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.

本文设计思路结合了 hook 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 the implementation, advantages and disadvantages of some existing Android application behavior monitoring systems, and present a high-performance application dynamic behavior capture system, which can run on Nexus 5x and is implemented by using hook technology 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 brought by hook technology and the stability and high performance brought by modification of source code, which can be used for designing similar tools of 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

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[8], 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[10], 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[9]。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[9]. 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%[7]。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[14]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[5] 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[3] 动态分析系统。该系统使用了 Taintdroid[5] 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[1]。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[13] 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, 而部分恶意应用会检测虚拟机运行环境从而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[11] 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, 受恶意应用针对虚拟机攻击的影响 2. 没有对 Java 层进行具体的监控, 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[12] 的动态分析系统。

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。

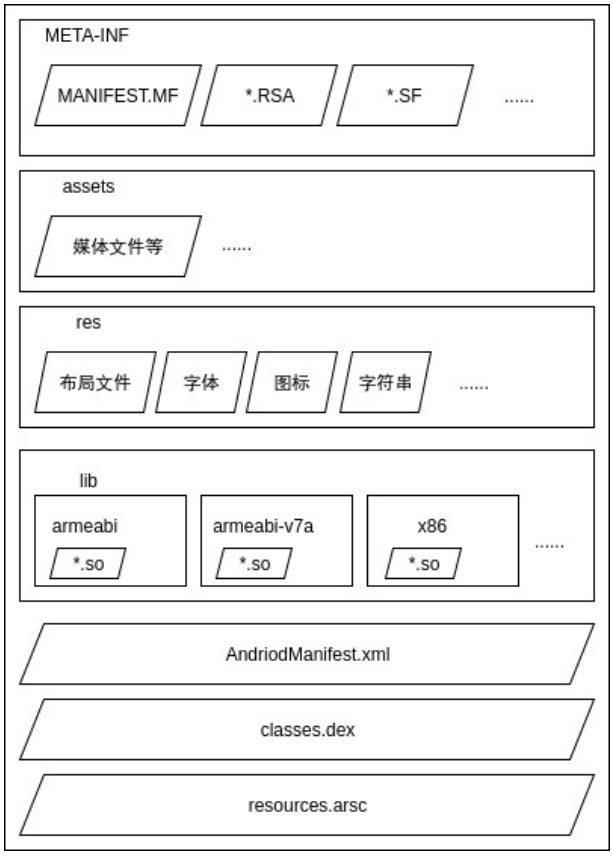

图 2.2 APK 文件结构

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过程)。

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

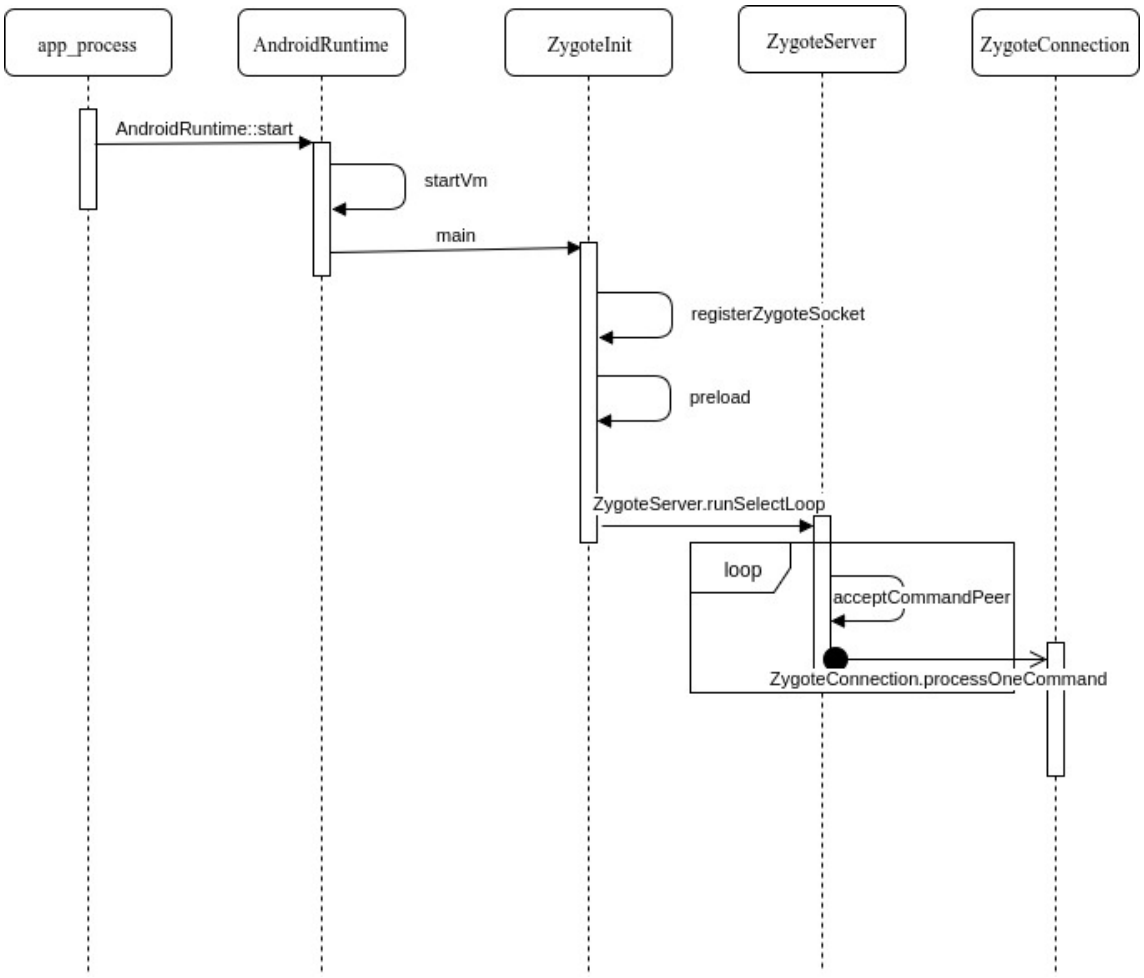

图 2.3　Zygote 进程启动过程

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 方法参数来然 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。

