北京大学学报(自然科学版) 第 54 卷 第 6 期 2018 年 11 月
Acta Scientiarum Naturalium Universitatis Pekinensis, Vol. 54, No. 6 (Nov. 2018)
doi: 10.13209/j.0479-8023.2018.083

支持加壳应用的Android非侵入式重打包方法研究

黎桐辛[1] 韩心慧[1,†] 简容[1,2] 肖建国[1]

1. 北京大学计算机科学技术研究所, 北京 100871; 2. 北京航空航天大学, 北京 100083;
† 通信作者, E-mail: hanxinhui@pku.edu.cn

摘要 通过分析 Android 的应用特点, 提出一种新的 Android 重打包方法。该方法可以在不反编译、不修改原有应用代码的基础上, 实现对 Android 应用的重打包, 并支持主流加壳工具。该方法利用多种新的代码注入技术, 引入额外代码; 加载 Hook 框架, 提供代码修改能力; 最后动态修改应用行为, 实现应用重打包。实现了原型框架, 并通过实验, 验证了该框架在多个 Android 系统版本及多个加壳服务上的有效性。既证明了现有加壳技术的缺陷, 又可以用于对Android应用的动态调试、防御功能部署以及应用修改等。

关键词 Android; 重打包; 非侵入式; 加壳
中图分类号 TP317

Noninvasive Repackaging Method Research for Android Supporting Packed Apps

LI Tongxin[1], HAN Xinhui[1,†], JIAN Rong[1,2], XIAO Jianguo[1]

1. Institute of Computer Science and Technology, Peking University, Beijing 100871; 2. Beihang University, Beijing 100083; † Corresponding author, E-mail: hanxinhui@pku.edu.cn

Abstract The authors proposed a new Android repackaging method based on Android app characteristics. The new method can repackage apps without decompiling nor modifying the code and also supports packed apps. The method leverages multiple new code injection techniques to attach code to the app. Then, it adds a hook framework to provide capabilities to modify the code behaviors. Finally, the app's behaviors will be changed during runtime, thus the app is repackaged. A prototype framework is also implemented. The experiments demonstrate that the framework is compatible to different Android platforms and multiple packers. This research has proved that the current packing techniques have some flaws and the method can be used in dynamic code analysis, defense policies deployment and app modification.

Key words Android; repackaging method research; noninvasive; packed apps

随着移动生态的不断发展, 移动互联网与百姓日常生活的关系愈发紧密, 移动安全也更加重要, 相关技术不断涌现。因 Android 的开放性与绝对的市场份额, 其安全受到广泛关注。2017 年第一季度, Android 的市场份额达到 85%[1], 成为市场主流。同时 Android 生态中的安全威胁也在不断发展, 据报道, 2016 年累计截获 Android 病毒 1403.3 万[2]。

保障 Android 安全的一个重要技术是应用重打包技术, 该技术可以修改已有应用的代码与逻辑。由于 apktool[3]等开源工具的存在, Android 上重打包变得十分容易。

重打包技术广泛应用于防御中。例如, AutoPatchDroid 利用重打包技术来修补漏洞[4]; Aurasium 通过重打包技术来部署规则, 实现用户态的沙箱[5]; DroidMonitor 通过重打包技术插入监控代码, 进行动态分析[6]。但是, 重打包也能用于破解应用

国家自然科学基金(61772308)资助
收稿日期: 2018–01–15; 修回日期: 2018–03–16; 网络出版日期: 2018–11–08

和插入广告, 甚至注入恶意代码。2011 年, 21 款感染 DroidDream 的恶意应用出现在官方电子市场, 一些应用就是由其他应用重打包, 注入恶意代码而来[7]。重打包恶意应用泛滥的一个原因在于 Android 生态中有大量第三方电子市场, 但这些电子市场的监管不严格。

近年来, 为了对抗反编译及重打包, 出现 Android 混淆、抗反编译[8]以及加壳等技术, 其中加壳技术最具影响力, 且效果最佳。常见的加壳流程是将原有代码加密, 在应用运行过程中将其还原。由于反编译后无法看到原有应用代码, 仅可见壳部分代码, 且修改壳代码易引发异常, 因此, 加壳技术可以有效地阻止重打包。

目前, 已有大量应用使用加壳技术来保护其应用代码的安全。例如, 梆梆安全已服务于超过 70 万应用[9]。但是, 对于现有加壳技术对抗重打包是否存在缺陷还留有疑问。加壳技术虽然保护了应用, 但也增加了应用分析、病毒分析和防御策略部署等的难度。例如, Duan 等[10]收集了 9 万左右恶意应用, 其中 13.89%的恶意应用利用加壳服务来对抗分析。

面对加壳应用, 典型的重打包思路是先脱壳, 然后使用传统方式修改代码, 实现重打包。这种方法的局限在于, 需要用正确的脱壳技术, 提取出原有代码, 并能修复脱壳后的代码, 从而保证应用能够正常运行。另外, 反射、动态加载等代码操作也会影响传统重打包的效果。

针对这种情况, 本文提出一种新的支持加壳应用的重打包方法。该方法在不反编译、不修改原有应用代码的基础上, 通过注入额外代码, 引入 Hook 框架, 然后利用 Hook 相关函数修改应用行为, 实现非侵入式的应用重打包。该方法能在不脱壳的情况下, 对 Android 加壳应用进行重打包。通过分层设计, 提高了可扩展性。此方法可以用于动态分析、防御策略部署、应用修改等。本文在多款主流应用及加壳系统上进行测试, 证明此方法的有效性以及加壳程序在防重打包上的不足。

1 相关背景与研究

1.1 Android 应用结构

Android 应用是一个以 apk 为后缀名的 zip 压缩包, 其中包含 AndroidManifest.xml、代码文件、签名信息和资源文件。

1) AndroidManifest.xml 是应用的整体描述性文件, 定义包名、权限、应用组件和代码入口等。一个 Android 程序可以在 AndroidManifest 文件的 <application>标签中指定一个 Application 的子类作为应用的入口, 在应用启动时, 该类会被首先实例化并执行。因此, 可以将该类认为是应用的入口。

2) 代码文件包括 dex 文件和 so 文件。Dex 文件是由 Java 语言编译而成, 包含 Dalvik 字节码。so 文件通常由 c/c++编译而成, 以 ELF 文件的形式存在, 包含与处理器指令集相关的二进制代码。

3) 签名信息记录 zip 压缩包内各文件的哈希值, 并由开发者的私钥进行签名。利用工具, 可以验证应用的开发者信息。开发者也可以通过相关 API 验证签名是否改变, 得知应用是否被修改。

