

Rockchip Crypto/HWRNG 开发指南

文件标识: RK-KF-YF-852

发布版本: V1.3.0

日期: 2024-04-24

文件密级: ☐绝密 ☐秘密 ☐内部资料 ☒公开

免责声明

本文档按“现状”提供，瑞芯微电子股份有限公司（“本公司”，下同）不对本文档的任何陈述、信息和内容的准确性、可靠性、完整性、适销性、特定目的性和非侵权性提供任何明示或暗示的声明或保证。本文档仅作为使用指导的参考。

由于产品版本升级或其他原因，本文档将可能在未经任何通知的情况下，不定期进行更新或修改。

商标声明

“Rockchip”、“瑞芯微”、“瑞芯”均为本公司的注册商标，归本公司所有。

本文档可能提及的其他所有注册商标或商标，由其各自拥有者所有。

版权所有 © 2024 瑞芯微电子股份有限公司

超越合理使用范畴，非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

地址: 福建省福州市铜盘路软件园A区18号

网址: www.rock-chips.com

客户服务电话: +86-4007-700-590

客户服务传真: +86-591-83951833

客户服务邮箱: fae@rock-chips.com

前言

概述

本文档主要介绍 Rockchip Crypto 和 HWRNG(TRNG) 的开发，包括驱动开发与上层应用开发。

产品版本

芯片名称	内核版本
RK 系列芯片	Linux 4.19
RK 系列芯片	Linux 5.10

读者对象

本文档（本指南）主要适用于以下工程师：

技术支持工程师

软件开发工程师

修订记录

版本号	作者	修改日期	修改说明
V1.0.0	林金寒/张志杰/王小滨	2022-01-25	初始版本
V1.1.0	张志杰	2022-02-28	增加 user space 调用 hwrng 的说明以及其他补充说明
V1.2.0	林金寒/张志杰	2022-09-14	1. rk_crypto_mem_alloc增加dma-heap分配支持 2. 增加cipher aead模式支持 3. 增加rsa算法支持 4. 增加crypto v3和trng v1说明 5. 增加日志打印级别说明 6. librkcrypto添加对kernel版本的依赖说明
V1.2.1	林金寒	2022-10-09	修复描述笔误
V1.2.2	林金寒	2023-03-15	CRYPTO配置说明补充
V1.2.3	林金寒	2024-01-12	补充CRYPTO V1/V2/V3/V4的说明
V1.3.0	林金寒	2024-04-24	补充硬件CRYPTO性能影响因素的说明

目录

Rockchip Crypto/HWRNG 开发指南

- 概述
 - crypto v1
 - crypto v2
 - crypto v3
 - crypto v4
 - 各平台版本情况
- 驱动开发
 - 驱动代码说明
 - hwrng
 - crypto
 - 启用硬件 hwrng
 - Menuconfig 配置
 - 板级 dts 文件配置
 - 新增芯片 dtsi 文件配置
 - 确认 hwrng 已启用的方法
 - 启用硬件 crypto

- Menuconfig 配置
- 板级 dts 文件配置
- 新增芯片平台支持
- 确认硬件 crypto 已启用的方法

应用层开发

- user space 调用硬件 hwrng

- 读取 kernel 驱动节点

- 调用 librcrypto API

- user space 调用硬件 crypto

- 适用范围

- 版本依赖

- V1.2.0

- 注意事项

- 数据结构

- rk_crypto_mem

- rk_cipher_config

- rk_ae_config

- rk_hash_config

- rk_rsa_pub_key

- rk_rsa_pub_key_pack

- rk_rsa_priv_key

- rk_rsa_priv_key_pack

- 常量

- RK_CRYPTO_ALGO

- RK_CIPHER_MODE

- RK_OEM_OTP_KEYID

- RK_CRYPTO_OPERATION

- RK_RSA_KEY_TYPE

- RK_RSA_CRYPT_PADDING

- RK_RSA_SIGN_PADDING

- 其他常量

- API

- 数据类型

- 返回值

- rk_crypto_mem_alloc

- rk_crypto_mem_free

- rk_crypto_init

- rk_crypto_deinit

- rk_hash_init

- rk_hash_update

- rk_hash_update_virt

- rk_hash_final

- rk_cipher_init

- rk_cipher_crypt

- rk_cipher_crypt_virt

- rk_cipher_final

- rk_get_random

- 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_ae_init

- rk_ae_set_aad

- rk_ae_set_aad_virt

- rk_ae_crypt

- rk_ae_crypt_virt

- rk_ae_final

- rk_rsa_pub_encrypt
- rk_rsa_priv_decrypt
- rk_rsa_priv_encrypt
- rk_rsa_pub_decrypt
- rk_rsa_sign
- rk_rsa_verify
- debug日志
- 硬件 crypto 性能数据
 - 影响性能的因素
 - uboot 层硬件 crypto 性能数据
 - crypto v1 性能数据
 - crypto v2 / v3 /v4 性能数据
- References
- 附录
 - 术语

概述

关于CRYPTO：当前 RK 平台上 CRYPTO IP 有四个版本，包括CRYPTO V1/V2/V3/V4。其中V1是比较早期的IP，支持的算法和功能比较少。因此重新设计了CRYPTO V2的IP核，在V2的基础上又进行了一系列的功能强化和更新，V2/V3/V4在基础代功能上完全兼容，V3在V2基础上添加了多线程切换的支持，V4在V3的基础上实现CRYPTO IP核的复用，降低IP的芯片面积占用。

关于TRNG: 之前大部分芯片平台的硬件随机数模块都是存在于硬件 CRYPTO IP 之中 (包括CRYPTO V1 和部分CRYPTO V2), 从 RK356x 开始, HWRNG (TRNG) 是独立的硬件模块。

crypto v1

算法	描述
DES/TDES	支持 ECB/CBC 两种模式，其中 TDES 支持 EEE 和 EDE 两种密钥模式
AES	支持 ECB/CBC/CTR/XTS 模式，支持 128/192/256 bit 三种密钥长度
HASH	支持 SHA1/SHA256/MD5。
RSA	支持 512/1024/2048 三种密钥长度。（RK3126、RK3128、RK3288 和 RK3368 不支持）
TRNG	支持 256bit 硬件随机数

crypto v2

算法	描述
DES/TDES	支持 ECB/CBC/OFB/CFB 四种模式，其中 TDES 只支持 EDE 密钥模式。
AES	支持 ECB/CBC/OFB/CFB/CTR/CTS/XTS/CCM/GCM/CBC-MAC/CMAC。
SM4	支持 ECB/CBC/OFB/CFB/CTR/CTS/XTS/CCM/GCM/CBC-MAC/CMAC。（可选）
HASH	支持 MD5/SHA1/SHA224/SOA256/SOA384/SOA512/SM3/SOA512-224/SOA512-256 带硬件填充。（SM3是可选的）
MAC	支持 MD5/SOA1/SOA256/SOA512/SM3 带硬件填充

