学　　号　2015301500150
密　　级

武汉大学本科毕业论文

移动应用的动态行为捕获

院 (系) 名 称:　国家网络安全学院

专 业 名 称:　信息安全

学 生 姓 名:　蹇奇芮

指 导 教 师:　傅建明　教授

二〇一九年五月

BACHELOR'S DEGREE THESIS
OF WUHAN UNIVERSITY

Dynamic behavior capture for mobile applications

School (Department): School of Cyber Science and Engineering

Major: Information Security

Candidate: QiRui Jian

Supervisor: Prof. JianMing Fu

Wuhan University

May, 2019

郑 重 声 明

本人呈交的学位论文, 是在导师的指导下, 独立进行研究工作所取得的成果, 所有数据、图片资料真实可靠. 尽我所知, 除文中已经注明引用的内容外, 本学位论文的研究成果不包含他人享有著作权的内容. 对本论文所涉及的研究工作做出贡献的其他个人和集体, 均已在文中以明确的方式标明. 本学位论文的知识产权归属于培养单位.

本人签名: _____ 日期: _____

摘　　要

智能移动终端设备的盛行使得针对移动操作系统的恶意应用迅速增加. 目前的移动操作系统中, 安卓系统巨大的市场份额和其相对开放的应用分发和权限管理方式使得其成为的攻击者的主要目标. 为了识别出恶意应用并阻止其传播, 我们需要对应用的行为进行分析. 然而单纯的静态分析在如今安卓应用成熟的混淆和加壳机制的保护下无法很好的揭示应用的行为, 因此需要动态的对安卓应用的行为进行捕获和分析.

本文分析了目前已有的一些安卓系统动态分析系统, 并且通过 hook 技术以及对安卓 8.1 源代码的修改设计和实现了一个运行于 Nexus 5x(Google 的一款智能手机) 的高性能应用动态行为捕获系统. 该系统能够捕获到 Java 层的所有方法调用以及 Native 层的重要函数调用, 并且支持动态地调整需要监控的目标方法 (Java 层) 和函数 (Native 层). 本文使用常用应用对该系统进行了测试, 结果显示与同样能捕获到所有 java 层方法调用的 Android Device Monitor 相比本系统的性能开销明显更低.

本文设计思路结合了 hooking 技术带来的灵活性和以及修改源代码的稳定性以及高性能, 对其他开源平台的类似工具设计有一定参考作用, 但在应用中应当注意两种方式可能的冲突问题.

关键词: 安卓应用; 动态行为; 高性能

ABSTRACT

Malicious applications on mobile operating systems boom with the prevalence of smart mobile devices. The huge market share of Android, one of current mobile operation systems, and its relatively open application distribution and privilege management make it attackers' major target. In order to identify malicious applications and prevent them from spreading, we need to analyze the behavior of applications. However, only static analysis can not handle the mature obfuscation and packing mechanism of Android applications, so it is necessary to dynamically capture and analyze the behavior of Android apps.

In this paper, I analyze some existing Android dynamic analysis systems and present a high-performance application dynamic behavior capture system, which can run on Nexus 5x and is implemented by using hooking and modifying Android source code. The system captures all method invocations of Java layer and important function calls of Native layer, and supports dynamic adjustment of the target methods (Java layer) and functions (Native layer) that need monitoring. I evaluate the system with common applications and the result shows that the overhead is significantly lower than that of Android Device Monitor when capturing all java layer method invocations.

The design of the system in this paper combines the flexibility of hooking and the stability and high performance brought by modification of source code, which can be used for similar tools on other open source platforms, but we should pay attention to the

possible conflicts between the two approaches;

Key words: Android application; dynamic behavior; high performance

目　录

摘要 III

ABSTRACT IV

1 绪论 1

1.1 研究背景与意义 1

1.2 国内外研究现状和发展方向 2

1.3 论文主要工作 4

1.4 论文组织结构 4

2 背景技术分析 6

2.1 Android 系统架构 6

2.2 Android 应用结构 9

2.2.1 应用安装包结构 9

2.2.2 应用组织结构 11

2.3 Android 运行时环境 12

2.3.1 应用的启动 12

2.3.2 类的加载 14

2.3.3 方法的执行 16

2.4 Android 动态分析相关技术和实现工具 19

2.4.1 Virtual Machine Introspection 19

2.4.2 ptrace 系统调用 20

2.4.3 Application Instrumentation 20

2.4.4 DVM/ART Instrumentation 21

2.4.5 Hooking 技术 21
2.4.6 Frida 21
2.4.7 Xposed 22
2.4.8 Valgrind 22
2.5 Android 应用保护技术 22
3 系统设计实现 25
3.1 概览 25
3.2 监控应用启动 25
3.3 监控 Java 方法调用 27
3.4 监控本地函数调用 27
3.5 脱壳功能 27
3.6 日志系统 28
4 实验与结果分析 30
4.1 实验总体方案 30
4.2 实验运行环境 30
4.3 功能测试 30
4.3.1 监控应用启动功能测试 31
4.3.2 监控 Java 方法调用功能测试 31
4.3.3 脱壳功能测试 33
4.3.4 监控本地函数调用功能测试 34
4.4 性能测试 35
4.5 结果分析 36
5 总结与展望 37
5.1 论文工作总结 37
5.2 进一步工作展望 37
参考文献 38
致谢 42

1 绪论

1.1 研究背景与意义

随着移动互联网和物联网的蓬勃发展, 智能移动终端设备迅速普及。截止 2018 年 9 月, 国内智能手机用户数量已达到 7.8 亿[1], 占总人口数的 55% 以上。如此众多的用户极大地促进了移动应用的发展, 2018 年的数据显示[2], 谷歌公司的 Google Play 上已经有超 260 万应用软件, 苹果公司的 App Store 上也有超过 200 万应用软件可供下载使用。这些应用软件涵盖了娱乐, 社交, 购物, 出行, 金融服务, 身份服务等等领域, 极大地便利了人们的生活。但与此同时, 各种服务通过应用软件集中于智能手机使得智能手机与个人隐私, 财产安全甚至人身安全的联系变得更加紧密, 从而不可避免地吸引了大量攻击者开发和传播恶意应用来牟取不正当利用。

目前市场上的智能移动终端设备运行的移动操作系统几乎均为 Android 和 Apple iOS[3]。其中 Android 以其免费, 开源的特点吸引了大量智能手机厂商, 占据了超过 75% 的市场份额[3]. 最新数据显示, 在 2019 年 Q1 国内智能手机销售量中搭载 Android 的手机销量占比达 78.2%[4]。然而, Android 本身宽松的权限管理以及开放的应用分发方式使其很容易受到攻击, 加上巨大的市场体量, 造成了绝大多数恶意应用把 Android 作为攻击目标的局面。虽然近年来 Android 的权限管理和安全机制不断加强, 同时工信部各大应用分发平台的监管加强, 一定程度上遏制了恶意应用的发展, 但数据显示[5]2018 年 Android 新增恶意软件达 800.62 万个, 感染用户数近 1.13 亿, 数量仍然庞大。另外, 恶意软件的类型也持续朝着多样化隐秘化方向发展，新式的恶意软件通过更加难以分析的加壳和混淆技术隐藏自己的恶意行为, 绕过安全软件的查杀。因此, Android 平台的安全问题依然严峻。

为了阻止恶意应用被下载运行, 保护用户手机的信息安全, 各大应用分发平台需要能够精确有效地判断开发者提交的应用是否为恶意应用, 而捕获应用的行为是分析一个新应用是否为恶意应用的必要前提。对应用软件的行为获取方法有两

个大类: 静态方式和动态方式。静态方式即在不运行应用软件的情况下对应用软件内部的资源文件、代码、数据等进行分析, 获取应用的特征, 代码逻辑等; 动态方式则是运行应用软件, 在执行过程中对软件的代码执行路径, 数据访问等进行监控和记录。在目前 Android 平台的应用加壳和混淆技术成熟的情况下, 单独的静态分析无法获取包含应用的真正逻辑的代码和数据, 因而无法获取到应用行为, 必须通过动态的方式才能捕获到包含应用真实目的的行为, 获取相应的数据, 从而判断应用是否为恶意应用。另外, 动态分析还能够在运行中捕获到执行应用真正逻辑的代码和数据 (脱壳), 从而结合静态分析揭示更加完整的应用行为。因此, 对 Android 平台移动应用的动态行为捕获技术进行研究, 有助于识别和分析隐蔽性越来越强的恶意应用, 从而遏制恶意应用的传播, 提升 Android 平台的安全性。

1.2 国内外研究现状和发展方向

Android 系统从发布至今已有 10 年, 目前国内外已有许多 Android 应用动态分析相关的研究成果发表。这些成果借鉴了传统 PC 平台的动态分析方法, 并结合了 Android 平台的自身特点, 在本地指令层面, 系统调用层面, 本地函数层面, Java 指令层面, Java 方法层面中部分或全部层面对应用的运行进行跟踪记录, 并在此基础上结合污点传播技术实现了隐私数据泄露的检测功能, 结合对应用加壳混淆机制的研究实现了脱壳和去混淆功能, 给恶意应用的分析提供了许多强有力的工具。下文介绍了一些有代表性的成果。

Enck William 等构建了名为 Taintdroid[6] 的隐私数据跟踪系统。该系统采用了污点传播技术, 通过修改 Android 系统 Java 层与隐私数据获取相关的 API 给隐私数据添加标记, 通过修改 Android Runtime 的 Dalvik 虚拟机运行机制实现了带标记隐私数据在虚拟机内部的透明传播, 通过修改 Android 系统进程间通信的接口实现了带标记隐私数据跨进程传播, 通过修改 Java 层文件和网络的 API 实现记录带标记的隐私数据去向, 从而能够检测到应用泄露隐私数据的行为。不过该系统有以下局限性: 1. 没有对应用的所有敏感行为进行监控, 例如拨打电话, 发送短信等; 2. 没有对 Native 层的函数进行监控, 无法检测到应用通过 JNI 接口调用自身 Native 模块泄露隐私数据的行为; 3. 针对特定 Android 版本, 并且不再支持 Android4.3 以后的版本使用;

Desnos Anthony 等构建了名为 Droidbox 的[7] 动态分析系统。该系统使用了 Taintdroid[6] 来监控隐私数据泄露, 另外通过修改 Android 系统源代码中敏感 API 的方式实现对 Java 层的电话, 短信, 网络, 文件, Java 类动态加载, 加密等 API 调用的监控, 能够记录应用在 Java 层的敏感行为。之后为避免频繁修改系统以适应 Android 版本变化, 该系统更改了监控 Java 层敏感 API 调用的方式, 通过反编译需要监控的应用并在敏感 API 调用前插入监控代码, 再重新打包生成修改后的应用的方式实现监控, 并为将实现新的监控方案的工具命名为 APIMonitor[8]。但该系统只涉及了 Java 层预定义的敏感 API 的监控, 没有实现对应用自身的 Java 方法调用的监控, 并且没有实现对应用本地函数层调用的监控, 因此无法完整的揭示应用的行为; 另外该系统也针对特定 Android 版本开发, 并且不支持 Android4.1 之后的系统使用; 对于采用修改应用本身实现监控的 APIMonitor, 由于目前加壳和混淆以及应用完整性检查技术的成熟, 已经几乎失效。

Yan Lok Kwong 等构建了名为 DroidScope[9] 的动态分析系统。该系统通过修改运行 Android 系统的 qemu 虚拟机以及 Android 系统中的 Dalvik 虚拟机实现了了对本地指令层面和 Java 指令层面的监控追踪, 并在此基础上实现了污点传播分析数据泄露, 监控 Java 和本地次层敏感 API 调用等功能。该系统在底层实现了对应用运行的全面监控, 并提供了接口用以在特定事件 (例如执行系统调用, 执行本地函数, 读写内存等) 发生时添加自定义的处理逻辑, 可以用来开发特定用途 (例如脱壳) 的工具。但是该系统也有一些局限性: 1. 提供了对底层执行的监控功能因而性能开销较大; 2. 依赖于虚拟机环境, 受模拟器检测技术的影响, 恶意应用会在检测到模拟器运行环境时隐藏自己的恶意行为, 使得分析结果不准确 3. Java 部分的监控通过修改 Android 系统中 Dalvik 虚拟机来完成, 对特定 Android 版本有效, 然而目前 Android 系统已经使用 Art 虚拟机代替了 Dalvik 虚拟机, 该系统无法应用于目前的应用分析。

Tam Kimberly 等构建了名为 CopperDroid[10] 的动态分析系统。该系统基于 qemu 虚拟机, 通过 VMI 技术捕获应用运行时调用的系统调用序列及相应参数, 然后通过解析记录的系统调用序列和对应参数重建出应用在本地函数层面和 Java 方法层面的具体行为, 例如发送短信, 拨打电话, 进行网络传输, 进行文件读写, 启动进程, 进程间通信等。由于只需要系统调用序列, 该系统不需要对 Android 系统进行

修改, 能够较好的兼容 Android 版本的升级, 但仍由一些局限性: 1. 系统基于虚拟机环境, 受模拟器检测技术的影响; 2. 没有对 Java 层方法进行具体的监控, 会丢失掉一些 Java 层的行为; 3. 重构应用行为依赖于特定 Java 层行为与系统调用的映射关系, 而这个关系可能发生变化而使得结果不准确。

