User's Manual

CONFIDENTIAL

Crypto Cell Rich Execution Environment Driver for Linux

User's Manual: Software

R-Car H3/M3/M3N/E3/D3 Series

Renesas Electronics
www.renesas.com

Rev.3.00 Dec. 2021

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan

www.renesas.com

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/.

Trademarks

How to Use This Manual

- **[Readers]**
  This manual is intended for engineers who develop products which use the R-Car H3/M3/M3N/E3/D3 processor.

- **[Purpose]**
  This manual is intended to give users an understanding of the functions of the R-Car H3/M3/M3N/E3/D3 processor device driver and to serve as a reference for developing hardware and software for systems that use this driver.

- **[How to Read This Manual]**
  It is assumed that the readers of this manual have general knowledge in the fields of electrical
  — Engineering, logic circuits, microcontrollers, and Linux.
  → Read this manual in the order of the CONTENTS.
  — To understand the functions of a multimedia processor for R-Car H3/M3/M3N/E3/D3
  → See the R-Car H3/M3/M3N/E3/D3 User's Manual.
  — To know the electrical specifications of the multimedia processor for R-Car H3/M3/M3N/E3/D3
  → See the R-Car H3/M3/M3N/E3/D3 Data Sheet.

- **[Conventions]**
  The following symbols are used in this manual.
  Data significance: Higher digits on the left and lower digits on the right
  Note: Footnote for item marked with Note in the text
  Caution: Information requiring particular attention
  Remark: Supplementary information
  Numeric representation: Binary ... ××××, 0b××××, or ××××B
  Decimal ... ××××
  Hexadecimal ... 0x×××× or ××××H
  Data type: Double word ... 64 bits
  Word … 32 bits
  Half word ... 16 bits
  Byte ... 8 bits

Table of Contents

1. Overview

1.1 Overview

R-Car H3/M3/M3N/E3/D3 supports Secure Engine that provides the security infrastructure and services for the SoC.
Secure Engine contains Secure Core, PKA Core and Public Core. The Secure Core and PKA core will be controlled in TEE side (OP-TEE), and Public core will be controlled in REE side (Linux). In Linux, CCREE Driver is implemented for controlling Public core. This manual explains the Public Core and CCREE Driver functionality, how to build and use CCREE Driver.

1.2 Function

Table 1-1 shows algorithms that Public Core support. The specification of these algorithms refer to Linux kernel source tree under “Documentation/crypto (document [7]). These algorithms can be used by Kernel Crypto API, which provides cryptographic functions to the Linux kernel internal processing. User space application cannot access these APIs directly. User can access these APIs only by kernel program which execute in kernel space and call Linux Kernel Crypto API.

Table 1-1 List of the algorithm provided Public Core

Cryptographic algorithm	Details (Cryptographic name that registered Linux kernel)
AES	ecb(aes), cbc(aes), ctr(aes), ofb(aes), xts(aes), cts(cbc(aes)), cts1(cbc(aes)) Note) xts(aes) supports only data sizes that are multiples of 16bytes between 32 and 8192. Note) ecb(aes), cbc(aes), ctr(aes), ofb(aes), cts(cbc(aes)), cts1(cbc(aes)) supports only data size >=16.
MULTI2	cbc(multi2), ofb(multi2) Note) multi2 do not have native support in the Linux kernel crypto API. This is an additional support.
HASH	md5, sha1, sha224, sha256
HMAC	hmac(md5), hmac(sha1), hmac(sha224), hmac(sha256)
AES MAC	xcbc(aes), cmac(aes)
Authenticated Encryption	authenc(hmac(sha1),cbc(aes)), authenc(hmac(sha256),cbc(aes)), authenc(xcbc(aes),cbc(aes)), authenc(hmac(sha1),rfc3686(ctr(aes))), authenc(hmac(sha256),rfc3686(ctr(aes))), authenc(xcbc(aes),rfc3686(ctr(aes))), ccm(aes), rfc4309(ccm(aes)), gcm(aes), rfc4106(gcm(aes)), rfc4543(gcm(aes))
Secure Key Algorithms	Secure AES: cbc(saes), ctr(saes), cts(cbc(saes)), cts1(cbc(saes)) Secure MULTI2: cbc(smulti2), ofb(smulti2) Secure copy: sbypass NOTE) This function needs to SS6.3-Secure Driver.

Note) cts1(cbc(aes)), cts1(cbc(saes)) are just for backward compatible use. For new implementation, cts(cbc(aes)) and cts(cbc(saes)) are recommended for using.

1.3 References

1.3.1 Related Document

The following table shows the related/reference documents.

Table 1-2 Related Document

No.	Issue	Title	Edition
1	NIST	NIST SP 800-38A Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a-add.pdf	-
2	NIST	NIST SP 800-38D Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf	-
3	Renesas Electronics	Linux Interface Specification Yocto recipe Start-Up Guide	#0
4	Renesas Electronics	Security Board Support Package	#0
5	GlobalPlatform	TEE Internal Core API Specification	Ver.1.1
6	Renesas Electronics	SS6.3-Secure Driver for OP-TEE User's Manual	#0
7	Linux kernel	Linux Kernel Crypto API https://www.kernel.org/doc/html/v5.4/crypto/index.html	
8	Community	https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation	
9	Renesas Electronics	Android Board Support Package Specification Release Note: Software	Android 11
10	Renesas Electronics	Memory Manager for Linux User's Manual: Software	Rev.2.50

Note) #0 : This manual refers to the latest edition.

1.3.2 Related Site for original software

Refer to the Chapter 1.3.1 of [4].

1.4 Restrictions

There is no restriction for this module.

2. Terminology and abbreviation

The following table shows the terminology and the abbreviation.

Table 2-1 Terminology

Terms	Explanation
Exception Levels (EL0/EL1/EL3)	The ARMv8-A architecture defines a set of Exception levels, EL0 to EL3, where: • If ELn is the Exception level, increased values of *n* indicate increased software execution privilege. • Execution at EL0 is called unprivileged execution. • EL2 provides support for virtualization of Non-secure operation. • EL3 provides support for switching between two Security states, Secure state and Non-secure state.
Secure World	It is one of the security states that defined ARMv8-A architecture. When in this state, the CPU can access both the Secure and Non-secure space.
Normal World	It is one of the security states that defined ARMv8-A architecture. When in this state, the CPU can access only Non-secure space.
SMC	Secure Monitor Call. An ARM assembler instruction that causes an exception that is taken synchronously into EL3.
Client Application (CA)	An application running outside of the Trusted Execution Environment making use of the TEE Client API to access facilities provided by Trusted Applications inside the Trusted Execution Environment.
Trusted Application (TA)	An application running inside the Trusted Execution Environment that provides security related functionality to Client Applications outside of the TEE or to other Trusted Applications inside the Trusted Execution Environment.
Provisioning Key	This key is AES 128-bit key unique per OEM (Original Equipment Manufacturer) and device series and used for protection of OEM secret assets.
Session Key	This key is AES 128-bit key that generate randomly at least once in a cold boot cycle. NOTE) When Suspend To RAM is executed, this key is not changed because the key is encrypted with Provisioning Key and saved.
Secure Key Package	This package contains parameters used in Secure Key Algorithm protected by Session Key. Refer to Chapter 7 about detail of Secure Key Algorithm. NOTE) When Suspend To RAM is executed, this package is not changed because the Session Key is saved. But when cold boot cycle is executed, this package needs to be generated again because Session Key is changed.

Table 2-2 Abbreviation and acronym

Terms	Explanation
ECDSA	Elliptic Curve Digital Signature Algorithm
ECDH	Elliptic Curve Diffie-Hellman
MAC	Message Authentication Code
Ksses	Session Key
Authenc	Authenticated Encryption
TEE	Trusted Execution Environment
REE	Rich Execution Environment
CCREE	Crypto Cell for REE
CCREE Driver	Crypto Cell Rich Execution Environment Driver
SKP	Secure Key Package

3. Operating Environment

3.1 Hardware Environment

The following table lists the hardware needed to use this function.

Table 3-1 Hardware environment (R-Car H3/M3/M3N/E3/D3)

Name	Explanation
Evaluation Board	R-Car H3 SiP System Evaluation Board (Salvator-X/Salvator-XS) Renesas Electronics R-Car M3 SiP System Evaluation Board (Salvator-X/Salvator-XS) Renesas Electronics R-Car M3N SiP System Evaluation Board Salvator-XS Renesas Electronics R-Car E3 SiP System Evaluation Board Ebisu Renesas Electronics R-Car D3 SiP System Evaluation Board Draak Renesas Electronics
Host PC 1	It is used as debugging environment. Terminal software is executed.
Host PC 2 (Linux)	TFTP server software It is used when HyperFlash is written by U-Boot or Image is downloaded.
	NFS server software It is used when File system is mounted by NFS.
	For build environment on HostPC, please set-up same environment as for build Yocto as in Reference document No.3.

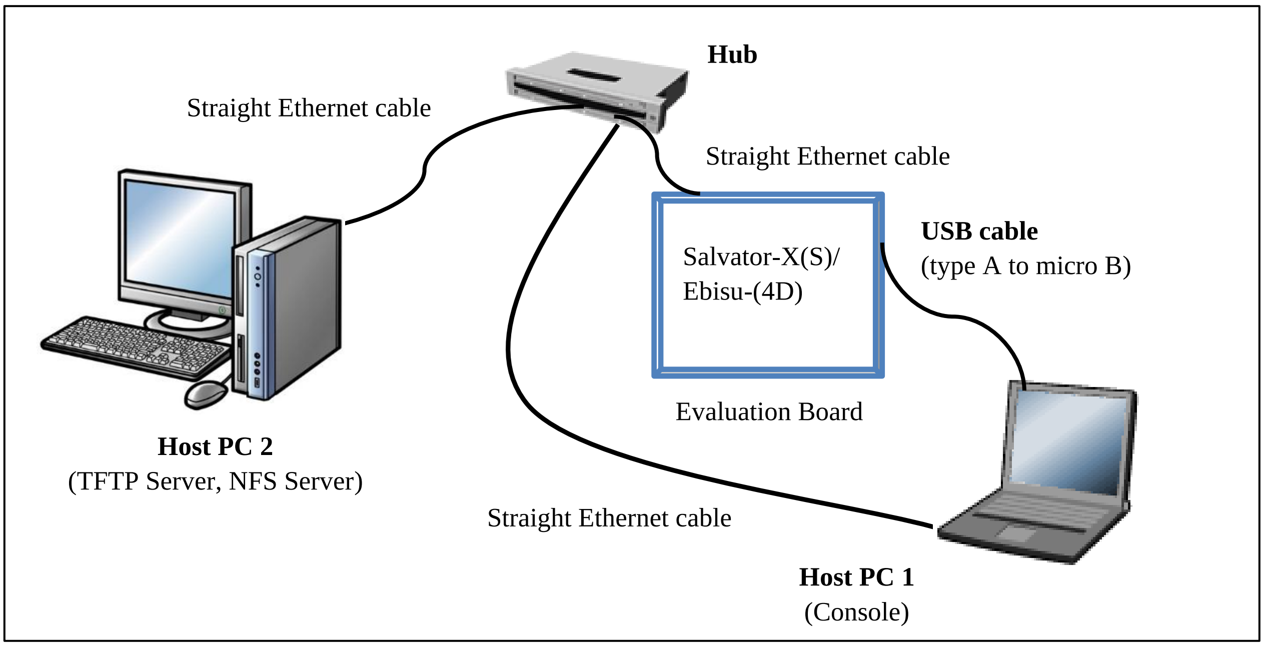
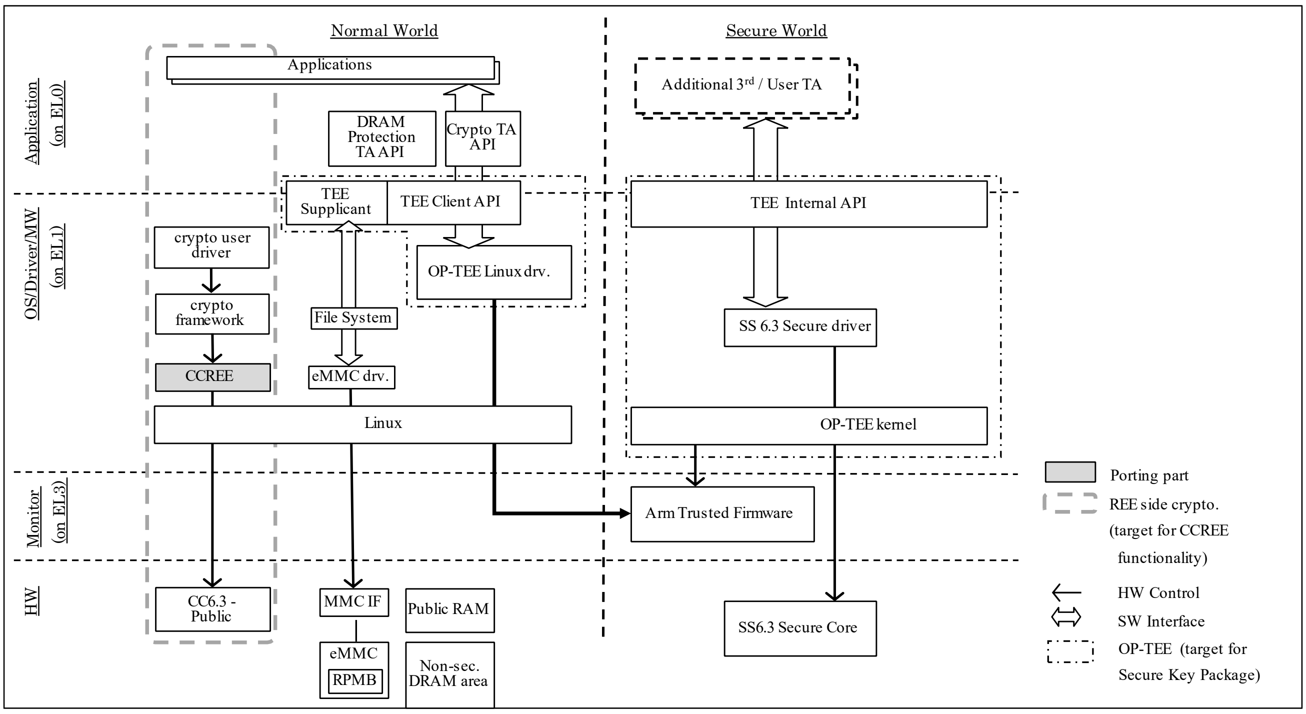


