, 0x62, 0xD9, 0xF8, 0x05, 0x53, 0x2E, 0xDF,
      0xF1, 0xEE, 0xD6, 0x87, 0xFB, 0x54, 0x15, 0x3D };
    static uint8_t CAM_CTR_KEY256_PTX[] = {
      0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
      0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
      0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
      0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,
      0x20, 0x21, 0x22, 0x23, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
    /* Note the padding bytes to make the PTX an integral #of blocks. */
    static uint8_t CAM_CTR_KEY256_CTX[] = {
      0xA4, 0xDA, 0x23, 0xFC, 0xE6, 0xA5, 0xFF, 0xAA,
      0x6D, 0x64, 0xAE, 0x9A, 0x06, 0x52, 0xA4, 0x2C,
      0xD1, 0x61, 0xA3, 0x4B, 0x65, 0xF9, 0x67, 0x9F,
      0x75, 0xC0, 0x1F, 0x10, 0x1F, 0x71, 0x27, 0x6F,
      0x15, 0xEF, 0x0D, 0x8D };
    static uint8_t CAM_CTR_KEY256_IV[] = {
      0x00, 0x1C, 0xC5, 0xB7, 0x51, 0xA5, 0x1D, 0x70,
      0xA1, 0xC1, 0x11, 0x48, 0x00, 0x00, 0x00, 0x01 };

    SfzCryptoContext *sfzcryptoctx_p = sfzcrypto_context_get();
    SfzCryptoCipherContext CiphContext;
    SfzCryptoCipherKey CiphKey;

    uint8_t OutputData[4*16];
    size_t OutputDataLen;
    SfzCryptoStatus ret;

    /* Test Camellia ECB encrypt with the "RFC-3713" test vectors */
    c_memset(&CiphContext, 0, sizeof(CiphContext));
    c_memset(&CiphKey, 0, sizeof(CiphKey));
    CiphContext.fbmode = SFZCRYPTO_MODE_ECB;
```

```
    SFZCRYPTO_CIPHER_KEY_INIT(&CiphKey,
                              SFZCRYPTO_KEY_CAMELLIA,
                              CAM_ECB_KEY128,
                              sizeof(CAM_ECB_KEY128));
    OutputDataLen = sizeof(OutputData);
    /* Note that the OutputData buffer is over-sized. */
    ret = sfzcrypto_symm_crypt(sfzcryptoctx_p,
                               &CiphContext, &CiphKey,
                               CAM_ECB_KEY128, sizeof(CAM_ECB_KEY128),
                               OutputData, &OutputDataLen,
                               SFZ_ENCRYPT);
    assert (ret == SFZCRYPTO_SUCCESS);
    assert (OutputDataLen == sizeof(CAM_ECB_KEY128_CTX));
    /* Note that OutputDataLen was adjusted. */
    assert (c_memcmp(OutputData, CAM_ECB_KEY128_CTX, OutputDataLen) == 0);

    /* Test Camellia ECB decrypt with the "RFC-3713" test vectors */
    c_memset(&CiphContext, 0, sizeof(CiphContext));
    CiphContext.fbmode = SFZCRYPTO_MODE_ECB;
    OutputDataLen = sizeof(OutputData);
    ret = sfzcrypto_symm_crypt(sfzcryptoctx_p,
                               &CiphContext, &CiphKey,
                               CAM_ECB_KEY128_CTX,
                               sizeof(CAM_ECB_KEY128_CTX),
                               OutputData, &OutputDataLen,
                               SFZ_DECRYPT);
    assert (ret == SFZCRYPTO_SUCCESS);
    assert (OutputDataLen == sizeof(CAM_ECB_KEY128));
    assert (c_memcmp(OutputData, CAM_ECB_KEY128, OutputDataLen) == 0);

    /* Test Camellia CTR encrypt with the "RFC-5528" test vectors */
    c_memset(&CiphContext, 0, sizeof(CiphContext));
    c_memset(&CiphKey, 0, sizeof(CiphKey));
    CiphContext.fbmode = SFZCRYPTO_MODE_CTR;
    c_memcpy(CiphContext.iv, CAM_CTR_KEY256_IV, sizeof(CAM_CTR_KEY256_IV));
    SFZCRYPTO_CIPHER_KEY_INIT(&CiphKey, SFZCRYPTO_KEY_CAMELLIA,
                              CAM_CTR_KEY256, sizeof(CAM_CTR_KEY256));

    OutputDataLen = sizeof(OutputData);
    ret = sfzcrypto_symm_crypt(sfzcryptoctx_p,
                               &CiphContext, &CiphKey,
                               CAM_CTR_KEY256_PTX,
                               sizeof(CAM_CTR_KEY256_PTX),
                               OutputData, &OutputDataLen,
                               SFZ_ENCRYPT);
    assert (ret == SFZCRYPTO_SUCCESS);
    assert (OutputDataLen == sizeof(CAM_CTR_KEY256_PTX));
    assert (c_memcmp(OutputData, CAM_CTR_KEY256_CTX,
                     sizeof(CAM_CTR_KEY256_CTX)) == 0);
    /* Verify that the Counter was incremented by 3. */
    assert (CiphContext.iv[15] == 0x04);
    assert (c_memcmp(CiphContext.iv, CAM_CTR_KEY256_IV,
                     sizeof(CAM_CTR_KEY256_IV) - 1) == 0);
}
```

4.5 MULTI2 Examples

The following example shows how CAL is used to verify the publicly available test vector for the MULTI2 cipher, see [MULTI2]. This appears to be the only publicized test vector for MULTI2.

Although not shown in the example, the MULTI2 *System Key* can also be stored in an asset. In that case, use the second argument of the `sfzcrypto_multi2_configure` API to pass the applicable *AssetId* value and set the third argument to NULL.

Currently, CAL's design assumes that the MULTI2 cipher is configured once. Note that the `SfzCryptoCipherKey` structure does not allow a 64-bit MULTI2 *Data Key* to be associated with a specific *System Key*, nor with a specific *NRounds* value. Consequently, all users of the MULTI2 cipher are supposed to share the same *System Key* and *NRounds* setting.

```
static void
Multi2CipherExample(void)
{
    /* ECB mode test vectors, taken from "0009.pdf" */
    static uint8_t M2_SYSTEM_KEY [32] = {0};
    static uint8_t M2_ECB_KEY64 [] = {
      0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef };
    static uint8_t M2_ECB_PTX [] = {
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 };
    static uint8_t M2_ECB_CTX [] = {
      0xF8, 0x94, 0x40, 0x84, 0x5E, 0x11, 0xCF, 0x89 };

    SfzCryptoContext *sfzcryptoctx_p = sfzcrypto_context_get();
    SfzCryptoCipherContext CiphContext;
    SfzCryptoCipherKey CiphKey;

    uint8_t OutputData[4*8];
    size_t OutputDataLen;
    SfzCryptoStatus ret;

    /* Configure the MULTI2 cipher: SystemKey is all-zero, NRounds=128. */
    ret = sfzcrypto_multi2_configure(128,
                                     SFZCRYPTO_ASSETID_INVALID,
                                     M2_SYSTEM_KEY);
    assert (ret == SFZCRYPTO_SUCCESS);

    /* Test MULTI2 ECB encrypt with the "0009.pdf" test vectors */
    c_memset(&CiphContext, 0, sizeof(CiphContext));
    c_memset(&CiphKey, 0, sizeof(CiphKey));
    CiphContext.fbmode = SFZCRYPTO_MODE_ECB;
    SFZCRYPTO_CIPHER_KEY_INIT(&CiphKey,
                              SFZCRYPTO_KEY_MULTI2,
                              M2_ECB_KEY64,
                              sizeof(M2_ECB_KEY64));
    OutputDataLen = sizeof(OutputData);
    ret = sfzcrypto_symm_crypt(sfzcryptoctx_p,
                               &CiphContext, &CiphKey,
                               M2_ECB_PTX, sizeof(M2_ECB_PTX),
                               OutputData, &OutputDataLen,
                               SFZ_ENCRYPT);
    assert (ret == SFZCRYPTO_SUCCESS);
    assert (OutputDataLen == sizeof(M2_ECB_CTX));
    assert (c_memcmp(OutputData, M2_ECB_CTX, OutputDataLen) == 0);
```

```
    /* Test MULTI2 ECB decrypt with the "0009.pdf" test vectors */
    c_memset(&CiphContext, 0, sizeof(CiphContext));
    CiphContext.fbmode = SFZCRYPTO_MODE_ECB;
    OutputDataLen = sizeof(OutputData);
    ret = sfzcrypto_symm_crypt(sfzcryptoctx_p,
                               &CiphContext, &CiphKey,
                               M2_ECB_CTX, sizeof(M2_ECB_CTX),
                               OutputData, &OutputDataLen,
                               SFZ_DECRYPT);
    assert (ret == SFZCRYPTO_SUCCESS);
    assert (OutputDataLen == sizeof(M2_ECB_PTX));
    assert (c_memcmp(OutputData, M2_ECB_PTX, OutputDataLen) == 0);
}
```

5 Asymmetric Key Cryptography with CAL

5.1 Generating an RSA Key

For generating an RSA key pair or any other asymmetric key pairs, two `SfzCryptoAsymKey` structures must be filled. One structure holds the public key, the other holds the private key. Each structure contains only pointers to arrays that store cryptographic data, not the actual arrays themselves. Therefore, according to the desired key type, different fields of the structure must be filled in with (pointers to) data buffers large enough to hold the element value.

For an RSA public key the fields that must be filled in are:

Field	Contents
`algo_type`	`SFZCRYPTO_ALGO_ASYMM_RSA_RAW`
`cmd_type`	`SFZCRYPTO_CMD_RSA_ENCRYPT`
`Key.rsaPubKey.modulus.p_num`	pointer to a buffer[1] for the modulus
`Key.rsaPubKey.pubexp.p_num`	pointer to a buffer[1] for the public exponent

[1] the size of the buffer must be modBitsLen / 8 bytes (rounded up).

For an RSA private key the fields are:

Field	Contents
`Key.rsaPrivKey.modulus.p_num`	pointer to a buffer[1] for the modulus
`Key.rsaPrivKey.pubexp.p_num`	pointer to a buffer[1] for the public exponent
`Key.rsaPrivKey.privexp.p_num`	pointer to a buffer[1] for the private exponent (d)
`Key.rsaPrivKey.primeP.p_num`	pointer to a buffer[2] for prime p
`Key.rsaPrivKey.primeQ.p_num`	pointer to a buffer[2] for prime q
`Key.rsaPrivKey.dmodP.p_num`	pointer to a buffer[2] for d mod (p-1)
`Key.rsaPrivKey.dmodQ.p_num`	pointer to a buffer[2] for d mod (q-1)
`Key.rsaPrivKey.cofQinv.p_num`	pointer to a buffer[2] for the inverse of q (mod p)

[1] the size of the buffer must be modBitsLen / 8 bytes (rounded up).

[2] the size of the buffer must be (modBitsLen / 2) / 8 bytes (rounded up).

The example below illustrates how to generate an RSA key pair:

```
static SfzCryptoAsymKey rsa_pub_key;
static SfzCryptoAsymKey rsa_priv_key;
static uint8_t rsa_keypair_mem[(6+2)*SFZCRYPTO_RSA_BYTES];

SfzCryptoContext *p_sfzcryptoctx = sfzcrypto_context_get();
SfzCryptoStatus  status;

uint32_t modBitsLen;
uint8_t * u8_p = rsa_keypair_mem;

/* Assign memory to the BigInt components of the RSA key (private & public part)
*/
rsa_priv_key.Key.rsaPrivKey.modulus.p_num = u8_p;
u8_p += SFZCRYPTO_RSA_BYTES;
rsa_priv_key.Key.rsaPrivKey.pubexp.p_num = u8_p;
u8_p += SFZCRYPTO_RSA_BYTES;
rsa_priv_key.Key.rsaPrivKey.privexp.p_num = u8_p;
u8_p += SFZCRYPTO_RSA_BYTES;
rsa_priv_key.Key.rsaPrivKey.primeP.p_num = u8_p;
u8_p += SFZCRYPTO_RSA_BYTES / 2;
```

```
rsa_priv_key.Key.rsaPrivKey.primeQ.p_num = u8_p;
u8_p += SFZCRYPTO_RSA_BYTES / 2;
rsa_priv_key.Key.rsaPrivKey.dmodP.p_num = u8_p;
u8_p += SFZCRYPTO_RSA_BYTES / 2;
rsa_priv_key.Key.rsaPrivKey.dmodQ.p_num = u8_p;
u8_p += SFZCRYPTO_RSA_BYTES / 2;
rsa_priv_key.Key.rsaPrivKey.cofQinv.p_num = u8_p;
u8_p += SFZCRYPTO_RSA_BYTES / 2;
rsa_pub_key.Key.rsaPubKey.modulus.p_num = u8_p;
u8_p += SFZCRYPTO_RSA_BYTES;
rsa_pub_key.Key.rsaPubKey.pubexp.p_num = u8_p;

/* Request the generation of a 1024-bit RSA key pair */
modBitsLen = 1024;
rsa_pub_key.mod_bits = modBitsLen;
rsa_priv_key.mod_bits = modBitsLen;
status = sfzcrypto_gen_rsa_key_pair(p_sfzcryptoctx,
                                    &rsa_pub_key,
                                    &rsa_priv_key,
                                    modBitsLen);
assert(status == SFZCRYPTO_SUCCESS);
```

Key generation functions also exist for ECDSA and DSA keys. They are used similarly to RSA key generation.

5.2 Performing asymmetric encryption and decryption

The RSA key pair generated in Chapter 5.1 can be used to encrypt/decrypt as shown below. Note that `RsaPubKey_p` and `RsaPrvKey_p` are assumed to point to `rsa_pub_key` respectively `rsa_priv_key` from the previous example:

```
static uint8_t plaintext_ibuf[4*4] = {
  0x12,0x34,0x56,0x78, 0x12,0x34,0x56,0x78,
  0x12,0x34,0x56,0x78, 0x12,0x34,0x56,0x78};
static uint8_t plaintext_obuf[SFZCRYPTO_RSA_WORDS*4];
static uint8_t ciphertext_buf[SFZCRYPTO_RSA_WORDS*4];

SfzCryptoContext *p_sfzcryptoctx = sfzcrypto_context_get();
SfzCryptoBigInt plaintext_in;
SfzCryptoBigInt plaintext_out;
SfzCryptoBigInt ciphertext;
SfzCryptoStatus status;
uint8_t * u8_p;

/* Setup the big integers used. */
plaintext_in.p_num = plaintext_ibuf;
plaintext_in.byteLen = sizeof(plaintext_ibuf);
plaintext_out.p_num = plaintext_obuf;
plaintext_out.byteLen = sizeof(plaintext_obuf);
ciphertext.p_num = ciphertext_buf;
ciphertext.byteLen = sizeof(ciphertext_buf);

/* Select the RSA-RAW encrypt algorithm and do the encrypt. */
RsaPubKey_p->algo_type = SFZCRYPTO_ALGO_ASYMM_RSA_RAW;
RsaPubKey_p->cmd_type = SFZCRYPTO_CMD_RSA_ENCRYPT;
status = sfzcrypto_rsa_encrypt(p_sfzcryptoctx,
```

```
                               RsaPubKey_p,
                               &plaintext_in,
                               &ciphertext);
assert(status == SFZCRYPTO_SUCCESS);

/* Select the RSA-RAW decrypt algorithm and do the decrypt. */
RsaPrvKey_p->algo_type = SFZCRYPTO_ALGO_ASYMM_RSA_RAW;
RsaPrvKey_p->cmd_type = SFZCRYPTO_CMD_RSA_DECRYPT;
status = sfzcrypto_rsa_decrypt(p_sfzcryptoctx,
                               RsaPrvKey_p,
                               &ciphertext,
                               &plaintext_out);
assert(status == SFZCRYPTO_SUCCESS);

/* Verify the decrypt result */
assert(plaintext_out.byteLen == (RsaPrvKey_p->mod_bits + 7) / 8);
u8_p = plaintext_obuf;
while ((u8_p - plaintext_obuf < sizeof(plaintext_obuf)) && (*u8_p == 0)) {
  /* Skip leading zeroes in plaintext_out */
  u8_p++;
}
assert((u8_p - plaintext_obuf)+sizeof(plaintext_ibuf) == plaintext_out.byteLen);
assert(0 == memcmp(plaintext_ibuf, u8_p, sizeof(plaintext_ibuf)));
```

5.3 Negotiating a secret with another party using the Diffie-Hellman algorithm

The following example illustrates negotiating a shared secret using Diffie-Hellman. The example is of course rather artificial. Normally, parties A and B are located on different machines that are connected via some network so that they can exchange messages:

```
static uint8_t DH_PARAMS[] = {
/* 512-bit prime p */
0x8c,0xa8,0xd2,0xc0,0xc0,0x41,0x01,0xb8,0xbd,0x0e,0x07,0x9d,0xd8,0xba,0xe6,0x89,
0x0c,0x60,0xec,0xfe,0xe4,0xbe,0x26,0x00,0xc4,0x3d,0xe5,0x40,0x19,0x06,0xf2,0x0a,
0x9c,0x25,0x87,0x0a,0x3c,0x25,0x44,0xc3,0x3b,0xa6,0x08,0x0e,0xaa,0x1e,0x44,0x2f,
0x0c,0x14,0x8f,0x20,0xc3,0xcc,0x52,0xb2,0xe0,0x68,0xfc,0xb7,0xff,0xaa,0x72,0x03,
/* 384-bit generator g */
0x87,0x4b,0x10,0xce,0x20,0x4c,0xc6,0x36,0x84,0x3c,0x74,0xc5,0xb9,0xfa,0xc9,0x35,
0xd4,0xf2,0x79,0xb7,0x9b,0xb0,0x5f,0x2a,0xbd,0xc7,0x5d,0xf9,0xee,0xb8,0x84,0xc5,
0x85,0x04,0x1b,0x4d,0xc8,0xd8,0x83,0xae,0x0e,0x7e,0x45,0x84,0x3b,0x7e,0x36,0xce
};

static uint8_t bigint_mem[4 * SFZCRYPTO_DH_BYTES];
static uint8_t shared_secret_a[SFZCRYPTO_DH_BYTES];
static uint8_t shared_secret_b[SFZCRYPTO_DH_BYTES];
uint32_t shared_secret_a_len, shared_secret_b_len;

SfzCryptoContext *p_sfzcryptoctx = sfzcrypto_context_get();
SfzCryptoAsymKey dh_cntxt_a, dh_cntxt_b;
SfzCryptoBigInt pubkey_a, pubkey_b;
SfzCryptoStatus status;
uint32_t modulus_len = (512 / 8);

/* Setup DH contexts for party A and B */
memset(&dh_cntxt_a, 0, sizeof(dh_cntxt_a));
dh_cntxt_a.Key.dhPrivKey.prime_p.p_num = DH_PARAMS;
dh_cntxt_a.Key.dhPrivKey.prime_p.byteLen = modulus_len;
```

```
dh_cntxt_a.Key.dhPrivKey.base_g.p_num = DH_PARAMS + modulus_len;
dh_cntxt_a.Key.dhPrivKey.base_g.byteLen = sizeof(DH_PARAMS) - modulus_len;
dh_cntxt_a.Key.dhPrivKey.privkey.p_num = bigint_mem + (0 * SFZCRYPTO_DH_BYTES);
dh_cntxt_a.Key.dhPrivKey.privkey.byteLen = SFZCRYPTO_DH_BYTES;

memset(&dh_cntxt_b, 0, sizeof(dh_cntxt_b));
dh_cntxt_b.Key.dhPrivKey.prime_p.p_num = DH_PARAMS;
dh_cntxt_b.Key.dhPrivKey.prime_p.byteLen = modulus_len;
dh_cntxt_b.Key.dhPrivKey.base_g.p_num = DH_PARAMS + modulus_len;
dh_cntxt_b.Key.dhPrivKey.base_g.byteLen = sizeof(DH_PARAMS) - modulus_len;
dh_cntxt_b.Key.dhPrivKey.privkey.p_num = bigint_mem + (1 * SFZCRYPTO_DH_BYTES);
dh_cntxt_b.Key.dhPrivKey.privkey.byteLen = SFZCRYPTO_DH_BYTES;

/* Party A calculates its ephemeral DH key pair */
pubkey_a.p_num = bigint_mem + (2 * SFZCRYPTO_DH_BYTES);
pubkey_a.byteLen = SFZCRYPTO_DH_BYTES;
status = sfzcrypto_dh_publicpart_gen(p_sfzcryptoctx,
                                     &dh_cntxt_a, &pubkey_a);
assert(status == SFZCRYPTO_SUCCESS);
assert(dh_cntxt_a.Key.dhPrivKey.privkey.byteLen <= modulus_len);
assert(pubkey_a.byteLen <= modulus_len);

/* Party B calculates its ephemeral DH key pair */
pubkey_b.p_num = bigint_mem + (3 * SFZCRYPTO_DH_BYTES);
pubkey_b.byteLen = SFZCRYPTO_DH_BYTES;
status = sfzcrypto_dh_publicpart_gen(p_sfzcryptoctx,
                                     &dh_cntxt_b, &pubkey_b);
assert(status == SFZCRYPTO_SUCCESS);
assert(dh_cntxt_b.Key.dhPrivKey.privkey.byteLen <= modulus_len);
assert(pubkey_b.byteLen <= modulus_len);