Xue Lei 等构建了名为 Malton[11] 的动态分析系统。该系统运行于真机上, 使用了 Valgrind 框架来获得指令级别的监控能力, 依靠 Valgrind 的 FunctionWrapper 机制来 hook 系统调用, 系统函数库和重要的 Android Runtime(ART) 函数。该系统利用 ART 会编译所有 Java 方法的机制, 收集 OAT 文件中 Java 方法对应的机器码的入口地址。通过每条机器执行前比较其地址是否为某个 Java 方法的地址, 该系统实现在机器执行层面准确的监控 Java 方法的调用。此外该系统还包括污点传播分析数据流, 执行路径扫描和分支强制执行等技术, 是一个完整有效的动态分析工具, 并且不受模拟器检测技术的影响。但在 Android7.1 以后 Android 系统不再在应用安装时编译所有 Java 方法而是根据应用的运行情况选择性的编译, 这对该系统会造成很大影响; 此外, Valgrind 框架的性能开销巨大, 会使得应用运行数十倍的变慢, 因此基于 Valgrind 框架实现的该系统性能开销很高。

文伟平等构建了一种动态监控系统[19]。该系统使用 LKM hook 系统调用来监控隐私文件读取, 基于 Netfilter 机制对应用程序的联网动作进行监测, 并利用 Xposed 框架监控应用层行为的动态行为（如发送短信, 拨打电话）的监控系统, 能够监控从应用层到内核层的大部分敏感行为。

1.3 论文主要工作

本文分析了目前已有的一些安卓系统应用行为监测系统的实现方式和优缺点, 并且通过 hook 技术以及对安卓 8.1 源代码的修改设计和实现了一个运行于 Nexus 5x(Google 的一款智能手机) 的高性能应用动态行为捕获系统. 该系统能够捕获到 Java 层的所有方法调用以及 Native 层的重要函数调用, 并且支持动态地调整需要监控的目标方法 (Java 层) 和函数 (Native 层). 本文使用常用应用对该系统进行了测试, 结果显示与同样能捕获到所有 java 层方法调用的 Android Device Monitor 相比本系统的性能开销明显更低.

1.4 论文组织结构

根据本文研究的特点, 本文的内容按如下方式组织:

第一章为绪论, 主要说明了本文课题的研究背景和研究意义, 简述了国内外对本文课题的研究成果及相关工具和技术, 介绍了本文的主要工作内容和文章组织结构。

第二章为背景技术介绍, 主要讲述了 Android 系统的基本架构, Android 应用的基本结构, Android 平台的常用动态分析技术和应用保护技术, Android Runtime 的运行机制, 以及本系统用到的一个 hook 框架—Frida。

第三章为系统设计和实现, 详细说明了本文提出的应用动态行为捕获系统的设计实现方案。

第四章为实验与结果分析, 描述了对本系统进行测试的实验环境, 实验方法并对实验结果进行了分析和总结。

第五章为总结与展望, 主要是整理本文所做的工作，并简要分析了本文提出系统的局限性和改进方案。

2　背景技术分析

2.1　Android 系统架构

Android 系统由多个软件层次构成, 这些层次功能分明, 每一层都对其上的一层提供服务, 构成一个 5 层的软件栈。软件栈最底层为 Linux 内核层, 其上为硬件抽象层, 之后为本地函数库层, 该层次包括了 Android Runtime 和其他的一些本地函数库, 再上层为 Android 框架层, 该层包括了提供给应用程序的 API 和系统管理服务程序, 最上层为应用层, 该层次为用户直接交互的应用程序运行的层次。图2.1给出了各层次的组件和关系。

图 2.1　Android 系统架构

Linux 内核层

Android 基于修改的 Linux 内核构建, Linux 内核为 Android 系统提供了操作系统的基本功能, 包括进程管理, 内存管理, 文件管理, 进程间通信 (共享内存和 Binder), 网络协议栈, 电源管理, 许多设备驱动程序 (音频, 视频, 蓝牙, 相机, 键盘, USB, WIFI 等) 以及访问控制机制 (基于用户和用户组的访问控制和 selinux)。这些功能通过系统调用的方式提供给上层使用, 因此, 监控系统调用的使用情况能够获取到应用对关键资源的访问行为。

硬件抽象层

硬件抽象层是定义了用于更高层次调用对应硬件驱动的接口, 屏蔽了不同厂商的同种设备驱动的差异, 降低 Android 系统与硬件的耦合度, 便于 Android 系统的移植。硬件抽象层包含多个库模块，其中每个模块都为特定类型的硬件组件实现一个接口，例如相机或蓝牙模块。当更高层次要求访问设备硬件时，Android 系统将为该硬件组件加载库模块。

本地库层

本地库层由许多由 C/C++ 开发的系统运行库组成。这些运行库主要分为两部分, 第一分部为 Android 运行时环境相关的库, 第二部分为其他系统运行库。

Android 运行时环境由给应用提供 Java 运行环境的虚拟机实现和实现 Java API 的核心运行时库组成。虚拟机实现在 Android4.4 之前为 Dalvik 虚拟机, Android4.4 时 ndroid Runtime(ART) 虚拟机作为实验特性加入, 并喝 Dalvik 虚拟机共存, Android5.0 之后只保留了 ART 虚拟机。Android 框架层的许多服务程序和应用层的应用软件就运行在自身的虚拟机实例中。核心运行时库，可提供 Java API 框架使用的 Java 编程语言大部分功能。

其他系统运行库包括许多重要的功能的实现, 主要包括以下部分: AUDIO MANAGER 用于管理音频输入输出; LIBC 提供了 c 语言标准函数库; MEDIA FRAMEWORK 提供了对常见音频和视频处理的支持; OPENGL/ES 提供了 2D/3D 图形绘制功能; SQLITE 提供了访问 SQLite 数据库的函数; SSL 提供了常见的加密功能;

SURFACE MANAGER 提供对显示子系统的支持和管理; WEBKIT 提供了浏览器引擎的实现。

本地库层的各种功能函数除了提供给 Android 系统自身以实现系统服务功能, 还可以通过 Android Native Development Kit(NDK) 让应用程序通过 Java Native Interface(JNI) 调用, 因此对本层函数调用情况的监控可以获取到应用的行为。

Android 框架层

Android 框架层包括了许多系统服务程序和组件以及提供给应用程序的访问系统资源的 Java API。这些服务程序和组件主要包括以下几部分:

1. 资源管理器，用于访问非代码资源，例如本地化的字符串、图形和布局文件

2. 通知管理器，可让所有应用在状态栏中显示自定义提醒

3. Activity 管理器，用于管理应用的生命周期，提供常见的导航返回栈

4. 内容提供程序，可让应用访问其他应用（例如“联系人”应用）中的数据或者共享其自己的数据

5. 丰富、可扩展的视图系统，可用以构建应用的 UI，包括列表、网格、文本框、按钮甚至可嵌入的网络浏览器

Android 框架层是与应用程序联系最紧密的层次, 也是应用程序最容易访问系统资源的层次, 因此对该层次提供的 API 的监控能够显示应用的主要行为。

应用层

应用层包括了所有用户直接使用的应用软件, 例如电话、短信、浏览器、微信、支付宝等。这些应用软件主要由 Java 语言开发, 通过调用 Android 框架层提供的 API 和 Java 语言的标准 API 实现功能, 每个应用运行于自己独立进程中的虚拟机实例中, 多个应用间借助框架层提供的 API 通信 (进程间通信最终由内核实现)。利用 NDK, 应用也可以实现自己的本地库, 访问 Android 系统本地库层的开放甚至隐藏的函数, 并通过 JNI 在应用的 Java 部分调用自身的本地库中的本地函数。由于应用能够直接执行本地代码, 增加了应用程序行为涉及的层次，需要同时在 Java 层次和本地层次监控应用的执行才能获取到应用的所有行为。

2.2 Android 应用结构

2.2.1 应用安装包结构

Android 应用程序主要以 Android Package(APK) 的文件形式分发和安装。APK 文件本质上是一种 zip 压缩文件, 由多个文件和文件夹组成其中包含了应用的代码文件、资源文件、证书文件和清单文件, 以 apk 作为文件后缀名。图2.2给出了 APK 文件的内部结构。

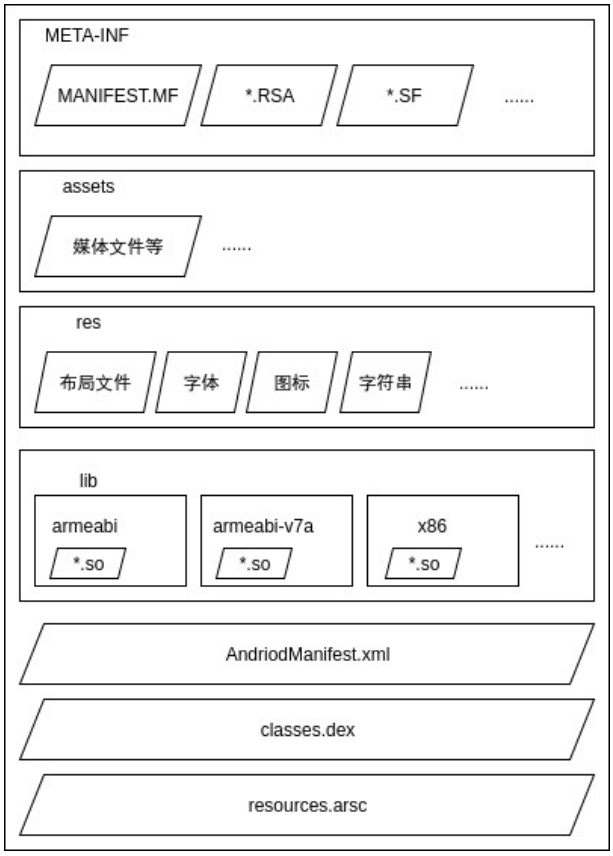

图 2.2 APK 文件结构

META-INF 文件夹包含了与 APK 文件的签名和校验相关的文件, 一般应该包括至少 3 个文件:MANIFEST.MF、*.RSA、*.SF(“*”表示文件名不确定)。其中 MANIFEST.MF 记录了 APK 中的所有文件名和经过 base64 编码的 SHA256 校验值(不包括自身); *.SF 记录了 MANIFEST.MF 文件的经过 base64 编码的 SHA256 校验值以及 MANIFEST.MF 中每一项记录的经过 base64 编码的 SHA256 校验值; *.RSA 文件中保存了公钥、所采用的加密算法以及对 CERT.SF 中的内容的用私钥进行加

密之后的值。

assets 文件夹包括了 res 中定义类型之外的其他类型的资源文件，例如音频文件，视频文件等媒体文件。该文件夹内的文件不会被编译处理, 因此可以放入任意格式的文件, 应用甚至可以把本地库文件放在这个文件里在运行时根据需要动态的加载。

res 文件夹内包含了除了字符串外其他较复杂资源文件, 例如布局文件, 字体文件, 图标文件, 颜色文件等。这些文件会在应用运行时根据 resources.arsc 中记录的资源 ID 对应的路径进行调用。

lib 文件夹包括了应用的本地库文件, 根据适用的 Application Binary Interface(ABI) 不同，这些本地库文件会被放在不同的子文件夹里。当 APK 被安装时, 系统会选择合适的本地库文件进行安装, 在启动应用时系统会自动加载对应的本地库文件。

AndroidManifest.xml 文件是重要的清单文件, 描述了应用的组件以及其他的一些配置, 通过该文件能够获取到应用的许多基本信息, 具体来说包含以下几个方面:

1. 记录了应用软件的名称, 该名称独一无二用于识别该应用;

2. 记录了应用的所有组件, 包括构成应用的 Activity、服务、广播接收器和内容提供程序, 以及这些组件可以处理的 Intent 消息;

3. 描述了应用需要使用的权限;

4. 声明了应用运行所需的最低 Android API 级别 (与 Android 版本相对应)

5. 列出应用必须链接到的本地库

classes.dex 文件为应用的 Java 代码编译后的运行于 ART 或者 Dalvik 虚拟机的可执行文件。该文件包含了应用自定义的类的实现代码, 通过反编译该文件可以得到应用的 Java 源代码, 因此通常会使用加壳和混淆的手段隐藏该文件真正的内容。对于大型应用, 可能会有多个 dex 文件, 命名为 classes2.dex、classes3.dex 等。

resources.arsc 文件为资源文件索引文件, 其本身包含了全局常量字符串以及其他较复杂资源的路径, 系统正是通过该文件来访问 res 中的其他资源文件的。通过修改该文件中其他资源对应的路径可以隐藏 res 文件夹, 从而隐藏其他资源文件。

2.2.2 应用组织结构

Android 应用的功能主要由四种组件实现, 这四种组件为 Activity、服务、内容提供程序、广播接收器。这些组件可能会存在相互依赖的情况，但每个组件都以独立实体形式存在，发挥特定作用, 能够被单独的调用。各组件的具体功能如下:

Activity 表示一个用户能够直接交互的界面。例如，电子邮件应用可能具有一个显示新电子邮件列表的 Activity、一个用于撰写电子邮件的 Activity 以及一个用于阅读电子邮件的 Activity。尽管这些 Activity 通过协作在电子邮件应用中形成了一种紧密结合的用户体验，但每一个 Activity 都独立于其他 Activity 而存在。因此，其他应用可以启动其中任何一个 Activity（如果电子邮件应用允许）。例如，相机应用可以启动电子邮件应用内用于撰写新电子邮件的 Activity，以便用户共享图片。

服务是一种在后台运行的组件，没有用户界面, 用户无法直接与之交互, 常用于执行长时间运行的操作或为远程进程执行作业。例如，当用户位于其他应用中时，服务可能在后台播放音乐或者通过网络获取数据，但不会阻断用户与 Activity 的交互。诸如 Activity 等其他组件可以启动服务，让其运行或与其绑定以便与其进行交互。

内容提供程序管理一组共享的应用数据。应用数据可以被存储在文件系统、SQLite 数据库、网络上或任何应用可以访问的永久性存储位置, 其他应用可以通过内容提供程序查询数据，甚至修改数据（如果内容提供程序允许）。例如，Android 系统可提供管理用户联系人信息的内容提供程序。因此，任何具有适当权限的应用都可以查询内容提供程序的某一部分（如 ContactsContract.Data）, 以读取和写入有关特定人员的信息。内容提供程序也适用于读取和写入您的应用不共享的私有数据。例如，记事本示例应用使用内容提供程序来保存笔记。

广播接收器是一种用于响应系统范围广播通知的组件。许多广播都是由系统发起的, 例如，通知屏幕已关闭、电池电量不足或已拍摄照片的广播。应用也可以发起广播, 例如，通知其他应用某些数据已下载至设备，并且可供其使用。尽管广播接收器不会显示用户界面，但它们可以创建状态栏通知，在发生广播事件时提醒用户。但广播接收器更常见的用途只是作为通向其他组件的“通道”，例如，它可以在接收到某个事件广播后启动一项服务来执行某项工作。

2.3 Android 运行时环境

Android 运行时环境本质上是一个虚拟机, 用于执行 Android 应用中 dex 文件内的字节码, Android 应用和许多 Android 系统服务都运行在 Android 运行时环境中。从 2008 年第一个 Android 版本发布至今, Android 运行时环境经历了许多变化, 其中最大的变化是从 Android4.4 前的 Dalvik 虚拟机变成了 Android5.0 之后的 ART 虚拟机。下面的内容将会根据 Android8.1 的 ART 虚拟机从应用启动、类的加载、方法的执行、三个方面分析应用在 Java 层执行流程。

2.3.1 应用的启动

Android 系统中有一个十分重要的进程叫做 Zygote, 大多数应用进程和系统进程都是由 Zygote 产生的。图2.3描述了 Zygote 进程的启动过程。

首先系统会启动/system/bin 目录下的本地程序 app_process, app_process 解析自身参数后若发现参数中有–zygote 就会将进程名改为 Zygote 然后调用 AndroidRuntime::start 启动 Android 运行时环境, 并把入口类设置为 com.android.internal.os.ZygoteInit。进入 AndroidRuntime 后, 会调用 startVM 启动 ART 虚拟机, 之后就会开始执行上述入口类 ZygoteInit 的 main 方法。该类的 main 方法中首先调用 registerZygoteSocket 注册一个 socket 用于之后从该 socket 接收创建应用进程的命令并执行, 然后会调用 preload 加载许多重要的类, 最后会调用 ZygoteServer.runselectLoop 进入一个死循环。至此, Zygote 进程就启动完成了。在 runselectLoop 的死循环中, 其会调用 acceptCommandPeer 等待创建应用的命令, 接收到命令后调用 ZygoteConnection.processOneCommand 使用 fork 机制来创建新进程, 所以 Zygote 在启动后会一直执行 runselectLoop 直到关机。

了解了 Zygote 进程的启动过程后, 下面是 Android 应用的启动过程。Android 应用是通过四大组件构成的, 这里通过启动一个 Activity 的过程来介绍应用的启动过程。图2.4和2.5描述了启动一个没有运行的应用的过程 (省略了一些不需要关心的调用)。

首先启动器调用 Activity 类的 startActivity 方法用于启动一个 Activity, 该方法又会调用 Instrumentation 类的 execStartActivity 方法。execStartActivity 通过 Binder

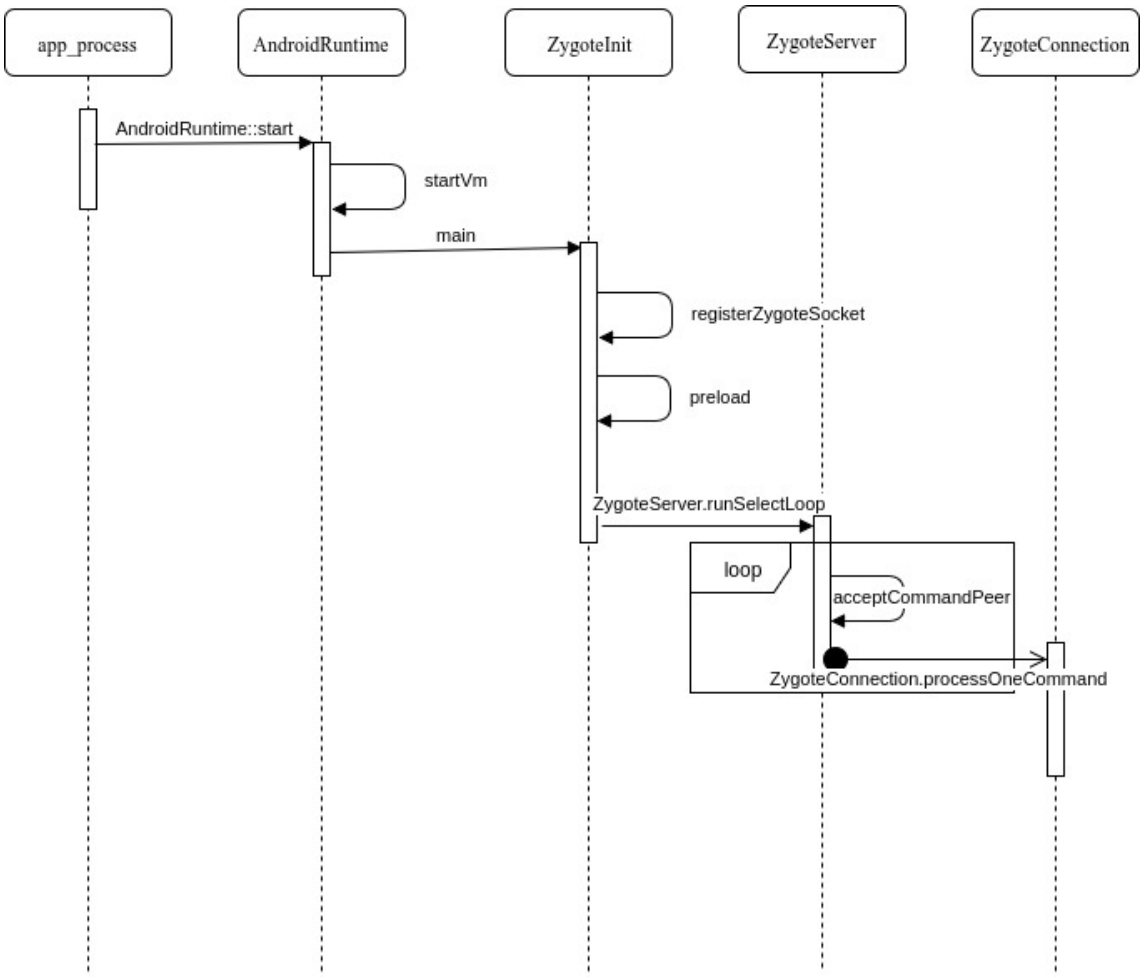

图 2.3　Zygote 进程启动过程

机制远程调用系统服务 ActivityManagerService(AMS) 的 startActivity 方法来创建 Activity。由于应用还没有启动, AMS 会调用 startProcessLocked 来创建新应用的进程, 该方法最终会调用 ZygoteProcess 类的 zygoteSendArgsAndGetResult 方法通过在介绍 Zygote 进程启动部分提到的 socket 来传递参数让 Zygote 进程生成一个新的应用进程。Zygote 进程接收到创建新进程的命令后执行 forkAndSpecialize 方法产生一个新的进程, 并把该进程 pid 返回给 AMS。新创建的进程通过 handleChildProc 来做一些初始化操作, 比如关闭 Zygote 进程的 socket, 之后会找到 android.app.ActivityThread 类并开始执行其 main 函数。这时候应用 APK 文件还没有被加载, 该进程还没有与具体的应用程序关联起来, 加载应用 APK 文件中的数据的操作通过 ActivityThread 类的 attach 方法完成。ActivityThread 类的 main 函数会调用本类的 attach 方法, 该方法又会通过 Binder 机制远程调用 AMS 中的 attachAppli-

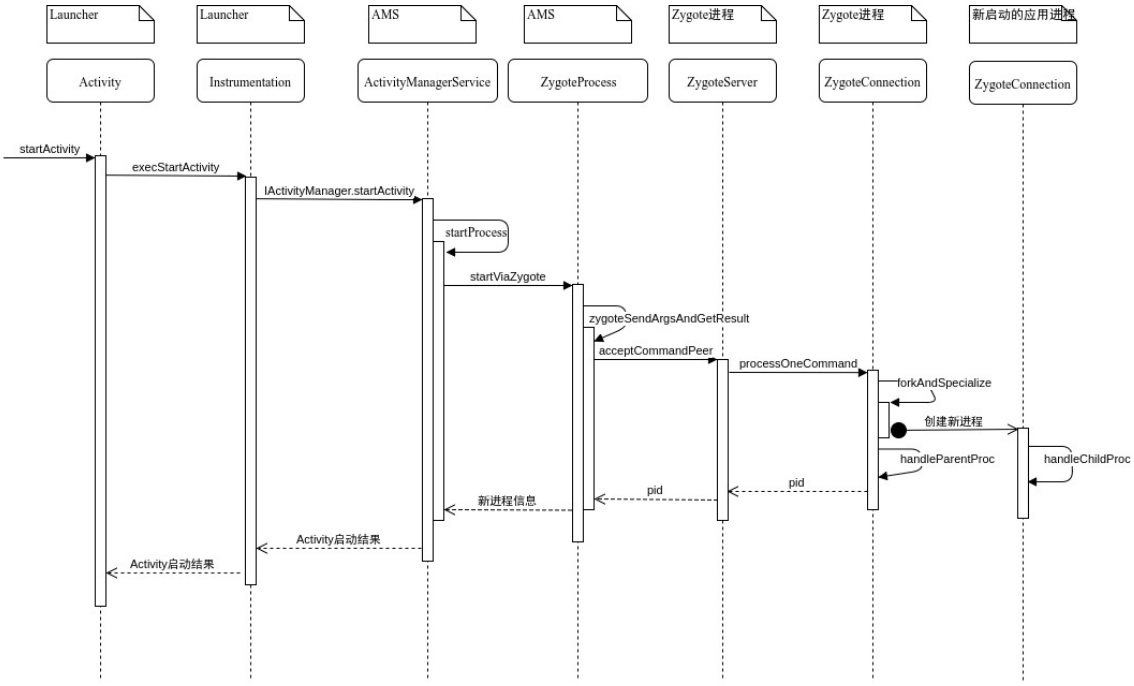

图 2.4　应用启动过程-1

cation 方法。attachAppication 方法完成新建应用的注册操作后会通过 Binder 机制远程调用新启动应用 ActivityThread 类中的 ApplicationThread 类的 bindApplication 方法。ApplicationThread 类是一个 Binder 类, 其 bindApplication 通过 Handle 机制调用 ActivityThread 类中的 handleBindApplication 方法完成应用的绑定与加载工作, 并创建 Application 实例。bindApplication 操作完成后, AMS 会通过类似机制最终调用 handlelaunchActivity 完成 Activity 的启动, 至此, 应用就成功启动了。

2.3.2　**类的加载**

Android 应用的类保存在 dex 文件里, 从 dex 文件加载类一般有 3 个过程: 首先时加载 dex 文件, 之后加载类, 最后链接类。图2.6描述了使用 Android 系统的 PathClassLoader 加载一个类的过程。PathClassLoader 用于加载应用 APK 文件中包含的 dex 文件中的类。PathClassLoader 的构造函数通过其父类的构造函数调用了 DexPathList 的构造函数来加载应用的所有 dex 文件。在 DexPathList 的构造函数中, 其调用了 makeDexElements 来加载每一个 dex 文件。makeDexElements 又会调用 DexFile 类的构造函数来获取一个 dex 文件对象。DexFile 类的构造函数通过