4) 资源文件保存在 resource.asrc、res 目录及 assets 目录下, 包括 xml 布局文件、字符串信息、图片和原始文件等。一部分加壳系统将壳的 so 代码和加密后的 dex 文件放在 assets 目录下, 应用启动时动态加载并还原。

Android 应用执行时, 其进程由 Zogyte 系统进程复制而来, 加载了 libc.so 等动态链接库、系统 framework 代码以及 Java 执行环境。在 Android 4.4 及以下版本, Java 执行环境是 Dalvik 虚拟机, 解释执行 dex 文件中的 Dalvik 字节码。从 Android 5.0 开始, Java 执行环境变为 ART 运行时, 会将 dex 文件首先编译成 oat 文件, 即从 Dalvik 字节码预编译为二进制指令。

Dalvik 虚拟机与 ART 运行时的一个区别在于是否支持多个 dex 文件: Dalvik 虚拟机不支持多个 dex, 需要利用 Multidex 库, 将其他 dex 动态加载至内存中; ART 运行时支持多个 dex, 能够将多个 dex 文件一起编译, 生成 oat 文件。

1.2 Android 重打包技术

由于 Android 应用大多以 Java 代码编写主要逻辑, 因此修改由 Java 代码生成的 dex 文件是 Android 重打包的关键。Baksmali/Smali[11]和 apktool[3]等工具可以方便地将 dex 文件反编译为文本格式保存的 Smali 代码。Smali 代码是 Dalvik 字节码的可读反编译表示, 与 Dalvik 字节码具有良好的对应关系。同时, 每一个 Smali 文件对应一个 Java 类, 定位和修改代码十分方便。利用 Baksmali/Smali 以及 apktool 还能将修改后的 Smali 文件编译为 dex。应用的重打包流程通常为: 1) 利用 Baksmali/Smali 或

apktool, 将 dex 反编译为 Smali 文件; 2) 修改 Smali 文件; 3) 编译生成 dex 和 apk; 4) 对 apk 进行重新签名。

此外, 重打包还可以修改资源文件, 常见的方法是利用 apktool 工具, 反编译 apk 文件, 然后替换或修改其中的 xml 文件和图片等, 再重新编译成 apk 并重签名。

1.3 Android 加壳技术

目前最常见的加壳方式为内存加密壳, 其基本原理是对原有 dex 进行加密和拆分, 保护应用的原始代码无法被反编译和修改。在应用运行中, 对 dex 进行还原, 让应用可以正常运行。多数情况下, 加壳后的应用会引入一个新的 Application 子类作为应用的入口, 确保壳代码作为应用的入口。然后, Application 子类调用壳的 so 代码, 在 native 层实现 dex 的还原。

加壳后, 应用的 AndroidManifest 文件与原有文件在应用入口和组件方面会出现差异。因此, 脱壳后需要修复 AndroidManifest 文件。此外, 壳可能有许多防御特性, 如反调试、多层次加密、完整验证和签名验证等。部分壳还将原有 Dalvik 代码直接转换为二进制的指令, 或者破坏 dex 的结构, 即使脱壳后也无法直接运行。因此, 脱壳后需要对 dex 文件做进一步的修复。

1.4 Android 脱壳相关研究

针对 Android 通用脱壳技术, 目前已有许多研究。DROIDUNPACK 通过修改 qemu 模拟器, 在应用执行过程中, 分析指令执行与内存操作, 提取解密后的代码[10]。KissKiss 通过 ptrace 操作, 搜索并 dump 内存中的 dex 文件[12]。Dexhunter 通过修改 Dalvik 虚拟机和 ART 运行时, 在解密后的 dex 加载至内存时还原 dex, 并写入磁盘[13]。AppSpear 选取合适的时机(如 MainActivity 启动后), 提取内存中的 Dalvik 数据结构(Dalvik Data Strut), 重构 dex 文件[14]。

然而, 随着加壳技术的发展, 部分通用脱壳技术已无法对某些壳进行处理[10], 如 Dexhunter 和 KissKiss。一些脱壳技术如 DROIDUNPACK 只能还原部分代码, 不能 dump 完整的 dex。因此, 先脱壳再重打包这种思路可能面临手工脱壳、脱壳失败、脱壳不完整以及 dex 需要修复等情况, 不能作为一种通用的方法。本文提出的重打包方案无需脱壳, 克服了上述缺陷。

同时, 现有通用脱壳技术虽然存在缺陷, 但可以还原部分应用代码, 如 DROIDUNPACK。这可以用于分析应用待修改函数, 然后利用本文提出的方法进行重打包, 并修改应用逻辑。

1.5 Android 重打包相关研究

在 Android 重打包方面, 目前的研究集中于 Android 重打包技术、检测 Android 重打包以及基于重打包进行应用的修补和防御等。

Apktool[3]和 Smali/Baksmali[11]是用得最多的重打包工具。FreeMarket 将 dex 转为 Java 字节码, 并修改原逻辑[15]。CodeInspect 支持基于 Jimple 语法修改原有应用[16]。但是, 这些工具都无法处理加壳应用, 而本文提出的方法可以重打包加壳应用。

许多研究利用重打包技术, 实现对应用的修改, 用于防御保护或动态分析。I-ARM-Droid[17]、DroidLogger[18]和 DroidDolphin[19]等技术利用重打包技术插入分析代码, 记录应用行为, 用于恶意行为检测等。Xu 等[5]、Chen 等[20]、Davis 等[21]以及李宇翔等[22]利用重打包技术向应用内插入代码, 用于策略检查以及访问控制。You 等[23]、Xie 等[4]以及 Azim 等[24]的研究利用重打包进行 Bug 及漏洞的修补。

许多研究关注对 Android 重打包的检测, 如 DroidMoss[25]、Juxapp[26]、WuKong[27]和 AnDarwin[28]等利用不同方法的代码相似性, 检查 Android 应用重打包。此外, DroidEagle[29]、View-Droid[30]以及 ResDroid[31]等也利用 UI 相似性进行重打包检测。

2 方法设计

本文的初衷是设计一种新的重打包方法, 并且支持主流加壳服务加固后的应用。其中存在许多挑战: 一方面, 一旦加壳, 应用的原始代码会被拆分、加密等多种方式保护, 在执行过程中被修复和执行, 因此依靠反编译再修改代码的传统重打包思路不再可行; 另一方面, 脱壳、修改代码再重打包, 看似直观可行, 但却依赖于能够完整正确地脱壳, 并且修复原有应用。事实上, 即使能够脱壳也需要额外处理如下情况。

