ICS 35.240
CCS L 70/84

T/ZISIA

团　　体　　标　　准

T/ZISIA 0102.1—2025

通用操作系统商用密码子系统功能调用接口规范 第1部分：内核态接口

General purpose operating system commercial cryptographic subsystem function calling interface specification
Part 1:Kernel mode interface

2025－12－31 发布　　2025－12－31 实施

中关村网络安全与信息化产业联盟　　发 布

目　　次

前　言

T/ ZISIA 0102《通用操作系统商用密码子系统功能调用接口规范》分为以下2个部分：

——第 1 部分：内核态接口

——第 2 部分：用户态接口

本文件为T/ ZISIA 0102《通用操作系统商用密码子系统功能调用接口规范》的第1部分。

本文件按照GB/T 1.1—2020《标准化工作导则 第1部分：标准化文件的结构和起草规则》的规定起草。

请注意本文件的某些内容可能涉及专利。本文件的发布机构不承担识别专利的责任。

本文件由中关村网络安全与信息化产业联盟提出并归口。

本文件起草单位：中关村网络安全与信息化产业联盟、麒麟软件有限公司、蚂蚁科技集团股份有限公司、中科方德软件有限公司、兴唐通信科技有限公司、北京天威诚信电子商务服务有限公司、阿里云计算有限公司、长春吉大正元信息技术股份有限公司。

本文件主要起草人：王震、张大朋、孟德慧、陈刚、张成龙、王宇辰、杨洋、洪澄、郭智慧、昌文婷、彭晋、胡昆、冯建茹、刘赢、郭海兵、魏刘虎、王辉、姚长远、王尧、李延昭、李世奇、张天佳、王克、胡金山、罗捷、刘海涛、林璟锵、何道君、申志军、王立臣。

引　言

T/ZISIA 0101—2025《通用操作系统商用密码子系统安全轮廓》要求操作系统密码子系统应具有OS内核态和用户态商密API，以及OS内核态和用户态商密资源挂接标准化接口。

本文件的主要目标是为通用操作系统密码子系统制定统一的内核态密码应用接口及密码资源挂接接口标准，为通用操作系统密码子系统产品的研制和使用，以及基于该类密码产品的应用开发提供标准依据和指导。

通用操作系统商用密码子系统功能调用接口规范
第 1 部分：内核态接口

1 范围

本文件规定了通用操作系统商用密码子系统内核态密码功能调用接口结构、数据类型定义及内核态商密API和内核态商密资源挂接接口。

本文件适用于通用操作系统密码子系统的研制和使用，以及基于该接口的密码模块或应用软件开发。

2 规范性引用文件

下列文件中的内容通过文中的规范性引用而构成本文件必不可少的条款。其中，注日期的引用文件，仅该日期对应的版本适用于本文件；不注日期的引用文件，其最新版本（包括所有的修改单）适用于本文件。

GB/T 32905 信息安全技术 SM3密码杂凑算法
GB/T 32907 信息安全技术 SM4分组密码算法
GB/T 32918（所有部分）信息安全技术 SM2椭圆曲线公钥密码算法
GB/T 35276 信息安全技术 SM2密码算法使用规范
GB/T 37092 信息安全技术 密码模块安全要求
GM/Z 4001 密码术语
T/ZISIA 0101-2025 通用操作系统商用密码子系统安全轮廓

3 术语和定义

GB/T 25069、GM/T 0028界定的以及下列术语和定义适用于本文件。

3.1
密码模块 cryptographic module
实现了安全功能的硬件、软件和/或固件的集合，并且被包含在密码边界内。
[来源：GB/T 37092，3.5]

3.2
引擎 engine
一种用于封装操作系统底层硬件或软件设备的机制，可为操作系统商密子系统提供标准化的密码资源挂接方法，实现商密资源的动态调度、性能优化和安全控制。

3.3
转换 transform
对一个可配置、可初始化的加密算法实现的抽象。它代表了执行特定密码学操作（如加密、解密、杂凑、生成认证标签等）所需的所有信息、状态和函数指针的集合。

3.4
回滚 rollback
将数据库或系统的状态从当前点撤销（Undo） 到一个更早的、已知的、一致性的状态的一系列操作。

4 符号和缩略语

下列缩略语适用于本文件。
API：应用程序接口/应用接口（Application Program Interface）
CA：证书认证机构（Certification Authority）

ECB：电码本模式（Electronic Code Book）
OS：操作系统（Operation System）
OSSM：操作系统商用密码（Operation System Commercial Cryptographic）
XTS：带密文挪用的XEX可调分组密码工作模式（XEX tweakable block cipher with Ciphertext Stealing）

5 密码功能调用接口框架

T/ZISIA 0101-2025【要求6.4-01】要求操作系统（OS）商密子系统应具有以下密码功能调用接口方式：

——OS 内核态、用户态商密 API。用于内核及用户态商密应用操作调用。

——OS 内核态、用户态商密资源挂接接口。用于商密子系统挂接 OS 内核态、用户态商密资源。

OS内核态和用户态商密资源包括：软硬件密码模块（如密码算法库、密码机、智能密码钥匙、TPM/TCM、CPU等）及其驱动程序、中间件等。

注：用户态和核心态使用的密码资源可以是同一密码资源，也可以是不同的密码资源。

本文件规范了OS内核态商密功能调用接口，包括OS内核态商密API（北向接口）和OS内核态商密资源挂接接口（南向接口）。北向接口由内核态商密应用操作调用，南向接口由密码资源厂家使用开发不同的商密资源。本文件规范的OS内核态商密功能调用接口框架如图1所示。

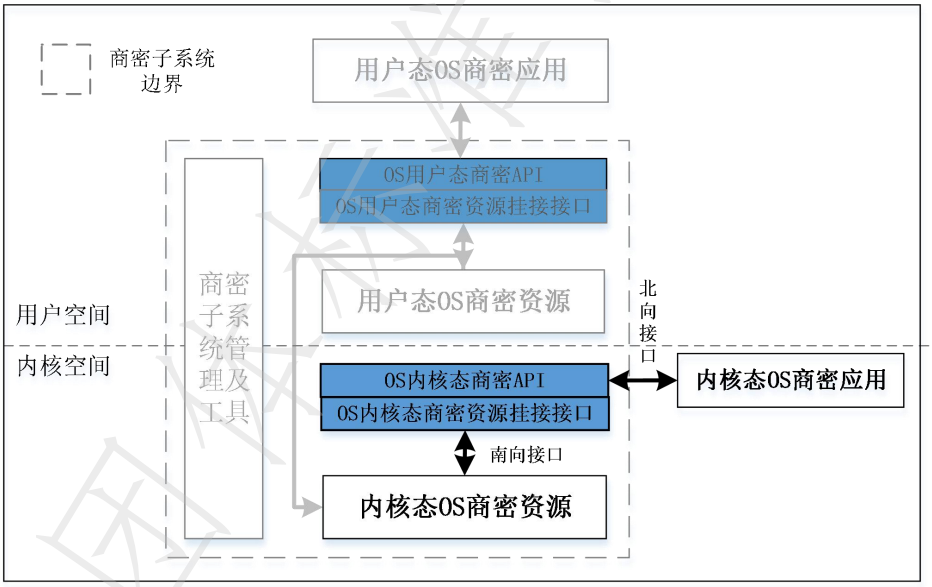

图 1 OS 内核态商密功能调用接口框架

6 数据类型的定义

6.1 算法标识

表 1 算法标识

算法标识	描述	OID
sm2	非对称算法	1.2.156.10197.1.301
sm3	杂凑算法	1.2.156.10197.1.401
sm4	对称算法	1.2.156.10197.1.104

6.2 基本数据类型定义

基本数据类型均按照C语言进行定义。

表 2 基本数据类型

类型名称	说明
char	有符号字符类型
unsigned char	无符号字符类型
int	有符号整数
unsigned int	无符号整数
unsigned long	无符号长整数
size_t	无符号整数
keylen	无符号整数
key_is_private	有符号整数，（0为假，非0为真）

6.3 数据结构定义

6.3.1 非对称密钥结构

原型：struct public_key；
描述：非对称密钥结构体，包含非对称密钥相关信息，具体内容由实现定义。
结构体成员说明：

成员名称	功能描述
key	密钥指针
keylen	以字节为单位的key长度
algo	算法对象标识符
params	算法参数指针
param_len	算法参数长度
key_is_private	密钥类型标识
id_type	密钥标识类型
pkey_algo	算法名称
key_eflags	密钥扩展标志位

密钥扩展标志（key_eflags 宏定义）：

宏定义	功能描述
KEY_EFLAG_CA	0，设置CA基本约束
KEY_EFLAG_DIGITALSIG	1，设置数字签名使用
KEY_EFLAG_KEYCERTSIGN	2，设置公钥用于签署其他证书

6.3.2 非对称算法数据结构

原型：struct ossm_crypto_sig；
描述：封装算法和核心处理逻辑的用户实体对象。
结构体成员说明：

成员名称	功能描述
base	通用加密算法实体对象

算法自测原型：struct akcipher_testvec；
描述：非对称密码测试。
结构体成员说明：

成员名称	功能描述
key	密钥指针
params	算法参数指针
m	明文（Message）数据指针，输入待加密或签名的原始数据
c	密文（Ciphertext）或签名结果指针，存储加密或签名后的输出数据。
keylen	以字节为单位的key长度
param_len	算法参数长度
m_size	明文数据大小
c_size	密文或签名数据大小
public_key_vec	标识测试是否使用公钥
siggen_sigver_test	标识测试类型
algo	算法对象标识符

6.3.3 对称算法数据结构

原型：struct skcipher_alg；
描述：对称密钥密码定义。
结构体成员说明：

成员名称	功能描述
min_keysize	转换支持的最小密钥大小。这是此转换算法支持的最小密钥长度。必须设置为预定义值之一，不针对特定硬件。该字段的可能值在“_MIN_KEY_SIZE” 描述。
max_keysize	转换支持的最大密钥大小。这是此转换算法支持的最大密钥长度。必须设置为预定义值之一，不针对特定硬件。该字段的可能值在“_MAX_KEY_SIZE”描述。
setkey	为转换设置密钥。该函数用于将提供的密钥编程到硬件中，或将密钥存储到转换上下文中，以便稍后进行编程。注意函数会修改变换上下文。此函数可在转换对象存在期间可被多次调用。须确保密钥被正确地重新编程到硬件中。该函数还负责检查密钥长度是否有效。如果在cra_init调用中设置了软件回退，那么在算法不支持所有密钥大小的情况下，该函数可能需要使用回退。
encrypt	加密区块的分散列表。此函数用于加密提供的包含数据块的分散列表。加密API用户负责正确对齐散列表的条目，并确保块的大小正确。如果在cra_init调用中设置了软件回退，那么在算法不支持所有密钥大小的情况下，该函数可能需要使用回退。如果密钥存储在转换上下文中，则可能需要在此函数中将密钥重新编程到硬件中。此函数不得修改转换上下文，因为此函数可能会与同一转换对象并行调用。
decrypt	解密单个区块。这是加密的反向对应函数，条件完全相同。
init	初始化密码转换对象。只在实例化时调用一次，即在转换上下文分配之后。如果加密硬件有一些特殊要求需要由软件处理，该函数应检查转换的精确要求，并将任何软件回退设置到位。
exit	取消初始化密码转换对象。这是init的对应函数，用于删除init中设置的各种