/* Party B calculates the shared secret with A's public key */
status = sfzcrypto_dh_sharedsecret_gen(p_sfzcryptoctx,
                                       &dh_cntxt_b, &pubkey_a,
                                       shared_secret_b,
                                       &shared_secret_b_len);
assert(status == SFZCRYPTO_SUCCESS);

/* Party A calculates the shared secret with B's public key */
status = sfzcrypto_dh_sharedsecret_gen(p_sfzcryptoctx,
                                       &dh_cntxt_a, &pubkey_b,
                                       shared_secret_a,
                                       &shared_secret_a_len);
assert(status == SFZCRYPTO_SUCCESS);

/* Verify that both parties obtained the same secret */
assert(shared_secret_a_len == shared_secret_b_len);
assert(0 == memcmp(shared_secret_a,
                   shared_secret_b, shared_secret_a_len));
```

Equivalent functions exist for the ECDH algorithm `sfzcrypto_ecdh_publicpart_gen()` and `sfzcrypto_ecdh_sharedsecretgen()`. These are identical in use to Diffie-Hellman but the functions perform calculations using elliptic curve cryptography instead of modular exponentiation.

5.4 *Signature Generation and Verification*

RSA sign and verify is performed with API functions `sfzcrypto_rsa_sign()` and `sfzcrypto_rsa_verify()`. The function prototypes are:

```
SfzCryptoStatus
sfzcrypto_rsa_sign(
    SfzCryptoContext *  p_sfzcryptoctx,
    SfzCryptoAsymKey *  p_sigctx,
    SfzCryptoBigInt *   p_signature,
    uint8_t *           p_hash_msg,
    uint32_t            hash_msglen);
```

```
SfzCryptoStatus
sfzcrypto_rsa_verify(
    SfzCryptoContext *  p_sfzcryptoctx,
    SfzCryptoAsymKey *  p_sigctx,
    SfzCryptoBigInt *   p_signature,
    uint8_t *           p_hash_msg,
    uint32_t            hash_msglen);
```

Set the `p_sigctx->algo_type` field to select one of the signing algorithms shown in Table 2. It is important to note that the hash must have been pre-calculated with `sfzcrypto_hash_data()`, using the hash algorithm implied by the signing algorithm. The key provided to the sign function must be the private key of the key pair:

Table 2 RSA Signing Algorithms

Value	Padding scheme	Hash algorithm
SFZCRYPTO_ALGO_ASYMM_RSA_PKCS1_SHA1	PKCS #1	SHA-1
SFZCRYPTO_ALGO_ASYMM_RSA_PKCS1_SHA224	PKCS #1	SHA-2-224
SFZCRYPTO_ALGO_ASYMM_RSA_PKCS1_SHA256	PKCS #1	SHA-2-256
SFZCRYPTO_ALGO_ASYMM_RSA_PKCS1_MD5	PKCS #1	MD5
SFZCRYPTO_ALGO_ASYMM_RSA_PSS_SHA1	PSS	SHA-1
SFZCRYPTO_ALGO_ASYMM_RSA_PSS_SHA224	PSS	SHA-2-224
SFZCRYPTO_ALGO_ASYMM_RSA_PSS_SHA256	PSS	SHA-2-256
SFZCRYPTO_ALGO_ASYMM_RSA_PSS_MD5	PSS	MD5

The following example code shows how to sign and verify using RSA-PKCS1-SHA1:

```
static uint8_t plaintext[] = {
  0x12,0x34,0x56,0x78, 0x9a,0xbc,0xde,0xf1,
  0x23,0x45,0x67,0x89, 0xab,0xcd,0xef,0x12,
  0x34,0x56,0x78,0x9a, 0xbc,0xde,0xf1,0x23,
  0x45,0x67,0x89,0xab, 0xab,0xab,0xab,0xab };
static uint8_t pubExp[] = {
  0x00,0x01,0x00,0x01 };
static uint8_t privExp[] = {
  0x1a,0xe3,0x6b,0x75, 0x22,0xf6,0x64,0x87,
  0xd9,0xf4,0x61,0x0d, 0x15,0x50,0x29,0x0a,
  0xc2,0x02,0xc9,0x29, 0xbe,0xdc,0x70,0x32,
  0xcc,0x3e,0x02,0xac, 0xf3,0x7e,0x3e,0xbc,
  0x1f,0x86,0x6e,0xe7, 0xef,0x7a,0x08,0x68,
```

```
  0xd2,0x3a,0xe2,0xb1, 0x84,0xc1,0xab,0xd6,
  0xd4,0xdb,0x8e,0xa9, 0xbe,0xc0,0x46,0xbd,
  0x82,0x80,0x37,0x27, 0xf2,0x88,0x87,0x01 };
static uint8_t modulus[] = {
  0xc0,0x76,0x47,0x97, 0xb8,0xbe,0xc8,0x97,
  0x2a,0x0e,0xd8,0xc9, 0x0a,0x8c,0x33,0x4d,
  0xd0,0x49,0xad,0xd0, 0x22,0x2c,0x09,0xd2,
  0x0b,0xe0,0xa7,0x9e, 0x33,0x89,0x10,0xbc,
  0xae,0x42,0x20,0x60, 0x90,0x6a,0xe0,0x22,
  0x1d,0xe3,0xf3,0xfc, 0x74,0x7c,0xcf,0x98,
  0xae,0xcc,0x85,0xd6, 0xed,0xc5,0x2d,0x93,
  0xd5,0xb7,0x39,0x67, 0x76,0x16,0x05,0x25 };

SfzCryptoContext *p_sfzcryptoctx = sfzcrypto_context_get();
SfzCryptoHashContext hashCtx;
SfzCryptoAlgoAsym algo_type;
SfzCryptoAsymKey rsactx;
SfzCryptoBigInt signature;
SfzCryptoStatus status;

uint32_t modulusLen = 64;
uint32_t mod_bits = 512;
uint32_t digestlen;
uint32_t hash_algo;
uint8_t sig_buf[SFZCRYPTO_RSA_BYTES];
uint8_t digest[32];

/* Select the RSA-PKCS1-SHA1 signing algorithm */
algo_type = SFZCRYPTO_ALGO_ASYMM_RSA_PKCS1_SHA1;
hash_algo = SFZCRYPTO_ALGO_HASH_SHA160;
digestlen = 20;

/* Setup RSA Sign Key / Context */
rsactx.algo_type = algo_type;
rsactx.cmd_type = SFZCRYPTO_CMD_SIG_GEN;
rsactx.mod_bits = mod_bits;
rsactx.Key.rsaPrivKey.cofQinv.p_num = NULL;
rsactx.Key.rsaPrivKey.dmodP.p_num = NULL;
rsactx.Key.rsaPrivKey.modulus.p_num = modulus;
rsactx.Key.rsaPrivKey.modulus.byteLen = modulusLen;
rsactx.Key.rsaPrivKey.privexp.p_num = privExp;
rsactx.Key.rsaPrivKey.privexp.byteLen = sizeof(privExp);

/* Hash data to be signed */
memset(&hashCtx, 0x00, sizeof(SfzCryptoHashContext));
hashCtx.algo = hash_algo;
status = sfzcrypto_hash_data(p_sfzcryptoctx,
                             &hashCtx,
                             plaintext,
                             sizeof(plaintext),
                             true, true);
assert(status == SFZCRYPTO_SUCCESS);

/* Generate signature */
memcpy(digest, hashCtx.digest, digestlen);
signature.p_num = sig_buf;
signature.byteLen = sizeof(sig_buf);
```

```
status = sfzcrypto_rsa_sign(p_sfzcryptoctx,
                            &rsactx,
                            &signature,
                            digest, digestlen);
assert(status == SFZCRYPTO_SUCCESS);
assert(signature.byteLen == modulusLen);

/* Setup RSA Verify Key / Context */
rsactx.algo_type = algo_type;
rsactx.cmd_type = SFZCRYPTO_CMD_SIG_VERIFY;
rsactx.mod_bits = mod_bits;
rsactx.Key.rsaPubKey.modulus.p_num = modulus;
rsactx.Key.rsaPubKey.modulus.byteLen = modulusLen;
rsactx.Key.rsaPubKey.pubexp.p_num = pubExp;
rsactx.Key.rsaPubKey.pubexp.byteLen = sizeof(pubExp);

/* Verify signature */
status = sfzcrypto_rsa_verify(p_sfzcryptoctx,
                              &rsactx,
                              &signature,
                              digest, digestlen);
assert(status == SFZCRYPTO_SUCCESS);
```

Note that `sfzcrypto_rsa_verify()` fails with status `SFZCRYPTO_SIGNATURE_CHECK_FAILED` if the signature does not match.

SafeZone also supports elliptic curve cryptography for signing, via `sfzcrypto_ecdsa_sign()` and `sfzcrypto_ecdsa_verify()`. Alternatively it is possible to use the DSA algorithm, via `sfzcrypto_dsa_sign()` and `sfzcrypto_dsa_verify()`. These ECDSA and DSA functions have the same parameters as the corresponding RSA functions, only the key type differs. Instead of RSA keys the ECDSA algorithm requires ECDSA keys and the DSA algorithm requires DSA keys.

Function prototypes of DSA/ECDSA sign/verify is as follows:

```
SfzCryptoStatus
sfzcrypto_dsa/ecdsa_sign/verify(
    SfzCryptoContext *  p_sfzcryptoctx,
    SfzCryptoAsymKey *  p_sigctx,
    SfzCryptoSign *     p_signature,
    uint8_t *           p_hash_msg,
    uint32_t            hash_msglen);
```

6 Authenticated Unlock / Secure Debug

Based on the *Secure Debug Application Note* which is provided in the *SafeXcel-IP-123 CM Firmware* package, a reference example is provided as part of the CAL functionality that illustrates the Authenticated Unlock Process for Secure Debug.

The next code snippet from the example code shows all essential parts for the Secure Debug functionality.

