Rockchip Crypto/HWRNG Developer Guide

ID：RK-KF-YF-852

Release Version: V1.2.2

Release Date: 2023-03-15

Security Level: □Top-Secret □Secret □Internal ■Public

Rockchip Electronics Co., Ltd.

No.18 Building, A District, No.89, software Boulevard Fuzhou, Fujian,PRC

Website: www.rock-chips.com

Customer service Tel: +86-4007-700-590

Customer service Fax: +86-591-83951833

Customer service e-Mail: fae@rock-chips.com

Preface

Overview

This document mainly introduces the development of Rockchip Crypto and HWRNG(TRNG), including driver development and upper application development.

Product version

Chipset name	Kernel version
All Rockchip chips with crypto IP	Linux 4.19
All Rockchip chips with crypto IP	Linux 5.10

Intended Audience

This document (this guide) is mainly intended for:

Technical support engineers
Software development engineers

Revision history

Date	Author	Date	Revision description
V1.0.0	Lin Jinhan, Elon Zhang, Wang Xiaobin	2022-01-25	The initial version
V1.1.0	Elon Zhang	2022-02-28	Added a description of user space calling hwrng and other supplementary notes
V1.2.0	Lin Jinhan, Elon Zhang	2022-09-14	1. rk_crypto_mem_alloc added dma-heap allocation support 2. Add support for the cipher aead mode 3. Add support for the rsa algorithm 4. Add support for crypto v3 and trng v1 5. Add log printing level 6. librkcrypto added the kernel version dependency description
V1.2.1	Lin Jinhan	2022-10-09	Fixed description error
V1.2.2	Lin Jinhan	2023-03-15	Add more information to CRYPTO configuration

Contents

1. Overview

There are three versions of crypto IP on the RK platform currently, including crypto V1 / V2 / V3. The V1 and V2 IP versions support different algorithms and use different modes. V3 is based on V2, most of the code can be reuse. Most of the previous hardware random number modules of chip platforms are in the hardware crypto IP. Since RK356x, HWRNG (TRNG) is an independent hardware module.

1.1 crypto v1

Algorithm	Description
DES/TDES	Support DES/3DES (ECB and CBC chain mode)
AES	Support AES 128/192/256 bits key mode, ECB/CBC/CTR/XTS chain mode
HASH	Support SHA1/SHA256/MD5 (with hardware padding) HASH function
RSA	Support PKA 512/1024/2048 bit Exp Modulator (RK3126, RK3128, RK3288, and RK3368 are not supported)
TRNG	Support 256 bit True Random Number Generator (TRNG)

1.2 crypto v2

Algorithm	Description
DES/TDES	Support ECB/CBC/OFB/CFB mode.
AES	Support ECB/CBC/OFB/CFB/CTR/CTS/XTS/CCM/GCM/CBC-MAC/CMAC mode.
SM4	Support ECB/CBC/OFB/CFB/CTR/CTS/XTS/CCM/GCM/CBC-MAC/CMAC mode. (SM4 is optional)
HASH	Support MD5/SHA1/SHA224/SHA256/SHA384/SHA512/SM3/SHA512-224/SHA512-256 with hardware padding. (SM3 is optional)
HMAC	Support HMAC of SHA-1, SHA-256, SHA-512, MD5, SM3 with hardware padding. (SM3 is optional)
RSA/ECC	Support up to 4096 bits PKA mathematical operations for RSA/ECC/SM2.
TRNG	Support 256 bit True Random Number Generator (TRNG)

1.3 crypto v3

On the basis of Crypto V2 algorithm, Crypto V3 add multithreading support. The Cyrpto V3 platform has been able to automatically identify supported algorithms since RV1106, so compatible uses "rockchip,crypto-v3" as the identifier.

1.4 The crypto IP version of chips

The crypto IP version of chips show below：

The platforms that use crypto V1 are:

> RK3399、RK3288、RK3368、RK3328/RK3228H、RK322x、RK3128、RK1108、RK3126

The platforms that use crypto V2 are:

> RK3326/PX30、RK3308、RK1808、RV1126/RV1109、RK2206、RK356x、RK3588

The platforms that use crypto V3 are:

> RV1106

2. Driver Development

2.1 Driver Code Description

2.1.1 hwrng

Crypto V1 / V2, TRNGV1, RKRNG four platforms are centralized in the same c file because HWRNG driver is relatively simple.

The driver does not distinguish the specific chip model, only according to `"rockchip,cryptov1-rng"` and `"rockchip,cryptov2-rng"`, `"rockchip,trngv1"`, `"rockchip,rkrng"` four compatible partition. Currently, `'rockchip,trngv1'` and `"rockchip,rkrng"` is a separate HWRNG module, and the other two HWRNG are built into the Crypto module.

Driver Source Code: `drivers/char/hw_random/rockchip-rng.c`

2.1.2 crypto

The current driver implementation algorithm is as follows:

crypto v1:

- **AES**: ECB/CBC
- **DES/TDES**: ECB/CBC
- **HASH**: SHA1/SHA256/MD5

crypto v2: (Some chipset may not support all algorithm)

- **AES**: ECB/CBC/OFB/CFB/CTR/GCM
- **DES/TDES**: ECB/CBC/CFB/OFB
- **SM4**: ECB/CBC/OFB/CFB/OFB/CTR/GCM
- **HASH**: SHA1/SHA256/SHA384/SHA512/MD5/SM3
- **HMAC**: HMAC_SHA1/HMAC_SHA256/HMAC_SHA512/HMAC_MD5/HMAC_SM3
- **RSA**: Support up to 4096 bits key length.

crypto v2/v3 Hardware full version(The mode driver for the following deleted lines has not been implemented yet):

- **AES(128/192/256)**: ECB/CBC/OFB/CFB/CTR/~~XTS~~/~~CTS~~/~~CCM~~/GCM/~~CBC-MAC~~/~~CMAC~~
- **SM4**: ECB/CBC/OFB/CFB/CTR/~~XTS~~/~~CTS~~/~~CCM~~/GCM/~~CBC-MAC~~/~~CMAC~~
- **DES/TDES**: ECB/CBC/OFB/CFB
- **HASH**: MD5/SHA-1/SHA256/SHA512/SM3/SHA224/SHA384/~~SHA512_224~~/~~SHA512_384~~
- **HMAC**: SHA-1/SHA-256/SHA-512/MD5/SM3
- **RSA**: Support up to 4096 bits key length.

crypto v2/v3 Hardware difference table

Chip	AES	DES/TDES	SM3/SM4	HASH	HMAC	RSA	multi - thread
RK3326/PX30/RK3308	√	√	×	√	√	√	×
RK1808	AES-128	×	×	SHA-1SHA-224/SHA-256/MD5	√	√	×
RV1126/RV1109	AES-128/AES-256	√	√	√	√	√	×
RK2206	√	√	×	√	√	√	×
RK3568/RK3588	√	√	√	√	√	√	×
RV1106	√	√	×	SHA-1/SHA-224/SHA-256/MD5	√	√	√

Note:

1. RK1808 : AES supports only 128bit. For kernel drivers, AES is not supported.
2. RV1126/RV1109: AES-192 is not supported, so the AES-192 part can only be implemented by the soft algorithm, but the soft algorithm cannot support all the modes of the hard algorithm. Therefore, it is recommended not to change the list of configured algorithms in the code.