Figure 3-1 Recommended Environment

3.2 Module Configuration

This section explains software relationship and configuration.

Figure 3-2 Cryptographic support in R-Car Series, 3rd Generation Linux system



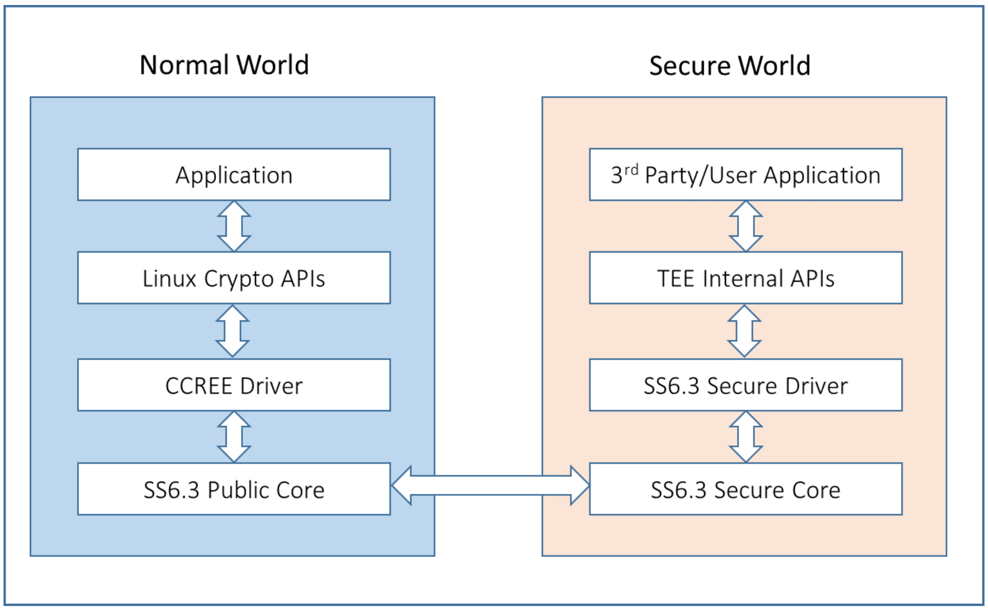
The figure shows whole software component that related to cryptographic functionality support in R-Car Series, 3rd Generation Linux system (including TEE and REE side.) This document mainly focuses on REE side (inside red box) and a part of TEE side (relate to secure key package feature.)

In REE side, target of this document is CCREE Driver. CCREE Driver is a device driver that is implemented base on and provides cryptographic methods to crypto framework in Linux kernel. And its processing is in kernel space; application layer cannot access to it directly. For using CCREE Driver, user can access only through crypto kernel APIs from other user driver. From application, user has to use crypto application framework (e.g. openssl) and appropriate support user driver (e.g. cryptodev-linux) to call to crypto framework APIs.

3.2.1 CCREE Driver (for secure key package feature)

CCREE Driver supports to communicate between Public Core and TEE Internal API. Figure 3.3 shows details of module configuration to access Public Core.

Figure 3-3 Detail of module configuration in Normal World



CCREE Driver can be used within a TrustZone system, where have two operating environments an OP-TEE kernel as Secure World (TEE side), and Linux kernel as Normal World (REE side). Within this system, the OP-TEE kernel controls an instance of SS6.3-Secure Driver, while the Linux kernel controls CCREE Driver.

CCREE Driver is used as a platform device driver and this driver provides functions to Linux Cryptographic API by registering services to Linux kernel as cryptographic algorithms. All CCREE Driver resources are allocated to include DMA mapped buffers and an interrupt resource.

A unique feature to CCREE Driver is its ability to securely process encrypted and signed configuration packages produced by SS6.3-Secure Driver in the context of the OP-TEE kernel. The function is called Secure Key Package and this function communicates with Public Core and Secure Core, and cryptographic processing is done with Secure Core.

The feature of this function is that World Switch is unnecessary for the movement of the worlds, so overhead can be reduced, and it is possible to process large data directly and process without using shared memory. Refers to Chapter 7 for how to use this function.

3.3 State Transition Diagram

There is no state transition diagram in this module.

3.4 DMA-resources

Crypto Public Engine Core has internal DMAC module inside.

4. External Interface

The following is components included in this package.

4.1 Device Node

On sysfs file system, there is below node for checking the current supported crypto drivers. It will list all driver/algorithm that can be used in system, also include all CCREE Driver supported crypto algorithm. If an algorithm is also provided by other drivers, please check to ensure CCREE is highest priority.

```
$ cat /proc/crypto           # show all current supported crypto drivers
```

4.2 External Interface

There is no external interface CCREE Driver. This module is implemented as backend of Linux Crypto framework. So, this chapter will briefly describe Linux Crypto framework APIs.

4.2.1 Linux Crypto framework/ Message digest APIs

No.	API name	Description
1	struct crypto_ahash * crypto_alloc_ahash(const char * *alg_name*, u32 *type*, u32 *mask*)	allocate ahash cipher handle
2	struct ahash_request * ahash_request_alloc(struct crypto_ahash * *tfm*, gfp_t *gfp*)	allocate request data structure
3	void ahash_request_set_callback(struct ahash_request * *req*, u32 *flags*, crypto_completion_t *compl*, void * *data*)	set asynchronous callback function
4	void ahash_request_set_crypt(struct ahash_request * *req*, struct scatterlist * *src*, u8 * *result*, unsigned int *nbytes*)	set data buffers
5	int crypto_ahash_init(struct ahash_request * *req*)	(re)initialize message digest handle
6	static inline int crypto_ahash_update(struct ahash_request *req)	add data to message digest for processing
7	int crypto_ahash_setkey(struct crypto_ahash * *tfm*, const u8 * *key*, unsigned int *keylen*)	set key for cipher handle
8	int crypto_ahash_finup(struct ahash_request * *req*)	update and finalize message digest
9	int crypto_ahash_digest(struct ahash_request * *req*)	calculate message digest for a buffer
10	int crypto_ahash_export(struct ahash_request * *req*, void * *out*)	extract current message digest state
11	int crypto_ahash_import(struct ahash_request * *req*, const void * *in*)	import message digest state

For detail description about parameters/structures and full description of these APIs, please refer to Linux Crypto framework / Asynchronous Message Digest APIs

4.2.1.1 Mapping and usage of Message digest API's parameters and algorithms

No.	Algorithm/template type	Parameter value		Remarks
		alg_name	**keylen *(bytes)***	
1	MD5 hash	md5	-	
2	SHA1 hash	sha1	-	
3	SHA256 hash	sha256	-	
4	SHA224 hash	sha224	-	
5	HMAC base on MD5	hmac(md5)	-	
6	HMAC base on SHA-1	hmac(sha1)	-	
7	HMAC base on SHA-224	hmac(sha224)	-	
8	HMAC base on SHA-256	hmac(sha256)	-	
9	AES MAC (xcbc mode)	xcbc(aes)	16, 24, 32	
10	AES MAC (cmac mode)	cmac(aes)	16, 24, 32	

4.2.2 Linux Crypto framework/ Symmetric ciphers APIs

No.	API name	Description
1	struct crypto_skcipher *crypto_alloc_skcipher(const char *alg_name, u32 type, u32 mask)	allocate symmetric key cipher handle
2	struct skcipher_request * skcipher_request_alloc(struct crypto_skcipher * *tfm*, gfp_t *gfp*)	allocate request data structure
3	void skcipher_request_set_callback(struct skcipher_request * *req*, u32 *flags*, crypto_completion_t *compl*, void * *data*)	set asynchronous callback function
4	int crypto_skcipher_setkey(struct crypto_skcipher * *tfm*, const u8 * *key*, unsigned int *keylen*)	set key for cipher
5	void skcipher_request_set_crypt(struct skcipher_request * *req*, struct scatterlist * *src*, struct scatterlist * *dst*, unsigned int *cryptlen*, void * *iv*)	set data buffers
6	int crypto_skcipher_encrypt(struct skcipher_request * *req*)	encrypt plaintext
7	int crypto_skcipher_decrypt(struct skcipher_request * *req*)	decrypt ciphertext

For detail description about parameters/structures and full description of these APIs, please refer to Linux Crypto framework / skcipher APIs.

4.2.2.1 Mapping and usage of Symmetric cipher API's parameters and algorithms

No.	Algorithm/template type	Parameter value		Remarks
		alg_name	keylen (bytes)	
1	AES algorithm/ECB mode	ecb(aes)	16, 24, 32	
2	AES algorithm/CBC mode	cbc(aes)	16, 24, 32	
3	AES algorithm/CTR mode	ctr(aes)	16, 24, 32	
4	AES algorithm/OFB mode	ofb(aes)	16, 24, 32	
5	AES algorithm/XTS mode	xts(aes)	32, 64	
6	AES algorithm/CTS(CBC) mode	cts(cbc(aes))	16, 24, 32	
7	AES algorithm/CTS1(CBC) mode	cts1(cbc(aes))	16, 24, 32	
8	MULTI2 algorithm/CBC mode	cbc(multi2)		41 (data key: 8B, systemkey: 32B, Round number: 1B (8, 64,128 round))
9	MULTI2 algorithm/OFB mode	ofb(multi2)		41 (data key: 8B, systemkey: 32B, Round number: 1B (8, 64,128 round))
10	SKP AES algorithm/CBC mode	cbc(saes)	16, 32	
11	SKP AES algorithm/CTR mode	ctr(saes)	16, 32	
12	SKP AES algorithm/CTS(CBC) mode	cts(cbc(saes))	16, 32	
13	SKP AES algorithm/CTS1(CBC) mode	cts1(cbc(saes))	16, 32	
14	SKP bypass	sbypass	16, 32	
15	SKP MULTI2 algorithm/CBC mode	cbc(smulti2)	41	(data key: 8B, systemkey: 32B, Round number: 1B (8, 64,128 round))
16	SKP MULTI2 algorithm/MULTI2 mode	ofb(smulti2)	41	(data key: 8B, systemkey: 32B, Round number: 1B (8, 64,128 round))

Note: description of abbreviation/acronym, please refer to document no.8 in chapter 1.3.1 Related Document.