1) 大多数壳会修改应用入口, 额外添加其他组件, 在脱壳后需要区分原有代码和壳代码, 并修复 AndroidManifest 文件中的相关内容。

2) 许多壳对原始代码实现多层次的解密流程[10], 一次脱壳也许无法完整地获取所有原始代码, 增加

了脱壳的难度。

3) 部分壳(如 360 加固)还将 dex 中的函数抽出, 用 native 代码实现, 脱壳后还需将相关函数修复后才能重打包。

因此, 脱壳后再重打包是一项复杂且不通用的方案。本文考虑用不脱壳的方式对应用进行重打包。不脱壳, 就意味着无法对原始代码直接进行静态修改, 只能通过动态的方式修改应用。同时, 重打包意味着注入额外代码至应用中, 但应用的原始代码已被保护, 壳引入的代码也可能存在复杂的保护机制。如何注入额外代码成为关键问题。

2.1 方法原则

在重打包方法设计中, 我们采用如下的原则。

1) 非侵入式与最小化修改原则: 重打包应用时, 不反编译修改原有的代码, 即不破坏原有 dex 与 so 文件, 而采取动态的方式修改应用逻辑。在该原则下, 重打包后的应用能尽可能地保持原有代码, 保障重打包后应用的正确执行。

2) 分层化与低耦合原则: 进行分层设计, 使不同模块完成不同的功能; 同时降低不同层次间的依赖性, 允许模块采取不同的技术或方案完成相同的功能。

2.2 总体设计

根据上述原则, 本文设计的重打包方法不反编译、不修改原有应用代码, 而是通过注入额外代码, 动态地修改应用行为。该方法不仅支持主流的加壳服务, 由于未对原有应用代码进行修改, 还能绕过其他反编译对抗技术, 如基于 apktool bug 的反编译对抗措施。

本文重打包方法的总体架构包含 3 个层次: 代码注入层、Hook 框架层与重打包插件层, 如图 1 所示。

2.2.1 代码注入层

代码注入层是重打包方法的难点与基础。该层能将额外的代码静态地注入应用中, 待执行应用时, 在进程空间内动态地修改应用行为。代码注入的整体方法是修改 AndroidManifest 文件, 同时添加额外的 dex 或 so 文件, 使得执行应用时既执行原有的代码(包括原应用代码和壳代码), 又能运行新注入的用于动态行为修改的代码。

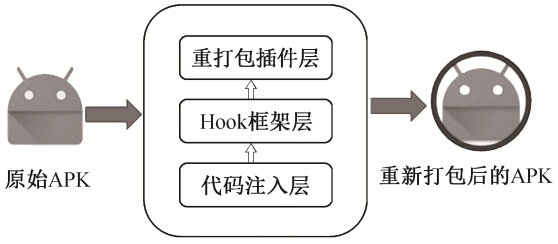

图 1 重打包方法总体架构

Fig. 1 Architecture of the repackaging method

该层的设计遵循非侵入式与最小化修改原则, 不破坏原有的 dex 与 so 文件, 仅修改 AndroidManifest 文件。本文共设计 4 种代码注入的方式, 以便支持不同的场景。由于方法的设计考虑了分层化与低耦合原则, 所以不同的代码注入方式不会对其他层的实现产生影响。同时, 为了避免代码注入层依赖 Hook 框架层, 代码注入层会直接利用 Java 原生 API, 绕过部分壳对应用的完整性检测。

2.2.2 Hook 框架层

Hook 框架层提供动态修改代码的能力。在代码注入后, 加载 Hook 框架层, 允许上层根据需求修改应用行为, 从而实现重打包。目前已有许多开源 Hook 框架或热补丁框架, 本文选取其中一种框架用于实现 Hook 框架层。

为了遵循分层化与低耦合原则, Hook 框架层会提供一套接口, 供重打包插件层使用。接口屏蔽了内部实现的细节, 允许未来更换其他 Hook 或热补丁框架, 以便兼容新版本的 Android 系统或提供更稳定的代码修改能力。

Hook 框架层的代码作为额外的 dex 和 so 重打包至应用的资源目录下, 当代码注入层添加的代码启动后, 可以动态地加载 Hook 框架层。这样能降低 Hook 框架层与代码注入层的耦合性。同时, 也可以动态地更新, 从网络下载新版本的 Hook 框架层来兼容新的设备和系统。

代码注入层添加的代码可以在任意时刻加载 Hook 框架层, 但最好的时机是原应用 Application 初始化之后, 原应用主 Activity 启动之前。此时, 壳代码已完成对应用代码的部分还原, 允许 Hook 框架根据重打包层的插件寻找待 Hook 的对象; 同时, 相关逻辑和原应用的主要逻辑还未被触发。

2.2.3 重打包插件层

重打包插件层是开发者利用 Hook 框架层的 API, 根据需求开发插件, 实现对应用行为的具体修改。开发者不仅可以 Hook 应用内部的 API, 还可以 Hook 系统 API。

由于采用分层化的设计, 虽然代码注入的方式

在重打包后已经固定，但 Hook 框架层和重打包插件层均可以实现动态更新，以兼容新的设备以及增加新的功能。

2.3　使用方式与方法优势

为了使用本文的方法进行应用重打包，开发者需要明确待 Hook 的函数，然后根据 Hook 框架层提供的 API，编写重打包插件。分析过程中，寻找需 Hook 的函数可以有以下几种方式。

1) 利用动态 Trace 的方式分析程序，了解内部的函数名。

2) 在动态调试环境下(如 xposed，frida 和 android jdb)遍历和搜索类名、函数名。

3) 进行不完善的脱壳操作，然后分析 dex。由于只用于静态分析，所获取的 dex 可以是只包含部分代码的 dex，也可以是有函数被隐藏在 native 代码中的 dex，无需考虑修复 dex。

4) 直接 Hook 系统中的 Framework API，避免分析应用代码。这是一种更通用的做法，在针对应用添加防御策略时，也推荐采用此方式。

确定需要 Hook 的函数后，根据 Hook 框架层接口，使用高层语言(如 Java)就可以开发属于自己的插件，修改程序逻辑。

从上述方法设计与使用方式可以看出，该框架具有以下优势。

1) 降低了重打包过程中对脱壳的依赖，开发者可以选择不完善的脱壳方式，也无需考虑修复脱壳后的应用。甚至开发者可以避免脱壳，只通过动态分析或是 Hook Framework API。