算法	支持 MD5/SHA1/SHA256/SHA512/SM3 市硬件单元。
RSA/ECC	支持最大 4096bit 的常用大数运算操作，通过软件封装该操作可实现 RSA/ECC 算法。
TRNG	支持 256bit 硬件随机数

crypto v3

在crypto v2算法基础上，增加多线程支持。从rv1106开始的crypto v3平台已经可以做到自动识别支持的算法，因此compatible上统一使用"rockchip,crypto-v3"作为标识符。

crypto v4

在crypto v3算法基础上，优化了IP的芯片面积，使用同一个CRPTO CORE硬件，例化出NSCRYPTO/SCRYPTO/KEYLAD三个硬件IP。在IP的驱动使用上和crypto v3可以正常兼容复用代码。此compatible上统一使用"rockchip,crypto-v4"作为标识符。

各平台版本情况

各个芯片平台的 crypto IP 版本如下：

采用 crypto v1 的平台有：

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

采用 crypto v2 的平台有：

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

采用 crypto v3 的平台有：

RV1106

采用 crypto v4 的平台有：

RK3528、RK3562

驱动开发

驱动代码说明

hwrng

由于 hwrng 驱动比较简单，因此 crypto v1/v2, trngv1, rkrng四种平台都集中到同一个.c 文件中。

驱动中不区分具体的芯片型号，只按照 "rockchip,cryptov1-rng" 和 "rockchip,cryptov2-rng", "rockchip,trngv1", "rockchip,rkrng" 四种 compatible 进行划分。目前 "rockchip,trngv1", "rockchip,rkrng" 为独立的 HWRNG 模块，其他两种 HWRNG 均内置在 CYRPTO 模块中。

驱动代码： drivers/char/hw_random/rockchip-rng.c

crypto

当前驱动实现的算法如下：

crypto v1:

- **AES:** ECB/CBC

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

crypto v2/v3/v4: (驱动已经实现的算法列表, 有些算法在某些平台上支持, 请对照算法支持表)

- **AES:** ECB/CBC/OFB/CFB/CTR/GCM
- **DES/TDES:** ECB/CBC/CFB/OFB
- **SM4:** ECB/CBC/OFB/CFB/CTR/GCM
- **HASH:** SHA1/SHA256/SHA384/SHA512/MD5/SM3
- **HMAC:** HMAC_SHA1/HMAC_SHA256/HMAC_SHA512/HMAC_MD5/HMAC_SM3
- **RSA:** 最大 4096bit

crypto v2/v3/v4 硬件完整版 (以下删除线部份模式驱动尚未实现) :

- **AES(128/192/256):** ECB/CBC/OFB/CFB/CTR/~~XTS/CTS/ECM~~/GCM/~~ECB-MAC/CMAC~~
- **SM4:** ECB/CBC/OFB/CFB/CTR/~~XTS/CTS/ECM~~/GCM/~~ECB-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:** 4096bit PKA 大数运算支持

crypto v2/v3/v4 硬件差异表

芯片平台	AES	DES/TDES	SM3/SM4	HASH	HMAC	RSA	多线程
RK3326/PX30/RK3308	√	√	×	√	√	√	×
RK1808	AES-128	×	×	SHA-1/SHA-224/SHA-256/MD5	√	√	×
RV1126/RV1109	AES-128/AES-256	√	√	√	√	√	×
RK2206	√	√	×	√	√	√	×
RK3568/RK3588	√	√	√	√	√	√	×
RV1106	√	√	×	SHA-1/SHA-224/SHA-256/MD5	√	√	√
RK3562	√	√	×	SHA-1/SHA-224/SHA-256/MD5	√	√	√
RK3528	√	√	√	√	√	√	√

注:

1. RK1808 : AES 仅支持 128bit, 对于 kernel 驱动来说可以认为不支持 AES。
2. RV1126/RV1109: 由于不支持 AES-192, 因此 AES-192 部分只能通过软算法实现, 但是软算法不能支持硬算法的所有模式。因此建议不要去改动代码里已配置好的算法列表。

驱动相关文件如下:

```
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/v4 Hardware related interface
implementation
|-- rk_crypto_v3.h          // crypto v3/v4 Structure and interface
declaration
|-- rk_crypto_v3_skcipher.c // crypto v3/v4 block cipher algorithm
implement
|-- rk_crypto_v3_ahash.c    // crypto v3/v4 hash algorithm implement
|-- rk_crypto_v3_reg.h      // crypto v3/v4 hardware register definition
`-- cryptodev_linux        // exporting the crypto interface to User
space

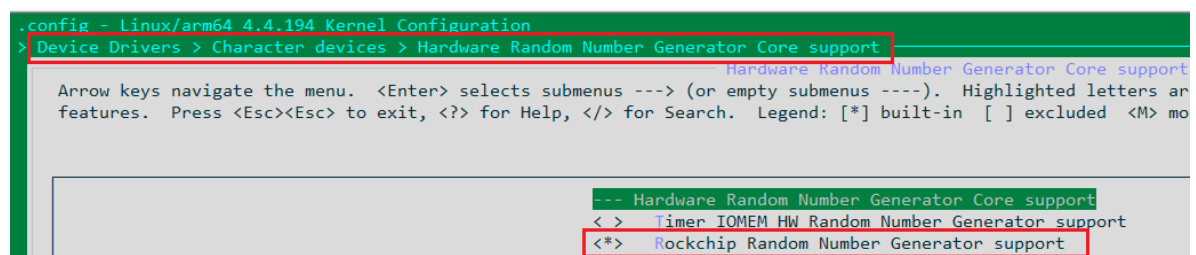
```