4.2.3 Linux Crypto framework/ AEAD APIs

No.	API name	Description
1	struct crypto_aead * crypto_alloc_aead(const char * *alg_name*, u32 *type*, u32 *mask*)	allocate AEAD cipher handle
2	int crypto_aead_setkey(struct crypto_aead * *tfm*, const u8 * *key*, unsigned int *keylen*)	set key for cipher
3	int crypto_aead_setauthsize(struct crypto_aead * *tfm*, unsigned int *authsize*)	set authentication data size
4	struct aead_request * aead_request_alloc(struct crypto_aead * *tfm*, gfp_t *gfp*)	allocate request data structure
5	void aead_request_set_callback(struct aead_request * *req*, u32 *flags*, crypto_completion_t *compl*, void * *data*)	set asynchronous callback function
6	int crypto_aead_setkey(struct crypto_aead * *tfm*, const u8 * *key*, unsigned int *keylen*)	set key for cipher
7	void aead_request_set_crypt(struct aead_request * *req*, struct scatterlist * *src*, struct scatterlist * *dst*, unsigned int *cryptlen*, u8 * *iv*)	set data buffers
8	void aead_request_set_ad(struct aead_request * *req*, unsigned int *assoclen*)	set associated data information
9	int crypto_aead_encrypt(struct aead_request * *req*)	encrypt plaintext
10	int crypto_aead_decrypt(struct aead_request * *req*)	decrypt ciphertext

For detail description about parameters/structures and full description of these APIs, please refer to Linux Crypto Framework / AEAD APIs

4.2.3.1 Mapping and usage of Symmetric cipher API’s parameters and algorithms

No.	**Algorithm/template type**	**Parameter value**		**Remarks**
		alg_name	***keylen (bytes)***	
1	AEAD template by conjunction of HMAC(SHA1) [for MAC] and CBC(AES) [for encryption]	authenc(hmac(sha1),cbc(aes))	16, 24, 32	
2	AEAD template by conjunction of HMAC(SHA256) and CBC(AES)	authenc(hmac(sha256),cbc(aes))	16, 24, 32	
3	AEAD template by conjunction of XCBC(AES) [for MAC] and CBC(AES) [for encryption]	authenc(xcbc(aes),cbc(aes))	16, 24, 32	
4	AEAD template by conjunction of HMAC(SHA1) [for MAC] and RFC3686(CTR(AES)) [for encryption]	authenc(hmac(sha1),rfc3686(ctr(aes)))	16, 24, 32	
5	AEAD template by conjunction of HMAC(SHA256) [for MAC] and RFC3686(CTR(AES)) [for encryption]	authenc(hmac(sha256),rfc3686(ctr(aes)))	16, 24, 32	
6	AEAD template by conjunction of XCBC(AES) [for MAC] and RFC3686(CTR(AES)) [for encryption]	authenc(xcbc(aes),rfc3686(ctr(aes)))	16, 24, 32	
7	Counter with CBC-MAC (conjunction of CBC(AES) [for MAC] and CTR(AES) [for encryption]	ccm(aes)	16, 24, 32	
8	The Use of CCM(AES) follow rfc4309 standard	rfc4309(ccm(aes))	16B + 3B, 24B + 3B, 32B + 3B	B: bytes
9	Galois/Counter Mode (using AES cipher)	gcm(aes)	16, 24, 32	
10	The Use of GCM(AES) follow rfc4106 standard	rfc4106(gcm(aes))	16B + 4B, 24B + 4B, 32B + 4B	
11	The Use of CCM(AES) follow rfc4543 standard	rfc4543(gcm(aes))	16B + 4B, 24B + 4B, 32B + 4B	

4.3 Definitions

No definitions that modules define.

4.4 Structure

No structures that modules are define.

4.5 Difference of Specification

Basically, CCREE Driver will provide functionality as description of Linux Crypto API, but there are some differences will be shown in this chapter. The following limitations apply to all crypto algorithms and to any use of the driver:

Cache coherency in the kernel and user mode

User data that shares the same cache line with an output buffer, must not be accessed (read or write) prior to the return of the driver completion callback. Otherwise either the user data or the driver output buffer may not be cache-coherent.

Scatter/gather list size in the kernel mode

The size of the scatter/gather list representing a single virtual data buffer is limited to 128 entries. Fragment count higher than 128 will result in error returned.

The size of the scatter/gather list representing a single virtual associated-data buffer is limited to 8 entries. Fragment count higher than 8 will result in error returned.

Fragment size in the kernel mode

The size of each physical fragment is limited to 64KB. Physical fragments larger than 64KB are broken into smaller fragments <= 64KB, and each of them is counted as an additional fragment in the data fragment count (item 1) or the associated data fragment count.

For example**:**

A single physically-contiguous data buffer with size [8MB+64KB], will be broken by the driver into 129 fragments of 64KB. Since the fragment count is higher than 128, the buffer will be rejected.

A single physically-contiguous associated-data buffer for authenc algorithm with size [516KB], will be broken by the driver into 9 fragments (8 fragments of 64KB and one of 4KB). Since the fragment count is higher than 8, which is the associated data limit, the buffer will be rejected.

If more data is to be sent to this driver, user must slice the input into segments with the limitation size of less. And the algorithms that require IV or counter need to update them for each slice. How to update IV or counter refers to [1] and [2]. But Authenc cannot input data larger than the upper limitation of the scatter/gather list and the fragment size due to Linux kernel's specification. The upper limit of the data size that can be entered by one cryptographic process is depends on size restriction of scatter/gather list and fragment size.

How to input data in the user mode

Secure Key Package process in user space uses the function of splice and vmsplice to avoid a copy operation into kernel space. The limitation for the number of the memory page is 16 pages because the pipe function to send data using splice and vmsplice has this page limitation. If more data is to be sent to the driver, user space must slice the input into segments with a maximum size of 16 pages. But Authenc cannot input data larger than the upper limitation of the scatter/gather list and the fragment size due to Linux kernel's specification. The upper limit of the data size that can be entered by one cryptographic process is 16 pages (limitation of pipe function).

Hash algorithm in the user mode

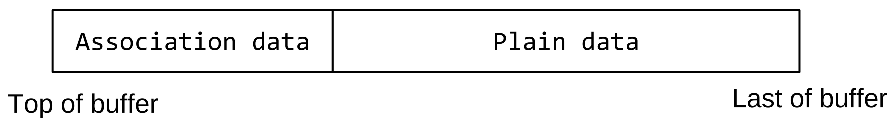
To perform cryptographic processing from the user space using the socket interface, user must use the setsockopt function to specify a key with an address of NULL and a size of 0 bytes. Since this process is specific to the driver, an error will occur if this process exists in the program when cryptographically processing with the Linux standard library.

Data buffer using Authenc algorithm in the kernel mode and user mode

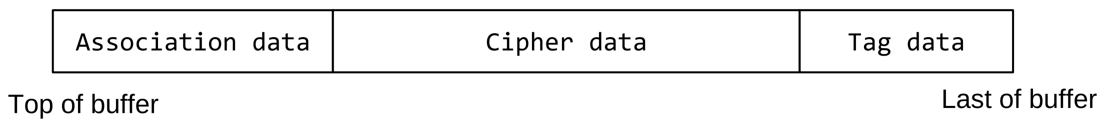
Authenc algorithm uses splice and vmsplice (sg_set_buf function in the case of kernel) to pass a buffer that connects some data to the driver. This buffer contains association data, Plain/Cipher data and tag data. Association data allows data of 8 entries of scatter/gather list and the zero byte of Plain/Cipher data is not allowed.

This buffer is connected data in the order shown in the figure below.

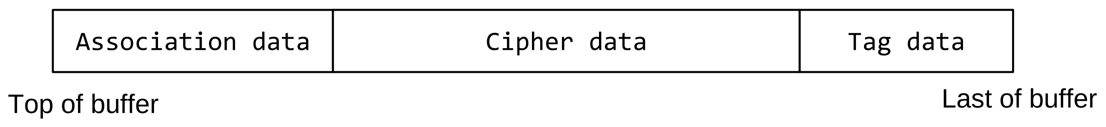
In case of the input buffer of Authenc encryption:



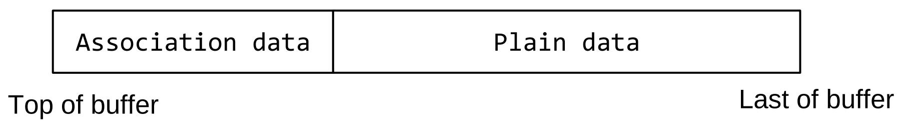
In case of the output buffer of Authenc encryption:



In case of the input buffer of Authenc decryption:



In case of the output buffer of Authenc decryption:



The output association data is output as the zero buffer that has the same size as the input association data. And when using the same buffer for input and output, the user allocates the same memory area as the larger buffer size. These specifications for the buffer are the same as using the Linux kernel software library.

Authenc algorithm in the user mode

Users input a size of data to be encrypted or decrypted as the fourth argument of aead_request_set_crypt (This function is defined in the kernel for cryptographic processing). This specification is the same as using the Linux kernel software library.

5. Integration Procedure

The following shows procedure for integration to build the CCREE Driver when build Linux kernel.

5.1 Directory Configuration

CCREE Driver is included in Linux kernel in the directory drivers/crypto/ccree. Details are as below:

```
drivers/
└── crypto/
    └── ccree/
        ├── cc_aead.c
        ├── cc_aead.h
        ├── cc_buffer_mgr.c
        ├── cc_buffer_mgr.h
        ├── cc_cipher.c
        ├── cc_cipher.h
        ├── cc_crypto_ctx.h
        ├── cc_debugfs.c
        ├── cc_debugfs.h
        ├── cc_driver.c
        ├── cc_driver.h
        ├── cc_fips.c
        ├── cc_fips.h
        ├── cc_hash.c
        ├── cc_hash.h
        ├── cc_host_regs.h
        ├── cc_hw_queue_defs.h
        ├── cc_kernel_regs.h
        ├── cc_lli_defs.h
        ├── cc_pm.c
        ├── cc_pm.h
        ├── cc_request_mgr.c
        ├── cc_request_mgr.h
        ├── cc_sram_mgr.c
        ├── cc_sram_mgr.h
        └── Makefile
```

5.2 Integration Procedure for Yocto environment

CCREE Driver can be integrated as built-in module in kernel image or can be built as separate kernel module with kernel image. When it is built as kernel module, it needs to be installed into root filesystem (rootfs) of target board, so that it is available for installing into running system.

Please follow below steps for configuration and build.

5.2.1 How to build CCREE Driver as built-in module in Linux BSP environment.

5.2.1.1 How to enable configure and build for CCREE Driver module.

```
$ cd rcar-linux-bsp
$ make defconfig
#Setting below in console of menuconfig
$ make menuconfig
```

```
  → Cryptographic API
     --- Hardware crypto devices
           <*>    Support for ARM TrustZone CryptoCell family of security processors
```

```
#Build kernel Image (already including CCREE Driver.)
$ make -j16 Image CONFIG_DEBUG_SECTION_MISMATCH=y
#Build device tree
$ make dtbs
```

5.2.1.2 How to confirm built-in CCREE Driver operation

When CCREE Driver is built-in in kernel Image, after booting the board, confirm on board. It will show the log that CCREE Driver is initialized as below:

```
$ dmesg | grep ”ARM ccree device initialized”

   ------------- confirm below log is shown out -------------
   ccree e6601000.crypto: ARM ccree device initialized
```

5.2.2 How to build CCREE Driver as loadable kernel module in Linux BSP environment.

5.2.2.1 How to enable configure and build for CCREE Driver module.

```
$ cd rcar-linux-bsp
$ make defconfig
# Setting below in console of menu config
$ make menuconfig
```

```
  → Cryptographic API
     --- Hardware crypto devices
           <*>    Support for ARM TrustZone CryptoCell family of security processors
```

```
# Build kernel Image (already including CCREE Driver.)
$ make -j16 Image CONFIG_DEBUG_SECTION_MISMATCH=y
# Build module (including CCREE Driver module.)
$ make -j16 modules
# Install all module into standard path of "kernel module" in rootfs
$ make -j16 modules_install INSTALL_MOD_PATH=$ROOTFS
$ make dtbs
```

5.2.2.2 How to confirm loadable CCREE Driver operation

When CCREE Driver is built as loadable module, after booting the board, confirm on board:

- In case the ccree.ko is installed in kernel module "standard path" in target board rootfs (usually, it is in ${rootfs}/lib/modules/x.y.z-yocto-standard/kernel/crypto/; with x.y.z is Linux kernel version e.g. 5.4.0)

```
# List out being installed module in system and confirm no CCREE Driver module
is installed
$ lsmod
# Install CCREE Driver kernel module into system
$ modprobe ccree
  ------------- confirm below log is shown out -------------
  ccree e6601000.crypto: ARM ccree device initialized
# List out being intalled module in system
$ lsmod
   ------- confirm CCREE Driver module is already installed similar as below -----
   Module              Size              Used by
   ccree              167936             0
```

<u>Note</u>: If ccree presents in result of lsmod step, it means CCREE Driver module is already installed so it does not need to do following steps of this.

