编号_____

西安科技大学

毕业设计（论文）

（ 2018 届）

题　　目	基于代码植入的 Android APP 运行时信息提取
学生姓名	胡宝全
学　　号	1408010120
专业班级	软件工程 1401 班
指导教师	刘晓建
所在学院	计算机科学与技术学院
日　　期	2018 年 6 月

西安科技大学

学 位 论 文 诚 信 声 明 书

本人郑重声明：所呈交的学位论文（设计）是我个人在导师指导下进行的研究（设计）工作及取得的研究（设计）成果。除了文中加以标注和致谢的地方外，论文（设计）中不包含其他人或集体已经公开发表或撰写过的研究（设计）成果，也不包含本人或其他人在其它单位已申请学位或为其他用途使用过的成果。与我一同工作的同志对本研究（设计）所做的任何贡献均已在论文中做了明确的说明并表示了致谢。

申请学位论文（设计）与资料若有不实之处，本人愿承担一切相关责任。

学位论文（设计）作者签名：　　日期：

学 位 论 文 知 识 产 权 声 明 书

本人完全了解学校有关保护知识产权的规定，即：在校期间所做论文（设计）工作的知识产权属西安科技大学所有。学校有权保留并向国家有关部门或机构送交论文的复印件和电子版。本人允许论文（设计）被查阅和借阅；学校可以公布本学位论文（设计）的全部或部分内容并将有关内容编入有关数据库进行检索，可以采用影印、缩印或其它复制手段保存和汇编本学位论文。

保密论文待解密后适用本声明。

学位论文（设计）作者签名：　　指导教师签名：

年　　月　　日

分 类 号　　密 级

学校代码 10704　　学 号 1408010120

西安科技大学

学 士 学 位 论 文

题目：基于代码植入的 Android APP 运行时信息提取

作者：胡宝全

指导教师：刘晓建　　专业技术职称：副教授

学科专业：软件工程　　申请学位日期：2018 年 6 月

摘　要

Android 作为 Google 公司开发的移动操作系统，以其开放性和易用性的特点，拥有大量的用户和应用开发者。根据 Gartner 的统计数据，截止 2018 年 2 月，Android 市场占有率已达 86%。然而,由于 Android 系统自身的开放性和应用的分发渠道的不规范性，造成该平台已成为了移动互联网恶意程序传播的主要平台。因此，如何高效检测有敏感信息泄露可能的 Android 应用程序，防止敏感信息被恶意使用成为了学术界和工业界广泛关注的话题。

为了应对日益增长的恶意软件，很多的安全分析人员，依旧是通过计算机端对 APK 进行人工分析、校验恶意代码的检测。现有工具对样本进行逆向操作，大都采用静态分析，即程序逻辑完全依靠分析人员的经验和精力来进行分析判断,在时间和效率上都大打折扣。本课题基于代码植入的 Android App 运行时信息提取方法，可以自动完成静态植入与动态分析，大大提高了检测效率。首先，确定调用 Intent 函数的位置，在调用函数后插入获取运行时信息的代码。然后,采用 MonkeyRunner 驱动程序自动运行,输出 Intent 通信以及敏感 API 调用信息，从而获取应用运行时的关键信息。这种基于代码植入动态的检测方法，提高了信息的准确性，能够获取的程序运行时的大量信息，有助于对安全性的进一步分析。

关键词：Android 反编译；smali 代码植入；Android 自动测试；程序行为提取

删除[dell-pc]: 如: CNCERT 数据显示,2015 年检测到的 147.2 万个移动互联网恶意程序中,基于 Android 平台的占比在 99.6%以上；又如:360 互联网安全中心 2016 年共截获 Android 平台新增恶意程序样本 1403.3 万个，平均每天新增多达 3.8 万个恶意程序样本。

删除[dell-pc]: 目前

删除[dell-pc]: 这样的方法笨重，而且重复性工作过多，

删除[dell-pc]: 在不断分析的过程中也难免会出现遗漏和疏忽的地方。

删除[dell-pc]: 最后

删除[dell-pc]: 根据应用 UI 控件进行有针对性的

删除[dell-pc]: 动态

删除[dell-pc]: 测试

删除[dell-pc]: 看程序运行过程中是否真正调用这些函数和

删除[dell-pc]: 各种

删除[dell-pc]: 获取

删除[dell-pc]: 可靠

删除[dell-pc]: 运行内部信息也比较多

ABSTRACT

Android is a mobile operating system developed by Google. With its openness and ease of use, Android has a large number of users and application developers. According to Gartner's statistics, as of February 2018, Android market share has reached 86%. However, due to the openness of the Android system itself and the non-standardized distribution of the application channels, the platform has become the main platform for the spread of malicious programs on the mobile Internet. For example: CNCERT data shows that of the 1,472,000 mobile Internet malicious programs detected in 2015, 99.6% were based on the Android platform. Another example: 360 Internet Security Center intercepted a new malicious program sample for the Android platform in 2016. Ten thousand, adding up to 38,000 malicious program samples every day. Therefore, how to efficiently detect potentially leaky Android applications with sensitive information and prevent malicious use of sensitive information has become a topic of widespread concern in academia and industry.

In order to deal with the growing number of malicious software, many security analysts still perform manual analysis, verification, and detection of malicious code through the computer. At present, the existing tools perform reverse operations on the samples, and most of them use static analysis. That is, the program logic completely relies on the analysts' experience and energy to analyze and judge. Such methods are bulky and have too many repetitive tasks. They are both large in terms of time and efficiency. discount. In the process of continuous analysis, it is also inevitable that there will be omissions and omissions. This topic? Android App runtime information extraction method based on code implantation, can automatically complete the static implantation and dynamic analysis, greatly improving the detection efficiency. First, determine the location of the call to the Intent function and insert the code to get runtime information after calling the function. Finally, according to the application of UI controls for targeted dynamic testing, see if the program is actually running during the process of running these functions and sensitive APIs, in order to obtain a variety of application runtime information. This dynamic detection method based on code implantation improves the reliability of the information obtained, and the internal information of the program that can be acquired is also relatively large.

Key Words: Android decompilation; smali code implantation; Android automatic testing; program behavior extraction

目 录

1 绪论

1.1 研究背景及意义

Android 作为 Google 公司开发的移动操作系统，以其开放性和易用性的特点，拥有大量的用户和应用开发者。根据 Gartner[1]的统计数据，截止 2018 年 2 月，Android 市场占有率已达 86%。然而,由于 Android 系统自身的开放性和应用的分发渠道的不规范性, 造成该平台已成为了移动互联网恶意程序传播的主要平台。如: CNCERT 数据显示[2],2015 年检测到的 147.2 万个移动互联网的恶意程序中,Android 平台应用程序的占比在 99.6%以上。又如:奇虎 360 公司互联网安全中心在 2016 年总共截获 Android 平台新增的恶意程序样本有 1403.3 万个, 平均每天新增多达 3.8 万个恶意程序样本[3]。因此, 如何高效检测有敏感信息泄露可能的 Android 应用程序，防止敏感信息被恶意使用成为了学术界和工业界广泛关注的话题。

本课题基于代码植入的 Android App 运行时信息提取，通过对 APK 文件进行解压，将获得的 DEX 文件进行反汇编获得 smali 文件，并分析 smali，通过植入输出代码，实现自动提取调用系统 API 的信息。可以深入并自动化对 APK 文件进行处理，从而获取程序运行时调用哪些系统 API 方法，以及运行时 Intent 的信息等，为检测 APK 文件中是否存在有恶意代码提供了有力并且更加深度的检测方式。目的在于减少了手动分析代码的工作量，并且提取的数据信息更加高效更加准确。

为了应对日益增长的恶意软件，很多的安全分析人员，依旧是通过计算机端对 APK 进行人工分析、校验,恶意代码的检测。目前现有工具对样本进行逆向操作，大都采用静态分析，即程序逻辑完全依靠分析人员的经验和精力来进行分析判断,这样的方法笨重，而且重复性工作过多，在时间和效率上都大打折扣。

本课题基于代码植入的 Android App 运行时信息提取方法, 可以通过动态测试在模拟器或真机上实际运行应用, 收集运行过程中的程序行为来判断应用是否调用敏感 API，以及获取程序运行时 Intent 携带的其他信息。首先, 使用静态分析技术分析 smali 代码中调用的敏感函数，确定调用敏感 API 的位置，在调用后插入获取运行时信息的代码。最后,根据应用 UI 控件的可疑度进行有针对性的动态测试,看程序运行过程中是否真正调用这些敏感 API，从而获取应用运行时的各种信息。这种基于代码植入动态的检测方法，提高了获取信息的可靠性，能够获取的程序运行内部信息也比较多。

1.2 研究现状

Android 系统的日益流行,以及其系统的开源特性使得系统遭受了大量恶意软件的攻击,该问题也引起了国内外信息安全领域诸多学者的注意,目前对如何预防以及检测手机恶意应用程序这一课题也已经做出了大量的分析和研究工作。

卿斯汉[6]对当前的Android安全情况进行了总结,把恶意软件进行了归类,并指出当前的恶意软件检测方式分为静态和动态检测方式。而两种分析方法一般都会基于特征,其中,两个主流方向是基于代码特征[7]和基于行为特征[8]。这些方法大多依赖大规模的静态或者动态分析技术,然后利用得到的信息来判断一个应用是否为恶意应用。

Zhang[9]基于 API 的依赖关系构造 API 依赖图,然后利用图相似性来和数据库中已有图进行相似度比较,确认应用中是否包含恶意行为。采用静态分析的技术是比较快速的,但是仅仅利用静态特征不全面,因为有些恶意行为是在运行时刻触发的,所以准确率有受到影响。

Xiao[10]采用动态检测技术,利用 Monkeyrunner 产 1000 个随机点击事件去执行应用,然后在这个过程中用 strace 工具追踪进程获得所有的系统调用函数,然后以此作为特征来分析应用程序是否包含恶意行为。通过动态检测的方式可以覆盖到那些只有在运行时刻才触发的恶意行为,但是很难保证测试用例完全,而且这种方式开销大。Huang[14]采用动态检测与静态检测态结合的方式检测 Virustotal 提交者上传的应用是否有恶意行为, 静态分析函数方法和静态页面，然后进行包内容分析，主要是分析其中的时间戳和证书信息等来初步判定应用是否为潜在的恶意应用，最后动态分析来确认恶意行为是否存在。这样静动态结合的方式能够有效的综合两种方式的优点和减小缺点，但是该方法是针对 Virustotal 中的恶意软件检测,有着局限性。

针对现有的静态检测恶意软件误报率高, 本文采用代码植入加动态结合的方式获取 Android 应用运行时信息。静态分析 smali 代码, 查看应用中是否直接调用敏感 API, 或通过变量赋值及数组赋值方式反射调用敏感 API,以此来判定一个应用是否为疑似恶意应用;再根据分析结果在敏感 API 调用处植入信息输出代码，最后生成测试脚本, 驱动应用执行, 并对其运行行为信息进行监测, 来确认该应用中是否有恶意行为被触发; 利用 UI 控件和 Android 代码映射分析方法, 将静态分析阶段识别的敏感 API 调用代码与对应的 UI 控件建立关联, 从而进一步可以提高动态确认效率。

1.3 论文结构

本文的主要内容是设计一个基于代码植入和 MonkeyRunner 自动化测试工具的应用程序行为抽取系统，可以对 APK 运行中调用的敏感 API 进行输出，本文共有六章，结构如下：

第 1 章，绪论。本章介绍了研究背景、意义以及研究现状，主要描述了当前阶段 Android 市场的发展状况和面临的安全问题，要处理恶意 APK 对 Android 用户造成的威胁必不可少的步骤就是分析测试 APK。

第 2 章，Android OS 地简洁介绍。在本章中着重介绍了 Android 系统的架构，包括 Linux 内核层、硬件抽象层(HAL)、Android 运行时 ART、Native C/C++ Libraries、Java API Framework、System Apps 这些层各自的功能，还介绍了 Android 中的 Dalvik 虚拟机与 ART 虚拟机的不同，接着介绍了 Android 组件通信的 Intent 机制，紧接着分析了 Android 安全机制及安全机制的不足之处。

第 3 章，技术背景。本章对这次课题涉及到的相关技术做了简明扼要的介绍，包括 Google 的反编译工具 apktool、反编译之后的 smali 语法、apk 签名机制、MonkeyRunner 自动测试测试工具、Android adb 调试桥等等。