启用硬件 hwrng

Menuconfig 配置

hwrng驱动会默认编译进内核，由dts文件决定是否使能。

配置如下列图所示（红色标记表示配置路径和需要配置的选项）：



或在 config 文件（rockchip_defconfig 中已默认配置好）中添加如下语句：

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

板级 dts 文件配置

当前大部分芯片 dtsi 都已配置好 hwrng 节点，只需在板级 dts 中将 rng 模块使能即可，如下所示：

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

新增芯片 dtsi 文件配置

当前大部分芯片平台均已配置好 rng 节点，如果 dtsi 未配置好 hwrng 节点，可以参考以下方式进行配置。

注意：

1. rng 基地址需要根据芯片 TRM 进行修改，rng 基地址即 CRYPTO 基地址
2. clocks 的宏不同平台可能略有不同，如果 dts 出现报错，可以去 include/dt-bindings/clock 目录下，grep -rn CRYPTO 查找对应的 clock 宏名称，如下所示：

```
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:

实际 TRNG 不需要依赖全部的 clock，只需依赖 hclk_crypto 一个即可

```
rng: rng@ff500400 {
    compatible = "rockchip,cryptov2-rng";
    reg = <0xff500400 0x80>; # 需要加上0x400，如果rng在crypto内部
    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:

目前RK3588和RV1106使用的是trng v1的随机数模块，该模块与crypto v2中拆分出的trng模块设计完全不同，提升了随机性。

```
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";
};
```

确认 hwrng 已启用的方法

1. 执行 `cat /sys/devices/virtual/misc/hw_random/rng_current` 可以看到信息为 rockchip，确定当前调用的是硬件驱动
2. linux: 执行 `cat /dev/hwrng | od -x | head -n 1` 可以获取到一行随机数，每次执行，随机数的内容都不相同
3. Android: 执行 `cat /dev/hw_random | od -x | head -n 1` 可以获取到一行随机数，每次执行，随机数的内容都不相同

启用硬件 crypto

当前驱动代码 crypto v1 支持 rk3328, crypto v2 支持 px30/rv1126/rk3568/rk3588, crypto v3支持 rv1106。对于以上平台，只需开启 config 和 dts node 即可启用硬件 crypto。

从rv1106开始，支持的feature均会在寄存器信息中体现，crypto v3可以自动适配后续新增芯片的新增功能。

Menuconfig 配置

在 menuconfig 配置中使能 Rockchip 加解密驱动支持，在 dts 中会自动根据芯片平台 compatible id 进行自动适配 v1/v2/v3。（要先确保CONFIG_CRYPTO_HW开启，才能看到硬件crypto的相关配置项）

```
.config - Linux/arm64 5.10.66 Kernel Configuration
> Cryptographic API > Hardware crypto devices
    Hardware crypto devices
    Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----).
    Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features.
    Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ] excluded
    <M> module < > module capable

    --- Hardware crypto devices
    < > Support for Microchip / Atmel ECC hw accelerator
    < > Support for Microchip / Atmel SHA accelerator and RNG
    [ ] Support for AMD Secure Processor
    < > Support for Cavium CNN55XX driver
    < > Cavium ZIP driver
    [*] Rockchip's Cryptographic Engine driver
    [*] Export rockchip crypto device for user space
    < > Inside Secure's SafeXcel cryptographic engine driver
```

或在 config 文件中添加如下语句，其中的 CONFIG_CRYPTO_DEV_ROCKCHIP_V3 只是示例，要根据实际的芯片配置，建议使用menuconfig的形式进行修改，会自动选择平台：

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

板级 dts 文件配置

确认 crypto 的 dts 节点配置正常后，直接在板级 dts 文件中开启 crypto 模块即可，如下所示：

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

新增芯片平台支持

如果芯片 dtsi 中没有配置 crypto 的 dts 节点，则需要按照以下步骤添加支持。

1. 确定芯片 crypto IP 的版本 v1/ v2/v3，从RV1106开始的V3版本，compatible基本已确定为"rockchip,crypto-v3"，算法的裁剪和feature均由软件自行适配，只需配置好dts即可。
2. `drivers/crypto/rockchip/rk_crypto_core.c` 中添加对应的 `algs_name`，`soc_data`，`compatible` 等信息。

```
/* 增加芯片支持的算法信息，px30属于crypto v2，支持的算法参见crypto_v2_algs */
/* 特别注意：crypto_v2_algs为crypto v2支持的所有算法。*/
/* 某些芯片在crypto v2上做了些裁剪，如rk1808不支持SHA512算法，因此需要对比TRM确认支持的算法 */
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",
};

/* 绑定px30_algs_name到px30_soc_data */
static const struct rk_crypto_soc_data px30_soc_data =
    RK_CRYPTO_V2_SOC_DATA_INIT(px30_algs_name, false);

/* 绑定px30_soc_data到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. 芯片 dtsi 增加 crypto 配置

注意:

1. 根据芯片 TRM 进行修改确定 CRYPTO 基地址
2. clocks 的宏不同平台可能略有不同, 如果 dts 出现报错, 可以去 `include/dt-bindings/clock` 目录下, `grep -rn CRYPTO` 查找对应的 clock 宏名称, 如下所示:

```
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: crypto-controller@ff8a0000 {                                /* 根据实际配置crypto
基地址*/
    compatible = "rockchip,rk3288-crypto";                        /* 修改芯片平台,
如"rk3399-crypto" */
    reg = <0x0 0xff8a0000 0x0 0x4000>;                             /* 根据实际配置crypto
基地址 */
    interrupts = <GIC_SPI 48 IRQ_TYPE_LEVEL_HIGH>;                /* 根据实际配置crypto
中断号 */
    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:

对于大部分 crypto v2 芯片, hwrng 的寄存器地址位于 crypto 中间, 因此配置 reg 时, 需要将 crypto 的地址空间拆分成两个部分, 第一部分为 CIPHER 使用的寄存器, 第二部分为 RSA 使用的寄存器。

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

```

crypto: crypto@ff500000 {                                /* 根据实际配置crypto
基地址 */
    compatible = "rockchip,rv1126-crypto";              /* 修改芯片平台 */
    reg = <0xff500000 0x400>, <0xff500480 0x3B80>;        /* 根据实际配置crypto
基地址 */
    interrupts = <GIC_SPI 3 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&cru CLK_CRYPTOCORE>, <&cru CLK_CRYPTOPKA>,
    <&cru ACLK_CRYPTOCORE>, <&cru HCLK_CRYPTOCORE>;
    clock-names = "aclk", "hclk", "sclk", "apb_pclk";
    power-domains = <&power RV1126_PD_CRYPTOCORE>;
    resets = <&cru SRST_CRYPTOCORE>;
    reset-names = "crypto-rst";
    status = "disabled";
};

```

4. 板级 dts 配置 crypto 开启

```

&crypto {
    status = "okay";
};