2) 以 Java 等高层语言编写重打包插件，无需像传统方式一样，用 Smali 语法修改程序逻辑。

3) 支持反射、动态加载等操作，这些操作传统重打包静态修改程序代码难以完成。

4) 具备对 VMP 壳的支持。由于框架不要求脱壳，且可以直接 Hook Framework API，因此也支持对使用 VMP 壳保护的应用进行重打包。

3　方法实现

3.1　术语约定

为方便后续论述的准确与简洁，我们做如下术语约定：1) 壳 dex/壳 so，指由加壳引入的 dex/so 文件；2) 原应用 dex/原应用 so，指加壳前的 dex/so 文件；3) 框架 dex/框架 so，指由重打包框架引入的 dex/so 文件；4) 框架 Application/壳 Application/原应用 Application，指在 AndroidManifest Application 标签中指定的 Application 类名，分别由框架/壳/原应用中的 dex 引入。

3.2　代码注入层的 4 种实现方式

为了不破坏原有 dex 和 so 文件，代码注入时，只能考虑注入新的 dex 或 so(即框架 dex/so)，其中的关键在于既能够让应用正常运行，又能运行新的 dex 或 so。我们设计了 4 种代码注入方式，其中两种用于注入 dex 文件，包括模拟壳机制以及利用 MultiDex 机制；另两种用于注入 so 文件，包括 so 的代理以及利用 Native Activity 机制。

本文均以加壳后的应用为例，介绍 4 种代码注入方式。这 4 种方案也适用于非加壳应用。

3.2.1　模拟壳机制

内存加密壳的典型流程是将原应用 dex 进行加密拆分，插入壳 dex 和壳 so。壳 Application 作为应用入口，在运行过程中还原原应用 dex，并正常执行。可以看到，如果不考虑对原应用 dex 的加密拆分操作，其流程正是我们期望的插入 dex 文件的一种方法。因此，可以模拟壳的机制，实现 dex 的插入。

在 AndroidManifest 文件中，加壳后应用的 Application 通常会被替换为壳 Application 来作为应用入口，确保壳 dex 首先执行。因此，模拟壳机制就是将框架 dex 作为应用的主 dex(classes.dex)修改 AndroidManifest 文件，将应用入口指向框架 Application。

然而，壳 dex 以及原应用 dex 中都可能存在自己的 Application 子类，是壳的入口以及原应用的入口，需要在应用启动时被调用。在未插入框架 dex 前，壳 Application 首先被实例化，然后还原原应用 dex，实例化原应用 Application，模拟 Framework 调用 Application 类的相关函数(如 attachBaseContext 函数)。

此外，Android 还提供 API 来获取 Application 的引用(如 Context.getApplicationContext)，应用空间内的默认 classloader 也指向壳 dex。因此壳代码还会将对 Application 的全局引用修改为原应用 Application，将 classloader 的全局引用修改为指向应用 dex 的 classloader。

因此，在模拟壳机制的方案中，除插入框架 dex 作为主 dex 以及修改 AndroidManifest 将应用入口指向框架 Application 外，框架 dex 也需要模拟壳

dex 完成相关操作。

重打包步骤如下。

1) 生成框架 dex。

2) 移动 classes.dex(壳 dex), 将框架 dex 重命名为 classes.dex。

3) 修改 AndroidManifest.xml, 将应用入口指向框架 Application。

4) 应用重新签名。

应用启动后, 框架 dex 完成如下操作。

1) 加载壳 dex。

2) 实例化壳 Application。

3) 将对 Application 的全局引用替换为对壳 Application 的引用。

4) 将 classloader 替换为指向壳 dex 的 classloader。

5) 模拟 Framework 调用壳 Application 的相关函数。

与后续方案相比, 此方案更加复杂, 且容易与壳本身的行为相冲突。但是, 该方案没有 Android 版本要求, 且由于模拟了加壳的原理, 对非加壳程序的适用性好。

3.2.2 对 MultiDex 机制的利用

对于 Android 4.4 及以下版本, Android 系统只支持一个应用包含一个 dex, 即 classes.dex。然而, 一个 dex 只能包含 65535 个函数, 限制了 Android 应用的大小。对此, Google 提出一种解决方案, 即 android-support-multidex.jar。开发者需要引入该 jar, 并使 Application 继承 MultiDexApplication 或在 Application 初始化时调用 Multidex.install API。对于 Android 5.0 及以上版本, Android 系统原生支持多个 dex。在 ART 运行时转换 apk 至 oat 的过程中, 所有的 dex (classes.dex, classes2.dex….)会被转换至同一个 oat 文件中。

对于 Android 5.0 以后的系统, 利用 MultiDex 机制, 可以采用一种更简单的方法, 插入 dex 至应用中。具体步骤如下。

1) 生成一个 Application 类(框架 Application), 继承 AndroidManifest 中声明的 Application (如壳 Application), 重写 attachBaseContext 函数, 实现自己的功能, 并调用父类的 attachBaseContext 函数。

2) 生成框架 dex, 将 dex 重命名为 classes (N+1).dex (N为应用中的 dex 数量), 然后插入原有应用。

3) 修改 AndroidManifest 文件, 将 Application 设置为框架 Application。

4) 对 APK 进行重新签名。

此方案的优点在于框架 Application 类继承了壳 Application, 而非重新加载壳 dex, 并实例化壳 Application。这是模拟壳机制无法完成的, 因为框架 dex 加载先于壳 dex, 若存在继承, 将出现异常 ClassNotFoundException。在 Multidex 机制中, 框架 dex 和壳 dex 的加载则是同时进行的。Framework 在回调框架 Application 时, 相当于回调了壳 Application。其他操作, 如对 Application 全局引用的处理及对 classloader 全局引用的处理, 都由壳 Application 完成。

此方案既简单, 又对壳的兼容性好, 缺点是只支持 Android 5.0 以后的系统。但是, 根据 Android 官网统计, 目前 Android 4.4 及以下版本仅占 20% 的市场份额, 未来比例将更低[32]。

3.2.3 so 代理机制

除直接注入 dex 文件外, 还可以直接注入 so 文件。因为壳通常需要直接操作内存, 并保护原应用 dex 的还原过程, 所以壳的许多逻辑都在 so 文件中实现。未加壳应用经常也包含 so 文件, 用于保护某些操作或提高效率。

与传统的二进制程序不同, Android 上 so 对 Java 层提供接口并不是通过导出函数表来实现, 而是通过 JNI 实现。Android 提供了 registerNative 接口, 实现了 Java JNI 接口到 so 中 JNI 函数的映射。因此, 一个 so 文件即使被重命名, 只要加载成功, 其 JNI 接口仍可以正确地在系统中注册。当 so 文件被加载时, JNI_onLoad 函数将被调用, 这个函数适合调用 API 来注册 JNI 函数。同时, so 的 JNI 函数还可以通过反射, 调用 Java 的 API。