```
{
    int MainReturnCode = 0;
    uint8_t NonceA[16];
    uint8_t NonceB[16];
    uint32_t NonceLength = 16;
    SfzCryptoStatus funcres;
    SfzCryptoAssetId AuthStateASId = SFZCRYPTO_ASSETID_INVALID;
    SfzCryptoBigInt Signature;

    // Initialize CAL environment
    cal_init();

    // Start the Authenticated Unlock operation
    // Note: The called function includes the find asset and create asset
    //       operations as shown in figure 4 of the Secure Debug Application
    //       Note.
    funcres = sfzcrypto_authenticated_unlock_start(gl_PublicKey_AssetNumber,
                                                   &AuthStateASId,
                                                   NonceB, &NonceLength);
    if (funcres != SFZCRYPTO_SUCCESS)
    {
        …
        goto main_error;
    }
    fprintf(stderr, "%s: Authenticated Unlock Start PASSED.\n", gl_Program);

    // Perform the Authentication Service actions
    if (AuthenticationService(NonceB, NonceA, &Signature) < 0)
    {
        …
        goto main_error;
    }
    fprintf(stderr, "%s: Authenticated Unlock signature generated.\n",
            gl_Program);

    // Perform the Authenticated Unlock verify operation, which on success
    // grants use of the sfzcrypto_secure_debug().
    funcres = sfzcrypto_authenticated_unlock_verify(AuthStateASId,
                                                    &Signature, NonceA, 16);
    if (funcres != SFZCRYPTO_SUCCESS)
    {
        …
        goto main_error;
    }
    fprintf(stderr, "%s: Authenticated Unlock Verify PASSED.\n", gl_Program);
```

```
    // Enable Secure Debug
    // - Asserts the Secure Debug port bits that are defined in the AuthInfo of
    //   the AuthKey.
    funcres = sfzcrypto_secure_debug(AuthStateASId, true);
    if (funcres != SFZCRYPTO_SUCCESS)
    {
        …
        goto main_error;
    }
    fprintf(stderr, "%s: Secure Debug enabled.\n", gl_Program);

    // Disable Secure Debug
    // - De-asserts the Secure Debug port bits that are defined in the AuthInfo
    //   of the AuthKey.
    funcres = sfzcrypto_secure_debug(AuthStateASId, false);
    if (funcres != SFZCRYPTO_SUCCESS)
    {
        fprintf(stderr, "%s: Secure Debug disable failed\n", gl_Program);
        MainReturnCode = 1;
    }
    fprintf(stderr, "%s: Secure Debug disabled.\n", gl_Program);

main_error:
    // Remove the Asset that holds the Authenticated Unlock state
    if (AuthStateASId != SFZCRYPTO_ASSETID_INVALID)
    {
        funcres = sfzcrypto_authenticated_unlock_release(AuthStateASId);
        if (funcres != SFZCRYPTO_SUCCESS)
        {
            fprintf(stderr, "%s: Authenticated Unlock cleanup failed\n",
                    gl_Program);
        }
    }

    return MainReturnCode;
}
```

The function `sfzcrypto_authenticated_unlock_start()` starts the Authenticated Unlock session. Note that the function hides the search for the `AuthKey` object and creation of the `AuthState` Asset that holds the session state. The Nonce (R_B) that is provided as result of a successful function call, must be passed on to the `AuthenticationService()` function which represents an external process. This external process must provide an additional Nonce (R_A) and a Signature of the message, which is result of the concatenation of R_A and R_B. The Authentication Service Signature and Nonce must be passed on to the function `sfzcrypto_authenticated_unlock_verify()`. Successful verification of the Signature unlocks the access (use) of the `sfzcrypto_secure_debug()` function, with which the Secure Debug port can be controlled.

When the Authenticated Unlock session is no longer needed, the session must be released with the function `sfzcrypto_authenticated_unlock_release()`.

7 Support for CPRM in CAL

The following paragraphs give an overview of how CAL supports CPRM. It is not intended as an introduction to CPRM, so the reader is assumed to be familiar with the basics of CPRM.

Most of CAL's CPRM related functionality is available through the `sfzcrypto_cprm_c2_derive` API. This API, combined with the *Asset Store* and `sfzcrypto_symm_crypt` APIs, allows all key material involved in accessing CPRM-protected content to be handled in a secure way. What we mean by secure here is that is all key material (starting with the *Device Keys* and ending with the *Content Keys*) stay within the security boundary provided by the *Crypto Module*.

Study the description of the `sfzcrypto_cprm_c2_derive` function in "`sfzcryptoapi_cprm.h`" to get familiar with the details. Then look at the code in 7.3 for how the `sfzcrypto_cprm_c2_derive` API is intended to be used to access CPRM content stored on an SD Card.

7.1 The "Facsimile" variant of the C2 cipher

The CPRM protection scheme uses the C2 block cipher with a set of *Secret Constants* that are only disclosed to licensees. For development and test purposes however, the 4C Entity provides some test vectors for the C2 block cipher based on an alternative set of constants, referred to as the "Facsimile" *Secret Constants*. Besides test vectors for the C2 block cipher itself, the 4C Entity also provides examples of MKB processing based on the "Facsimile" variant of C2. This entire set of test vectors can be downloaded from 4C Entity's website, see A.2.3.

7.2 Device Key Storage

Licensees of the CPRM technology are required to use state-of-the-art measures to prevent disclosure of the C2 cipher's *Secret Constants* and of the *Device Keys* embedded in each CPRM-compliant device. The CAL API in combination with a hardware-based *Crypto Module* with *Asset Store* functionality provides an excellent way to achieve the goal of protecting these constants. The C2 cipher, including its *Secret Constants*, is embedded in the *Crypto Module* while the *Device Keys* are saved in the *Asset Store*.

The example code below shows how to use CAL to put a CPRM *Device Key* in a *KeyBlob*. In fact, since the *KeyBlob* also holds the *Column* and *Row* number associated with the *Device Key*, the term *Device Key Object* is preferred to describe the *KeyBlob*'s content.

Alternatively, *Device Key Objects* can also be stored as static assets in NVM. In both cases, the store operation should be done in a secure production environment, i.e. before the device containing the *Crypto Module* is delivered to the end user.

```
static uint8_t DEVKEYSTORE_KEK_LABEL [] = "DeviceKeyStorage";
static uint8_t DEVKEY_ADDITIONAL_DATA [] = "C2DeviceKey";

#define DEVKEY_BLOB_SIZE SFZCRYPTO_KEYBLOB_SIZE(16)
static struct
{
  int ColumnNr;
  uint8_t WrappedDeviceKeyObject[DEVKEY_BLOB_SIZE];
  uint32_t ObjectSize;
} DeviceKeyObjectStore[2];

static void
WrapDeviceKeyObjects(void)
{
/* Note that SafeZone represents plain Device keys in the following format:
 * byte 0:      column number
 * byte 1..7:   actual device key
 * byte 8..9:   device key's row number, LSB first
```

```
 * byte 10..11: padding bytes, which can have any value.
 */
static const uint8_t C2_DEVKEY_COL3_ROW9 [] =
  {0x03, 0x76,0xCB,0xE4,0xBA,0xF6,0x3B,0x65, 0x09,0x00, 0xDC,0xDC};
static const uint8_t C2_DEVKEY_COL5_ROW9 [] =
  {0x05, 0x50,0x34,0xF4,0x48,0x49,0x5F,0xDF, 0x09,0x00, 0xDC,0xDC};

/* The above Device key data is taken from the 4C Entity's "CPRM -
   SD Cards" MKB processing example, Simple Test, "simple-test-
   device-key-sets.txt", device key set 0. */

SfzCryptoContext *sfzcryptoctx_p = sfzcrypto_context_get();
SfzCryptoAssetId RootKeyAssetID, KekAssetID, DeviceKeyAssetID;
SfzCryptoStatus ret;
int i;

/* Get the RootKey's asset id. */
RootKeyAssetID = sfzcrypto_asset_get_root_key();
assert (RootKeyAssetID != SFZCRYPTO_ASSETID_INVALID);

/* Create a KEK asset and load it (through derivation). */
ret = sfzcrypto_asset_alloc(sfzcryptoctx_p,
                            SFZCRYPTO_POLICY_SECURE_WRAP |
                            SFZCRYPTO_POLICY_SECURE_UNWRAP,
                            32, &KekAssetID);
assert (ret == SFZCRYPTO_SUCCESS);
ret = sfzcrypto_asset_derive(sfzcryptoctx_p,
                             KekAssetID,
                             RootKeyAssetID,
                             DEVKEYSTORE_KEK_LABEL,
                             sizeof(DEVKEYSTORE_KEK_LABEL));
assert (ret == SFZCRYPTO_SUCCESS);

/* Make KeyBlobs with the Col3, Row9 & Col5, Row9 Device Key Objects */
for (i = 0; i < 2; i++)
{
  uint8_t * KeyBlob_p = DeviceKeyObjectStore[i].WrappedDeviceKeyObject;
  const uint8_t * DevKeyObject_p = (i == 0) ? C2_DEVKEY_COL3_ROW9
                                            : C2_DEVKEY_COL5_ROW9;
  uint32_t KeyBlobSize;

  ret = sfzcrypto_asset_alloc(sfzcryptoctx_p,
                              SFZCRYPTO_POLICY_C2_KM_DERIVE |
                              SFZCRYPTO_POLICY_ALGO_CIPHER_C2,
                              12, &DeviceKeyAssetID);
  assert (ret == SFZCRYPTO_SUCCESS);


KeyBlobSize = DEVKEY_BLOB_SIZE;  /* must be >= actual KeyBlob size */
  ret = sfzcrypto_asset_load_key_and_wrap(sfzcryptoctx_p,
                                DeviceKeyAssetID,
                                DevKeyObject_p,
                                sizeof(C2_DEVKEY_COL3_ROW9),
                                KekAssetID,
                                DEVKEY_ADDITIONAL_DATA,
                                sizeof(DEVKEY_ADDITIONAL_DATA),
                                KeyBlob_p, &KeyBlobSize);
```

```
  assert (ret == SFZCRYPTO_SUCCESS);
  DeviceKeyObjectStore[i].ColumnNr = (int)DevKeyObject_p[0];


/* Save actual KeyBlob size: */
  DeviceKeyObjectStore[i].ObjectSize = KeyBlobSize;
  /* Free Device key asset. */
  ret = sfzcrypto_asset_free(sfzcryptoctx_p, DeviceKeyAssetID);
  assert (ret == SFZCRYPTO_SUCCESS);
} /* for */

/* Free KEK asset. */
ret = sfzcrypto_asset_free(sfzcryptoctx_p, KekAssetID);
assert (ret == SFZCRYPTO_SUCCESS);
}
```

The next paragraph shows how the `sfzcrypto_cprm_c2_devicekeyobject_rownr_get` API allows one to obtain the *Row* value associated with a *Device Key*, given a handle (*Asset ID*) for the *Device Key Object* with that key. That handle is obtained either during the process of importing the relevant *KeyBlob* or as the result of a `sfzcrypto_asset_search` operation.

7.3 C2 / CPRM Key Derivation

CAL only supports CPRM-related functionality in case the C2 cipher is supported. The following code shows how the `sfzcrypto_get_featurematrix` API is used to detect whether the C2 cipher is available. If so, a function is called that demonstrates how the CAL API (in particular the *Asset Store* API and the `sfzcrypto_cprm_c2_derive` API) is used to implement core parts of CPRM.