```

确认硬件 crypto 已启用的方法

通过命令 `cat /proc/crypto | grep rk` 可以查看系统注册的 RK 硬件 crypto 算法。（以 RV1126 为例）

```

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_edex-rk
driver      : cfb-des3_edex-rk
driver      : cbc-des3_edex-rk
driver      : ecb-des3_edex-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
```

通过命令cat /proc/rkcrypto(需要确保升级最新代码, 旧的代码不支持该功能), 可以查看rockchip crypto驱动的相关信息。包括crypto 版本, clock时钟频率, 当前可用的算法, 以及当前驱动运行的一些统计信息, 后续会不断进行完善补充。(CRYPTO Version中"CRYPTO V3.0.0.0 multi"表示当前平台支持crypto支持多线程)。

```
Rockchip Crypto Version: CRYPTO V2.0.0.0
```

```
use_soft_aes192 : false
```

```
clock info:
```

```
    ac1k      350000000
    hc1k      150000000
    sc1k      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

```

应用层开发

user space 调用硬件 hwrng

user space 有两种方式可以获取到硬件 hwrng 输出的随机数：

- 读取 kernel 驱动节点
- 调用 librcrypto 库中的接口

注意：

1. hwrng 硬件驱动注册成功后可以为 kernel random 驱动增加熵，hwrng 产生的随机数会输入到 random 驱动的熵池中。kernel 的 random 驱动是 CSPRNG (Cryptography Secure Pseudo Random Number Generator)，是符合密码学安全标准的。因此如果对随机数质量要求较高的话，可以读取 `/dev/random` 或者 `/dev/urandom` 节点获取随机数。

读取 kernel 驱动节点

若 kernel 已开启 rng，在 user space 可以通过读取节点方式获取到随机数。**Linux 平台读取的节点为 `/dev/hwrng`，Android 平台读取的节点为 `/dev/hw_random`。**参考代码如下：

```

#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_CRYPTOSUCCESS;
    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_CRYPTOERR_GENERIC;
    }
}

```

```

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

        close(hwrng_fd);

        return res;
    }
}

```

调用 librkcrypto API

参考 API 说明：[rk_get_random](#)。

user space 调用硬件 crypto

user space 使用 librkcrypto api 接口进行调用。本节是对 librkcrypto 的使用说明。

注意：使用前请确认 kernel 中硬件 crypto 是否已启用，启用方法与确认方法参考[启用硬件 crypto](#)和[确认硬件 crypto 已启用的方法](#)。

适用范围

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	√	√	√	

版本依赖

V1.2.0

V1.2.0版本librkcrypto库功能依赖kernel以下提交点，若kernel crypto驱动没更新到以下提交点，可能会导致部分功能不可用。

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
```

注意事项

- **对称算法的输入数据长度，要求与所选算法和模式的数据长度要求一致。** 比如 ECB/CBC 等要求 block 对齐，CTS/CTR 等则无数据长度对齐要求。API 中不做填充处理。
- **如果计算数据量较大，为了提高效率，建议选用通过 dma_fd 传递数据的算法接口。** 由于 crypto 只支持 4G 以内连续物理地址，因此 dma fd 分配的 buffer 必须是 4G 以内物理连续地址（CMA）。可以使用 librkcrypto 提供的 rk_crypto_mem 相关接口分配，也可以自行用 DRM 等内存分配接口分配得到 dma fd。
- **CMA 配置：** 由于 crypto 只支持 4G 以内的 CMA 地址访问，如果设备使用内存超过 4G，需要修改 dts 中 CMA 的配置，否则 rk_crypto_mem 虽然能分配成功，但是分配出的内存无法使用。以下以 rk3588-android.dtsi 平台为例。其中 0x10000000 为 CMA 的起始地址（256MB 处，尽量不要修改），0x00800000 为 CMA 的大小，可以根据实际需要进行修改。CMA 相关说明见文档 <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;
    };
```

- **使用以下接口前，需确保 TEE 功能可用，TEE 相关说明见 <Rockchip_Developer_Guide_TEE_SDK_CN>文档。**

```
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`：当设置的 `key_id` 为 `RK_OEM_OTP_KEY0/1/2` 时，设置成功后，会影响其他 OTP 区域的属性。例如部分 OTP 区域变为不可写，详见 <Rockchip_Developer_Guide_OTP_CN>文档。因此，建议优先使用 `RK_OEM_OTP_KEY3`。
- `rk_oem_otp_key_cipher_virt`：支持的 `len` 最大值受 TEE 的共享内存影响，如果使用本接口前已占用 TEE 共享内存，那么 `len` 的最大值可能比预期的小。

数据结构

rk_crypto_mem

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

- `vaddr` - memory 的虚拟地址
- `dma_fd` - memory 对应的 dma fd 句柄
- `size` - memory 区域的大小

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` - 算法类型，见[RK_CRYPTO_ALGO](#)，实际取值范围以 API 的描述为准，下同
- `mode` - 算法模式，见[RK_CIPHER_MODE](#)，支持 ECB/CBC/CTR/CFB/OFB/
- `operation` - 加解密模式见[RK_CRYPTO_OPERATION](#)
- `key` - 密钥明文，当使用 otp key 操作时无效
- `key_len` - key 的长度（单位：byte）
- `iv` - 初始向量，当 ECB 模式时无效，其他模式下，执行 `rk_cipher_crypt/crypt_virt` 会自动更新 iv，用于多次分段计算
- `reserved` - 预留

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 - 算法类型, 见[RK CRYPTO ALGO](#), 支持 AES/SM4
- mode - 算法模式, 见[RK CIPHER MODE](#), 支持 GCM/CCM
- operation - 加解密模式见[RK CRYPTO OPERATION](#)
- key - 密钥明文, 当使用 keyladder 操作时无效
- key_len - key 的长度 (单位: byte)
- iv - 初始向量
- iv_len - iv 的长度 (单位: byte)
- tag_len - tag 的长度 (单位: byte)
- aad_len - aad 的长度 (单位: byte)
- payload_len - payload 的长度 (单位: byte)
- reserved - 预留

rk_hash_config

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

- algo - 算法类型, 见[RK CRYPTO ALGO](#), 支持 HASH/HMAC 等多种算法
- key - hash-mac 密钥, 只有当 algo 为 HMAC 类型的算法才有效
- key_len - key 的长度 (单位: byte)

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 - 模长, 与 OpenSSL 相同, 大端模式
- e - 指数, 与 OpenSSL 相同, 大端模式
- n_len - 模长的长度
- e_len - 指数的长度

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 - 密钥类型，见[RK_RSA_KEY_TYPE](#)，支持明文密钥和OTP_KEY加密后的密文密钥，librkcrypto会将传入的密钥，用对应的otp key密钥解密之后再使用。
- key - 公钥内容，见[rk_rsa_pub_key](#)。

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 - 模长，与OpenSSL相同，大端模式
- e - 指数，与OpenSSL相同，大端模式
- d - 模反元素，即私钥。与OpenSSL相同，大端模式
- p - 可选
- q - 可选
- dp - 可选
- dq - 可选
- qp - 可选
- len - 各个元素的长度信息，此处不再赘述。

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 - 密钥类型，见[RK_RSA_KEY_TYPE](#)，支持明文密钥和OTP_KEY加密后的密文密钥，librkcrypto会将传入的密钥，用对应的otp key密钥解密之后再使用。
- key - 私钥内容，见[rk_rsa_priv_key](#)。

常量

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,
};