	更改。
ivsize	适用于转换的IV大小。用户必须提供执行加密或解密操作。
chunksize	区块大小，流密码情况下被设置为底层块大小。
walksize	与块大小相等，应该是块大小的倍数。除非在算法中并行操作多个块，效率更高。
stat	密码算法的统计数据
base	通用加密算法的定义。

除ivsize外，所有字段均为必填字段，必须填写。

原型：struct cipher_testvec；
描述：对称密码测试的结构。
结构体成员说明：

成员名称	功能描述
key	密钥指针
klen	以字节为单位的key长度
iv	指向IV的指针。如果为空，则使用全零的IV
iv_out	输出IV的指针（如果适用于该密码）
ptext	明文指针
ctext	密文指针
len	ptext和ctext 的长度（以字节为单位）
wk	测试是否需要 ossm_crypto_TFM_REQ_FORBID_WEAK_KEYS（例如，测试需要因弱密钥而失败）
fips_skip	在FIPS模式下跳过测试向量
generates_iv	加密时应忽略给定的IV，并输出iv_out。解密需要iv_out。需要用于AES Keywrap（“kw(sm4)”）
setkey_error	来自setkey()的预期错误。如果已设置，则不会测试加密或解密。
crypt_error	encrypt()和decrypt()的预期错误

6.3.4 杂凑算法标识和数据结构

异步信息摘要/异步密码操作；
原型：struct ossm_crypto_ahash；
描述：表示和操作异步杂凑算法的核心数据结构，是一个统一的接口句柄，用于执行杂凑计算、基于杂凑的消息认证码（HMAC）及其他依赖于杂凑算法的操作。描述算法本身的能力和实现，包含 init, update, final, setkey 等函数指针。
结构体成员说明：

成员名称	功能描述
init	初始化杂凑上下文
update	处理输入数据块，更新杂凑计算的中间状态
final	完成杂凑计算，生成最终的摘要（Digest）并清理上下文
finup	合并update和final 操作，处理最后一块数据并输出摘要
digest	单次操作完成完整杂凑计算（包括初始化、数据处理和摘要生成），适用于小数据块的快速处理
export	导出当前杂凑计算的中间状态
import	导入之前导出的中间状态，恢复杂凑计算的进度
setkey	设置密钥
statesize	算法所需的中间状态存储空间大小

reqsize	为ahash_request结构体分配的额外空间，用于存储私有数据或状态
base	继承自通用加密算法对象

同步消息摘要：

原型：struct ossm_crypto_shash；

描述：API 类型为 ossm_crypto_ALG_TYPE_SHASH，消息摘要API能够为调用者维护状态信息，同步消息摘要API可以在其shash_desc请求数据结构。

结构体成员说明：

成员名称	功能描述
descsize	杂凑算法所需的上下文状态（Context）的字节大小
base	继承自通用加密算法结构体

消息摘要码：

原型：struct ossm_crypto_shash *crypto_clone_shash；

描述：为信息摘要分配密码句柄。返回的结构ossm_crypto_shash 是密码句柄，在后续调用该信息摘要的应用程序接口时必须使用。

结构体成员说明：

成员名称	功能描述
ossm_crypto_shash	密码句柄，在后续调用该信息摘要的应用程序接口时必须使用

杂凑算法自测：

原型：struct hash_testvec；

描述：描述杂凑（消息摘要）测试的结构。

结构体成员说明：

成员名称	功能描述
key	密钥指针
plaintext	源数据指针
digest	预期摘要指针
psize	源数据长度
ksize	key长度，以字节为单位（无key时为 0）
setkey_error	来自setkey()的预期错误。如果已设置，则不会测试加密或解密。
digest_error	来自digest()的预期错误
fips_skip	在FIPS模式下跳过测试向量

6.3.5 加密硬件引擎挂载标识和数据结构

原型：struct ossm_crypto_engine；

描述：在内核层软件栈中，密码引擎硬件驱动注册到内核密码框架中，内核中其他模块可以通过密码框架来调用加密引擎。

结构体成员说明：

成员名称	功能描述
name	引擎名称
idling	空闲状态标志
busy	繁忙标志
running	运行状态标志
retry_support	重试支持标志：true表示硬件支持重新执行失败的积压请求（Backlog Request）

list	全局引擎链表节点：用于将引擎注册到内核全局加密引擎链表中
queue_lock	队列自旋锁：保护请求队列（queue）的并发访问，防止竞态条件
queue	加密请求队列：存储待处理的异步加密请求
dev	关联设备指针：指向与此引擎关联的硬件设备
rt	实时任务标志：true 表示请求处理线程以实时任务（Realtime Task）优先级运行
prepare_crypt_hardware	硬件准备回调：在请求到达前通知驱动准备硬件
unprepare_crypt_hardware	硬件释放回调：在无请求时通知驱动释放硬件资源
do_batch_requests	批量请求回调：执行多个请求的批量处理
kworker	内核工作线程：管理请求处理线程的任务队列
pump_requests	调度执行请求处理逻辑
priv_data	私有数据指针：指向引擎驱动的私有数据
cur_req	当前请求指针：指向正在处理的异步加密请求

6.3.6 加密硬件引擎操作标识和数据结构

原型：struct ossm_crypto_engine_op；

描述：加密硬件引擎操作。加密引擎只管理ossm_crypto_async_request形式的异步请求。它无法知道底层请求类型，因此只能访问转换结构。必须在转换上下文的开头加入ossm_crypto_engine结构。

结构体成员说明：

成员名称	功能描述
do_one_request	对当前请求进行加密

6.3.7 分散列表

原型：struct scatterlist；

描述：内核中用于高效管理非连续内存块（即“分散”在物理内存中的多个缓冲区）的数据结构，将多个物理上不连续的缓冲区在逻辑上组装成一个连续的序列。

结构体成员说明：

成员名称	功能描述
page_link	无符号长整数，包含指向内存页的指针和控制信息，存放页面指针并且对关于散列表（sg table）的信息进行编码。最低的两位（bit）被保留用于存储此信息。如果位0（bit 0）被设置，那么page_link包含一个指向下一个散列表（sg table list）的指针。否则，下一个条目位于sg + 1（即当前散列表项的下一个数组元素）。如果位1（bit 1）被设置，那么此散列表（sg）条目是一个列表中的最后一个元素。
offset	指定数据缓冲区在它所在物理内存页中的起始偏移量（以字节为单位）
length	指定了从 offset 开始的这个连续缓冲区的长度（以字节为单位）
dma_address	专为 DMA（直接内存访问）设计，是设备可以访问的总线地址
dma_length	专为 DMA（直接内存访问）设计，是映射的长度

7 OS 内核态商密 API

7.1 概述

OS内核态商密API是OS商密北向接口，内核态商密应用可调用注册到商密子系统中的密码算法、硬件密码资源。包括非对称算法类、对称算法类、杂凑算法类等接口函数。

7.2 非对称算法类接口

7.2.1 概述

非对称算法接口包括分配签名tfm句柄、释放签名tfm句柄、获取输出缓冲区的长度、签名、签名验证、设置公钥操作、设置私钥操作等函数。在分配句柄时，通过参数const char *alg_name传入“sm2”指定商密算法。

表 3 非对称算法类接口

接口名称	说明
ossm_crypto_alloc_sig()	分配签名tfm句柄
ossm_crypto_sig_maxsize()	获取输出缓冲区的长度
ossm_crypto_sig_set_pubkey()	设置公钥
ossm_crypto_sig_set_privkey()	设置私钥
ossm_crypto_sig_sign()	签名
ossm_crypto_sig_verify()	签名验证
ossm_crypto_free_sig()	释放签名tfm句柄

7.2.2 分配签名 tfm 句柄

原型：struct ossm_crypto_sig *crypto_alloc_sig(const char *alg_name, u32 type, u32 mask)；

描述：为公钥签名算法分配一个句柄。返回的crypto_sig结构是后续调用API进行签名操作时所需的句柄。

参数：

参数	功能描述
alg_name	是sm2签名算法的cra_name/名称或对应驱动程序名称“sm2”
type	指定算法类型
mask	指定算法的掩码

返回值：成功时，返回已分配的句柄；出错时，IS_ERR()为true，PTR_ERR()返回错误代码。

7.2.3 获取输出缓冲区的长度

原型：int ossm_crypto_sig_maxsize(struct ossm_crypto_sig *tfm)；

描述：获取输出缓冲区的长度，函数返回给定密钥所需的目标缓冲区大小。函数假定密钥已在转换中设置。如果在没有设置密钥或设置密钥失败的情况下调用此函数，将导致取消引用 NULL。

参数：

参数	功能描述
tfm	使用ossm_crypto_alloc_sig()分配的签名 tfm 句柄

返回值：返回给定密钥所需的目标缓冲区大小。

7.2.4 设置公钥

原型：int ossm_crypto_sig_set_pubkey(struct ossm_crypto_sig *tfm, const void *key, unsigned int keylen)；

描述：函数调用特定算法的设置密钥函数，该函数知道如何解码和解释编码密钥和参数。

参数：

参数	功能描述
tfm	使用ossm_crypto_alloc_sig()分配的签名tfm句柄
key	BER 编码的公钥、算法 OID、paramlen、BER编码的参数
keylen	密钥长度（不包括其他数据）

返回值：成功时返回0；出错时返回错误代码。

7.2.5 设置私钥

原型：int ossm_crypto_sig_set_privkey(struct ossm_crypto_sig *tfm, const void *key, unsigned int keylen)；

描述：函数调用特定算法的设置密钥函数，该函数知道如何解码和解释编码密钥和参数。

参数：

参数	功能描述
tfm	使用ossm_crypto_alloc_sig()分配的签名tfm句柄
key	BER编码的公钥、算法 OID、paramlen、BER编码的参数
keylen	密钥长度（不包括其他数据）

返回值：成功时返回0；出错时返回错误代码。

7.2.6 签名

原型：int ossm_crypto_sig_sign(struct ossm_crypto_sig *tfm, const void *src, unsigned int slen, void *dst, unsigned int dlen)；