第 4 章，基于代码植入的 Android APP 运行时信息提取系统的详细设计。本章对此系统的各模块的原理、设计思路、流程等进行了分析和介绍，并用实例对本系统的有效性进行了测试，同时统计了结果。

第 5 章，系统工具的操作使用。对工具使用流程进行简单介绍。

第 6 章，总结和展望。

2 Android 操作系统介绍

2.1 Android 操作系统架构

Android 是一个开源的，基于 Linux 的软件栈构建而成的操作系统，可以为各种设备提供系统支持。图 2-1 展示了 Android 系统平台的主要架构。

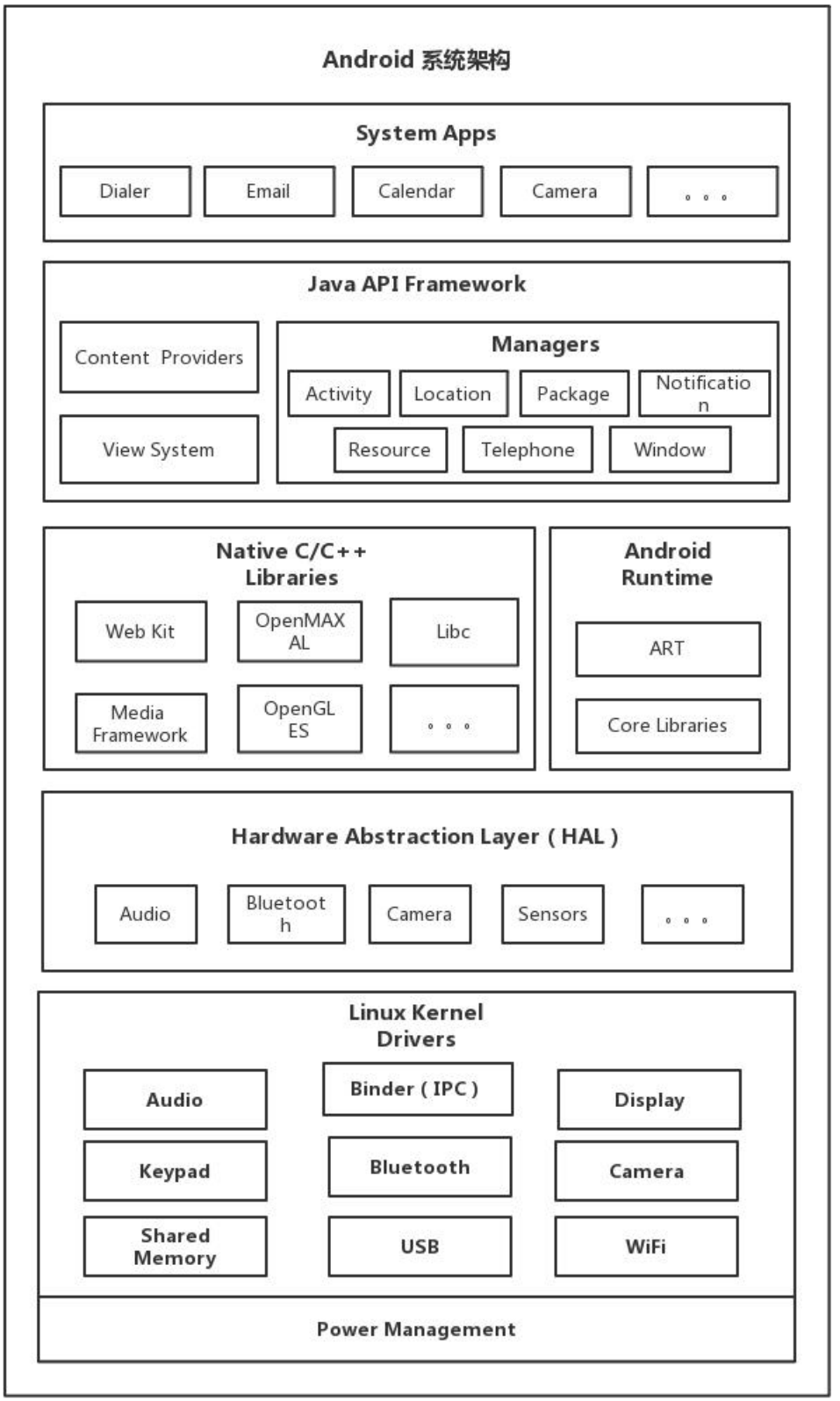


图 2-1 Android 系统架构

Linux 内核层。Linux 内核是 Android 平台能够正常运行的基础和核心。Android 的运行时 ART 底层功能完全需要依赖 Linux 内核来实现，例如内存的管理、进程的管理、驱动的管理、网络的管理等多种核心管理服务。从系统层面来说，Android 系统中统筹软件和硬件协作的中间层是 Linux 内核层。

硬件抽象层(HAL)。硬件抽象层(HAL)给上层提供标准接口，将设备硬件功能提供给更高一级的 Java API 框架层。HAL 由多个库模块组成，每个模块为特定类型的硬件组件实现一个接口，比如摄像头或蓝牙模块。当一个框架 API 方法调用访问设备硬件时，该硬件组件的库模块会被 Android 系统加载。

Android 运行时环境 ART。Android 版本 5.0 后，每个应用都拥有自己独立的进程中，并且拥有自己的 Android 运行时（ART）的实例。 ART 不同于 Android 虚拟运行环境 Dalvik，ART 针对小内存设备进行了优化，使 Dex 可以顺利运行创建进程实例在小内存设备之上。

本地 C/C++核心类库。许多核心的 Android 系统组件和服务，比如 ART 和 HAL，都是用本地代码构建的，这些代码需要用 C 和 C++编写的本地链接库。Android 运行平台提供 Java 框架 API，以将这些本地链接库文件的功能提供给应用程序使用。例如，可以通过 Android 框架的 Java OpenGL API 访问 OpenGL ES，从而在应用程序中添加对绘图和操作 2D 和 3D 图形的支持。如果正在开发一个需要 C 或 C++代码的应用程序，那么可以使用 Android NDK 直接从访问这些编写好的本地平台库。

Java API 框架。Android 的 API 是使用 Java 语言编写的，要编写 Android 软件需要使用大量的 Java API 来实现。

模块化系统组件和服务，包括以下内容:

①视图 View 系统，该系统可扩展，可以自定义 view，视图系统包括许多控件，比如按钮、文本框，列表、等等；

②资源管理器，提供对本地文件的访问；

③通知管理器，允许所有应用程序在状态栏中显示自定义通知；

④活动管理器，管理应用程序生命周期的；

⑤内容提供者，利用它可以共享自己数据给其他应用程序，也可以获取其他应用程序的数据比如，获取联系人，获取拍摄的图片等等。

⑥系统自带应用。Android 自带一套核心应用程序，包括电话、联系人、短信、电子邮件、图片管理、文件管理等等。

2.2 应用程序包结构

一个 Android 程序由多个文件以及文件夹组成，这些文件分别用于不同的功能，常用文件和文件夹如图 2-2 所示。

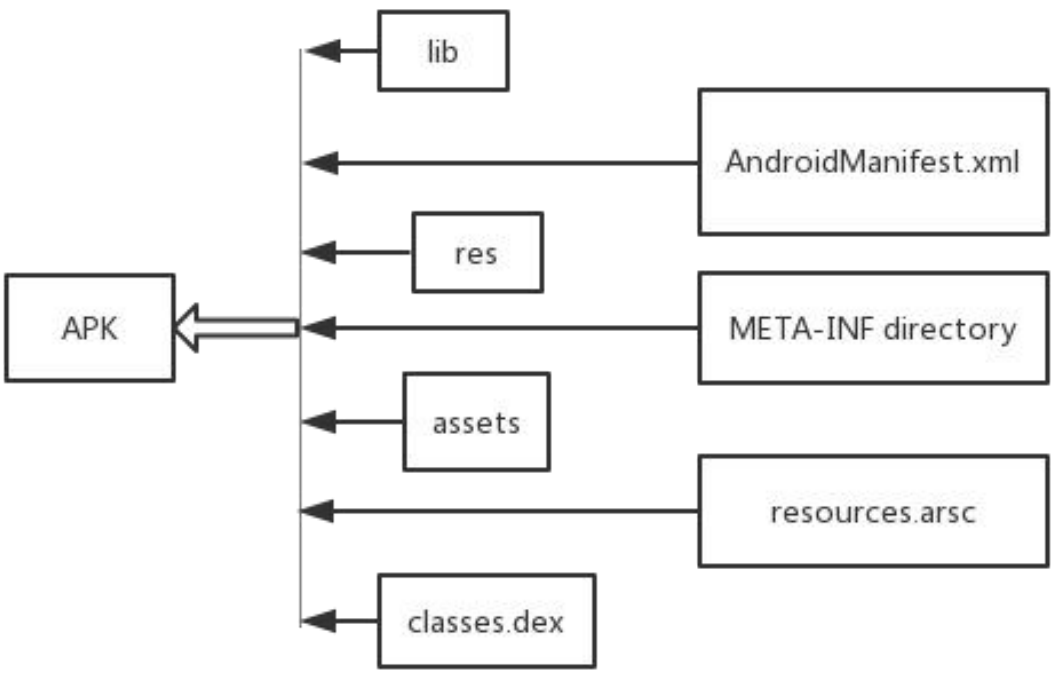


图 2-2 APK 文件结构字体统一

设置格式[dell-pc]: 突出显示

assets 目录：存放打包到 apk 文件中的静态文件

lib 目录：程序依赖的库文件

res 目录：存放应用程序的资源

META_INF 目录：存放应用程序签名和证书的目录

AndroidManifest.xml：Android 程序的配置文件，设置应用运行过程中的各种配置

classes.dex 文件：dex 可执行文件

resources.arsc：资源配置文件

2.3 ART 和 Dalvik 的区别

Dalvik 是 Google 公司早期设计专门用于运行 Android 程序的 Java 虚拟机。Dalvik 虚拟机可以运行 Dex 格式的机器码，每个 apk 中都包含一个或多个这种文件。Dalvik 针对移动设备进行了优化可以运行在内存和处理器速度有限的设备中。Dalvik 允许有限的内存中同时运行多个虚拟机实例，有效避免了应用程序崩溃不会影响其他应用程序的问题。

Android 操作系统发展到现在已经很成熟了，Google 公司 Android 项目成员将重点放在一些运行底层的组件，负责应用程序运行的 Dalvik 运行时是其中之一。Google 开发者已经开发出更加省电速度更快的 Android 运行时虚拟机 ART。 ART 代表 Android Runtime，其处理应用程序执行的方式完全不同于 Dalvik，Dalvik 是依靠一个 Just-In-Time (JIT)编译器去解释字节码。开发者编译后的应用代码需要通过一个解释器在用户的设备上运行，这一机制并不高效，但让应用能更容易在不同硬件和架构上运行。ART 改进了方法，在应用程序安装到设备时就预编译字节码到机器语言，运行时直接执行，这一机制叫 Ahead-Of-Time (AOT）编译。移除解释代码这一过程后，应用程序执行将更有效率，启动也变的更快。

ART 优点：

①系统性能的显著提升。

②应用启动更快、运行更快、体验更流畅、触感反馈更及时。

③更长的电池续航能力。

④支持更低的硬件。

ART 缺点：

①更大的存储空间占用，可能会增加 10%-20%。

②更长的应用安装时间。

总的来说 ART 的功效就是“空间换时间”

2.4 Android 组件通信 Intent 机制

Activity、Content Provider、Broadcast Receiver 和 Service 是 Android 应用的四大组件，构成应用程序运行的基础，它们的交互模型如图 2-3 所示。