因此, 我们设计了一种基于 so 代理的方案, 实现 so 文件的注入。具体地, 假设待替换的壳 so 名为 lib.so, 则将框架 so 重命名为 lib.so, 将壳 so 重命名为 lib-rename.so。框架 so 会在 JNI_onLoad 中利用反射机制, 通过 Java API 再次加载壳 so (lib-rename.so)。so 代理方案的步骤如下。

1) 生成一个 so, 在 JNI_onLoad 中利用 Java API 加载 lib-rename.so。

2) 将 APK 中的壳 so 重命名为 lib-rename. so。

3) 将框架 so 重命名为 lib.so (壳 so 的原名), 并放入 APK 中。

4) 对 APK 进行重新签名。

此方案的优点在于无 Android 版本要求，兼容性好，缺点是依赖 so 的存在。对于加壳应用，由于常见壳均包含 so，且 so 的加载时刻早，所以此缺点对加壳应用无影响。对于非加壳应用，so 可能不存在或加载时刻晚，需要根据不同的应用而定。

3.2.4　Native Activity 机制

Android 应用可以在 AndroidManifest 文件中声明一个 Activity 是 Native Activity，并指明对应的 so。当 Activity 需要启动时，Android Framework 会加载 so，并运行相关代码来展示 UI 界面。

在 so 代理方案中，框架 so 的加载依赖于应用对 so 的调用，而基于 Native Activity 的机制，可以强制要求应用加载框架 so。具体步骤如下。

1) 生成框架 so，实现 Native Activity 的代码逻辑，并自动调用应用的 Main Activity。将 so 放入 APK。

2) 修改 AndroidManifest 文件，更改原 Main Activity 属性，不再作为 Main Activity。

3) 修改 AndroidManifest 文件，注册一个新的 Activity，将其声明为 Main Activity，并在属性中注明该 Activity 为一个 Native Activity，且实现 Native Activity 的 so 为框架 so。

4) 对 APK 重新进行签名。

以后，每次应用被打开时，首先启动 Native Activity，加载框架 so，然后启动原 Main Activity。

此方案的优点是无 Android 版本要求，不依赖于应用对 so 的加载，而是主动加载框架 so。缺点是框架 so 的加载时刻晚，所以若应用在 Application 实例化过程中存在完整性验证，则可能失败。另一个缺点是 Android 应用存在多个组件入口(Activity 和 Service 等)，Main Activity 不一定是第一个启动的组件。因此，该方案适用于对应用的动态分析，而不适用于对应用修改功能后的发布。

3.2.5　4 种方案的对比

表 1 对比 4 种方案的优劣，包括注入的对象、平台适用范围、修改的文件、重命名的文件以及启动的时间等。

由于部分壳及应用检测了应用是否被修改，因此注入对象的启动时机决定框架是否有机会绕过应用完整性保护。模拟壳机制、Multidex 机制以及 so 代理机制启动时机一般早于应用完整性检测，而 Native Activity 机制的启动时机稍晚。

3.3　应用完整性保护绕过

当一个应用被第三方修改时，必然出现被修改的文件，并需要重新签名。因此，常见的应用完整性保护方式包括签名的验证及文件的完整性检查。

虽然可以直接利用 Hook 框架层绕过应用完整性保护，但将增加对 Hook 框架层的依赖。为此，我们基于 Java 的原生机制(反射调用及动态代理)来绕过应用完整性保护。Java 语言的动态代理机制允许对一个接口的实例生成代理对象，代理对象提供访问原始对象 API 的能力，但是可以修改请求和返回结果。

3.3.1　绕过签名验证

Android 提供 API 来获取特定应用的签名信息。图 2 是获取签名的常见方法。一般而言，壳在调用相关 API 时都是通过 native 代码反射调用 Java API。如要直接修改签名验证逻辑，则需要在 native 代码中准确定位代码并修改。

本文使用的方法是修改 PackageManager 的 getPackageInfo API。由于 PackageManager 是一个接口类型，且对象会在内存空间中被缓存，每次调用 context.getPackagemanager，实际上是返回缓存后的对象，因此，可以利用 Java 的动态代理机制，生成 PackageManager 的代理对象，并且替换原先的对象。每次通过代理对象调用 getPackageInfo 时，修改返回结果中的 signatures 属性，将其设置为未重打包前的签名。

3.3.2　绕过文件完整性检查

不同于签名验证，Android 未提供 API 来实现

表 1　不同代码注入方案比较

Table 1　Comparisons among different code injection techniques

方案	注入对象	Android 平台	修改文件	重命名文件	启动时机
模拟壳机制	dex	所有	无	classes.dex	Application 实例化
Multidex 机制	dex	> 4.4	AndroidManifset	无	Application 实例化
so 代理	so	所有	无	壳/原应用 so	壳/原应用 so 加载
Native Activity	so	所有	AndroidManifset	无	Activity 启动

```
PackageManager manager = context.getPackageManager( ) ;
PackageInfo packageInfo = manager.getPackageInfo
    (packageName, PackageManager.GET_SIGNATURES) ;
Signature[ ] signs = packageInfo.signatures ;
Signature sign = signs[0] ;
```

图 2 签名获取方法
Fig. 2 A method to get signature

文件完整性的检查。虽然不同的应用/壳可以实现不同的检查逻辑，但是获取应用 APK 的原始保存路径是必要的操作步骤，可以通过 PackageInfo 的 sourceDir 属性，以及 Context.getPackageCodePath 函数(实际上调用了 LoadedApk 的 getAppDir 函数)来实现。

对于 PackageInfo，可以通过构造 PackageManager 的代理对象来更改返回结果。LoadedApk 不是一个接口类，因此，代理对象并不合适。但是，LoadedApk 的 getAppDir 函数每次返回的其实是 LoadedApk 的一个属性，这个属性可以通过 Java 的反射方法直接修改。

通过上述方式，可以首先构造或释放一个未重打包前的 APK，然后当应用尝试获取 APK 保存路径时，返回未重打包前 APK 的路径，从而绕过文件完整性检查。

3.4 Hook 框架层实现

代码注入层之上是 Hook 框架层，提供函数 Hook 能力，用于动态修改应用的行为。目前 Hook 框架百花齐放，从需要 Root 权限的 Xposed[33]和 Frida[34]，到非 Root 框架 Legend[35]和 YAHFA[36]。此外，热修补框架 Sophix[37]、Tinker[38]和 Amigo[39]等也提供动态修改应用行为的能力。

