由硬件支持的 Keystore

借助系统芯片 (SoC) 中提供的可信执行环境，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。寻求 Android 专用扩展程序的开发者应访问 android.security.keystore (https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.html)。

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 (HAL) 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。Android 6.0 中的 Keystore 在该 Keystore API 的基础上进行了扩展，能够提供更广泛的功能。

在 Android 6.0 中，Keystore 不仅增加了对称加密基元 (/security/keystore/features)（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。要了解详情，请参阅授权标记 (/security/keystore/tags)和函数 (/security/keystore/implementer-ref)页面。

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

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 (/security/keystore/attestation)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 (/security/keystore/version-binding)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。

在 Android 8.0 中，Keymaster 3 从旧式 C 结构硬件抽象层 (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。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。要了解详情，请参阅函数 (/security/keystore/implementer-ref)页面。

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 (/security/keystore/attestation#id-attestation)。 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。

在 Android 9 中，更新包括：

- 更新到 Keymaster 4 (https://android.googlesource.com/platform/hardware/interfaces/+/master/keymaster/4.0/)
- 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

术语库

下面简要介绍了各个 Keystore 组件及其关系。

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，包含在应用自己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 (https://android.googlesource.com/platform/system/security/+/refs/heads/master/keystore/binder/android/security/keystore/IKeystoreService.aidl) 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。（此名称未进行标准化，仅用于说明概念。）

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。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。

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）。

架构

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。 最终的架构如下所示：

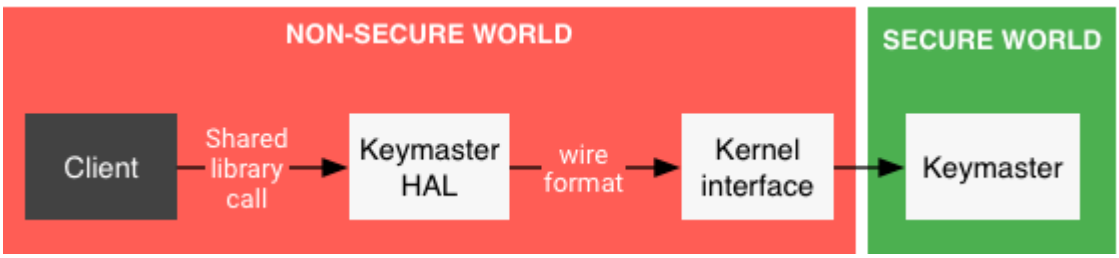

图 1. 访问 Keymaster

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），但在本文档中可以将其忽略。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。Android 开发者网站 (https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.html)对更高层 API 进行了介绍。

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。

与之前版本的兼容性

Keymaster 1 HAL 与之前发布的 HAL（例如，Keymaster 0.2 和 0.3）完全不兼容。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。

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。

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。大多数可信执行环境 (TEE) 实现人员应该都会发现 C++ 工具更加方便，因此本文档仅讨论 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，请参阅“参考”部分 (/reference/)。

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，由于 **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。请注意，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到 **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 keymaster0、keymaster1 或 keymaster2 HAL 的设备。