描述： 函数调用给定算法的特定签名操作。

参数：

参数	功能描述
tfm	使用ossm_crypto_alloc_sig()分配的签名tfm句柄
src	源缓冲区
slen	源长度
dst	目标缓冲区
dlen	目标长度

返回值：成功时返回0；出错时返回错误代码。

7.2.7 签名验证

原型：int ossm_crypto_sig_verify(struct ossm_crypto_sig *tfm, const void *src, unsigned int slen, const void *digest, unsigned int dlen)；

描述： 函数为给定算法调用特定的签名验证操作。

参数：

参数	功能描述
tfm	使用ossm_crypto_alloc_sig()分配的签名tfm句柄
src	源缓冲区
slen	源长度
digest	摘要
dlen	目标长度

返回值：成功时返回0；出错时返回错误代码。

7.2.8 释放签名 tfm 句柄

原型：static inline void ossm_crypto_free_sig(struct ossm_crypto_sig *tfm)；

描述：释放签名tfm句柄。

参数：

参数	功能描述
tfm	使用ossm_crypto_alloc_sig()分配的签名tfm句柄。如果tfm为NULL或错误指针，则此函数不执行任何操作。

返回值：无。

7.3 对称算法类接口

7.3.1 概述

对称加密接口包括分配对称密钥密码句柄、释放对称密钥密码句柄、搜索算法是否存在、获取IV大小、获取对称密钥密码块大小、获取对称密钥密码块大小、设置密钥、从请求中获取转换句柄、加密操作、解密操作、获取请求数据结构的大小、更新请求中的密码句柄引用、分配请求数据结构、归零并释放请求的数据结构、设置异步回调函数、设置数据缓冲区等函数。

表 4 对称算法类接口

接口名称	说明
ossm_crypto_has_skcipher()	搜索对称加密算法是否存在
ossm_crypto_alloc_skcipher()	分配对称密钥密码句柄
ossm_crypto_skcipher_ivsize()	获取IV大小
ossm_crypto_skcipher_blocksize()	获取块大小
ossm_crypto_skcipher_chunksize()	获取分配的块大小
ossm_crypto_skcipher_reqsize()	获取请求数据结构的大小
ossm_crypto_skcipher_setkey()	设置密钥
ossm_skcipher_request_alloc()	分配请求数据结构
ossm_skcipher_request_set_tfm()	更新请求中的密码句柄引用
ossm_skcipher_request_set_callback()	设置异步回调函数
ossm_skcipher_request_set_crypt()	设置数据缓冲区
ossm_crypto_skcipher_reqtfm()	从请求中获取tfm句柄
ossm_crypto_skcipher_encrypt()	加密操作
ossm_crypto_skcipher_decrypt()	解密操作
ossm_skcipher_request_free()	归零并释放请求的数据结构
ossm_crypto_free_skcipher()	释放对称密钥密码句柄

7.3.2 搜索对称加密算法是否存在

原型：int ossm_crypto_has_skcipher(const char *alg_name, u32 type, u32 mask)；
描述：搜索对称加密算法是否存在。
参数：

参数	功能描述
alg_name	sm4加密密钥的名称或sm4驱动程序名称
type	指定密码类型
mask	指定密码操作的保留位掩码，一般由算法资源实现者定义，默认为0

返回值：成功时返回true；否则返回false。

7.3.3 分配对称密钥密码句柄

原型：struct ossm_crypto_skcipher *crypto_alloc_skcipher(const char *alg_name, u32 type, u32 mask)；
描述：分配密码句柄时，调用者只需指定密码类型。然而，对称密码通常支持多种密钥大小。这些密钥大小取决于所提供密钥的长度，接口不提供选择特定对称密码密钥大小的单独方法。返回的struct ossm_crypto_skcipher是后续API调用时所需的密码句柄。
参数：

参数	功能描述
alg_name	sm4加密密钥的名称或sm4驱动程序名称
type	指定密码类型
mask	指定密码的掩码

返回值：成功时返回已分配的密码句柄；出错时IS_ERR()为true，PTR_ERR()返回错误代码。

7.3.4 获取 IV 大小

原型：static inline unsigned int ossm_crypto_skcipher_ivsize(struct ossm_crypto_skcipher *tfm)
描述：返回密码句柄引用的对称算法的 IV大小，返回值以字节为单位的IV大小，如果不需要IV，IV可能为零。
参数：

参数	功能描述
tfm	密码tfm句柄

返回值：返回IV字节大小。

7.3.5 获取块大小

原型：static inline unsigned int ossm_crypto_skcipher_blocksize(struct ossm_crypto_skcipher *tfm)；
描述：获取块大小。
参数：

参数	功能描述
tfm	密码句柄

返回值：返回block字节。

7.3.6 获取分配的块大小

原型：static inline unsigned int ossm_crypto_skcipher_chunksize(struct ossm_crypto_skcipher *tfm)；
描述：获取分配的块大小，返回分配的块大小字节。
参数：

参数	功能描述
tfm	密码句柄

返回值：返回block字节。

7.3.7 获取请求数据结构的大小

原型：static inline unsigned int ossm_crypto_skcipher_reqsize(struct ossm_crypto_skcipher *tfm)；
描述：获取请求数据结构的大小。
参数：

参数	功能描述
tfm	密码句柄

返回值：返回字节数。

7.3.8 设置密钥

原型：int ossm_crypto_skcipher_setkey(struct ossm_crypto_skcipher *tfm, const u8 *key, unsigned int keylen)；
描述：设置密钥。
参数：

参数	功能描述
tfm	密码句柄
key	密钥缓存
keylen	密钥字节长度

返回值：成功返回0，其余返回<0。

7.3.9 分配请求数据结构

原型：static inline struct ossm_skcipher_request *skcipher_request_alloc(struct ossm_crypto_skcipher *tfm, gfp_t gfp)；
描述：分配请求数据结构。
参数：

参数	功能描述

tfm	要在请求中注册的密码句柄
gfp	由API调用传递给kmalloc的内存分配标志

返回值：成功则返回请求句柄，失败则为空。

7.3.10 更新请求中的密码句柄引用

原型：static inline void ossm_skcipher_request_set_tfm(struct ossm_skcipher_request *req, struct ossm_crypto_skcipher *tfm)；

描述：更新请求中的密码句柄引用。允许调用者使用不同的数据结构替换请求中已有的对称算法句柄。

参数：

参数	功能描述
req	要被修改的请求句柄
tfm	要加入请求句柄中的密码句柄

返回值：无。

7.3.11 设置异步回调函数

原型：static inline void ossm_skcipher_request_set_callback(struct ossm_skcipher_request *req, u32 flags, crypto_completion_t compl, vid *data)；

描述：设置异步回调函数该函数允许设置密码操作完成后触发的回调函数。回调函数使用skcipher_request句柄注册，必须符合模板。此函数允许设置在密码操作完成后触发回调函数。

参数：

参数	功能描述
req	请求句柄
flags	指定零或多个标志的 ORing CRYPTO_TFM_REQ_MAY_BACKLOG 表示请求队列可能会回溯日志，并使等待队列超过初始最大值； CRYPTO_TFM_REQ_MAY_SLEEP 表示请求处理可能会休眠。
compl	将与请求句柄一起注册的回调函数指针
data	数据指针指的是内核加密API不使用，但提供给回调函数使用的内存。在这里，调用者可以提供回调函数可以操作的内存引用。由于回调函数是异步调用相关功能的，因此可能需要访问相关功能的数据结构，而这些数据结构可以使用该指针引用。回调函数可通过提供给回调函数的ossm_crypto_async_request数据结构中的“data”字段访问内存。

返回值：无。

7.3.12 设置数据缓冲区

原型：static inline void ossm_skcipher_request_set_crypt(struct ossm_skcipher_request *req, struct scatterlist *src, struct scatterlist *dst, unsigned int cryptlen, void *iv)；

描述：该函数允许设置源数据和目标数据分散/收集列表。加密时，源数据被视为明文，目标数据被视为密文。在解密操作中，使用方法正好相反——源数据是密文，目标数据是明文。

参数：

参数	功能描述
req	请求句柄
src	源散点图/收集列表
dst	目标散点图/收集列表
cryptlen	从src开始要处理的字节数
iv	密码操作的IV，必须符合ossm_crypto_skcipher_ivsize 定义的IV大小

返回值：无。

7.3.13 获取请求中的句柄

原型：static inline struct ossm_crypto_skcipher *crypto_skcipher_reqtfm(struct skcipher_request *req)；
描述：获取请求中的句柄。
参数：

参数	功能描述
req	从中获取密码句柄的请求

返回值：无。

7.3.14 加密

原型：int ossm_crypto_skcipher_encrypt(struct skcipher_request *req)；
描述：对明文进行加密操作，设置成功返回0，否则<0。
参数：

参数	功能描述
req	对skcipher_request句柄的引用，该句柄保存执行密码操作所需的所有信息

返回值：成功返回0，其余返回<0。

7.3.15 解密

原型：int ossm_crypto_skcipher_decrypt(struct skcipher_request *req)；
描述：对密文进行解密，设置成功返回0，否则<0。
参数：

参数	功能描述
req	对skcipher_request句柄的引用，该句柄保存执行密码操作所需的所有信息

返回值：成功返回0，其余返回<0。

7.3.16 归零并释放请求的数据结构

原型：static inline void ossm_skcipher_request_free(struct ossm_skcipher_request *req)；
描述：归零并释放请求的数据结构。
参数：

参数	功能描述
req	请求释放数据结构密码句柄

返回值：无。

7.3.17 归零并释放对称密钥密码句柄

原型：static inline void ossm_crypto_free_skcipher(struct ossm_crypto_skcipher *tfm)；
描述：归零并释放密码句柄。
参数：

参数	功能描述
tfm	待释放的句柄。如果tfm为NULL或错误指针，则此函数不执行任何操作。

返回值：成功时返回0；出错时返回错误代码。

7.4 杂凑算法类接口

7.4.1 异步消息摘要/异步密码操作接口

7.4.1.1 概述

异步消息摘要/异步密码操作接口包括分配杂凑密码句柄、归零并释放杂凑密码句柄、搜索杂凑密码可用性、获取密码块大小、获取消息摘要大小、获取杂凑状态的大小、获取请求中的密码句柄、获取请求数据结构的大小、设置密钥、更新并最终确定消息摘要、计算消息摘要、计算缓冲区的消息摘要、提取当前消息摘要状态、导入消息摘要状态性、初始化/重新初始化消息摘要句柄、向消息摘要中更新数据以进行处理等函数。