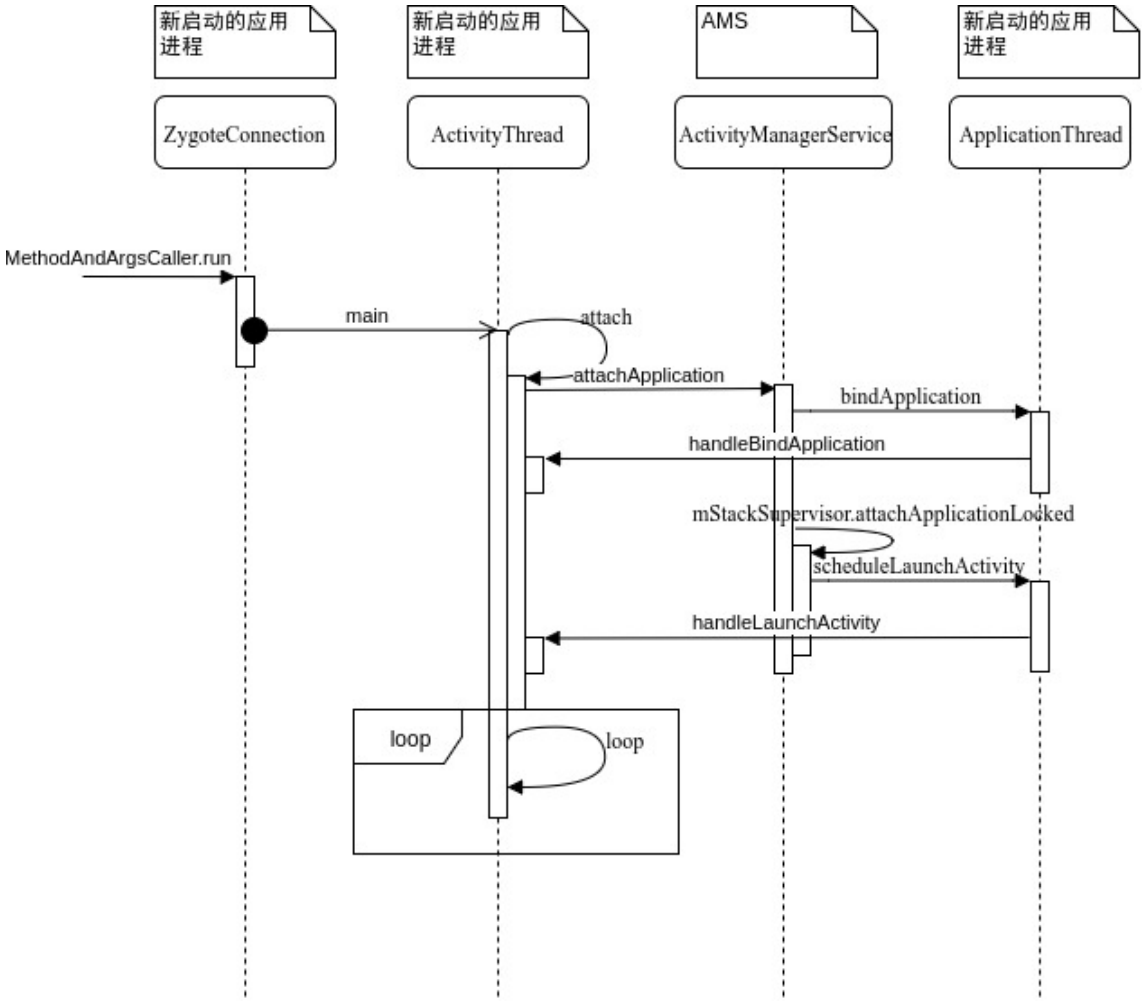

图 2.5　应用启动过程-2

openDexFileNative 等一系列 JNI 方法进入本地层调用相关函数将 dex 文件加载到内存中, 最后调用了本地层的 DexFile 类的构造函数完成对 dex 文件各字段的解析得到本地层的 DexFile 对象。该对象最后被传递给 Java 层的 DexFile 对象, 并记录在其 mCookie 成员变量中, 完成了 dex 文件的加载。

当某个类需要被加载时, 会调用 PathClassLoader 的 loadClass 方法。在该类还未被加载过时, loadClass 方法会调用 findClass 方法, findClass 方法会接着调用记录着 DexFile 对象的 DexPathList 对象的 findClass 方法。DexPathList 对象的 findClass 方法会尝试遍历所有的 DexFile 对象, 并调用每一个 DexFile 对象的 loadClassBinaryName 对象来加载目标类, 直到在某个 DexFile 对象成功加载该类为止。DexFile 对象的 loadClassBinaryName 会通过 defineClassNative 这个 JNI 方法调用本地层的 OatDexFile::FindClassDef 方法找到该类在 dex 文件中的位置, 之后调用

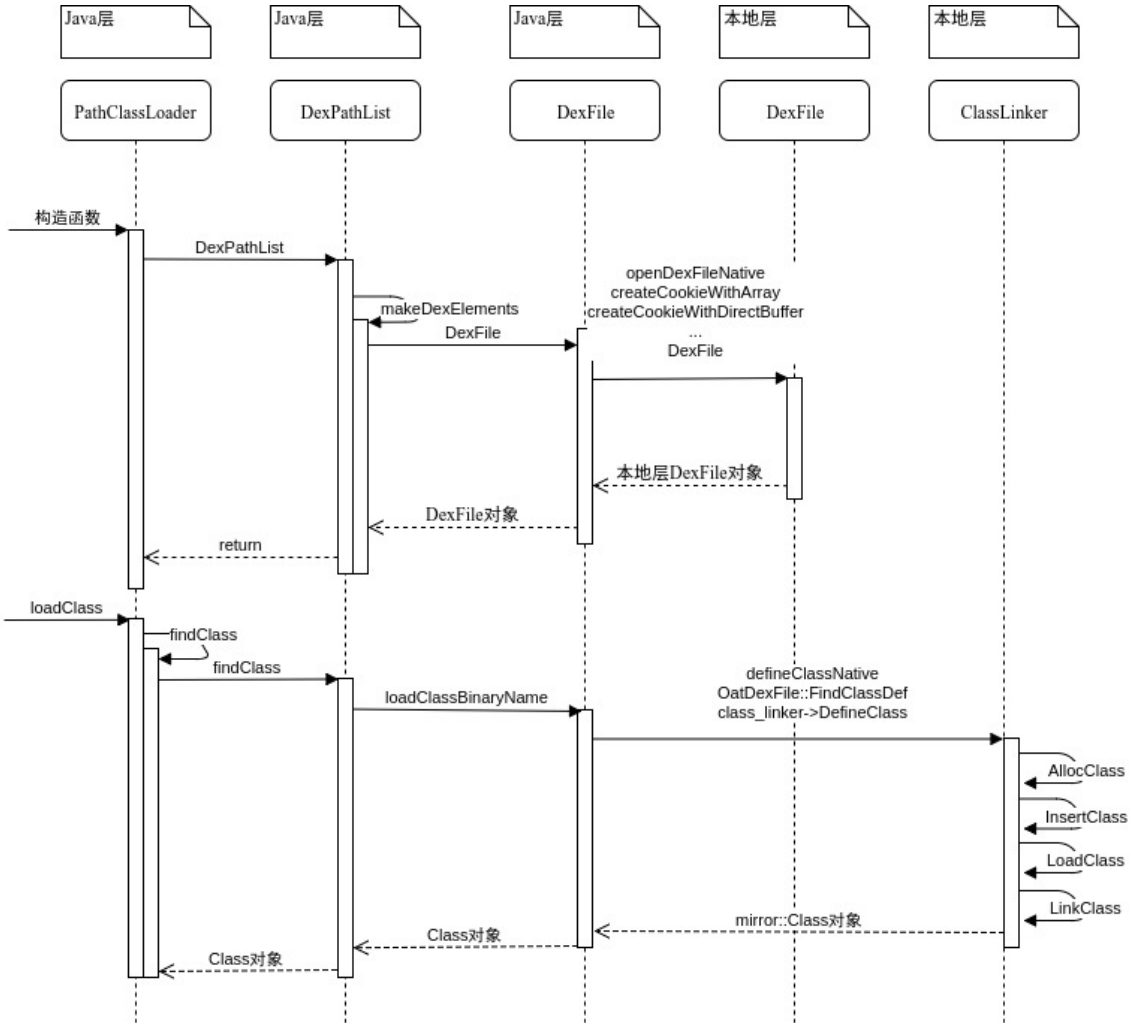

图 2.6　类的加载过程

ClassLinker::DefineClass 完成类的加载和链接工作, 并生成本地层的 mirror::Class 对象代表该类。通过该对象构造的 Java 层的 Class 对象最终会成为 findClass 的返回值, 这样一个类就完成了加载和链接, 可以被调用了。

2.3.3　**方法的执行**

为了提高运行效率 ART 虚拟机的运行机制一直在发生变化, 在 Android8.1 中, 应用的 dex 文件已经不会在安装时通过 dex2oat 全部编译而是在之后根据应用运行的情况按需编译, 同时 JIT 机制被再次启用, 并增加了对方法执行情况的统计用于在系统空闲时按需编译特定方法。因此, 一个非 JNI 方法有三种执行方式: 第一种是通过解释器执行; 第二种是通过 Just-In-Time(JIT) 编译后的机器

码执行; 第三种是通过 Ahead Of Time(AOT) 编译后的机器码执行。但总的说来, 方法的执行只有两种, 一种是解释执行字节码, 另一种是执行机器码。所有执行机器码的方法, 都会调用 Android 源代码中的/art/runtime/ArtMethod.cc 中的 ArtMethod::Invoke 函数来执行, 所有执行字节码的方法都会调用 Android 源代码中的/art/runtime/interpreter/interpreter.cc 中的 Execute 函数来执行, 此外经过 Java 反射机制从 Java 层或者 JNI 反射机制从本地层调用 Java 方法会经过 ArtMethod::Invoke 来执行。图2.7和2.8描述了 ART 几种不同的方法的执行流程。

