Android Keystore 与 Keymaster 介绍

文档版本	V1.5
发布日期	2020-08-10

紫光展锐科技有限公司

紫光展锐科技有限公司

前 言

概述

本文档详细地描述了 Android Keystore 与 Keymaster 功能、结构及原理。

读者对象

本文档主要适用于与 Android Keystore 与 Keymaster 相关的开发和测试人员。

缩略语

缩略语	英文全名	中文解释
3DES	Triple Data Encryption Algorithm	三重数据加密算法
AES	Advanced Encryption Standard	高级加密标准
CA	Client Application	客户应用程序
CTS	Compatibility Test Suite	兼容性测试套件
ECDSA	Elliptic Curve Digital Signature Algorithm	圆锥曲线数字签名算法
HAL	Hardware Abstraction Level	硬件抽象层
HIDL	Hardware Interface Description Language	硬件接口描述语言
HMAC	Hash-based Message Authentication Code	哈希算法消息验证码
RSA	Rivest-Shamir-Adleman	一种非对称加密算法
SoC	System on Chip	系统芯片
TA	Trusted Application	可信应用程序
TEE	Trusted Execution Environment	可信执行环境
VTS	Vendor Test Suite	供应商测试套件

符号约定

在本文中可能出现下列标志，它所代表的含义如下。

符号	说明
说明	用于突出重要/关键信息、补充信息和小窍门等。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害。

变更信息

文档版本	发布日期	修改说明
V1.0	2019-06-11	初始版本。
V1.1	2019-06-19	修复常见问题两处版本描述错误。
V1.2	2019-06-20	适配新文档模板。
V1.3	2019-06-20	修改 Platform 信息。
V1.4	2020-04-02	更新模板及内容优化。
V1.5	2020-08-10	• 增加 Android 11.0 内容，如 1.5、1.6、4.1.22~4.1.27 和 5.2 章节。 • 文档名称从《Android Keystore&Keymaster 介绍》更新为《Android Keystore 与 Keymaster 介绍》。 • 调整文档结构，优化文档描述。

关键字

Keystore、Keymaster、HIDL、Keymaster HAL、TEE Keymaster、CTS、VTS。

目 录

1 概述

借助系统芯片(SoC)中提供的可信执行环境(TEE)，Android 设备可以为 Android 操作系统、平台服务甚至是第三方应用提供由硬件支持的强大安全服务。

在 Android 6.0 之前的版本中，Android 已有一个非常简单的由硬件支持的加密服务 API（由 0.2 和 0.3 版的 Keymaster 硬件抽象层提供）。该 Keystore 能够提供数字签名和验证操作，以及不对称签名密钥对的生成和导入操作。该 API 在许多设备上都已实现，但有许多安全目标无法只通过一个签名 API 来轻松达成。

1.1 Android 6.0

在 Android 6.0 中，Keystore 在之前 Keystore API 的基础上进行了扩展，能够提供更广泛的功能。不仅增加了对称加密基元（AES 和 HMAC），还增加了针对由硬件支持的密钥的访问控制系统。访问控制在密钥生成期间指定，并会在密钥的整个生命周期内被强制执行。可以将密钥限定为仅在用户通过身份验证后才可使用，并且只能用于指定的目的或只有在具有指定的加密参数时才可使用。

除了扩大加密基元的范围外，Android 6.0 中的 Keystore 还增加了以下内容：

- 一个使用控制方案，用于限制密钥的使用，并降低因滥用密钥而损害安全性的风险。
- 一个访问控制方案，用于限定只有指定的用户和客户端能够使用相应密钥，并且只能在规定的时间范围内使用。

1.2 Android 7.0

- 在 Android 7.0 中，Keymaster 2 增加了对密钥认证和版本绑定的支持。
- 密钥认证提供公钥证书，这些证书中包含密钥及其访问控制的详细描述，以使密钥存在于安全硬件中并使其配置可以远程验证。
- 版本绑定将密钥绑定至操作系统和补丁程序级别版本。这样可确保在旧版系统或 TEE 软件中发现漏洞的攻击者无法将设备回滚到易受攻击的版本，也无法使用在较新版本中创建的密钥。此外，在已经升级到更新的版本或补丁程序级别的设备上使用指定版本和补丁程序级别的密钥时，需要先升级该密钥才能使用，因为该密钥的旧版本已失效。当设备升级时，密钥会随着设备一起“升级”，但是将设备恢复到任何一个旧版本都会导致密钥无法使用。

1.3 Android 8.0

- 在 Android 8.0 中，Keymaster 3 从旧式 C 结构硬件抽象层(HAL)转换到从新的硬件接口定义语言(HIDL)的定义生成的 C++ HAL 接口。在此变更过程中，很多参数类型发生了变化，尽管这些类型和方法与旧的类型和 HAL 结构体方法具有一对一的对应关系。

- 除了此接口修订之外，Android 8.0 还扩展了 Keymaster 2 的认证功能，以支持 ID 认证。ID 认证提供了一种受限且可选的机制来严格认证硬件标识符，例如设备序列号、产品名称和手机 ID(IMEI/MEID)。要实现此附加功能，需更改 ASN.1 认证架构以添加 ID 认证。Keymaster 实现需要通过某种安全方式来检索相关的数据项，还需要定义一种安全永久地停用该功能的机制。

1.4 Android 9.0

在 Android 9.0 中，更新包括：

- 更新到 Keymaster 4（Strongbox 支持）。
- 对嵌入式安全元件的支持。
- 对安全密钥导入的支持。
- 对 3DES 加密的支持。
- 更改了版本绑定，以便 boot.img 和 system.img 分别设置版本以允许独立更新。

1.5 Android 10.0

在 Android 10.0 中，未更新 Keymaster 版本。

1.6 Android 11.0

在 Android 11.0 中，Keymaster 更新到 4.1。Keymaster 4.1 是 Keymaster 4.0 的一个小扩展，内容包括：

- 部分硬件对 UNLOCKED_DEVICE_REQUIRED 密钥的强制执行。
- 设备唯一认证。
- 早期启动只能通过密钥。
- 优化了清理客户端死机时没有完成或中止的操作步骤。
- attestKey()必须生成 keymasterVersion 4.1 的认证。
- 原始编号中的一个疏忽没有为副版本号留出空间，因此从 Keymaster 4.1 开始，版本将被编号为 major_version * 10 +副版本号。
- 添加新的能被证实的标记以及再次对认证模式进行轻微的更改，因此 attestationVersion 必须为 4。

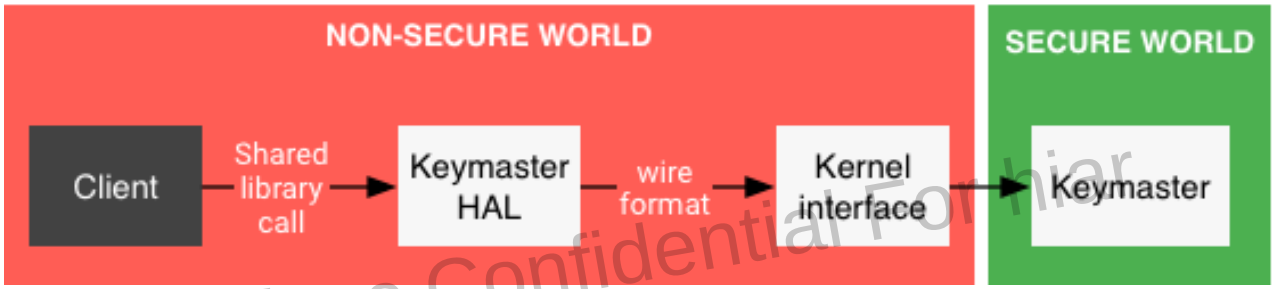
2 架构

2.1 框图

Android Keystore API 和底层 Keymaster HAL 提供了一套基本的但足以满足需求的加密基元，以便使用访问受控且由硬件支持的密钥实现相关协议。

Keymaster HAL 是由原始设备制造商(OEM)提供的动态加载库，Keystore 服务使用它来提供由硬件支持的加密服务。为了确保安全性，HAL 实现不会在用户空间（甚至是内核空间）中执行任何敏感操作。敏感操作会被分配给通过某个内核接口连接的安全处理器。最终的架构如图 2-1 所示：

图2-1 访问 Keymaster



在 Android 设备中，Keymaster HAL 的“客户端”包含多个层（例如，应用、框架、Keystore 守护进程）。这意味着，所介绍的 Keymaster HAL API 为底层 API，供平台内部组件使用，不面向应用开发者提供。

Keymaster HAL 的目的不是实现安全敏感型算法，而只是对发送到安全域的请求进行编排和解排。传输格式是由实现定义的。

2.2 Keystore 组件

- AndroidKeystore

  是供应用访问 Keystore 功能的 Android Framework API 和组件。它是作为 Java Cryptography Architecture API 的扩展程序实现的，包含在应用本身的进程空间中运行的 Java 代码。AndroidKeystore 通过将与 Keystore 行为有关的应用请求转发到 Keystore 守护进程来执行这些请求。

- Keystore 守护进程

  是 Android 系统中的一个守护进程，该进程通过 Binder API 提供对所有 Keystore 功能的访问权限。Keystore 守护进程负责存储“密钥 Blob”。密钥 Blob 中包含已加密的实际密钥材料，因此 Keystore 可以存储这些材料，但无法使用或显示这些材料。

- keymasterd

  是一个 HIDL 服务器，可提供对 Keymaster TA 的访问权限。该组件位于 /vendor/bin/hw/android.hardware.keymaster@3.0-service（Keymaster 3）。

- Keymaster TA（可信应用）

  是在安全环境（大多数情况为 ARM SoC 上的 TrustZone）中运行的软件，通常与 trusty OS 打包在一起，不可以 DynamicTA 方式进行加载。它可提供所有安全的 Keystore 操作，能够访问原始密钥材料，在密钥上验证所有访问控制条件等等。