The driver file are show as below:

```
drivers/crypto/rockchip
|-- procfs.c                    // proc statistics info (clock rate, algo list,
etc.)
|-- procfs.h                    // proc head file
|-- rk_crypto_bignum.c          // crypto PKA bignum api
|-- rk_crypto_bignum.h          // crypto PKA bignum file
|-- rk_crypto_core.c            // linux crypto Driver framework and public
interface
|-- rk_crypto_core.h            // linux crypto common head file
|-- rk_crypto_ahash_utils.c     // ahash common api
|-- rk_crypto_ahash_utils.h     // ahash common head file
```

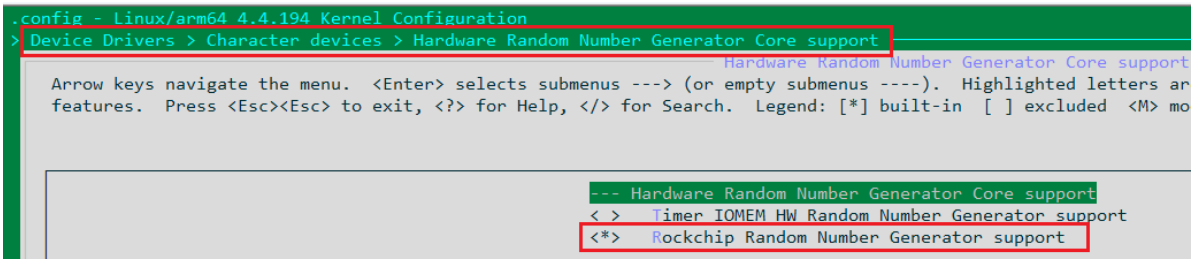
```
|-- rk_crypto_skcipher_utils.c    // skcipher common api
|-- rk_crypto_skcipher_utils.h    // skcipher common head file
|-- rk_crypto_utils.c             // crypto common api
|-- rk_crypto_utils.h             // crypto common head file
|-- rk_crypto_v1.c                // crypto v1 hardware related interface
implementation
|-- rk_crypto_v1.h                // crypto v1 structure and interface
declaration
|-- rk_crypto_v1_skcipher.c       // crypto v1 block cipher algorithm implement
|-- rk_crypto_v1_ahash.c          // crypto v1 hash algorithm implement
|-- rk_crypto_v1_reg.h            // crypto v1 hardware register definition
|-- rk_crypto_v2.c                // crypto v2 hardware related interface
implementation
|-- rk_crypto_v2.h                // crypto v2 structure and interface
declaration
|-- rk_crypto_v2_skcipher.c       // crypto v2 block cipher algorithm implement
|-- rk_crypto_v2_ahash.c          // crypto v2 hash algorithm implement
|-- rk_crypto_v2_akcipher.c       // crypto v2 RSA algorithm implement
|-- rk_crypto_v2_pka.c            // crypto v2 pka operation implement
|-- rk_crypto_v2_reg.h            // crypto v2 hardware register definition
|-- rk_crypto_v3.c                // crypto v3 Hardware related interface
implementation
|-- rk_crypto_v3.h                // crypto v3 Structure and interface
declaration
|-- rk_crypto_v3_skcipher.c       // crypto v3 block cipher algorithm implement
|-- rk_crypto_v3_ahash.c          // crypto v3 hash algorithm implement
|-- rk_crypto_v3_reg.h            // crypto v3 hardware register definition
`-- cryptodev_linux               // exporting the crypto interface to User space
```

2.2 Enable hwrng

2.2.1 Menuconfig

The HWRNG driver is compiled into the kernel by default, and the dts file determines whether to enable it.

The configuration is shown in the following figure (Red marks indicate the configuration path and the options that need to be configured).



Or add the following statement to the config file (which is configured by default in rockchip_defconfig) :

```
CONFIG_HW_RANDOM=y
CONFIG_HW_RANDOM_ROCKCHIP=y
```

2.2.2 Enable rng node in dts

At present, most chips dtsi file have been configured with HWRNG nodes, and you only need to enable the rng module in the board-level dts, as shown below:

```
&rng {
    status = "okay";
}
```

2.2.3 Added rng node for new chip

Most of the chip platforms have been configured with rng nodes. If the dtsi of chip has not been configured with rng nodes, you can perform the following operations to configure them.

Attention:

1. The RNG base address needs to be modified according to the chip TRM. The RNG base address is the CRYPTO base address.
2. Clocks macros may differ from platform to platform. If there is an error on dts, you can go to the `include/dt-bindings/clock` directory and execute `grep -rn CRYPTO` to find the corresponding clock macros name, as shown below:

   ```
   troy@inno:~/kernel/include/dt-bindings/clock$ grep -rn CRYPTO
   rk3328-cru.h:57:#define SCLK_CRYPTO             59
   rk3328-cru.h:206:#define HCLK_CRYPTO_MST        336
   rk3328-cru.h:207:#define HCLK_CRYPTO_SLV        337
   rk3328-cru.h:284:#define SRST_CRYPTO             68
   ```

crypto v1:

```
rng: rng@ff060000 {
    compatible = "rockchip,cryptov1-rng";
    reg = <0x0 0xff060000 0x0 0x4000>;
    clocks = <&cru SCLK_CRYPTO>, <&cru HCLK_CRYPTO_SLV>;
    clock-names = "clk_crypto", "hclk_crypto";
    assigned-clocks = <&cru SCLK_CRYPTO>, <&cru HCLK_CRYPTO_SLV>;
    assigned-clock-rates = <150000000>, <100000000>;
    status = "disabled";
};
```

crypto v2:

The actual TRNG does not need to rely on all clocks, but only on hclk_crypto

```
rng: rng@ff500400 {
    compatible = "rockchip,cryptov2-rng";
    reg = <0xff500400 0x80>; # Need to add 0x400 if rng is inside crypto
    clocks = <&cru HCLK_CRYPTO>;
    clock-names = "hclk_crypto";
    power-domains = <&power RV1126_PD_CRYPTO>;
    resets = <&cru SRST_CRYPTO_CORE>;
    reset-names = "reset";
    status = "disabled";
};
```

trng v1:

At present, RK3588 and RV1106 use the random number module of TRNG V1, which is completely different from the TRNG module split in Crypto V2 in design and improves the randomness.

```
rng: rng@fe378000 {
    compatible = "rockchip,trngv1";
    reg = <0x0 0xfe378000 0x0 0x200>;
    interrupts = <GIC_SPI 400 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&scmi_clk SCMI_HCLK_SECURE_NS>;
    clock-names = "hclk_trng";
    resets = <&scmi_reset SRST_H_TRNG_NS>;
    reset-names = "reset";
    status = "disabled";
};
```

2.2.4 Verify that HWRNG is enabled

1. Execute `cat /sys/devices/virtual/misc/hw_random/rng_current` can see the information as the rockchip, determine the current call is hardware driver.
2. linux: execute `cat /dev/hwrng | od -x  | head -n 1` will get a row of random numbers, and every time you run it, the content of the random number is different.
3. Android: execute `cat /dev/hw_random | od -x  | head -n 1` will get a row of random numbers, and every time you run it, the content of the random number is different.

2.3 Enable hardware crypto

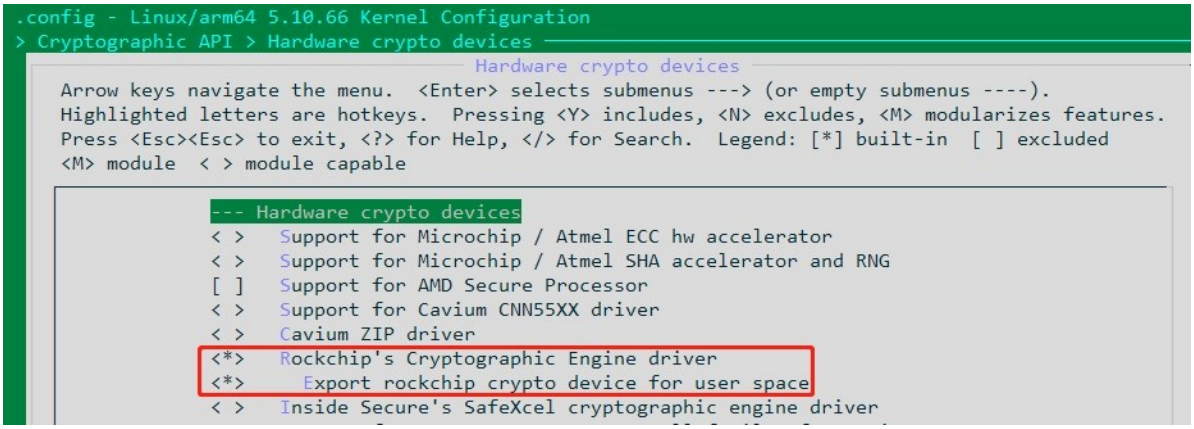
Current driver code crypto v1 support rk3328, crypto v2 support px30 / rv1126 / rk3568 / rk3588, crypto v3 support rv1106 . For the above platforms, just enable config and dts node to enable hardware crypto.

The supported features will be reflected in the register information since RV1106, and Crypto V3 can automatically adapt to the new chip.

2.3.1 Menuconfig

After menuconfig configuration, Rockchip crypto driver will be compiled into kernel. It will automatically adapt crypto IP version according to the chip platform compatible ID.

You can see the related configuration items of hardware crypto only after ensuring that `CONFIG_CRYPTO_HW` is enabled.