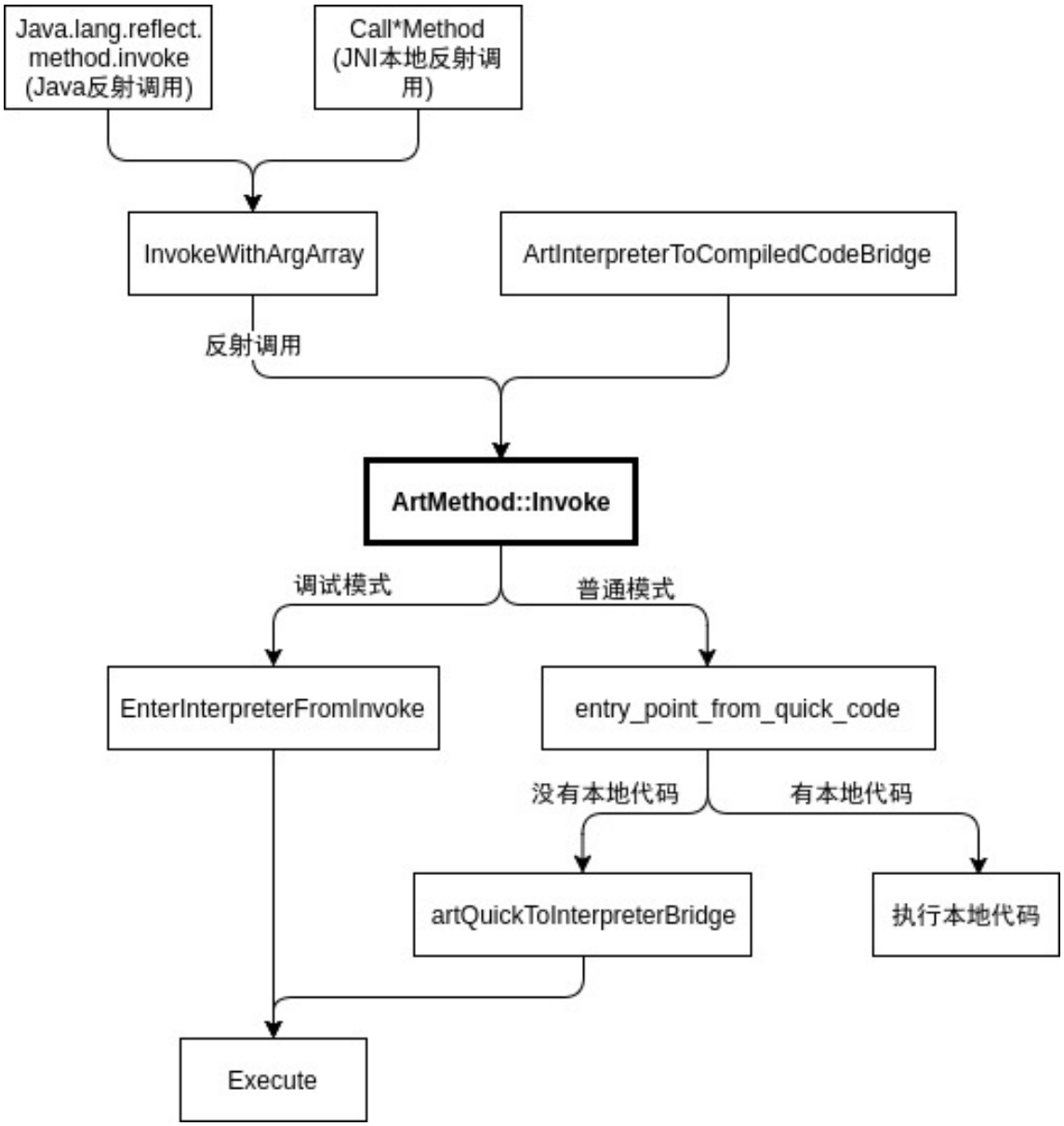

图 2.7　ART 中方法执行过程-1

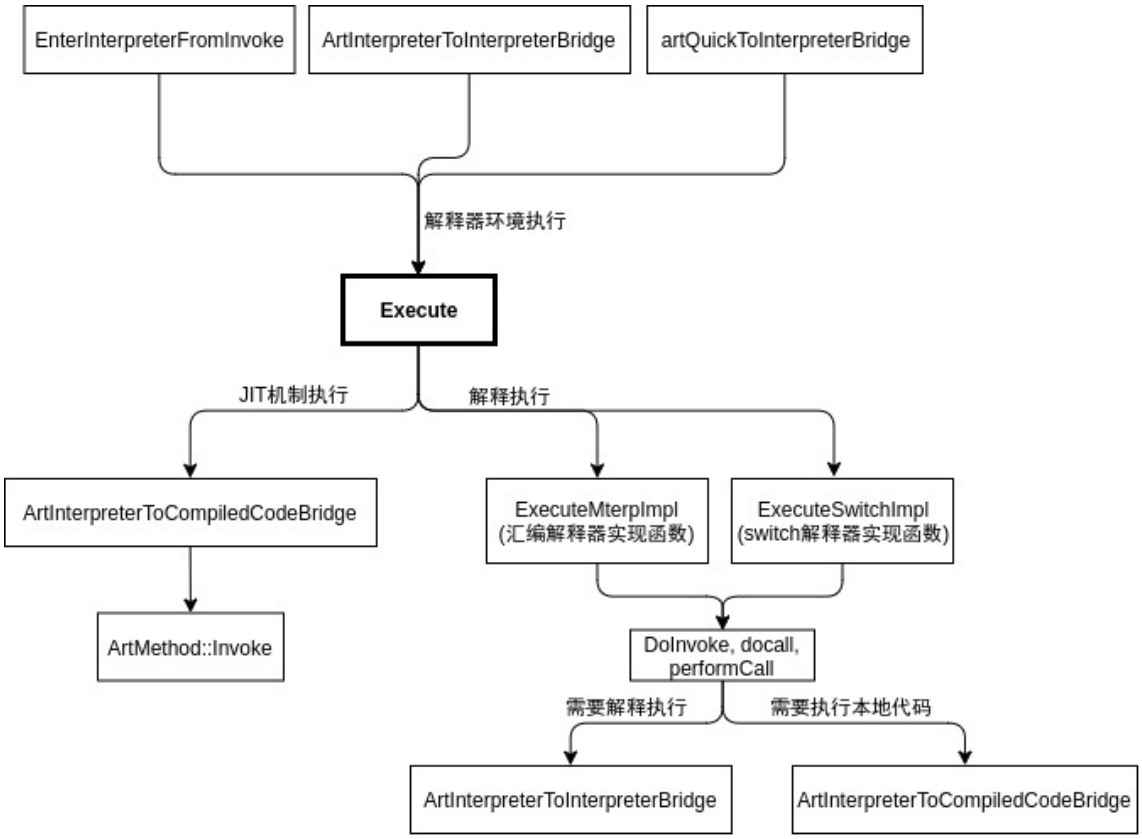

图 2.8　ART 中方法执行过程-2

图2.7主要描述了调用 ArtMethod::Invoke 来执行一个方法的情况。ArtMethod::Invoke 有两个入口, 第一个入口是使用 Java 反射 (Java.lang.reflect.method.invoke 方法) 或 JNI 反射机制 (Android 源代码/art/runtime/reflection.cc 中的一系列 Call*Method 函数) 经过一系列调用后会调用 InvokeWithArgArray。InvokeWithArgArray 会打包好方法的参数然后调用 ArtMethod::Invoke 来执行方法。该路径的调用的方法可能没有本地代码可以执行, 因此方法的入口点 (entry_point_from_quick_code) 可能会为 artQuickToInterpreterBridge, 执行方法的入口后会进入解释模式并调用 Execute 来完成方法的执行; 第二个入口是解释模式执行的方法通过 ArtInterpreterToCompiledCodeBridge 调用有本地代码的方法, 此时若应用没有运行在调试模式, 会执行应用的本地代码。以上两个入口, 在应用处于调试模式时, 都会调用 EnterInterpreterFromInvoke 进入解释模式最后调用 Execute 执行目标方法。

图2.7主要描述了调用 Execute 来执行一个方法的情况。Execute 函数有三个入口, 第一个是 EnterInterpreterFromInvoke, 用于从 ArtMethod::Invoke 中进入解释

模式执行; 第二个是 ArtInterpreterToInterpreterBridge, 用于解释执行的方法调用解释执行的方法; 第三个是 artQuickToInterpreterBridge, 用于从本地代码中进入解释模式执行。在通过 Execute 函数解释执行方法时, 可能会启动 JIT 机制编译当前方法然后通过 ArtInterpreterToCompiledCodeBridge 进入 ArtMethod::Invoke 执行编译生成的本地代码。在没有 JIT 机制没有启动时会调用解释器函数开始执行方法。Android8.1 系统中有两个不同实现方案的解释器函数: 第一个是 ExecuteMterpImpl, 该函数通过汇编代码实现, 效率很高, 但实现复杂, 不支持调试; 第二个是 ExecuteSwitchImpl, 该函数通过 switch 语句实现, 支持单步调试, 实现也比较简单但运行效率很低, 只有在调试模式的情况下使用。解释器函数在执行到调用其他方法的字节码时, 会依次调用 DoInvoke、DoCall、performCall。performCall 会根据方法的类型和是否处于调试模式来决定调用 ArtInterpreterToInterpreterBridge 以解释模式执行还是调用 ArtInterpreterToCompiledCodeBridge 执行方法的本地代码。

2.4 Android 动态分析相关技术和实现工具

Android 平台的动态分析同传统 PC 环境有相似之处, 即都是通过追踪应用软件的控制流和数据流实现对应用软件行为的揭示。但 Android 系统的多层次架构使得动态分析系统需要能够同时对本地层和 Java 层对应用的执行进行监控才能够获取到应用的完整行为。对各层次的监控, 常用的技术如下:

2.4.1 Virtual Machine Introspection

Virtual Machine Introspection(VMI) 是一种实时监控虚拟机运行状态的技术。通过该技术, 能够实现在指令层面监控应用的运行, 因此也可以实现对系统调用, 本地函数调用的监控。并且由于监控代码运行于客户系统之外, 客户系统内被监控的应用无法检测到自己处于被监控状态, 因此是一种十分有效的监控应用运行的技术。对于 Android 平台的动态监控而言, 使用 VMI 技术无需修改 Android 源代码, 可以适应 Android 版本的变化, 但该技术的运用依赖于模拟器, 一般通过修改模拟器加入监控代码来实现, 但由于依赖模拟器, 容易受到应用对模拟器环境检测的影响。Cooperdroid[10]、DroidScope[9] 使用了该技术来实现动态监控。

2.4.2 ptrace 系统调用

ptrace 系统调用是 Unix 和一些类 Unix 系统中的一种系统调用。通过使用 ptrace 系统调用, 一个进程可以监控另外一个进程的执行, 读取和修改其内存和寄存器。具体来说, ptrace 系统调用能够实现监控应用调用系统调用的情况, 能够实现指令单步执行和断点, 许多调试工具都依赖于 ptrace 系统调用实现, 如 gdb, lldb, strace, ltrace 等。由于 ptrace 系统调用能够实现单步执行, 通过 ptrace 系统调用也能实现在指令层面监控应用的执行。加上 ptrace 系统调用能够监控应用调用系统调用的情况, 通过 ptrace 系统调用可以实现对本地函数调用的监控。对于 Android 平台的动态监控而言, 使用 ptrace 系统调用也无需修改 Android 源代码, 可以适应 Android 版本的变化, 并且相比 VMI 技术, ptrace 系统调用不依赖于模拟器, 可以运行于真机上, 但存在一些反 ptrace 跟踪的技术, 例如一个进程只能被一个进程跟踪, 所以应用可以跟踪自身, 从而防止被其他工具跟踪。ptrace 系统调用一般还会和 hooking 技术结合起来使用, 可以实现对目标函数的劫持和监控。Crowdroid[12]、Glassbox[13] 使用了该技术来实现动态监控。

2.4.3 Application Instrumentation

Application Instrumentation 指通过修改需要监控的目标应用, 向其中插入监控代码实现监控应用执行的技术。对于 Android 平台而言, 该技术一般用于监控应用调用 Java 层 API 的情况, 具体来说, 通过反编译应用的 dex 文件得到 smali 代码, 搜索其中调用敏感 API 的代码, 将监控代码插入调用敏感 API 代码的周围, 再将修改后的 smali 代码编译重新打包成包含监控代码的应用, 这样在应用执行时就会执行监控代码, 输出监控信息。该技术不需要修改 Android 系统源代码, 但受到 Android 系统 API 变化的影响, 因此会在一定程度上受 Android 版本变化的影响。此外, 由于应用完整性校验以及加壳和混淆技术的广泛应用, 修改后的应用很可能无法运行, 并且一般无法从应用安装包获取到包含应用真实逻辑的 dex 文件, 因此该技术目前几乎已经失效。APIMonitor[8] 使用了该技术实现动态监控。

2.4.4 DVM/ART Instrumentation

DVM/ART Instrumentation 指通过修改 Android 系统的 DVM 或者 ART 运行时环境, 在关键的部分加入监控逻辑实现监控应用在 Java 层执行情况的技术。一般来说, 可以修改 Android 运行时环境中方法执行相关的函数来实现对应用执行的方法以及其参数和返回值的监控, 也可以修改 Android 运行时环境中的字节码解释器实现对 Java 层指令级别的监控, 在2.3节有关于 Android 运行时环境更详细的描述。该方法有些类似于上面提到的 VMI 技术, 在虚拟机层面监控虚拟机内部运行的应用, 在实现全面监控应用运行的同时也使得应用无法检测到自己处于受监控状态, 因此是监控应用 Java 层行为的一种十分有效的技术。并且该技术不依赖于模拟器, 利用其开发的动态监控系统可以运行于真机上, 不受应用检测模拟器机制的影响。但该技术的实现依赖于对 Android 源代码的修改, 并且 Android 运行时环境在不同版本上变化较大, 特别是 Android5.0 之前和 Android5.0 之后 Android 运行时环境有 DVM 替换为了 ART, 因此需要经常调整以适用于最新的 Android 版本。DroidScope[9]、Glassbox[13] 使用了该技术实现动态监控。

2.4.5 Hooking 技术

hooking 技术是一类劫持函数调用的技术, 通过 hooking 技术, 我们可以获取目标函数的参数和返回值, 可以改变目标函数的行为从而实现对目标函数的监控。对于 Android 平台, hooking 技术可以用于监控本地函数也可以用于监控 Java 方法, 其具体的实现方式有很多种, 例如针对本地函数有 Procedure Link Table(PLT) hooking、inline hooking、Import Address Table(IAT) hooking; 针对 Java 方法有修改 vtable, 修改 ArtMethod 对象的入口地址等。hooking 技术一般比较灵活, 结合一些动态代码追踪工具, 例如 frida, 能够动态的调整监控目标。由于 ptrace 系统调用能够修改被追踪进程的内存, linux 系统的 hooking 技术一般会利用 ptrace 系统调用实现。REAPER[14] 使用了该技术实现动态监控。

2.4.6 Frida

Frida[15] 是一个著名的开源跨平台动态代码追踪工具。

2.4.7 Xposed

Xposed[16] 是一个著名的 Android 平台的开源动态代码劫持框架。Xposed 框架可以在不修改应用 APK 文件的情况下劫持应用的 Java 方法从而执行一些操作从而改变应用的行为。Xposed 框架通过使用修改后的 app_process 程序替换 Android 系统中/system/bin/app_process 程序来实现对 Zygote 进程的控制，并在启动时加载 XposedBridge.jar 提供从 Java 层调用 Xposed 框架的接口。根据2.3.1节介绍的 Zygote 进程与应用启动的关系, 可以看到应用会保留 Zygote 进程中加载的所有内容, 因此控制了 Zygote 进程就实现了对所有应用进程的控制。Xposed 框架提供了劫持 Java 方法的 API 但不支持对本地函数劫持。Inspeckage[17]、InsightDroid[18] 文献 [19] 中的系统使用该技术实现了动态监控。

2.4.8 Valgrind

Valgrind[20] 是一个用于构建动态分析工具的开源代码追踪框架, 可以被编译运行在许多类 Unix 系统中。Android 系统基于 Linux 构建, 因此利用 Valgrind 来构建动态分析系统。Malton[11]、PackerGrind[21] 使用了该技术来实现动态监控。

2.5 Android 应用保护技术

为保护知识产权, 防止逆向分析, 许多 Android 应用都采用了加固技术来保护自己的代码, 目前常用的技术包括完整性校验[22]、名称混淆、方法执行混淆[?]、dex 文件动态加载、dex 文件动态修改、类动态加载、方法本地实现、模拟器检测[23]、反调试等。

完整性校验 该技术为应用在运行具体的业务逻辑前先对自身的代码文件, 资源文件等文件计算校验值, 然后把校验值与应用未经过修改时的校验值进行比较, 如果相同则表示应用没有被修改, 可以运行; 如果不同则表示应用被篡改, 停止执行。完整性校验可以只在应用启动时执行, 但容易被绕过, 也可以在应用内随机执行, 极大地增加绕过的难度。