```

RK_CIPHER_MODE

```

/* crypto mode */
enum RK_CIPHER_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
};

```

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
};
```

RK_CRYPTO_OPERATION

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

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,
};
```

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*/
};
```

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*/
};
```

其他常量

```
/* 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_CRYPTOMAX_DATA_LEN (1 * 1024 * 1024)
```

API

数据类型

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

返回值

```
/* API return codes */
#define RK_CRYPTTO_SUCCESS                0x00000000
#define RK_CRYPTTO_ERR_GENERIC            0xF0000000
#define RK_CRYPTTO_ERR_PARAMETER          0xF0000001
#define RK_CRYPTTO_ERR_STATE              0xF0000002
#define RK_CRYPTTO_ERR_NOT_SUPPORTED      0xF0000003
#define RK_CRYPTTO_ERR_OUT_OF_MEMORY      0xF0000004
#define RK_CRYPTTO_ERR_ACCESS_DENIED      0xF0000005
#define RK_CRYPTTO_ERR_BUSY               0xF0000006
#define RK_CRYPTTO_ERR_TIMEOUT            0xF0000007
#define RK_CRYPTTO_ERR_UNINITED           0xF0000008
#define RK_CRYPTTO_ERR_KEY                0xF0000009
#define RK_CRYPTTO_ERR_VERIFY             0xF000000A
#define RK_CRYPTTO_ERR_PADDING            0xF000000B
#define RK_CRYPTTO_ERR_PADDING_OVERFLOW   0xF000000C
#define RK_CRYPTTO_ERR_MAC_INVALID        0xF000000D
```

rk_crypto_mem_alloc

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

功能

申请一块内存，返回 rk_crypto_mem，包含内存的虚拟地址和 dma_fd 等信息。

参数

- [in] size - 待申请内存的大小
- [out] memory - 返回的内存地址，见[rk_crypto_mem](#)

注意

1. 申请内存允许的最大值依赖于 kernel CMA buffer 大小以及使用情况。

rk_crypto_mem_free

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

功能

释放通过 rk_crypto_mem_alloc 申请的内存。

参数

- [in] memory - 内存地址，见[rk_crypto_mem](#)

rk_crypto_init

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

功能

crypto 初始化，例如打开设备节点等。

参数

- 无

rk_crypto_deinit

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

功能

释放 crypto 相关资源，例如关闭设备节点等。

参数

- 无

rk_hash_init

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

功能

初始化 hash 算法，支持 MD5/SHA1/SHA224/SHA256/SHA384/SHA512/SM3。

参数

- [in] config - hash/hmac 配置
- [out] handle - hash/hmac 句柄

注意

1. init 成功后，无论 rk_hash_update() 或者 rk_hash_update_virt 是否成功执行，都必须调用 rk_hash_final() 销毁相关资源。
2. 如果 init 返回 RK_CRYPT_ERR_BUSY，则说明当前平台不支持多线程，同时只能有一个 handle 在工作。需要等待前一个 handle 释放掉，才能 init 申请新的 handle。

rk_hash_update

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

功能

接收 dma_fd 数据作为输入，计算 hash/hmac 值，支持分组多次计算。

参数

- [in] handle- hash/hmac 句柄
- [in] data_fd - 待计算 hash/hmac 的一组数据的句柄
- [in] data_len - data 的长度（单位：byte）

注意

1. handle 必须经过 rk_hash_init() 初始化。
2. 可以分多次调用，多次喂入需要计算哈希的数据。
3. 若 data 不是最后一组数据，则数据长度 data_len 必须 64 字节对齐，最后一组数据无此限制。

rk_hash_update_virt

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

功能

接收虚拟地址数据作为输入，计算 hash 值，支持分组多次计算。

参数

- [in] handle - hash/hmac 句柄
- [in] data - 待计算 hash/hmac 的一组数据
- [in] data_len - data 的长度（单位：byte）

注意

1. handle 必须经过 `rk_hash_init()` 初始化。
2. 可以分多次调用，多次喂入需要计算哈希的数据。
3. 若 data 不是最后一组数据，则数据长度 data_len 必须 64 字节对齐，最后一组数据无此限制。

rk_hash_final

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

功能

在计算完所有的数据后，调用这个接口获取最终的 hash/hmac 值，并释放句柄。如果在计算过程中，需要中断计算，也必须调用该接口结束 hash 计算。

参数

- [in] handle- hash/hmac 句柄
- [out] hash - 输出的 hash/hmac 数据

注意

1. handle 必须经过 `rk_hash_init()` 初始化。
2. 存放哈希数据的内存 hash 大小必须大于等于哈希长度。

rk_cipher_init

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

功能

对称分组算法的初始化，支持 TDES/AES/SM4算法，支持 ECB/CBC/CTR/CFB/OFB模式。

参数

- [in] config - 算法、模式、密钥、iv 等，见[rk_cipher_config](#)
- [out] handle - cipher 的 handle

注意

1. init 成功后，无论 `rk_cipher_crypt/crypt_virt()` 是否成功执行，都必须调用 `rk_cipher_final()` 销毁相关资源。

rk_cipher_crypt

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

功能

接收 dma_fd 数据使用对称分组算法执行加解密。

参数

- [in] handle - cipher 的 handle

- [in] in_fd - 输入数据
- [out] out_fd - 输出计算结果
- [in] len - 输入数据的长度（单位：byte）

注意

1. handle 必须经过 rk_cipher_init() 初始化。
2. in_fd 可以和 out_fd 相同，即支持原地加解密。
3. 计算完成之后，rk_cipher_config 中的iv会被更新。重复多次调用，即可实现分段调用。

rk_cipher_crypt_virt

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

功能

接收虚拟地址数据使用对称分组算法执行加解密。

参数

- [in] handle - cipher 的 handle
- [in] in - 输入数据 buffer
- [out] out - 输出计算结果
- [in] len - 输入数据的长度（单位：byte）