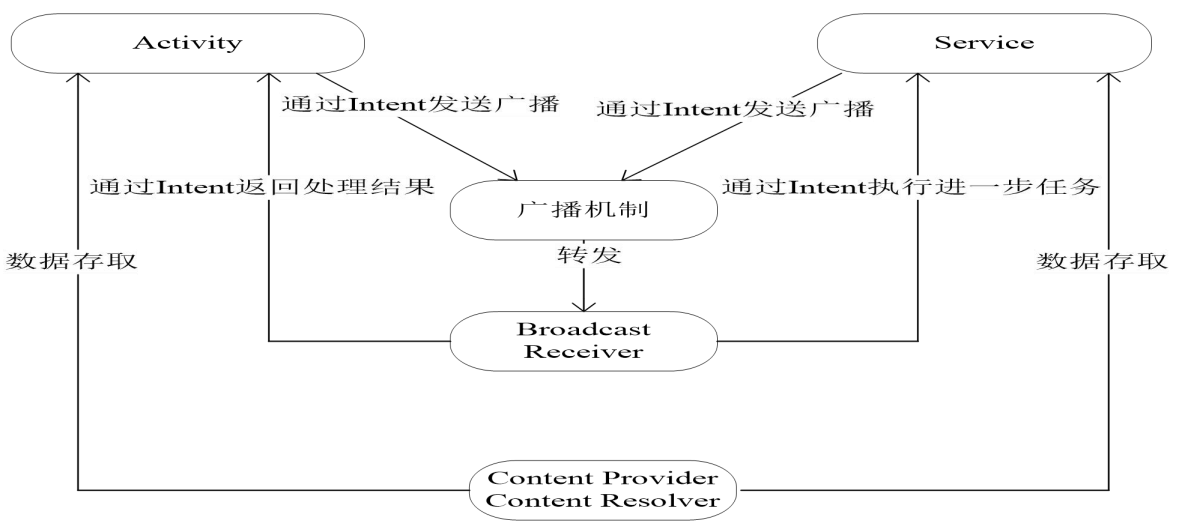


图 2-3 四大组件交互通信模型

Intent 用于各个组件之间的跳转，并且可以携带数据，在程序运行时连接两个组件；Intent 分为显式 Intent 和隐式 Intent，显式 Intent 常用在本应用之间组件的跳转，隐式 Intent 常用于在不同应用之间组件的跳转。通过 Intent 机制，你可以向 Android 系统提交一个请求，Android 系统会根据 Intent 的意愿来选择合适的组件来响应改 Intent 请求。它可以与 startActivity 一起使用以启动一个 Activity，Intent 广播将其发送广播，注册改事件的广播接收该广播可以做出反应。Service 可以使用 startService（Intent）或 bindService（Intent，ServiceConnection，int）与后台服务进行通信。

Activity 是 Android 应用程序中最基本的组件，每个应用程序中，可以有一个或多个 Activity，但是必须有一个入口的 Activity 当程序打开时首先显示给用户。Activity 展示在屏幕上，可以与程序进行交互操作。可以通过向 startActivity()传递一个意图来启动一个新的活动实例，Intent 可以携带一些数据到要启动的活动。如果希望在结束时收到活动的结果，使用 starticvityforresult()方法，在本活动的 onActivityResult()函数回调中接收调用结果。

Service 没有用户界面，它同样是一个 Android 组件，它通常不用和用户进行交互，也没有图形用户界面，它会一直运行在后台，它具有自己的生命周期。可以使用服务类的方法启动服务，将 Intent 传递给 startService()来启动一个服务来执行一次性操作(例如下载文件)。如果服务是 C/S 接口设计的，您可以通过将意图传递给 bindService()来绑定

到服务。

Broadcase Receiver 翻译过来就是是广播消息接收器，它不负责执行某个任务。它其实相当于一个全局的监听器。开发者创建自己的类继承 Broadcast Receiver 这个类，然后重写 onReceive()方法，当其他组件发送广播消息时，如果 IntentFilter 中配置的消息与其相同，则 onReceiver()方法被触发。

2.5 Android 系统安全机制

2.5.1 系统和内核安全

在操作系统层面，Android 平台提供了 Linux 内核的安全性，以及安全的进程间通信（IPC）功能，可在不同进程中运行的应用程序之间实现安全通信。操作系统级别的这些安全功能旨在确保即使是本机代码也要受应用沙盒的限制。无论该代码是包含应用程序行为的结果还是应用程序漏洞的利用，该系统都旨在防止流氓应用程序损害其他应用程序、Android 系统或设备本身。

Linux 安全：

Linux 内核是 Android 平台能够正常运行的基础和核心。Linux 内核已被广泛使用多年，并在数百万安全敏感环境中使用。通过数千计开发人员不断研究，攻击和修复的历史，Linux 已成为众多公司和安全专业人士信任的安全稳定的内核。

作为移动计算环境的基础，Linux 内核为 Android 提供了几个关键的安全功能，其中包括：

- 基于用户的权限模型
- 进程隔离
- 用于实现安全 IPC 的可扩展机制
- 能够移除不必要的和可能不安全的部分内核

删除[dell-pc]:

Linux 是多用户系统，Linux 内核基本的安全目标是隔离用户资源，使用户资源不受其他因素影响。因此，Linux 可以提供一下支持：

- 阻止用户 A 读取用户 B 的文件
- 确保用户 A 不会耗尽用户 B 的内存
- 确保用户 A 不会耗尽用户 B 的 CPU 资源
- 确保用户 A 不会耗尽用户 B 的设备（例如电话，GPS，蓝牙）

应用程序沙箱

Android 操作系统利用 Linux 系统基于用户的保护机制来隔离应用程序资源。Android

系统为每个 Android 应用程序都分配一个唯一的用户 ID（UID），并将 UID 作为该程序在单独的虚拟机进程中运行的身份。在 Android 系统中，应用程序可以以不同用户权限运行。

这样就相当于设置了一个内核级别应用沙盒。内核会在进程级别通过标准的 Linux 内容（例如分配给应用程序的 UID 和组 ID）使其强制执行应用和系统之间的安全功能。默认情况下，应用不能彼此交互，而且应用对操作系统的访问权限会受到限制。假如应用 X 尝试执行恶意操作，获取应用 Y 的数据，操作系统会组织该操作。或者应用想要拨打电话或发送短信，系统会判断该应用是否有足够的权限，如果没有权限组织操作。

由于应用沙盒位于内核中，因此该安全模型的保护范围扩展到了本机代码和操作系统应用。位于内核上方的所有软件（例如，操作系统库、应用框架、应用运行时和所有应用）都会在应用沙盒中运行。在某些平台上，为了强制执行安全功能，会限制开发者只能使用特定的开发框架、API 组或语言。在 Android 上，并没有限制必须如何编写应用才能强制执行安全功能；在这一方面，native 代码与解释代码一样安全。

在某些操作系统中，一个应用中的内存损坏错误可能会导致位于同一内存空间中的其他应用出现损坏，进而导致设备的安全性完全遭到破坏。但在 Android 系统上，所有应用及其资源都在操作系统级别的沙盒内，因此，如果出现内存损坏错误，将只有在位于内存错误的应用的环境中才能发生随意意执行代码的行为，而且只能是以操作系统确立的权限执行代码。

与所有安全功能一样，应用沙盒并不是坚不可摧的。不过，要在经过适当配置的设备上攻破应用沙盒这道防线，必须要先攻破 Linux 内核的安全功能。

系统分区和安全模式

设置格式[dell-pc]: 字体: 加粗

设置格式[dell-pc]: 缩进: 首行缩进: 8.5 毫米

Android 系统分区包含 Android 的内核以及操作系统运行库，Android 应用程序运行时，应用程序框架和系统应用程序，此分区设置为只读。当用户将设备引导至安全模式时，第三方应用程序可能会由设备所有者手动启动，但默认情况下不会启动。

文件系统权限

设置格式[dell-pc]: 字体: 加粗

设置格式[dell-pc]: 缩进: 首行缩进: 8.5 毫米

在 UNIX 风格的环境中，一个用户不能更改或读取另一个用户的文件是文件系统权限确保的。在 Android 的情况下，每个应用程序都以自己的用户身份运行。除非开发人员与其他应用程序明确共享文件，否则由一个应用程序创建的文件不能被其他应用程序读取或更改。

安全增强的 Linux

设置格式[dell-pc]: 字体: 加粗

设置格式[dell-pc]: 缩进: 首行缩进: 8.5 毫米

Android 使用安全增强型 Linux（SELinux）来控制应用访问策略，并在进程上建立强制访问控制。

序号怎么来的（6）验证启动

设置格式[dell-pc]: 突出显示

Android 6.0 之后支持验证启动的功能和 device-mapper-verity。在启动过程中，无论是在哪个阶段，都会在进入下一个阶段之前先验证下一个阶段的完整性和真实性。Android 7.0 及更高版本支持严格强制执行的验证启动，这意味着遭到入侵的设备将无法启动。

（7）加密

设置格式[dell-pc]: 突出显示

这些包括标准和常用的加密基元的实现，如 SHA、DSA、RSA、和 AES 等。另外，还为 SSL 和 HTTPS 更高级别的协议提供 API。Android 4.0 引入了 KeyChain 类，允许应用程序使用私钥和证书链的系统凭证存储。

（8）设备的 ROOT 权限

设置格式[dell-pc]: 突出显示

一般情况下，在 Android 系统中以 root 权限运行的只有内核和一小部分核心应用程序。Android 系统并不会阻止具有 root 权限的用户或应用程序去修改操作系统内核或任何其他应用程序。通常，root 拥有对所有应用程序和所有应用程序数据的完全访问权限。更改 Android 设备权限以授予应用程序 root 权限的 Android 用户会增加恶意应用程序的安全风险和潜在的应用程序缺陷。

作为 Android 开发人员应该具备修改 Android 操作系统的能力。因为，在许多 Android 设备上，用户可以解锁引导加载程序以允许安装其他操作系统。这些备用操作系统可能允许所有者获得 root 访问权限，以用于调试应用程序和系统组件，或访问未通过 Android API 呈现给应用程序的功能。

（9）用户安全功能

Android 3.0 之后提供完整的文件系统加密，所有用户数据都可以在内核中加密。Android 5.0 之后版本支持文件全盘加密。全磁盘加密使用一个用用户设备密码保护的密钥来保护整个设备的用户数据分区。启动后，用户必须在磁盘的任何部分可访问之前提供其凭据。Android 7.0 和更高版本支持基于文件的加密。基于文件的加密允许使用不同的密钥对不同的文件进行加密，这些密钥可以独立解锁。

（10）设备管理

Android 2.2 之后的版本提供了 Android 设备管理 API，可在系统级提供设备管理功能。例如，用户在丢手机后利用远程擦除手机数据或者锁定设备。除了用于 Android 系统附带的应用程序外，这些 API 还可用于设备管理解决方案的第三方应用使用。

2.5.2 应用安全

（1）Android 权限模型，访问受保护的 API

Android 上的所有应用均在应用沙盒（本文档的前面对其进行了介绍）内运行。应用程序只能访问系统内特定的系统资源，Android 系统负责控制管理 Android 程序能够访问的资源。如果资源使用不当或被恶意使用，可能会给用户体验、网络或设备上的数据带来不利影响。

实现这些限制可以通过多种不同的形式。有些功能会因 Android 有意未提供针对敏感功能的 API（例如，Android 中没有用于直接操控 SIM 卡的 Android API）而受到限制。在某些情况下，角色分离能够提供一种安全措施，就像按应用隔离存储空间一样。在其他情况下，敏感 API 只能提供给可信任的应用使用，由一种称为“权限”的 Android 安全机制进行保护。

这些受保护的 API 包括：

- 相机功能
- 位置数据（GPS）
- 蓝牙功能
- 电话功能
- 短信/彩信功能
- 网络/数据连接

以上这些敏感资源只能通过操作系统间接进行访问。要使用设备上受保护的 API，应用必须在其清单中配置所需的权限。在准备安装应用时，系统会向用户显示一个对话框，其中会注明应用请求的权限并询问是否继续安装。如果用户确认安装，那么系统默认应用程序拥有权限列表中的权限，在 Android 6.0 之前用户不能单独授予或拒绝个别权限，而是必须要一起授予或拒绝应用请求的所有权限。

获得授权后，应用只要安装在设备上，便会拥有这些权限。为了避免用户混淆，系统不会再次通知用户向应用授予的权限，而核心操作系统中包含的应用或由原始设备制造商 OEM 绑定的应用不会向用户请求权限。应用在卸载后，该应用拥有的权限也会被移除，因此如果用户之后重新安装卸载的应用，Android 系统会再次显示应用请求的权限。

在设备设置菜单中，用户可以查看之前安装的应用的权限。此外，用户还可以根据需要在全局范围内停用某些功能，例如停用 GPS、无线功能或 WLAN。

如果应用尝试使用未在其清单中声明的受保护功能，权限失败通常会导致系统向应用抛回一个安全异常。受保护 API 权限检查会在最底层被强制执行，以防止出现规避行为。

Android 还提供了新的 IPC 机制：

• Binder ：一种基于功能的轻量型远程过程调用机制，在执行进程内调用和跨进程调用时能够实现出色的性能。

• Service：Service 可提供能够使用 Binder 直接访问的接口。

• Intent ：Intent 是简单的消息对象，表示想要执行某项操作的“意图”。例如，如果你的应用想要显示某个网页，则会创建一个 Intent 实例并将其传递给系统，以此来表示想要访问相应网址的“意图”。然后，系统会找到一些知道如何处理该 Intent 的其他应用，然后处理该意图。Intent 也可用于在系统范围发送广播，启动服务，跳转界面等。

