基于Android内存镜像的恶意软件检测研究



重庆大学硕士学位论文

学生姓名：

指导教师：

专　　业：计算机软件与理论

学科门类：工学

　重庆大学计算机学院

二〇一六年四月

**Study on Properties and Utilization in**

**Building Materials of bottom ashes from**

**Circulating Fluidized Bed Combustion**

****

**A Thesis Submitted to Chongqing University   
in Partial Fulfillment of the Requirement for the**

**Degree of Master of Engineering  
by**

**Supervisor:**

**Major: Computer Software and Theory**

**College of Computer Science of**

**Chongqing University, Chongqing, China.**

**April,2016**

摘 要

Android系统由于其开源特性，吸引了大批的开发者投入其中。然后正是由于其开发特定，导致了恶意软件的泛滥。

**关键词：**Android系统，内存镜像，隐藏进程，恶意应用，检测

ABSTRACT

Android

**Keywords：**Android system,

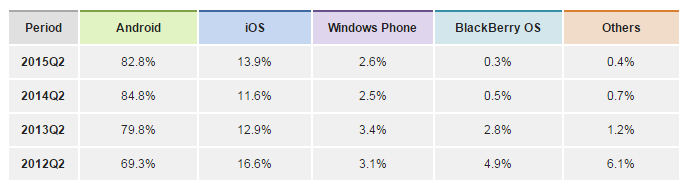
# 绪论

## 研究背景与意义

随机通信技术和电子信息技术的发展、普及和应用，各类通信电子设备层出不穷，不断涌现。移动智能设备，特别是智能手机，日益成为人们生活中不可或缺的工具，它不断改变这人们的工作、生活、学习和娱乐等各方面。智能手机已经取代传统的功能手机，成为了市场中的主流[1]，而智能手机中的操作系统是其能够快速发展的主要因素。现阶段，智能手机的操作系统主要有Android、IOS以及Windows Phone，其中Android系统占据大部分份额。表1.1展示了从2012年到2015年第二季度中各个智能手机系统的市场占有率[2]，从中可以看出Android系统已经占据了智能手机市场的绝大部分，并且增长迅速。

表1.1表智能手机操作系统市场占有率分布

Table 1.1 Table Smartphone OS Market Share



快速增长的原因是由于其开放特性。一堆好处

正是由于其开放特性使得Android用户面临更多潜在的安全威胁，越来越多的恶意软件开始将矛头指向Android系统。

恶意代码分类、感染方式、常用反逆向技术

## 国内外研究现状

恶意软件检测技术

研究成果

恶意软件发现

## 本文的主要工作

Android系统的恶意软件可以分为系统级的内核模块，以及Android应用层的恶意应用。本文针对恶意软件的检测也分为两种：内核级的主要是通过检测隐藏进程来找到对应的恶意软件；Android应用层的主要是通过对内存镜像的分析找到可疑的进程，然后从中提取Android应用的执行代码，最后通过敏感API的调用来实现恶意应用的检测。

## 本文的组织结构

# Android和Linux相关机制介绍

了解Android框架和应用结构是Android平台下恶意应用检测的基础。而Android系统是基于Linux系统的，对其有基本的了解，能够更好的完成对Android系统的内存镜像分析。本章首先对Android系统进行综述性描述，介绍了Android的体系结构、Android应用的组成、应用运行环境基础Dalvik虚拟机等。其次对一些恶意软件常用的Java机制进行了介绍，包括JNI接口和反射技术。最后对Linux相关机制进行了介绍，包括内核符号表、proc虚拟文件系统以及动态可加载内核模块LKM等。

## Android系统综述

本节对Android系统的基本概念进行概述，主要包括三个部分：首先介绍Android系统体系结构，主要内容为其分层结构；接着对Android应用进行介绍，主要内容为Android应用安装包APK的结构；然后对作为Android应用运行环境的Dalvik虚拟机进行了简单介绍；最后介绍了Android系统的安全机制。