本文未重新设计 Hook 框架，而是从现有框架中选择合适的作为本层的原型实现。重打包框架并不限制使用固定的 Hook 框架，Hook 框架层的作用在于提供代码动态修改能力，因此，任何合适的无需 Root 的动态代码修改框架都可以应用在本层。Android 的运行环境分为 Dalvik 虚拟机和 ART 运行时。Dalvik 下的 Hook 框架比较成熟，典型的有 Dexposed[40]。对于 ART，由于 Android 版本变化以及厂商定制，碎片化比较严重，对 Hook 框架的兼容性要求较高，例如，Legend 目前只支持 6.0.1 以下的版本。

对于 ART 运行时的动态代码修改，从原理来看，包括 ART Method 的整体替换(如 Sophix)、ART Method 中代码入口的替换(如 YAHFA)、函数 entrypoint 所指代码的直接修改[41]、基于虚函数表分发的 hook[42]以及利用类查找机制的类替换(如 Tinker 和 Sophix)。不同的原理在兼容性和易用性方面不同。从兼容性来看，类替换>ART Method 整体替换>其他方案。类替换要求有类的完整实现，此方法会增大框架的使用要求(需要对提取类的完整实现)。Sophix 使用的是 ART Method 整体替换，兼容性好，但无法调用原函数。其他方案受系统版本影响大。

本文选取基于 Sophix 原理的 AndroidMethodHook 框架[43]作为原型系统中 Hook 框架层的实现。该框架支持 Dalvik 和 ART 环境，利用 ART Method 的整体替换，并且支持调用原函数。

事实上，直接使用 Sophix 也是一个较好的选择。由于 Android 框架源码是开源的，若考虑修改系统函数，可直接使用 Sophix 的函数替换能力。同时，若通过反编译或部分脱壳得到应用中的待 Hook 函数，也可以使用 Sophix 的函数替换能力。若得到某一个类的实现，还可直接使用类替换的方式，实现代码的动态修改。

对于 Hook 框架层的接口 API，本文采用类似 Xposed 的接口形式。多个 Hook 框架采用类似的接口，包括 Dexposed, AndroidMethodHook 和 epic[44]等。

4 实验

本文选取 5 种常用的壳加固后的应用：360 加固保、腾讯乐固、阿里聚安全加固、爱加密和梆梆加密。针对这 5 种应用，对不同代码注入方案的有效性和 Hook 系统函数和应用函数的效果进行测试。

4.1 样本选取与测试

本文选取的样本均为知名厂商的最新应用(截至 2017 年 12 月 12 日)，因此可以认为这些样本能够极大地体现壳的保护力度与最新特性，样本信息如表 2 所示。

通过分析样本，可以得到壳的保护措施：签名验证、DEX 完整性保护以及文件完整性保护。签名验证指壳会检查签名的更改，拒绝重打包应用的运行。DEX 完整性保护指对壳 DEX 反编译、修改、再重新编译的过程会出现异常，或重打包后无法正常运行。文件完整性保护指替换文件或增加文

件后, 应用无法正常运行。从表 3 可以看出, 360 加固保和爱加密包含签名验证, 360 加固保、腾讯乐固和阿里聚安全加固都包含 DEX 完整性保护, 爱加密和梆梆加密包含文件完整性保护。

由于不修改任何 DEX, 利用框架重打包后的应用可以绕过 DEX 完整性保护。同时, 基于反射和动态代理的应用完整性绕过技术可以对抗签名验证和文件完整性这两种保护措施。

4.2　代码注入层测试

本文实验在 Android 4.4.4 (Nexus 5), Android 5.1.1 (Nexus 5), Android 6.0.1 (Nexus 6)和 Android 7.1.2 (Nexus 5X)上进行测试, 验证模拟壳机制、MultiDex 机制、so 代理机制和 Native Activity 机制这 4 种代码注入方案的有效性。之所以选择 Android 4.4.4~7.1.2 版本, 是因为这些版本是目前主流的系统版本, Android 8.0 目前市场占有率低, 且 Android Hook 框架支持能力弱。

实验结果如表 4 所示, 每一个壳都存在至少两种可行的代码注入方案, 证明了代码注入层的有效性。每一种代码注入方案都在至少两种壳上有效, 证明了代码注入具体方案的有效性。

So 代理机制在所有壳和所有平台上都有效; Native Activity 机制在不存在签名验证和文件完整性保护的腾讯乐固和阿里聚安全上有效; MultiDex 在除梆梆加密外的其他壳上和 Android 5 及以上的平台上有效; 模拟壳机制在全平台支持爱加密和梆梆, Android 5 及以上的版本支持 360 加固保。

从失败原因上分析, Native Activity 方案中注入的代码无法运行在壳保护代码之前, 所以无法绕过签名验证或文件完整性保护。MultiDex 机制只支持 Android 5 及以上的版本。梆梆加密在壳 Application 类初始化时就加载 so 进行文件完整性验证, 而 MultiDex 机制需要继承壳 Application, 壳 Application 的类初始化早于注入的代码, 所以 MultiDex 在梆梆加密上未能成功。模拟壳机制在部分壳和部分平台失败的原因在于与壳本身的操作存在冲突。

4.3　框架完整测试

在代码注入测试的基础上, 进行了完整的框架测试。使用每一种可行的代码注入方案作为代码注入层的实现, 利用 Hook 框架 AndroidMethodHook 作为 Hook 框架层, 然后 Hook 系统函数(如 Activity.onResume)以及应用自身代码(例如 MainActivity.onCreate)作为重打包插件层。将重打包后的应用运行于 Android 多个平台上, 验证能否实现对系统函数或应用代码的修改, 作为对框架的完整测试。

测试结果见表 5。可以看到, 除 Android 5.1.1 上的梆梆加密外, 在所有的平台和所有的壳上, 都能 Hook 应用代码和系统函数。但是, 由于 AndroidMethodHook 本身的 Bug, 应用被梆梆加密后, 在 Android 5.1.1 上, 重打包后的应用可以 Hook 应用代码, 而不能 Hook 系统函数。这个 Bug 可以通过更换 Hook 框架来规避。

实验证明了框架的有效性: 无需脱壳, 可以利用代码注入和 Hook 能力, 动态修改系统函数及应用代码, 实现加壳应用的行为修改。随着 Hook 框架的不断发展, 选择更合适的 Hook 框架作为 Hook 层的实现, 会让重打包框架有更好的效果。