- Keymaster CA（NonSecure 接口）

  NonSecure 环境提供访问 Keymaster TA 接口，将来自 NonSecure 环境的请求通过 NonSecure Kernel 接口、Trusty OS 接口层层转发至 Keymaster TA。该组件位于 vendor/lib64/hw/keystore.sprdtrusty.so 或者 vendor/lib/hw/keystore.sprdtrusty.so。

- LockSettingsService

  是负责用户身份验证（包括密码和指纹）的 Android 系统组件。它不是 Keystore 的一部分却与其相关，因为很多 Keystore 密钥操作都需要对用户进行身份验证。LockSettingsService 与 Gatekeeper TA 和 Fingerprint TA 进行交互以获取身份验证令牌，并将其提供给 Keystore 守护进程，这些令牌最终将由 Keymaster TA 应用使用。

- Gatekeeper TA（可信应用）

  是在安全环境中运行的另一个组件，它负责验证用户密码并生成身份验证令牌（用于向 Keymaster TA 证明已在特定时间点完成对特定用户的身份验证）。

- Fingerprint TA（可信应用）

  是在安全环境中运行的另一个组件，它负责验证用户指纹并生成身份验证令牌（用于向 Keymaster TA 证明已在特定时间点完成对特定用户的身份验证）。

3 HIDL 概览

硬件接口定义语言(HIDL)提供了一种独立于实现语言的机制来指定硬件接口。HIDL 工具目前支持生成 C++和 Java 接口。本文档仅讨论 C++表示法。

HIDL 接口由一组方法组成，表示如下：

```
methodName(INPUT ARGUMENTS) generates (RESULT ARGUMENTS);
```

有很多不同的预定义类型，而 HAL 可以定义新的枚举和结构类型。要详细了解 HIDL，请参考“hardware/interfaces/keymaster/3.0/IKeymasterDevice.hal”文件。

- 下面显示了 Keymaster 3 IKeymasterDevice.hal 中的一个示例方法：

```
generateKey(vec<KeyParameter> keyParams)
        generates(ErrorCode error, vec<uint8_t> keyBlob,
                KeyCharacteristics keyCharacteristics);
```

- 这相当于 keymaster2 HAL 中的以下方法：

```
keymaster_error_t (*generate_key)(
        const struct keymaster2_device* dev,
        const keymaster_key_param_set_t* params,
        keymaster_key_blob_t* key_blob,
        keymaster_key_characteristics_t* characteristics);
```


在 HIDL 版本中，由 dev 参数采用隐式形式，因此已被移除。params 参数不再是包含引用了一组 key_parameter_t 对象的指针的结构体，而是包含 KeyParameter 对象的 vec（矢量）。返回值在“generates”子句中列出，其中包含密钥 Blob 的 uint8_t 值的矢量。

由 HIDL 编译器生成的 C++虚拟方法为：

```
Return<void> generateKey(const hidl_vec<KeyParameter>& keyParams,
                        generateKey_cb _hidl_cb) override;
```

其中，generate_cb 是一个函数指针，定义如下：

```
std::function<void(ErrorCode error, const hidl_vec<uint8_t>& keyBlob,
                const KeyCharacteristics& keyCharacteristics)>
```

也就是说，generate_cb 是一个将接受 generate 子句中列出的返回值的函数。HAL 实现类会覆盖此 generateKey 方法并调用 generate_cb 函数指针，以将操作结果返回给调用程序。

说明

该函数指针调用是同步的。调用程序调用 generateKey，同时 generateKey 调用所提供的函数指针，该指针在完成执行操作后将控件返回 generateKey 实现，而后该实现又返回到调用程序。

要查看详细示例，请参阅 hardware/interfaces/keymaster/3.0/default/KeymasterDevice.cpp 中的默认实现。该默认实现可向后兼容采用旧式 keymaster 0、keymaster 1 或 keymaster 2 HAL 的设备。

4 Keymaster 功能

本章对 Keymaster 硬件抽象层(HAL)实现进行详细介绍。其中介绍了 API 中的每个函数、提供函数的 Keymaster 版本以及函数的默认实现方法。

4.1 主要功能

4.1.1 getHardwareFeatures

getHardwareFeatures 方法向客户端披露了底层安全硬件的一些重要特征。该方法不需要任何参数，且返回四个值（都是布尔值）：

- 如果密钥始终存储在安全硬件（TEE 等）中，则 isSecure 为 true。
- 如果硬件支持采用 NIST 曲线（P-224、P-256、P-384 和 P-521）的椭圆曲线加密，则 supportsEllipticCurve 为 true。
- 如果硬件支持对称加密（包括 AES 和 HMAC），则 supportsSymmetricCryptography 为 true。
- 如果硬件支持生成使用注入安全环境中的密钥进行签名的 Keymaster 公钥认证证书，则 supportsAttestation 为 true。

该方法仅可能返回以下错误代码：ErrorCode::OK、ErrorCode::KEYMASTER_NOT_CONFIGURED 或指示无法与安全硬件通信的错误代码之一。

4.1.2 configure

该函数在 Keymaster 2 中引入，并在 Keymaster 3 中被弃用，因为这些信息已在系统属性文件中提供，制造商实现会在启动期间读取这些文件。

配置 Keymaster：

- 该方法在打开设备之后、使用设备之前被调用一次。
- 可使用此方法向 Keymaster 提供 KM_TAG_OS_VERSION 和 KM_TAG_OS_PATCHLEVEL。
- 在该方法被调用之前，所有其他方法会返回 KM_ERROR_KEYMASTER_NOT_CONFIGURED。
- 该方法提供的值仅在每次启动时被 Keymaster 接受一次。之后的调用都会返回 KM_ERROR_OK，但不执行任何操作。

如果 Keymaster 实现在安全的硬件中进行，且提供的操作系统版本和补丁程序级别的值与引导加载程序向安全硬件提供的值不匹配（或者引导加载程序未提供任何值），则该方法会返回 KM_ERROR_INVALID_ARGUMENT，所有其他方法仍然返回 KM_ERROR_KEYMASTER_NOT_CONFIGURED。

4.1.3 addRngEntropy

该函数在 Keymaster 1 中引入（名为 add_rng_entropy），并在 Keymaster 3 中进行了重命名。

该函数将调用程序提供的熵添加到 Keymaster 1 实现生成随机数（用作密钥、IV 等）所用的池中。

Keymaster 1 实现需要将收到的熵安全地混合到所使用的池中，该池中还必须包含由硬件随机数生成器在内部生成的熵。对混合操作的处理应该实现：即使攻击者能够完全控制 addRngEntropy 提供的数位或硬件生成的数位（但不能同时控制这两者），他们在预测从熵池生成的数位方面也不具有明显的优势。

尝试估算内部池中的熵的 Keymaster 实现假定 addRngEntropy 提供的数据不包含熵。如果在单次调用中向 Keymaster 实现提供的数据超过 2KiB，则这些实现可能会返回 ErrorCode::INVALID_INPUT_LENGTH。

4.1.4 generateKey

该函数在 Keymaster 1 中引入（名为 generate_key），并在 Keymaster 3 中进行了重命名。

该函数会生成一个新的加密密钥，同时指定关联的授权，这些授权会永久绑定到该密钥。Keymaster 实现确保任何与生成密钥时指定的授权不一致的方式使用密钥。对于安全硬件无法强制执行的授权，安全硬件的义务仅限于确保与密钥关联的无法强制执行的授权不能被修改，以便每次调用 getKeyCharacteristics 时都会返回原始值。此外，由 generateKey 返回的特性将授权正确地分配到由硬件强制执行的列表和由软件强制执行的列表。如需了解详情，请参见 **4.1.5** getKeyCharacteristics。

向 generateKey 提供的参数取决于要生成的密钥的类型。本部分总结了每种类型密钥的必需和可选标记。Tag::ALGORITHM 始终为必需的标记，用于指定密钥类型。

- RSA 密钥

  以下参数是生成 RSA 密钥所必需的参数。

  - Tag::KEY_SIZE 用于指定公开模数的大小（以位计）。如果缺少此参数，该方法会返回 ErrorCode::UNSUPPORTED_KEY_SIZE。支持的值包括 1024、2048、3072 和 4096。最好支持所有为 8 的倍数的密钥大小。
  - Tag::RSA_PUBLIC_EXPONENT 用于指定 RSA 公开指数的值。如果缺少此参数，该方法会返回 ErrorCode::INVALID_ARGUMENT。支持的值包括 3 和 65537。最好支持不超过 2^64 的所有质数值。

  以下参数不是生成 RSA 密钥所必需的参数，但如果在缺少这些参数的情况下生成 RSA 密钥，生成的密钥将无法使用。不过，如果缺少这些参数，generateKey 函数不会返回错误。

  - Tag::PURPOSE 用于指定允许的目的。对于 RSA 密钥，需要支持采用任意组合的所有目的。
  - Tag::DIGEST 用于指定可与新密钥配合使用的摘要算法。不支持任何摘要算法的实现需要接受包含不受支持的摘要的密钥生成请求。不支持的摘要应放入“由软件强制执行”的列表内的返回密钥特性中。这是因为相应密钥能够与其他摘要配合使用，但添加摘要会在软件中进行。然后，将调用硬件按 Digest::NONE 摘要算法执行相应操作。
  - Tag::PADDING 用于指定可与新密钥配合使用的填充模式。如果指定了任何不受支持的摘要算法，则不支持所有摘要算法的实现需要将 PaddingMode::RSA_PSS 和 PaddingMode::RSA_OAEP 放入由软件强制执行的密钥特性列表中。

- ECDSA 密钥
  - 只有 Tag::KEY_SIZE 是生成 ECDSA 密钥所必需的参数。此参数用于选择 EC 组。支持的值包括 224、256、384 和 521，这些值分别表示 NIST p-224、p-256、p-384 和 p521 曲线。
  - DIGEST 可使生成的 ECDSA 密钥生效，但此参数不是生成 ECDSA 密钥的必需参数。