Or add the following statement to the config file. `CONFIG_CRYPTO_DEV_ROCKCHIP_V3` is just an example. You must change it according to the actual chip configuration. It is recommended to use menuconfig, which will automatically select the platform.

```
CONFIG_CRYPTO_HW=y
CONFIG_CRYPTO_DEV_ROCKCHIP=y
CONFIG_CRYPTO_DEV_ROCKCHIP_V3=y
CONFIG_CRYPTO_DEV_ROCKCHIP_DEV=y
```

2.3.2 Enable crypto node dts

After confirming that the crypto dts node is correctly configured, just enable the crypto module in the board-level dts file, as shown in the following:

```
&crypto {
    status = "okay";
};
```

2.3.3 Added crypto node for new chip

If there is no crypto node in chip dtsi file, follow these steps to add support.

1. Determine the version of the chip crypto IP V1 / V2 /V3. V3 version from RV1106, compatible has been determined as `"rockchip,crypto-v3"`, algorithm tailoring and feature are adapted by the software..
2. `drivers/crypto/rockchip/rk_crypto_core.c` add algs_name， soc_data，compatible info。

```
/* Add the algorithm information supported by the chip. Px30 belongs to
crypto V2. For the supported algorithms, see crypto_V2_algs */
/* Attention: Crypto_v2_algs is full algorithms supported by crypto V2。*/
/* Some chips are trimmed on Crypto V2. For example, RK1808 does not support
SHA512 algorithm, so it is necessary to compare TRM to confirm the algorithm
supported */
static char *px30_algs_name[] = {
    "ecb(aes)", "cbc(aes)", "xts(aes)",
    "ecb(des)", "cbc(des)",
    "ecb(des3_ede)", "cbc(des3_ede)",
    "sha1", "sha256", "sha512", "md5",
};

/* bind px30_algs_name to px30_soc_data */
```

```
static const struct rk_crypto_soc_data px30_soc_data =
    RK_CRYPTO_V2_SOC_DATA_INIT(px30_algs_name, false);

/* bind px30_soc_data to id_table */
static const struct of_device_id crypto_of_id_table[] = {
    /* crypto v2 in belows */
    {
        .compatible = "rockchip,px30-crypto",
        .data = (void *)&px30_soc_data,
    },
    {
        .compatible = "rockchip,rv1126-crypto",
        .data = (void *)&rv1126_soc_data,
    },
    /* crypto v1 in belows */
    {
        .compatible = "rockchip,rk3288-crypto",
        .data = (void *)&rk3288_soc_data,
    },
    { /* sentinel */ }
};
```

3. Add crypto node for new chip

   Attention:

   1. Determine the CRYPTO base address according to the chip TRM
   2. Clocks macros may differ from platform to platform. If there is an error on dts, you can go to `include/dt-bindings/clock` and `grep -rn CRYPTO` to find the corresponding clock macros name, as shown below:

      ```
      troy@inno:~/kernel/include/dt-bindings/clock$ grep -rn CRYPTO
      rk3328-cru.h:57:#define SCLK_CRYPTO             59
      rk3328-cru.h:206:#define HCLK_CRYPTO_MST        336
      rk3328-cru.h:207:#define HCLK_CRYPTO_SLV        337
      rk3328-cru.h:284:#define SRST_CRYPTO             68
      ```

   crypto v1:

   ```
   crypto: cypto-controller@ff8a0000 {                    /* crypto base
   address */
       compatible = "rockchip,rk3288-crypto";             /* platform "rk3399-
   crypto" */
       reg = <0x0 0xff8a0000 0x0 0x4000>;                 /* crypto base
   address */
       interrupts = <GIC_SPI 48 IRQ_TYPE_LEVEL_HIGH>;     /* crypto interrupts
   number */
       clocks = <&cru ACLK_CRYPTO>, <&cru HCLK_CRYPTO>,
                <&cru SCLK_CRYPTO>, <&cru ACLK_DMAC1>;
       clock-names = "aclk", "hclk", "sclk", "apb_pclk";
       resets = <&cru SRST_CRYPTO>;
       reset-names = "crypto-rst";
       status = "disabled";
   };
   ```

   crypto v2:

For most crypto V2 chips, the register address of HWRNG is in the middle of crypto. Therefore, you need to split the crypto address space into two parts when configuring reg address. The first part is the register used by the CIPHER and the second part is the register used by RSA.

```
---------------------- reg map ----------------------
|      cipher/hash     |      rng     |     pka     |
-----------------------------------------------------
```

```
crypto: crypto@ff500000 {                                   /* crypto base
address */
    compatible = "rockchip,rv1126-crypto";                  /* modify platform
*/
    reg = <0xff500000 0x400>, <0xff500480 0x3B80>;          /* crypto base
address */
    interrupts = <GIC_SPI 3 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&cru CLK_CRYPTO_CORE>, <&cru CLK_CRYPTO_PKA>,
    <&cru ACLK_CRYPTO>, <&cru HCLK_CRYPTO>;
    clock-names = "aclk", "hclk", "sclk", "apb_pclk";
    power-domains = <&power RV1126_PD_CRYPTO>;
    resets = <&cru SRST_CRYPTO_CORE>;
    reset-names = "crypto-rst";
    status = "disabled";
};
```

4. enable crypto node

```
&crypto {
    status = "okay";
};
```

2.3.4 Verify that hardware crypto is enabled

You can view the registered rockchip hardware crypto algorithm by running commands `cat /proc/crypto | grep rk`.(Take RV1126 as an example)

```
driver       : pkcs1pad(rsa-rk,sha256)
driver       : rsa-rk
driver       : hmac-sm3-rk
driver       : hmac-md5-rk
driver       : hmac-sha512-rk
driver       : hmac-sha256-rk
driver       : hmac-sha1-rk
driver       : sm3-rk
driver       : md5-rk
driver       : sha512-rk
driver       : sha256-rk
driver       : sha1-rk
driver       : ofb-des3_ede-rk
driver       : cfb-des3_ede-rk
driver       : cbc-des3_ede-rk
driver       : ecb-des3_ede-rk
driver       : ofb-des-rk
driver       : cfb-des-rk
```

```
driver       : cbc-des-rk
driver       : ecb-des-rk
driver       : xts-aes-rk
driver       : ctr-aes-rk
driver       : cfb-aes-rk
driver       : cbc-aes-rk
driver       : ecb-aes-rk
driver       : xts-sm4-rk
driver       : ctr-sm4-rk
driver       : ofb-sm4-rk
driver       : cfb-sm4-rk
driver       : cbc-sm4-rk
driver       : ecb-sm4-rk
```

You can view the information about the Rockchip crypto driver by running the `cat /proc/rkcrypto` command. (Make sure to update the latest code. The old code does not support this function.) Including crypto version, clock frequency, currently available algorithms, and some statistics of current driver operation, will continue to improve and supplement in the future. ("CRYPTO V3.0.0.0 multi" indicates that the current platform supports CRYPTO support multithreading).

```
Rockchip Crypto Version: CRYPTO V2.0.0.0

use_soft_aes192 : false

clock info:
        aclk        350000000
        hclk        150000000
        sclk        350000000
        pka         350000000

Valid algorithms:
        CIPHER:
                ecb(sm4)
                cbc(sm4)
                cfb(sm4)
                ofb(sm4)
                ctr(sm4)
                ecb(aes)
                cbc(aes)
                cfb(aes)
                ofb(aes)
                ctr(aes)
                ecb(des)
                cbc(des)
                cfb(des)
                ofb(des)
                ecb(des3_ede)
                cbc(des3_ede)
                cfb(des3_ede)
                ofb(des3_ede)

        AEAD:
                gcm(sm4)
                gcm(aes)

        HASH:
                sha1
```

```
                sha224
                sha256
                sha384
                sha512
                md5
                sm3

        HMAC:
                hmac(sha1)
                hmac(sha256)
                hmac(sha512)
                hmac(md5)
                hmac(sm3)

        ASYM:
                rsa

Statistic info:
        busy_cnt      : 1
        equeue_cnt    : 28764
        dequeue_cnt   : 28765
        done_cnt      : 310710
        complete_cnt  : 28765
        fake_cnt      : 0
        irq_cnt       : 310710
        timeout_cnt   : 0
        error_cnt     : 0
        last_error    : 0

Crypto queue usage [0/50], ever_max = 1, status: idle
```

3. User space development

3.1 user space invoke hwrng

There are two ways for user space to obtain random numbers output by hwrng:

- read kernel driver node
- invoke librkcrypto API

Attention:

1. After the hwrng hardware driver is successfully registered, entropy can be added to the kernel random driver. The random numbers generated by hwrng are input to the entropy pool of the random driver. The random driver of kernel is CSPRNG (Cryptography Secure Pseudo Random Number Generator), which meets the security standards of cryptography. For high quality random numbers,, you can read `/dev/random` or `/dev/urandom` nodes for random numbers.