表 5 异步消息摘要/异步密码操作接口

接口名称	说明
ossm_crypto_has_ahash()	搜索杂凑密码可用性
ossm_crypto_alloc_ahash()	分配杂凑密码句柄
ossm_crypto_ahash_blocksize()	获取密码块大小
ossm_crypto_ahash_digestsize()	获取消息摘要大小
ossm_crypto_ahash_statesize()	获取杂凑状态的大小
ossm_crypto_ahash_init()	初始化/重新初始化消息摘要句柄
ossm_crypto_ahash_setkey	设置密钥
ossm_crypto_ahash_reqsize()	获取请求数据结构的大小
ossm_crypto_ahash_reqtfm()	获取请求中的密码句柄
ossm_crypto_ahash_update()	更新操作状态句柄的消息摘要状态
ossm_crypto_ahash_export()	提取当前消息摘要状态
ossm_crypto_ahash_import()	导入已保存的摘要状态
ossm_crypto_ahash_finup()	更新并最终确定消息摘要
ossm_crypto_ahash_final()	计算消息摘要
ossm_crypto_ahash_digest()	计算缓冲区的消息摘要
ossm_crypto_free_ahash()	归零并释放杂凑密码句柄

7.4.1.2 搜索杂凑密码可用性

原型：int ossm_crypto_has_ahash(const char *alg_name, u32 type, u32 mask)；
描述：搜索杂凑密码可用性。
参数：

参数	功能描述
alg_name	算法名称
type	指定密码类型
mask	指定密码的掩码

返回值：成功时返回true，否则返回false。

7.4.1.3 分配杂凑密码句柄

原型：struct ossm_crypto_ahash *crypto_alloc_ahash(const char *alg_name, u32 type, u32 mask)；
描述：分配杂凑密码句柄。
参数：

参数	功能描述
alg_name	算法名称
type	指定密码类型
mask	指定密码的掩码

返回值：成功时分配密码句柄；IS_ERR（）在发生错误时为true，PTR_ERR（）返回错误代码。

7.4.1.4 获取密码块大小

原型：static inline unsigned int ossm_crypto_ahash_blocksize(struct ossm_crypto_ahash *tfm)；
描述：获取密码块大小。
参数：

参数	功能描述
tfm	密码句柄

返回值：返回用密码句柄引用的消息摘要密码的块大小。

7.4.1.5 获取消息摘要大小

原型：static inline unsigned int ossm_crypto_ahash_digestsize(struct ossm_crypto_ahash *tfm)；
描述：获取消息摘要大小。
参数：

参数	功能描述
tfm	密码句柄

返回值：返回由密码句柄引用的消息摘要密码创建的消息摘要的大小。

7.4.1.6 获取杂凑状态的大小

原型：static inline unsigned int ossm_crypto_ahash_statesize(struct ossm_crypto_ahash *tfm)；
描述：获取杂凑状态的大小。使用ossm_crypto_ahash_export()函数，调用者可以将状态导出到一个缓冲区，该缓冲区的大小由该函数定义。
参数：

参数	功能描述
tfm	密码句柄

返回值：返回杂凑状态的大小。

7.4.1.7 初始化/重新初始化消息摘要句柄

原型：static inline int ossm_crypto_ahash_init(struct ahash_request *req)；
描述：初始化/重新初始化消息摘要句柄。
参数：

参数	功能描述
req	已经使用ahash_request_* API函数用所有必要的数据初始化的ahash_request句柄

返回值：成功返回0；异常返回<0。

7.4.1.8 设置密钥

原型：int ossm_crypto_ahash_setkey(struct ossm_crypto_ahash *tfm, const u8 *key, unsigned int keylen)；
描述：设置密码密钥句柄。为杂凑密码设置调用方提供的密钥。为了使此函数成功，密码句柄必须指向一个键控散列。
参数：

参数	功能描述
tfm	密码句柄
key	密钥缓存
keylen	密钥字节长度

返回值：成功返回0；异常返回<0。

7.4.1.9 获取请求数据结构的大小

原型：static inline unsigned int ossm_crypto_ahash_reqsize(struct ossm_crypto_ahash *tfm)；
描述：获取请求数据结构的大小。
参数：

参数	功能描述
tfm	密码句柄

返回值：返回请求数据的大小。

7.4.1.10 获取请求中的密码句柄

原型：static inline struct ossm_crypto_ahash *crypto_ahash_reqtfm(struct ahash_request *req)；
描述：从请求中获取密码句柄。
参数：

参数	功能描述

req	包含对杂凑密码句柄的引用的异步请求句柄。

返回值：返回注册在异步请求句柄ahash_request中的杂凑密码句柄。

7.4.1.11 更新操作状态句柄的消息摘要状态

原型：static inline int ossm_crypto_ahash_update(struct ahash_request *req)；
描述：向消息摘要中添加数据以进行处理。
参数：

参数	功能描述
req	已经使用ahash_request_* API函数用所有必要的数据初始化的ahash_request句柄

返回值：无。

7.4.1.12 提取当前消息摘要状态

原型：static inline int ossm_crypto_ahash_export(struct ahash_request *req, void *out)；
描述：提取当前消息摘要状态。
参数：

参数	功能描述
req	对状态导出的ahash_request句柄的引用
out	足够大小的输出缓冲区，可以保存杂凑状态

返回值：成功返回0；异常返回<0。

7.4.1.13 导入已保存的摘要状态

原型：static inline int ossm_crypto_ahash_import(struct ahash_request *req, const void *in)；
描述：导入已保存的消息摘要状态。
参数：

参数	功能描述
req	对状态导出的ahash_request句柄的引用
in	保存状态的缓冲区

返回值：成功返回0；异常返回<0。

7.4.1.14 更新并最终确定消息摘要

原型：int ossm_crypto_ahash_finup(struct ahash_request *req)；
描述：更新并最终确定消息摘要。对包含执行密码操作所需的所有信息的ahash_request句柄的引用。
参数：

参数	功能描述
req	对包含执行密码操作所需的所有信息的ahash_request句柄的引用

返回值：无。

7.4.1.15 计算消息摘要

原型：int ossm_crypto_ahash_final(struct ahash_request *req)；
描述：计算消息摘要。
参数：

参数	功能描述
req	对包含执行密码操作所需的所有信息的ahash_request句柄的引用

返回值：正常计算则EINPROGRESS返回0，如果队列已满，应稍后重新提交请求返回EBUSY，其他情况<0。

7.4.1.16 计算缓冲区的消息摘要

原型：int ossm_crypto_ahash_digest(struct ahash_request *req)；

描述：计算缓冲区的消息摘要。

参数：

参数	功能描述
req	对包含执行密码操作所需的所有信息的ahash_request句柄的引用

返回值：无。

7.4.1.17 归零并释放杂凑密码句柄

原型：static inline void ossm_crypto_free_ahash(struct ossm_crypto_ahash *tfm)；

描述：归零并释放杂凑密码句。成功时分配密码句柄；IS_ERR（）在发生错误时为true，PTR_ERR（）返回错误代码。

参数：

参数	功能描述
tfm	待释放的句柄。如果tfm为NULL或错误指针，则此函数不执行任何操作。

返回值：成功时分配密码句柄；IS_ERR（）在发生错误时为true，PTR_ERR（）返回错误代码。

7.4.2 异步杂凑请求句柄

7.4.2.1 概述

请求数据结构包含指向异步密码操作所需数据的所有指针，包括密码句柄（可由多个请求实例使用）、指向明文和消息摘要输出缓冲区的指针、异步回调函数等。

表 6 异步杂凑请求句柄接口

接口名称	说明
ossm_ahash_request_alloc()	分配请求数据结构
ossm_ahash_request_set_tfm()	更新请求中的密码句柄引用
ossm_ahash_request_set_callback()	设置异步回调函数
ossm_ahash_request_set_crypt()	设置数据缓冲区
ossm_ahash_request_free()	归零并释放请求的数据结构

7.4.2.2 分配请求数据结构

原型：static inline struct ossm_ahash_request *ahash_request_alloc(struct ossm_crypto_ahash *tfm, gfp_t gfp)；

描述：分配请求数据结构。

参数：

参数	功能描述
tfm	要在请求中注册的密码句柄
gfp	由API调用传递给kmalloc的内存分配标志

返回值：成功返回请求句柄，失败则为空。

7.4.2.3 更新请求中的密码句柄引用

原型：static inline void ossm_ahash_request_set_tfm(struct ossm_ahash_request *req, struct ossm_crypto_ahash *tfm)；

描述：更新请求中的密码句柄引用。允许调用者使用不同的数据结构替换请求中已有的对称算法句柄。

参数：

参数	功能描述
req	要被修改的请求句柄
tfm	要加入请求句柄中的密码句柄

返回值：无。

7.4.2.4 设置异步回调函数

原型：static inline void ossm_ahash_request_set_callback(struct ossm_ahash_request *req, u32 flags, crypto_completion_t compl, void *data)；

描述：设置异步回调函数。设置异步回调函数该函数允许设置密码操作完成后触发的回调函数。回调函数使用skcipher_request句柄注册，必须符合模板。此函数允许设置在密码操作完成后触发回调函数。

参数：

参数	功能描述
req	请求句柄
flags	指定零或多个标志的 ORing CRYPTO_TFM_REQ_MAY_BACKLOG表示请求队列可能会回溯日志，并使等待队列超过初始最大值； CRYPTO_TFM_REQ_MAY_SLEEP表示请求处理可能会休眠。
compl	将与请求句柄一起注册的回调函数指针
data	数据指针指的是内核加密API不使用，但提供给回调函数使用的内存。在这里，调用者可以提供回调函数可以操作的内存引用。由于回调函数是异步调用相关功能的，因此可能需要访问相关功能的数据结构，而这些数据结构可以使用该指针引用。回调函数可通过提供给回调函数的ossm_crypto_async_request数据结构中的“data”字段访问内存。

返回值：无。

7.4.2.5 设置数据缓冲区

原型：static inline void ossm_ahash_request_set_crypt(struct ossm_ahash_request *req, struct scatterlist *src, u8 *result, unsigned int nbytes);

描述：该函数允许设置源数据和目标数据分散/收集列表。加密时，源数据被视为明文，目标数据被视为密文。在解密操作中，使用方法正好相反——源数据是密文，目标数据是明文。

参数：

参数	功能描述
req	要更新的请求句柄
src	源散点图/收集列表
result	由消息摘要填充的缓冲区，调用者必须确保缓冲区有足够的空间，例如，通过调用ossm_crypto_ahash_digestsize（）
nbytes	要从源分散/收集列表中处理的字节数

返回值：无。

7.4.2.6 归零并释放请求的数据结构

原型：static inline void ossm_ahash_request_free(struct ossm_ahash_request *req)；

描述：归零并释放请求的数据结构。

参数：

参数	功能描述
req	请求释放数据结构密码句柄

返回值：无。

7.4.3 同步消息摘要接口

7.4.3.1 概述

同步消息摘要API与类型为ossm_CRYPTO_ALG_TYPE_SHASH的密码一起使用。消息摘要API能够维护调用者的状态信息。同步消息摘要API可以在其shash_desc请求数据结构中存储与用户相关的上下文。