```

typedef struct
{
    SfzCryptoAssetId AssetArray[4];
} C2AssetContext_t;

/* Flags that can be used to request 'c2_asset_derive' to perform
   a defined sequence of C2 key derivations steps. */
#define C2_ASSET_DERIVE_MAKE_KEK          (1 << 0)
#define C2_ASSET_DERIVE_STEP_LOAD_DK_1    (1 << 1)
#define C2_ASSET_DERIVE_STEP_MAKE_KM_1    (1 << 2)
#define C2_ASSET_DERIVE_STEP_TEST_CCR     (1 << 3)
#define C2_ASSET_DERIVE_STEP_LOAD_DK_2    (1 << 4)
#define C2_ASSET_DERIVE_STEP_MAKE_KM_2    (1 << 5)
#define C2_ASSET_DERIVE_STEP_TEST_KM_2    (1 << 6)
#define C2_ASSET_DERIVE_STEP_MAKE_KMU     (1 << 7)
#define C2_ASSET_DERIVE_STEP_DO_AKE_P1    (1 << 8)
#define C2_ASSET_DERIVE_STEP_CALC_RESP    (1 << 9)
#define C2_ASSET_DERIVE_STEP_DO_AKE_P2    (1 << 10)
#define C2_ASSET_DERIVE_STEP_TEST_KS      (1 << 11)
#define C2_ASSET_DERIVE_STEP_MAKE_KT_1    (1 << 12)
#define C2_ASSET_DERIVE_STEP_TEST_KT_1    (1 << 13)

#define C2_ASSET_DERIVE_MAKE_AND_USE_KT_1 0x03FFF

static void
CPRMExample(void)
{
```

```
  SfzCryptoFeatureMatrix features;
  C2AssetContext_t cntxt;
  SfzCryptoStatus status;

  /* Only run the example if the C2 cipher is supported */
  status = sfzcrypto_get_featurematrix(&features);
  assert (status == SFZCRYPTO_SUCCESS);
  if (features.f_keytypes[SFZCRYPTO_KEY_C2])
  {
    c2_asset_derive(&cntxt, C2_ASSET_DERIVE_MAKE_AND_USE_KT_1);
  }
}
```

Due to its length, the code of the `c2_asset_derive` function is broken up in several sections. After each section, some remarks are made on the code covered in that section.

7.3.1 C2 Key Derivation Example: Test Data

```
/*----------------------------------------------------------------------
 * c2_asset_derive
 *
 * Helper function that performs the sequence of C2 key derivation and
 * test steps specified by 'steps'.
 */
static void
c2_asset_derive(
    C2AssetContext_t * const cntxt_p,
    uint32_t steps)
{
/* The following data is taken from the 4C Entity's "CPRM - SD Cards"
   MKB processing example, Simple Test 3b, with device key set 0.
   The expected MediaKey to be produced is: 2C52E97257FAC3 */

static const uint8_t MKB_VERIFY_DATA [] =
  {0xCA, 0xBD, 0x4F, 0x63, 0xB8, 0xDE, 0x8B, 0x59};
static const uint8_t MKB_Dke_C3R9 [] =
  {0x7A, 0x45, 0x0E, 0xC1, 0x95, 0x08, 0x3F, 0xF0};
static const uint8_t MKB_CCR_DCE_DATA [] =
  {0xDC, 0xC9, 0x4B, 0x27, 0x4A, 0x2D, 0x01, 0xC8};
static const uint8_t MKB_Dkde_C5R9 [] =
  {0x78, 0xF6, 0x50, 0x2E, 0x44, 0x56, 0x7C, 0x70};

/* The following values do not correspond to any official, publicly
   available test data. */

/* An arbitrary ID_MEDIA value: */
static const uint8_t ID_MEDIA [] =
  {0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF};
/* The corresponding MediaUniqueKey, i.e. C2_G(MediaKey, ID_MEDIA) */
static const uint8_t MEDIA_UNIQUE_KEY [] =
  {0xFC, 0x65, 0x28, 0x9A, 0x56, 0x6D, 0x50};

/* For this test, the AKE protocol is executed in such a way that the
   resulting SessionKey is always C2_G(~MediaUniqueKey, 0). The
```

```
   following data is the result of C2_ECBC(SessionKey, 00010203..
   0F1011..17), and is used to verify if the expected SessionKey
   was derived. */
static const uint8_t PATT_UNDER_KS [] =
  {0x0c, 0x48, 0x6e, 0x98, 0x57, 0x39, 0x46, 0x75,
   0xf6, 0x17, 0xeb, 0xad, 0x1e, 0xd6, 0x2f, 0x31,
   0x66, 0x46, 0x52, 0xb8, 0x1a, 0x31, 0x10, 0xf3};
/* The following data is C2_E(MediaUniqueKey, 0311223344556677), i.e.
   a simulated item from the Protected Area of an SD Card containing a
   ContentKey with the value 11223344556677 and a CCI (copy control
   information) value of 0x03. */
static const uint8_t ENC_KT_1_CCI [] =
  {0xfb, 0xf3, 0x78, 0x64, 0x5d, 0x90, 0x20, 0x04};
/* The following data is C2_ECBC(ContentKey, 00010203..0F1011..17),
   and is used to verify if the expected ContentKey was derived. */
static const uint8_t PATT_UNDER_KT_1 [] =
  {0xbc, 0xc5, 0xbf, 0x81, 0xb9, 0xdc, 0x0e, 0xeb,
   0x79, 0x53, 0xdd, 0x29, 0x06, 0x7d, 0xb2, 0x1e,
   0x2b, 0x58, 0xb4, 0xcd, 0x2e, 0x02, 0x8b, 0xf4};
/* An arbitrary "SD Card command argument" value used in AKE phase 1: */
static const uint8_t SD_ARG [] = {0xb0, 0xb1, 0xb2, 0xb3};

static const uint8_t DEAD_BEEF [] =
  {0xDE, 0xAD, 0xBE, 0xEF};
static const uint8_t PLAIN_CCR_COL_GENERATION [] =
  {0x05, 0x00, 0x00, 0x01};
```

The part of the CPRM example code above mainly declares test data. Some of that data (notably the MKB elements and the Device Keys used in 7.2) was taken from publicly available test data, in particular the MKB processing examples for SD Cards provided by the 4C Entity in [C2-MKB-SD-CARDS].

The rest of the test data was chosen arbitrarily and/or obtained through software implementations of the "Facsimile" variant of the C2 cipher in its various modes: C2_E, C2_D, C2_ECBC, C2_DCBC and C2_G.

7.3.2 C2 Key Derivation Example: MKB processing

```
SfzCryptoContext *sfzcryptoctx_p = sfzcrypto_context_get();
SfzCryptoAssetId RootKeyAssetID, KekAssetID, DeviceKeyAssetID;
SfzCryptoStatus ret;
uint8_t Challenge[8];
uint8_t Response[8];
uint16_t RowNr;
int i;

for (i = 0; (1 << i) <= C2_ASSET_DERIVE_STEP_TEST_KT_1; i++)
{
  uint8_t InputData[128];
  uint8_t OutputData[128];
  uint32_t OutputDataLen = 0xFF;
  uint32_t ExpectedOutLen = 0;
  bool fTestPattern = false;

  switch (steps & (1 << i))
  {
```

```
    case 0:
      /* Skip a step if bit 'i' of 'steps' is not set. */
      ret = SFZCRYPTO_SUCCESS;
      OutputDataLen = 0;
      break;

    case C2_ASSET_DERIVE_MAKE_KEK:
      /* Re-construct the KEK that was used to wrap the Device
         Key Objects. So get the RootKey's asset id. */
      RootKeyAssetID = sfzcrypto_asset_get_root_key();
      assert (RootKeyAssetID != SFZCRYPTO_ASSETID_INVALID);

      /* Create a KEK asset and load it through derivation. */
      ret = sfzcrypto_asset_alloc(sfzcryptoctx_p,
                                  SFZCRYPTO_POLICY_SECURE_WRAP |
                                  SFZCRYPTO_POLICY_SECURE_UNWRAP,
                                  32, &KekAssetID);
      assert (ret == SFZCRYPTO_SUCCESS);
      ret = sfzcrypto_asset_derive(sfzcryptoctx_p,
                                   KekAssetID,
                                   RootKeyAssetID,
                                   DEVKEYSTORE_KEK_LABEL,
                                   sizeof(DEVKEYSTORE_KEK_LABEL));
      OutputDataLen = 0;
      break;

    case C2_ASSET_DERIVE_STEP_LOAD_DK_1:
      /* Load the device key for column 3, row 9 and save the
         asset id in AssetArray[0]. */
      ret = sfzcrypto_asset_alloc(sfzcryptoctx_p,
                                  SFZCRYPTO_POLICY_C2_KM_DERIVE |
                                  SFZCRYPTO_POLICY_ALGO_CIPHER_C2,
                                  12,
                                  &cntxt_p->AssetArray[0]);
      assert (ret == SFZCRYPTO_SUCCESS);
      assert (DeviceKeyObjectStore[0].ColumnNr == 3);
      ret = sfzcrypto_asset_import(sfzcryptoctx_p,
                      cntxt_p->AssetArray[0],
                      KekAssetID,
                      DEVKEY_ADDITIONAL_DATA,
                      sizeof(DEVKEY_ADDITIONAL_DATA),
                      DeviceKeyObjectStore[0].WrappedDeviceKeyObject,
                      DeviceKeyObjectStore[0].ObjectSize);
      assert (ret == SFZCRYPTO_SUCCESS);
      /* Get the Device key's Row number and verify that it is 9. */
      ret = sfzcrypto_cprm_c2_devicekeyobject_rownr_get(
                                        cntxt_p->AssetArray[0],
                                        &RowNr);
      assert (ret == SFZCRYPTO_SUCCESS);
      assert (RowNr == 9);
      OutputDataLen = 0;
      break;

    case C2_ASSET_DERIVE_STEP_MAKE_KM_1:
      /* Derive a provisional MediaKey from a Calculate Media Key
         Record and the device key that was just loaded. After that,
         free the device key asset. Use AssetArray[1] to store the
```