同时, 从实验结果可以看出, 目前这些加壳服务在防止重打包的能力上仍十分薄弱, 需要进一步加强。

5　讨论

5.1　对探测代码注入的讨论

新重打包方法利用代码注入并加载了 Hook 框

表 2　加壳样本信息

Table 2　Packed app samples

加壳服务	应用名称	包名	版本号	更新时间	MD5
360 加固保	360 超级 root	com.qihoo.permmgr	8.1.0.0	2017.11.28	898f3956c378cd3a97c0cc6c2687f776
腾讯乐固	腾讯课堂	com.tencent.edu	3.14.0.11	2017.11.21	ff04f906e7ff4416c754626af0b30cc3
阿里聚安全	墨迹天气	com.moji.mjweather	7.0204.02	2017.12.5	de95abb9611e4ccf914934e141183137
爱加密	顺丰速运	com.sf.activity	8.9.4	2017.12.8	d4a335502c3de931a94f0ef950c58b00
梆梆加密	(移动)和生活	com.whty.wicity.china	4.4.0	2017.9.28	ace18dba14abad773e4aca164858b1ae

表 3 壳的保护措施

Table 3 Protections in packers

加壳服务	签名验证	DEX 完整性保护	文件完整性保护
360 加固保	√	√	×
腾讯乐固	×	√	×
阿里聚安全	×	√	×
爱加密	√	×	√
梆梆加密	×	×	√

说明: √表示有保护, ×表示无保护。

表 4 不同代码注入方案在不同平台和不同壳上的有效性验证

Table 4 Effectiveness of different code injection techniques for different platforms and packers

加壳服务	4.4.4	5.1.1	6.0.1	7.1.2
360 加固	3	1, 2, 3	1, 2, 3	1, 2, 3
腾讯乐固	3, 4	2, 3, 4	2, 3, 4	2, 3, 4
阿里聚安全	3, 4	2, 3, 4	2, 3, 4	2, 3, 4
爱加密	1, 3	1, 2, 3	1, 2, 3	1, 2, 3
梆梆加密	1, 3	1, 3	1, 3	1, 3

说明: 1 代表模拟壳机制, 2 代表 multidex 机制, 3 代表 so 代理机制, 4 代表 native activity 机制。

表 5 框架完整测试

Table 5 Complete experiments for the framework

加壳服务	4.4.4	5.1.1	6.0.1	7.1.2
360 加固	1, 2	1, 2	1, 2	1, 2
腾讯乐固	1, 2	1, 2	1, 2	1, 2
阿里聚安全	1, 2	1, 2	1, 2	1, 2
爱加密	1, 2	1, 2	1, 2	1, 2
梆梆加密	1, 2	2	1, 2	1, 2

说明: 1 代表成功 Hook 系统函数, 2 代表成功 Hook 应用代码。

架。检测代码注入和 Hook 框架看似一种可能的防御方法。但是从以上实验结果来看, 主流的几款壳服务都无法抵御我们的新型重打包方法。因此, 它们并未检测代码注入和 Hook 框架, 或检测失败。

从原理来看, 若壳代码想检测代码注入和 Hook 框架, 可以从静态和动态两方面进行。

静态方面, 可以检测应用安装包是否包含外部代码或已知 Hook 框架。这实际上属于应用的完整性检测。本文实验表明, 梆梆加密和爱加密等壳的应用完整性均可被绕过。因此, 加壳服务商还需改进现有的文件完整性检测方法。

动态方面, 壳代码可以分析内存布局和类加载信息, 尝试发现外部代码和 Hook 框架, 或是分析 Hook 行为。但是, 由于 Android 的定制化和碎片化, 不同设备上的系统类存在差异, 同时, 原应用也可能动态地加载外部代码。对于壳代码而言, 从内存中区分重打包框架注入的外部代码是一件复杂的事情。在已知 Hook 框架特征的情况下, 壳代码可以检测 Hook 框架。但是, Hook 框架在不断地发展和更新, 壳代码需要不停地维护新的 Hook 框架特征。然而, 我们可以修改 Hook 框架特征或使用新的 Hook 框架, 绕过壳代码的检测。此外, 应用开发者也使用 Hook 框架作为热修补框架, 用户也利用 Hook 框架增加设备的功能。壳代码检测 Hook 框架特征或行为时, 可能误判。

因此, 若想成功探测代码注入或 Hook 框架, 加壳服务商还需要克服许多难题。

5.2 对加壳服务商的建议

加壳服务商若想提高在防重打包上的能力, 需要改进应用完整性检测的保护方案。具体而言, 应避免直接使用 Android Framework API 来获取签名或应用的原始保存路径, 同时还需进行结果校验。任何使用 Android Framework API 的操作都有可能被拦截(Hook)并修改。获取签名时, 可以利用 native 代码和 Java 反射操作直接与 Binder 交互, 减少 API 的调用。获取应用原始保存路径时, 可以通过使用 native 代码和 Java 反射操作与 Binder 直接交互来获取 PackageInfo, 读取 APK 路径; 或是直接读取系统文件(如/proc/pid/maps 等), 并分析 APK 路径; 或是验证文件路径的前缀是否为系统目录, 避免被篡改。

6 总结

本文提出一种新的重打包方法, 可以无需反编译、不修改原有应用代码, 实现对 Android 应用的重打包。该方法支持主流壳加固后的应用, 且无需脱壳。同时, 本文提出 4 种不修改原有应用代码、实现代码注入的方法。通过实现原型框架及实验, 证明了新的重打包方法在多个主流加壳服务上的有效性, 也证明现有加壳服务仍存在缺陷。

参考文献

[1] IDC. Smartphone OS market share, 2017Q1 [EB/ OL].

(2017–05–01) [2017–12–24]. https://www.idc.com/promo/smartphone-market-share/os

[2] 360 互联网安全中心. 2016 年中国手机安全状况报告[EB/OL]. (2017–02–06) [2017–12–24]. http://zt.360.cn/1101061855.php?dtid=1101061451&did=490260073

[3] Winsniewski R. Apktool: a tool for reverse engineering android APK files [EB/OL]. (2017–09–21) [2017–12–24]. https://ibotpeaches.github.io/Apktool/

[4] Xie J, Fu X, Du X, et al. AutoPatchDroid: a framework for patching inter-app vulnerabilities in Android application // 2017 IEEE International Conference on Communications. Paris, 2017: 1–6

[5] Xu R, Anderson R. Aurasium: practical policy enforcement for Android applications // 21st Usenix Conference on Security Symposium. Bellevue, 2012: no. 27

[6] Liang D, Chen R, Sun H. DroidMonitor: a high-level programming model for dynamic API monitoring on Android // Proceedings of the 2014 International Conference on Network Security and Communication Engineering. Hong Kong, 2014: 93–96