注意

1. handle 必须经过 rk_cipher_init() 初始化。
2. in 和 out 可以为相同地址，即支持原地加解密。
3. 计算完成之后，rk_cipher_config 中的iv会被更新。重复多次调用，即可实现分段调用。

rk_cipher_final

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

功能

对称分组算法，结束计算，清除 handle。

参数

- [in] handle - cipher 的 handle，必须经过 rk_cipher_init() 初始化。

rk_get_random

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

功能

从 HWRNG 获取指定长度的随机数。

参数

- [out] data - 输出的随机数
- [in] len - 需要获取的随机数的长度（单位：byte）

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);
```

功能

把密钥明文写到指定的 OEM OTP 区域。

OEM OTP 的相关特性说明，见<Rockchip_Developer_Guide_OTP_CN>文档。

参数

- [in] key_id - 将要写的 key 区域索引
- [in] key - 密钥明文
- [in] key_len - 密钥明文长度（单位：byte）

注意

1. key_id 默认支持 RK_OEM_OTP_KEY0 - 3 共 4 个密钥，对于 RV1126/RV1109，额外支持 key_id 为 RK_OEM_OTP_KEY_FW 的密钥。RK_OEM_OTP_KEY_FW 为 BootROM 解密 loader 时用的密钥，rk_oem_otp_key_cipher_virt 接口支持用这个密钥去做业务数据加解密。
2. 对于 RK_OEM_OTP_KEY_FW，key_len 仅支持 16 字节，对于其他密钥，key_len 支持 16、24、32 字节。

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);
```

功能

判断密钥是否已经写入指定的 OEM OTP 区域。

OEM OTP 的相关特性说明，见<Rockchip_Developer_Guide_OTP_CN>文档。

参数

- [in] key_id - 将要写的 key 区域索引。
- [out] is_written - 判断是否已经写入密钥，1 表示已写入，0 表示未写入。

返回值

当返回值为 #define RK_CRYPTO_SUCCESS 0x00000000 时，is_written 值才有意义。

RK3588 平台还会判断 key_id 是否被 lock，若对应 key_id 被 lock 则会返回错误 #define RK_CRYPTO_ERR_ACCESS_DENIED 0xF0000005。

注意

1. key_id 默认支持 RK_OEM_OTP_KEY0 - 3 共 4 个密钥，对于 RV1126/RV1109，额外支持 key_id 为 RK_OEM_OTP_KEY_FW 的密钥。

rk_set_oem_hr_otp_read_lock

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

功能

设置指定 OEM OTP 区域的 read lock 标志，设置成功后，该区域禁止写数据，并且该区域已有的数据 CPU 软件不可读，可通过 rk_oem_otp_key_cipher_virt 接口使用密钥。

OEM OTP 的相关特性说明，见 Rockchip_Developer_Guide_OTP_CN 文档。

参数

- [in] key_id - 将要设置的 key_id，支持 RK_OEM_OTP_KEY0 - 3

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);
```

功能

选择 OEM OTP 区域的密钥，以 dma_fd 的方式，进行 cipher 单次计算。

参数

- [in] key_id - 将要使用的 otp key 索引
- [in] config - 算法、模式、密钥、iv 等
- [in] in_fd - 待计算数据，支持等同于 out_fd，即支持原地加解密
- [out] out_fd - 输出计算结果
- [in] len - 输入和输出数据的长度（单位：byte）

注意

1. key_id 默认支持 RK_OEM_OTP_KEY0 - 3，对于 RV1126/RV1109，额外支持 RK_OEM_OTP_KEY_FW。
2. 算法模式支持 AES/SM4-ECB/CBC/CTS/CTR/CFB/OFB。
3. 密钥长度支持 16、24、32 字节，若是 RV1109/RV1126 平台，密钥长度仅支持 16、32，当 key_id 为 RK_OEM_OTP_KEY_FW 时密钥长度仅支持 16。
4. in_fd 与 out_fd 可以相同，即支持原地加解密。

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);
```

功能

选择 OEM OTP 区域的密钥，执行 cipher 单次计算。

参数

- [in] key_id - 将要使用的 otp key 索引
- [in] config - 算法、模式、密钥、iv 等
- [in] src - 待计算数据的 buffer
- [out] dst - 计算结果的 buffer
- [in] len - 输入和输出数据 buffer 的长度（单位：byte）

注意

1. key_id 默认支持 RK_OEM_OTP_KEY0 - 3，对于 RV1126/RV1109，额外支持 RK_OEM_OTP_KEY_FW。
2. 算法模式支持 AES/SM4-ECB/CBC/CTS/CTR/CFB/OFB。
3. 密钥长度支持 16、24、32 字节，若是 RV1109/RV1126 平台，密钥长度仅支持 16、32，当 key_id 为 RK_OEM_OTP_KEY_FW 时密钥长度仅支持 16。
4. src 与 dst 可以为相同地址，即支持原地加解密。

5. 输入和输出 buffer 的长度 len 默认最大支持 1MB，对于 RV1126/RV1109，len 最大约为 500KB。

rk_ae_init

功能

AEAD算法的初始化，支持 AES/SM4，当前仅支持 GCM模式。

参数

- [in] config - 算法、模式、密钥、iv、aad长度、tag长度等，见[rk_ae_config](#)
- [out] handle - AEAD的 handle

注意

init 成功后，无论后续 是否成功执行，都必须调用 `rk_ae_final()` 销毁相关资源。

rk_ae_set_aad

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

功能

接收 dma_fd 数据设置aad参数。

参数

- [in] handle - cipher 的 handle
- [in] aad_fd - AAD输入数据

注意

1. handle 必须经过 `rk_ae_init()` 初始化。
2. 目前 `rk_ae_set_aad` 在一次 `rk_ae_init` 后只能调用一次，不支持AAD分段多次更新。多次更新会导致AAD的内容不断被覆盖，以最后一次调用的aad_fd为准。

rk_ae_set_aad_virt

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

功能

接收虚拟地址数据设置aad数据。

参数

- [in] handle - cipher 的 handle
- [in] in - aad输入数据 buffer

注意

1. handle 必须经过 `rk_ae_init()` 初始化。
2. 目前`rk_ae_set_virt`在一次init后只能调用一次，不支持add分段多次更新。多次更新会导致add的内容不断被覆盖，以最后一次调用的buffer为准。

rk_ae_crypt

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

功能

接收 dma_fd 数据使用AEAD算法执行加解密和TAG计算/验证。

参数

- [in] handle - ae的 handle
- [in] in_fd - 输入数据
- [out] out_fd - 输出计算结果
- [in] len - 输入数据的长度（单位：byte）
- [in/out] tag - 当加密时，tag为out。当解密时，tag为in。