表 7 同步杂凑请求句柄接口

接口名称	说明
ossm_crypto_alloc_shash()	分配消息摘要句柄
ossm_crypto_shash_blocksize()	获取密码块大小
ossm_crypto_shash_digestsize()	获取消息摘要大小
ossm_crypto_shash_descsize()	获取操作状态大小
ossm_crypto_shash_setkey()	设置消息摘要密钥
ossm_crypto_shash_init()	初始化/重新初始化消息摘要
ossm_crypto_shash_update()	更新操作状态句柄的消息摘要状态
ossm_crypto_shash_export()	提取消息摘要的操作状态
ossm_crypto_shash_import()	导入已保存的摘要状态
ossm_crypto_shash_final()	完成消息摘要操作
ossm_crypto_shash_finup()	完成消息摘要并将摘要输出到指定的缓冲区
ossm_crypto_shash_digest()	计算缓冲区的消息摘要
ossm_crypto_shash_tfm_digest()	计算转换对象缓冲区的消息摘要
ossm_crypto_free_shash()	归零并释放消息摘要句柄
ossm_crypto_transfer_hash_request_to_engine()	杂凑算法传入
ossm_crypto_finalize_hash_request()	杂凑算法结束

7.4.3.2 分配消息摘要句柄

原型：struct ossm_crypto_shash *crypto_alloc_shash(const char *alg_name, u32 type, u32 mask)；
描述：分配消息摘要密码句柄。返回的struct ossm_crypto_shash是该消息摘要的任何后续API调用都需要的密码句柄。
参数：

参数	功能描述
alg_name	sm3名称或sm3驱动程序名称
type	指定密码类型
mask	指定密码的掩码

返回值：成功时分配密码句柄，IS_ERR（）在发生错误时为true，PTR_ERR（）返回错误代码。

7.4.3.3 获取密码块大小

原型：static inline unsigned int ossm_crypto_shash_blocksize(struct ossm_crypto_shash *tfm)；
描述：获取密码块大小。
参数：

参数	功能描述
tfm	密码句柄

返回值：返回用密码句柄引用的消息摘要密码的块大小。

7.4.3.4 获取消息摘要大小

原型：static inline unsigned int ossm_crypto_shash_digestsize(struct ossm_crypto_shash *tfm)；
描述：获取消息摘要大小。
参数：

参数	功能描述
tfm	密码句柄

返回值：返回用密码句柄引用的消息摘要密码的块大小。

7.4.3.5 获取操作状态大小

原型：static inline unsigned int ossm_crypto_shash_descsize(struct ossm_crypto_shash *tfm)；
描述：获取操作状态大小。这个大小是计算内存需求所必需的，以允许调用者为操作状态分配足够的内存。
参数：

参数	功能描述
tfm	密码句柄

返回值：返回密码句柄引用的散列返回密码在操作期间所需的操作状态的大小。

7.4.3.6 设置消息摘要密钥

原型：static inline unsigned int ossm_crypto_shash_setkey (tfm, key, key_len)；
描述：设置消息摘要密钥。为消息摘要密码设置调用方提供的密钥。为了使此函数成功，密码句柄必须指向一个密钥消息摘要密码。上下文可为任意值。
参数：

参数	功能描述
tfm	密码句柄
key	密钥缓存
keylen	密钥字节长度

返回值：成功返回0；异常返回<0。

7.4.3.7 初始化/重新初始化消息摘要

原型：static inline int ossm_crypto_shash_init(struct shash_desc *desc)；
描述：调用初始化/重新初始化由操作状态句柄引用的消息摘要。先前操作创建的任何可能存在的状态都将被丢弃。上下文可为任意值。
参数：

参数	功能描述
desc	已经填充的操作状态句柄

返回值：成功返回0；异常返回<0。

7.4.3.8 更新操作状态句柄的消息摘要状态

原型：int ossm_crypto_shash_update(struct shash_desc *desc, const u8 *data, unsigned int len)；
描述：更新操作状态句柄的消息摘要状态。向消息摘要中添加数据以进行处理。上下文可为任意值。
参数：

参数	功能描述
desc	已经初始化的操作状态句柄
data	要添加到消息摘要中的输入数据
len	输入数据长度

返回值：成功返回0；异常返回<0。

7.4.3.9 提取消息摘要的操作状态

原型：ossm_crypto_shash_export；
描述：提取消息摘要的操作状态。这个函数将操作状态句柄的杂凑状态导出到调用者分配的输出缓冲区中，该缓冲区必须足够大（例如通过调用ossm_crypto_shash_descsize）。上下文可为任意值。
参数：

参数	功能描述
desc	对状态被导出的操作状态句柄的引用
out	足够大小的输出缓冲区，可以保存杂凑状态

返回值：成功返回0；异常返回<0。

7.4.3.10 导入已保存的摘要状态

原型：static inline int ossm_crypto_shash_import(struct shash_desc *desc, const void *in)；
描述：导入已保存的摘要状态。上下文可为任意值。
参数：

参数	功能描述
desc	引用导入状态的操作状态句柄
in	保存状态的缓冲区

返回值：成功返回0；异常返回<0。

7.4.3.11 完成消息摘要操作

原型：int ossm_crypto_shash_final(struct shash_desc *desc, u8 *out)；
描述：完成消息摘要操作，并基于添加到密码句柄中的所有数据创建消息摘要。消息摘要放在输出缓冲区中。调用者必须通过使用ossm_crypto_shash_digestsize来确保输出缓冲区足够大。上下文可为任意值。
参数：

参数	功能描述
desc	已经填充了数据的操作状态句柄
out	用消息摘要填充的输出缓冲区

返回值：成功返回0；异常返回<0。

7.4.3.12 完成消息摘要操作并将摘要输出到指定的缓冲区

原型：int ossm_crypto_shash_finup(struct shash_desc *desc, const u8 *data, unsigned int len, u8 *out)；
描述：完成杂凑计算并将最终摘要输出到指定的缓冲区。
参数：

参数	功能描述
desc	已经填充了数据的操作状态句柄
data	要添加到消息摘要中的输入数据
len	输入数据长度
out	用消息摘要填充的输出缓冲区

返回值：无。

7.4.3.13 计算缓冲区的消息摘要

原型：int ossm_crypto_shash_digest(struct shash_desc *desc, const u8 *data, unsigned int len, u8 *out)；
描述：计算缓冲区的消息摘要。上下文可为任意值。
参数：

参数	功能描述
desc	已经填充了数据的操作状态句柄
data	要添加到消息摘要中的输入数据
len	输入数据长度
out	用消息摘要填充的输出缓冲区

返回值：成功返回0；异常返回<0。

7.4.3.14 计算转换对象缓冲区的消息摘要

原型：int ossm_crypto_shash_tfm_digest(struct ossm_crypto_shash *tfm, const u8 *data, unsigned int len, u8 *out)；

描述：直接接收一个杂凑转换对象缓冲区的消息摘要。上下文可为任意值。注意，这个堆栈分配可能相当大。

参数：

参数	功能描述
tfm	杂凑变换对象
data	要添加到消息摘要中的输入数据
len	输入数据长度
ou	用消息摘要填充的输出缓冲区

返回值：成功返回0；异常返回<0。

7.4.3.15 归零并释放消息摘要句柄

原型：static inline void ossm_crypto_free_shash(struct ossm_crypto_shash *tfm)；

描述：归零并释放消息摘要句柄。成功时分配密码句柄；IS_ERR（）在发生错误时为true，PTR_ERR（）返回错误代码。

参数：

参数	功能描述
tfm	待释放的句柄。如果tfm为NULL或错误指针，则此函数不执行任何操作。

返回值：成功时分配密码句柄，IS_ERR（）在发生错误时为true，PTR_ERR（）返回错误代码。

7.5 密码引擎接口

7.5.1 概述

引擎操作接口包括非对称算法传入引擎、对称算法传入引擎、杂凑算法传入引擎、密码引擎注册/注销密码算法等函数。

可以通过调用引擎接口来注册多个密码资源，以供应用软件选择使用。

表 8 引擎接口

接口名称	说明
ossm_crypto_transfer_akcipher_request_to_engine()	非对称算法传入引擎请求
ossm_crypto_finalize_akcipher_request()	非对称算法请求结束
ossm_crypto_transfer_skcipher_request_to_engine()	对称算法传入引擎请求
ossm_crypto_finalize_skcipher_request()	对称算法请求结束
ossm_crypto_transfer_hash_request_to_engine()	杂凑算法传入引擎请求
ossm_crypto_finalize_hash_request()	杂凑算法请求结束

7.5.2 非对称算法传入密码引擎请求

原型：int ossm_crypto_transfer_akcipher_request_to_engine(struct ossm_crypto_engine *engine, struct akcipher_request *req)；

描述：非对称算法传入。

参数：

参数	功能描述
engine	处理请求的密码引擎
req	非对称加密请求对象

返回值：成功时返回0；出错时返回错误代码。

7.5.3 非对称算法请求结束

原型：void ossm_crypto_finalize_akcipher_request(struct ossm_crypto_engine *engine, struct akcipher_request *req, int err)；

描述：非对称算法结束。

参数：

参数	功能描述
engine	处理请求的密码引擎
req	非对称加密请求对象
err	处理结果状态码

返回值：成功时返回0；出错时返回错误代码。

7.5.4 对称算法传入引擎请求

原型：int ossm_crypto_transfer_skcipher_request_to_engine(struct ossm_crypto_engine *engine, struct skcipher_request *req)；

描述：对称算法传入。

参数：

参数	功能描述
engine	处理请求的密码引擎
req	对称加密请求对象

返回值：成功时返回0；出错时返回错误代码。

7.5.5 对称算法请求结束

原型：void ossm_crypto_finalize_skcipher_request(struct ossm_crypto_engine *engine, struct skcipher_request *req, int err)；

描述：对称算法结束。

参数：

参数	功能描述
engine	处理请求的密码引擎
req	对称加密请求对象
err	处理结果状态码

返回值：成功时返回0；出错时返回错误代码。

7.5.6 杂凑算法传入引擎请求

原型：int ossm_crypto_transfer_hash_request_to_engine(struct ossm_crypto_engine *engine,struct ahash_request *req)；

描述：杂凑算法传入。

参数：

参数	功能描述
engine	处理请求的密码引擎
req	杂凑请求对象

返回值：成功时返回0；出错时返回错误代码。

7.5.7 杂凑算法请求结束

原型：void ossm_crypto_finalize_hash_request(struct ossm_crypto_engine *engine, struct ahash_request *req, int err)；