3.1.1 read kernel driver node

If the kernel has enabled rng module, random numbers can be obtained by reading nodes in the user space. **Linux reads the node `/dev/hwrng`, Android reads the node `/dev/hw_random`.** Reference code is as follows:

```
#ifdef ANDROID
#define HWRNG_NODE      "/dev/hw_random"
#else
#define HWRNG_NODE      "/dev/hwrng"
#endif

RK_RES rk_get_random(uint8_t *data, uint32_t len)
{
        RK_RES res = RK_CRYPTO_SUCCESS;
        int hwrng_fd = -1;
        int read_len = 0;

        hwrng_fd = open(HWRNG_NODE, O_RDONLY, 0);
        if (hwrng_fd < 0) {
                E_TRACE("open %s error!", HWRNG_NODE);
                return RK_CRYPTO_ERR_GENERIC;
        }

        read_len = read(hwrng_fd, data, len);
        if (read_len != len) {
                E_TRACE("read %s error!", HWRNG_NODE);
                res = RK_CRYPTO_ERR_GENERIC;
        }

        close(hwrng_fd);

        return res;
}
```

3.1.2 invoke librkcrypto API

Refer to the API description: rk_get_random.

3.2 user space invoke hardware crypto

user space calls the librkcrypto interface to operate the hardware cipher module. This section provides instructions for librkcrypto.

Attention： Check whether hardware crypto has been enabled in the kernel before use, and the enabling method and confirmation method refer toEnable hardware crypto and Verify that hardware crypto is enabled.

3.2.1 Scope of application

API	RK3588	RK356x	RV1109/1126	others
rk_crypto_mem_alloc/free	√	√	√	
rk_crypto_init/deinit	√	√	√	
rk_get_random	√	√	√	
rk_hash_init/update/update_virt/final	√	√	√	
rk_cipher_init/crypt/crypt_virt/final	√	√	√	
rk_ae_init/set_aad/set_aad_virt/crypt/crypt_virt/final	√	√	√	
rk_rsa_pub_encrypt/priv_decrypt/priv_encrypt/pub_decrypt	√	√	√	
rk_rsa_sign/verify	√	√	√	
rk_write_oem_otp_key	√	√	√	
rk_oem_otp_key_is_written	√	√	√	
rk_set_oem_hr_otp_read_lock	√			
rk_oem_otp_key_cipher	√	√	√	
rk_oem_otp_key_cipher_virt	√	√	√	

3.2.2 Version Dependencies

3.2.2.1 V1.2.0

Librkcrypto V1.2.0 library functions depend on the following kernel commit points. If the kernel crypto driver is not updated to the following commit points, some functions may be unavailable.

1. kernel 4.19

```
commit c255a0aa097afbf7f28e3c0770c5ab778e5616b2
Author: Lin Jinhan <troy.lin@rock-chips.com>
Date:   Tue Sep 13 17:20:46 2022 +0800

    crypto: rockchip: rk3326/px30 add aes gcm support

    Signed-off-by: Lin Jinhan <troy.lin@rock-chips.com>
    Change-Id: I75949554d4f573c63092841eef76765a69cc6b24
```

2. kernel 5.10

```
commit 47e85085826daf6401265b803ac9ac7116ae6bb4
Author: Lin Jinhan <troy.lin@rock-chips.com>
Date:   Tue Sep 13 17:20:46 2022 +0800

    crypto: rockchip: rk3326/px30 add aes gcm support

    Signed-off-by: Lin Jinhan <troy.lin@rock-chips.com>
    Change-Id: I75949554d4f573c63092841eef76765a69cc6b24
```

3.2.3 Attention

- **The input data length of the symmetric algorithm is required to be consistent with the data length of the selected algorithm and mode.** For example, ECB and CBC require block alignment, while CTS and CTR do not require data length alignment. There's no padding in the API.
- **If the amount of calculated data is pretty large, the algorithm interface that transmits data through dma_fd is recommended to improve the efficiency.** Since crypto only supports contiguous physical addresses up to 4G, the buffer allocated by dma fd must be physically contiguous addresses up to 4G (CMA). It can be allocated using the `rk_crypto_mem interface` provided by librkcrypto, or allocated by yourself using a memory allocation interface such as DRM to get dma fd.
- **CMA config：** Crypto only supports access to CMA addresses up to 4G. If the device uses more than 4G memory, you need to modify the CMA configuration in dts. Otherwise, `rk_crypto_mem` can be allocated successfully, but the allocated memory cannot be used.The following takes rk3588-android.dtsi platform as an example. In the preceding command, 0x10000000 indicates the start address of the CMA (256MB, do not change the value), and 0x00800000 indicates the size of the CMA. You can change the value as required. For details about the CMA, see the documentation <Rockchip_Developer_Guide_Linux_CMA_CN>。

```
--- a/arch/arm64/boot/dts/rockchip/rk3588-android.dtsi
+++ b/arch/arm64/boot/dts/rockchip/rk3588-android.dtsi
@@ -70,7 +70,8 @@
                cma {
                        compatible = "shared-dma-pool";
                        reusable;

-                       size = <0x0 (8 * 0x100000)>;
+                       //size = <0x0 (8 * 0x100000)>;
+                       reg = <0x0 0x10000000 0x0 0x00800000>;
                        linux,cma-default;
                };
```

- **Before using the following interface, ensure that the TEE function is available. See <Rockchip_Developer_Guide_TEE_SDK_CN> for details.**

> rk_write_oem_otp_key
>
> rk_oem_otp_key_is_written
>
> rk_set_oem_hr_otp_read_lock
>
> rk_oem_otp_key_cipher
>
> rk_oem_otp_key_cipher_virt

- **`rk_set_oem_hr_otp_read_lock`：If key_id is set to `RK_OEM_OTP_KEY0/1/2`, the attributes of other OTP areas will be affected after the setting is successful.** For example, some OTP regions become unwritable, see the <Rockchip_Developer_Guide_OTP_CN> document. Therefore, `RK_OEM_OTP_KEY3` is recommended.
- **`rk_oem_otp_key_cipher_virt`：The maximum len value supported is affected by the shared memory of the TEE, which may be smaller than expected if the TEE shared memory is occupied before this interface is used.**

3.2.4 Data Structure

3.2.4.1 rk_crypto_mem

```
typedef struct {
    void                *vaddr;
    int                 dma_fd;
    size_t              size;
} rk_crypto_mem;
```

- vaddr - virtual address of memory
- dma_fd - dma_fd of memory
- size - size of memory

3.2.4.2 rk_cipher_config

```
typedef struct {
        uint32_t        algo;
        uint32_t        mode;
        uint32_t        operation;
        uint8_t         key[64];
        uint32_t        key_len;
        uint8_t         iv[16];
        void            *reserved;
} rk_cipher_config;
```

- algo - algorithms type, See RK_CRYPTO_ALGO. The actual value range is subject to the API description. The same below.
- mode - algorithms mode, See RK_CIPIHER_MODE. Support ECB/CBC/CTR/CFB/OFB/ mode.
- operation - encrypt or decrypt, See RK_CRYPTO_OPERATION.
- key - plaintext key. Invalid when otp key is used.
- key_len - key length (Unit: byte)
- iv - the initial vector is not valid in ECB mode. In other modes, executing `rk_cipher_crypt/crypt_virt` will automatically update iv for multiple segmentation calculations
- reserved - for future use

3.2.4.3 rk_ae_config

```
typedef struct {
        uint32_t        algo;
        uint32_t        mode;
        uint32_t        operation;
        uint8_t         key[32];
        uint32_t        key_len;
        uint8_t         iv[16];
        uint32_t        iv_len;
        uint32_t        tag_len;
        uint32_t        aad_len;
        uint32_t        payload_len;
        void            *reserved;
} rk_ae_config;
```

- algo - algorithms type, See RK_CRYPTO_ALGO. Support AES/SM4.
- mode - algorithms mode, See RK_CIPIHER_MODE. Support GCM/CCM.
- operation - encrypt or decrypt, See RK_CRYPTO_OPERATION.
- key - plaintext key. Invalid when otp key is used.
- key_len - key length (Unit: byte)
- iv - initial vector
- iv_len - initial vector length (Unit: byte)
- tag_len - tag length (Unit: byte)
- aad_len - aad length (Unit: byte)
- payload_len - payload length (Unit: byte)
- reserved - for future use

3.2.4.4 rk_hash_config

```
typedef struct {
        uint32_t        algo;
        uint8_t         *key;
        uint32_t        key_len;
} rk_hash_config;
```

- algo - algorithms type, See RK_CRYPTO_ALGO. Support HASH/HMAC.
- key - key for hash-mac, only valid when algo is HMAC
- key_len - key length (Unit: byte)

3.2.4.5 rk_rsa_pub_key

```
typedef struct {
    const uint8_t       *n;
    const uint8_t       *e;

    uint16_t        n_len;
    uint16_t        e_len;
} rk_rsa_pub_key;
```

- n - module, same as OpenSSL, big-endian mode
- e - exponent, same as OpenSSL, big-endian mode
- n_len - module length

- e_len - exponent length

3.2.4.6 rk_rsa_pub_key_pack

```
typedef struct {
    enum RK_RSA_KEY_TYPE    key_type;
    rk_rsa_pub_key      key;
} rk_rsa_pub_key_pack;
```

- key_type - key type, See RK_RSA_KEY_TYPE . Support plaintext key and OTP_KEY encrypted ciphertext key, librkcrypto will pass the key to crypto driver and decrypt it with the corresponding otp key before use.
- key - public key, See rk_rsa_pub_key .