- In case the ccree.ko is not installed in kernel module "standard path" in target board rootfs. After build module, please copy ccree.ko (from linux/drivers/crypto/ccree/ccree.ko) to home folder on target rootfs. And after booting up target board, please execute below:

```
# Copy ccree.ko to home folder in Linux Host PC
$ cp ccree.ko export/nfs/home/root/
# Change to home folder(e.g. /home/root if user "root") on System board
$ cd /home/root
# List out being installed module in system and confirm no CCREE Driver module
is installed
$ lsmod
# Install CCREE Driver kernel module into system
$ insmod ccree.ko
  ------------- confirm below log is shown out -------------
  ccree e6601000.crypto: ARM ccree device initialized
# List out being intalled module in system
$ lsmod
  ----------- confirm CCREE Driver module is already installed similar as below ---
--
  Module              Size                Used by
  ccree             167936                0
```

5.3 Integration Procedure for Android environment

5.3.1 How to build CCREE driver as a built-in module in Android BSP environment

Below procedure is applied for both user-debug mode and release mode.

- Follow Android Build procedure as chapter 2 in [10].
- At chapter “2.5. Building Android, IPL, U-Boot, and Kernel sources”, after step executing “./walkthrough.sh”, please do below:

```
$ cd ${workspace}/mydroid/device/renesas/kernel/arch/arm64/configs/
$ vi android_salvator_defconfig
------------------- Please append below to configured file -------------------
# Enable CCREE
CONFIG_CRYPTO_DEV_CCREE=y
CONFIG_CRYPTO_USER=y
CONFIG_CRYPTO_MANAGER_DISABLE_TESTS=y

CONFIG_CRYPTO_USER_API=y
CONFIG_CRYPTO_USER_API_HASH=y
CONFIG_CRYPTO_USER_API_SKCIPHER=y
CONFIG_CRYPTO_USER_API_AEAD=y
CONFIG_CRYPTO_USER_API_RNG=y

CONFIG_CRYPTO=y
CONFIG_CRYPTO_ALGAPI=y
CONFIG_CRYPTO_ALGAPI2=y
CONFIG_CRYPTO_AEAD=y
CONFIG_CRYPTO_AEAD2=y
CONFIG_CRYPTO_BLKCIPHER=y
CONFIG_CRYPTO_BLKCIPHER2=y
CONFIG_CRYPTO_HASH=y
CONFIG_CRYPTO_HASH2=y
CONFIG_CRYPTO_RNG=y
CONFIG_CRYPTO_RNG2=y
CONFIG_CRYPTO_PCOMP2=y
CONFIG_CRYPTO_MANAGER=y
CONFIG_CRYPTO_MANAGER2=y
CONFIG_CRYPTO_GF128MUL=y
CONFIG_CRYPTO_NULL=y
CONFIG_CRYPTO_WORKQUEUE=y
CONFIG_CRYPTO_CRYPTD=y
CONFIG_CRYPTO_AUTHENC=y
CONFIG_CRYPTO_TEST=m
CONFIG_CRYPTO_CCM=y
CONFIG_CRYPTO_GCM=y
CONFIG_CRYPTO_SEQIV=y
CONFIG_CRYPTO_CBC=y
CONFIG_CRYPTO_CTR=y
CONFIG_CRYPTO_CTS=y
CONFIG_CRYPTO_ECB=y
CONFIG_CRYPTO_XTS=y
CONFIG_CRYPTO_HMAC=y
CONFIG_CRYPTO_XCBC=y
CONFIG_CRYPTO_CRC32C=y
CONFIG_CRYPTO_GHASH=y
CONFIG_CRYPTO_MD5=y
CONFIG_CRYPTO_SHA1=y
CONFIG_CRYPTO_SHA1_ARM=y
CONFIG_CRYPTO_SHA256=y
CONFIG_CRYPTO_SHA512=y

CONFIG_CRYPTO_AES=y
CONFIG_CRYPTO_AES_ARM=y
CONFIG_CRYPTO_DES=y
---------------------------------------
```

- Continue remained steps in chapter 2.5 of [10] to build Android.
- Confirm CCREE operation: please do similar as 5.2.1.2 How to confirm built-in CCREE Driver operation.

5.3.2 How to build CCREE driver as loadable kernel module in Android BSP environment

5.3.2.1 How to build CCREE driver kernel module in user-debug mode

For build CCREE driver, please do similar as chapter "5.3.1 How to build CCREE driver as a built-in module in Android BSP environment", except below configure please change "y" to "m" before build process:

```
CONFIG_CRYPTO_DEV_CCREE=m    # change to 'm' instead of 'y'
```

5.3.2.2 How to confirm loadable CCREE driver kernel module in user-debug mode

After system boot up successfully, please do below steps in Android shell:
(To access Android shell, please access target board console or if from host PC by "adb shell").

```
$ su
$ cd ${workspace}/mydroid/vendor/lib/module
$ ls . | grep ccree          # confirm that ccree.ko exist
$ insmod ccree.ko
```

Confirm CCREE driver initialized as at chapter 5.2.2.2 How to confirm loadable CCREE Driver operation.

5.3.2.3 How to build loadable CCREE driver kernel module in release mode

For build CCREE driver, please do similar as chapter "5.3.1 How to build CCREE driver as a built-in module in Android BSP environment", except:

- Below configure please change "y" to "m" before build process:

```
CONFIG_CRYPTO_DEV_CCREE=m    # change to 'm' instead of 'y'
```

- And change init file (for system load kernel module automatically after booting up) as below. Please add gray line into file and continue build process.

```
$ cd ${workspace}/mydroid
$ vi device/renesas/common/init/init.common.rc
-------------------------------------
on boot
    write /sys/power/wake_lock ws10_suspend_wa
    chmod 0666 /sys/class/rfkill/rfkill0/state
    insmod /vendor/lib/modules/ccree.ko
    insmod /vendor/lib/modules/qos.ko
    insmod /vendor/lib/modules/vspm.ko
    insmod /vendor/lib/modules/vspm_if.ko
    insmod /vendor/lib/modules/mmngr.ko
    insmod /vendor/lib/modules/mmngrbuf.ko
    insmod /vendor/lib/modules/uvcs_drv.ko
    # default country code
    setprop ro.boot.wificountrycode 00
    restorecon /dev/stune/foreground/tasks
-------------------------------------
```

5.3.2.4 How to confirm loadable CCREE driver kernel module in release mode

In system boot log, please confirm below line exist (please find and check by eye.)

```
ccree e6601000.crypto: ARM ccree device initialized
```

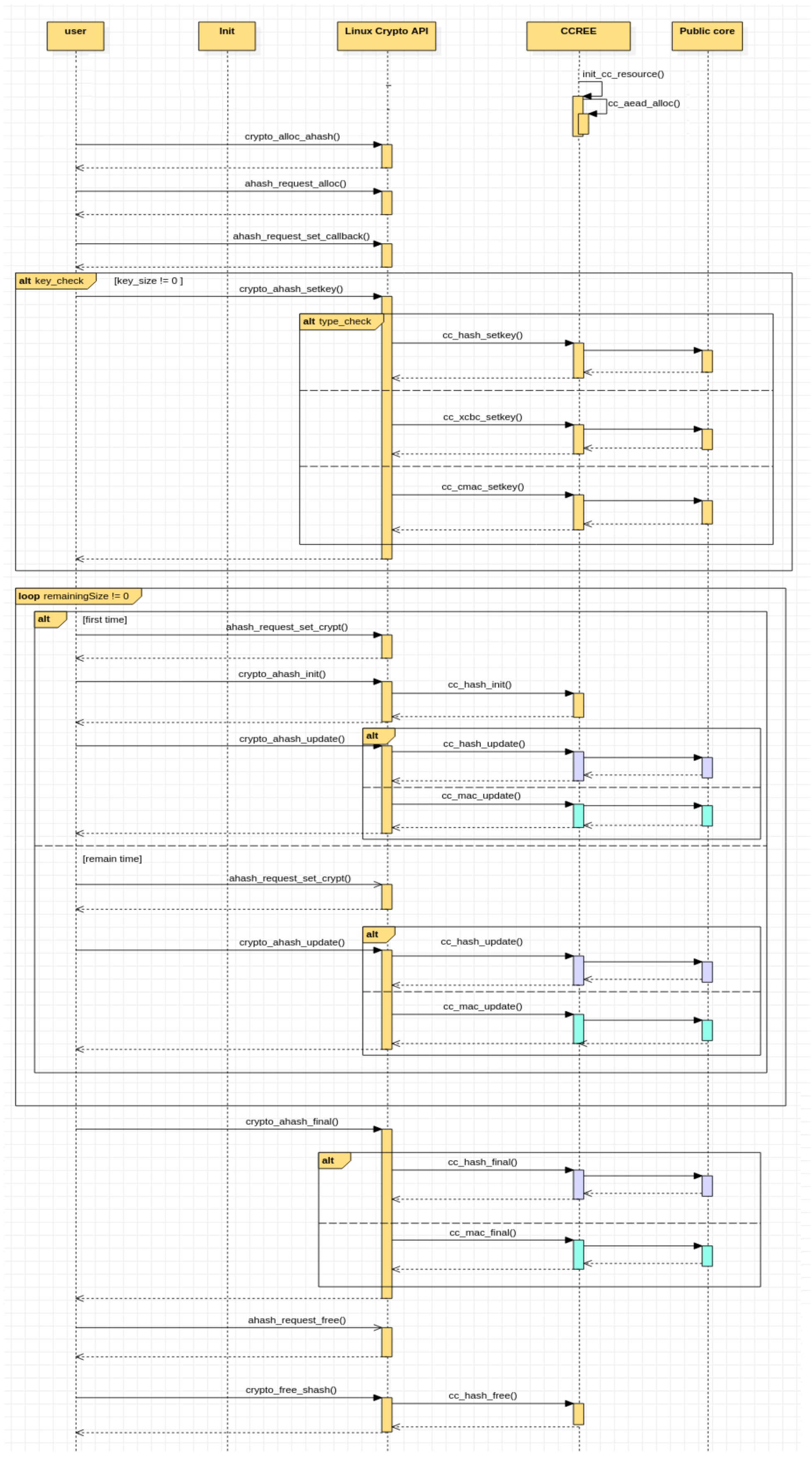
6. How to use CCREE Driver

This chapter describes how CCREE Driver is used. It will show simple processing flow from Linux crypto APIs call to CCREE Driver source code and give a sample example for using Linux crypto APIs at 4.2 External Interface.

6.1 Message Digest Usage

6.1.1 CCREE Driver - Message Digest simple processing flow

Figure 6-1: CCREE Driver - Message Digest simple processing flow



6.1.2 Message Digest usage example

Below figure shows example for calling Linux crypto APIs to perform Message Digest function provided by CCREE Driver. It gives short comment to describe purpose of each function and the calling order of function in calling flow.