### Android系统体系结构

Android系统是Google开发的基于Linux平台的开源手机操作系统，采用分层结构，从上到下依次为：应用层、应用框架层、系统库和Android运行时以及底层的Linux内核层，如图2.1所示，每层功能简要介绍如下。

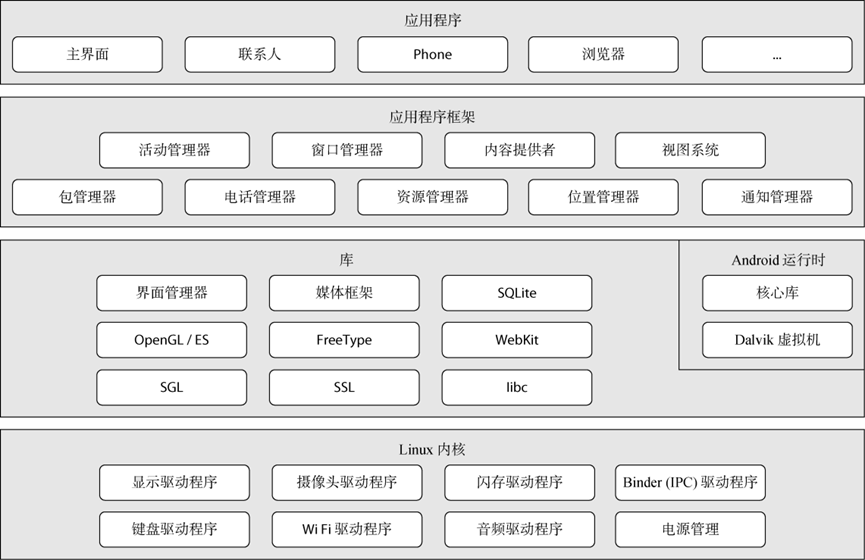


图2.1 Android系统体系结构

Fig 2.1 The System Structure of Android OS

1) 应用层：该层是用户交互的入口，除了提供一些核心应用，开发者还可以使用应用框架层提供的API开发自己的应用，各个应用彼此平等。应用层是Android系统对外提供服务的顶层，用户的所有操作都是通过应用来体现的，也是恶意软件入侵的入口。

2) 应用框架层:该层是Android应用开发的基础，开发人员在大部分情况下都是和它打交道，Android API就是在该层提供的。任何一个应用都可以发布自身的功能模块，其他应用可以直接使用这些已发布的功能模块，这种设计简化了组件的重用，用户可以很方便的替换平台本身的各种应用。

3) 系统库和Android运行时：系统库总共包括九个子系统，是一些C/C++函数库，通过Java的JNI机制暴露给应用框架层的API，进而提供给开发者调用。JNI机制还可以让开发者编写自己的链接库文件，使得APK加壳等反逆向功能得以实现，增加了APK破解难度，但同时给恶意软件逃避检测提供了机会，增大了恶意软件检测特别是静态检测的难度。该层同时还有Android运行时，包括了核心库和Dalvik虚拟机。前者兼容了大多数Java语言所需要调用的功能函数还包括了Android核心库，比如android.os、android.media、android.net等，后者是一种基于寄存器的Java虚拟机，主要完成应用生命周期、堆栈管理、线程管理、异常管理以及垃圾回收等总要功能，是Android应用能够跨平台的基础。

表2.1 Android系统版本和Linux内核版本对应关系

Table 2.1 The Corresponding Relationship between Android and Linux kernel version