• 内容提供者：ContentProvider 是 Android 系统一个数据存储库，用于访问设备上其它应用的数据，或者共享自己数据给别的应用；典型的示例就是用于访问用户通讯录的 ContentProvider。应用可以访问其他应用通过 ContentProvider 提供的数据，还可以定义自己的 ContentProviders 来提供自己的数据。

虽然可以使用其他机制（例如，网络套接字或全局可写文件）来实现 IPC，但上面这些都是建议使用的 Android IPC 框架。

2.6 Android 系统安全机制的缺陷

Android 操作系统存在以下漏洞和问题：

（1）签名机制管理松散。为了提高 Android 系统的开放性，Google 采用了数字自签名的方法，这种宽松的签名证书意味着不用数字证书认证机构授权，可以随意发布应用程序[16]，意味着恶意程序也能随意发布至网上，因此导致恶意程序泛滥。

（2）Android 应用程序权限的控制只针对单个的应用程序。应用程序权限控制针对单个应用程序能有效地防止权限越级，但是多个应用在组合以后，能轻易提升权限，达到隐式权限提升[17]，使得恶意应用获得更加高级的权限，导致用户信息泄露、Android 系统遭受破坏，这也是目前很多 Android 恶意应用程序获得运行的主要途径。

3 技术背景

3.1 反编译工具 Apktool

ApkTool 可以将 APK 文件反编译成 smali 机器码，植入代码时需要 smali 文件，所以 apktool 必不可少。

• 使用控制台命令行，进入 apktool.jar 和 xxx.apk 所在的文件夹（将 apk 和工具置于同一个文件夹，方便操作）

• 输入 java -jar apktool.jar，可以看到相关的使用命令的提示，

```
>>java -jar apktool_2.3.1.jar
Apktool v2.3.1 - a tool for reengineering Android apk files
with smali v2.2.2 and baksmali v2.2.1
Copyright 2014 Ryszard Wi?niewski <brut.alll@gmail.com>
usage: apktool
 -advance,--advanced   prints advance information.
 -version,--version    prints the version then exits
 -p,--frame-path <dir>   Stores framework files into <dir>.
usage: apktool d[ecode] [options] <file_apk>
 -f,--force             Force delete destination directory.
 -o,--output <dir>      The name of folder that gets written. Default is apk.out

 -s,--no-src            Do not decode sources.
usage: apktool b[uild] [options] <app_path>
 -f,--force-all         Skip changes detection and build all files.
 -o,--output <dir>      The name of apk that gets written. Default is dist/name.apk
For additional info, see: http://ibotpeaches.github.io/Apktool/
For smali.baksmali info see: http://github.com/JesusFreke/smali
```

• 再输入 java -jar apktool.jar d xxx.apk，可以完成反编译得到 smali 文件

• 再使用命令 java -jar apktool.jar b [需要要打包的文件夹]，可以可完成重打包工作

3.2 APK 签名机制

所有的 Android 应用程序在发布时都要被打包成.apk 文件，APK 文件在发布时都必须进行签名，这与在信息安全领域中所使用的数字证书的用途有所不同，它是应用包用来进行自我认证的一种机制。它不需要权威的数字证书签名机构的认证，而是由开发者自己来进行数字证书的使用和控制。Android 系统利用数字签名来对应用程序的作者进行标识，并在应用程序间建立一种相互的信任关系，而不是用来判定应用程序是否应该被安装。

Android 的签名有发布模式和调试模式，在 Android studio 中开发时安装到模拟器的过程中 IDE 自动使用调试模式进行签名，调试模式的应用只能用于开发者调试使用。当

开发者需要在 Google Play 或其他 APP Store 市场上将自己的应用程序发布时，就必须用安卓的签名机制使用其私有密钥签署应用，在应用安装时对其进行验证，用以实现对应用程序安装时的来源鉴定。

APK 文件中包含了应用程序的.dex 可执行文件以及布局、配置、图片等其他非代码资源。Android 要求所有的应用程序(代码和非代码资源)都进行数字签名，从而使应用程序的作者对该应用负责。只要证书有效，且签名通过验证，签名后的文件才是有效的才能安装到设备之中。

Apk 签名命令，将重打包的 apk 文件进行签名：

```
>>jarsigner -verbose -keystore [签名秘钥] -signedjar [签名后的apk名称]  [签名前的apk]  [签名秘钥别名]
```

3.3 MonkeyRunner 测试工具

MonkeyRunner 测试工具可以提供在 Android 代码外编写控制 Android 设备或虚拟机程序运行的 API。可以使用 MonkeyRunner 编写 Python 程序来安装 Android 应用程序或测试包，运行应用，向应用发送按键，截图并把图片存放到工作目录。MonkeyRunner 工具主要用来在功能/框架层测试应用和设备，执行单元测试组，但你也可以用作其他目的。

MonkeyRunner 工具为 Android 测试提供了以下独特功能：

• 多设备控制：可以组成测试组一次测试多个 apk 应用程序。你可以一次性手动链接所有设备或启动所有模拟器（或两者同时进行），也可以通过编程方式操作，然后执行一或多组测试。

• 功能测试：MonkeyRunner 可以在 Android 应用上执行完整的自动化测试。可以模拟人工按键或触摸测试，并截图查看测试结果。

• 回归测试：MonkeyRunner 可以通过运行应用并将截图与已知正确的截图集合相比较来测试应用稳定性。

• 可扩展自动化的工具：MonkeyRunner 提供 API，允许开发人员使用这些 API 用 python 编写测试脚本，来测试 apk 的功能。

3.4 Android 调试桥 ADB

Android 调试桥是指 adb.exe 工具（Android Debug Bridge，ADB），存在于 SDK 的 platform-tools 目录中，允许开发人员与模拟器或者连接的 Android 设备进行通信。为了使用该指令快速完成某项操作需要将 adb.exe 所在的目录配置到 path 环境变量中，之后就可以在命令行窗口中使用了。

ADB 的常见指令介绍如下：

- adb start-server：开启 adb 服务。
- adb devices：列出所有设备。
- adb connect ip:port 连接设备
- adb logcat：查看日志。
- adb kill-server：关闭 adb 服务。
- adb shell：挂载到 Linux 的空间。

4 基于代码植入的 Android 运行时信息提取系统的设计实现

4.1 系统总体框架

本系统的实现思路是：

（1）首先利用 APKTOOL 工具对需要分析的 apk 包进行反编译，得到其 smali 代码；

（2）然后对 smali 代码进行日志插桩，即遍历程序中所有的 smali 文件，定位到 Intent 调用处，在其有 Intent 调用的代码的下一行注入 smali 语法格式的日志代码段；

（3）利用 APKTOOL 工具回编译，得到新的未签名的 apk，未签名的 apk 不能运行在安卓设备上，所以还需要利用 signAPK 这个工具进行签名，签名完成之后在命令提示符中用命令的方式将签名之后的 apk 安装在安卓模拟器上；

（4）利用 Android SDK 自带的随机测试工具 MonkeyRunner 进行反编译后的 apk 测试。程序运行到插入的代码段时，就会把调用信息打印在屏幕并保存到日志文件；

（5）随机测试结束后，分析日志文件 Logcat，就可以获取程序运行时的调用信息。

具体过程如图 4-1 所示。

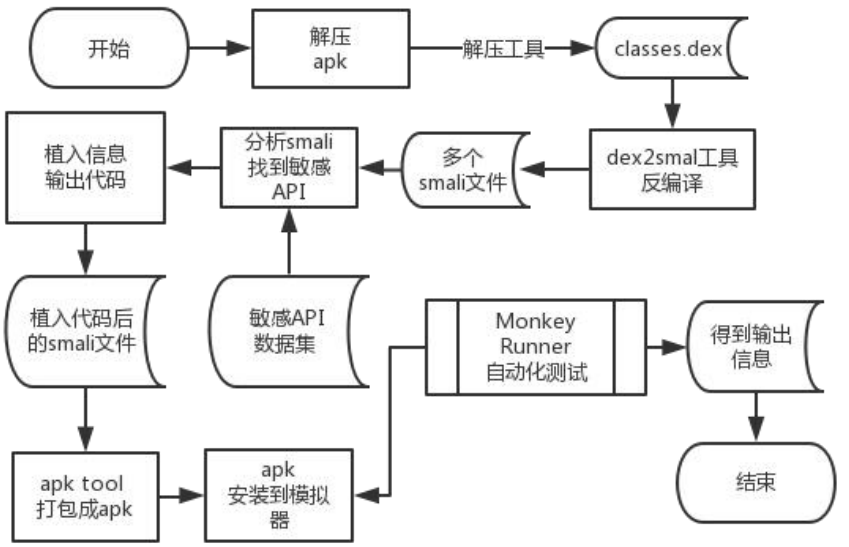


图 4-1 系统流程图

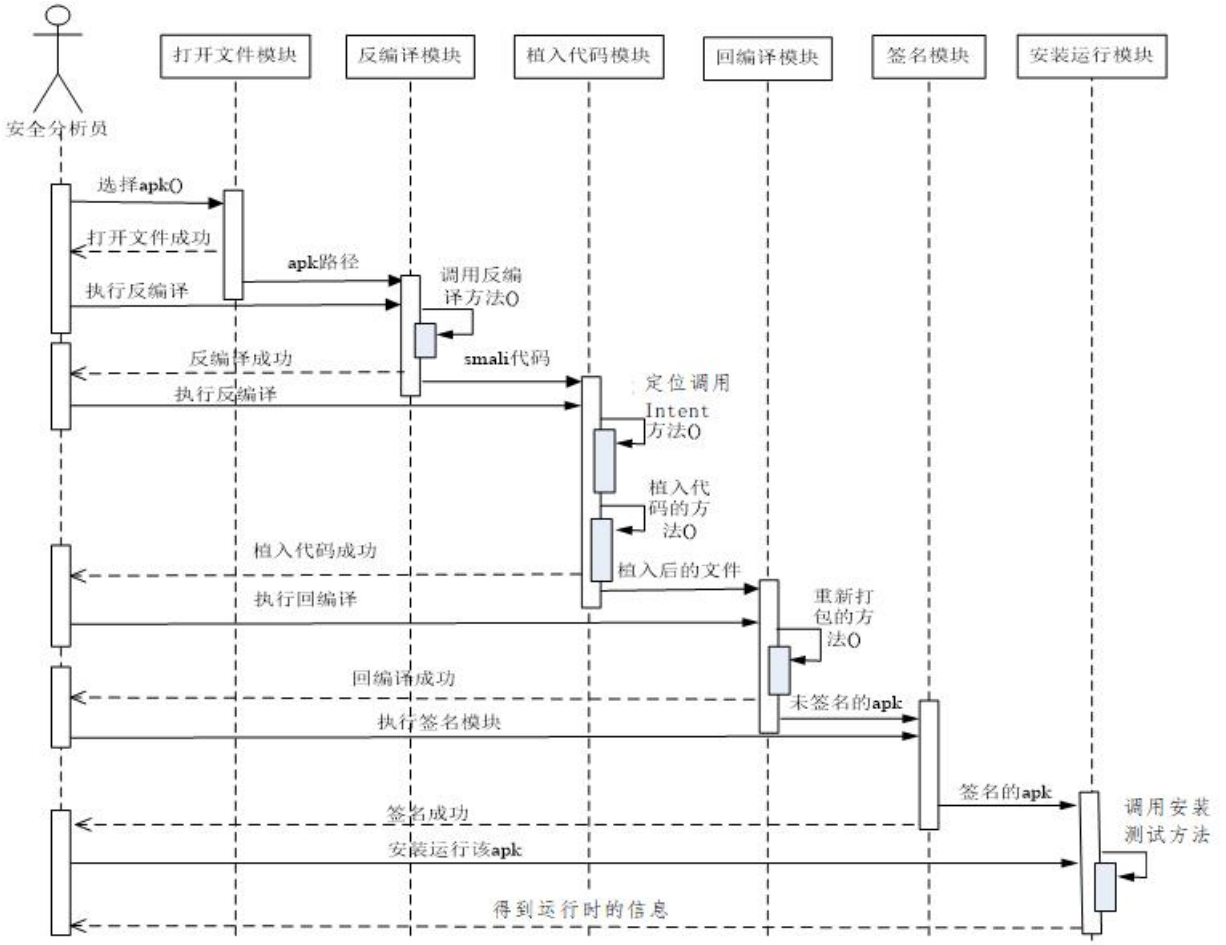


图 4-2 系统序列图

本系统界面用 Java 语言开发，系统的界面展示如图 4-3 所示，系统将 smali 代码的插桩过程以及其他操作封装成了 Python 模块，用户只需要按步骤执行即可实现查看 apk 中运行过程中用的信息。