描述：杂凑算法结束。

参数：

参数	功能描述

engine	处理请求的密码引擎
req	杂凑请求对象
err	处理结果状态码

返回值：成功时返回0；出错时返回错误代码。

8 OS内核态商密资源挂接接口

8.1 概述

OS内核态商密资源挂接接口（南向接口），可扩展或替换商用密码子系统内置的密码算法，挂接新增的商密资源。挂接的商密资源可以是软件密码模块，也可以密码硬件模块，比如CPU内置硬件密码模块等。商密资源厂商通过调用本章定义的商密资源挂接接口来开发密码资源插件。

内核态应用程序可按照本文件7.5密码引擎操作接口按需使用密码资源。

8.2 新增密码算法接口

8.2.1 概述

新增密码算法（资源）注册接口包括对称算法注册及杂凑算法注册函数。

表9 新增密码算法（资源）注册接口

接口名称	说明
ossm_crypto_engine_register_akcipher（）	密码引擎注册非对称算法
ossm_crypto_engine_unregister_akcipher（）	密码引擎取消注册的非对称算法
ossm_crypto_engine_register_skcipher（）	密码引擎注册对称算法
ossm_crypto_engine_unregister_skcipher（）	密码引擎取消注册的对称算法
ossm_crypto_engine_register_skciphers（）	密码引擎注册多个对称算法
ossm_crypto_engine_unregister_skciphers（）	密码引擎取消注册的多个对称算法
ossm_crypto_engine_register_ahash（）	密码引擎注册杂凑算法
ossm_crypto_engine_unregister_ahash（）	密码引擎取消注册的杂凑算法
ossm_crypto_engine_register_ahashes（）	密码引擎注册多个杂凑算法
ossm_crypto_engine_unregister_ahashes（）	密码引擎取消注册的多个杂凑算法

8.2.2 密码引擎注册非对称算法

原型：int ossm_crypto_engine_register_akcipher(struct akcipher_engine_alg *alg)；
描述：密码引擎注册非对称算法。
参数：

参数	功能描述
alg	非对称加密算法引擎描述符结构体

返回值：成功时返回0；出错时返回错误代码。

8.2.3 密码引擎取消注册的非对称算法

原型：void ossm_crypto_engine_unregister_akcipher(struct akcipher_engine_alg *alg)；
描述：密码引擎取消注册的非对称算法。
参数：

参数	功能描述
alg	非对称加密算法引擎描述符结构体

返回值：成功时返回0；出错时返回错误代码。

8.2.4 密码引擎注册对称算法

原型：int ossm_crypto_engine_register_skcipher(struct skcipher_engine_alg *alg)；

描述：密码引擎注册对称算法。
参数：

参数	功能描述
alg	对称加密算法引擎描述符结构体

返回值：成功时返回0；出错时返回错误代码。

8.2.5 密码引擎取消注册的对称算法

原型：void ossm_crypto_engine_unregister_skcipher(struct skcipher_engine_alg *alg)；
描述：密码引擎取消注册的非对称算法。
参数：

参数	功能描述
alg	对称加密算法引擎描述符结构体

返回值：成功时返回0；出错时返回错误代码。

8.2.6 密码引擎注册多个对称算法

原型：int ossm_crypto_engine_register_skciphers(struct skcipher_engine_alg *algs, int count)；
描述：密码引擎注册非对称算法。
参数：

参数	功能描述
alg	对称加密算法结构体
count	数量

返回值：成功时返回0；出错时返回错误代码。

8.2.7 密码引擎取消注册的多个对称算法

原型：void ossm_crypto_engine_unregister_skciphers(struct skcipher_engine_alg *algs, int count)；
描述：密码引擎取消注册的非对称算法。
参数：

参数	功能描述
alg	对称加密算法结构体
count	数量

返回值：成功时返回0；出错时返回错误代码。

8.2.8 密码引擎注册杂凑算法

原型：int ossm_crypto_engine_register_ahash(struct ahash_engine_alg *alg)；
描述：密码引擎注册。
参数：

参数	功能描述
alg	异步杂凑算法引擎描述符结构体

返回值：成功时返回0；出错时返回错误代码。

8.2.9 密码引擎取消注册的杂凑算法

原型：void ossm_crypto_engine_unregister_ahash(struct ahash_engine_alg *alg)；
描述：密码引擎取消注册的杂凑算法。
参数：

参数	功能描述
alg	异步杂凑算法引擎描述符结构体

返回值：成功时返回0；出错时返回错误代码。

8.2.10 密码引擎注册多个杂凑算法

原型：int ossm_crypto_engine_register_ahashes(struct ahash_engine_alg *algs, int count);
描述：密码引擎注册多个杂凑算法。
参数：

参数	功能描述
algs	要注册的杂凑算法结构体
count	数量

返回值：成功时返回0；出错时返回错误代码。

8.2.11 密码引擎取消注册的多个杂凑算法

原型：void ossm_crypto_engine_unregister_ahashes(struct ahash_engine_alg *algs, int count);
描述：密码引擎取消注册的多个杂凑算法。
参数：

参数	功能描述
alg	要取消注册的杂凑算法结构体
count	数量

返回值：成功时返回0；出错时返回错误代码。

8.3 商密资源引擎挂接接口

8.3.1 概述

商密资源引擎挂接接口包括商密资源引擎初始化、商密资源引擎启动/停止等。

表 10 商密资源引擎挂接接口

接口名称	说明
ossm_crypto_engine_alloc_init()	商密资源引擎初始化对象，关联到硬件设备。
ossm_crypto_engine_start()	商密资源引擎启动
ossm_crypto_engine_stop()	商密资源引擎停止
ossm_crypto_engine_exit()	商密资源引擎退出

8.3.2 商密资源引擎初始化

原型：struct ossm_crypto_engine *crypto_engine_alloc_init(struct device *dev, bool rt);
描述：商密资源引擎初始化对象，关联到硬件设备。
参数：

参数	功能描述
dev	关联设备
rt	实时任务标志

返回值：成功时返回0；出错时返回错误代码。

8.3.3 商密资源引擎启动

原型：int ossm_crypto_engine_start(struct ossm_crypto_engine *engine);
描述：商密资源引擎启动。
参数：

参数	功能描述
engine	需要启动的硬件引擎

返回值：成功时返回0；出错时返回错误代码。

8.3.4 商密资源引擎停止

原型：int ossm_crypto_engine_stop(struct ossm_crypto_engine *engine);
描述：商密资源引擎停止。

参数：

参数	功能描述
engine	需要停止的硬件引擎

返回值：成功时返回0；出错时返回错误代码。

8.3.5 商密资源引擎退出

原型：int ossm_crypto_engine_exit(struct ossm_crypto_engine *engine)；
描述：商密资源引擎退出。
参数：

参数	功能描述
engine	需要退出的硬件引擎

返回值：成功时返回0；出错时返回错误代码。

8.4 检验测试类接口

8.4.1 概述

检验测试类接口函数包括杂凑算法自检、对称密码算法自检及非对称算法自测。

表 11 检验测试类接口

接口名称	描述
ossm_hash_testvec	杂凑算法自测
ossm_cipher_testvec	对称密码算法自测
ossm_akcipher_testvec	非对称算法自测

8.4.2 杂凑算法自测

原型：struct ossm_hash_testvec；
描述：杂凑算法测试结构体。
参数：

参数	功能描述
key	密钥指针（无则为空）
plaintext	源数据指针
digest	预期摘要指针
psize	源数据长度
ksize	key长度，以字节为单位（无key时为0）
setkey_error	来自setkey()的预期错误
digest_error	来自digest()的预期错误
fips_skip	在FIPS模式下跳过测试向量

返回值：成功时返回0；出错时返回错误代码。

8.4.3 对称密码算法自测

原型：struct ossm_cipher_testvec；
描述：对称密码测试的结构。
参数：

参数	功能描述
key	密钥指针
klen	以字节为单位的key长度
iv	指向IV的指针。如果为空，则使用全零的IV
iv_out	输出IV的指针（如果适用于该密码）
ptext	明文指针

ctext	密文指针
len	ptext和ctext 的长度（以字节为单位）
wk	测试是否需要 CRYPTO_TFM_REQ_FORBID_WEAK_KEYS（例如，测试需要因弱密钥而失败）
fips_skip	在FIPS模式下跳过测试向量
generates_iv	加密时应忽略给定的IV，并输出iv_out。解密需要iv_out。需要用于AES Keywrap（"kw(sm4)"）
setkey_error	来自setkey()的预期错误。如果已设置，则不会测试加密或解密。
crypt_error	encrypt()和decrypt()的预期错误

返回值：成功时返回0；出错时返回错误代码。

8.4.4 非对称算法自测

原型：struct ossm_akcipher_testvec；
描述：非对称密码测试。
参数：

参数	功能描述
acipher_testvec	描述非对称密码测试的结构
key	密钥指针
keylen	以字节为单位的key长度

返回值：成功时返回0；出错时返回错误代码。。

附 录 A
（规范性）
密码资源接口错误代码

表A.1给出密码资源接口错误代码定义。

表 A.1 密码资源接口错误代码

错误码	描述	典型场景
-EINVAL	Invalid Argument 含义：参数无效或非法。	➢ 密钥长度不符合算法要求。 ➢ 初始化向量（IV）长度不匹配。 ➢ 算法不支持请求的操作（如ECB模式不支持流式加密）。 ➢ 请求的数据对齐不符合硬件加速要求。
-ENOMEM	Out of Memory 含义：内存分配失败。	➢ 无法为加密上下文（crypto_skcipher_ctx）分配内存。 ➢ DMA缓冲区分配失败（硬件加速场景）。
-EAGAIN	Try Again 含义：操作需要重试（通常用于异步操作）。	➢ 异步加密请求队列已满（CRYPTO_TFM_REQ_MAY_BACKLOG 标志启用时）。 ➢ 资源暂时不可用（如硬件加速器忙）。
-EBUSY	Device or Resource Busy 含义：资源被占用，无法立即处理请求。	➢ 加密硬件加速器正在处理其他任务。 ➢ 加密上下文已被锁定。
-ENOKEY	No Key 含义：未设置密钥时尝试执行加密/解密操作。	➢ 调用 ossm_crypto_skcipher_encrypt() 或 ossm_crypto_skcipher_decrypt() 前未调用 ossm_crypto_skcipher_setkey()。
-EBADMSG	Bad Message 含义：数据完整性校验失败。	➢ 密文被篡改或损坏。
-EIO	Input/Output Error 含义：底层硬件或传输错误。	➢ 硬件加密引擎（如CAAM、QAT）的DMA传输失败。 ➢ 加密设备驱动返回不可恢复的错误。
-ENOSYS	Function Not Implemented 含义：请求的操作未实现。	➢ 算法不支持特定模式（如某些硬件加速器不支持 XTS 模式）。 ➢ 调用未实现的回调函数（如init_tfm未定义）。
-EPROTO	Protocol Error 含义：加密协议或状态异常。	➢ 加密上下文状态不一致（如多次调用finup()而未重置）。 ➢ 异步操作时序错误。
-ENODEV	Protocol Error 含义：加密协议或状态异常。	➢ 加密上下文状态不一致如多次调用finup()而未重置）。 ➢ 异步操作时序错误。
-ENODEV	No Such Device 含义：加密设备不存在或未初始化。	➢ 请求的硬件加速器未注册或已卸载。 ➢ 算法驱动未正确加载。
-EFAULT	Bad Address	➢ copy_from_user()失败