名称混淆 该技术即在应用发布时按照一定的规则将开发应用时定义的有意义的类名、方法名、和变量名替换称无意义的字符, 从而增加了逆向分析方法用途的难度。

方法执行混淆 该技术通过 hooking 技术使用将一个方法的代码用另外一个方法替换从而使得动态监控系统记录的方法调用和实际的方法调用不同, 因此能够隐藏应用行为, 极大地增加了动态分析的难度。

dex **文件动态加载** 该技术通过先将包含应用真实逻辑的 dex 文件加密, 在应用运行时再调用解密代码释放 dex 文件并动态加载来实现隐藏包含应用真实逻辑的 dex 文件。该技术用于对抗静态分析十分有效, 可以使得静态分析无法获取到应用的真实逻辑。

dex **文件动态修改** 该技术为 dex 文件动态加载技术的改进。采用该技术时, 加载到内存的 dex 文件并不完全解密, 而是在具体的方法调用前修改方法对应 dex 文件中的部分为方法的真正指令, 在方法执行后又抹去对应指令。该技术用于对抗一些 Android 平台的脱壳工具, 使其无法得到完整的 dex 文件。

类动态加载 该技术把类方法的字节码分散到多个 dex 文件中, 在某个类被调用时动态的解密对应的 dex 文件来加载被调用的类, 从而极大地增加了脱壳工具获取包含应用真实逻辑的 dex 文件的难度。

方法本地实现 该技术将某些方法替换成本地方法, 使用本地指令实现方法的内容, 从而加大了分析方法用途的难度。有些应用加固工具还结合了 Virtual Machine Protection(VMP) 技术, 将原始指令转换成自己的私有指令集并通过私有虚拟机执行, 进一步增加了分析方法内容的难度。

模拟器检测 该技术使用了许多 Android 模拟器的特征来识别应用的运行环境是否为模拟器, 例如许多模拟器的 International Mobile Equipment Identity(IMEI) 为全 0。由于许多动态分析工具依赖于模拟器, 所以一旦识别出当前运行在模拟器环境, 应用就可以退出或者表现出一些不同于真机上的行为, 从而阻碍动态分析。

反调试 该技术通过多种方式检测应用是否处于被调试状态, 并阻止对应用的调试。例如, 应用通过调用 ptrace 追踪自身从而避免被其他监控进程追踪; 应用通过搜索内存中某些著名调试工具的名称, 如 strace, ltrace, valgrind 等来确定自身是否被调试; 应用 hook 自身的某些函数, 如 open, wrtie 等来阻止自己的数据被调试工具输出。反调试技术给真机上的动态分析系统带来了较大的障碍而基于模拟器的动态分析系统可以把监控功能部署在 Android 系统的外部, 使得应用无法检测到监控工具的存在从而避免反调试技术的影响。

3 系统设计实现

3.1 概览

本文的背景技术分析部分介绍了当前的动态分析相关技术和实现工具以及 Android 系统的部分运行原理。结合前面的分析和系统实现的可行性, 本系统整体实现方案如下:

采用2.4.4节提到的 ART Instrumentation 的技术, 修改 Android8.1 系统源代码中 ART 虚拟机部分来实现 Java 层次的方法调用监控和简单的脱壳功能。通过 Frida 工具来实现对本地函数调用以及 Java 层特定方法的监控, 同时增加监控系统的灵活性和可扩展性。修改 Android 源代码中与应用启动相关的部分实现监控应用的启动情况, 在目标应用启动时开启其他监控功能。通过简化日志内容和使用内存缓存日志的方式实现更为高效的日志系统用于保存监控系统记录的数据。通过系统属性来实现目标应用和其他配置选项的设置。

图3.1描述了系统的总体设计, 加粗的矩形部分为本系统的模块。其中 EvMonitor-core 模块运行在 ART 内部, 负责监控 Java 方法调用, dex 文件解析, 并且实现了日志系统。该部分还通过 JNI 给运行在应用层的 EvMonitor-startLogger 模块提供了接口, 可以启动和关闭监控功能, 调用日志系统等, 这些接口也通过导出函数的方式提供给 EvMonitor-Frida 调用。EvMonitor-Frida 是一个本地动态链接库, 用于实现对本地层函数调用以及某些特别感兴趣的 Java 层方法的监控, 在目标应用进程加载应用文件之前被加载到目标进程中, 开启对目标应用的监控。

3.2 监控应用启动

本功能由 EvMonitor-startLogger 实现。2.3.1节详细介绍了 Android 系统应用的启动过程。由于 Android 应用进程不是通过直接使用应用 APK 文件启动, 而是通过 Zygote 进程 fork 产生, 因此无法按照传统 PC 上启动子进程, 然后加载监控模块开

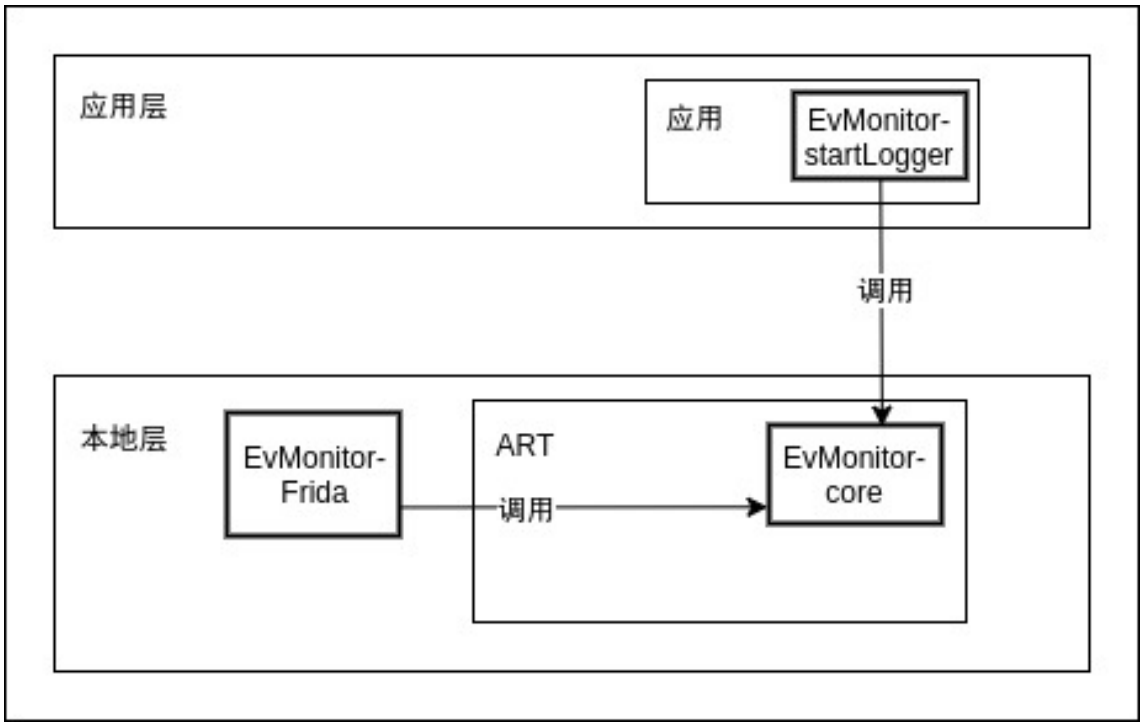

图 3.1　EvMonitor 概览

启监控, 最后再加载目标应用开始执行的方式来实现在应用加载前启动对应用的监控。只有通过监控应用的启动, 在能够获取到应用的名称并且应用自身代码还未执行时开启完成监控应用的初始化操作。ActivityThread 类的 handleBindApplication 方法开始的位置符合上述条件, 因此本系统选择在 handleBindApplication 方法的适当位置添加监控代码。

监控代码的实现逻辑如下: 首先读取当前进程名和应用名 (有些应用会启动多个进程), 使用日志系统记录当前启动的应用名和进程名。接着读取系统属性 em.target_type, 确定是只监控某个进程还是监控该应用的全部进程。接着根据上述结果读取系统属性获取要监控的进程名或者应用名并同本进程或者应用名比较, 如果不同则退出监控代码, 不启动监控系统; 如果相同则调用 Evmonitor-core 提供的初始化监控的接口来启动对该应用的监控。之后再根据当前运行环境是 32 位还是 64 位加载对应的 EvMonitor-Frida 动态链接库, 完成监控系统的初始化。

为了便于调用 JNI 和被调用, 我将上述监控代码作为一个新的静态方法 logAppStartAndSetTarget_em 加入了 com.android.internal.os.Zygote 类中, 并通过在 Zygote 类中增加本地方法 nativeEnableMonitor 的方式调用了 EvMonitor-core 提供的

本地层接口。

3.3 监控 Java 方法调用

本功能由 EvMonitor-core 实现。为了能够获取到应用所有 Java 方法调用行为, 本系统选择在 ART 内执行方法的函数中加入监控代码。2.3.3节详细介绍了 ART 虚拟机执行方法的过程。根据分析, 在 ArtMethod::Invoke 和 Execute 函数中插入监控代码就能够记录所有的 Java 方法调用情况。

ART 虚拟机中的每个 Java 方法都使用一个 ArtMethod 类型的对象来表示, 而在 ArtMethod::Invoke 和 Execute 执行方法时都能够获取到代表正在执行的方法的 ArtMethod 对象, 因此本系统在 ArtMethod 类 (/art/runtime/artmethod.cc) 中加入一个方法 log_em 用于记录当前方法并写入日志中。log_em 方法会调用当前 ArtMethod 对象的 PrettyMethod 方法获取方法信息然后调用 EvMonitor-core 中的日志系统将其写入日志。

本系统在 ArtMethod::Invoke 和 Execute 函数的入口和所有出口位置调用了上述 log_em 函数记录方法执行的开始和结束。

3.4 监控本地函数调用

本功能由 EvMonitor-Frida 实现。本系统使用 Frida 工具的动态链接库形式 (frida-gadget), 通过在应用加载自身代码之前加载 frida-gadget 并执行监控代码来实现对本地层函数调用的监控。目前的监控较为简单, 主要是利用了 Frida 工具提供的 Interceptor 来 hook 了 open、execve、socket、dlopen、dlsym 函数来监控它们的调用情况。由于 Frida 工具实现了 Javascript 的 API, 因此可以灵活的添加监控函数和其他监控逻辑, 增强本系统。

3.5 脱壳功能

本功能由 EvMonitor-core 实现。目前 Android 平台应用加固工具的加壳机制十分复杂, 本系统根据2.3.2节介绍的类加载过程, 通过在用于 dex 文件加载的关

键函数, 即位于安卓源代码中/art/runtime/DexFile.cc 中 DexFile 类的构造函数 DexFile() 中加入代码来实现简单脱壳获取 dex 文件的功能。DexFile 类的构造函数原型图3.2所示。其中第一个参数是 dex 文件被加载到内存的起始地址, 第二个 dex 文

```
DexFile::DexFile(const uint8_t* base,
                 size_t size,
                 const std::string& location,
                 uint32_t location_checksum,
                 const OatDexFile* oat_dex_file)
```

图 3.2　DexFile() 原型

件的大小。本系统插入该函数的监控代码会先检查是否开启对当前进程的监控, 如果监控功能没有开启则不在执行后续操作。如果监控功能开启会先调用日志系统记录当前加载的 dex 文件起始地址和大小, 然后再检查是否启动脱壳功能。如果脱壳功能启动则读取这两个参数后会调用 EvMonitor-core 中用于保存 dex 文件的 dumpDex 函数, 将加载的 dex 文件写入初始化时设定的位置。

3.6　日志系统

本功能由 EvMonitor-core 实现。各种动态分析系统中, 输入监控记录都是一个十分重要的功能, 但由于 IO 的速度很慢, 在频繁的调用中记录日志信息对应用本身的执行速度影响很大。在最初的测试中本系统使用 Android 系统本身的日志系统接口来记录监控信息, 但在测试中发现 Java 层方法的调用很多, 通过 Android 系统本身的日志系统输出会导致应用运行变得非常慢, 所以开发了本系统的日志部分。

表3.1描述了日志系统用到的一些变量和提供的接口函数。日志系统会在初始化时使用 mmap 创建一个 4M 的内存缓冲区, 并在被监控应用的目录下以当前时间和进程号创建一个文件夹用来保存此次执行时的日志数据。缓冲区的起始地址会使用 log_base 来记录, 并且 log_data 会被初始化为同 log_base 一致。当 log 函数被调用来记录日志时, 其会先检查 log_spare_space, 如果空间足够就会把接收到的日志信息加上当前线程号写入 log_data 位置处的内存缓冲区中并更新 log_spare_space、log_data 的值 。如果空间不足就会调用 writeLog 函数试图把缓冲区的日志写入文件。writeLog 函数会检查 log_file_amount 的值确定日志文件数量是否达到最大, 如

表 3.1　日志系统变量和函数

	名称	作用
变量	log_base	记录日志缓冲区起始地址
	log_data	记录日志缓冲区空闲区起始地址
	log_spare_space	记录日志缓冲区空闲空间
	log_file_amount	记录已经写入磁盘的日志文件数量
函数	log()	写入日志
	writeLog()	把缓冲区日志写入文件