图 4-3 系统界面

本工具使用说明：本工具是基于 Java swing 设计开发的，用户在使用过程中必须严格按照顺序执行，即在本代码注入工具中从工具栏中最左边的打开文件按钮开始到最右边的安装运行测试按钮结束，执行过程中必须等上一步执行结束才能开始下一步，具体过程为：

① 点击打开按钮选择感兴趣的 apk
② 点击反编译按钮将该 apk 中的 dex 文件反编译为 smali 文件
③ 点击植入代码按钮在 smali 文件中插入监听的日志代码
④ 点击回编译按钮将植入后的文件打包成 apk
⑤ 点击签名按钮对通过回编译得到的 apk 进行签名
⑥ 点击启动模拟器，可以启动 Android 模拟器
⑦ 点击安装按钮将该 apk 安装到模拟器中

⑧ 点击启动测试开始运行并测试 APP

⑨ 点击运行时信息可以打开查看 APP 运行过程中的调用信息

4.2 反编译模块

ApkTool 可以将 APK 文件反编译成 smali 机器码，植入代码时需要 smali 文件，所以这一步操作必不可少。

表 4-1 反编译模块 python 代码

```
1.  import sys
2.  import os
3.  apkfile_path = sys.argv[1]
4.  def decompile(apkfile_path):
5.          apktool_path = os.getcwd()+r"\tools\apktool_2.3.2.jar"
6.          os.system("java -jar "+apktool_path+" "+" d "+apkfile_path)
7.          print u"反编译完成，输出文件夹为："+apkfile_path.split(".")[0]
8.  decompile(apkfile_path)
```

执行完批处理程序后会在指定的目录下生成一系列文件和文件夹，如图 4-5 所示。

图 4-5 反编译输出文件

4.3 Log 输出代码植入

由于 Android 应用大部分都是使用 Java 语言编写，使用 Android 的 SDK 进行开发，

在 Java 源代码中使用的函数名在反编译成 smali 后函数名并不会改变（如表 4-2），可以据此找到要植入的位置。如 Activity 调用中使用的函数是 startActivity() 和 startActivityForResult()这两个函数，那么只要在 smali 中找到这两个函数调用的位置，在函数调用后进行植入就可以了。

表 4-2 Java 代码与 smali 代码对比

Java 源代码	Intent myintent = new Intent(MainActivity.this, StartActivity.class); **startActivity**(myintent);
反编译后的 smali 代码	.line 79 new-instance v2, Landroid/content/Intent; const-class v4, Lcn/hubaoquan/graduationproject/StartActivity; invoke-direct {v2, p0, v4}, Landroid/content/Intent;-><init>(Landroid/content/Context;Ljava/lang/Class;)V .line 80 .local v2, "myintent":Landroid/content/Intent; invoke-virtual {p0, v2}, Lcn/hubaoquan/graduationproject/MainActivity;->**startActivity**(Landroid/content/Intent;)V

4.3.1 显示 Intent 调用信息输出代码植入

已经知道植入位置在 startActivity()和 startActivityForResult()之后，现在关键要找到植入信息，也就是从 A 调用了 B 这两者的信息。对于显示 Intent 是从一个 Activity 调用了另一个 Activity 界面，对于隐式 Intent 是从一个 Activity 调用另一个 Action 操作。显示调用需要输出两个 Activity 的信息，隐式调用需要输出 Activity 和 Action 的信息。

通过观察大量反编译后的 smali 代码会发现，在构造一个显示 Intent 时 smali 代码步骤几乎一样。

例如表 4-3 代码，是构造 Intent 并调用 startActivity()，从 MainActivity 跳转到 StartActivity 的 smali 代码：

表 4-3 构造 Intent 并调用 startActivity 方法的 smali 代码

```
1.  new-instance v2, Landroid/content/Intent;
2.  const-class v4, Lcn/hubaoquan/graduationproject/StartActivity;
3.  invoke-direct {v2, p0, v4},
    Landroid/content/Intent;-><init>(Landroid/content/Context;Ljava/lang/Class;)V
4.  .line 80
5.  .local v2, "myintent":Landroid/content/Intent;
6.  invoke-virtual {p0, v2},
    Lcn/hubaoquan/graduationproject/MainActivity;->startActivity(Landroid/content/Intent;)V
```

忽略包名，观察第 2 行的 const-class 型变量 StartActivity，是被调用的 Activity 名称，第六行的 MainActivity 是当前运行的 Activity，这就是需要要输出的两个 Activity 的信息。

（1）首先需要找到 startActivity()，startActivityForResult()函数的位置，因为只要调用了这两个函数说明有 Intent 的调用。找出 startActivity()，startActivityForResult()函数的位置使用递归遍历每个 smali 文件，算法流程如图 4-6：

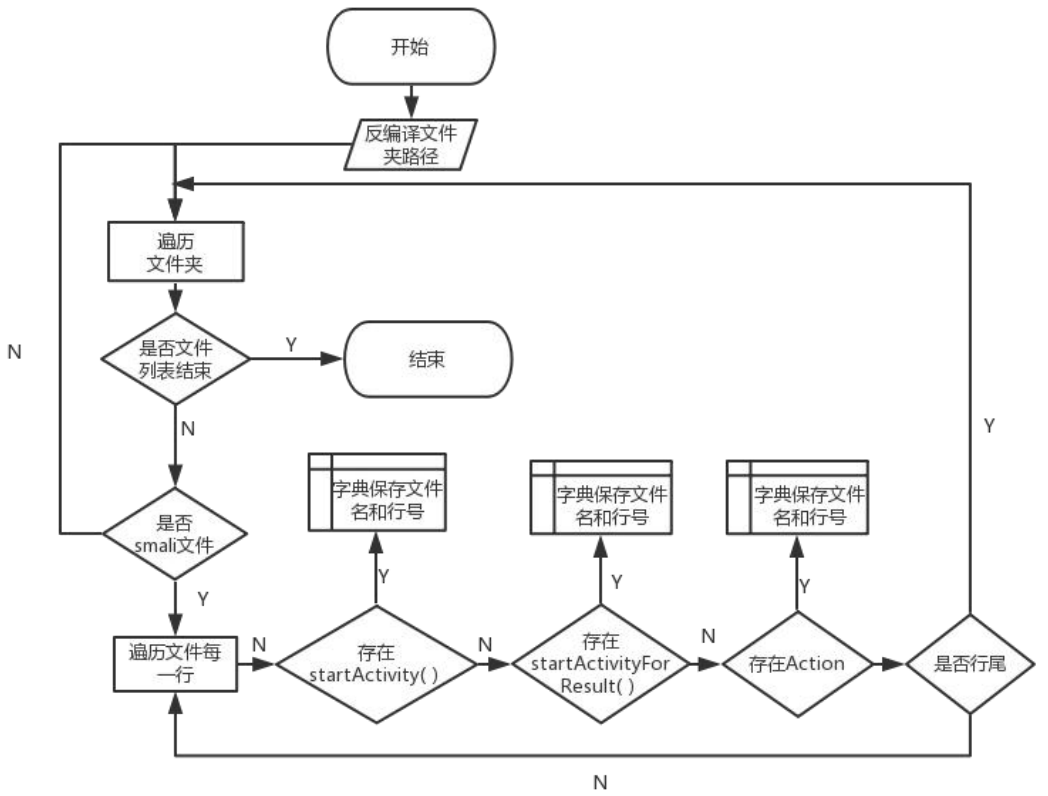


图 4-6 查找植入文件和行号算法流程图

表 4-2 查找植入文件和行号 python 函数

```
1.  def search(path):
2.      for filename in os.listdir(path):
3.          fp = os.path.join(path, filename)
4.          if os.path.isfile(fp) and aim_dir in fp:
5.              line_num = 1
6.              with open(fp) as f:
7.                  for line in f:
8.                      if "startActivity(" in line:
9.                          list_startActivity[line_num] = fp
10.                     if "startActivityForResult(" in line:
11.                         global list_startActivityForResult
```

续表 4-2 查找植入文件和行号 python 函数

```
12.                                 list_startActivityForResult[line_num] = fp
13.                             if "android.intent.action." in line:
14.                                 global list_yinshi_intent
15.                                 list_yinshi_intent[line_num] =    fp
16.                             line_num = line_num + 1
17.         elif os.path.isdir(fp):
            search(fp)
```

（2）找到要植入的文件和行号后需要找出 Activity A 调用 Activity B 的信息和 Activity 调用 Action 信息。算法流程如图 4-7;

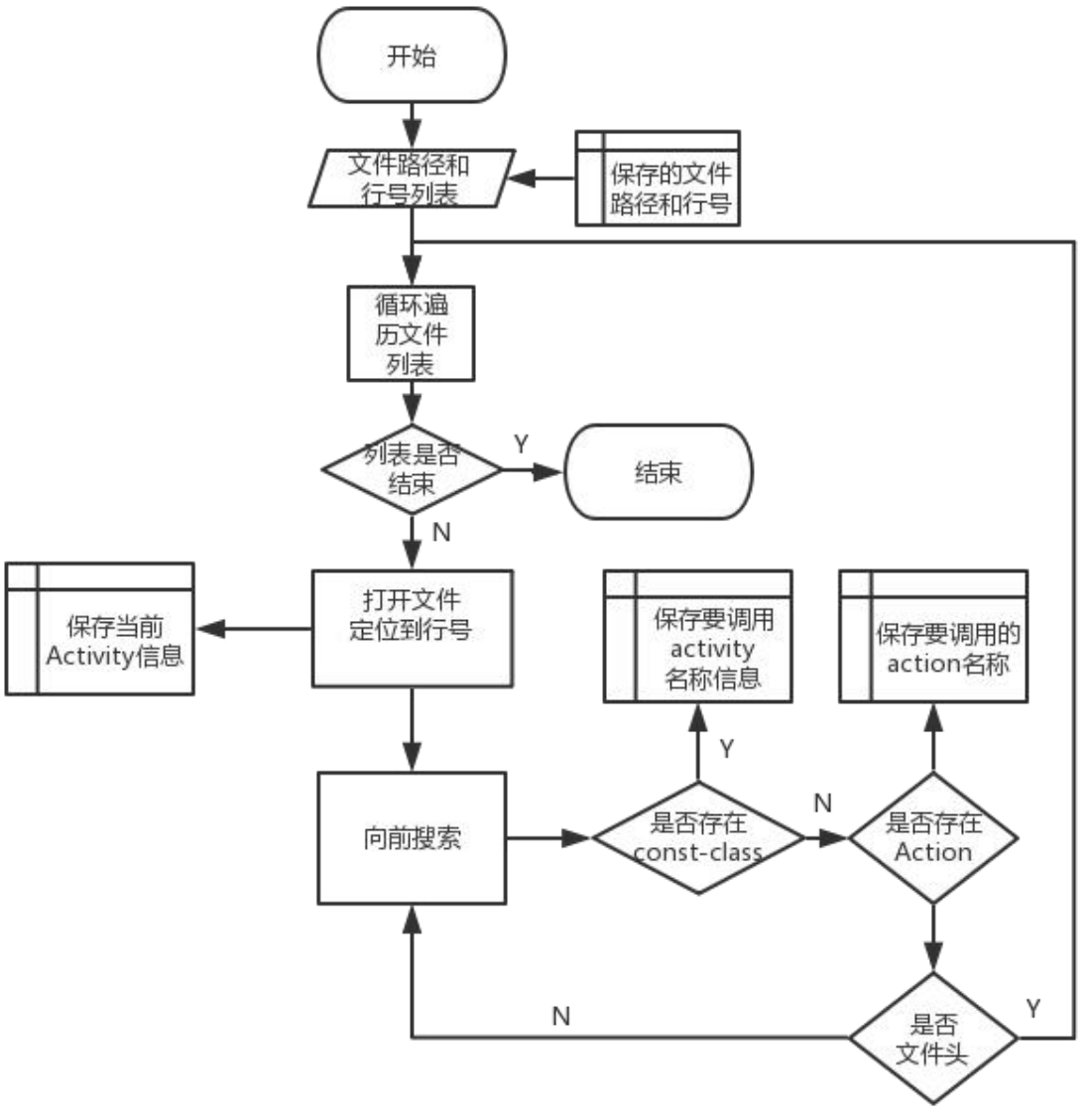


图 4-7 查找 Activity 和 Action 调用信息

表 4-3 查找 Activity 和 Action 调用信息 python 代码

```
def inject_startactivity():
    for smalifile in list_startActivity:
        with open(list_startActivity[smalifile]) as f:
            lines = f.readlines()
            const_class_line = smalifile - 1
            start_fun = smalifile - 1
            while (True):
                const_class_line = const_class_line - 1
                if "const-class" in lines[const_class_line]:
                    try:
                        print lines[start_fun].split(", L")[1].split(";->")[0].replace("/", ".") + "---->" + \
                            lines[const_class_line].split("L")[1].replace("\n", "").replace("/", ".").replace(";","") + "\n"
                        s = lines[start_fun].split(", L")[1].split(";->")[0].replace("/", ".").replace("\n","")
                        t = "----->" + lines[const_class_line].split(", L")[1].replace("\n", "").replace("/", ".").replace(";", "")
                        code = '\n    const-string/jumbo v1, "' + s + '"\n\n    const-string/jumbo v2, "' + t + '"\n\n    invoke-static {v1, v2}, Landroid/util/Log;->e(Ljava/lang/String;Ljava/lang/String;)I\n'
                        # print code
                        edit(list_startActivity[smalifile], smalifile, code)
                    except:
                        continue
                if "new-instance" in lines[const_class_line]:
                    break
                if const_class_line<=5:
                  break
```