```
        asset id for the provisional MediaKey. */
     ret = sfzcrypto_asset_alloc(sfzcryptoctx_p,
                                 SFZCRYPTO_POLICY_C2_KMU_DERIVE |
                                 SFZCRYPTO_POLICY_ALGO_CIPHER_C2,
                                 8,
                                 &cntxt_p->AssetArray[1]);
     assert (ret == SFZCRYPTO_SUCCESS);
     ret = sfzcrypto_cprm_c2_derive(SFZCRYPTO_CPRM_C2_KM_DERIVE,
                                    cntxt_p->AssetArray[0],
                                    SFZCRYPTO_ASSETID_INVALID,
                                    cntxt_p->AssetArray[1],
                                    MKB_Dke_C3R9,
                                    sizeof(MKB_Dke_C3R9),
                                    NULL,
                                    &OutputDataLen);
     assert (ret == SFZCRYPTO_SUCCESS);
     ret = sfzcrypto_asset_free(sfzcryptoctx_p,
                                cntxt_p->AssetArray[0]);
     break;

   case C2_ASSET_DERIVE_STEP_TEST_CCR:
     /* Verify that the Dce part of a Conditionally Calculate Media
        Key Record can be successfully decrypted in this case.
        Note that it is not an MKB processing error if this decryption
        fails with ret == SFZCRYPTO_VERIFY_FAILED. It only implies that
        this CPRM-capable device must ignore this Record. */
     ret = sfzcrypto_cprm_c2_derive(SFZCRYPTO_CPRM_C2_KM_VERIFY,
                                    cntxt_p->AssetArray[1],
                                    SFZCRYPTO_ASSETID_INVALID,
                                    SFZCRYPTO_ASSETID_INVALID,
                                    MKB_CCR_DCE_DATA,
                                    sizeof(MKB_CCR_DCE_DATA),
                                    OutputData,
                                    &OutputDataLen);
     assert (ret == SFZCRYPTO_SUCCESS);
     ExpectedOutLen = 8;
     assert (c_memcmp(OutputData, DEAD_BEEF, 4) == 0);
     assert (c_memcmp(OutputData + 4,
                      PLAIN_CCR_COL_GENERATION, 4) == 0);
     break;

   case C2_ASSET_DERIVE_STEP_LOAD_DK_2:
     /* Load the Device Key for column 5, row 9 and save the asset id
        in AssetArray[0]. */
     ret = sfzcrypto_asset_alloc(sfzcryptoctx_p,
                                 SFZCRYPTO_POLICY_C2_KM_DERIVE |
                                 SFZCRYPTO_POLICY_ALGO_CIPHER_C2,
                                 12,
                                 &cntxt_p->AssetArray[0]);
     assert (ret == SFZCRYPTO_SUCCESS);
     assert (DeviceKeyObjectStore[1].ColumnNr == 5);

     ret = sfzcrypto_asset_import(sfzcryptoctx_p,
                      cntxt_p->AssetArray[0],
                      KekAssetID,
                      DEVKEY_ADDITIONAL_DATA,
                      sizeof(DEVKEY_ADDITIONAL_DATA),
```

```
                        DeviceKeyObjectStore[1].WrappedDeviceKeyObject,
                        DeviceKeyObjectStore[1].ObjectSize);
      ret = sfzcrypto_cprm_c2_devicekeyobject_rownr_get(
                        cntxt_p->AssetArray[0],
                        &RowNr);
      assert (ret == SFZCRYPTO_SUCCESS);
      assert (RowNr == 9);
      OutputDataLen = 0;
      break;

    case C2_ASSET_DERIVE_STEP_MAKE_KM_2:
      /* Derive the next provisional MediaKey from the current MediaKey,
         Dkde data from the Conditionally Calculate Media Key Record and
         the device key that was just loaded. After that, free the asset
         with the device key. */
      ret = sfzcrypto_cprm_c2_derive(SFZCRYPTO_CPRM_C2_KM_UPDATE,
                                      cntxt_p->AssetArray[1],
                                      cntxt_p->AssetArray[0],
                                      cntxt_p->AssetArray[1],
                                      MKB_Dkde_C5R9,
                                      sizeof(MKB_Dkde_C5R9),
                                      NULL,
                                      &OutputDataLen);
      assert (ret == SFZCRYPTO_SUCCESS);
      ret = sfzcrypto_asset_free(sfzcryptoctx_p,
                                  cntxt_p->AssetArray[0]);
      break;

    case C2_ASSET_DERIVE_STEP_TEST_KM_2:
      /* Verify that the exepcted final MediaKey has been produced. */
      ret = sfzcrypto_cprm_c2_derive(SFZCRYPTO_CPRM_C2_KM_VERIFY,
                                      cntxt_p->AssetArray[1],
                                      SFZCRYPTO_ASSETID_INVALID,
                                      SFZCRYPTO_ASSETID_INVALID,
                                      MKB_VERIFY_DATA,
                                      sizeof(MKB_VERIFY_DATA),
                                      OutputData,
                                      &OutputDataLen);
      ExpectedOutLen = 8;
      assert (c_memcmp(OutputData, DEAD_BEEF, 4) == 0);
      break;
```

The code above shows how to do MKB processing with CAL. Please observe the following:

- This example uses *KeyBlobs* with *Device Key Objects* that were created as shown in 7.2.
- The `SFZCRYPTO_CPRM_C2_KM_DERIVE` flag must be passed to the `sfzcrypto_cprm_c2_derive` function to handle a *Calculate Media Key* record from the MKB. This function then fills a previously allocated asset, which must have `SFZCRYPTO_POLICY_C2_KMU_DERIVE | SFZCRYPTO_POLICY_ALGO_CIPHER_C2` as policy, with the (provisional) *MediaKey*. This key is called 'provisional' since it is usually updated one or more times before the actual *MediaKey* is produced.
- The `SFZCRYPTO_CPRM_C2_KM_VERIFY` flag must be passed to the `sfzcrypto_cprm_c2_derive` function to check if a *Conditionally Calculate Media Key* record should be skipped or not. If the record should not be skipped, this function also reveals the record's *Column* number

and generation info. Note that the same function is used later to verify if the correct *MediaKey* was produced.

- The `SFZCRYPTO_CPRM_C2_KM_UPDATE` flag must be passed to the `sfzcrypto_cprm_c2_derive` function to update the current *MediaKey* value in case a *Conditionally Calculate Media Key* record must be actually processed.
- Use the `sfzcrypto_cprm_c2_derive` function with the `SFZCRYPTO_CPRM_C2_KM_VERIFY` flag to process the *Verify Media Key* record, in order to check whether the current *MediaKey* value is the correct, final value.

7.3.3 C2 Key Derivation Example: Media Unique Key and Session Key Derivation

```
case C2_ASSET_DERIVE_STEP_MAKE_KMU:
  /* Derive the MediaUniqueKey and store its asset id in
     AssetArray[2]. */
  ret = sfzcrypto_asset_alloc(sfzcryptoctx_p,
                             SFZCRYPTO_POLICY_C2_KS_DERIVE |
                             SFZCRYPTO_POLICY_C2_KZ_DERIVE |
                             SFZCRYPTO_POLICY_ALGO_CIPHER_C2,
                             8,
                             &cntxt_p->AssetArray[2]);
  assert (ret == SFZCRYPTO_SUCCESS);
  ret = sfzcrypto_cprm_c2_derive(SFZCRYPTO_CPRM_C2_KMU_DERIVE,
                                cntxt_p->AssetArray[1],
                                SFZCRYPTO_ASSETID_INVALID,
                                cntxt_p->AssetArray[2],
                                ID_MEDIA,
                                sizeof(ID_MEDIA),
                                NULL,
                                &OutputDataLen);
  break;

case C2_ASSET_DERIVE_STEP_DO_AKE_P1:
  /* Execute the first phase for deriving a SessionKey. Use
     AssetArray[3] to store the asset id of the SessionKey. */
  ret = sfzcrypto_asset_alloc(sfzcryptoctx_p,
                             SFZCRYPTO_POLICY_FUNCTION_ENCRYPT |
                             SFZCRYPTO_POLICY_FUNCTION_DECRYPT |
                             SFZCRYPTO_POLICY_ALGO_CIPHER_C2,
                             8,
                             &cntxt_p->AssetArray[3]);
  assert (ret == SFZCRYPTO_SUCCESS);
  ret = sfzcrypto_cprm_c2_derive(SFZCRYPTO_CPRM_C2_AKE_PHASE1,
                                cntxt_p->AssetArray[2],
                                SFZCRYPTO_ASSETID_INVALID,
                                cntxt_p->AssetArray[3],
                                SD_ARG,
                                sizeof(SD_ARG),
                                OutputData,
                                &OutputDataLen);
  if (OutputDataLen == 8)
  {
    c_memcpy(Challenge, OutputData, 8);
  }
  ExpectedOutLen = 8;
  break;
```

```
    case C2_ASSET_DERIVE_STEP_CALC_RESP:
      /* Calculate the response on the challenge generated in the
         previous step. Note that we're simulating an operation here
         that is normally done by the SD Card. */
      {
        SfzCryptoCipherContext CiphContext = {0};
        SfzCryptoCipherKey CiphKey;

        CiphContext.fbmode = SFZCRYPTO_MODE_ECB;
        SFZCRYPTO_CIPHER_KEY_INIT(&CiphKey,
                                  SFZCRYPTO_KEY_C2,
                                  MEDIA_UNIQUE_KEY,
                                  sizeof(MEDIA_UNIQUE_KEY));
        ret = sfzcrypto_symm_crypt(sfzcryptoctx_p,
                                   &CiphContext,
                                   &CiphKey,
                                   Challenge, 8,
                                   Response,
                                   &OutputDataLen,
                                   SFZ_ENCRYPT);

        if (OutputDataLen == 8)
        {
          int k;

          /* Finish the C2_G(MediaUniqueKey, Challenge) operation. */
          for (k = 0; k < 8; k++)
          {
            Response[k] ^= Challenge[k];
          }
        }

        ExpectedOutLen = 8;
      }
      break;

    case C2_ASSET_DERIVE_STEP_DO_AKE_P2:
      /* Execute the second phase of the AKE protocol, i.e. check the
         response on our challenge and respond to the challenge from the
         SD Card. Note that we set the SD Card's challenge equal to our
         own, in order to get a predictable SessionKey = C2_G(~Media-
         UniqueKey, 0). */
      c_memcpy(InputData, Response, 8);
      c_memcpy(InputData + 8, Challenge, 8);
      ret = sfzcrypto_cprm_c2_derive(SFZCRYPTO_CPRM_C2_AKE_PHASE2,
                                     cntxt_p->AssetArray[2],
                                     SFZCRYPTO_ASSETID_INVALID,
                                     cntxt_p->AssetArray[3],
                                     InputData, 8 + 8,
                                     OutputData, &OutputDataLen);
      ExpectedOutLen = 8;
      break;
```

The code section above shows how to use CAL to derive a *MediaUniqueKey* and how to use that key to run the AKE (Authentication and Key Exchange) protocol and produce a *SessionKey*. Note that:

- The AKE protocol must be executed in order get access to the *Protected Area* of an SD Card. CPRM uses that area to store data needed for the recovery of one or more *ContentKey*s, i.e. the keys needed to decrypt CPRM-protected content.
- An asset with policy `SFZCRYPTO_POLICY_C2_KZ_DERIVE` | `SFZCRYPTO_POLICY_C2_KS_DERIVE` |`SFZCRYPTO_POLICY_ALGO_CIPHER_C2` must be created to hold the *MediaUniqueKey*.
- The `SFZCRYPTO_CPRM_C2_KMU_DERIVE` flag must be passed to the `sfzcrypto_cprm_c2_derive` function to derive the *MediaUniqueKey* value. This value depends on the *MediaKey* (produced through MKB processing) and some ID_MEDIA value.
- An asset with policy `SFZCRYPTO_POLICY_C2_DECRYPT` |`SFZCRYPTO_POLICY_ALGO_CIPHER_C2` must be created to hold the *SessionKey*. Adding `SFZCRYPTO_POLICY_C2_ENCRYPT` to the policy is allowed but typically unnecessary.
- The `SFZCRYPTO_CPRM_C2_AKE_PHASE1` flag must be passed to the `sfzcrypto_cprm_c2_derive` function to perform the first phase of the AKE protocol. In this phase, a random challenge (*Challenge1*) is produced in accordance with the SD Card specification. Note that this implies that *Challenge1* is also used to securely send some 32-bit command argument to the SD Card.
- The `SFZCRYPTO_CPRM_C2_AKE_PHASE2` flag must be passed to the `sfzcrypto_cprm_c2_derive` function to perform the second phase of the AKE protocol. In this phase, the SD Card's response on *Challenge1* is verified. If OK, a response on *Challenge2* (from the SD Card) is calculated and also the *SessionKey* value is derived and stored in the asset allocated for it.

7.3.4 C2 Key Derivation Example: Use the Session Key Asset for Bulk Decryption