- AES 密钥

- 只有 Tag::KEY_SIZE 是生成 AES 密钥所必需的参数。如果缺少此参数，该方法会返回 ErrorCode::UNSUPPORTED_KEY_SIZE。支持的值包括 128 和 256，可以选择支持 192 位 AES 密钥。
- 以下参数仅与 AES 密钥相关，但它们并不是生成 AES 密钥所必需的参数：
  - Tag::BLOCK_MODE 用于指定可与新密钥配合使用的分块模式。
  - Tag::PADDING 用于指定可以使用的填充模式。此参数仅与 ECB 和 CBC 模式相关。
  - 如果指定的是 GCM 分块模式，则提供 Tag::MIN_MAC_LENGTH。如果缺少此参数，该方法会返回 ErrorCode::MISSING_MIN_MAC_LENGTH。该标记的值为 8 的倍数且介于 96 和 128 之间。

- HMAC 密钥

  以下参数是生成 HMAC 密钥所必需的参数：
  - Tag::KEY_SIZE 用于指定密钥大小（以位计）。不支持小于 64 以及不是 8 的倍数的值。支持介于 64 和 512 之间并且是 8 的倍数的所有值。可以支持更大的值。
  - Tag::MIN_MAC_LENGTH 用于指定可通过相应密钥生成或验证的 MAC 的最小长度。此参数的值是 8 的倍数，并且不小于 64。
  - Tag::DIGEST 用于指定相应密钥的摘要算法。只能指定一种摘要，否则返回 ErrorCode::UNSUPPORTED_DIGEST。如果 Trustlet 不支持指定的摘要，则返回 ErrorCode::UNSUPPORTED_DIGEST。

- 密钥特性

  如果特性参数为非 NULL 值，generateKey 会返回新生成密钥的特性（相应划分到由硬件强制执行的列表和由软件强制执行的列表中）。要了解哪些特性会划分到哪个列表中，请参见 4.1.5 getKeyCharacteristics。
  - 返回的特性包含为生成密钥而指定的所有参数，但 Tag::APPLICATION_ID 和 Tag::APPLICATION_DATA 除外。如果密钥参数中包含这两个标记，为确保无法通过查看返回的密钥 Blob 找到这两个标记的值，系统会将这两个标记从返回的特性中移除。不过，这两个标记以加密形式绑定到密钥 Blob，以便在使用相应密钥时，如果未提供正确的值，使用会失败。
  - Tag::ROOT_OF_TRUST 同样会以加密形式绑定到相应密钥，但在生成或导入密钥期间可以不指定此标记，并且在任何情况下都不会返回此标记。
  - 除了收到的标记外，Trustlet 还会添加 Tag::ORIGIN（值为 KeyOrigin::GENERATED）；如果相应密钥可抗回滚，则还会添加 Tag::ROLLBACK_RESISTANT。抗回滚意味着，在使用 deleteKey 或 deleteAllKeys 删除密钥后，可以通过安全硬件保证绝对无法再使用该等密钥。不具抗回滚功能的实现通常会将生成或导入的密钥材料作为密钥 Blob（一种经过加密和身份验证的形式）返回给调用程序。当 Keystore 删除密钥 Blob 后，相应密钥将会消失，但之前已设法获取密钥材料的攻击者可能能够将相应密钥材料恢复到设备上。
  - 如果安全硬件保证被删除的密钥以后无法被恢复，那么相应密钥便可抗回滚。安全硬件通常是通过将额外的密钥元数据存储在攻击者无法操控的可信位置来做到这一点。在移动设备上，用于实现这一点的机制通常为 Replay Protected Memory Block(RPMB)。由于可创建的密钥数量基本上没有限制，而用于抗回滚的可信存储空间的大小则可能有限制，因此，该方法需要在即使无法为新密钥提供抗回滚功能的情况下同样取得成功。在这种情况下，不得将 Tag::ROLLBACK_RESISTANT 添加到密钥特性中。

4.1.5 getKeyCharacteristics

该函数在 Keymaster 1 中引入（名为 get_key_characteristics），并在 Keymaster 3 中进行了重命名。

用于返回与收到的密钥关联的参数和授权，并且返回的参数和授权会划分为两组：一组由硬件强制执行，一组由软件强制执行。此处的说明同样适用于通过 generateKey 和 importKey 返回的密钥特性列表。

如果在生成或导入密钥期间提供了 Tag::APPLICATION_ID，则在 clientId 参数中为此方法提供相同的值。否则，此方法会返回 ErrorCode::INVALID_KEY_BLOB。同样，如果在生成或导入密钥期间提供了 Tag::APPLICATION_DATA，则在 appData 参数中为此方法提供相同的值。

此方法返回的特性完整地说明了指定密钥的类型和用法。

要确定某个指定标记是属于由硬件强制执行的列表，还是属于由软件强制执行的列表，一般规则是：如果该标记的含义完全由安全硬件来保证，则属于由硬件强制执行的列表，否则属于由软件强制执行的列表。下面列出了可能无法明确确定到底属于哪个列表的具体标记：

- Tag::ALGORITHM、Tag::KEY_SIZE 和 Tag::RSA_PUBLIC_EXPONENT 是密钥的固有属性。对于任何由硬件来保障安全的密钥，这些标记都将位于由硬件强制执行的列表中。
- 由安全硬件支持的 Tag::DIGEST 值位于由硬件支持的列表中。不受支持的摘要则位于由软件支持的列表中。
- Tag::PADDING 的值通常位于由硬件支持的列表中，除非存在这样一种可能性，某种特定的填充模式必须要由软件来执行，在这种情况下，这些值将位于由软件强制执行的列表中。如果 RSA 密钥允许使用不是由安全硬件支持的摘要算法进行 PSS 或 OAEP 填充，则存在这种可能性。
- Tag::USER_SECURE_ID 和 Tag::USER_AUTH_TYPE 仅在硬件强制执行用户身份验证时，才会由硬件强制执行。要让用户身份验证由硬件强制执行，Keymaster Trustlet 和相关的身份验证 Trustlet 都必须是安全的，且共用一个用于签署和验证身份验证令牌的 HMAC 密钥。要了解详情，请参阅身份验证页面。
- Tag::ACTIVE_DATETIME、Tag::ORIGINATION_EXPIRE_DATETIME 和 Tag::USAGE_EXPIRE_DATETIME 标记要求能够访问可验证的正确时钟。大多数安全硬件只能访问由非安全操作系统提供的时间信息，这意味着这些标记由软件强制执行。
- 对于绑定到硬件的密钥，Tag::ORIGIN 始终位于硬件列表中。如果此标记出现在硬件列表中，更高的层级便可据此确定相应密钥是由硬件支持的。

4.1.6 attestKey

利用密钥认证，将应用中使用的密钥存储在设备硬件支持的 Keystore 中更安全。

在生产级环境中验证设备的硬件支持的密钥属性之前，您应确保设备支持硬件级密钥认证。为此，您应确保认证证书链包含由 Google 认证根密钥签署的根证书，且密钥说明数据结构中的 attestationSecurityLevel 元素设置为 TrustedEnvironment 安全级别。

在密钥认证期间，您可以指定密钥对的别名。认证工具为此提供了一个证书链，您可以用它来验证该密钥对的属性。

如果设备支持硬件级密钥认证，将使用认证根密钥签署此链中的根证书，设备制造商已在出厂时将根密钥注入到设备的硬件支持的 Keystore 中。

要实现密钥认证，请完成以下步骤：

步骤 1 使用 Keystore 对象的 getCertificateChain()方法获取指向与硬件支持的 Keystore 关联的 X.509 证书链的引用。

步骤 2 使用 CRL 对象的 isRevoked()方法检查每个证书的有效性。

说明

尽管您可以在应用中直接完成此流程，但在您信任的独立服务器上检查证书的调用列表更加安全。

步骤 3 创建一个 Attestation 对象，在证书链的第一个元素中将其作为参数传递。认证对象会提取此证书中的扩展数据，并将此信息存储为更容易访问的格式。如需了解有关扩展数据架构的更多详情，请参阅证书扩展数据架构。

步骤 4 使用 Attestation 类中的访问器方法从证书中检索扩展数据。这些方法使用与证书扩展数据架构中相同的名称和结构层次。

步骤 5 将来自 Attestation 对象的扩展数据与期望硬件支持的密钥将包含的值集合进行比较。

----结束

4.1.7 importKey

该函数在 Keymaster 1 中引入（名为 import_key），并在 Keymaster 3 中进行了重命名。

用于将密钥材料导入到 Keymaster 硬件中。密钥定义参数和输出特性的处理方式与 generateKey 相同，但存在以下例外情况：

- Tag::KEY_SIZE 和 Tag::RSA_PUBLIC_EXPONENT（仅限 RSA 密钥）不是输入参数中必需的标记。如果未收到这两个标记，Trustlet 会根据收到的密钥材料推导出这两个标记的值，并将适当的标记和值添加到密钥特性中。如果收到了这两个参数，Trustlet 会根据密钥材料对其进行验证。如果收到的值与密钥材料中的值不一致，该方法会返回 ErrorCode::IMPORT_PARAMETER_MISMATCH。
- 返回的 Tag::ORIGIN 与 KeyOrigin::IMPORTED 的值相同。

4.1.8 exportKey

该函数在 Keymaster 1 中引入（名为 export_key），并在 Keymaster 3 中进行了重命名。

用于从 KeymasterRSA 密钥对或 E 密钥对中导出公钥。

- 如果在生成或导入密钥期间提供了 Tag::APPLICATION_ID，则在 clientId 参数中为此方法提供相同的值。否则，此方法会返回 ErrorCode::INVALID_KEY_BLOB。
- 如果在生成或导入密钥期间提供了 Tag::APPLICATION_DATA，则在 appData 参数中为此方法提供相同的值。