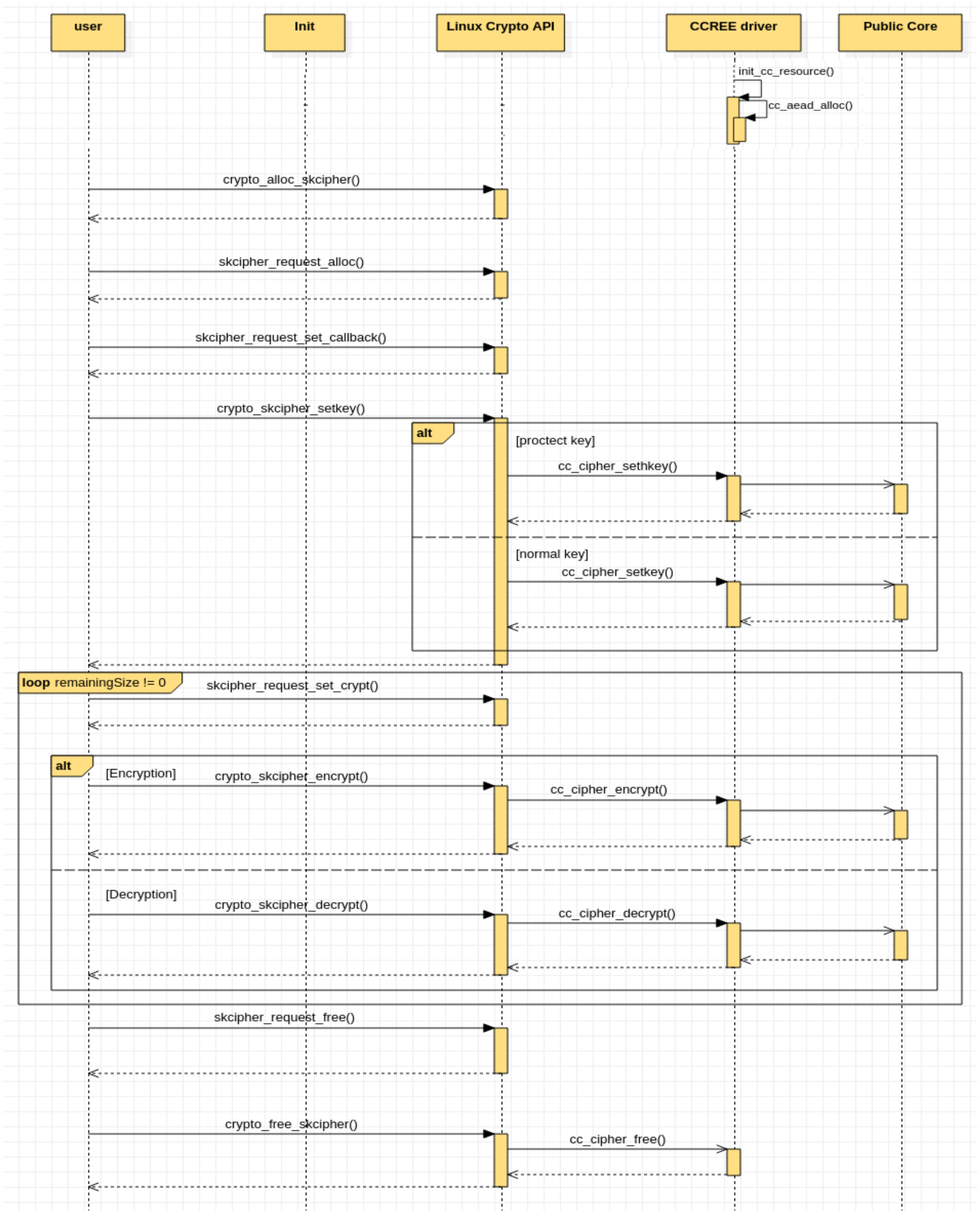
Figure 6-2: Message Digest example

```
struct tcrypt_result {
        struct completion completion;
        int err;
};

static void tcrypt_complete(struct crypto_async_request *req, int err)
{
        struct tcrypt_result *res = req->data;
        if (err == -EINPROGRESS)
                return;

        res->err = err;
        complete(&res->completion);
}

static int test_hash(void)
{
        int err;
        struct crypto_ahash *tfm;
        struct tcrypt_result ahresult;
        struct ahash_request *req;
        const char *alg_name = "hmac(sha256)"; // Algorithm name that is provided by CCREE (*)
        u32 type, u32 mask

        /* Test input pattern */
        char *key = "\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b";
        unsigned char ksize = 16;
        const char *plaintext = "hmac(sha256)test";
        unsigned char psize = 16;
        char *result[16];
        char *xbuf[16];
        struct scatterlist sg[8];

        /* Transform creation */
        tfm = crypto_alloc_ahash(alg_name, 0, 0); /* allocate ahash transform */
        if (IS_ERR(tfm))
                err = ...;  // Set err return and take action if any ...

        init_completion(&ahresult.completion);

        memcpy(xbuf, plaintext, psize);
        sg_init_one(&sg[0], xbuf, psize);

        /* Allocate request & set-up request input */
        req = ahash_request_alloc(tfm, GFP_KERNEL);

        ahash_request_set_callback(req, CRYPTO_TFM_REQ_MAY_BACKLOG,
                tcrypt_complete, &ahresult); //Set callback function
        ret = crypto_ahash_setkey(tfm, key, ksize);       //Set key
        ahash_request_set_crypt(req, sg, result, psize);

        /* Perform hash request */
        crypto_ahash_init(req);
        crypto_ahash_update(req);
        crypto_ahash_final(req);

        printk("hmac(sha256) test result: %s\n", result); /* Show HMAC result */

        /* Free source before close test */
        ahash_request_free(req);
        crypto_free_ahash(tfm);

out_err:
        return err;
}
```

6.2 Symmetric cipher Usage

6.2.1 CCREE Driver – Symmetric cipher simple processing flow

Figure 6-3: CCREE Driver – Symmetric cipher simple processing flow



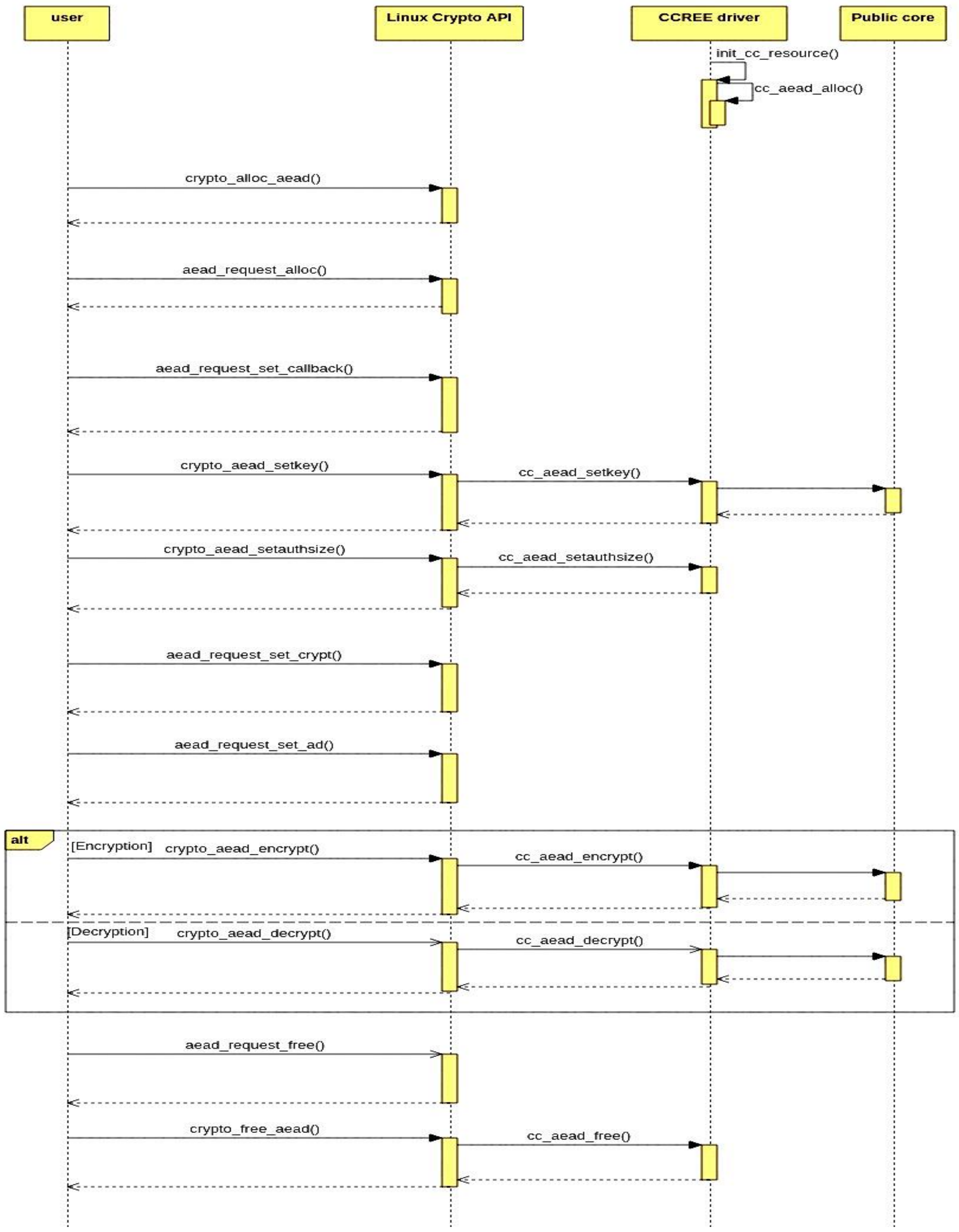
6.2.2 Symmetric cipher usage example

Please refer to Code Example For Symmetric Key Cipher Operation

6.3 AEAD Usage

6.3.1 CCREE Driver – AEAD simple processing flow

Figure 6-4: CCREE Driver – AEAD simple processing flow



6.3.2 AEAD usage example

Below figure shows example for calling Linux crypto APIs to perform AEAD function provided by CCREE Driver. It gives short comment to describe purpose of each function and the calling order of functions.

Figure 6-5: AEAD example

```
struct tcrypt_result {
        struct completion completion;
        int err;
};
static void tcrypt_complete(struct crypto_async_request *req, int err)
{
        struct tcrypt_result *res = req->data;

        if (err == -EINPROGRESS)
                return;
        res->err = err;
        complete(&res->completion);
}
static int test_aead(void)
{
        struct crypto_aead *tfm;
        int err = 0;
        struct tcrypt_result ahresult;
        struct aead_request *req;
        const char *alg_name = "authenc(hmac(sha1),cbc(aes))";//Algorithm is provided by CCREE (*)
        u32 type, u32 mask

        /* Test input pattern */
        char *key = "\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b";
        unsigned char ksize = 16;
        const char *plaintext = "authenc(hmac(sha1),cbc(aes))";
        unsigned char psize = ;
        char *result[16];
        char *xbuf[16];
        struct scatterlist sg[8];
        unsigned int authsize = 16, iv_len = 16;
        char *iv = "0123456789abcdef";

        /* Transform creation */
        tfm = crypto_alloc_aead(alg_name, 0, 0);  /* allocate ahash transform */
        if (!req)
                err = ...; // Set err return and take action if any ...

        init_completion(&ahresult.completion);

        memcpy(xbuf, plaintext, psize);
        sg_init_one(&sg[0], xbuf, psize);

        /* Allocate request & set-up request input */
        req = aead_request_alloc(tfm, GFP_KERNEL);
        aead_request_set_callback(req, CRYPTO_TFM_REQ_MAY_BACKLOG,
                                  tcrypt_complete, &result); //Set callback function
        ret = crypto_aead_setkey(tfm, key, ksize); //Set key
        ret = crypto_aead_setauthsize(tfm, authsize);
        aead_request_set_crypt(req, sg, sg, iv_ilen, iv);
        aead_request_set_ad(req, alen);
        ahash_request_set_crypt(req, sg, result, psize);

        /* Perform aead encrypt request */
        crypto_aead_encrypt(req)

        /* Show AEAD result */
        printk("authenc(hmac(sha1),cbc(aes)) test result: %s\n", result);

        /* Free source before close test */
        aead_request_free(req);
        crypto_free_aead(tfm);

out_err:
        return err;
}
```

7. Secure Key Package Algorithm

- This chapter will explain the procedure that user needs to do for using SKP feature. Summarily, the user needs to do two main tasks:
  - Implement a CA-TA pair so that the CA can call to TA to get SKP data.
    - The TA can generate SKP by implementing TA_InvokeCommandEntryPoint() that calls to RCAR_GenSkeyPackage(). The RCAR_GenSkeyPackage() is function in OPTEE-OS that generate SKP data.
  - Perform cipher (encryption/decryption) as normal procedure but using the SKP data as a key.
    - This step is not in scope of CA. But for convenience, it is described in same source file of CA.
    - CA can be a user-space and kernel-space CA. User-space CA will use socket interface to perform cipher, while kernel space CA can call directly to Linux crypto framework inteface in Linux kernel to use CCREE driver for cipher. But both of them have same purpose.
- In description below, it does not show full/complete source code of CA. It just picks main functions for explanation and illustration, so user needs to understand how to use these functions and implement full source code by self (description of APIs are used in CA can be referred in Linux kernel documentation and Linux source.)
- For using SKP feature, it needs SS6.3 Secure Driver is installed in OPTEE-OS.