	含义：用户空间指针非法	
-ENOENT	No Such Entry 含义：请求的算法未注册	➢ ossm_crypto_alloc_skcipher()失败
-ETIMEDOUT	Timeout 含义：硬件加密操作超时	➢ 引擎驱动可能返回此错误
-EALREADY	Already in Progress 含义：异步操作已提交但未完成。	➢ 异步操作错误。

附 录 B
（资料性）
密码资源接口引擎接口示例

以下C代码是一个Linux内核模块的开头部分，其功能是为TCM硬件设备提供驱动支持，以SM2商用密码算法为例的引擎接口功能。

主要内容包括：

1) 引入必要的 Linux 内核头文件，涵盖模块、PCI 设备、中断处理、DMA 映射以及加密引擎相关功能。
2) 定义驱动名称、DMA 传输大小、中断标志和超时参数等常量。
3) 声明硬件寄存器和支持的算法类型的枚举值。
4) 定义多个数据结构，用于存储设备上下文、请求信息和算法特定的上下文。
5) 声明用于处理不同密码算法请求的函数原型。
6) 实现中断处理函数，用于处理硬件完成的密码操作。
7) 实现引擎核心函数，负责调度和处理各种密码请求。
8) 实现 SM2 算法的具体处理逻辑和相关操作函数。
9) 实现 PCI 驱动的探测和移除函数，用于初始化和清理设备。