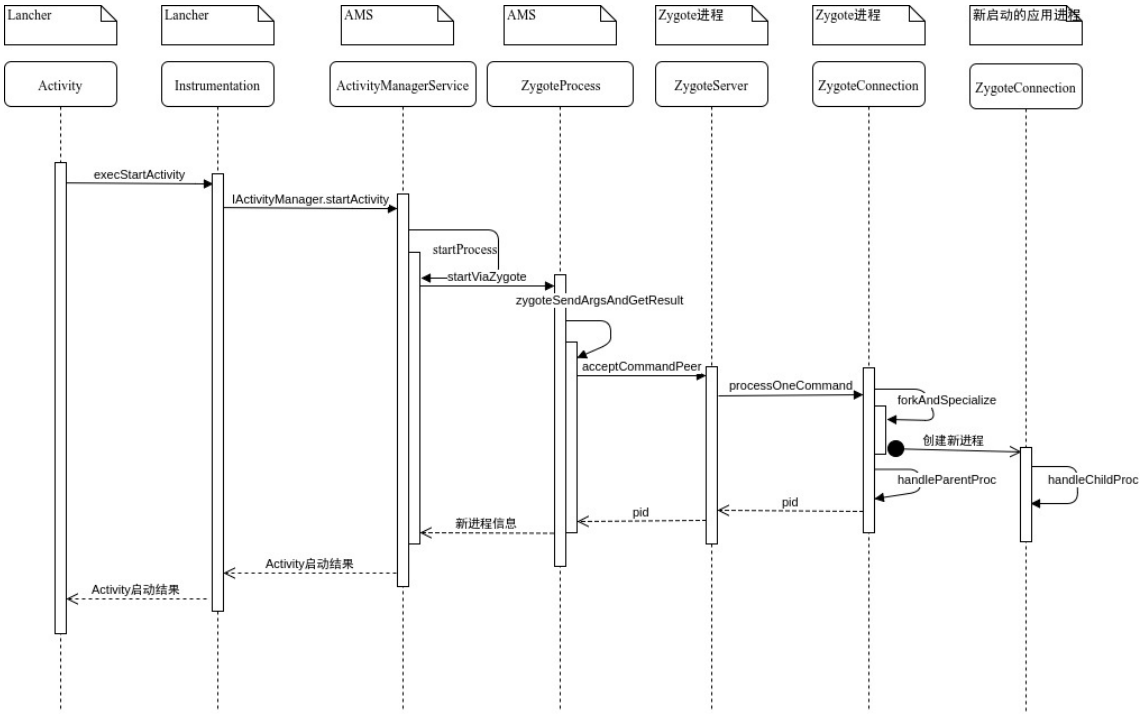

图 2.4　应用启动过程-1

2.3.2　**类的加载**

2.3.3　**方法的执行**

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。

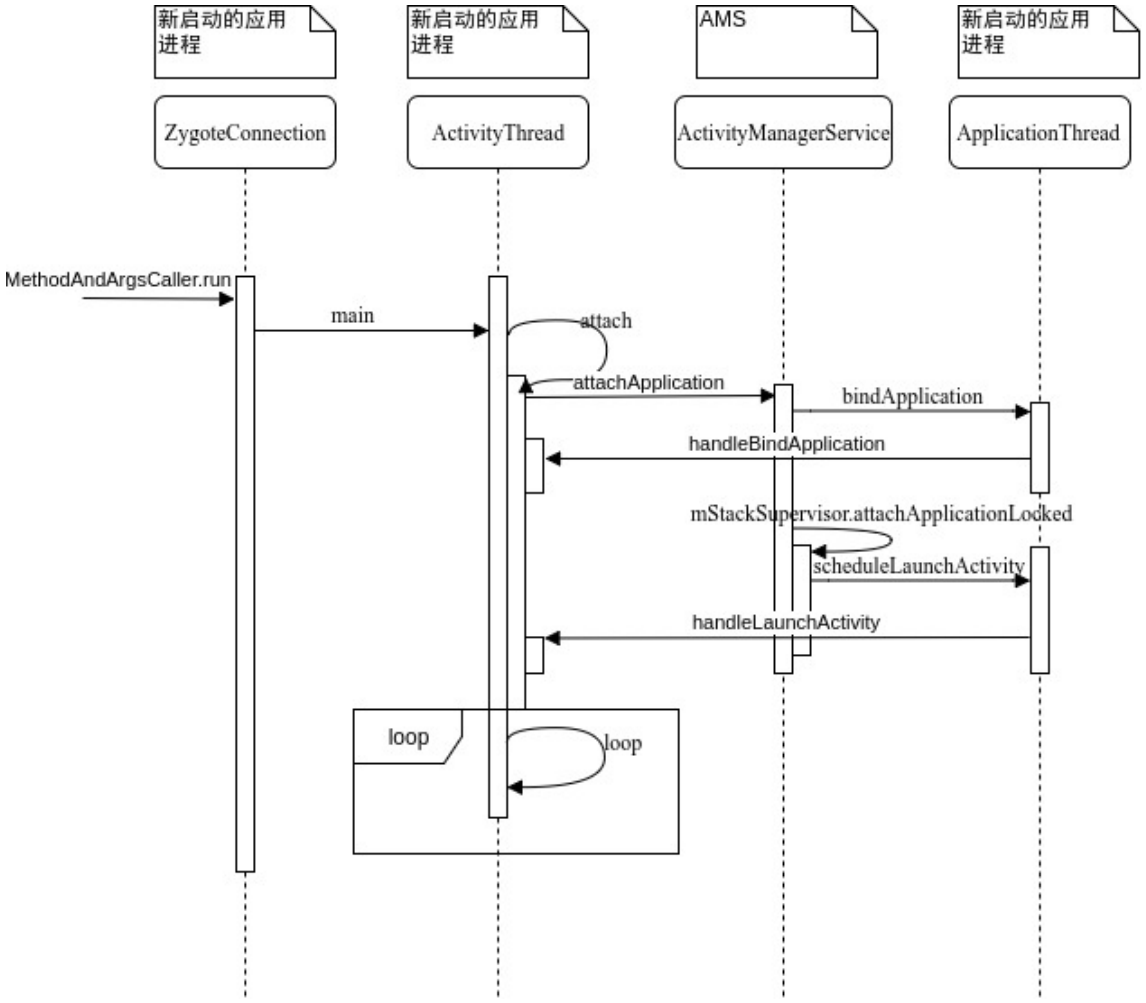

图 2.5　应用启动过程-2

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[11]、DroidScope[13] 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[2]、Glassbox[6] 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[1] 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[13]、Glassbox[6] 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[4] 使用了该技术实现动态监控。

2.4.6 Frida

Frida 是一个著名的开源 hook 框架。

2.4.7 Xposed

2.4.8 Valgrind

2.5 Android 应用加固技术

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名称混淆、方执行混淆、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、反调试等。