3.2.4.7 rk_rsa_priv_key

```
typedef struct {
    const uint8_t       *n;
    const uint8_t       *e;
    const uint8_t       *d;
    const uint8_t       *p;
    const uint8_t       *q;
    const uint8_t       *dp;
    const uint8_t       *dq;
    const uint8_t       *qp;

    uint16_t        n_len;
    uint16_t        e_len;
    uint16_t        d_len;
    uint16_t        p_len;
    uint16_t        q_len;
    uint16_t        dp_len;
    uint16_t        dq_len;
    uint16_t        qp_len;
} rk_rsa_priv_key;
```

- n - module, same as OpenSSL, big-endian mode
- e - exponent, same as OpenSSL, big-endian mode
- d - private key, same as OpenSSL, big-endian mode
- p - optional
- q - optional
- dp - optional
- dq - optional
- qp - optional
- len - the length information of each element will not be described here

3.2.4.8 rk_rsa_priv_key_pack

```
typedef struct {
    enum RK_RSA_KEY_TYPE    key_type;
    rk_rsa_priv_key     key;
} rk_rsa_priv_key_pack;
```

- key_type - key type, See RK_RSA_KEY_TYPE . Support plaintext key and OTP_KEY encrypted ciphertext key, librkcrypto will pass the key, with the corresponding otp key decrypted before use.
- key - private key, See rk_rsa_priv_key .

3.2.5 Constant

3.2.5.1 RK_CRYPTO_ALGO

```
/* crypto algorithm */
enum RK_CRYPTO_ALGO {
        RK_ALGO_CIPHER_TOP = 0x00,
        RK_ALGO_AES,
        RK_ALGO_DES,
        RK_ALGO_TDES,
        RK_ALGO_SM4,
        RK_ALGO_CIPHER_BUTT,

        RK_ALGO_HASH_TOP = 0x10,
        RK_ALGO_MD5,
        RK_ALGO_SHA1,
        RK_ALGO_SHA256,
        RK_ALGO_SHA224,
        RK_ALGO_SHA512,
        RK_ALGO_SHA384,
        RK_ALGO_SHA512_224,
        RK_ALGO_SHA512_256,
        RK_ALGO_SM3,
        RK_ALGO_HASH_BUTT,

        RK_ALGO_HMAC_TOP = 0x20,
        RK_ALGO_HMAC_MD5,
        RK_ALGO_HMAC_SHA1,
        RK_ALGO_HMAC_SHA256,
        RK_ALGO_HMAC_SHA512,
        RK_ALGO_HMAC_SM3,
        RK_ALGO_CMAC_AES,
        RK_ALGO_CBCMAC_AES,
        RK_ALGO_CMAC_SM4,
        RK_ALGO_CBCMAC_SM4,
        RK_ALGO_HMAC_BUTT,
};
```

3.2.5.2 RK_CIPIHER_MODE

```
/* crypto mode */
enum RK_CIPIHER_MODE {
        RK_CIPHER_MODE_ECB = 0x00,
        RK_CIPHER_MODE_CBC,
        RK_CIPHER_MODE_CTS,
        RK_CIPHER_MODE_CTR,
        RK_CIPHER_MODE_CFB,
        RK_CIPHER_MODE_OFB,
        RK_CIPHER_MODE_XTS,
        RK_CIPHER_MODE_CCM,
        RK_CIPHER_MODE_GCM,
        RK_CIPHER_MODE_BUTT
};
```

3.2.5.3 RK_OEM_HR_OTP_KEYID

```
enum RK_OEM_OTP_KEYID {
        RK_OEM_OTP_KEY0 = 0,
        RK_OEM_OTP_KEY1,
        RK_OEM_OTP_KEY2,
        RK_OEM_OTP_KEY3,

        RK_OEM_OTP_KEY_FW = 10,         //key id of fw_encryption_key
        RK_OEM_OTP_KEYMAX
};
```

3.2.5.4 RK_CRYPTO_OPERATION

```
/* Algorithm operation */
#define RK_OP_CIPHER_ENC        1
#define RK_OP_CIPHER_DEC        0
```

3.2.5.5 RK_RSA_KEY_TYPE

```
enum RK_RSA_KEY_TYPE {
    RK_RSA_KEY_TYPE_PLAIN = 0,
    RK_RSA_KEY_TYPE_KEY0_ENC = RK_OEM_OTP_KEY0 +1,
    RK_RSA_KEY_TYPE_KEY1_ENC,
    RK_RSA_KEY_TYPE_KEY2_ENC,
    RK_RSA_KEY_TYPE_KEY3_ENC,
    RK_RSA_KEY_TYPE_MAX,
};
```

3.2.5.6 RK_RSA_CRYPT_PADDING

```
enum RK_RSA_CRYPT_PADDING {
    RK_RSA_CRYPT_PADDING_NONE = 0x00,    /* without padding */
    RK_RSA_CRYPT_PADDING_BLOCK_TYPE_0,  /* PKCS#1 block type 0 padding*/
    RK_RSA_CRYPT_PADDING_BLOCK_TYPE_1,  /* PKCS#1 block type 1padding*/
    RK_RSA_CRYPT_PADDING_BLOCK_TYPE_2,  /* PKCS#1 block type 2 padding*/
    RK_RSA_CRYPT_PADDING_OAEP_SHA1,     /* PKCS#1 RSAES-OAEP-SHA1 padding*/
    RK_RSA_CRYPT_PADDING_OAEP_SHA224,   /* PKCS#1 RSAES-OAEP-SHA224 padding*/
    RK_RSA_CRYPT_PADDING_OAEP_SHA256,   /* PKCS#1 RSAES-OAEP-SHA256   padding*/
    RK_RSA_CRYPT_PADDING_OAEP_SHA384,   /* PKCS#1 RSAES-OAEP-SHA384   padding*/
    RK_RSA_CRYPT_PADDING_OAEP_SHA512,   /* PKCS#1 RSAES-OAEP-SHA512   padding*/
    RK_RSA_CRYPT_PADDING_PKCS1_V1_5,    /* PKCS#1 RSAES-PKCS1_V1_5    padding*/
};
```

3.2.5.7 RK_RSA_SIGN_PADDING

```
enum RK_RSA_SIGN_PADDING {
    RK_RSA_SIGN_PADDING_PKCS1_V15_SHA1 = 0x100,/* PKCS#1 RSASSA_PKCS1_V15_SHA1
signature*/
    RK_RSA_SIGN_PADDING_PKCS1_V15_SHA224,   /* PKCS#1 RSASSA_PKCS1_V15_SHA224
signature*/
    RK_RSA_SIGN_PADDING_PKCS1_V15_SHA256,   /* PKCS#1 RSASSA_PKCS1_V15_SHA256
signature*/
    RK_RSA_SIGN_PADDING_PKCS1_V15_SHA384,   /* PKCS#1 RSASSA_PKCS1_V15_SHA384
signature*/
    RK_RSA_SIGN_PADDING_PKCS1_V15_SHA512,   /* PKCS#1 RSASSA_PKCS1_V15_SHA512
signature*/
    RK_RSA_SIGN_PADDING_PKCS1_PSS_SHA1, /* PKCS#1 RSASSA_PKCS1_PSS_SHA1
signature*/
    RK_RSA_SIGN_PADDING_PKCS1_PSS_SHA224,   /* PKCS#1 RSASSA_PKCS1_PSS_SHA224
signature*/
    RK_RSA_SIGN_PADDING_PKCS1_PSS_SHA256,   /* PKCS#1 RSASSA_PKCS1_PSS_SHA256
signature*/
    RK_RSA_SIGN_PADDING_PKCS1_PSS_SHA384,   /* PKCS#1 RSASSA_PKCS1_PSS_SHA1
signature*/
    RK_RSA_SIGN_PADDING_PKCS1_PSS_SHA512,   /* PKCS#1 RSASSA_PKCS1_PSS_SHA256
signature*/
};
```