（2）植入信息的生成。找出 Activity 和 Action 后需要生成植入信息，先看一下在 Android 源码中打印一条日志信息与反编译后的 Log 打印信息的代码对比，如表 4-4 Log 日志代码对比。

表 4-4 Log 打印日志代码对比

Log 打印 Java 源码	Log.e("MainActivity", "Log Error");
Smali 代码	const-string/jumbo v1, "MainActivity" const-string/jumbo v2, "Log Error" invoke-static {v1, v2}, Landroid/util/Log;->e(Ljava/lang/String;Ljava/lang/String;)I

可以据此生成植入信息。例如，我们找到的 Activity 信息是 MainActivity 调用了 StartActivity。则可以植入的输出信息如表 4-5 ：

表 4-5 植入信息

```
const-string/jumbo v1, "MainActivity"

const-string/jumbo v2, "----->StartActivity"

invoke-static {v1, v2}, Landroid/util/Log;->e(Ljava/lang/String;Ljava/lang/String;)I
```

以上 Log 预计输出结果为： MainActivity ----->StartActivity。

这样就可以直接生成代码植入模板了，生成需要植入的 code 之后就可以进行植入操作了。植入模板如表 4-6，其中 s 变量是当前的 Activity，t 变量是被调用的 Activity。

表 4-6 植入代码模板

```
code
= '       const-string/jumbo v1, "'
+ s
+ '"\n\n       const-string/jumbo v2, "'
+ t
  + '"\n\n       invoke-static {v1, v2}, Landroid/util/Log;->e(Ljava/lang/String;Ljava/lang/String;)I\n'
```

植入代码生成后就可以调用 edit()函数进行植入操作了。 植入函数接收需要植入的文件、植入行号、植入代码。edit()代码如表 4-7。

表 4-7 植入函数 python 代码

```
1.    def edit(smali, line_num, code):
2.        fp = file(smali)
3.        lines = []
4.        for line in fp:   # 内置的迭代器, 效率很高
5.            lines.append(line)
6.        fp.close()
7.        lines.insert(line_num, code)   # 在第 line_num 行插入
8.        s = ''.join(lines)
9.        fp = file(smali, 'w')
10.       fp.write(s)
11.       fp.close()
12.       print "Line:" + str(line_num) + u"  植入成功！"
```

植入完成后的代码对比，如表 4-7

表 4-7 植入前后代码对比

植入前	new-instance v1, LAndroid/Content/Intent; const-class v4, Lcn/hubaoquan/graduationproject/StartActivity; invoke-direct {v2, p0, v4}, Landroid/content/Intent;-><init>(Landroid/content/Context;Ljava/lang/Class;)V .line 80 .local v2, "myintent":Landroid/content/Intent;

续表 **4-7** 植入前后代码对比

	Lcn/hubaoquan/graduationproject/MainActivity;->startActivity(Landroid/content/Intent;)V
植入后	new-Instance v2, Landroid/content/Intent; const-class v4, Lcn/hubaoquan/graduationproject/StartActivity; invoke-direct {v2, p0, v4}, Landroid/content/Intent;-><init>(Landroid/content/Context;Ljava/lang/Class;)V .line 80 .local v2, "myintent":Landroid/content/Intent; invoke-virtual {p0, v2}, Lcn/hubaoquan/graduationproject/MainActivity;->startActivity(Landroid/content/Intent;)V const-string/jumbo v1, "cn.hubaoquan.graduationproject.MainActivity" const-string/jumbo v2, "----->cn.hubaoquan.graduationproject.StartActivity" invoke-static {v1, v2}, Landroid/util/Log;->e(Ljava/lang/String;Ljava/lang/String;)I

4.3.2 隐式 Intent 信息输出代码植入

对于隐式的 Intent 是由 Activity 调用了一个 Action，需要输出的信息是 Activity 和 Action 信息。方法和显示 Intent 差别不大，就是在查找 Action 时与查找 Activity 不同。其他流程基本相似。

4.3.3 服务和广播的植入

广播和服务的植入思路和 Activity 的一样，找到调用广播和服务的函数，然后在函数之后进行信息植入。发送广播的函数有：sendBroadcast()、sendOrderedBroadcast()、sendStickyBroadcast()，启动服务的函数有：startService()、bindService()，在 smali 文件中找到这些函数的位置，操作同 Activity 植入。

4.4 回编译 APK

植入完成后需要将所有文件打包回 APK 文件，这需要用到 ApkTool 工具，执行打包命令：

```
>>java  -jar  apktool.jar  b  [需要打包的文件夹]
```

执行完成后会在文件夹中生成一个 dist 文件夹，重打包的 apk 存放在该文件夹下。

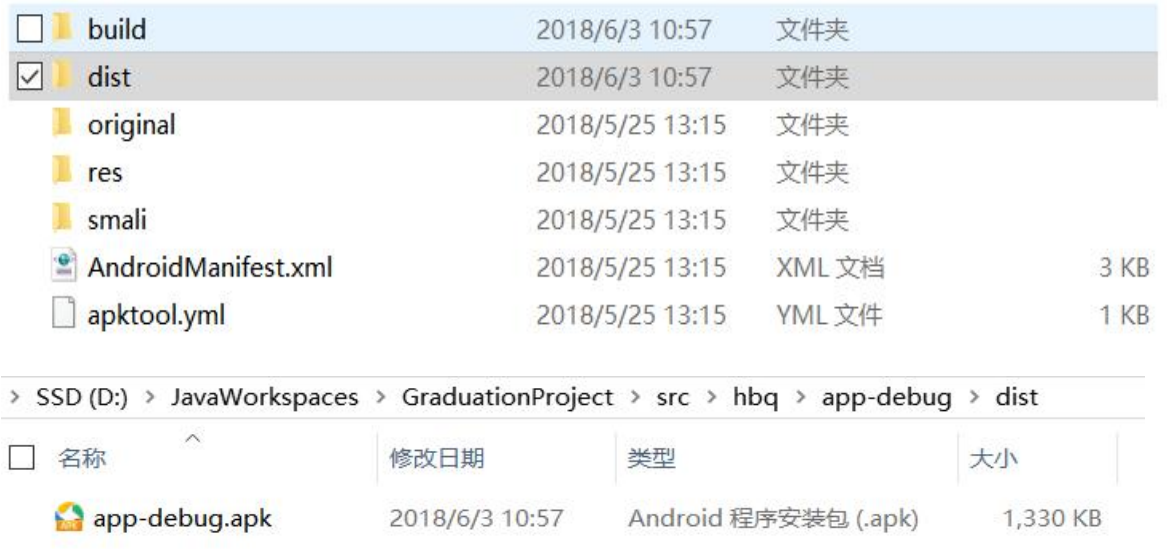


图 4-8 重打包生成的文件和文件夹

4.5 APK 签名与安装

首先使用 keytool 密钥和证书管理工具创建签名密钥，生成命令如下：

```
>>keytool -genkey -v -keystore hbq.keystore -alias hbq.keystore -keyalg RSA -validity 361
```

以上命令执行后会生成一个名为 hbq.keystore 的密钥有效期为 361 天。生成密钥后就可以使用该密钥对 apk 进行签名了。使用以下命令进行签名：

```
>>jarsigner -verbose -keystore hbq.keystore -signedjar [签名 apk 文件名]  [签名前的 apk 名]  hbq.keystore
```

生成后的签名文件就可以进行安装了。启动模拟器后使用 adb 命令进行 apk 安装：

```
>>adb install -t [签名后的 apk 名称]
```

安装完成后接下来就可以进行运行和测试了。

4.6 运行测试与运行时信息获取

4.6.1 MonkeyRunner 脚本录制

使用 MonkeyRunner 进行植入代码后的 App 测试脚本录制,录制完成后进行自动测试。

•首先启动 Android 模拟器，可以手动启动，用 cmd 命令启动方式如下：emulator -avd phone

图 4-9　Android 模拟器

- 启动 MonkeyRunner，运行命令：MonkeyRunner

```
>>MonkeyRunner
Jython 2.5.3 (2.5:c56500f08d34+, Aug 13 2012, 14:54:35)
[Java HotSpot(TM) 64-Bit Server VM (Oracle Corporation)] on java1.8.0_144
>>>
```

- 依次输入执行以下指令

```
>>> from com.android.monkeyrunner import MonkeyRunner as mr
>>> from com.android.monkeyrunner.recorder import MonkeyRecorder as recorder
>>> device = mr.waitForConnection()
>>> recorder.start(device)
```