4.1.9 deleteKey

该函数在 Keymaster 1 中引入（名为 delete_key），并在 Keymaster 3 中进行了重命名。

用于删除收到的密钥。此方法为可选方法，只能由提供抗回滚功能的 Keymaster 模块来实现。

4.1.10 deleteAllKeys

该函数在 Keymaster 1 中引入（名为 delete_all_keys），并在 Keymaster 3 中进行了重命名。

用于删除所有密钥。此方法为可选方法，只能由提供抗回滚功能的 Keymaster 模块来实现。

4.1.11 destroyAttestationIds

可以使用 destroyAttestationIds()方法永久停用新的 ID 认证功能。该功能是一项可选功能，但建议启用该功能。

- 如果 TEE 无法确保在调用此方法后永久停用相应的 ID 认证，则一定不得实现 ID 认证。
  此方法不执行任何操作并返回 ErrorCode::UNIMPLEMENTED。
- 如果支持 ID 认证，则需要实现此方法且必须永久停用未来尝试进行 ID 认证的所有操作。
- 此方法的调用次数不受限制。
- 如果已永久停用 ID 认证，则此方法不会执行任何操作并返回 ErrorCode::OK。

此方法只可能返回以下错误代码：ErrorCode::UNIMPLEMENTED（如果不支持 ID 认证）、ErrorCode::OK、ErrorCode::KEYMASTER_NOT_CONFIGURED 或指示无法与安全硬件通信的错误代码之一。

4.1.12 begin

使用指定的密钥和参数（视情况而定）针对指定的目的开始执行加密操作，并返回用于与 update 和 finish 配合使用来完成操作的操作句柄。该操作句柄还会在经过身份验证的操作中用作“质询”令牌，并且对于此类操作，该操作句柄包含在身份验证令牌的 challenge 字段中。

Keymaster 实现支持至少 16 个并行操作。Keystore 最多使用 15 个，留一个给 vold 用于对密码进行加密。当 Keystore 有 15 个操作正在进行（已调用 begin，但尚未调用 finish 或 abort）时，Keystore 收到开始第 16 个操作的请求，它会对最近使用最少的操作调用 abort，以便将进行中的操作减少到 14 个，然后再调用 begin 来开始执行新请求的操作。

如果在生成或导入密钥期间指定了 Tag::APPLICATION_ID 或 Tag::APPLICATION_DATA，那么调用 begin 时会包含这两个标记，标记的值为最初在 inParams 参数中为此方法指定的值。

- 密钥授权强制执行
  在执行此方法期间，如果实现将以下密钥授权放入到了“由硬件强制执行的”特性中，并且相应操作不是公钥操作，那么这些密钥授权将由 Trustlet 来强制执行。即使不符合授权要求，也会允许公钥操作（即使用 RSA 或 EC 密钥进行的 KeyPurpose::ENCRYPT 和 KeyPurpose::VERIFY）成功完成。
  - Tag::PURPOSE：除非请求的操作是公钥操作，否则在 begin()调用中指定的目的必须与密钥授权中的某个目的一致。如果指定的目的不一致且操作并非公钥操作，则 begin 会返回 ErrorCode::UNSUPPORTED_PURPOSE。公钥操作是不对称加密或验证操作。
  - 只有可信 UTC 时间源可用时才能强制执行 Tag::ACTIVE_DATETIME。如果当前日期和时间早于此标记的值，该方法会返回 ErrorCode::KEY_NOT_YET_VALID。
  - 只有可信 UTC 时间源可用时才能强制执行 Tag::ORIGINATION_EXPIRE_DATETIME。如果当前日期和时间晚于此标记的值，并且目的是 KeyPurpose::ENCRYPT 或 KeyPurpose::SIGN，该方法会返回 ErrorCode::KEY_EXPIRED。
  - 只有可信 UTC 时间源可用时才能强制执行 Tag::USAGE_EXPIRE_DATETIME。如果当前日期和时间晚于此标记的值，并且目的是 KeyPurpose::DECRYPT 或 KeyPurpose::VERIFY，该方法会返回 ErrorCode::KEY_EXPIRED。
  - Tag::MIN_SECONDS_BETWEEN_OPS 会与指明相应密钥上次使用时间的可信相对计时器进行比较。如果上次使用时间加上此标记的值小于当前时间，该方法会返回 ErrorCode::KEY_RATE_LIMIT_EXCEEDED。要查看重要的实现详情，请参阅标记说明。

- Tag::MAX_USES_PER_BOOT 会与用于跟踪自系统启动以来相应密钥使用次数的安全计数器进行比较。如果之前的使用次数已超过此标记的值，该方法会返回 ErrorCode::KEY_MAX_OPS_EXCEEDED。
- 仅当相应密钥还具有 Tag::AUTH_TIMEOUT 时，此方法才会强制执行 Tag::USER_SECURE_ID。如果相应密钥同时具有这两个标记，此方法必须在 inParams 中收到有效的 Tag::AUTH_TOKEN。要使身份验证令牌有效，必须满足以下所有条件：
  - HMAC 字段可正确验证。
  - 相应密钥中有至少一个 Tag::USER_SECURE_ID 值与令牌中至少一个安全 ID 值一致。
  - 相应密钥具有与令牌中的身份验证类型一致的 Tag::USER_AUTH_TYPE。
  - 如果上述条件中有任何一个不满足，该方法就会返回 ErrorCode::KEY_USER_NOT_AUTHENTICATED。
- Tag::CALLER_NONCE 允许调用程序指定随机数或初始化矢量(IV)。如果相应密钥没有此标记，但调用程序向此方法提供了 Tag::NONCE，则返回 ErrorCode::CALLER_NONCE_PROHIBITED。
- Tag::BOOTLOADER_ONLY 用于指定相应密钥只能由引导加载程序使用。如果在引导加载程序执行完毕后调用此方法，并且提供的是仅限引导加载程序使用的密钥，则返回 ErrorCode::INVALID_KEY_BLOB。

- RSA 密钥

  执行任何 RSA 密钥操作时，都会在 inParams 中指定一种且只能指定一种填充模式。如果未指定或指定了多次，该方法会返回 ErrorCode::UNSUPPORTED_PADDING_MODE。

  RSA 签名和验证操作需要摘要，就像使用 OAEP 填充模式进行 RSA 加密和解密操作一样。在这类情况下，调用程序会在 inParams 中指定一种且只能指定一种摘要。如果未指定或指定了多次，该方法会返回 ErrorCode::UNSUPPORTED_DIGEST。

  - 私钥操作（KeyPurpose::DECYPT 和 KeyPurpose::SIGN）要求摘要和填充获得授权，也就是说，密钥授权需要包含指定的值。否则，该方法会根据具体情况返回 ErrorCode::INCOMPATIBLE_DIGEST 或 ErrorCode::INCOMPATIBLE_PADDING。
  - 公钥操作（KeyPurpose::ENCRYPT 和 KeyPurpose::VERIFY）可以使用未经授权的摘要或填充。

  除了 PaddingMode::NONE 之外，所有 RSA 填充模式都仅适用于特定目的。具体来说就是，PaddingMode::RSA_PKCS1_1_5_SIGN 和 PaddingMode::RSA_PSS 仅支持签名和验证，而 PaddingMode::RSA_PKCS1_1_1_5_ENCRYPT 和 PaddingMode::RSA_OAEP 仅支持加密和解密。如果指定的模式不支持指定的目的，该方法会返回 ErrorCode::UNSUPPORTED_PADDING_MODE。

  填充模式与摘要之间存在以下非常重要的相互关系：

  - PaddingMode::NONE 表示执行“原始”RSA 操作。如果是进行签名或验证，则指定 Digest::NONE 这种摘要。如果是进行非填充式加密或解密，则不需要摘要。
  - PaddingMode::RSA_PKCS1_1_5_SIGN 填充需要摘要。摘要可以是 Digest::NONE，在这种情况下，Keymaster 实现无法构建适当的 PKCS#1 v1.5 签名结构，因为它无法添加 DigestInfo 结构。不过，实现会构建 0x00 || 0x01 || PS || 0x00 || M，其中 M 是收到的消息，PS 是填充字符串。RSA 密钥的大小要比消息至少多 11 个字节，否则该方法会返回 ErrorCode::INVALID_INPUT_LENGTH。
  - PaddingMode::RSA_PKCS1_1_1_5_ENCRYPT 填充不需要摘要。
  - PaddingMode::RSA_PSS 填充需要摘要，并且摘要不能是 Digest::NONE。如果指定了 Digest::NONE，该方法会返回 ErrorCode::INCOMPATIBLE_DIGEST。此外，RSA 密钥的大小必须至少比摘要的输出大小大 2 + D 字节，其中 D 表示摘要的大小（以字节计）。否则，该方法会返回 ErrorCode::INCOMPATIBLE_DIGEST。盐的大小为 D。

- PaddingMode::RSA_OAEP 填充需要摘要，并且摘要不能是 Digest::NONE。如果指定了 Digest::NONE，该方法会返回 ErrorCode::INCOMPATIBLE_DIGEST。

- EC 密钥

  执行 EC 密钥操作时，会在 inParams 中指定一种且只能指定一种填充模式。如果未指定或指定了多次，该方法会返回 ErrorCode::UNSUPPORTED_PADDING_MODE。

  - 私钥操作(KeyPurpose::SIGN)要求摘要和填充获得授权，也就是说，密钥授权需要包含指定的值。否则返回 ErrorCode::INCOMPATIBLE_DIGEST。
  - 公钥操作(KeyPurpose::VERIFY)可以使用未经授权的摘要或填充。