```
    case C2_ASSET_DERIVE_STEP_TEST_KS:
      /* Test the SessionKey by decrypting some data in C2_DCBC mode. */
      {
        SfzCryptoCipherContext CiphContext = {0};
        SfzCryptoCipherKey CiphKey;

        CiphContext.fbmode = SFZCRYPTO_MODE_C_CBC;
        SFZCRYPTO_CIPHER_KEY_INIT_WITHOUT_KEYDATA(&CiphKey,
                                                  SFZCRYPTO_KEY_C2,
                                                  7);
        CiphKey.asset_id = cntxt_p->AssetArray[3];
        ret = sfzcrypto_symm_crypt(sfzcryptoctx_p,
                                   &CiphContext,
                                   &CiphKey,
                                   (uint8_t *)PATT_UNDER_KS,
                                   sizeof(PATT_UNDER_KS),
                                   OutputData,
                                   &OutputDataLen,
                                   SFZ_DECRYPT);
        ExpectedOutLen = sizeof(PATT_UNDER_KS);
        fTestPattern = true;
      }
      break;
```

The code above shows how to do a C-CBC mode decrypt with a C2 key asset intended for bulk data en/decryption. In this case, the C2 key asset used is the *SessionKey* as derived in 7.3.3.

7.3.5 C2 Key Derivation Example: Derive and Use a Content Key Asset for Bulk Decryption

```
    case C2_ASSET_DERIVE_STEP_MAKE_KT_1:
      /* Derive a Title/Content Key (without binding), using
         ENC_KT_1_CCI as the data that was supposedly read from the
         SD Card's Protected Area and store the ContentKey in
         AssetArray[0]. */
      ret = sfzcrypto_asset_alloc(sfzcryptoctx_p,
                                  SFZCRYPTO_POLICY_FUNCTION_ENCRYPT |
                                  SFZCRYPTO_POLICY_FUNCTION_DECRYPT |
                                  SFZCRYPTO_POLICY_ALGO_CIPHER_C2,
                                  8,
                                  &cntxt_p->AssetArray[0]);
      assert (ret == SFZCRYPTO_SUCCESS);
      ret = sfzcrypto_cprm_c2_derive(SFZCRYPTO_CPRM_C2_KZ_DERIVE,
                                     cntxt_p->AssetArray[2],
                                     SFZCRYPTO_ASSETID_INVALID,
                                     cntxt_p->AssetArray[0],
                                     ENC_KT_1_CCI,
                                     sizeof(ENC_KT_1_CCI),
                                     OutputData,
                                     &OutputDataLen);
      ExpectedOutLen = sizeof(ENC_KT_1_CCI);
      assert (OutputData[0] == 0x03);
      {
        int k;

        for (k = 1; k < 8; k++)
        {
          assert (OutputData[k] == 0x00);
        }
      }
      break;

    case C2_ASSET_DERIVE_STEP_TEST_KT_1:
      /* Test the ContentKey by decrypting some data in C2_DCBC mode. */
      {
        SfzCryptoCipherContext CiphContext = {0};
        SfzCryptoCipherKey CiphKey;

        CiphContext.fbmode = SFZCRYPTO_MODE_C_CBC;
        SFZCRYPTO_CIPHER_KEY_INIT_WITHOUT_KEYDATA(&CiphKey,
                                                  SFZCRYPTO_KEY_C2,
                                                  7);
        CiphKey.asset_id = cntxt_p->AssetArray[0];
        ret = sfzcrypto_symm_crypt(sfzcryptoctx_p,
                                   &CiphContext,
                                   &CiphKey,
                                   (uint8_t *)PATT_UNDER_KT_1,
                                   sizeof(PATT_UNDER_KT_1),
                                   OutputData,
                                   &OutputDataLen,
                                   SFZ_DECRYPT);
        ExpectedOutLen = sizeof(PATT_UNDER_KT_1);
        fTestPattern = true;
      }
      break;

    default:
```

```
        assert (false);
      break;
  } // switch
```

The code section above shows how to use CAL to derive and use a *ContentKey*. Note that:

- An asset with policy `SFZCRYPTO_POLICY_C2_DECRYPT` | `SFZCRYPTO_POLICY_ALGO_CIPHER_C2` must be created to hold the *ContentKey*. Adding `SFZCRYPTO_POLICY_C2_ENCRYPT` to the policy is allowed but typically unnecessary.
- The `SFZCRYPTO_CPRM_C2_KZ_DERIVE` flag must be passed to the `sfzcrypto_cprm_c2_derive` function to derive the *ContentKey* value. This function then also returns *CCI* or *UsageRule* information associated with the *ContentKey*. The *Crypto Module* does not interpret this information in any way, i.e. it is the responsibility of the entity using the *Crypto Module* (through CAL) to make sure that the CPRM-protected content is handled according to this usage information.
- The way to use the C2 asset with the *ContentKey* for (bulk) data decryption is basically the same as already shown for the *SessionKey*.
- The `SFZCRYPTO_CPRM_C2_KZ_DERIVE2` flag must be passed to the `sfzcrypto_cprm_c2_derive` function when additional ID binding was used for the *ContentKey*. In that case, the function uses the first 8 bytes of the *InputData* argument as the ID value to bind to.
- The CAL API also supports the SD-SD (Secure Digital - Separate Delivery) CPRM scenario. This requires an extra `SFZCRYPTO_CPRM_C2_KZ_DERIVE` derivation step, just before the *ContentKey* derivation step just shown, to setup a *UserKey* asset (with policy: `SFZCRYPTO_POLICY_C2_KZ_DERIVE` | `SFZCRYPTO_POLICY_ALGO_CIPHER_C2`).

7.3.6 C2 Key Derivation Example: Auxiliary Checking Code

```
assert (SFZCRYPTO_SUCCESS == ret);
assert (OutputDataLen == ExpectedOutLen);

if (fTestPattern)
{
  unsigned int n;

  for (n = 0; n < OutputDataLen; n++)
  {
    if (OutputData[n] != (uint8_t)(n & 0xFF))
      break;
  }
  assert (n == OutputDataLen);
}
```

The above code simply does some checks after each C2 key derivation/test step. After each C2 key test step, the `fTestPattern` flag is set, in which case the result of the decrypt operation is verified.

Attention: *Since this is just example code, not all assets that were used are freed. In production code, always make sure to free any assets that are no longer used. The Crypto Module only frees assets when requested to do so, and thus will eventually run out of space in the Asset Store if no-longer-needed assets are not freed.*

7.4 C2 Cipher Examples

The following example code is provided to show the following:

- How to use CAL to verify the C2 block cipher test vectors for C2_E, C2_D, C2_ECBC, C2_DCBC and C2_H given by the 4C Entity in "C2withFacsimileSBox_010111.txt", see [C2-FACSIMILE].
- How to perform a multipart C2 bulk en/decrypt in C-CBC mode.
- That although the effective length of a C2 key is 7 bytes (56 bits), C2 assets must have a size of 8 bytes. When loading a C2 asset with plain key bytes, a dummy byte must be pre-pended to the actual key bytes. This does not apply to C2 *Device Keys*. C2 *Device Keys* are stored in a 12-byte asset, as described in the example code given in 7.2.

Since the state of an unfinished C-CBC mode operation consists of sensitive data, it may only be saved in an asset. This constitutes a major difference between C2's C-CBC mode and the regular CBC mode supported by most other block ciphers. The state of a regular CBC mode operation consists of the IV, which is (in general) not handled as sensitive data.

```
static void
C2CipherExample(void)
{
/* ECB mode test vectors, taken from "C2withFacsimileSbox_010111.txt" */
static uint8_t C2_ECB_KEY[] = {
  0x5E, 0x91, 0x6A, 0xEF, 0x34, 0x1F, 0xA3 };
static uint8_t C2_ECB_PTX[] = {
  0x89, 0x06, 0x7f, 0x2b, 0xe2, 0xa6, 0x0d, 0x6f };
static uint8_t C2_ECB_CTX[] = {
  0x8f, 0xe6, 0x5f, 0xe4, 0xf7, 0xba, 0x80, 0x05 };
/* C-CBC mode test vectors, taken from "C2withFacsimileSbox_010111.txt"
   !! NOTE -- C2_CBC_KEY starts with a dummy byte here,       !!
   !!         to simplify code to load this key in an asset.  !! */
static uint8_t C2_CBC_KEY[] = {
  0xDC, 0x7c, 0xb3, 0xc4, 0xdb, 0x09, 0x47, 0x13 };
static uint8_t C2_CBC_PTX[] = {
  0xa2, 0x46, 0x32, 0xd8, 0x24, 0x32, 0x08, 0x44,
  0x7d, 0x81, 0x11, 0xdf, 0x8c, 0xe2, 0x41, 0x72,
  0x76, 0xbe, 0x42, 0xd7, 0x0d, 0xb1, 0x44, 0x18 };
static uint8_t C2_CBC_CTX[] = {
  0x50, 0xfc, 0x09, 0xd1, 0x69, 0x1c, 0x51, 0x02,
  0x54, 0x1d, 0x32, 0x2f, 0x68, 0xe7, 0xfd, 0x79,
  0x91, 0xa8, 0x0c, 0x3d, 0x9d, 0x9f, 0x31, 0x0d };
/* C2_H test vectors, taken from "C2withFacsimileSbox_010111.txt" */
static uint8_t C2_H0_SECRET [] = {
  0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF };
static uint8_t C2_H_MSG[] = {
  0xa2, 0x46, 0x32, 0xd8, 0x24, 0x32, 0x08, 0x44,
  0x7d, 0x81, 0x11, 0xdf, 0x8c, 0xe2, 0x41, 0x72,
  0x76, 0xbe, 0x42, 0xd7, 0x0d, 0xb1, 0x44, 0x18 };
static uint8_t C2_H_OUT[] = {
  0x29, 0x9e, 0xb6, 0xdb, 0xee, 0xe7, 0x50, 0x99 };