7.1 Secure Key Package Algorithm introduction

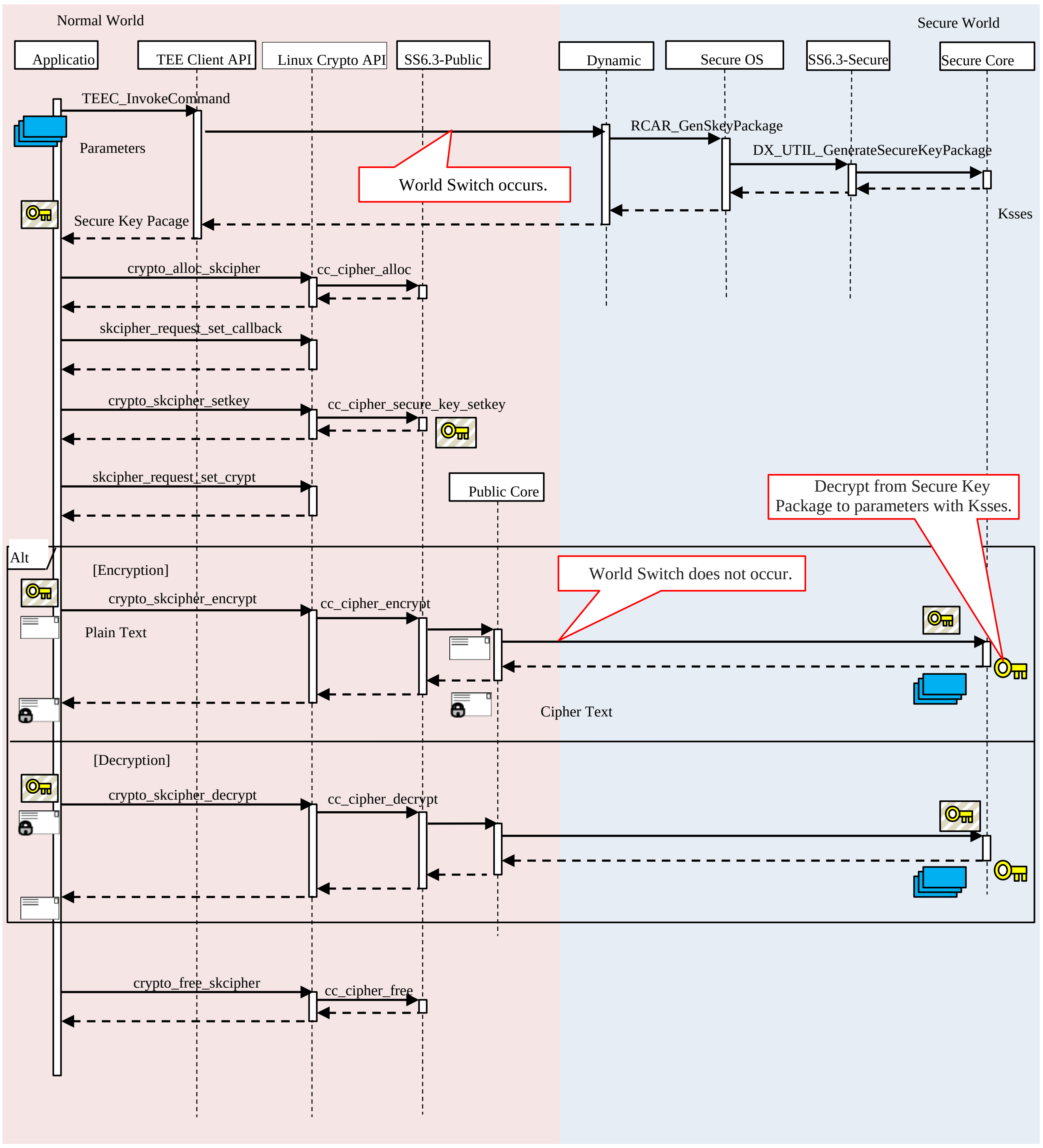
CCREE Driver has a function to securely process a package containing information used for encryption or decryption. This package is called Secure Key Package and is generated by encrypting parameters entered by the user with Ksses. Generating Secure Key Package uses RCAR_GenSkeyPackage of SS6.3 Secure Driver function and refers to Chapter 4 of [5]. Encryption or decryption using packages is performed by Public Core, but decrypt Secure Key Package using Ksses for getting parameters is performed by Secure Core via Public Core when cryptographic processing is executed. Since the contents of the package cannot be read from the Normal World application, it can perform secure cryptographic processing.

For example, this function is used to enable media playback code to decrypt DRM-protected content into protected memory shared with a hardware codec.

Figure 7-1 shows the sequence to realize the Secure Key Package function.

Secure Key Package can also use the user space. In this case, the cryptographic processing interface of Linux replaces the function for user space, but the sequence of encryption processing is almost the same. If users want to know implementation instruction, refer to chapter 7.3 User space Secure Key Package and 7.4 Kernel space secure key package. These chapters explain the encryption and decryption processing using Secure Key Package to be executed in the user space and the kernel space.

Figure 7-1 Sequence to realize the Secure Key Package function in the kernel mode



Next parts will figure main components and make sample for how to implement a Secure Key Package.

7.2 General about Secure Key Package

Software composition

Figure below shows the software relationship, and software belongs SKP scope is described in this document is painted gray. User-space/kernel-space CA requests SKP to SS 6.3 Secure driver. After that, it uses SKP as key to call to CCREE encrypt/decrypt by calling Linux crypto framework.

Table 7-1: SKP software composition

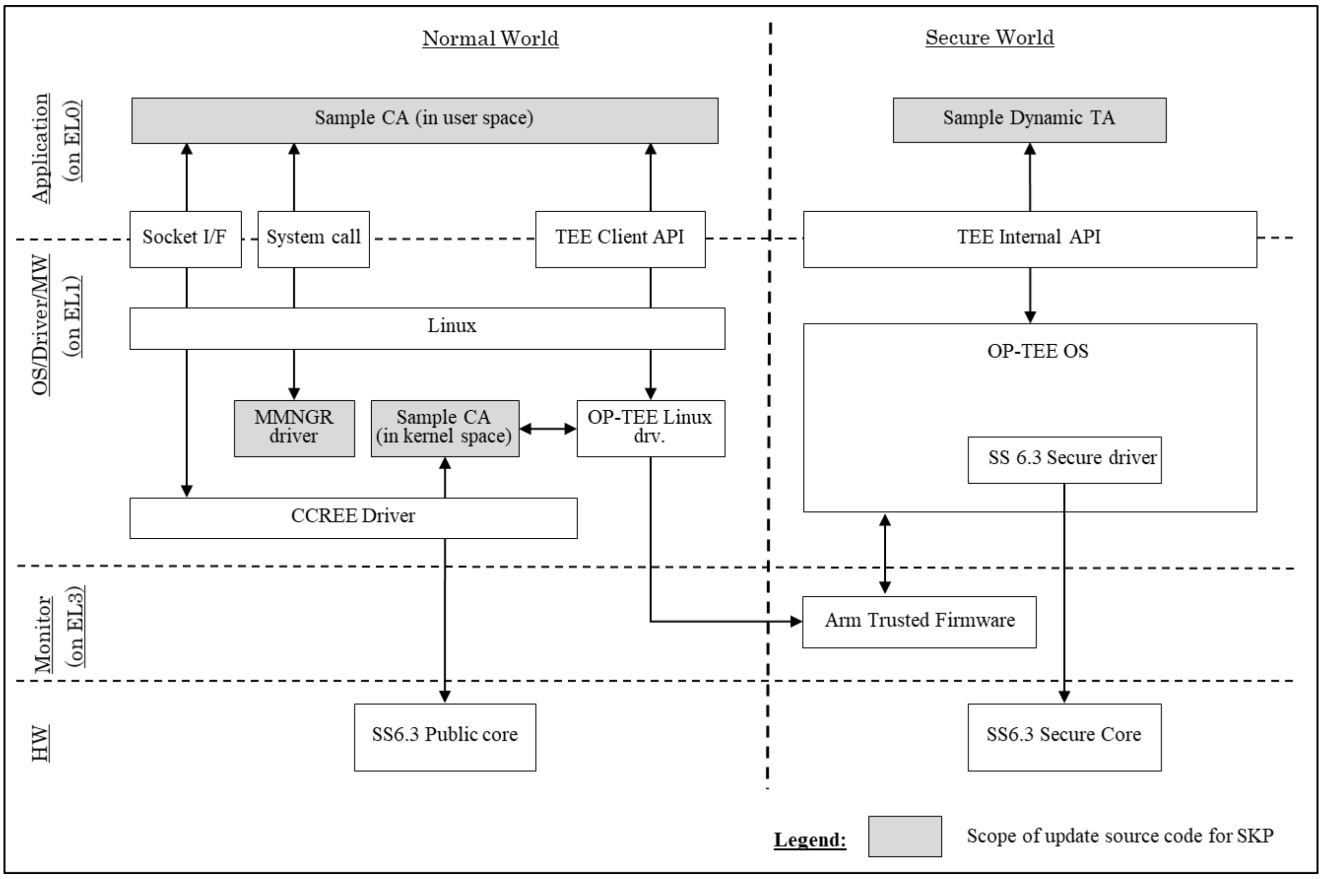


Table 7-2: SKP related software modules

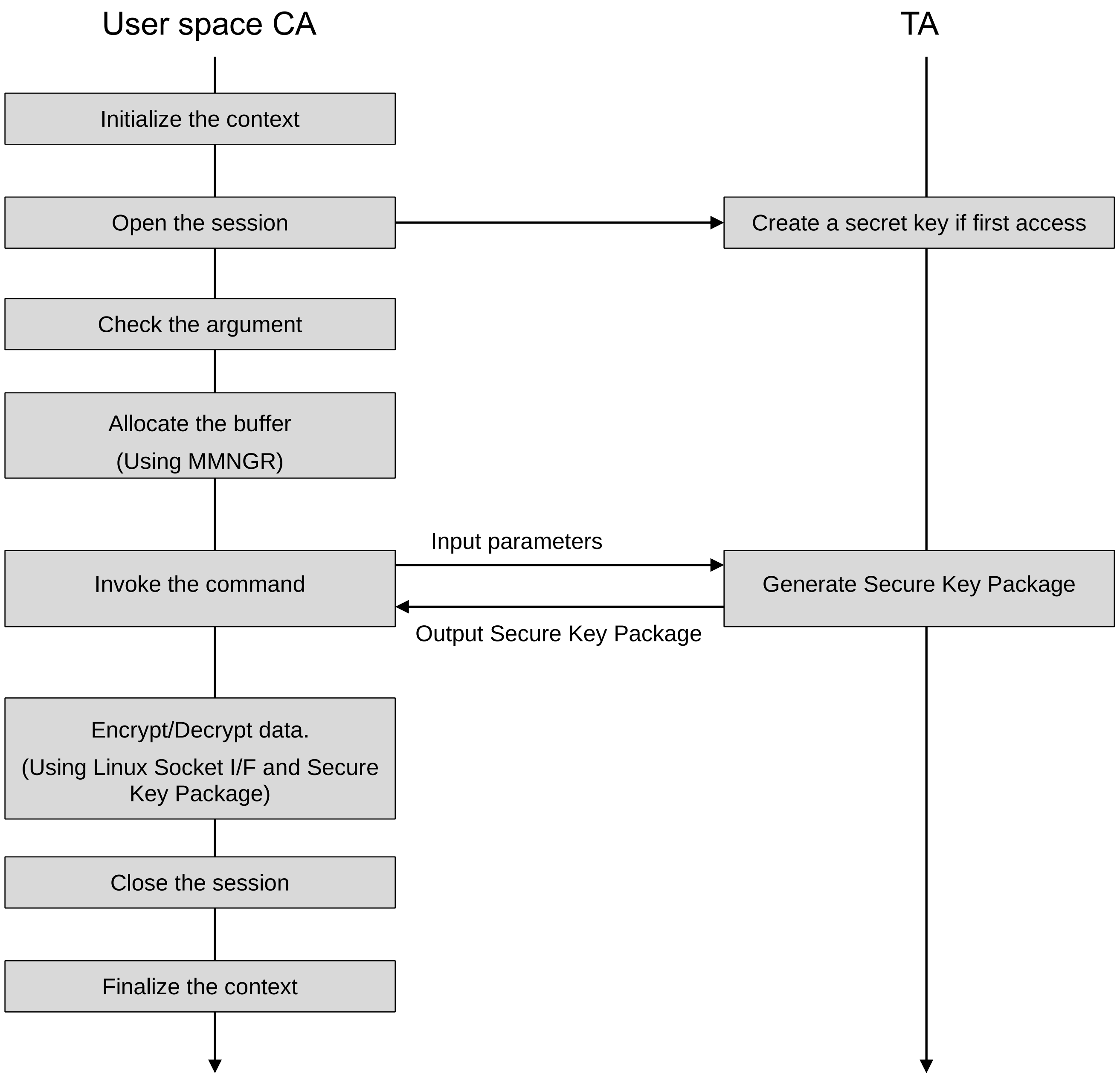
Name	Note
Sample CA in the user space	This CA is executed in the user space of Normal World. It encrypts or decrypts data using Secure Key Package.
Sample CA in the kernel space	This CA is executed in the kernel space of Normal World. It encrypts or decrypts data using Secure Key Package.
Sample Dynamic TA	This TA has parameters to generate Secure Key Package. This package is used as key for encryption/decryption in CA. Sample CA in the user space and the kernel space use the same Sample Dynamic TA.
MMNGR Driver	This driver is used to allocate a contiguous memory area. This area is used as a buffer for transferring input/output data between CA in the user space and Dynamic TA.