- AES 密钥

  执行 AES 密钥操作时，会在 inParams 中指定一种且只能指定一种分块模式和填充模式。如果有任何一项未指定或指定了多次，则返回 ErrorCode::UNSUPPORTED_BLOCK_MODE 或 ErrorCode::UNSUPPORTED_PADDING_MODE。指定的模式必须要通过相应密钥授权，否则，该方法会返回 ErrorCode::INCOMPATIBLE_BLOCK_MODE 或 ErrorCode::INCOMPATIBLE_PADDING_MODE。

  - 如果分块模式是 BlockMode::GCM，则在 inParams 中指定 Tag::MAC_LENGTH。指定的值是 8 的倍数，并且不大于 128，也不小于密钥授权中 Tag::MIN_MAC_LENGTH 的值。
    - 如果 MAC 长度大于 128 或不是 8 的倍数，则返回 ErrorCode::UNSUPPORTED_MAC_LENGTH。
    - 如果 MAC 长度小于密钥最小长度，则返回 ErrorCode::INVALID_MAC_LENGTH。
  - 如果分块模式是 BlockMode::GCM 或 BlockMode::CTR，那么指定的填充模式必须是 PaddingMode::NONE。
  - 如果分块模式是 BlockMode::ECB 或 BlockMode::CBC，那么指定的填充模式可以是 PaddingMode::NONE 或 PaddingMode::PKCS7。
  - 如果填充模式不符合这些条件，则返回 ErrorCode::INCOMPATIBLE_PADDING_MODE。
  - 如果分块模式是 BlockMode::CBC、BlockMode::CTR 或 BlockMode::GCM，则需要初始化矢量或随机数。在大多数情况下，调用程序都不应提供 IV 或随机数。在这种情况下，Keymaster 实现会生成一个随机 IV 或随机数，并通过 outParams 中的 outParams 将其返回。CBC 和 CTR IV 均为 16 个字节。GCM 随机数为 12 个字节。如果密钥授权包含 Tag::CALLER_NONCE，调用程序可能会通过 inParams 中的 Tag::NONCE 提供 IV/随机数。
    - 如果在 Tag::CALLER_NONCE 未获得授权的情况下提供了随机数，则会返回 ErrorCode::CALLER_NONCE_PROHIBITED。
    - 如果在 Tag::CALLER_NONCE 获得授权的情况下未提供随机数，则会生成随机 IV/随机数。

- HMAC 密钥

  执行 HMAC 密钥操作时，会在 inParams 中指定 Tag::MAC_LENGTH。指定的值必须是 8 的倍数，并且不大于摘要长度，也不小于密钥授权中 Tag::MIN_MAC_LENGTH 的值。

  - 如果 MAC 长度大于摘要长度或不是 8 的倍数，则返回 ErrorCode::UNSUPPORTED_MAC_LENGTH。
  - 如果 MAC 长度小于密钥最小长度，则返回 ErrorCode::INVALID_MAC_LENGTH。

4.1.13 update

用于提供要在通过 begin 开始且正在进行的操作中处理的数据。操作是通过 operationHandle 参数指定的。

为了更灵活地处理缓冲区，此方法的实现可以选择不消耗完收到的数据。调用程序负责执行循环操作，以便在后续调用中馈送其余数据。在 inputConsumed 参数中返回所消耗的输入数据量。实现始终消耗至少一个字节，除非相应操作无法再接受更多字节；如果收到了零个以上的字节，但消耗了零字节，调用程序会将此视为错误并中止相应操作。

实现还可以选择返回多少数据（作为 update 的结果）。这仅与加密和解密操作有关，因为在调用 finish 之前，签名和验证操作不会返回任何数据。尽早返回数据，而不是缓冲数据。

- 错误处理

  如果此方法返回除 ErrorCode::OK 之外的错误代码，那么相应操作会被中止，操作句柄也会变为无效。如果以后再将该句柄与此方法、finish 或 abort 配合使用，则都会返回 ErrorCode::INVALID_OPERATION_HANDLE。

- 密钥授权强制执行

  密钥授权强制执行主要在 begin 中进行。不过，密钥存在以下情况时例外：一个或多个 Tag::USER_SECURE_IDs，且没有 Tag::AUTH_TIMEOUT。在这种情况下，对于每一项操作，密钥都会要求提供授权，并且 update 方法会在 inParams 参数中收到 Tag::AUTH_TOKEN。HMAC 会验证该令牌有效且包含匹配的安全用户 ID，与密钥的 Tag::USER_AUTH_TYPE 匹配，并且包含质询字段中当前操作的操作句柄。如果不符合这些条件，则返回 ErrorCode::KEY_USER_NOT_AUTHENTICATED。

  调用程序会在每次调用 update 和 finish 时提供身份验证令牌。实现只需对该令牌验证一次（如果它倾向于这么做）。

- RSA 密钥

  对于使用 Digest::NONE 的签名和验证操作，此方法会在单次 update 中接受要签署或验证的整个分块。此方法不会只消耗分块的一部分。不过，如果调用程序选择在多次 update 中提供数据，此方法会接受相应数据。如果调用程序提供的要签署的数据多于可以消耗的数据（数据长度超出 RSA 密钥大小），则返回 ErrorCode::INVALID_INPUT_LENGTH。

- ECDSA 密钥

  对于使用 Digest::NONE 的签名和验证操作，此方法会在单次 update 中接受要签署或验证的整个分块。此方法不会只消耗分块的一部分。

  不过，如果调用程序选择在多次 update 中提供数据，此方法会接受相应数据。如果调用程序提供的要签署的数据多于可以消耗的数据，则以静默方式截断这些数据。（这与处理在类似 RSA 操作中提供的超量数据不同，因为此方法与旧版客户端兼容。）

- AES 密钥

  AES GCM 模式支持通过 inParams 参数的中的 Tag::ASSOCIATED_DATA 标记提供的“相关身份验证数据”。可以在重复调用（如果数据太大而无法在单个分块中发送，那么重复调用非常重要）中提供相关数据，但始终先于要加密或解密的数据提供。update 调用可以同时接收相关数据以及要加密/解密的数据，但后续 update 中不会包含相关数据。如果调用程序已在某次调用 update 时提供了要加密/解密的数据，若再次向 update 调用提供相关数据，则返回 ErrorCode::INVALID_TAG。

  对于 GCM 加密，此标记会通过 finish 附加到密文中。在解密期间，向上一次 update 调用提供的数据的最后 Tag::MAC_LENGTH 个字节就是此标记。由于 update 的指定调用无法得知自己是否为最后一次调用，因此它会处理除标记长度之外的所有数据，并缓冲调用 finish 期间可能用到的标记数据。

4.1.14 finish

用于完成通过 begin 开始且正在进行的操作，负责处理 update 所提供的所有尚未处理的数据。

此方法是操作期间调用的最后一个方法，因此会返回所有处理后的数据。

无论是成功完成还是返回错误，此方法都会结束相应操作，从而使收到的操作句柄无效。如果以后再将该句柄与此方法、update 或 abort 配合使用，则都会返回 ErrorCode::INVALID_OPERATION_HANDLE。

签名操作将返回签名作为输出。验证操作将接受 signature 参数中的签名，并且不会返回任何输出。

- 密钥授权强制执行

  密钥授权强制执行主要在 begin 中进行。不过，密钥存在以下情况时例外：一个或多个 Tag::USER_SECURE_IDs，且没有 Tag::AUTH_TIMEOUT。在这种情况下，对于每一项操作，密钥都会要求提供授权，并且 update 方法会在 inParams 参数中收到 Tag::AUTH_TOKEN。HMAC 会验证该令牌有效且包含匹配的安全用户 ID，与密钥的 Tag::USER_AUTH_TYPE 匹配，并且包含质询字段中当前操作的操作句柄。如果不符合这些条件，则返回 ErrorCode::KEY_USER_NOT_AUTHENTICATED。

  调用程序会在每次调用 update 和 finish 时提供身份验证令牌。实现只需对该令牌验证一次（如果它倾向于这么做）。

- RSA 密钥

  有一些附加要求，具体取决于填充模式：

  - PaddingMode::NONE：对于非填充式签名和加密操作，如果收到的数据比密钥短，那么在签名/加密之前，在数据左侧填充零来补齐。如果数据与密钥一样长度，但数值较大，则返回 ErrorCode::INVALID_ARGUMENT。对于验证和解密操作，数据必须与密钥一样长。否则返回 ErrorCode::INVALID_INPUT_LENGTH。
  - PaddingMode::RSA_PSS：对于 PSS 填充式签名操作，PSS 盐不得短于 20 个字节，并且是随机生成的。盐可以更长；参考实现使用的是长度最大的盐。调用 begin 时使用 inputParams 中的 Tag::DIGEST 指定的摘要会用作 PSS 摘要算法，而 SHA1 会用作 MGF1 摘要算法。
  - PaddingMode::RSA_OAEP：调用 begin 时使用 inputParams 中的 Tag::DIGEST 指定的摘要会用作 OAEP 摘要算法，而 SHA1 会用作 MGF1 摘要算法。

- ECDSA 密钥

  如果为非填充式签名或验证操作提供的数据太长，则要将其截断。

- AES 密钥

  有一些附加条件，具体取决于分块模式：

  - BlockMode::ECB 或 BlockMode::CBC：如果填充模式是 PaddingMode::NONE，并且数据长度不是 AES 分块大小的倍数，则返回 ErrorCode::INVALID_INPUT_LENGTH。如果填充模式是 PaddingMode::PKCS7，则按照 PKCS#7 规范填充数据。请注意，PKCS#7 建议，如果数据长度是分块长度的倍数，则添加一个额外的填充分块。
  - BlockMode::GCM：在加密期间，处理所有明文之后，会计算此标记（Tag::MAC_LENGTH 个字节）并将其附加到返回的密文中。在解密期间，会将最后 Tag::MAC_LENGTH 个字节作为标记处理。如果标记验证失败，则返回 ErrorCode::VERIFICATION_FAILED。