SfzCryptoContext *sfzcryptoctx_p = sfzcrypto_context_get();
SfzCryptoCipherMacContext CiphMacContext;
SfzCryptoCipherContext CiphContext;
SfzCryptoCipherKey CiphKey;
SfzCryptoCipherKey HashKey;
SfzCryptoAssetId KeyAssetId, TempAssetId;
uint8_t OutputData[4*8];
size_t OutputDataLen;
```

```
SfzCryptoStatus ret;

/* Test C2_E with the "Facsimile" test vectors. */
c_memset(&CiphContext, 0, sizeof(CiphContext));
c_memset(&CiphKey, 0, sizeof(CiphKey));
CiphContext.fbmode = SFZCRYPTO_MODE_ECB;
SFZCRYPTO_CIPHER_KEY_INIT(&CiphKey,
                          SFZCRYPTO_KEY_C2,
                          C2_ECB_KEY,
                          sizeof(C2_ECB_KEY));
OutputDataLen = sizeof(OutputData);
/* Note that the OutputData buffer is over-sized. */
ret = sfzcrypto_symm_crypt(sfzcryptoctx_p,
                           &CiphContext,
                           &CiphKey,
                           C2_ECB_PTX, sizeof(C2_ECB_PTX),
                           OutputData, &OutputDataLen,
                           SFZ_ENCRYPT);
assert (ret == SFZCRYPTO_SUCCESS);
assert (OutputDataLen == sizeof(C2_ECB_CTX));
/* Note that OutputDataLen was adjusted. */
assert (c_memcmp(OutputData, C2_ECB_CTX, OutputDataLen) == 0);

/* Test C2_ECBC with the "Facsimile" test vectors. */
c_memset(&CiphContext, 0, sizeof(CiphContext));
c_memset(&CiphKey, 0, sizeof(CiphKey));
CiphContext.fbmode = SFZCRYPTO_MODE_C_CBC;
SFZCRYPTO_CIPHER_KEY_INIT(&CiphKey,
                          SFZCRYPTO_KEY_C2,
                          C2_CBC_KEY + 1, /* Skip dummy byte */
                          sizeof(C2_CBC_KEY) - 1);
OutputDataLen = sizeof(OutputData);
ret = sfzcrypto_symm_crypt(sfzcryptoctx_p,
                           &CiphContext,
                           &CiphKey,
                           C2_CBC_PTX, sizeof(C2_CBC_PTX),
                           OutputData, &OutputDataLen,
                           SFZ_ENCRYPT);
assert (ret == SFZCRYPTO_SUCCESS);
assert (OutputDataLen == sizeof(C2_CBC_CTX));
assert (c_memcmp(OutputData, C2_CBC_CTX, OutputDataLen) == 0);

/* To demonstrate multipart decryption, assets must be used. */

/* Create and load an (8-byte) asset with the C-CBC mode test key. */
ret = sfzcrypto_asset_alloc(sfzcryptoctx_p,
                            SFZCRYPTO_POLICY_FUNCTION_DECRYPT |
                            SFZCRYPTO_POLICY_ALGO_CIPHER_C2,
                            8, &KeyAssetId);
assert (ret == SFZCRYPTO_SUCCESS);
ret = sfzcrypto_asset_load_key(sfzcryptoctx_p,
                               KeyAssetId,
                               C2_CBC_KEY, /* Include dummy byte */
                               sizeof(C2_CBC_KEY));
assert (ret == SFZCRYPTO_SUCCESS);

/* Create a temporary asset to store the C2 continuation state,
```

```
   needed for multipart C2 en/decryption. */
ret = sfzcrypto_asset_alloc_temporary(sfzcryptoctx_p,
                                      SFZCRYPTO_KEY_C2,
                                      SFZCRYPTO_MODE_C_CBC,
                                      0, /* Dummy HashAlgo */
                                      KeyAssetId,
                                      &TempAssetId);
assert (ret == SFZCRYPTO_SUCCESS);

/* Test C2_DCBC with the first 8 bytes of "Facsimile" ptx/ctx data. */
c_memset(&CiphContext, 0, sizeof(CiphContext));
c_memset(&CiphKey, 0, sizeof(CiphKey));
CiphContext.fbmode = SFZCRYPTO_MODE_C_CBC;
CiphContext.iv_loc = SFZ_TO_ASSET;     /* Start multipart operation */
CiphContext.iv_asset_id = TempAssetId;
SFZCRYPTO_CIPHER_KEY_INIT_WITHOUT_KEYDATA(&CiphKey,
                                          SFZCRYPTO_KEY_C2,
                                          sizeof(C2_CBC_KEY) - 1);
                                          /* Exclude dummy byte */
CiphKey.asset_id = KeyAssetId;
OutputDataLen = sizeof(OutputData);
ret = sfzcrypto_symm_crypt(sfzcryptoctx_p,
                           &CiphContext,
                           &CiphKey,
                           C2_CBC_CTX, 8,
                           OutputData, &OutputDataLen,
                           SFZ_DECRYPT);
assert (ret == SFZCRYPTO_SUCCESS);
assert (OutputDataLen == 8);
assert (c_memcmp(OutputData, C2_CBC_PTX, OutputDataLen) == 0);

/* To 'destroy' the state of the C2 engine, test C2_H with the
   "Facsimile" test vectors now. Resume the multipart C2_DCBC test
   after that. */
c_memset(&CiphMacContext, 0, sizeof(CiphMacContext));
c_memset(&HashKey, 0, sizeof(HashKey));
CiphMacContext.fbmode = SFZCRYPTO_MODE_C2_H;
SFZCRYPTO_CIPHER_KEY_INIT(&HashKey,
                          SFZCRYPTO_KEY_C2,
                          C2_H0_SECRET,
                          sizeof(C2_H0_SECRET));
ret = sfzcrypto_cipher_mac_data(sfzcryptoctx_p,
                                &CiphMacContext,
                                &HashKey,
                                C2_H_MSG, sizeof(C2_H_MSG),
                                true, true);
assert (ret == SFZCRYPTO_SUCCESS);
assert (c_memcmp(CiphMacContext.iv, C2_H_OUT, sizeof(C2_H_OUT)) == 0);

/* Resume the multipart C2_DCBC test, i.e. decrypt the last 16 bytes. */
OutputDataLen = sizeof(OutputData);

/* Verify that a multipart operation is in progress: */
assert (CiphContext.iv_loc == SFZ_IN_ASSET);
ret = sfzcrypto_symm_crypt(sfzcryptoctx_p,
                           &CiphContext,
                           &CiphKey,
```

```
                           C2_CBC_CTX + 8, 16,
                           OutputData, &OutputDataLen,
                           SFZ_DECRYPT);
assert (ret == SFZCRYPTO_SUCCESS);
assert (OutputDataLen == 16);
assert (c_memcmp(OutputData, C2_CBC_PTX + 8, OutputDataLen) == 0);

/* Free the assets that were used. */
ret = sfzcrypto_asset_free(sfzcryptoctx_p, TempAssetId);
assert (ret == SFZCRYPTO_SUCCESS);
ret = sfzcrypto_asset_free(sfzcryptoctx_p, KeyAssetId);
assert (ret == SFZCRYPTO_SUCCESS);
}
```

A Conventions, References and Compliances

A.1 *Conventions Used in this Manual*

A.1.1 Acronyms

3DES	Triple DES
AES	Advanced Encryption Standard
AKE	Authentication and Key Exchange
API	Application Programming Interface
ARC4	Alleged Ron’s Code 4
ARCFOUR	Alleged Ron’s Code 4
CAL	Cryptographic Abstraction Layer
CBC	Cipher-Block Chaining
CBCMAC	Cipher-Block Chaining-based Message Authentication Code
C-CBC	Converted Cipher-Block Chaining
CM	Crypto Module
CMAC	Cipher-based Message Authentication Code
CPRM	Content Protection for Recordable Media
CTR	CounTeR
DES	Data Encryption Standard
DH	Diffie Hellman
DMA	Direct Memory Access
DSA	Digital Signature Algorithm
ECB	Electronic CodeBook
ECDH	Elliptic Curve Diffie Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
HMAC	Hash-based Message Authentication Code
ICM	Integer Counter Mode
IV	Initialization Vector
KDK	Key Derivation Key
KEK	Key Encryption Key
MAC	Message Authentication Code
MKB	Media Key Block
NIST	National Institute of Standards and Technology
NVM	Non Volatile Memory
PKCS	Public Key Cryptography Standard
PSS	Probabilistic Signature Scheme
RK	Root Key
RSA	Rivest-Shamir-Adleman
S2V	String to Vector
SD	Secure Digital
SHA	Secure Hash Algorithm

A.2 *References*

A.2.1 Test Vectors from NIST's Toolkit Examples

The list of open standards, Internet Engineering Task Force (IETF) and National Institute of Standards and Technology (NIST) standards referenced:

[Toolkit Examples]
Overview of example algorithms
http://csrc.nist.gov/groups/ST/toolkit/examples.html

[Toolkit Examples, SHA1]
http://csrc.nist.gov/groups/ST/toolkit/documents/Examples/SHA1.pdf

[Toolkit Examples, HMAC-SHA256]
http://csrc.nist.gov/groups/ST/toolkit/documents/Examples/HMAC_SHA256.pdf

[Toolkit Examples, AES-CBC]
http://csrc.nist.gov/groups/ST/toolkit/documents/Examples/AES_CBC.pdf

[Toolkit Examples, AES-CMAC]
http://csrc.nist.gov/groups/ST/toolkit/documents/Examples/AES_CMAC.pdf

A.2.2 Test Vectors for Camellia & MULTI2

A list of test vectors for Camellia and MULTI2:

[RFC-3713]
RFC3713 - Camellia Encryption Algorithm
http://www.ietf.org/rfc/rfc3713.txt

[RFC-5528]
RFC5528 - Camellia Counter Mode and Camellia Counter with CBC-MAC Mode Algorithms
http://tools.ietf.org/html/rfc5528

[MULTI2]
ISO9979 / 0009 Algorithm Register Entry
http://www.isg.rhul.ac.uk/~cjm/ISO-register/0009.pdf

A.2.3 Test Vector for the C2 Cipher and CPRM

A list of specifications on C2 and CPRM:

[C2-FACSIMILE]
C2 Facsimile Secret Constants and Test Vectors
http://www.4centity.com/facsimile.html
http://www.4centity.com/docs/C2_Facsimile_S-Box.txt
http://www.4centity.com/docs/C2withFacsimileSbox_010111.txt

[C2-MKB-SD-CARDS]
http://www.4centity.com/facsimile2.html
http://www.4centity.com/docs/CPRM_SD_Card.zip

(End of Document)