3.2.5.8 Other Constants

```
/* Algorithm block length */
#define DES_BLOCK_SIZE          8
#define AES_BLOCK_SIZE          16
#define SM4_BLOCK_SIZE          16
#define SHA1_HASH_SIZE          20
#define SHA224_HASH_SIZE        28
#define SHA256_HASH_SIZE        32
#define SHA384_HASH_SIZE        48
#define SHA512_HASH_SIZE        64
#define MD5_HASH_SIZE           16
```

```
#define SM3_HASH_SIZE           32
#define AES_AE_DATA_BLOCK       128
#define MAX_HASH_BLOCK_SIZE     128
#define MAX_TDES_KEY_SIZE       24
#define MAX_AES_KEY_SIZE        32
#define MAX_AE_TAG_SIZE         16

#define RSA_BITS_1024           1024
#define RSA_BITS_2048           2048
#define RSA_BITS_3072           3072
#define RSA_BITS_4096           4096
#define MAX_RSA_KEY_BITS        RSA_BITS_4096

#define RK_CRYPTO_MAX_DATA_LEN  (1 * 1024 * 1024)
```

3.2.6 API

3.2.6.1 Data Type

```
typedef uint32_t RK_RES;
typedef uint32_t rk_handle;
```

3.2.6.2 Return Codes

```
/* API return codes */
#define RK_CRYPTO_SUCCESS                   0x00000000
#define RK_CRYPTO_ERR_GENERIC               0xF0000000
#define RK_CRYPTO_ERR_PARAMETER             0xF0000001
#define RK_CRYPTO_ERR_STATE                 0xF0000002
#define RK_CRYPTO_ERR_NOT_SUPPORTED         0xF0000003
#define RK_CRYPTO_ERR_OUT_OF_MEMORY         0xF0000004
#define RK_CRYPTO_ERR_ACCESS_DENIED         0xF0000005
#define RK_CRYPTO_ERR_BUSY                  0xF0000006
#define RK_CRYPTO_ERR_TIMEOUT               0xF0000007
#define RK_CRYPTO_ERR_UNINITED              0xF0000008
#define RK_CRYPTO_ERR_KEY                   0xF0000009
#define RK_CRYPTO_ERR_VERIFY                0xF000000A
#define RK_CRYPTO_ERR_PADDING               0xF000000B
#define RK_CRYPTO_ERR_PADDING_OVERFLOW      0xF000000C
#define RK_CRYPTO_ERR_MAC_INVALID           0xF000000D
```

3.2.6.3 rk_crypto_mem_alloc

```
rk_crypto_mem *rk_crypto_mem_alloc(size_t size);
```

function

To request a block of memory, `rk_crypto_mem` is returned, which contains the virtual address of the memory and dma_fd.

parameters

- [in] size - size of the memory to be alloced
- [out] memory - memory address info, See rk_crypto_mem .

Attention

1. The maximum memory size depends on the kernel CMA buffer size and usage.

3.2.6.4 rk_crypto_mem_free

```
void rk_crypto_mem_free(rk_crypto_mem *memory);
```

function

Free the memory allocated by `rk_crypto_mem_alloc`.

parameters

- [in] memory - memory address info, See rk_crypto_mem .

3.2.6.5 rk_crypto_init

```
RK_RES rk_crypto_init(void);
```

function

crypto initialization, such as open device nodes, etc.

parameters

- None

3.2.6.6 rk_crypto_deinit

```
void rk_crypto_deinit(void);
```

function

Release crypto resources, such as close device nodes.

parameters

- None

3.2.6.7 rk_hash_init

```
RK_RES rk_hash_init(rk_hash_config *config, rk_handle *handle);
```

function

Initializes the hash algorithm, support MD5/SHA1/SHA224/SHA256/SHA384/SHA512/SM3。

parameters

- [in] config - hash/hmac configure
- [out] handle - hash/hmac handle

Attention

1. After `rk_hash_init` succeeds, `rk_hash_final` must be called to destroy resources regardless of whether `rk_hash_update()` or `rk_hash_update_virt` was executed successfully or not.
2. If `rk_hash_init` returns RK_CRYPTO_ERR_BUSY, the platform does not support multithreading and only one handle can work at a time. You need to wait for the previous handle to be released before applying for a new handle.

3.2.6.8 rk_hash_update

```
RK_RES rk_hash_update(rk_handle handle, int data_fd, uint32_t data_len);
```

function

Receive dma_fd of data as input, calculates the hash/hmac value, and supports multiple packet calculations.

parameters

- [in] handle - hash/hmac handle
- [in] data_fd - handle of data to be calculated
- [in] data_len - data length (Unit: byte)

Attention

1. handle must be initialized with `rk_hash_init()`.
2. It can be called multiple times to feed the data that needs to be computed.
3. The data length data_len must be 64 bytes aligned if data is not the last group of data.

3.2.6.9 rk_hash_update_virt

```
RK_RES rk_hash_update_virt(rk_handle handle, uint8_t *data, uint32_t data_len);
```

function

Receive virtual address data as input and computes the hash value. Supports multiple packet computations.

parameters

- [in] handle - hash/hmac handle
- [in] data - data to be calculated
- [in] data_len - data length (Unit: byte)

Attention

1. handle must be initialized with `rk_hash_init()`.
2. It can be called multiple times to feed the data that needs to be computed.
3. The data length data_len must be 64 bytes aligned if data is not the last group of data.

3.2.6.10 rk_hash_final

```
RK_RES rk_hash_final(rk_handle handle, uint8_t *hash);
```

function

After all the data is computed, the interface is called to get the final hash/hmac value and release the handle. If you need to interrupt the computation, you must also call this interface to end the hash computation.

parameters

- [in] handle- hash/hmac handle
- [out] hash - output of hash/hmac

Attention

1. handle must be initialized with `rk_hash_init()`.
2. The hash/hmac output buffer size must be greater than or equal to the hash length.

3.2.6.11 rk_cipher_init

```
RK_RES rk_cipher_init(rk_cipher_config *config, rk_handle *handle);
```

function

Initialize symmetric cipher algorithms. Support TDES/AES/SM4 algorithms type. Support ECB/CBC/CTR/CFB/OFB mode.

parameters

- [in] config - algorithm, pattern, key, iv, etc. See rk_cipher_config
- [out] handle - handle of cipher

Attention

1. After init succeeds, `rk_cipher_final()` must be called to destroy the related resources, regardless of whether `rk_cipher_crypt/crypt_virt()` was successfully executed.

3.2.6.12 rk_cipher_crypt

```
RK_RES rk_cipher_crypt(rk_handle handle, int in_fd, int out_fd, uint32_t len);
```

function

Receive dma_fd of data to perform encryption and decryption by using symmetric cipher algorithm.

parameters

- [in] handle - handle of cipher
- [in] in_fd - dma_fd of input data
- [out] out_fd - dma_fd of output data
- [in] len - input data length(Unit: byte)

Attention

1. handle must be initialized with `rk_cipher_init()`.
2. in_fd and out_fd can be the same dma_fd.
3. After the calculation, iv in `rk_cipher_config` will be updated. Repeat many times, can realize the segmented call.

3.2.6.13 rk_cipher_crypt_virt

```
RK_RES rk_cipher_crypt_virt(rk_handle handle, const uint8_t *in, uint8_t *out,
uint32_t len);
```

function

Receive virtual address of data to perform encryption and decryption by using symmetric cipher algorithm.

parameters

- [in] handle - handle of cipher
- [in] in - input data buffer
- [out] out - output data buffer
- [in] len - input data length (Unit: byte)

Attention

1. handle must be initialized with `rk_cipher_init()`.
2. input and output can be the same buffer.
3. After the calculation, iv in `rk_cipher_config` will be updated. Repeat many times, can realize the segmented call.

3.2.6.14 rk_cipher_final

```
RK_RES rk_cipher_final(rk_handle handle);
```

function

End the calculation process and release the handle.

parameters

- [in] handle - handle of cipher that must be initialized with `rk_cipher_init()`.

3.2.6.15 rk_get_random

```
RK_RES rk_get_random(uint8_t *data, uint32_t len)
```

function

Gets a random number of the specified length from HWRNG.

parameters

- [out] data - random data
- [in] len - the length of the random number to get (Unit: byte)

3.2.6.16 rk_write_oem_otp_key

```
RK_RES rk_write_oem_otp_key(enum RK_OEM_OTP_KEYID key_id, uint8_t *key,
                            uint32_t key_len);
```

function

Writes the key in plaintext to the specified OEM OTP region.