运行之后会出现，图形界面录屏工具。如下图：

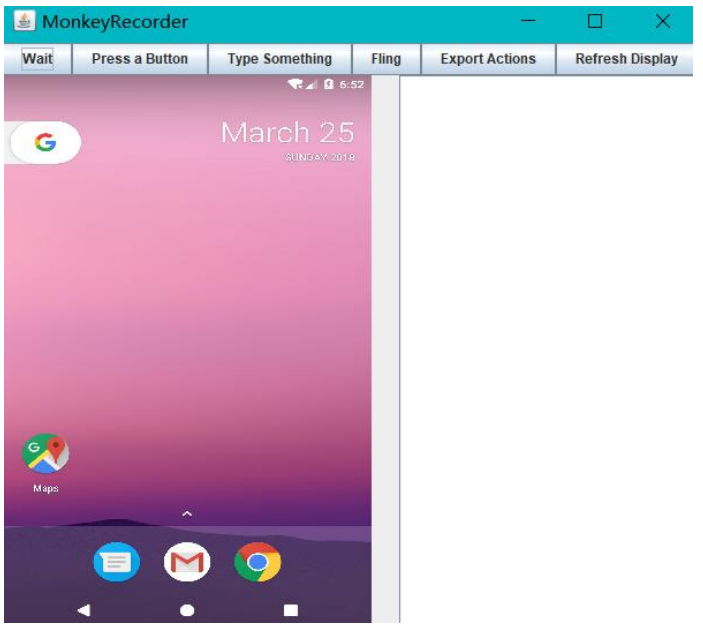


图 4-10 脚本录制工具

此时每点击该录屏工具的每一步都会被记录下来。依次点击按钮控件，录制 APP 的点击测试过程。录制过程中会自动生成的步骤脚本，如图 4-11：

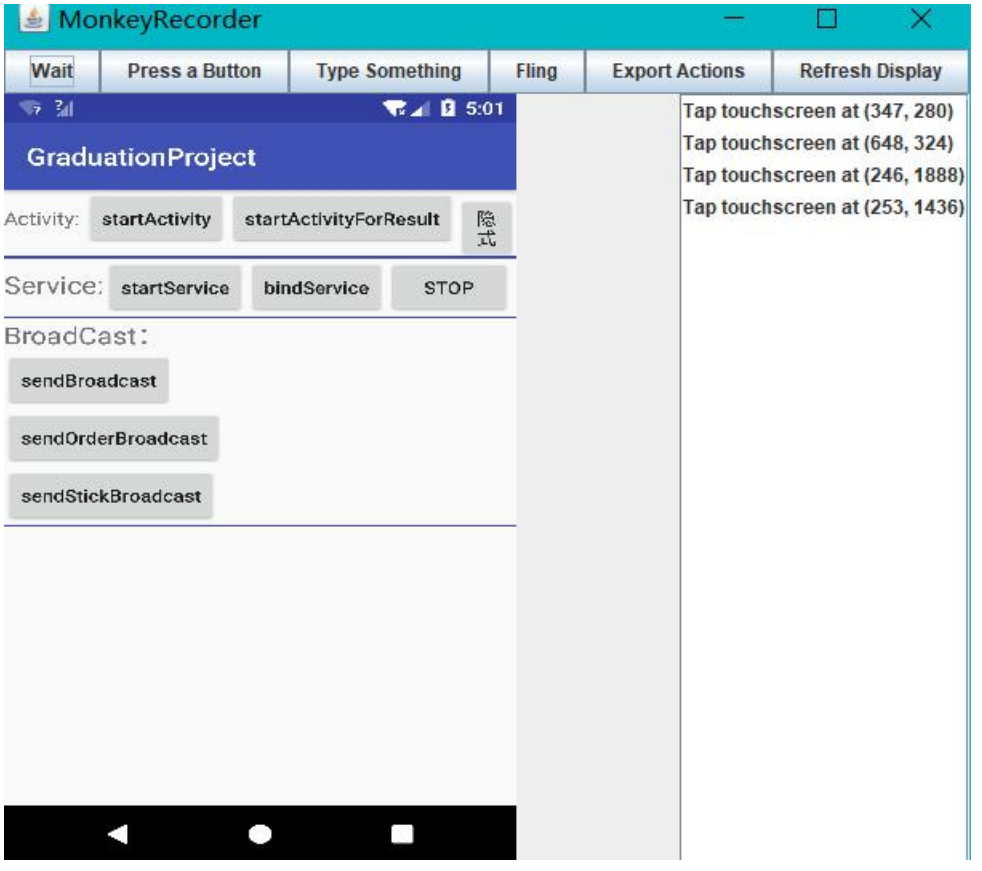


图 4-11 录制脚本

• 保存脚本，点击工具的 ExportAction 可以导出脚本，导出的脚本内容

表 4-8 自动测试脚本

```
WAIT|{'seconds':3.5,}
TOUCH|{'x':307,'y':320,'type':'downAndUp',}
WAIT|{'seconds':2.0,}
TOUCH|{'x':229,'y':1848,'type':'downAndUp',}
WAIT|{'seconds':2.0,}
TOUCH|{'x':594,'y':312,'type':'downAndUp',}
WAIT|{'seconds':2.0,}
TOUCH|{'x':270,'y':1844,'type':'downAndUp',}
WAIT|{'seconds':2.0,}
TOUCH|{'x':999,'y':340,'type':'downAndUp',}
WAIT|{'seconds':2.0,}
TOUCH|{'x':249,'y':1848,'type':'downAndUp',}
WAIT|{'seconds':2.0,}
TOUCH|{'x':226,'y':1836,'type':'downAndUp',}
TOUCH|{'x':266,'y':1864,'type':'downAndUp',}
WAIT|{'seconds':2.0,}
TOUCH|{'x':330,'y':456,'type':'downAndUp',}
WAIT|{'seconds':3.5,}
TOUCH|{'x':580,'y':464,'type':'downAndUp',}
WAIT|{'seconds':3.5,}
TOUCH|{'x':901,'y':500,'type':'downAndUp',}
WAIT|{'seconds':3.5,}
TOUCH|{'x':182,'y':716,'type':'downAndUp',}
WAIT|{'seconds':3.5,}
TOUCH|{'x':249,'y':896,'type':'downAndUp',}
WAIT|{'seconds':3.5,}
TOUCH|{'x':337,'y':1020,'type':'downAndUp',}
```

4.6.2 运行脚本

运行脚本,编辑以下 Python 代码保存为 player_back.py

表 4-9 测试脚本播放代码

```
import sys
from com.android.monkeyrunner import MonkeyRunner
CMD_MAP = {
    'TOUCH': lambda dev, arg: dev.touch(**arg),
    'DRAG': lambda dev, arg: dev.drag(**arg),
    'PRESS': lambda dev, arg: dev.press(**arg),
    'TYPE': lambda dev, arg: dev.type(**arg),
 'WAIT': lambda dev, arg: MonkeyRunner.sleep(**arg)
    }
# Process a single file for the specified device.
def process_file(fp, device):
    for line in fp:
        (cmd, rest) = line.split('|')
        try:
            # Parse the pydict
            rest = eval(rest)
        except:
            print 'unable to parse options'
            continue
        if cmd not in CMD_MAP:
            print 'unknown command: ' + cmd
            continue
        CMD_MAP[cmd](device, rest)
def main():
    file = sys.argv[1]
    fp = open(file, 'r')
    device = MonkeyRunner.waitForConnection()
    process_file(fp, device)
    fp.close();
if __name__ == '__main__':
        main()
```

将录制的脚本文件 script.txt 放在目录下。

手动运行以下命令：

```
>>MonkeyRunner D:\GraduationProject\Works\monkeyrunner\player_back.py
D:\GraduationProject\Works\monkeyrunner\script.txt
```

运行之后就会看见模拟器自动打开 App 进行测试。工具已经将上述过程自动化，不需要手动执行命令，只需用工具选择要运行的脚本即可。输出的测试结果如图 4-12 所示，测试结果保存在项目根目录下的 IntentInfo.txt 文件中，如表 4-10 所示。

图 4-12 测试结果

表 4-10 运行时信息获取结果

06-07 05:52:10.609 3865 3865 E cn.hubaoquan.graduationproject.MainActivity: ----->cn.hubaoquan.graduationproject.StartActivity
06-07 05:52:14.617 3865 3865 E cn.hubaoquan.graduationproject.MainActivity: ----->cn.hubaoquan.graduationproject.StartForResultActivity
06-07 05:52:18.644 3865 3865 E cn.hubaoquan.graduationproject.MainActivity: ----->android.intent.action.DIAL
06-07 05:52:24.678 3865 3865 E cn.hubaoquan.graduationproject.MainActivity: ----->cn.hubaoquan.graduationproject.MyServiceStart
06-07 05:52:28.192 3865 3865 E cn.hubaoquan.graduationproject.MainActivity: ----->cn.hubaoquan.graduationproject.MyServiceBind
06-07 05:52:35.206 3865 3865 E cn.hubaoquan.graduationproject.MainActivity:----->cn.hubaoquan.graduationproject.MYBROADCAST RECEIVER: cn.hubaoquan.graduationproject.MyReceiverSend
06-07 05:52:38.723 3865 3865 E

续表 4-10 运行时信息获取结果

cn.hubaoquan.graduationproject.MainActivity:----->cn.hubaoquan.graduationproject.ORDERCAST RECEIVER: cn.hubaoquan.graduationproject.MyReceiverSendOrder 06-07 05:52:42.245 3865 3865 E cn.hubaoquan.graduationproject.MainActivity:----->cn.hubaoquan.graduationproject.STICKYCAST RECEIVER: cn.hubaoquan.graduationproject.MyReceiverSendSticky

5 集成工具软件操作流程

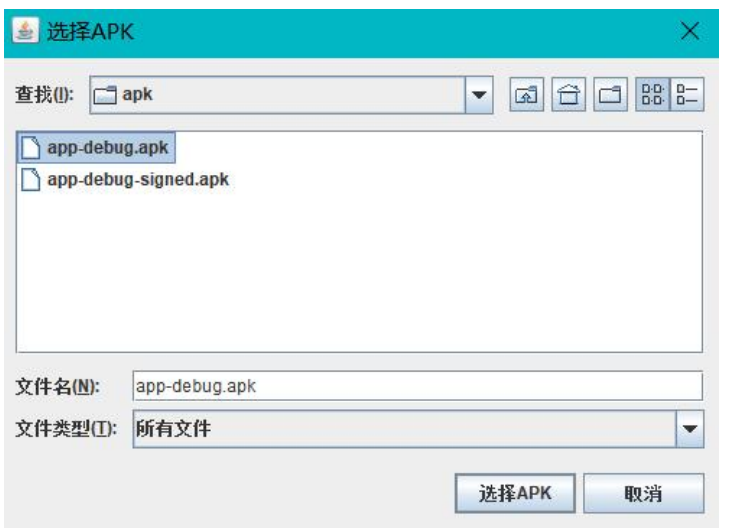
5.1 选择 APK 文件进行反编译

5.1.1 选择文件

运行软件后如图 5-1，

图 5-1 软件主界面

点击打开 APK 按钮后进入 apk 选择界面，找到要分析的 apk 点击选择 apk 就可以了，如图 5-2。



5-2 选择 APK

5.1.2 反编译

选择完 APK 包后，点击反编译 APK 按钮，自动进行反编译操作，如图 5-3。

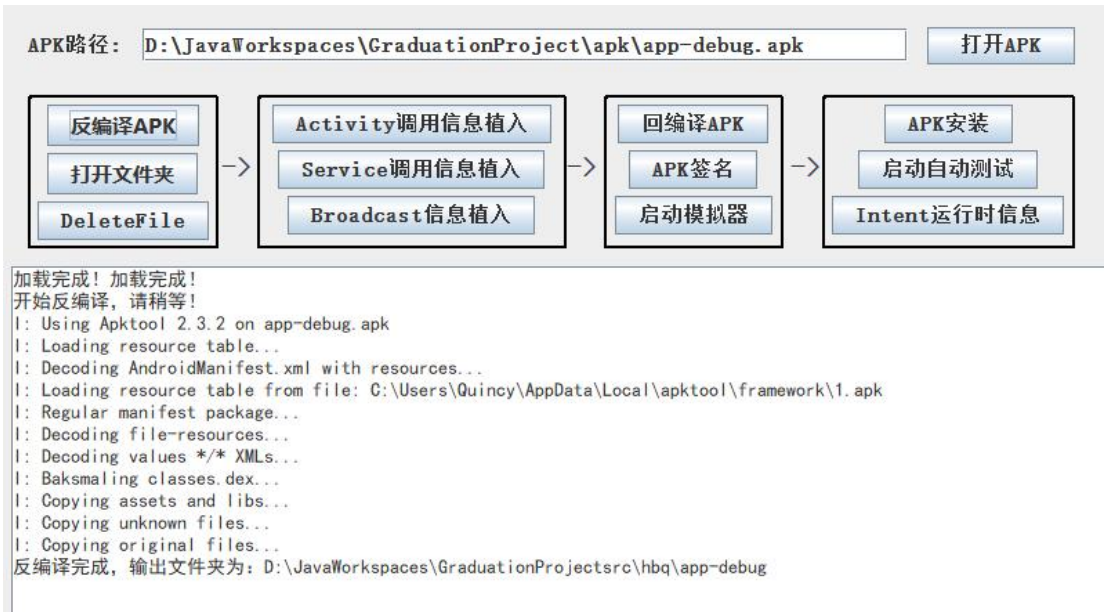


图 5-3 反编译操作

反编译完成后可以点击文件保存位置按钮，进入反编译文件夹存放位置。删除工作文件按钮会删除反编译输出的文件夹，除非需要开始一个新的 apk 反编译，否则不要删除文件夹。

5.2 植入代码操作

5.2.1 Activity 调用信息植入

点击 Activity 调用信息植入按钮自动植入 Activity 调用的显示和隐式信息，同时在文本区域显示存在 Intent 调用的 smali 文件及调用行号，如图 5-4。

图 5-4 Activity 植入

5.2.2 Service 调用植入

点击 Service 调用信息植入，植入可以查看 apk 中使用了那些服务的 log，如图 5-5

```
开始植入Service调用信息!
□[1;31;m

存在startService()函数调用的文件及调用行号:
298 D:\JavaWorkspaces\GraduationProject\\src\hbq\app-debug\smali\cn\hubaoquan\graduationproject\MainActivity.smali

Service调用情况:
cn.hubaoquan.graduationproject.MainActivity---->cn.hubaoquan.graduationproject.MyServiceStart

Line:298 植入成功!
********************************************************************************************


存在bindService()函数调用的文件及调用行号:
328 D:\JavaWorkspaces\GraduationProject\\src\hbq\app-debug\smali\cn\hubaoquan\graduationproject\MainActivity.smali

Service调用情况:
cn.hubaoquan.graduationproject.MainActivity----->cn.hubaoquan.graduationproject.MyServiceBind
Line:328 植入成功!
********************************************************************************************
```