7.3 User space Secure Key Package

7.3.1 Software processing flow

The following shows the outline of process of CA in the user space and Dynamic TA for Sample Secure Key Package.

Figure 7-2: SKP processing flow with user-space CA



7.3.2 How to modify and build MMNGR driver

In user space SKP, MMNGR driver is used to allocate contiguous physical memory area (when size of input file is greater than 4KBs) for transferring the data between user space CA and Dynamic TA. For using MMNGR driver for SKP, it needs to modify the driver source code, build and install as description in this chapter.

For more other change or configure of MMNGR, please refer to Reference document [10].

Step 1: Prepare MMNGR driver source code

Please checkout the source code as below:

```
$ cd $WORK
$ git clone https://github.com/renesas-rcar/mmngr_drv.git
$ cd mmngr_drv
$ git checkout 24398024264741363 12bd10bc4c143fbf1c84850
```

Note: This source version is for build with Yocto version 5.5.0. If newer version of Yocto is used, please checkout appropriate MMNGR driver source with it.

Step 2: Modify MMNGR driver

Please modify file mmngr_drv/mmngr/mmngr-module/files/mmngr/drv/mmngr_drv.c as below. The light blue highlight text is added source code.

```
static int mmap(struct file *filp, struct vm_area_struct *vma) {

 phys_addr_t  off;
  unsigned long  len;
  struct MM_PARAM  *p = filp->private_data;
  unsigned long  loop;
  unsigned long  virt_addr;
  unsigned long  phy_addr;
 … … …
#if 0
  if (p->flag != MM_KERNELHEAP_CACHED) {
    vma->vm_page_prot = pgprot_writecombine(vma->vm_page_prot);
    vma->vm_flags |= (VM_IO | VM_DONTEXPAND | VM_DONTDUMP);
  … … …
  if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff,
          vma->vm_end - vma->vm_start, vma->vm_page_prot))
    return -EAGAIN;
#else

  vma->vm_page_prot = pgprot_writecombine(vma->vm_page_prot);

  virt_addr = vma->vm_start;
  phy_addr = off;

  for (loop = 0; loop < ((((vma->vm_end - vma->vm_start) + PAGE_SIZE) - 1) / PAGE_SIZE);
       loop++ ) {
    if (vm_insert_page(vma, virt_addr, pfn_to_page(phy_addr >> PAGE_SHIFT)) != 0) {
      pr_debug("%s: vm_insert_page failed\n", __func__);
      return -EAGAIN;
    }
    virt_addr += PAGE_SIZE;
    phy_addr += PAGE_SIZE;
  }
#endif
```

Step 3: Extract the Yocto toolchain (Software Development Kit (SDK))

Extract the SDK to “sdk” folder in work directory as below:

```
$ cd ${WORK}
$ bash poky-glibc-x86_64-core-image-weston-sdk-aarch64-salvator-x-toolchain-3.1.8.sh

Poky (Yocto Project Reference Distro) SDK installer version 3.1.8
=================================================================
Enter target directory for SDK (default: /opt/poky/3.1.8): sdk
$ SDK=${WORK}/sdk
```

Step 4: Configure, build and deploy MMNGR driver

```
$ unset LD_LIBRARY_PATH
$ source ${SDK}/environment-setup-aarch64-poky-linux
$ export INCSHARED=$SDKTARGETSYSROOT/usr/local/include

$ export KERNELSRC=${WORKDIR}/rcar-bsp-linux
$ export CP=cp
$ export MMNGR_CONFIG=MMNGR_SALVATORX
$ export MMNGR_SSP_CONFIG=MMNGR_SSP_DISABLE
$ export MMNGR_IPMMU_MMU_CONFIG=IPMMU_MMU_DISABLE
$ export INCSHARED=$SDKTARGETSYSROOT/usr/local/include

$ cd ${WORK}/mmngr_drv/mmngr/mmngr-module/files/mmngr/drv
$ ls mmngr.ko   # check .ko file exist
$ make clean
$ make

$ cp mmngr.ko   <path to rootfs>/lib/modules/5.10.41-yocto-standard/extra/
```

Note:

- The version of SDK "3.1.8” is generated by Yocto version 5.5.0. It can be changed with newer Yocto version.
- “5.10.41-yocto-standard” will depend on kernel version, please change it for appropriate with rootfs.
- After copying to appropriate location, mmngr.ko will be inserted automatically when target system boot. In case it is not inserted automatically, please perform “insmod mmngr.ko” manually.

7.3.3 How to implement user space CA and Dynamic TA

Step 1: git clone source code

```
$ cd $WORK
$ git clone https://github.com/linaro-swg/optee_examples.git
$ cd optee_examples
$ git checkout 3.15.0
```

Step 2: Modify CA source code for SKP

Below modifition is applied in: optee_examples\hello_world\host\main.c. After modiying, this file is not just a simple CA as original, it becomes a sample for using secure key package.

- Adding new function to allocate memory buffer (by using MMNGR driver) for containing input and output files. After allocating memory, read the input file into first half of buffer; the second half is for output.

```
int32_t alloc_memory(char *inFileName, struct MM_PARAM *k_mem, void **mp, size_t *readSize)
{
        /* Open MMNGR device driver */
        char *file_name = "/dev/rgnmm";   //MMNGR device file
        Driver_File = open(file_name, O_RDWR);

        /* Allocate a memory area with size is double of input file size (round to page size) */
        k_mem->size = ((st.st_size + PAGE_SIZE - 1U) & ~(PAGE_SIZE - 1U)) * 2U;
        iores = ioctl(Driver_File, MM_IOC_ALLOC, k_mem);
        /* Get allocated memory for using */
        iores = ioctl(Driver_File, MM_IOC_GET, k_mem);

        /* Map memory area into process address space and copy input content into it */
        *mp = mmap(NULL, k_mem->size, PROT_READ | PROT_WRITE, MAP_SHARED, Driver_File, 0);
        f_in = fopen(inFileName, "rb");
        *readSize = fread(*mp, 1U, st.st_size, f_in);
        fclose(f_in);
}
```

- Adding new function for getting secure key package, main content is as below:

```
TEEC_Result get_secure_key_package(TEEC_Session *sess, uint8_t *atp_data,
                struct ta_cmd_parm *cmd, struct MM_PARAM *k_mem)
{
        op.paramTypes = TEEC_PARAM_TYPES(TEEC_MEMREF_TEMP_INOUT, TEEC_MEMREF_TEMP_INOUT,
                                         TEEC_MEMREF_TEMP_INOUT, TEEC_MEMREF_TEMP_INOUT);

        op.params[0].tmpref.buffer = (void*)atp_data; //for containing SKP data
        op.params[0].tmpref.size = MAX_KPKG_SIZE;
        op.params[1].tmpref.buffer = (void*)cmd;   //Algorithm name, direction (encrypt/decrypt)
        op.params[1].tmpref.size = sizeof(struct ta_cmd_parm);
        op.params[3].tmpref.buffer = (void *)k_mem;
        op.params[3].tmpref.size = sizeof(struct MM_PARAM);

        res = TEEC_InvokeCommand(sess, 0, &op, &err_origin);
}
```

- Adding new function to encrypt/decrypt input file with using SKP. This is performed by using socket interface as in this reference link in Linux kernel documentation. Please refer link for more detail.

```
TEEC_Result crypto_user_data(struct crypto_data *in, uint8_t *outBuf,
                uint32_t *outBufSize, TEEC_Session *sess,
                struct ta_cmd_parm *cmd, struct MM_PARAM *k_mem)
{
        struct sockaddr_alg sa = {
                .salg_family = AF_ALG,
                .salg_type = "skcipher",
        };
        ret = get_secure_key_package(sess, atp_data, cmd, k_mem);

        (void)set_socket_alg(in->alg_tag, (char *)sa.salg_name);
        tfmfd = socket(AF_ALG, SOCK_SEQPACKET, 0);
        bind(tfmfd, (struct sockaddr *)&sa, sizeof(sa));

        key = atp_data; //using secure key package
        keySize = MAX_KPKG_SIZE;
        setsockopt(tfmfd, SOL_ALG, ALG_SET_KEY, key, keySize);

        opfd = accept(tfmfd, NULL, 0);

        iv->ivlen = in->iv_size;
        memcpy(iv->iv, in->iv, in->iv_size);

        hex_dmp("User Crypto Key", key, keySize);
        hex_dmp("User Crypto IV", iv->iv, in->iv_size);

        socket_ret = sendmsg(opfd, &msg, 0);

        splice_ret = vmsplice(pipes[1], &iov, 1, SPLICE_F_GIFT);
        splice_ret = splice(pipes[0], NULL, opfd, NULL, splice_ret, SPLICE_F_MORE);
        socket_ret = read(opfd, outBuf, iov.iov_len);
        close(opfd);

        close(tfmfd);

}
```

- Modify the main() function in optee_examples\hello_world\host\main.c as below.

```
int main(void)
{
        …    …    …
#if 0   //comment the TEEC_InvokeCommand() part.
        op.paramTypes = TEEC_PARAM_TYPES(TEEC_VALUE_INOUT, TEEC_NONE,
        …    …    …
        res = TEEC_InvokeCommand(&sess, TA_HELLO_WORLD_CMD_INC_VALUE, &op,
                                 &err_origin);
        …    …    …
        printf("TA incremented value to %d\n", op.params[0].value.a);
#endif
        /* Allocate memory buffer */
        alloc_memory(inFileName, k_mem, &mp, readSize)

        /* Perform encrypt/decrypt using SKP */
        crypto_user_data(in, outBuf, outBufSize, sess, cmd, k_mem) ;

        /* Free memory after using */
        iores = ioctl(Driver_File, MM_IOC_FREE, &k_mem);

}
```

Step 3: Modify Dynamic TA source code for SKP

Below modification is applied in: optee_examples\hello_world\ta\hello_world_ta.c. After modifying, this file will generate SKP for using by CA. Below illustration is for "cbc(saes)" algorithm.