注册PCI驱动和各种密码算法，使驱动能够被内核识别和使用。

```
/* SPDX-License-Identifier: GPL-2.0 */
/*
 * TCM卡硬件加速驱动示例，支持SM2算法。for KylinOS 6.6
 */
#include <linux/module.h>
#include <linux/pci.h>
#include <linux/interrupt.h>
#include <linux/scatterlist.h>
#include <linux/dma-mapping.h>
#include <linux/crypto/engine.h>
#include <crypto/internal/akcipher.h>
#include <crypto/internal/skcipher.h>
#include <crypto/internal/hash.h>
#include <crypto/scatterwalk.h>
#include <crypto/sm2.h>

// 驱动名称定义
#define DRV_NAME "tcm_crypto"
// DMA突发传输大小
#define DMA_BURST_SIZE 256
// 中断完成标志
#define TCM_IRQ_COMPLETION 0x1
// 硬件操作超时时间（1毫秒）
#define TCM_TIMEOUT 1000

/* 硬件寄存器定义枚举，指定TCM设备各寄存器的偏移地址 */
enum tcm_registers {
    TCM_CTRL            = 0x00,     // 控制寄存器
```

```
    TCM_STATUS      = 0x04,     // 状态寄存器
    TCM_IRQ_STAT    = 0x08,     // 中断状态寄存器
    TCM_DMA_SRC     = 0x0C,     // DMA源地址寄存器
    TCM_DMA_DST     = 0x10,     // DMA目标地址寄存器
    TCM_DATA_LEN    = 0x14,     // 数据长度寄存器
    TCM_KEY_SLOT    = 0x18,     // 密钥槽位寄存器
    TCM_OP_MODE     = 0x1C,     // 操作模式寄存器
    TCM_IV_REG      = 0x20      // 初始化向量寄存器
};

/* 支持的算法类型枚举，用于标识当前操作使用的SM2算法 */
enum tcm_algorithms {
    ALG_SM2 = 0x01,  // SM2非对称密码算法
};

/* 密码引擎上下文结构，存储引擎实例与TCM设备的关联信息 */
struct tcm_engine_ctx {
    struct ossm_crypto_engine *engine;  // 密码引擎句柄
    struct tcm_device *tcm;             // TCM设备指针
    int key_slot;                       // 密钥槽位编号
};

/* 请求信息结构，用于管理异步密码请求的上下文数据 */
struct tcm_request_info {
    struct list_head list;              // 链表节点，用于挂起请求队列
    struct ossm_crypto_async_request *req;// 密码异步请求指针
    int src_nents;                      // 源散射列表条目数
    int dst_nents;                      // 目标散射列表条目数
    int iv_nents;                       // 初始化向量散射列表条目数
    struct scatterlist result_sg;    // 结果数据的散射列表项
    struct scatterlist iv_sg;        // 初始化向量的散射列表项
};

/* TCM设备主结构，包含硬件资源和驱动状态信息 */
struct tcm_device {
    struct pci_dev *pdev;               // PCI设备指针
    void __iomem *base;                 // 硬件寄存器基地址（I/O映射）
    struct ossm_crypto_engine *engine;    // 密码引擎实例
    spinlock_t lock;                    // 自旋锁，保护共享资源
    int irq;                            // 中断号
    struct dma_pool *dma_pool;          // DMA内存池
    struct list_head pending;           // 待处理请求队列
    int dma_alignment;                  // DMA对齐要求
};


/* 函数前置声明，定义各算法处理函数的接口 */
static int tcm_sm2_process(struct ossm_crypto_async_request *areq);

/*----------------------- 中断处理模块 -----------------------*/
/**
```

```
 * 中断处理函数，处理TCM设备的完成中断
 * @param irq 中断号
 * @param dev_id 设备上下文指针（TCM设备结构）
 * @return 中断处理结果（IRQ_HANDLED表示已处理）
 */
static irqreturn_t tcm_interrupt(int irq, void *dev_id)
{
    struct tcm_device *tcm = dev_id;
    struct ossm_crypto_async_request *req, *tmp;
    unsigned long flags;
    u32 irq_status;
    LIST_HEAD(complete_list);

    // 读取中断状态寄存器
    irq_status = ioread32(tcm->base + TCM_IRQ_STAT);
    // 检查是否为完成中断
    if (!(irq_status & TCM_IRQ_COMPLETION))
        return IRQ_NONE;

    // 清除中断标志
    iowrite32(irq_status, tcm->base + TCM_IRQ_STAT);

    // 加锁保护待处理请求队列
    spin_lock_irqsave(&tcm->lock, flags);
    // 将待处理队列转移到完成列表
    list_splice_init(&tcm->pending, &complete_list);
    spin_unlock_irqrestore(&tcm->lock, flags);

    // 处理所有完成的请求
    list_for_each_entry_safe(req, tmp, &complete_list, list) {
        struct tcm_request_info *info = container_of(req, struct tcm_request_info, req);
        // 根据算法类型释放DMA映射并完成请求
        if (req->tfm->__crt_alg->cra_type == &ossm_crypto_akcipher_type) {
            struct akcipher_request *akreq = akcipher_request_cast(req);
            // 释放源数据和目标数据DMA映射
            dma_unmap_sg(&tcm->pdev->dev, akreq->src, info->src_nents, DMA_TO_DEVICE);
            dma_unmap_sg(&tcm->pdev->dev, akreq->dst, info->dst_nents,
DMA_FROM_DEVICE);
        }
        // 从列表中删除请求并完成密码操作
        list_del(&req->list);
        ossm_crypto_request_complete(req, 0);
        // 释放请求信息结构内存
        kfree(info);
    }
    return IRQ_HANDLED;
}

/*------------------------ 引擎核心处理模块 ------------------------*/
/**
 * 处理单个密码请求，根据算法类型分发给对应处理函数
 * @param engine 密码引擎实例
 * @param req 密码异步请求指针
```

```
 * @return 操作结果（-EINPROGRESS表示异步处理中）
 */
static int tcm_do_one_request(struct ossm_crypto_engine *engine,
                              struct ossm_crypto_async_request *req)
{
    struct tcm_device *tcm = ossm_crypto_engine_ctx(engine);
    struct tcm_request_info *info;
    int ret = -EINVAL;

    // 分配请求信息结构内存
    info = kzalloc(sizeof(*info), GFP_ATOMIC);
    if (!info)
        return -ENOMEM;

    // 初始化链表节点并关联请求
    INIT_LIST_HEAD(&info->list);
    info->req = req;

    // 根据算法类型分发处理
    if (req->tfm->__crt_alg->cra_type == &ossm_crypto_akcipher_type) {
        ret = tcm_sm2_process(req);
    } else {
        dev_err(&tcm->pdev->dev, "不支持的算法类型\n");
        kfree(info);
        return -EINVAL;
    }

    // 处理异步请求状态
    if (ret == -EINPROGRESS) {
        unsigned long flags;
        // 加锁并将请求加入待处理队列
        spin_lock_irqsave(&tcm->lock, flags);
        list_add_tail(&info->list, &tcm->pending);
        spin_unlock_irqrestore(&tcm->lock, flags);
    } else {
        // 处理完成后释放资源
        kfree(info);
        ossm_crypto_request_complete(req, ret);
    }
    return ret;
}

/*---------------------- SM2 非对称实现 ------------------------*/
/**
 * 处理SM2非对称密码请求
 * @param areq 异步密码请求
 * @return -EINPROGRESS表示操作正在进行中
 */
static int tcm_sm2_process(struct ossm_crypto_async_request *areq)
{
    struct akcipher_request *req = akcipher_request_cast(areq);
    struct tcm_engine_ctx *ctx = ossm_crypto_engine_ctx(areq->engine);
    struct tcm_device *tcm = ctx->tcm;
    struct tcm_request_info *info = container_of(areq, struct tcm_request_info, req);
```

```
u32 op_mode;
int src_nents, dst_nents;

// 根据请求操作类型设置相应的操作模式
switch (req->op) {
case AKCIPHER_REQUEST_SIGN:
        op_mode = 0x10000000;   // 签名操作
        break;
case AKCIPHER_REQUEST_VERIFY:
        op_mode = 0x20000000;   // 验证操作
        break;
default:
        dev_err(&tcm->pdev->dev, "无效的SM2操作类型: %d\n", req->op);
        return -EINVAL;
}

// 配置操作模式寄存器
iowrite32(ALG_SM2 | op_mode, tcm->base + TCM_OP_MODE);

// 将源数据和目标数据映射到DMA地址空间
src_nents = dma_map_sg(&tcm->pdev->dev, req->src, sg_nents(req->src), DMA_TO_DEVICE);
dst_nents = dma_map_sg(&tcm->pdev->dev, req->dst, sg_nents(req->dst),
                DMA_FROM_DEVICE);

// 检查映射是否成功
if (!src_nents || !dst_nents) {
        if (src_nents)
                dma_unmap_sg(&tcm->pdev->dev, req->src, src_nents, DMA_TO_DEVICE);
        if (dst_nents)
                dma_unmap_sg(&tcm->pdev->dev, req->dst, dst_nents, DMA_FROM_DEVICE);
        return -ENOMEM;
}

// 保存散射列表条目数
info->src_nents = src_nents;
info->dst_nents = dst_nents;

// 配置源数据DMA传输
struct scatterlist *sg;
int i;
for_each_sg(req->src, sg, src_nents, i) {
        iowrite32(sg_dma_address(sg), tcm->base + TCM_DMA_SRC);
        iowrite32(sg_dma_len(sg), tcm->base + TCM_DATA_LEN);
}

// 配置目标数据DMA传输
for_each_sg(req->dst, sg, dst_nents, i) {
        iowrite32(sg_dma_address(sg), tcm->base + TCM_DMA_DST);
}

// 启动SM2操作
iowrite32(ALG_SM2 | 0x80000000, tcm->base + TCM_CTRL);

// 返回异步处理状态
```

```
    return -EINPROGRESS;
}

/**
 * 初始化SM2转换模块
 * @param tfm 非对称密码转换模块
 * @return 0表示成功
 */
static int tcm_sm2_init_tfm(struct ossm_crypto_akcipher *tfm)
{
    struct tcm_engine_ctx *ctx = akcipher_tfm_ctx(tfm);

    // 获取TCM设备上下文
    ctx->tcm = ossm_crypto_engine_ctx(ossm_crypto_akcipher_engine(tfm));
    if (!ctx->tcm) {
        pr_err("无法获取TCM设备上下文\n");
        return -ENODEV;
    }

    // 初始化密钥槽位
    ctx->key_slot = -1;
    return 0;
}

/**
 * 清理SM2转换模块资源
 * @param tfm 非对称密码转换模块
 */
static void tcm_sm2_exit_tfm(struct ossm_crypto_akcipher *tfm)
{
    struct tcm_engine_ctx *ctx = akcipher_tfm_ctx(tfm);
    struct tcm_device *tcm = ctx->tcm;

    // 如果有分配的密钥槽位，释放它
    if (ctx->key_slot >= 0) {
        int timeout = TCM_TIMEOUT;

        // 等待硬件空闲
        while ((ioread32(tcm->base + TCM_STATUS) & 0x80000000) && (timeout-- > 0))
            udelay(1);

        if (timeout <= 0)
            dev_err(&tcm->pdev->dev, "等待硬件空闲超时\n");
        else
            // 释放密钥槽位
            iowrite32(ALG_SM2 | (ctx->key_slot << 16), tcm->base + TCM_KEY_SLOT);

        ctx->key_slot = -1;
    }
}

/**
 * SM2签名操作
 * @param req 非对称密码请求
```

```
 * @return -EINPROGRESS表示操作正在进行中
 */
static int tcm_sm2_sign(struct akcipher_request *req)
{
	return tcm_sm2_process(&req->base);
}

/**
 * SM2验证操作
 * @param req  非对称密码请求
 * @return -EINPROGRESS表示操作正在进行中
 */
static int tcm_sm2_verify(struct akcipher_request *req)
{
	return tcm_sm2_process(&req->base);
}

/**
 *  设置SM2私钥
 * @param tfm  非对称密码转换模块
 * @param key  私钥数据
 * @param keylen  私钥长度
 * @return 0表示成功
 */
static int tcm_sm2_set_priv_key(struct ossm_crypto_akcipher *tfm, const void *key, unsigned int keylen)
{
	struct tcm_engine_ctx *ctx = akcipher_tfm_ctx(tfm);
	struct scatterlist sg;
	int ret;

	// 初始化散射列表
	sg_init_one(&sg, key, keylen);

	// 将私钥映射到DMA地址空间
	ret = dma_map_sg(&ctx->tcm->pdev->dev, &sg, 1, DMA_TO_DEVICE);
	if (ret <= 0)
		return -ENOMEM;

	// 配置控制寄存器，准备加载私钥
	iowrite32(ALG_SM2 | 0x40000000, ctx->tcm->base + TCM_CTRL);

	// 设置DMA源地址和数据长度
	iowrite32(sg_dma_address(&sg), ctx->tcm->base + TCM_DMA_SRC);
	iowrite32(keylen, ctx->tcm->base + TCM_DATA_LEN);

	// 解除DMA映射
	dma_unmap_sg(&ctx->tcm->pdev->dev, &sg, 1, DMA_TO_DEVICE);

	return 0;
}

/**
 *  获取SM2最大处理数据大小
 * @param tfm  非对称密码转换模块
```

```
 * @return  最大数据大小（字节）
 */
static unsigned int tcm_sm2_max_size(struct ossm_crypto_akcipher *tfm)
{
	return 512;
}

// SM2算法驱动注册结构
static struct akcipher_alg sm2_alg = {
	.sign                = tcm_sm2_sign,              // 签名回调函数
	.verify              = tcm_sm2_verify,            // 验证回调函数
	.set_priv_key        = tcm_sm2_set_priv_key, // 设置私钥回调函数
	.max_size            = tcm_sm2_max_size,          // 最大数据大小
	.init                = tcm_sm2_init_tfm,          // 初始化回调函数
	.exit                = tcm_sm2_exit_tfm,          // 清理回调函数
	.base = {
		.cra_name           = "sm2",                      // 算法名称
		.cra_driver_name= "sm2-tcm",                  // 驱动名称
		.cra_priority       = 400,                        // 驱动优先级
		.cra_flags          = CRYPTO_ALG_ASYNC,           // 异步算法标志
		.cra_ctxsize        = sizeof(struct tcm_engine_ctx), // 上下文大小
		.cra_module         = THIS_MODULE,                // 模块指针
	},
};

/*------------------------ PCI驱动核心 ------------------------*/
// 支持的PCI设备ID列表
static const struct pci_device_id tcm_ids[] = {
	{ PCI_DEVICE(0x1a2b, 0x3c4d) }, // 实际硬件VID/PID
	{ 0, }                                  // 列表结束标记
};
MODULE_DEVICE_TABLE(pci, tcm_ids);

/**
 * PCI设备探测函数，初始化TCM驱动
 * @param pdev PCI设备结构
 * @param id  匹配的设备ID
 * @return 0表示成功，错误码表示失败
 */
static int tcm_probe(struct pci_dev *pdev, const struct pci_device_id *id)
{
	struct tcm_device *tcm;
	int ret;

	// 启用PCI设备
	ret = pci_enable_device(pdev);
	if (ret)
		return ret;

	// 设置为总线主设备，允许DMA
	pci_set_master(pdev);
```

```
// 分配并初始化设备结构
tcm = devm_kzalloc(&pdev->dev, sizeof(*tcm), GFP_KERNEL);
if (!tcm) {
	ret = -ENOMEM;
	goto err_disable_pci;
}

// 映射PCI设备的IO空间
tcm->base = pci_iomap(pdev, 0, 0);
if (!tcm->base) {
	dev_err(&pdev->dev, "无法映射IO空间\n");
	ret = -ENODEV;
	goto err_disable_pci;
}

// 设置DMA掩码，优先使用64位地址，否则使用32位
if (dma_set_mask_and_coherent(&pdev->dev, DMA_BIT_MASK(64))) {
	if (dma_set_mask_and_coherent(&pdev->dev, DMA_BIT_MASK(32))) {
		dev_err(&pdev->dev, "无法设置DMA掩码\n");
		ret = -ENODEV;
		goto err_iounmap;
	}
}

// 分配并初始化密码引擎
tcm->engine = ossm_crypto_engine_alloc_init(&pdev->dev, true);
if (!tcm->engine) {
	dev_err(&pdev->dev, "无法初始化密码引擎\n");
	ret = -ENOMEM;
	goto err_iounmap;
}

// 设置引擎上下文
ossm_crypto_engine_set_ctx(tcm->engine, tcm);

// 启动密码引擎
ret = ossm_crypto_engine_start(tcm->engine);
if (ret) {
	dev_err(&pdev->dev, "无法启动密码引擎\n");
	goto err_engine_exit;
}

// 注册中断处理函数
ret = request_irq(pdev->irq, tcm_interrupt, IRQF_SHARED, DRV_NAME, tcm);
if (ret) {
	dev_err(&pdev->dev, "无法注册中断处理程序\n");
	goto err_engine_stop;
}

 // 注册SM2算法
ret = ossm_crypto_register_akcipher(&sm2_alg);
if (ret) {
	dev_err(&pdev->dev, "注册SM2算法失败\n");
	goto err_free_irq;
```

```
    }

    // 初始化自旋锁和待处理请求队列
    spin_lock_init(&tcm->lock);
    INIT_LIST_HEAD(&tcm->pending);

    // 保存设备上下文
    pci_set_drvdata(pdev, tcm);

    return 0;

// 错误处理标签：释放中断
err_free_irq:
    free_irq(pdev->irq, tcm);

// 错误处理标签：停止密码引擎
err_engine_stop:
    ossm_crypto_engine_stop(tcm->engine);

// 错误处理标签：退出密码引擎
err_engine_exit:
    ossm_crypto_engine_exit(tcm->engine);

// 错误处理标签：解除IO映射
err_iounmap:
    pci_iounmap(pdev, tcm->base);

// 错误处理标签：禁用PCI设备
err_disable_pci:
    pci_disable_device(pdev);
    return ret;
}

/**
 * PCI设备移除函数，清理资源
 * @param pdev PCI设备结构
 */
static void tcm_remove(struct pci_dev *pdev)
{
    struct tcm_device *tcm = pci_get_drvdata(pdev);

    // 卸载已注册的算法
    ossm_crypto_unregister_akcipher(&sm2_alg);

    // 释放中断
    free_irq(pdev->irq, tcm);

    // 停止并退出密码引擎
    ossm_crypto_engine_stop(tcm->engine);
    ossm_crypto_engine_exit(tcm->engine);

    // 解除IO映射并禁用PCI设备
    pci_iounmap(pdev, tcm->base);
    pci_disable_device(pdev);
```

```
}

// PCI驱动注册结构
static struct pci_driver tcm_driver = {
    .name       = DRV_NAME,              // 驱动名称
    .id_table   = tcm_ids,               // 支持的设备ID表
    .probe      = tcm_probe,             // 探测函数
    .remove     = tcm_remove,            // 移除函数
};
```

参 考 文 献

[1] https://www.kernel.org/doc/html/latest/crypto/index.html
[2] Linux kernel 6.6源码