果是, 则通过 Android 系统的 log 系统记录一个日志文件以达到数量上限的错误, 并不在执行后续操作, 如果不是则将缓冲区的数据写入文件, 并将 log_base 的值赋给 log_data, 设置 log_spare_space 的值为缓冲区最大容量, 同时把 log_file_amount 加 1。

4 实验与结果分析

4.1 实验总体方案

本系统的测试分为两个部分, 第一部分是功能测试, 第二部分是性能测试。功能测试部分主要测试监控应用启动, 监控 Java 方法调动, 监控本地函数执行以及脱壳功能是否正常运行, 结果是否准确; 性能测试部分主要测试在监控系统运行时的性能开销, 并同 Android Device Monitor 做对比测试。

4.2 实验运行环境

由于本系统基于 Android8.1 构建, 并运行于真机上, 因此本次实验会涉及两台设备。设备 1 是用于配置和控制本系统的个人电脑, 设备 2 是运行该系统的智能手机, 两台设备的硬件和软件环境如表4.1所示。

表 4.1 实验设备环境

设备	型号	CPU(**型号/主频**)	**内存**	操作系统
设备 1	Lenovo XiaoXin 700-15ISK	Intel core i5 6300HQ / 2.3GHZ	8GB	深度操作系统 15.9.3
设备 2	Google Nexus 5X	MSM8992 / 1.8GHZ	2GB	Android8.1

4.3 功能测试

本次功能测试使用了支付宝应用的 10.1.62 版本 (应用软件包名 com.eg.android.AlipayGphone) 本文作者开发的一款名为 noticer 的应用 (应用软件包名 top.january147.noticer) 来进行。noticer 应用经过 360 加固保[24] 加固, 以检测本系统功能的有效性。

4.3.1 监控应用启动功能测试

在设备 1 上开启终端, 输入 adb logcat|grep EvMonitor 等待读取本系统监控应用启动的记录。然后执行如下操作: 在设备 2 上启动支付宝应用, 然后退出支付宝应用再启动 noticer 应用。

从终端中读取的监控记录如图4.1所示。监控记录显示支付宝应用的首先启动,

```
january@january-PC:~/Desktop$ adb logcat|grep EvMonitor
02-05 16:43:31.675 26032 26032 D EvMonitor:: app started--ProcessName:[com.eg.android.AlipayGphone]--PackageName:[com.eg.android.AlipayGphone]
02-05 16:43:32.183 26106 26106 D EvMonitor:: app started--ProcessName:[com.eg.android.AlipayGphone:tools]--PackageName:[com.eg.android.AlipayGphone]
02-05 16:43:32.359 26138 26138 D EvMonitor:: app started--ProcessName:[com.eg.android.AlipayGphone:push]--PackageName:[com.eg.android.AlipayGphone]
02-05 16:43:48.236 26660 26660 D EvMonitor:: app started--ProcessName:[top.january147.noticer]--PackageName:[top.january147.noticer]
02-05 16:43:57.803 26716 26716 D EvMonitor:: app started--ProcessName:[com.eg.android.AlipayGphone:push]--PackageName:[com.eg.android.AlipayGphone]
02-05 16:43:58.338 26777 26777 D EvMonitor:: app started--ProcessName:[com.eg.android.AlipayGphone:tools]--PackageName:[com.eg.android.AlipayGphone]
```

图 4.1　应用启动监控记录

该应用共计启动了 3 个进程, 进程名分别为 com.eg.android.AlipayGphone、com.eg.android.AlipayGphone:push、和 com.eg.android.AlipayGphone:tools; 接着记录了 noticer 的启动, 该应用只启动了一个进程, 进程名为 top.january147.noticer; 后边又记录了支付宝启动的两个进程,com.eg.android.AlipayGphone:tools 和 com.eg.android.AlipayGphone:push 而测试流程中并没有再次执行支付宝, 可以看到支付宝有自动启动的行为。

4.3.2 监控 Java 方法调用功能测试

在设备 1 上开启终端, 键入 adb shell 进入设备 2 的 shell。此时利用 setprop 命令设置系统属性 em.target_app 为 noticer 软件的包名 top.january147.noticer, 然后再执行 logcat | grep Evmonitor 查看系统运行情况。在设备 2 上启动 noticer, 启动后界面如图4.2所示, 点击底部的电话通知服务按钮启动电话通知服务, 等待启动完成再次点击该按钮关闭电话通知服务, 完成后退出应用。此时在设备 1 上运行脚本 pull_log_dir.sh 获取目标应用日志文件夹。

图4.3是终端中显示的本系统的运行信息。可以看到, 应用启动监控模块检测到目标应用启动后调用本地层的接口进行了一系列初始化操作, 包括创建日志文件夹, 分配日志缓冲区等等, 此外还记录了 dex 文件解析的情况。

打开脚本 pull_log_dir.sh 从目标应用目录下取得的日志文件夹会发现一个以时间命名的文件夹, 该文件夹记录了应用在上述执行过程中的日志数据, 其文件如

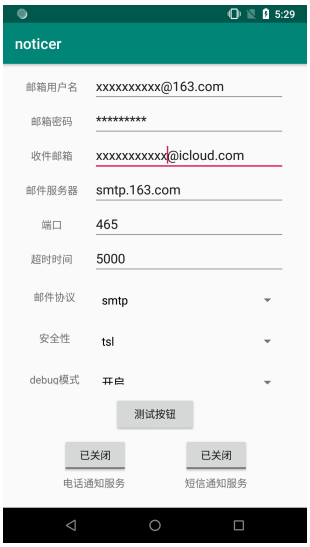

图 4.2　noticer 界面

```
bullhead:/ # logcat | grep EvMonitor
02-05 20:02:29.542 29973 29973 D EvMonitor:: app started--ProcessName:[top.january147.noticer]--PackageName:[top.january147.noticer]
02-05 20:02:29.543 29973 29973 D EvMonitor: log file dir[/data/data/top.january147.noticer/log_dir/1970-02-05_20:02:29]
02-05 20:02:29.543 29973 29973 D EvMonitor: embeded log enabled, log buff address[0xd73c5000]
02-05 20:02:29.543 29973 29973 D EvMonitor: log buff size[4194304]
02-05 20:02:29.544 29973 29973 D EvMonitor: native interface seted up, pid[29973]
02-05 20:02:29.545 29973 29973 D EvMonitor:: enable monitor, pid is 29973
02-05 20:02:30.391 29973 29973 D EvMonitor: dexFileParsed--file address:[0xd6fa301c]--file size:[1221252]
02-05 20:02:30.395 29973 29973 D EvMonitor: file dumped, path[/data/data/top.january147.noticer/dumped_file/vmxsrp_3096150]
02-05 20:02:30.947 29973 29973 D EvMonitor: dexFileParsed--file address:[0xd0cc1000]--file size:[3401212]
02-05 20:02:30.957 29973 29973 D EvMonitor: file dumped, path[/data/data/top.january147.noticer/dumped_file/btazle_3096150]
02-05 20:02:30.968 29973 29973 D EvMonitor: dexFileParsed--file address:[0xd0a6b000]--file size:[1221252]
02-05 20:02:30.971 29973 29973 D EvMonitor: file dumped, path[/data/data/top.january147.noticer/dumped_file/jvffki_3096150]
```

图 4.3　监控系统运行日志

图4.4所示。可以看到有许多 em_ 开头后接数字编号编号的 log 文件, 编号即为 log 文件生成的顺序, 这些文件记录了应用在上述过程执行中全部 Java 方法调用。

```
january@january-PC:~/workplace/graduation_project/EvMonitor/EvMonitor_outer_tools/log_dir/1970-02-05_20:02:29$ ls
em_0.log   em_12.log  em_15.log  em_18.log  em_20.log  em_23.log  em_26.log  em_29.log  em_31.log  em_5.log  em_8.log
em_10.log  em_13.log  em_16.log  em_19.log  em_21.log  em_24.log  em_27.log  em_2.log   em_3.log   em_6.log  em_9.log
em_11.log  em_14.log  em_17.log  em_1.log   em_22.log  em_25.log  em_28.log  em_30.log  em_4.log   em_7.log
```

图 4.4　Java 调用记录日志文件

限于论文篇幅原因, 本文使用脚本 log_filter.sh 过滤上述日志文件, 生成只包含 noticer 自身方法调用的记录文件, 并在其中搜索关于执行应用运行时两次点击按钮的调用记录, 结果如图4.5和4.6所示。由于启动服务需要进行许多初始化操作, 流程较长图4.5是服务启动开始时和完成时的部分调用记录。图4.5是关闭服务的完整调用记录。每条记录的格式为” 线程号执行方式方法名称”, 执行方式由两个大写字母和”–” 组成, 第一个字母为 I 表示方法开始执行, 为 O 表示方法执行结束; 第二个

字母为 V 表示通过 ArtMethod::Invoke 函数执行, 为 E 表示通过 Execute 函数执行。

```
29973 II-- void top.january147.noticer.MainActivity.onButton1Click(android.view.View)
29973 IE-- void top.january147.noticer.MainActivity.onButton1Click(android.view.View)
29973 IE-- void top.january147.noticer.MainActivity.requestPermission()
29973 OE-- void top.january147.noticer.MainActivity.requestPermission()
29973 IE-- void top.january147.noticer.MainActivity.initNoticeService()
29973 IE-- boolean top.january147.noticer.NoticerNoticeService$NoticerBinder.getWorkingState()
29973 IE-- boolean top.january147.noticer.NoticerNoticeService.access$000(top.january147.noticer.NoticerNoticeService)
29973 OE-- boolean top.january147.noticer.NoticerNoticeService.access$000(top.january147.noticer.NoticerNoticeService)
29973 OE-- boolean top.january147.noticer.NoticerNoticeService$NoticerBinder.getWorkingState()
29973 IE-- void top.january147.noticer.NoticerNoticeService$NoticerBinder.startService()
29973 IE-- void top.january147.noticer.NoticerNoticeService.access$100(top.january147.noticer.NoticerNoticeService)
29973 IE-- void top.january147.noticer.NoticerNoticeService.startWorking()
29973 IE-- void top.january147.noticer.IncomingCallMngr.<init>()
29973 IE-- void top.january147.noticer.TimerMngr.<init>()
29973 OE-- void top.january147.noticer.TimerMngr.<init>()
29973 OE-- void top.january147.noticer.IncomingCallMngr.<init>()
29973 IE-- void top.january147.noticer.NoticerNoticeService.initPhoneListen()
...
29973 IE-- java.util.Properties top.january147.noticer.ConfigManager.generateMailConfigs()
29973 OE-- java.util.Properties top.january147.noticer.ConfigManager.generateMailConfigs()
29973 IE-- void top.january147.noticer.MailManager.updateConfig(java.util.Properties)
29973 IE-- void top.january147.noticer.MailManager.init()
29973 IE-- java.util.Properties top.january147.noticer.MailManager.serverConfig()
29973 OE-- java.util.Properties top.january147.noticer.MailManager.serverConfig()
29973 IE-- void top.january147.noticer.MailManager$1.<init>(top.january147.noticer.MailManager)
29973 OE-- void top.january147.noticer.MailManager$1.<init>(top.january147.noticer.MailManager)
29973 OE-- void top.january147.noticer.MailManager.init()
29973 OE-- void top.january147.noticer.MailManager.updateConfig(java.util.Properties)
29973 OE-- void top.january147.noticer.NoticerNoticeService$NoticerBinder.configMail()
29973 OE-- void top.january147.noticer.MainActivity.initNoticeService()
29973 OE-- void top.january147.noticer.MainActivity.onButton1Click(android.view.View)
29973 OI-- void top.january147.noticer.MainActivity.onButton1Click(android.view.View)
```

图 4.5　启动电话通知服务的调用记录

```
29973 II-- void top.january147.noticer.MainActivity.onButton1Click(android.view.View)
29973 IE-- void top.january147.noticer.MainActivity.onButton1Click(android.view.View)
29973 IE-- void top.january147.noticer.MainActivity.requestPermission()
29973 OE-- void top.january147.noticer.MainActivity.requestPermission()
29973 IE-- void top.january147.noticer.MainActivity.stopNoticeService()
29973 IE-- void top.january147.noticer.NoticerNoticeService$NoticerBinder.stopService()
29973 IE-- void top.january147.noticer.NoticerNoticeService.access$200(top.january147.noticer.NoticerNoticeService)
29973 IE-- void top.january147.noticer.NoticerNoticeService.stopWorking()
29973 IE-- void top.january147.noticer.IncomingCallMngr.clearAll()
29973 IE-- void top.january147.noticer.TimerMngr.clearAll()
29973 OE-- void top.january147.noticer.TimerMngr.clearAll()
29973 OE-- void top.january147.noticer.IncomingCallMngr.clearAll()
29973 IE-- void top.january147.noticer.NoticerLog.log(java.lang.String)
29973 OE-- void top.january147.noticer.NoticerLog.log(java.lang.String)
29973 OE-- void top.january147.noticer.NoticerNoticeService.stopWorking()
29973 OE-- void top.january147.noticer.NoticerNoticeService.access$200(top.january147.noticer.NoticerNoticeService)
29973 OE-- void top.january147.noticer.NoticerNoticeService$NoticerBinder.stopService()
29973 OE-- void top.january147.noticer.MainActivity.stopNoticeService()
29973 IE-- void top.january147.noticer.MainActivity.toastShow(java.lang.String)
29973 OE-- void top.january147.noticer.MainActivity.toastShow(java.lang.String)
29973 OE-- void top.january147.noticer.MainActivity.onButton1Click(android.view.View)
29973 OI-- void top.january147.noticer.MainActivity.onButton1Click(android.view.View)
```