4.1.15 abort

用于中止正在进行的操作。调用 abort 之后，如果后续再将收到的操作句柄与 update、finish 或 abort 配合使用，则返回 ErrorCode::INVALID_OPERATION_HANDLE。

4.1.16 get_supported_algorithms

用于返回一个列表，其中包含 Keymaster 硬件实现支持的算法。

- 如果是软件实现，则返回一个空列表。

- 如果是混合实现，则返回一个仅包含硬件支持的算法的列表。

Keymaster 1 实现支持 RSA、EC、AES 和 HMAC。

该函数在 Keymaster 1 中定义，但在 Keymaster 2 中已被移除。

4.1.17 get_supported_block_modes

用于返回一个列表，其中包含对于指定的算法和目的，Keymaster 硬件实现支持的 AES 分块模式。

对于不是分块加密算法的 RSA、EC 和 HMAC，无论是任何有效目的，此方法都会返回一个空列表。如果目的无效，则应导致此方法返回 ErrorCode::INVALID_PURPOSE。

Keymaster 1 实现支持使用 ECB、CBC、CTR 和 GCM 进行 AES 加密和解密。

该函数在 Keymaster 1 中定义，但在 Keymaster 2 中已被移除。

4.1.18 get_supported_padding_modes

用于返回一个列表，其中包含对于指定的算法和目的，Keymaster 硬件实现支持的填充模式。

HMAC 和 EC 并没有填充这一概念，因此针对所有有效目的，此方法都会返回一个空列表。如果目的无效，则应导致此方法返回 ErrorCode::INVALID_PURPOSE。

对于 RSA，Keymaster 1 实现支持：

- 非填充式加密、解密、签名和验证。对于非填充式加密和签名，如果消息比公开模数短，实现必须要在消息左侧填充零来补齐。对于非填充式解密和验证，输入长度必须与公开模数的大小一致。
- PKCS#1 v1.5 加密和签名填充模式。
- 盐最小长度为 20 的 PSS。
- OAEP。

对于采用 ECB 和 CBC 模式的 AES 算法，Keymaster 1 实现支持无填充和 PKCS#7 填充。CTR 和 GCM 模式仅支持无填充。

该函数在 Keymaster 1 中定义，但在 Keymaster 2 中已被移除。

4.1.19 get_supported_digests

用于返回一个列表，其中包含对于指定的算法和目的，Keymaster 硬件实现支持的摘要模式。

任何 AES 模式都不支持摘要，也不需要摘要，因此无论是任何有效目的，此方法都会返回一个空列表。

实现 Keymaster 1 时可以只实现一部分已定义的摘要。实现会提供 SHA-256，且可以提供 MD5、SHA1、SHA-224、SHA-256、SHA384 和 SHA512（完整的已定义摘要集）。

该函数在 Keymaster 1 中定义，但在 Keymaster 2 中已被移除。

4.1.20 get_supported_import_formats

用于返回一个列表，其中包含指定算法的 Keymaster 硬件实现支持的导入格式。

Keymaster 1 实现支持 PKCS#8 格式（无密码保护），以便导入 RSA 密钥对和 EC 密钥对，并且支持以原始格式导入 AES 密钥材料和 HMAC 密钥材料。

该函数在 Keymaster 1 中定义，但在 Keymaster 2 中已被移除。

4.1.21 get_supported_export_formats

用于返回一个列表，其中包含指定算法的 Keymaster 硬件实现支持的导出格式。

Keymaster1 实现支持 X.509 格式，以便导出 RSA 公钥和 EC 公钥。不支持导出私钥或非对称密钥。

该函数在 Keymaster 1 中定义，但在 Keymaster 2 中已被移除。

4.1.22 getHmacSharingParameters

该函数在 Keymaster 4 中引入。

getHmacSharingParameters 函数用于创建与另一个 IKeymasterDevice 实现共享的 HMAC 密钥，并返回 HmacSharingParameters。该函数是就共享密钥达成一致的过程中的第一步，与 computeSharedHmac 配合使用。

- HMAC 密钥用于 MAC 和验证 authentication tokens(HardwareAuthToken，VerificationToken 和 ConfirmationToken 都用这个 HMAC 密钥)，必须在 TEE 和 StrongBox 之间共享以便互相验证对方生成的 tokens。
- 该函数在启动时被 Android 调用。系统在每个 HAL 实例上调用它并收集结果，为第二步做准备。
- 该函数可能返回三种错误代码。
  - 如果没有错误，返回 ErrorCode::OK。
  - 如果 HMAC 协议没有实现，返回 ErrorCode::UNIMPLEMENTED。
  - 如果参数不能返回，返回 ErrorCode::UNKNOWN_ERROR。

说明

- 任何带有 StrongBox 的 IKeymasterDevice 都有两个 IKeymasterDevice 实例，因为必须有一个 TEEKeymaster。
- 所有 4.0::IKeymasterDevice HALS 必须实现 HMAC 协议，无论 HAL 是否用于带有 StrongBox 的设备。
- seed 必须在给定设备上的每一个调用方法中包含相同的值，而 nonce 必须在开机阶段为每次调用返回相同的值。

4.1.23 computeSharedHmac

该函数在 Keymaster 4 中引入。

computeSharedHmac 函数用于完成 HMAC 密钥的创建，与另一个 IKeymasterDevice 实现共享。该函数是就共享密钥达成一致的过程中的第二步，也是最后一步，与 getHmacSharingParameters 配合使用。

- HMAC 密钥用于 MAC 和验证 authentication tokens，必须在 TEE 和 StrongBox 之间共享以便互相验证对方生成的 tokens。
- 该函数在启动时被 Android 调用。系统在每个 HAL 实例上调用它并收集结果，并将所有 HALs 返回的所有 HmacSharingParameters 都发送给它。
- 为了确保 HmacSharingParameters 的一致顺序，调用者必须按顺序对参数进行排序。详情请参阅 /support/keymaster_util .cpp 中对顺序的定义：

```
bool operator<(const HmacSharingParameters& a, const HmacSharingParameters& b) {
return std::tie(a.seed, a.nonce) < std::tie(b.seed, b.nonce);
}
```

- 该方法计算共享的 32 字节 HMAC，其中：

H = CKDF(key = K, context = P1 || P2 || ... || Pn, label = "KeymasterSharedMac")

- CKDF 是 NIST SP 800-108 计数器模式下的 AES-CMAC KDF 标准。标准中对 key、context 和 lable 进行定义，lable 字符串是 UTF-8 编码的。计数器有前缀和后缀，后缀长度为 L，参阅标准第 12 页的结构。
- “ || ”表示连接。
- Pi 是 params 向量中第 i 个 HmacSharingParameters 值。
  - 目前只支持两个 IKeymasterDevice 实现，但是这种机制无需修改即可扩展到任意数量的 IKeymasterDevice 实现。
  - HmacSharingParameters 的编码是它的两个字段的连接，即 seed || nonce。
  - lable“KeymasterSharedMac”是字符串的 18 字节 UTF-8 编码。
- K 是一个预先建立的共享密钥，在恢复出厂设置时重置。建立这个共享密钥的机制是由实现定义的，请参阅下面推荐的建立 K 的流程。该方法假设 TEEIKeymasterDevice 没有可用的存储空间，而 StrongBox IKeymasterDevice 有可用的存储空间。

  任何能够确保攻击者无法获取或派生 K 值的安全建立 K 的方法都是可接受的。建议的方法是在每个恢复出厂设置期间执行。它依靠工厂安装的认证密钥来减少中间人攻击。该协议要求其中一个实例具有安全的持久存储。之所以选择这个模型，是因为根据定义 StrongBox 具有安全的持久存储，但 TEE 可能没有。假设没有存储的实例能够派生唯一的硬件绑定密钥 HBK，该密钥仅用于此目的，不能在安全环境之外派生。下面是一个实例，其中 T 为无存储的 IKeymasterDevice 实例，S 为有存储的 IKeymasterDevice。

a T 生成一个暂时的 EC P-256 密钥对 K1。

b T 发送 K1_pub 给 S，用的认证密钥签名。

c S 验证 K1_pub 上的签名。

d S 生成一个临时的 EC P-256 密钥对 K2。

e S 将{K1_pub, K2_pub}发送给 T，T 用 S 的认证密钥签名。

f T 验证{K1_pub, K2_pub}上的签名。

g T 使用{K1_priv, K2_pub}和 ECDH 计算用户密钥 Q。

h T 生成一个随机的 seed S。

i T 计算 K = KDF(HBK, S)，其中 KDF 是安全密钥派生函数。

j T 将 M = AES-GCM-ENCRYPT(Q, {S || K})发送给 S。

k S 使用{K2_priv, K1_pub}和 ECDH 计算用户密钥 Q。

l S 计算 S || K = AES-GCM-DECRYPT(Q, M)并存储 S 和 K。

说明

- 所有 IKeymasterDevice 实例执行相同的计算，以得到相同的结果。
- IKeymasterDevice 实例创建的所有密钥都必须以密码方式绑定到 K 的值，这样建立一个新的 K 就会永久性地破坏它们。
- 当 S 接收到 getHmacSharingParameters 调用时，它将存储的 S 作为 seed 和 nonce 返回。当 T 接收到相同的调用时，它返回一个空的 seed 和 nonce。当 T 收到 computeSharedHmac 调用时，它使用 S 提供的 seed 来计算 K(S 存储了 K)。

- 参数 params 为调用 getHmacSharingParameters 时，所有 IKeymasterDevice 实例返回的 HmacSharingParameters 数据。

- 该函数可能返回两种错误代码。
  - 如果没有错误，返回 ErrorCode::OK。
  - 如果任何一个提供的 params 不是之前调用的 getHmacParameters()返回的值，返回 ErrorCode::UNIMPLEMENTED。
- 该函数将返回一个 32 字节的 sharingCheck，用于验证所有 IKeymasterDevice 实例都计算了相同的共享 HMAC 密钥。计算方法如下。

```
sharingCheck = HMAC(H, "KeymasterHMACVerification")
```