[7] Balanza M, Abendan O, Alintanahin K, et al. DroidDreamLight lurks behind legitimate Android apps // 6th International Conference on Malicious and Unwanted Software. Fajardo, 2011: 73–78

[8] 梅瑞, 武学礼, 文伟平. 基于 Android 平台的代码保护技术研究. 信息网络安全, 2013(7): 10–15

[9] 梆梆安全. 关于梆梆安全[EB/OL]. (2016–10–12) [2017–12–24]. https://www.bangcle.com/abouts/index?select=first

[10] Duan Y, Zhang M, Bhaskar A V, et al. Things you may not know about Android (Un)Packers: a systematic study based on whole-system emulation // 25th Annual Network & Distributed System Security Symposium (NDSS). San Diego, 2018: 1–15

[11] Freke J. Smali/Baksmali [EB/OL]. (2017–10–31) [2017–12–12]. https://github.com/JesusFreke/smali

[12] Strazzere T. Android hacker protection level 0 // DEFCON 2014. Las Vegas, 2014

[13] Zhang Y, Luo X, Yin H. Dexhunter: toward extracting hidden code from packed android applications // 20th European Symposium on Research in Computer Security. Vienna, 2015: 293–311

[14] Yang W, Zhang Y, Li J, et al. AppSpear: bytecode decrypting and DEX reassembling for packed Android malware // 18th International Symposium on Research in Attacks, Intrusions and Defenses. Kyto, 2015: 359–381

[15] Daniel R, Shin E C R, Magrino T R, et al. FreeMarket: shopping for free in Android applications // 19th Annual Network & Distributed System Security Symposium (NDSS). San Diego, 2012: 1–16

[16] Rasthofer S. Codeinspect [EB/OL]. (2017–10–08) [2017–12–24]. https://codeinspect.sit.fraunhofer.de

[17] Davis B, Sanders B, Khodaverdian A, et al. I-ARM-Droid: a rewriting framework for in-app reference monitors for android applications // Mobile Security Technologies. San Francisco, 2012: 1–9

[18] Dai S, Wei T, Zou W. DroidLogger: reveal suspicious behavior of Android applications via instrumentation // 7th International Conference on Computing and Convergence Technology. Seoul, 2013: 550–555

[19] Wu W C, Hung S H. DroidDolphin: a dynamic Android malware detection framework using big data and machine learning // Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems. Towson, 2014: 247–252

[20] Chen K Z, Johnson N, D'Silva V, et al. Contextual policy enforcement in Android applications with permission event graphs // 20th Annual Network & Distributed System Security Symposium (NDSS). San Diego, 2013: 1–19

[21] Davis B, Chen H. RetroSkeleton: retrofitting android apps // Proceeding of the 11th annual international conference on Mobile systems, applications, and services. Taipei, 2013: 181–192

[22] 李宇翔, 林柏钢. 基于 Android 重打包的应用程序安全策略加固系统设计. 信息网络安全, 2014(1): 43–47

[23] You W, Liang B, Shi W, et al. Reference hijacking: patching, protecting and analyzing on unmodified and non-rooted android devices // Proceedings of the 38th International Conference on Software Engineering. Austin, 2017: 959–970

[24] Azim M T, Neamtiu I, Marvel L M. Towards self-healing smartphone software via automated patching // Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. Vsters, 2014: 623–628

[25] Zhou W, Zhou Y, Jiang X, et al. Detecting repackaged smartphone applications in third-party android mar-

ketplaces // 2nd ACM Conference on Data and Application Security and Privacy. San Antonio, 2012: 317–326

[26] Hanna S, Huang L, Wu E, et al. Juxtapp: a scalable system for detecting code reuse among Android applications // 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Heraklion, 2012: 62–81

[27] Wang H, Guo Y, Ma Z, et al. WuKong: a scalable and accurate two-phase approach to Android app clone detection // 2015 International Symposium on Software Testing and Analysis. Maryland, 2015: 71–82

[28] Crussell J, Gibler C, Chen H. AnDarwin: scalable detection of semantically similar Android applications // 18th European Symposium on Research in Computer Security. Berlin, 2013: 182–199

[29] Sun M, Li M, Lui J C S. DroidEagle: seamless detection of visually similar Android apps // 8th ACM Conference on Security & Privacy in Wireless and Mobile Network. New York, 2015: no. 9

[30] Zhang F, Huang H, Zhu S, et al. ViewDroid: towards obfuscation-resilient mobile application repackaging detection // Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks.Oxford, 2014: 25–36

[31] Shao Y, Luo X, Qian C, et al. Towards a scalable resource-driven approach for detecting repackaged Android applications // Proceedings of the 30th Annual Computer Security Applications Conference. New Orleans, 2014: 56–65

[32] Android. Android dashboards [EB/OL]. (2017–12–11) [2017–12–24]. https://developer.android.com/about/dashboards/index.html

[33] Rovo89. Xposed [EB/OL]. (2017–12–10) [2017–12–24]. https://github.com/rovo89/Xposed

[34] Frida. Frida [EB/OL]. (2017–09–29) [2017–12–24]. https://www.frida.re/docs/android/

[35] AsLody. Legend [EB/OL]. (2016–08–03) [2017–12–24]. https://github.com/asLody/legend

[36] Rk700. YAHFA [EB/OL]. (2017–12–08) [2017–12–24]. https://github.com/rk700/YAHFA

[37] 阿里云. 阿里云热修补[EB/OL]. (2017–09–04) [2017–12–24]. https://www.aliyun.com/product/hotfix

[38] Tencent. Tinker [EB/OL]. (2017–12–06) [2017–12–24]. https://github.com/Tencent/tinker

[39] Eleme. Amigo [EB/OL]. (2017–08–21) [2017–12–24]. https://github.com/eleme/Amigo

[40] Alibaba. Dexposed [EB/OL]. (2015–10–21) [2017–12–24]. https://github.com/alibaba/dexposed

[41] Wiβfeld M. ArtHook: callee-side method hook injectt ion on the new Android runtime ART [D]. Saarbrücken: Saarland University, 2015

[42] Costamagna V, Zheng C. ARTDroid: a virtual-method hooking framework on Android ART runtime // Proceedings of the Workshop on Innovations in Mobile Privacy and Security IMPS at ESSoS'16. London, 2016: 20–28

[43] Panhongwei. AndroidMethodHook [EB/OL]. (2017–11–03) [2017–12–24]. https://github.com/panhongwei/AndroidMethodHook

[44] Tiann. Epic [EB/OL]. (2017–12–21) [2017–12–24]. https://github.com/tiann/epic