For details of OEM OTP features, see <Rockchip_Developer_Guide_OTP_CN>.

parameters

- [in] key_id - the key region index to be written
- [in] key - plaintext of key
- [in] key_len - plaintext key length (Unit: byte)

Attention

1. Key_id supports four keys `RK_OEM_OTP_KEY0 - 3` by default. For RV1126 / RV1109, key_id `RK_OEM_OTP_KEY_FW` is supported. `RK_OEM_OTP_KEY_FW` is the key used by BootROM to decrypt loader. `rk_oem_otp_key_cipher_virt` supports the encryption and decryption of service data using this key.
2. For `RK_OEM_OTP_KEY_FW`, key_len supports 16 byte. For other keys, key_len supports 16, 24, and 32 byte.

3.2.6.17 rk_oem_otp_key_is_written

```
RK_RES rk_oem_otp_key_is_written(enum RK_OEM_OTP_KEYID key_id, uint8_t
*is_written);
```

function

Check whether the key has been written to the specified OEM OTP area.
For details of OEM OTP features, see <Rockchip_Developer_Guide_OTP_CN>.

parameters

- [in] key_id - the key region index to be written
- [out] is_written - check whether the secret key has been written. 1 indicates that the secret key has been written. 0 indicates that the secret key has not been written.

return value

is_written values are meaningful only when the return value is `#define RK_CRYPTO_SUCCESS 0x00000000`.

The RK3588 platform also checks whether the key_id is locked. If the key_id is locked, error `#define RK_CRYPTO_ERR_ACCESS_DENIED 0xF0000005` is displayed.

Attention

1. key_id supports four keys `RK_OEM_OTP_KEY0 - 3` by default. For RV1126/RV1109, the key whose key_id is `RK_OEM_OTP_KEY_FW` is supported.

3.2.6.18 rk_set_oem_hr_otp_read_lock

```
RK_RES rk_set_oem_hr_otp_read_lock(enum RK_OEM_OTP_KEYID key_id);
```

function

Set the read lock flag for the specified OEM OTP area. After the read lock flag is set successfully, the OTP area is forbidden to write data, and the existing data in the OTP area is unreadable by CPU software. You can use the key through the `rk_oem_otp_key_cipher_virt` interface.
For details of OEM OTP features, see <Rockchip_Developer_Guide_OTP_CN>.

parameters

- [in] key_id - The key_id to be set supports `RK_OEM_OTP_KEY0 - 3`

3.2.6.19 rk_oem_otp_key_cipher

```
RK_RES rk_oem_otp_key_cipher(enum RK_OEM_OTP_KEYID key_id, rk_cipher_config
*config,
                             int32_t in_fd, int32_t out_fd, uint32_t len);
```

function

Select the key in the OEM OTP area and perform single cipher calculation in dma_fd mode.

parameters

- [in] key_id - the otp key index to use
- [in] config - algorithm, pattern, key, iv, etc
- [in] in_fd - dma_fd of input which can be the same as output
- [out] out_fd - dma_fd of output
- [in] len - input data length (Unit: byte)

Attention

1. Key_id supports `RK_OEM_OTP_KEY0 - 3` by default. For RV1126 / RV1109, `RK_OEM_OTP_KEY_FW` is supported.
2. Support AES/SM4 algorithms type and ECB/CBC/CTS/CTR/CFB/OFB mode.
3. The key length can be 16, 24, or 32 bytes. On the RV1109/RV1126 platform, the key length can be 16 or 32 bytes. When the key_id is `RK_OEM_OTP_KEY_FW` , the key length can be 16 bytes.
4. in_fd and out_fd can be the same.

3.2.6.20 rk_oem_otp_key_cipher_virt

```
RK_RES rk_oem_otp_key_cipher_virt(enum RK_OEM_OTP_KEYID key_id, rk_cipher_config
*config,
                                  uint8_t *src, uint8_t *dst, uint32_t len);
```

function

Select the key in the OEM OTP area, and perform single cipher calculation in virtual addr mode.

parameters

- [in] key_id -the otp key index to use
- [in] config - algorithm, pattern, key, iv, etc
- [in] src - input data buffer
- [out] dst - output data buffer
- [in] len - input data length (Unit: byte)

Attention

1. Key_id supports `RK_OEM_OTP_KEY0 - 3` by default. For RV1126 / RV1109, `RK_OEM_OTP_KEY_FW` is supported.
2. Support AES/SM4 algorithms type and ECB/CBC/CTS/CTR/CFB/OFB mode.
3. The key length can be 16, 24, or 32 bytes. On the RV1109/RV1126 platform, the key length can be 16 or 32 bytes. When the key_id is `RK_OEM_OTP_KEY_FW` , the key length can be 16 bytes.
4. in_fd and out_fd can be the same.
5. The default maximum length of the input and output buffers len is 1MB. For RV1126/RV1109, len is 500KB.

3.2.6.21 rk_ae_init

function

AEAD algorithms initialization. AES/SM4 is supported. Currently, only GCM is supported.

parameters

- [in] config - algorithm, mode, key, iv, aad length, tag length, etc. See rk_ae_config .
- [out] handle - handle of AEAD

Attention

After init succeeds, you must call `rk_ae_final()` to destroy resources regardless of whether subsequent execution succeeds.

3.2.6.22 rk_ae_set_aad

```
RK_RES rk_ae_set_aad(rk_handle handle, int aad_fd);
```

function

Receiving dma_fd of data and sets aad parameters.

parameters

- [in] handle - handle of cipher
- [in] aad_fd - dma_fd of aad data

Attention

1. handle must be initialized with `rk_ae_init()` .
2. Currently, `rk_ae_set_aad` can only be called once after `rk_ae_init` . Multiple AAD data updates are not supported. Multiple updates cause the contents of AAD to be overwritten.

3.2.6.23 rk_ae_set_aad_virt

```
RK_RES rk_ae_set_aad_virt(rk_handle handle, uint8_t *aad_virt);
```

function

Receiving virtual address to sets AAD data.

parameters

- [in] handle - handle of cipher
- [in] in - buffer of aad data

Attention

1. handle must be initialized with `rk_ae_init()`.
2. Currently, `rk_ae_set_aad_virt` can only be called once after `rk_ae_init`. Multiple AAD data updates are not supported. Multiple updates cause the contents of AAD to be overwritten.

3.2.6.24 rk_ae_crypt

```
RK_RES rk_ae_crypt(rk_handle handle, int in_fd, int out_fd, uint32_t len, uint8_t
*tag);
```

function

AEAD algorithm is used to perform encryption and decryption and tag calculation/verification for receiving dma_fd of data.

parameters

- [in] handle - handle of ae
- [in] in_fd - dma_fd of input data
- [out] out_fd - dma_fd of output data
- [in] len - input data length(Unit: byte)
- [in/out] tag - input for encryption, output for decryption

Attention

1. handle must be initialized with `rk_ae_init()`.
2. in_fd and out_fd can be the same dma_fd.
3. Multiple call segments are not supported.

3.2.6.25 rk_ae_crypt_virt

```
RK_RES rk_ae_crypt_virt(rk_handle handle, const uint8_t *in, uint8_t *out,
uint32_t len, uint8_t *tag);
```

function

AEAD algorithm is used to encrypt and decrypt virtual address data.

parameters

- [in] handle - handle of ae

- [in] in - input buffer
- [out] out - output buffer
- [in] len - input data length(Unit: byte)
- [in/out] tag - input for encryption, output for decryption

Attention

1. handle must be initialized with `rk_ae_init()`.
2. input and output can be the same buffer..
3. Multiple call segments are not supported.

3.2.6.26 rk_ae_final

```
RK_RES rk_ae_final(rk_handle handle);
```

function

End the calculation process and release the handle.

parameters

- [in] handle - handle of AE that must be initialized with `rk_ae_init()`.

3.2.6.27 rk_rsa_pub_encrypt

```
RK_RES rk_rsa_pub_encrypt(const rk_rsa_pub_key_pack *pub, enum
RK_RSA_CRYPT_PADDING padding, const uint8_t *in, uint32_t in_len, uint8_t *out,
uint32_t *out_len);
```

function

The public key is used to encrypt data, and encryption padding is supported.

parameters

- [in] pub - rsa public key information. See rk_rsa_pub_key_pack .
- [in] padding - encrypt padding type. See RK_RSA_CRYPT_PADDING .
- [in] in - input data
- [in] in_len - input data length(Unit: byte). If the input data is too long, the error code `RK_CRYPTO_ERR_PADDING_OVERFLOW` is returned.
- [out] out - ciphertext encrypted by the public key
- [out] out_len - ciphertext length

Attention

1. The public key must be set correctly to specify whether the key is in plaintext or the ciphertext encrypted by OTP KEY. This step will affect key parsing.
2. In-situ encryption and decryption are not recommended.