字符串用 UTF-8 编码，长度为 27 字节。

说明

- 任何带有 StrongBox 的 IKeymasterDevice 都有两个 IKeymasterDevice 实例，因为必须有一个 TEEIKeymasterDevice。
- 如果返回所有的值都与 IKeymasterDevice 实例不匹配，客户端必须认为 HMAC 协议失败。

4.1.24 verifyAuthorization

该函数在 Keymaster 4 中引入。

verifyAuthorization 用于验证另一个 IKeymasterDevice 实例的授权。在同时具有 StrongBox 和 TEEIKeymasterDevice 实例的系统上，有时让 TEE IKeymasterDevice 验证在 StrongBox 中存放的密钥的授权是有用的。

- 对于每个 StrongBox 操作，Keystore 都需要在 TEE Keymaster 上调用这个函数，传递 StrongBox 密钥的硬件强制授权列表和由 StrongBox begin()返回的操作句柄。
- TEE IKeymasterDevice 必须验证它所能验证的所有授权，并在 VerificationToken 中返回所有验证过的授权。
- 如果不能验证任何内容，VerificationToken 的 parametersVerified 字段必须为空。然后 Keystore 将 VerificationToken 传递给随后的 StrongBox update()和 finish()调用。
- StrongBox 实现必须返回 ErrorCode:: unimplementation。
- 该函数有三个参数 operationHandle、parametersToVerify 和 authToken。
  - 参数 operationHandle 为由 StrongBox Keymaster 的 begin()返回的操作句柄。
  - 参数 parametersToVerify 要验证的授权集。如果唯一需要的信息是 TEE 时间戳，调用者将会提供一个空向量。
  - 参数 authToken 为 HardwareAuthToken，如果需要授权密钥使用的话。
- 该函数可能三种错误代码。
  - 如果没有错误，返回 ErrorCode::OK。
  - 如果 IKeymasterDevice 是 StrongBox，返回 ErrorCode:: unimplementation。
  - 如果 IKeymasterDevice 不能验证 parametersToVerify 的一个或多个元素，则不能返回错误代码，而是仅忽略 VerificationToken 中未验证的参数。
- 返回值 token 为 VerificationToken。VerificationToken 详情请参阅 types.hal。

4.1.25 importWrappedKey

该函数在 Keymaster 4.1 中引入。

importWrappedKey 函数用于安全地导入一个密钥或密钥对，返回一个 keyBlob 和对导入的密钥的描述。

- 参数 wrappedKeyData 为要导入的 wrapped key 的材料，wrapped key 使用 DER-encoded ASN.1 模式，由下面的模板给定:

```
KeyDescription ::= SEQUENCE(
        keyFormat INTEGER,                  # Values from KeyFormat enum.
        keyParams AuthorizationList,
)

SecureKeyWrapper ::= SEQUENCE(
        version INTEGER,                    # Contains value 0
        encryptedTransportKey OCTET_STRING,
        initializationVector OCTET_STRING,
        keyDescription KeyDescription,
        encryptedKey OCTET_STRING,
        tag OCTET_STRING
)
```

其中:

- keyFormat 是 keyFormat enum 中的一个整数，keyFormat enum 定义了明文密钥材料的格式。
- keyParams 是要导入的密钥的特征（与 generateKey 或 importKey 一样）。如果安全导入成功，这些特征必须与密钥准确关联，就像密钥材料没有通过 @3.0::IKeymasterDevice::importKey 进行安全导入一样。有关 AuthorizationList 模式的文档，请参阅 **4.1.6** attestKey。
- encryptedTransportKey 是一个 256 位的 AES 密钥，与 maskingkey 进行异或运算，然后使用 wrappingKeyBlob 指定的 wrapped key 进行加密。
- keyDescription 为以上的密钥描述。
- encryptedKey 是要导入的密钥的密钥材料，格式为 keyFormat，并在 AES-GCM 模式下使用 encryptedEphemeralKey 进行加密，作为附加的已认证数据，使用 DER 编码的 KeyDescription 表示来提供。
- tag 是 encryptedKey 经过 AES-GCM 加密产生的标签。

- 因此，importWrappedKey 做如下操作:

a 获取 wrappingKeyBlob 的私钥材料，验证 wrapping key 具有 Purpose:: KEY_WRAP、填充模式 RSA_OAEP 和摘要 SHA_2_256，如果任一要求失败，返回对应的错误代码。

b 从 SecureKeyWrapper 中提取 encryptedTransportKey 字段，并用 wrapping key 进对其解密。

c 用 maskingKey 与步骤 **b** 的结果进行异或运算。

d 使用步骤 **c** 的结果作为 AES-GCM 密钥对 encryptedKey 解密，使用 keyDescription 的编码值作为附加已认证数据。为下一步调用结果“keyData”。

e 执行与调用 importKey(keyParams、keyFormat、keyData) 等价的操作，其中原始标记设置为 securely_import。

- 参数 wrappingKeyBlob 为 generateKey()或 importKey()返回的不透明密钥描述。这个密钥必须是用 Purpose::WRAP_KEY 创建的。
- 参数 maskingKey 为在 SecureWrappedKey 结构中参与传输密钥异或运算的值。它长度为 32 个字节。
- 参数 unwrappingParams 必须包含执行 unwrapping 操作所需的所有参数。例如，如果 wrapping key 是 AES 密钥，则必须在此参数中指定块和填充模式。

- 参数 passwordSid 指定拥有正在安装的密钥的用户的密码安全 ID(SID)。
  - 如果 wrappedKeyData 中的授权列表包含 Tag::USER_SECURE_ID 且设置了 HardwareAuthenticatorType::PASSWORD 位的值，则构造的密钥必须绑定到此参数提供的 SID 值。
  - 如果 wrappedKeyData 不包含这样的标记和值，则必须忽略此参数。
- 参数 biometricSid 指定拥有正在安装的密钥的用户的生物特征安全 ID(SID)。
  - 如果 wrappedKeyData 中的授权列表包含 Tag::USER_SECURE_ID 且设置了 HardwareAuthenticatorType::FINGERPRINT 位的值，则构造的密钥必须绑定到此参数提供的 SID 值。
  - 如果 wrappedKeyData 不包含这样的标记和值，则必须忽略此参数。
- 返回值 keyBlob 为导入的密钥的不透明描述。建议 keyBlob 包含密钥材料的副本，且该副本包装在安全硬件之外的不可用密钥中。

4.1.26 deviceLocked

该函数在 Keymaster 4.1 中引入。

deviceLocked 函数由客户端调用，用于通知 IKeymasterDevice 当前设备已锁定，且带有 UNLOCKED_DEVICE_REQUIRED 标记的密钥将不再可用。

IKeymasterDevice 应在该函数被调用时注意当前的时间戳，所有尝试使用 UNLOCKED_DEVICE_REQUIRED 密钥的操作都将被拒绝，并返回错误代码 Error::DEVICE_LOCKED，直到出现一个新的带有时间戳的 authentication token。

- 如果“passwordOnly”参数设置为 true，那么 sufficiently-recent authentication token 必须表明用户使用密码进行验证，而不是生物特征验证。
- IKeymasterDevice 的 UNLOCKED_DEVICE_REQUIRED 与 KeyStore 强制执行的 UNLOCKED_DEVICE_REQUIRED 含义略有不同。
  - Keystore 根据每个用户来处理设备锁定。因为 auth tokens 不包含 Android 用户 ID，不可能在 IKeymasterDevice 中复制 keyStore 的强制执行逻辑。
  - 从 IKeymasterDevice 的角度来看，任何 user unlock 都会解锁所有的 UNLOCKED_DEVICE_REQUIRED 密钥。Keystore 将继续执行每个用户的设备锁定。
- 参数 passwordOnly 在能够使用 UNLOCKED_DEVICE_REQUIRED 密钥之前指定设备是否必须使用密码解锁，而不是生物特征解锁。
- 参数 verificationToken 用于 IKeymasterDevice 的 StrongBox 实现，并且为 StrongBox IKeymasterDevice 提供了一个新的、可用于设备锁定时间的 MACed 时间戳，以便将来在尝试使用 UNLOCKED_DEVICE_REQUIRED 密钥进行操作时与 auth tokens 进行比较。除非 auth token 的时间戳比 verificationToken 中的时间戳更新，否则设备仍然要被锁定。

说明

如果 StrongBox IKeymasterDevice 接收到一个比上次接收的 deviceLocked()调用中的时间戳少，且带有 verification token 时间戳的 deviceLocked()的调用，就必须忽略新的时间戳。TEE IKeymasterDevice 实现将收到一个空的 verificationToken（零值和空向量），并且使用自己的时钟作为设备锁定时间。

4.1.27 earlyBootEnded

该函数在 Keymaster 4.1 中引入。

earlyBootEnded 函数由客户端调用，用于通知 IKeymasterDevice 设备已经结束早期启动，且带有 EARLY_BOOT_ONLY 标记的密钥不能再使用。

在调用此函数后，所有尝试使用 EARLY_BOOT_ONLY 密钥的操作都将被拒绝，并返回错误代码 Error::INVALID_KEY_BLOB。

4.2 历史功能

4.2.1 Keymaster 0

以下函数属于最初的 Keymaster 0 定义。它们出现在 Keymaster 1 结构 keymaster1_device_t 中。不过，Keymaster 1.0 中并没有实现这些函数，且它们的函数指针设为了 NULL。

- generate_keypair
- import_keypair
- get_keypair_public
- delete_keypair
- delete_all
- sign_data
- Verify_data