图 4.6　关闭电话通知服务的调用记录

4.3.3　**脱壳功能测试**

4.3.2小节测试流程中本系统已经启动并执行了脱壳功能, 在本小节的测试中直接使用脚本 pull_dumped_file.sh 从目标应用文件夹获取得到包含 dex 文件的文件夹。该文件夹中为 3 个 dex 文件, 使用 dex2jar 工具转换成 jar 文件后文件夹内

容如图4.7所示。没有后缀名的 3 个文件为脱壳工具抓取的 dex 文件, 其他 3 个为 dex2jar 工具转换后对应的 jar 文件。

```
january@january-PC:~/workplace/graduation_project/EvMonitor/EvMonitor_outer_tools/extra/dumped_file$ ls
btazle_3096150              jvffki_3096150              vmxsrp_3096150
btazle_3096150-dex2jar.jar  jvffki_3096150-dex2jar.jar  vmxsrp_3096150-dex2jar.jar
```

图 4.7　脱壳工具抓取的 dex 文件

使用 jd-gui 打开 3 个 jar 文件后成功在名为 btazle_3096150-dex2jar.jar 的文件中找到了实现应用真实功能的类, 成功实现了脱壳。图4.8是 jd-gui 工具查看的结果, top.january147.noticer 包下的类即为应用本身的类, 右侧显示的为4.3.2小节中 Java 方法调用记录中的 onButton1Click 方法源代码。

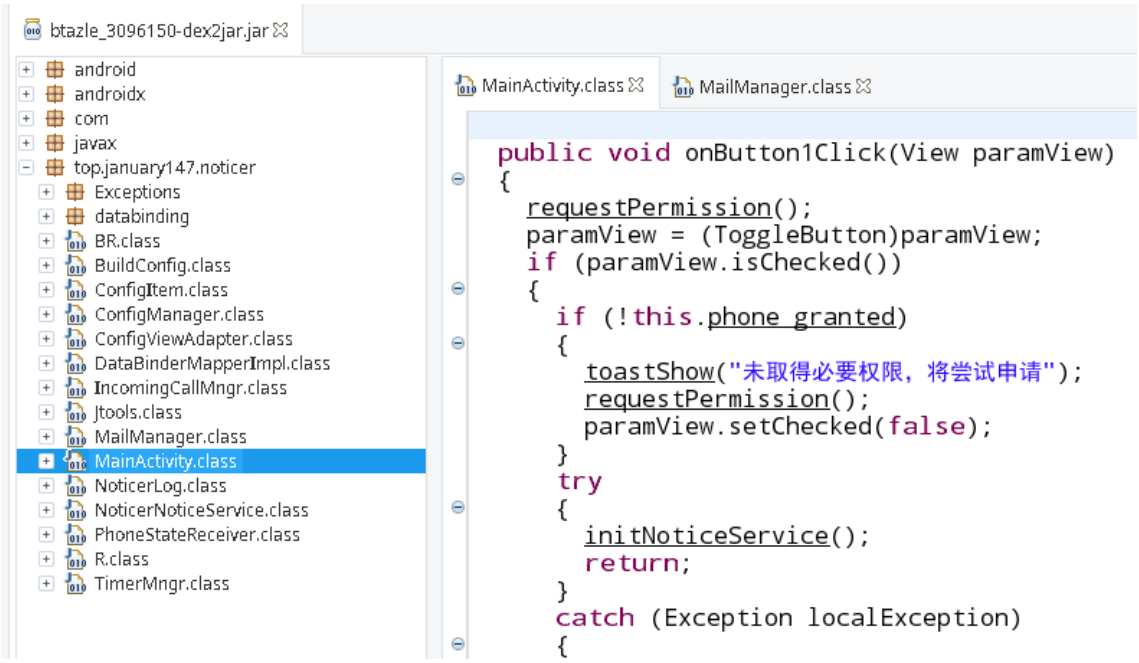

图 4.8　脱壳后得到应用本身的类

4.3.4　监控本地函数调用功能测试

开启监控目标应用 noticer, 本系统记录的本地调用信息会保存到应用目录下的 log_dir 文件夹中, 从中获取的本地函数调用信息如图4.9所示。

```
dlsym(JNI_OnLoad)
open(/proc/self/cmdline)
open(/data/app/top.january147.noticer-zAqLdZKYwC77-EZGTMyC-g==/base.apk.arm.flock)
open(/data/app/top.january147.noticer-zAqLdZKYwC77-EZGTMyC-g==/oat/arm/base.vdex)
dlsym(oatdata)
dlsym(oatlastword)
dlsym(oatbss)
dlsym(oatbsslastword)
dlsym(oatbssmethods)
dlsym(oatbssroots)
open(/data/app/top.january147.noticer-zAqLdZKYwC77-EZGTMyC-g==/base.apk)
open(/system/framework/arm/boot.art)
open(/data/dalvik-cache/arm/system@framework@boot.art)
open(/data/app/top.january147.noticer-zAqLdZKYwC77-EZGTMyC-g==/oat/arm/base.art)
open(/data/app/top.january147.noticer-zAqLdZKYwC77-EZGTMyC-g==/base.apk)
open(/system/build.prop)
open(/data/data/top.january147.noticer/.jiagu/libjiagu.so)
...
open(/dev/ion)
open(/data/vendor/gpu/esx_config.txt)
open(/data/misc/gpu/esx_config.txt)
open(/data/vendor/gpu/esx_config.txt)
open(/data/misc/gpu/esx_config.txt)
dlsym(eglSetBlobCacheFuncsANDROID)
open(/storage/emulated/0/Android/data/top.january147.noticer/files/config.txt)
dlsym(HIDL_FETCH_IMapper)
dlsym(HMI)
open(/dev/ion)
open(/data/user_de/0/top.january147.noticer/code_cache/com.android.opengl.shaders_cache)
open(/data/app/top.january147.noticer-zAqLdZKYwC77-EZGTMyC-g==/base.apk)
```

图 4.9　本地函数调用记录

4.4　性能测试

由于本系统的主要性能开销在于记录方法调用时的性能开销, 针对 cpu 性能的性能测试工具无法成功检测本系统的性能, 因此本文开发了一个简单的测试 Java 方法调用性能检测的应用, 该应用将一个简单的无内部调用的 java 方法执行固定次数并测量总用时用于评价 Java 方法调用性能。本文使用了该应用对本系统和 Android Device Monitor 的 Method Profile 功能进行了测试, 结果如图4.10所示。其

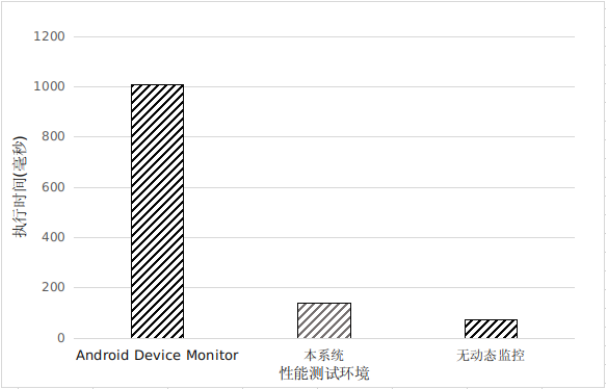

图 4.10　本地函数调用记录

中的执行时间为 10 次测量结果取平均值得到的。

4.5 结果分析

功能测试的结果显示本系统的基本功能正常, 能够实现对应用启动、Java 方法调用、本地函数调用的监控和以及脱壳功能, 系统设计原理可行。性能测试的结果显示本系统的执行耗时约为正常执行时的两倍, 而 Android Device Monitor 执行耗时约为正常执行的 14 倍, 本系统为 Android Device Monitor 的 Method Profile 功能运行效率的 7 倍, 因此十分高效。

5 总结与展望

5.1 论文工作总结

Android 平台的应用, 无论是正常的还是恶意的, 都在利用不断变化的技术来对抗应用分析, 然而对应用进行分析又是辨别出恶意应用比不可少的环节, 因此应用开发者想尽办法隐藏应用的真实行为, 应用分析者想尽办法找出应用隐藏的行为就成了一场持续的对抗。学习 Android 系统的设计原理和现有技术则是加入这场对抗十分重要的部分, 在充分理解已有知识的基础上, 才能产生新的知识。本文将主要工作重心放在学习 Android 系统和已有的动态分析技术上, 并在新的 Android 系统上进行了部分实践, 具体来说, 论文的主要工作内容如下:

研究了 Android 系统的整体架构, Android 应用开发相关知识, 研究了国内外关于 Android 平台动态分析的多项技术成果并总结了对抗应用分析的常用技术和实现动态分析常用的技术和工具, 分析了其优势和劣势。

深入源代码分析了 Android8.1 系统中 ART 虚拟机的设计和运行机制, 在 3 个重要方面, 即应用的启动、类的加载和方法的执行方面给出了关键调用图。

利用上述研究的成果设计和实现了一个 Android 应用的动态行为捕获系统, 并实现了简单的脱壳功能。对本文实现的系统进行了详细的测试, 验证了设计方案的可行性和有效性。

5.2 进一步工作展望

本文对于 Android 系统, Andriod 平台动态分析技术及其相关技术 (如调试技术) 的认识还不够深刻, 并且本文所有构建的系统只是对设计思路的简单测试, 并不是一个成熟的高可用性的系统, 因此未来的未来的工作分为两部分, 第一部分是继续深入研究和测试已有的调试技术和动态分析技术, 深入研究 Android 系统各个部分的设计和实现原理, 第二部分是完善本系统的设计和实现, 具体来说有以下方

面:

本系统对本地函数的监控方案不够好, 存在较严重的稳定性问题, 需要更稳定有效的实现方案。

本系统的缺乏有效的运行时控制方式, 没有提供在系统运行时用户可以控制系统的接口, 需要提供接口来保证灵活性。

本系统产生的 Java 调用日志数量众多且不易阅读, 需要更有效的过滤方案来去除无效信息和冗余信息。

参考文献

[1] newzoo. Top 50 countries/markets by smartphone users and penetration. https://newzoo.com/insights/rankings/top-50-countries-by-smartphone-penetration-and-users/, 2018.

[2] statista. App stores - statistics & facts. https://www.statista.com/topics/1729/app-stores/.

[3] statcounter. Mobile operating system market share worldwide. http://gs.statcounter.com/os-market-share/mobile/worldwide.

[4] Kantar. Smartphone os sales market share evolution. https://www.kantarworldpanel.com/global/smartphone-os-market-share/, 2019.

[5] 腾讯移动安全实验室. 腾讯移动安全实验室 2018 年手机安全报告. https://m.qq.com/security_lab/news_detail_489.html, 2018.

[6] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones [c]. In Proceedings of the 9th USENIX conference on Operating systems design and implementation. USENIX, pages 1–6, 2010.

[7] Anthony Desnos and Patrik Lantz. Droidbox: An android application sandbox for dynamic analysis. Lund Univ., Lund, Sweden, Tech. Rep, 2011.

[8] droidbox - apimonitor.wiki. https://code.google.com/archive/p/droidbox/wikis/APIMonitor.wiki.

[9] Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis. In Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12), pages 569–584, 2012.

[10] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In Ndss, 2015.

[11] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. Malton: Towards on-device non-invasive mobile malware analysis for {ART}. In 26th {USENIX} Security Symposium ({USENIX} Security 17), pages 289–306, 2017.

[12] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, pages 15–26. ACM, 2011.

[13] Paul Irolla and Eric Filiol. Glassbox: Dynamic analysis platform for malware android applications on real devices. arXiv preprint arXiv:1609.04718, 2016.

[14] Michalis Diamantaris, Elias P Papadopoulos, Evangelos P Markatos, Sotiris Ioannidis, and Jason Polakis. Reaper: Real-time app analysis for augmenting the android permission system. 2019.

[15] oleavr. Frida. https://www.frida.re/.

[16] rovo89. Xposed. https://github.com/rovo89/Xposed.

[17] ac pm. Inspeckage. https://github.com/ac-pm/Inspeckage.

[18] 孙国梓邱凌志, 张梓雄. Insightdroid—动态追踪 android 应用方法.

[19] 谌力文伟平, 汤炀. 一种基于 android 内核的 app 敏感行为检测方法及实现. 信息网络安全, (8):18–23, 2016.

[20] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In ACM Sigplan notices, volume 42, pages 89–100. ACM, 2007.

[21] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. Adaptive unpacking of android apps. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 358–369. IEEE, 2017.

[22] 丰生强. Android 软件安全权威指南. 电子工业出版社, 2019.

[23] Yiming Jing, Ziming Zhao, Gail Joon Ahn, and Hongxin Hu. Morpheus: Automatically generating heuristics to detect android emulators. In Computer Security

Applications Conference, 2014.

[24] 360 加固保. http://jiagu.360.cn/#/global/index.

致　　谢