- Adding new below secure key parameter arrays:

```
#include <tee_internal_api.h>
#include <tee_internal_api_extensions.h>
#include <utee_defines.h>
#include <string.h>
#include "user_ta_header_defines.h"

#define MAX_KPKG_SIZE 112U

/* Memory area that is allocated to contain input & output data. This struct is defined similar as
in mmngr_drv\mmngr\mmngr-module\files\mmngr\include\mmngr_private_cmn.h */
struct MM_PARAM {
        size_t size;            //size of memory area
        unsigned long long phy_addr;
        unsigned int    hard_addr;      //hardware address
        unsigned long   user_virt_addr;
        unsigned long   kernel_virt_addr;
        unsigned int flag;
 };

RCAR_SkeyParams_t skeyParamsArray = {
        0,                      //direction
        TEE_SKEY_CIPHER_CBC,    //mode
        0,                      //lowerBound
        0,                      //upperBound
        {0},                    //nonceBuf
        {0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,
         0x39,0x30,0x31,0x32,0x33,0x34,0x35,0x36},//keyBuf
        TEE_SKEY_AES_KEY128,    //keyType
        0,                      //keyNumRounds
        {0},                    //nonceCtrBuff
        0,                      //nonceLen
        0,                      //ctrLen
        0,                      //dataRange
        0,                      //isNonSecPathOp
}
```

- Modify TA_InvokeCommandEntryPoint() function as below. Other CA client functions are empty or just return successfully.

```
TEE_Result TA_InvokeCommandEntryPoint(void *sess_ctx __unused, uint32_t cmd_id __unused,
                uint32_t param_types __unused, TEE_Param params[4])
{
        TEE_Result ret = TEE_SUCCESS;
        RCAR_SkeyParams_t *skeyParams = NULL;
        uint8_t skeyPackageBuf[MAX_KPKG_SIZE] = {0};
        uint32_t skeySize = 0U;
        struct MM_PARAM *k_mem = NULL;
        struct ta_cmd_parm *ta_cmd;

        DMSG("command entry point for Dynamic ta \"%s\"", TA_NAME);
        ta_cmd = (struct ta_cmd_parm*)params[1].memref.buffer;

        //get template
        skeyParams = &skeyParamsArray;

        //set data
        skeyParams->direction = ta_cmd->Enc_flag; //encrypt or decrypt

        skeySize = sizeof(skeyPackageBuf);
        k_mem = (struct MM_PARAM *)params[3].memref.buffer;
        skeyParams->lowerBound = k_mem->hard_addr;
        skeyParams->upperBound = k_mem->hard_addr + k_mem->size;
        /* Generate Secure Key Package & copy it back to share memory buffer */
        ret = RCAR_GenSkeyPackage(skeyParams, skeyPackageBuf, skeySize); //generation SKP
        if (TEE_SUCCESS == ret) {
                DMSG("Generate Secure Key Package for Encrypt");
                params[0].memref.size = skeySize;
                params[2].value.a = skeyParams->isNonSecPathOp;
                /* copy secure key package buffer to parameter 0 */
                memcpy(params[0].memref.buffer, skeyPackageBuf, skeySize);
        }
        memset(skeyPackageBuf, 0, skeySize);
        return ret;
}
```

Step 4: Build source code

- Before build the source code, please export SDK as at step 3 of chapter 7.3.2. For build optee_example/hello_world, please follow below steps, beside that it is recommended to refer to this link for more information.

```
$ TC_PATH=${SDK}/sysroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux
$ export PATH=${TC_PATH}:$PATH
$ export CROSS_COMPILE=${TC_PATH}/aarch64-poky-linux-
$ export SDKTARGETSYSROOT="${SDK}/sysroots/aarch64-poky-linux"
$ export ARCH=arm64
$ export CC="aarch64-poky-linux-gcc  -march=armv8-a -mtune=cortex-a57.cortex-a53 -Wformat -
Wformat-security -Werror=format-security --sysroot=$SDKTARGETSYSROOT"
$ export LIBGCC_LOCATE_CFLAGS="--sysroot=${SDK}/sysroots/aarch64-poky-linux"

$ export TEEC_EXPORT=${SDK}/sysroots/aarch64-poky-linux/usr/
$ export TA_DEV_KIT_DIR=<path to optee_os>/out/arm-plat-rcar/export-ta_arm64
$ unset LDFLAGS
$ unset CFLAGS
$ cd ${WORK}/optee_examples/hello_world
$ make clean
$ make
```

Step 5: Deploy the CA and TA binary image

```
$ cd ${WORK}/optee_examples/hello_world
$ cp host/optee_example_hello_world      <target rootfs path>/home/root
$ mkdir -p <target rootfs path>/lib/optee_armtz     # create folder if it has not existed.
$ cp ta/out/8aaaf200-2450-11e4-abe2-0002a5d5c51b.ta      <target rootfs path>/lib/optee_armtz
```

Step 6: Confirm the result of CA execution

After boot target board, please execute the user space CA as below.

```
$ cd  ~
$ optee_example_hello_world
```

For confirming crypto result, please do 2 steps:

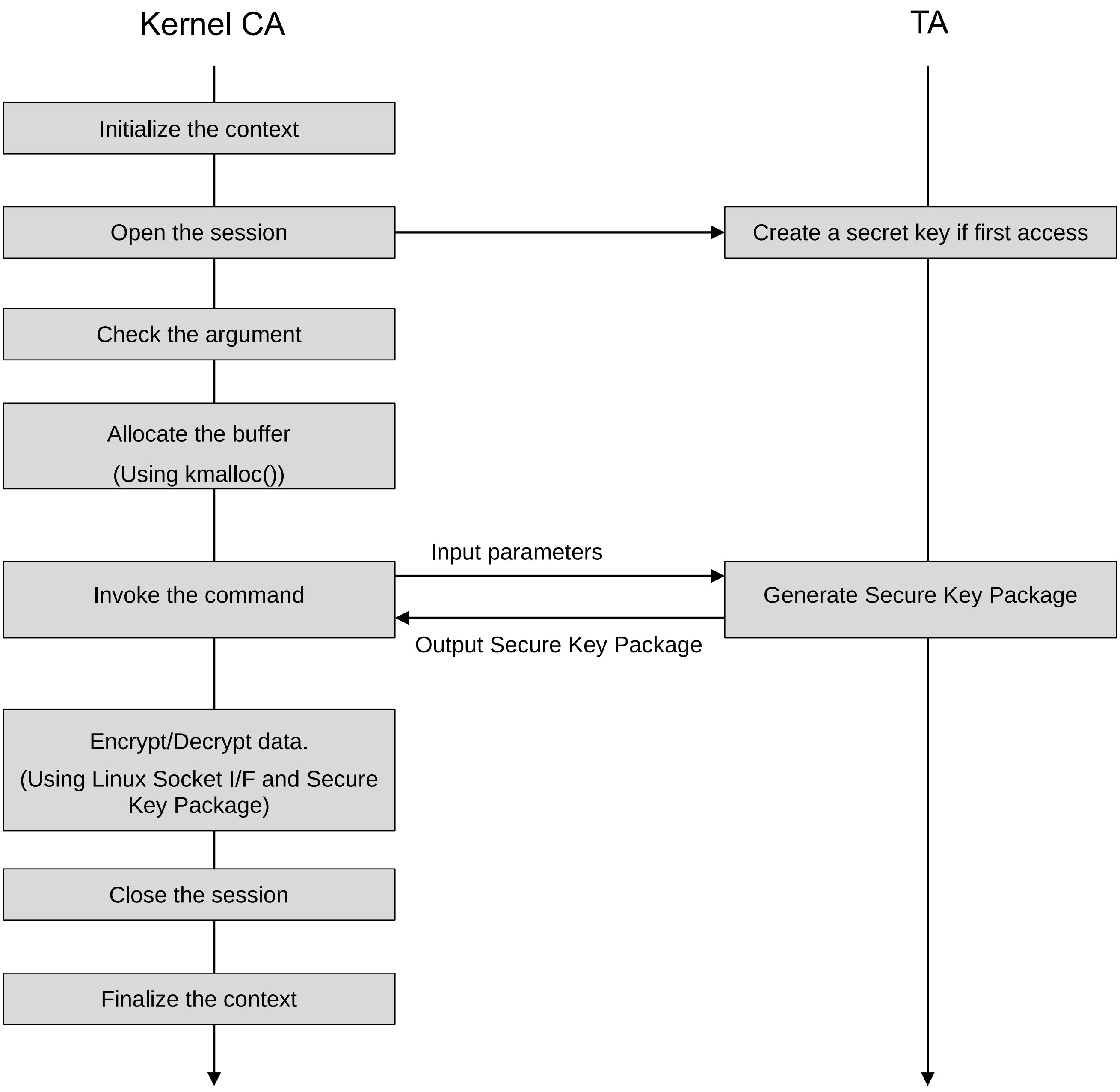
- Step 1: Encrypt the input file.
- Step 2: Decrypt the output result of step 1.

If result of step 2 is same with input file, SKP executes successfully.

7.4 Kernel space secure key package

7.4.1 Software processing flow

Figure 7-3: SKP processing flow with kernel-space CA



7.4.2 How to implement kernel space CA

The kernel space CA should be implemented as a normal kernel module. Below description are an example for its main content, user needs to implement more for their full CA. The content of CA in kernel space has two main parts as below.

- Getting secure key package data, main content is as below:

```
/* Open context */
ctr_ptr = tee_client_open_context(NULL, tp_version_check, &v, NULL);

/* Open session to call to TA (with session_arg.uuid is same UUID of TA) */
tee_client_open_session(ctr_ptr, &session_arg, &ta_param[0]);

/* Allocate memory to contain request command, share memory and secure key package */
tee_shm_alloc(ctr_ptr, MAX_KPKG_SIZE, TEE_SHM_MAPPED | TEE_SHM_DMA_BUF);

/* Invoke to TA to get secure key package. */
tee_client_invoke_func(ctr_ptr, &ivoke_arg, ta_param);
```

- Use secure key package data as key to encrypt/decrypt input file. This process bases on symmetric cipher flow of Linux crypto framework as at chapter 6.2.1.

```
/* Create a symmetric transform, with algorithm is "cbc(saes)"  or "ctr(saes)" */
tfm = crypto_alloc_skcipher(algoType, 0, 0);

/* Allocate request */
skcipher_request_alloc(tfm, GFP_KERNEL);

/* Set key for cipher. Secure Key Package is used at this step */
crypto_skcipher_setkey(tfm, data->key, data->key_size);

/* Set input/output, iv */
skcipher_request_set_crypt(req, sg, sg2, loopSize, ivBuf);

/* Execute the encrypt/decrypt */
crypto_skcipher_encrypt(req);  /  crypto_skcipher_decrypt(req);
```

7.4.3 Dynamic TA for SKP preparation

For Dynamic TA, it is used the same TA that is created at step 3 of chapter 7.3.3 How to implement user space CA and Dynamic TA.

REVISION HISTORY	CCREE Driver for Linux User's Manual: Software

Rev.	Date	Description	
		Page	Summary
2.50	Jun. 09th, 2020	—	Created for the initial release.
2.51	Dec. 01st, 2020	All	- Unification: change OPTEE to OP-TEE. - Unification: change ccree to CCREE Driver when calling CCREE module name.
		1	Table 1.1: - Add MULTI2 and Secure Key Algorithms. - No longer support DES algorithm and remove.
		3	Table 3.1: Add "Secure Key Package" definition.
		6	Figure 3.2: Add OP-TEE side and update explanation for secure key package.
		7	Add chapter 3.2.1.
		13	Chapter 5.4: Change related to "associated-data buffer fragment count" from 4 to 8.
		14	Add paragraph "How to input data in the user mode".
		27,28	Add chapter 7.
2.52	Aug 16th, 2021	1	Add "cts1(cbc(aes))" and cts1(cbc(saes)) into table 1.1.
		17,18	Add "How to" at beginning of sub-chapters in chapter 5.2.
		20,21	Add procedure to build and boot CCREE driver for Android BSP.
		All	Add R-Car D3 support
3.00	Dec. 10th, 2021	-	Update chapter 7 to show SKP usage and SKP source code implementation guide

CCREE Driver for Linux User’s Manual: Software

Publication Date: Rev.3.00 Dec. 10, 2021

Published by: Renesas Electronics Corporation

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口

http://www.renesas.com

※営業お問合せ窓口の住所は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス株式会社 〒135-0061 東京都江東区豊洲3-2-24（豊洲フォレシア）

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：https://www.renesas.com/contact/

Crypto Cell Rich Execution Environment Driver for Linux
User's Manual

Renesas Electronics Corporation