3.2.6.28 rk_rsa_priv_decrypt

```
RK_RES rk_rsa_priv_decrypt(const rk_rsa_priv_key_pack *priv, enum
RK_RSA_CRYPT_PADDING padding, const uint8_t *in, uint32_t in_len, uint8_t *out,
uint32_t *out_len);
```

function

The private key is used to decrypt the data, and padding parsing is supported.

parameters

- [in] priv - rsa private key information. See rk_rsa_priv_key_pack
- [in] padding - the padding type must be the same as the encryption padding. See RK_RSA_CRYPT_PADDING
- [in] in - input data
- [in] in_len - input data length(Unit: byte)
- [out] out - plaintext decrypted by private key
- [out] out_len - plaintext length

Attention

1. The private key must be set correctly to specify whether the key is in plain text or the ciphertext encrypted by OTP KEY. This step will affect key parsing.
2. In-situ encryption and decryption are not recommended.

3.2.6.29 rk_rsa_priv_encrypt

```
RK_RES rk_rsa_priv_encrypt(const rk_rsa_priv_key_pack *priv, enum
RK_RSA_CRYPT_PADDING padding, const uint8_t *in, uint32_t in_len, uint8_t *out,
uint32_t *out_len);
```

function

The private key is used to encrypt data, and encryption padding is supported.

parameters

- [in] priv - rsa private key information. See rk_rsa_priv_key_pack
- [in] padding - encrypt padding type. See RK_RSA_CRYPT_PADDING
- [in] in - input data
- [in] in_len - input data length(Unit: byte). If the input data is too long, the error code `RK_CRYPTO_ERR_PADDING_OVERFLOW` is returned.
- [out] out - ciphertext encrypted by the private key
- [out] out_len - ciphertext length

Attention

1. The private key must be set correctly to specify whether the key is in plain text or the ciphertext encrypted by OTP KEY. This step will affect key parsing.
2. In-situ encryption and decryption are not recommended.

3.2.6.30 rk_rsa_pub_decrypt

```
RK_RES rk_rsa_pub_decrypt(const rk_rsa_pub_key_pack *pub, enum
RK_RSA_CRYPT_PADDING padding, const uint8_t *in, uint32_t in_len, uint8_t *out,
uint32_t *out_len);
```

function

The public key is used to decrypt the data, and padding parsing is supported.

parameters

- [in] pub - rsa public key information. See rk_rsa_pub_key_pack .
- [in] padding - the padding type must be the same as the encryption padding. see RK_RSA_CRYPT_PADDING
- [in] in - input data
- [in] in_len - input data length(Unit: byte)
- [out] out - plaintext decrypted by public key
- [out] out_len - plaintext length

Attention

1. The public key must be set correctly to specify whether the key is in plain text or the ciphertext encrypted by OTP KEY. This step will affect key parsing.
2. In-situ encryption and decryption are not recommended.

3.2.6.31 rk_rsa_sign

```
RK_RES rk_rsa_sign(const rk_rsa_priv_key_pack *priv, enum RK_RSA_SIGN_PADDING
padding, const uint8_t *in, uint32_t in_len, const uint8_t *hash, uint8_t *out,
uint32_t *out_len);
```

function

Use the private key pair to calculate the hash of input data or use the external hash value for signature.

parameters

- [in] priv - rsa private key information. See rk_rsa_priv_key_pack
- [in] padding - signature padding type. See RK_RSA_SIGN_PADDING
- [in] in - input data will calculating hash first and signed by private key (optional)
- [in] in_len - input data length (Unit: byte)
- [in] hash - hash value(optional), the hash length is determined according to the padding algorithm, and the hash value can be signed directly by using the private key
- [out] out - signature data signed using the private key
- [out] out_len - signature data length

Attention

1. The private key must be set correctly to specify whether the key is in plain text or the ciphertext encrypted by OTP KEY. This step will affect key parsing.
2. When `in != NULL && hash == NULL`, the interface calculates the hash value of input data (the hash algorithm is determined by the padding format) and then executes the padding signature on the hash value.
3. The interface padding the incoming hash value directly if `hash != NULL`.

3.2.6.32 rk_rsa_verify

```
RK_RES rk_rsa_verify(const rk_rsa_pub_key_pack *pub, enum RK_RSA_SIGN_PADDING
padding, const uint8_t *in, uint32_t in_len, const uint8_t *hash, uint8_t *sign,
uint32_t sign_len);
```

function

Verify the signature using the public key.

parameters

- [in] pub - rsa public key information. See rk_rsa_pub_key_pack .
- [in] padding - the padding type must be the same as the signing padding. See RK_RSA_SIGN_PADDING
- [in] in - input data will calculating hash first and signed by private key (optional)
- [in] in_len - input data length (Unit: byte)
- [in] hash - hash value(optional)
- [in] sign - signature data
- [in] sign_len - signature data length (Unit: byte)

Attention

1. The public key must be set correctly to specify whether the key is in plain text or the ciphertext encrypted by OTP KEY. This step will affect key parsing.
2. When `in != NULL && hash == NULL`, the interface calculates the hash value of in (the hash algorithm is determined by the padding format) and then executes the padding verify the hash value.
3. The interface padding the incoming hash value directly if `hash != NULL`.

3.2.7 debug log

librkcrypto logs are currently divided into the following levels.

```
enum RKCRYPTO_TRACE_LEVEL {
    TRACE_TOP     = 0,
    TRACE_ERROR   = 1,
    TRACE_INFO    = 2,
    TRACE_DEBUG   = 3,
    TRACE_VERBOSE = 4,
    TRACE_BUTT,
};
```

The default log level is `TRACE_INFO`. You can change the log level by setting the environment variable (which must be set before the librkcrypto library is loaded). It can also be set through the `rkcrypto_set_trace_level` interface before `rk_crypto_init`.

Android：

```
setprop vendor.rkcrypto.trace.level 1/2/3/4
```

Linux：

```
export rkcrypto_trace_level=1/2/3/4
```

4. Hardware Crypto Performance

4.1 hardware crypto performance under uboot

4.1.1 crypto v1 performance

Test environment (uboot RK3399):

clock Rate: CRYPTO_CORE = 200M, The highest frequency varies slightly from chip to chip.

CIPHER/HASH Performance：

ALGO	Actual (MBps)	Theoretical (MBps)
DES	-	<=94
TDES	-	<=31
AES-128	-	<=290
AES-192	-	<=246
AES-256	-	<213
MD5	125	<196
SHA1	125	<158
SHA256	125	-

RSA Performance:

RSA key length (nbits)	pub enc/priv dec (ms)
2048	8 / 632

4.1.2 crypto v2 performance

Test environment (uboot RV1126) :

Clock Rate: CRYPTO_CORE = 200M，CRYPTO_PKA=300M, DDR=786M

Hash/HMAC: A total of 128M data is tested, and 4M data is calculated each time.

DES/3DES/AES/SM4： A total of 128M data is tested, 4M plaintext and 4M aad data are calculated each time.

ALGO	MODE	Actual (MBps)	Theoretical (MBps)
HASH/HMAC	MD5	183	196
	SHA1	148	158
	SHA256/224	183	196
	SHA512/384/512_224/512_256	288	316
	SM3	183	-
DES	ECB	289	352
	CBC/CFB/OFB	79	88
3DES	ECB	107	116
	CBC/CFB/OFB	27	29
AES (128 | 192 | 256)	ECB/CTR/XTS	447 | 442 | 436	1066 | 914 | 800
	CBC/CFB/OFB/CTS	234 | 204 | 180	266 | 228 | 200
	CMAC/CBC_MAC	245 | 212 | 186	266 | 228 | 200
	CCM(data+aad)	180 | 162 | 146	-
	GCM(data+aad)	196 | 184 | 174	-
SM4	ECB/CTR/XTS	320	-
	CBC/CFB/OFB/CTS	87	-
	CMAC/CBC_MAC	89	-
	CCM(data+aad)	156	-
	GCM(data+aad)	114	-

RSA test： Generate rsa keys, including n, e, and d, and perform public key encryption and private key decryption tests

Public key encrypt: ciphertext = d ^e % n

Private key decrypt: plaintext = d^d % n

5. References

6. Appendix

6.1 Professional Term