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。

3　系统设计实现

3.1　概览

综合本文前面部分的分析和实际情况, 本系统采用 ART Instrumentation 的技术, 通过修改 Android8.1 系统源代码并结合 hooking 技术, 利用 frida 来实现捕获移动应用动态行为的功能。系统的总体设计如图3.1所示。

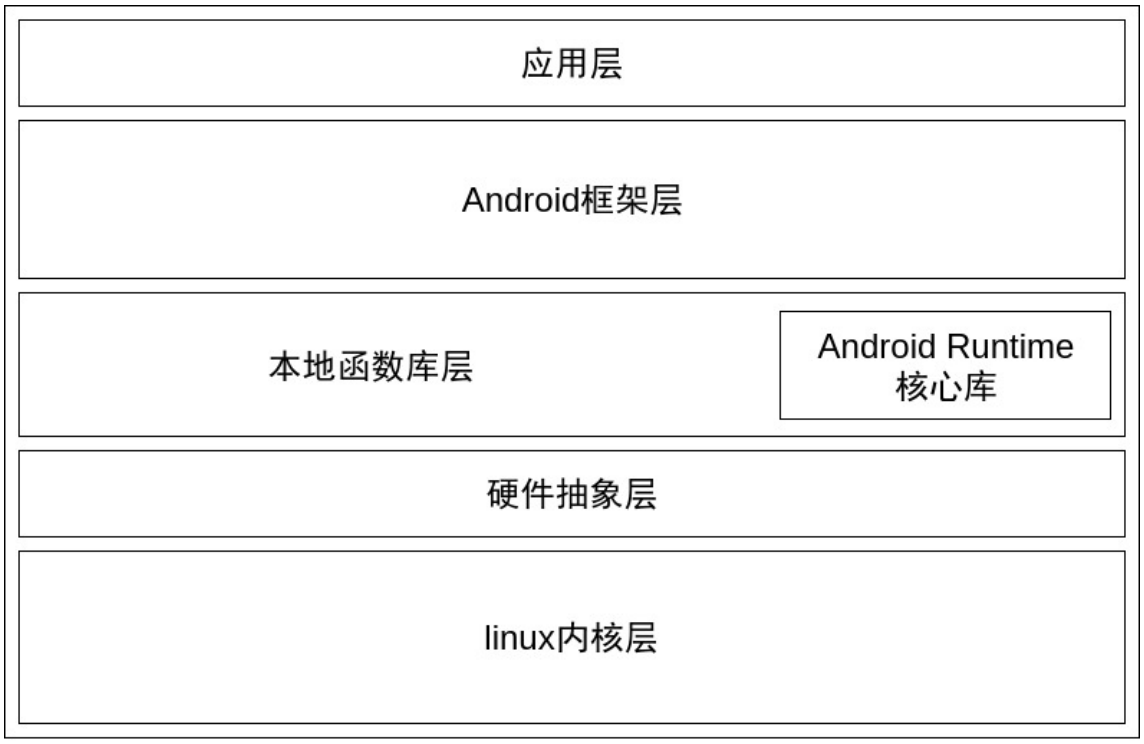

图 3.1　CMonitor 概览

3.2 **启动监控**

3.3 Java **方法调用监控**

3.4 **本地函数调用监控**

3.5 **脱壳功能**

3.6 log **系统**

4　实验与结果分析

4.1　监控数据

4.2　性能

5　总结与展望

本系统还不够成熟

参考文献

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

[2] I. Burguera, U. Zurutuza, and S. 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.

[3] A. Desnos and P. Lantz. Droidbox: An android application sandbox for dynamic analysis. Lund Univ., Lund, Sweden, Tech. Rep, 2011.

[4] M. Diamantaris, E. P. Papadopoulos, E. P. Markatos, S. Ioannidis, and J. Polakis. Reaper: Real-time app analysis for augmenting the android permission system. 2019.

[5] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. 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.

[6] P. Irolla and E. Filiol. Glassbox: Dynamic analysis platform for malware android applications on real devices. arXiv preprint arXiv:1609.04718, 2016.

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

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

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

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

1729/app-stores/.

[11] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In Ndss, 2015.

[12] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu. Malton: Towards on-device non-invasive mobile malware analysis for {ART}. In 26th {USENIX} Security Symposium ({USENIX} Security 17), pages 289–306, 2017.

[13] L. K. Yan and H. 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.

[14] 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.

致　　谢