注意

1. handle 必须经过 rk_ae_init() 初始化。
2. in_fd 可以和 out_fd 相同，即支持原地加解密。
3. 不支持分段多次调用。

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);
```

功能

接收虚拟地址数据使用AEAD算法执行加解密。

参数

- [in] handle - ae 的 handle
- [in] in - 输入数据 buffer
- [out] out - 输出计算结果
- [in] len - 输入数据的长度（单位：byte）
- [in/out] tag - 当加密时，tag为out。当解密时，tag为in。

1. 注意

1. handle 必须经过 rk_ae_init() 初始化。
2. in可以和 out为相同地址，即支持原地加解密。
3. 不支持分段多次调用。

rk_ae_final

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

功能

AEAD算法，结束计算，清除 handle。

参数

- [in] handle - ae 的 handle，必须经过 rk_ae_init() 初始化。

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);
```

功能

使用公钥对数据进行加密，支持加密填充。

参数

- [in] pub - rsa的公钥信息，见[rk_rsa_pub_key_pack](#)
- [in] padding - 加密填充格式，见[RK_RSA_CRYPT_PADDING](#)
- [in] in - 输入数据
- [in] in_len - 输入数据的长度（单位：byte），输入数据过长将会返回错误码
`RK_CRYPT_ERR_PADDING_OVERFLOW`
- [out] out - 公钥加密后的数据
- [out] out_len - 加密后的数据长度

注意

1. pub必须正确设置，指明当前传递的key是明文，还是OTP KEY加密后的密文，该步骤会影响密钥的解析。
2. 不推荐原地加解密。

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);
```

功能

使用私钥对数据进行解密，支持PADDING解析。

参数

- [in] priv - rsa的私钥信息，见[rk_rsa_priv_key_pack](#)
- [in] padding - 加密填充格式，需要与加密时使用的格式一致。见[RK_RSA_CRYPT_PADDING](#)
- [in] in - 输入数据
- [in] in_len - 输入数据的长度（单位：byte）
- [out] out - 私钥解密后的数据
- [out] out_len - 解密后的数据长度

注意

1. priv 必须正确设置，指明当前传递的key是明文，还是OTP KEY加密后的密文，该步骤会影响密钥的解析。
2. 不推荐原地加解密。

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);
```

功能

使用私钥对数据进行加密，支持加密填充。

参数

- [in] priv - rsa的私钥信息，见[rk_rsa_priv_key_pack](#)
- [in] padding - 加密填充格式，见[RK_RSA_CRYPT_PADDING](#)

- [in] in - 输入数据
- [in] in_len - 输入数据的长度（单位：byte），输入数据过长将会返回错误码 RK_CRYPT_ERR_PADDING_OVERFLOW
- [out] out - 私钥加密后的数据
- [out] out_len - 加密后的数据长度

注意

1. priv必须正确设置，指明当前传递的key是明文，还是OTP KEY加密后的密文，该步骤会影响密钥的解析。
2. 不推荐原地加解密。

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);
```

功能

使用公钥对数据进行解密，支持PADDING解析。

参数

- [in] pub - rsa的公钥信息，见[rk_rsa_pub_key_pack](#)
- [in] padding - 加密填充格式，需要与加密时使用的格式一致。见[RK_RSA_CRYPT_PADDING](#)
- [in] in - 输入数据
- [in] in_len - 输入数据的长度（单位：byte）
- [out] out - 公钥解密后的数据
- [out] out_len - 解密后的数据长度

注意

1. pub必须正确设置，指明当前传递的key是明文，还是OTP KEY加密后的密文，该步骤会影响密钥的解析。
2. 不推荐原地加解密。

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);
```

功能

使用私钥对输入数据算hash或者直接使用外部传入的hash值，进行签名。

参数

- [in] priv - rsa的私钥信息，见[rk_rsa_priv_key_pack](#)
- [in] padding - 签名填充格式，见[RK_RSA_SIGN_PADDING](#)
- [in] in - 待签名的数据（可选），支持先算hash，后签名。
- [in] in_len - 待签名数据的长度（单位：byte）
- [in] hash - 哈希值（可选），根据填充算法确定哈希长度，支持直接对哈希值进行签名
- [out] out - 私钥签名后的数据
- [out] out_len - 签名后的数据长度

注意

1. priv必须正确设置，指明当前传递的key是明文，还是OTP KEY加密后的密文，该步骤会影响密钥的解析。
2. 当 `in != NULL && hash == NULL` 时，该接口会先对in计算hash值（hash算法由padding格式确定），然后再对hash值进行padding后签名。
3. 当 `hash != NULL` 时，该接口会直接对传入hash值进行padding后签名。