4.2.2 Keymaster 1

以下函数属于 Keymaster 1 定义，但在 Keymaster 2 中已被移除，一同被移除的还有上述 Keymaster 0 函数。

- get_supported_algorithms
- get_supported_block_modes
- get_supported_padding_modes
- get_supported_digests
- get_supported_import_formats
- get_supported_export_formats

4.2.3 Keymaster 2

Configure 函数属于 Keymaster 2 定义，但在 Keymaster 3 中已被移除，一同被移除的还有上述 Keymaster 1 函数。

4.3 附加功能

产线部署工具通过访问 Keymaster 将 Keybox 文件写入 EMMC RPMB 分区，因此 Keybox 操作功能也集成 Keymaster 中。

5 Keymaster 配置

5.1 使能 Keymaster HAL

下面以使能 keymaster HAL 3.0 接口为例进行介绍，其它版本类似。

1. 增加 HIDL 3.0 接口

编辑 device/sprd/XXXXXX/common/manifest.xml 文件，增加 HIDL 接口：

```
<hal format="hidl">
        <name>android.hardware.keymaster</name>
            <transport>hwbinder</transport>
<version>3.0</version>
          <interface>
                    <name>IKeymasterDevice</name>
                    <instance>default</instance>
</interface>
</hal>
```

2. 使能 Keymaster 3.0 Service

编辑 device/sprd/XXXXXX/common/DeviceCommon.mk，增加：

```
PRODUCT_PACKAGE += \
android.hardware.keymaster@3.0-impl    \
android.hardware.keymaster@3.0-service    \
```


说明

XXXXXX：代表芯片系列，如 sharkl3、sharkle、Pike2 及 Sharkl5 等。使能 Keymaster 4.0 方法类似，就不在此一一描述。

5.2 使能 TEE Keymaster

请确保配置文件优先定义 BOARD_TEE_CONFIG（BOARD_TEE_CONFIG := trusty），编辑 device/sprd/XXXXXX/common/security_feature.mk 文件，在 ifeq ($(strip $(BOARD_TEE_CONFIG)), trusty) 与 endif 之间增加粗体字体部分，用于使能 TEE Keymaster。

```
ifeq ($(strip $(BOARD_TEE_CONFIG)), trusty)
……
PRODUCT_PROPERTY_OVERRIDES += \
            ro.hardware.keystore=sprdtrusty

PRODUCT_PACKAGES += \
                keystore.sprdtrusty
endif
```

在 Android 11.0 中，修改 module/security/main.mk 文件如下。

如果系统定义了 TARGET_HAS_SOFT_KEYMASTER：＝true，将使用 google 原生的 android.hardware.keymaster@4.1-service，否则使用 TEE Keymaster（android.hardware.keymaster@4.1-unisoc.service）

```
# Keymaster HAL
ifneq ($(strip $(TARGET_HAS_SOFT_KEYMASTER)), true)
PRODUCT_PACKAGES += \
    android.hardware.keymaster@4.1-unisoc.service
else
# km 4.1 is for zebu/haps no TOS environment if TARGET_HAS_SOFT_KEYMASTER := true
PRODUCT_PACKAGES += \
    android.hardware.keymaster@4.1-service
endif
```

6 VTS 和 CTS 测试

为了验证 Keystore/Keymaster 的功能、性能及兼容性，google 设计了 CTS/VTS 等相关组件。

6.1 VTS 测试

VtsHalKeymasterV3.0Target 测试子项用来验证 Keystore/Keymaster 模块基本功能，使用 google 发行的 VTS 测试包，运行“run vts –m VtsHalKeymasterV3.0Target”命令即可启动 Keystore/Keymaster 模块测试。

VtsHalKeymasterV3.0Target 测试子项会分别对 ARM32/AARCH64 平台进行测试，Android 9.0 测试条目 108 条，需全部通过，测试耗时低于 5 分钟。

6.2 CTS 测试

CtsKeystoreTestCases 测试子项用来验证 Keystore/Keymaster 模块功能、性能及兼容性，使用 google 发行的 CTS 测试包，运行“run cts –m CtsKeystoreTestCases”命令即可启动 Keystore/Keymaster 模块测试。

CtsKeystoreTestCases 测试子项会分别对 ARM32/AARCH64 平台进行测试，Android 9.0 测试条目 1011 条，需全部通过，对于 AARCH64 系统来说，完成测试需要将近 1 小时。一般情况下，在 PTR2 验证阶段，项目需要进行超过 48 小时“Hotplug+CtsKeystoreTestCases”复合测试。

7 兼容性

- Keymaster 1

  Keymaster 1 HAL 与之前发布的 HAL（例如，Keymaster 0.2 和 0.3）不完全兼容。为了在运行 Android 5.0 及更早版本（采用旧版 Keymaster HAL）的设备上实现互用性，Keystore 提供了一个可通过调用现有硬件库来实现 Keymaster 1 HAL 的适配器，但最终仍不能提供 Keymaster 1 HAL 中的全部功能。特别是，它仅支持 RSA 和 ECDSA 算法，而且所有密钥授权强制执行都由该适配器在非安全域中进行。

- Keymaster 2

  通过移除 get_supported_*方法并允许 finish()方法接受输入，进一步简化了 HAL 接口。在可一次性获得所有输入的情况下，这样可以减少到 TEE 的往返次数，并简化 AEAD 解密的实现过程。

- Keymaster 3

  在 Android 8.0 中，Keymaster 3 从旧式 C 结构 HAL 转换到了从采用新的硬件接口定义语言(HIDL)的定义生成的 C++ HAL 接口。通过对生成的 IKeymasterDevice 类进行子类化，并实现纯虚方法创建了新式 HAL 实现。在此变更过程中，很多参数类型发生了变化，尽管这些类型和方法与旧的类型和 HAL 结构体方法具有一对一的对应关系。

8 常见问题

1. **问题描述：**OTA 升级后系统无法正常开机进 recovery 模式，报 KM_ERROR_INVALID_KEY_BLOB。

   分析及解决方法：确认 OTA 升级前后操作系统 PLATFORM_SECURITY_PATCH 信息是否正确，如果升级前系统 PLATFORM_SECURITY_PATCH (如:2019-05-05)大于升级后系统 PLATFORM_SECURITY_PATCH(如: 2019-04-05)，升级后系统就无法启动并报(KM_ERROR_INVALID_KEY_BLOB)错误，请为 OTA 升级后系统设置正确的 PLATFORM_SECURITY_PATCH（升级后系统 PLATFORM_SECURITY_PATCH 应大于等于初始系统 PLATFORM_SECURITY_PATCH）。

2. **问题描述：**OTA 升级后系统需要输入密码进入系统，VOLD 调用 Keymaster 功能报 KM_ERROR_KEY_REQUIRES_UPGRADE。

   分析及解决方法：确认 OTA 升级前后操作系统 PLATFORM_SECURITY_PATCH 信息是否正确，如果升级后系统 PLATFORM_SECURITY_PATCH (如:2019-05-05)大于升级前系统 PLATFORM_SECURITY_PATCH(如: 2019-04-05)，且 VOLD 模块缺少 Key 升级逻辑，升级后系统启动过程中(KM_ERROR_KEY_REQUIRES_UPGRADE)错误，请联系 VOLD 模块 owner 提供 PATCH 处理该问题。

3. **问题描述：**使用 OTA 方法进行版本回退，系统无法开机。

   分析及解决方法：Keymaster 认为该操作为危险操作，影响系统安全，在 Keymaster 运行过程中，会进行版本检查，禁止进行版本回退，详细描述可参考："https://source.android.com/security/keystore"。

4. **问题描述：**Keymaster service 启动失败，系统无法开机。

   分析及解决方法：此类问题原因较多，需要进一步查看系统 android log（极端情况下：无法开机导致系统反复重启，须采用将 logcat 重定向串口导出 android log）。通常情况下，可能是以下问题导致该问题，需要结合 android log 进行精确定位：

   - Keymaster service 对所依赖 android.hardware.keymaster@3.0-service.rc 文件是否有正确访问权限。
   - device/sprd/XXX/common/manifest_main.xml（XXX：表示具体平台，如 sharkl3/sharkle/pike2）文件是否定义 Keymaster 接口。
   - Keymaster HAL 访问的设备文件“/dev/trusty-ipc-dev0”是否存在。
   - TEE Keymaster 工作是否正常。

5. **问题描述：**Keystore 调用 Keymaster 无应答，导致 Android 定屏。

   分析及解决方法：一般情况下，此问题为 TEE Keymaster 异常所致，可通过 Kernel log 进一步确认 TEE Keymaster 异常原因。

6. **问题描述：**无法设置开机密码及图案锁。

   分析及解决方法：通常情况下，为 TEE Gatekeeper 调用 TEE Keymaster 接口无响应所致，可通过 kernel log 进一步确认 TEE Keymaster 异常原因。

7. **问题描述：**GTS(Google Mobile Services Test Suite)测试子项 com.google.android.gts.security.AttestationRootHostTest#testEcAttestationChain 和 com.google.android.gts.security.AttestationRootHostTest#testRsaAttestationChain 测试失败。

分析及解决方法：

- 若必现，请检查是否写入 Keybox 文件。
- 若概率性出现，需要提供 Keybox 原始文件给 SPRD 作进一步分析，一般情况下，写入 Keybox 不完整可导致该问题。

8. **问题描述：**CTS 测试子项 CtsKeystoreTestCases testKeyStore_LargeNumberOfKeysSupported_EC 和 CtsKeystoreTestCases testKeyStore_LargeNumberOfKeysSupported_RSA 测试失败。

   分析及解决方法：该问题为平台性能不足所致，需要进行系统性分析已确定性能瓶颈，通常可从 CryptoEngine 工作频率，系统主频，DDR 工作频率、EMMC 器件参数及系统调度入手。

9 参考文档

1. https://developer.android.com/training/articles/keystore
2. https://source.android.com/security/keystore/implementer-ref
3. https://source.android.com/security/keystore