图 5-5 Service 植入

5.2.3 Broadcast 调用植入

点击 Broadcast 信息植入，完成 Broadcast 调用信息植入如图 5-6。

图 5-6　Broadcast 植入

5.3 回编译签名 APK 和 Android 模拟器的启动

5.3.1 回编译 APK

植入代码操作完成后就可以进行回编译操作了，点击回编译 APK 进行回编译，如图 5-7。

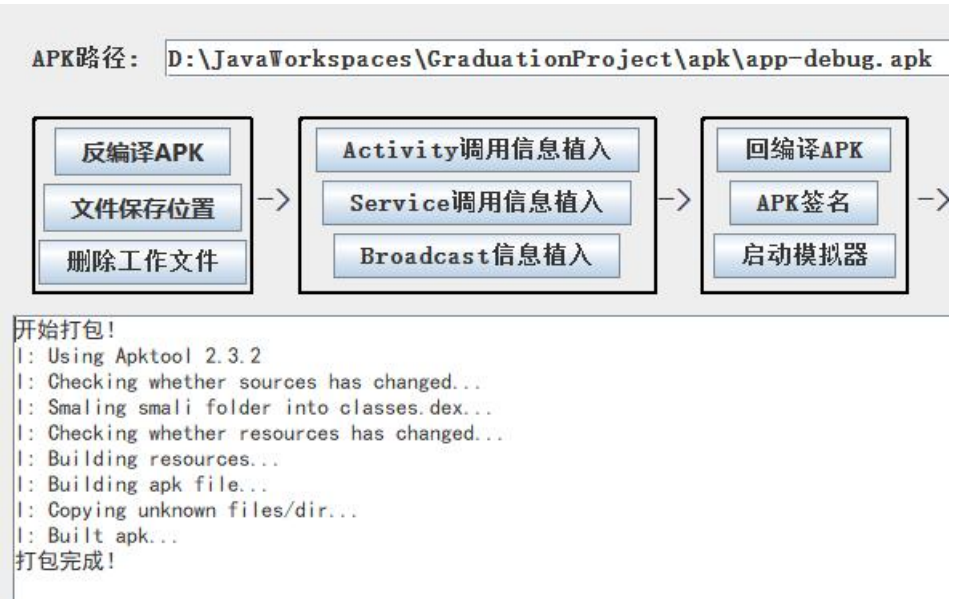


图 5-7 回编译操作

回编译完成后会在反编译文件夹生成 dist 文件夹，重打包的 apk 文件保存在里边。如图 5-8 所示。

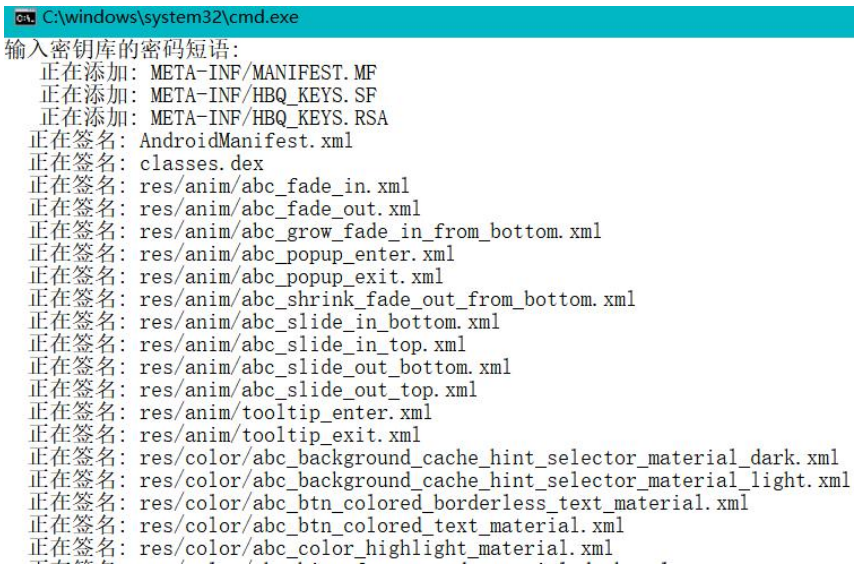
图 5-8 生成的 apk 文件

5.3.2 apk 签名

点击 apk 签名按钮进入 apk 签名界面，如图 5-9。

图 5-9　签名界面

输入密钥后开始签名，如图 5-10。



5-10 签名过程

签名成功后在项目根目录下生成签名的 APK 文件。

5.3.3　创建并启动模拟器

本文测试环境使用 Android Studio SDK 自带的 Android 模拟器，首先在 Android Studio 中创建一个模拟器，然后启动它，就可以使用了。启动模拟器的按钮执行以下命令：emulator　@phone 启动名为 phone 的模拟器，如果没有将 emulator 加入环境变量，或者创建的模拟器名不为 phone 该按钮都将无效。无论如何只要启动了一个模拟器端口是 5554，都可以进行接下来的测试工作。

5.3.4　APK 安装

签名后的安装包可以直接安装到模拟器，点击工具 apk 安装按钮就可以自动安装。如果自动安装没有成功，可以使用 Android 的 adb 命令手动安装签名后的测试 apk 文件到模拟器，手动安装命令：adb install -t [apk 文件名]。

5.4 运行测试与信息获取

Apk 安装完成后可以点击启动自动测试按钮进行自动测试，自动测试需要选择录制好的脚本如图 5-11。如果没有录制脚本则可以选择随机测试如图 5-12。

图 5-11 选择测试脚本

测试运行结果如图 5-12 所示

图 5-12 运行测试结果

运行测试的结果保存在项目工作根目录下的 IntentInfo.txt 文件中，以便于之后分析利用，测试结果如表 5-1 所示。

表 5-1 运行时信息调用结果

06-07 05:52:10.609 3865 3865 E
cn.hubaoquan.graduationproject.MainActivity: ----->cn.hubaoquan.graduationproject.StartActivity
06-07 05:52:14.617 3865 3865 E
cn.hubaoquan.graduationproject.MainActivity: ----->cn.hubaoquan.graduationproject.StartForResultActivity
06-07 05:52:18.644 3865 3865 E
cn.hubaoquan.graduationproject.MainActivity: ----->android.intent.action.DIAL
06-07 05:52:24.678 3865 3865 E
cn.hubaoquan.graduationproject.MainActivity: ----->cn.hubaoquan.graduationproject.MyServiceStart
06-07 05:52:28.192 3865 3865 E
cn.hubaoquan.graduationproject.MainActivity: ----->cn.hubaoquan.graduationproject.MyServiceBind
06-07 05:52:35.206 3865 3865 E
cn.hubaoquan.graduationproject.MainActivity:----->cn.hubaoquan.graduationproject.MYBROADCAST
RECEIVER: cn.hubaoquan.graduationproject.MyReceiverSend
06-07 05:52:38.723 3865 3865 E
cn.hubaoquan.graduationproject.MainActivity:----->cn.hubaoquan.graduationproject.ORDERCAST
RECEIVER: cn.hubaoquan.graduationproject.MyReceiverSendOrder
06-07 05:52:42.245 3865 3865 E
cn.hubaoquan.graduationproject.MainActivity:----->cn.hubaoquan.graduationproject.STICKYCAST
RECEIVER: cn.hubaoquan.graduationproject.MyReceiverSendSticky

6 总结与展望

6.1 总结

本文首先介绍了 Android 平台的发展状况以及在发展过程中面临的安全威胁，并对当前国际的研究现状进行了调研，由于恶意 APK 实现某些功能时往往调用了敏感 API，由此引出本课题的研究意义。在进行本课题开始时首先对 Android 系统的架构和 apk 包的结构进行了详细的介绍，分析了 Android 系统的安全机制和安全机制存在的不足之处。之后介绍了本次毕业设计中所用到的技术，包括 apktool、smali 语法、apk 签名机制、monkey 随机测试、adb 调试桥等等。然后根据需求分析设计完成了该基于 monkeyrunner 自动测试的应用程序行为提取系统，并对各个模块分别进行了详细的介绍，随后用实例进行了测试，统计了执行结果，验证本系统的有效性。

6.2 展望

本次的设计只是对恶意 APK 的初步研究与探索，要深入解决 Android 系统面临的安全问题还需要很大的改进和完善，并且随着技术的发展恶意的 APK 也会层出不穷，未来的 Android 市场需要不断的维护和改善，这注定是一个长期而艰巨的任务。为了保护广大的 Android 用户不被黑客攻击，所有的 Android 安全人员需要不断丰富和提高自己的知识量和技术水平，不断创造出新的更有效的措施来预防和解决 Android 系统的安全问题，本系统将来也会加入更多有效的方案进一步完善自身的不足。作者将会在研究生期间进一步学习和研究该课题，相信会提出更多切实可行的方案来预防和解决 Android 系统中的恶意 APK。

致　谢

美好的大学生涯即将以这最后几个月的毕业设计而宣告结束，这是大学最后的作业。这四年的大学，因为学校，因为老师，我可以容易的学到教程要求的知识，因为学校给予我一个广阔自由的平台，老师的授课欢快又不失严肃，这种环境使我可以在学校安心的充实自己，丰富自己。同学校友间的融洽相处，互帮互助让我心里倍感到温暖。一个离家求学的人仿佛又找到了自己的新家如此的温暖。学会了如何去关心他人，帮助他人，正确的为人处事的方式积极乐观的生活态度。我还学会了学习一切重在自己的学习方法，这会让我在以后的工作中受益匪浅。

在这里感谢我的指导老师刘晓建老师一步一步耐心的指导和讲解，感谢我的同学伊扬贵在完成本次课题过程中给予我的帮助。同时，也感谢陪伴我大学四年的同学们，尤其是室友，他们给我大学生活增添了许多乐趣。

最后，希望自己能够以这份毕设完美的结束自己的学生生涯，再用激情去迎接更加充满希望而美好的明天。

参考文献

[1] 《Android 和 iOS 彻底瓜分完全球移动操作系统市场》，
http://tech.ifeng.com/a/20180223/44884886_0.shtml

[2] 互联网协会，国家互联网应急中心，《中国移动互联网发展状况及其安全报告》，2016 年 5 月

[3] 360 手机卫士，《2016 年安卓恶意软件专题报告》，2017 年 2 月

[4] 国家计算机网络应急技术处理协调中心，《移动互联网恶意代码描述规范》，YD/T 2439-2012，行业标准.

[5] 李子锋,程绍银,蒋凡.一种 Android 应用程序恶意行为的静态检测方法,中国科学技术大学,信息安全测评中心.

[6] 卿斯汉,“Android 安全研究进展”,软件学报,2016,27(1)p45-71

[7] K. Griffin, S. Schneider, X. Hu and TC. Chiueh. Automatic Generation of String Signatures for Malware Detection[C]// RecentAdvances in Intrusion Detection(RAID) .pp.101-120.2009.

[8] W. Enck, P. Gilbert, BG. Chun, LP. Cox, J. Jung, P. McDaniel and AN. Sheth . TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones[C]// Usenix Symposium on Operating Systems Design and Implementation(OSDI). pp.1 - 6.2010.

[9] M. Zhang, Y. Duan, H. Yin and Z. Zhao. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs[C]// ACM, pp.1105-1116, 2014.

[10] X. Xiao, X. Xiao, Y. Jiang and R. Ye. Identifying Android malware with system call co-occurrence matrices[J].//Transactions on Emerging Telecommunications Technologies, pp.675-684,2016.

[11] 杨欢，张玉清，胡予濮，刘奇旭，“基于多类特征的 Android 应用恶意行为检测系统 ”，计算机学报，2014.

[12] Google 开发者文档
https://developer.android.com/reference/android/app/Instrumentation.html

[13] Instrumentation 简介，https://www.jianshu.com/p/b4d6e9bbcfce

[14] H. Huang, C. Zheng, J. Zeng, W. Zhou, S. Zhu, P. Liu, S. Chari and C. Zhang. Android malware development on public malware scanning platforms: A large-scale data-driven study[C]// IEEE International Conference on Big Data(Big Data). IEEE, pp.1090-1099,2016.

[15]丰生强.Android 软件安全与逆向工程[M].北京:人民邮电出版社,2013,2,156-157

[16]张中文,雷灵光,王跃武.Android Permission 机制的实现与安全分析[C].第 27 次全国计算机安全学术交流会.2012.8:3-5

[17]左玲,基于 Android 恶意软件检测系统的设计与实现.[D].成都:电子科技大学,2012,13-18