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);
```

功能

使用公钥对签名进行验签。

参数

- [in] pub - rsa的公钥信息，见[rk_rsa_pub_key_pack](#)
- [in] padding - 签名填充格式，需要与签名时使用的格式一致。见[RK_RSA_SIGN_PADDING](#)
- [in] in - 签名的原始数据（可选）
- [in] in_len - 签名的原始数据的长度（单位：byte）
- [in] hash - 签名的原始哈希值（可选）
- [in] sign - 签名数据
- [in] sign_len - 签名数据的长度（单位：byte）

注意

1. pub必须正确设置，指明当前传递的key是明文，还是OTP KEY加密后的密文，该步骤会影响密钥的解析。
2. 当 `in != NULL && hash == NULL` 时，该接口会先对in计算hash值（hash算法有padding格式确定），然后再对hash值进行验签。
3. 当 `hash != NULL` 时，该接口会直接对传入hash值进行验签。

debug日志

当前librkcrypto的日志划分为以下几个级别。

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

默认为 `TRACE_INFO`，可以通过设置环境变量的方式修改日志的级别（需要在librkcrypto库加载之前设置好环境变量才会生效），也可以在 `rk_crypto_init` 之前通过 `rkcrypto_set_trace_level` 接口来进行设置。

Android:

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

Linux:

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

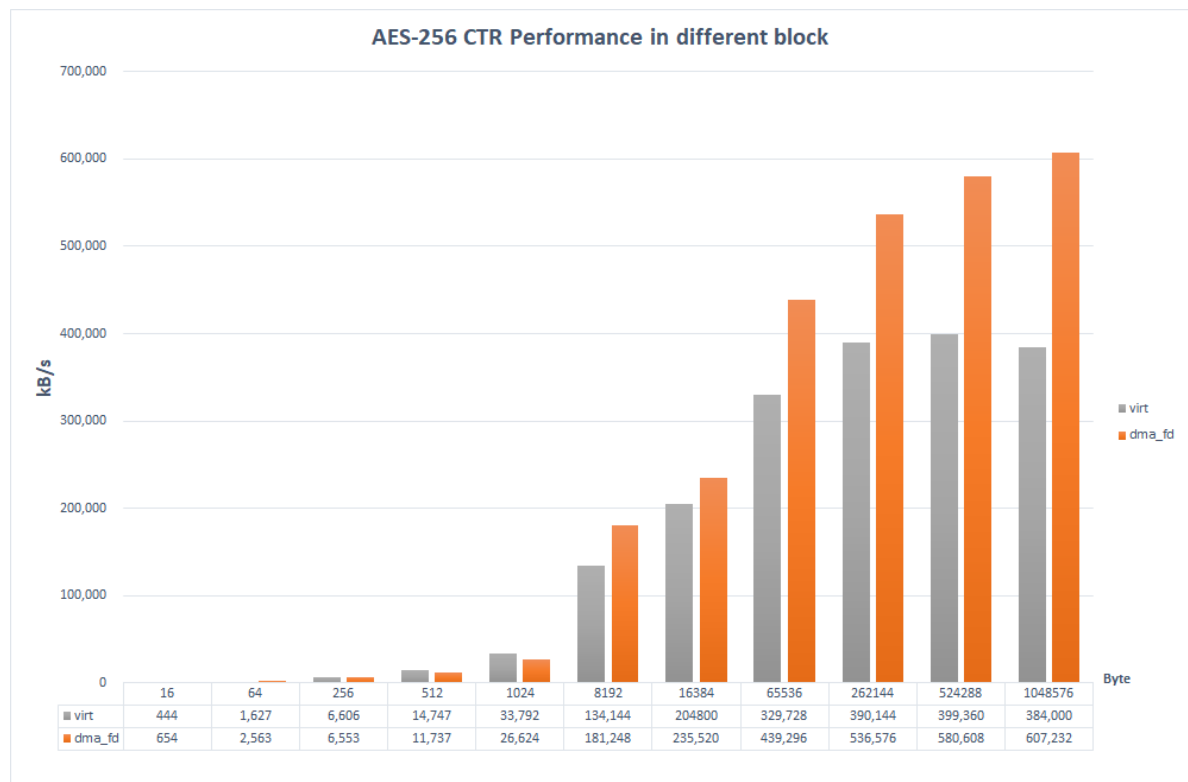
硬件 crypto 性能数据

以下列出了典型的芯片在uboot下的性能作为参考，具体详细的性能见librkcrypto源码中的perf_reports目录。perf_reports目录的性能为应用层调用kernel crypto内核驱动在最高CPU/DDR定频下的实测，由于系统调用的原因，性能会比uboot测试的性能略低。

影响性能的因素

1. 用户的数据对应的物理地址是否连续，起始地址是否64字节对齐。由于CRYPTO IP内部没有集成MMU，因此只能处理连续内存的数据，并且起始地址有对齐的要求。在使用虚拟地址virt的情况下，如果数据不满足地址对齐和连续性，应用的一次计算请求在驱动中有可能被拆分成多次CRYPTO硬件计算，将用户数据拷贝到驱动内部32K物理连续的内存中再进行计算，性能会受到影响。
2. 单次计算的分块大小，分块大小越大，实际性能越接近理论性能。CRYPTO驱动除了实际硬件计算消耗的时间外，还有诸如内核调用，任务队列，cache刷新等额外开销，额外开销的时间比较恒定。在分块小的情况下，额外开销的时间占比会增大，导致实际性能急剧下降。

针对分块大小对计算性能的影响，以RK3588的librkcrypto_test测试为例，测试性能结果如下：



由上图可以知道，优先使用1MByte分块大小，使用dma_fd的连续内存进行计算，可以发挥出较好的性能。针对小数据量如64KByte以下的数据，不建议使用硬件CRYPTO，优先使用软算法（有些芯片支持NEON/CE指令集，对加解密有加速作用）。

uboot 层硬件 crypto 性能数据

crypto v1 性能数据

测试环境（uboot RK3399）：

时钟：CRYPTO_CORE = 200M，不同芯片的最高频率略有不同

CIPHER/HASH 算法性能测试：

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 算法性能测试:

RSA 算法长度(nbits)	公钥加密/私钥解密 (ms)
2048	8 / 632

crypto v2 / v3 /v4 性能数据

这几个版本的CRYPTO IP计算核心没有发生大的变化，性能差异只体现在CRYPTO IP的工作主频上，CRYPTO V3的主频最高可以到350M，其理论性能可以在V2的基础上进行简单换算。

测试环境 (uboot RV1126 V2) :

时钟: CRYPTO_CORE = 200M, CRYPTO_PKA=300M, DDR=786M

Hash/HMAC: 总共测试 128M 的数据，每次计算 4M 的数据

DES/3DES/AES/SM4: 总共测试 128M 数据，每次计算 4M 的明文和 4M 的 aad 数据

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	-		
	ECB/CTR/XTS	320			-		
SM4	CBC/CFB/OFB/CTS	87			-		
	CMAC/CBC_MAC	89			-		
	CCM(data+aad)	156			-		
	GCM(data+aad)	114			-		
	ECB/CTR/XTS	320			-		

RSA 测试方法: 生成 rsa key, 包含 n, e, d, 执行加密和解密测试

加密测试: 密文 = $d^e \% n$

解密测试：明文 = $d^d \% n$

	ENC/DEC	Time(ms)
RSA-1024	ENC	< 1
	DEC	12
RSA-2048	ENC	1
	DEC	93
RSA-3072	ENC	1
	DEC	304
RSA-4096	ENC	2
	DEC	710

References

附录

术语