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 英文名 | 中文名 | Android版本 | 发布时间 | Linux内核版本 |
| Cupcake | 纸杯蛋糕 | 1.5 | 2009.2.2 | 2.6.27 |
| Donut | 甜甜圈 | 1.6 | 2009.4.30 | 2.6.29 |
| Froyo | 冻酸奶 | 2.1 | 2010.1.12 | 2.6.32 |
| Gingerbread | 姜饼 | 2.3 | 2010.12.7 | 2.6.35 |
| Honeycomb | 蜂巢 | 3.0 | 2011.2.2 | 2.6.36 |
| Ice Cream SW | 冰淇淋三明治 | 4.0 | 2011.11.14 | 3.0.8 |
| Jelly  Bean | 果冻豆 | 4.2 | 2012.10.30 | 3.4.0 |
| Kit Kat | 奇巧巧克力 | 4.4 | 2013.9.4 | 3.8.0 |
| Lollipop | 棒棒糖 | 5.0 | 2014.10.15 | 3.10.0 |
| Marshmallow | 棉花糖 | 6.0 | 2015.5.28 | 3.10.73 |

4) Linux内核：Linux内核是Android核心系统服务的基础，Android系统通过Linux来完成操作系统的基本功能，包括内存管理、进程管理、网络协议栈和驱动模型等。由于移动设备资源有限，内核还需要提供资源管理和优化、电源管理和优化等功能 。同时，Linux内核也是Android系统软件和硬件的抽象接口，Android硬件抽象层通过对Linux内存驱动的封装，把对硬件的支持分成了两部分。Android系统的随着版本的更新，其底层的Linux内核版本也在不断的改变，表2.1展示了Android系统版本和Linux内核版本的对应关系。

Android系统的框架结构采用分层思想来构建，极大的减少了各层之间的依赖性，便于独立分发、容易收敛问题和错误。各个层次可以单独编译，方便Android设备厂商定制具有自己特色的Android系统，也可以更好的适配更多的硬件平台。

此外，为了保护一些硬件提供商的知识产权，Android系统提出硬件抽象层(HAL)，如图2.2所示。Android系统中把对硬件的支持分成两层：一层在Linux内核中为内核驱动层，一层在用户层为硬件抽象层。HAL包含了对硬件控制的所有操作，Linux driver仅仅完成一些简单的数据交互操作。由于Android系统在内核空间中的驱动程序对硬件的支持不完整，增加了移植Android系统的难度。

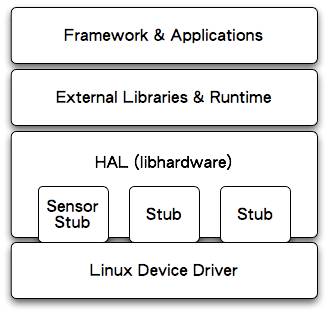


图2.2 Android系统硬件抽象层

Fig 2.2 HAL of Android OS

### Android应用结构概述

Android应用是Android系统的主要构成部分，实现了智能设备的功能的多样性，对Android系统的推广普及产生了极大的帮助。Android的应用主要是用户界面方面的，通常用Java语言进行编写，还包含各种资源文件。Java程序及相关资源经过编译后，将打包生成一个APK包。Android系统本身提供了主屏幕(Home)、联系人(Contact)、电话(Phone)、浏览器(browser)等众多的核心应用，同时应用的开发者还可以开发自己的应用，通过调用应用框架层提供的API，丰富Android系统的功能。

Android APK全称是Android application package(Android应用包)，每个要安装到Android系统上的应用都要单独打包成为一个APK格式的文件，其实质是一个ZIP格式的压缩文件，解压APK文件后看到的目录结构如表2.2所示。

表2.2 Android 应用包目录结构

Table 2.2 The Directory Structure of APK

|  |  |
| --- | --- |
| 文件或目录 | 说明 |
| META-INF/ | 一个 manifest目录 ，从jar文件引入包描述信息目录 |
| res/ | 资源文件目录 |
| libs/ | 存放NDK编译出来的.so链接库 |
| AndroidMainfest.xml | 应用的全局配置文件 |
| classes.dex | Java类编译后的文件，所有的Java都存储在该文件中 |
| resources.ars | 编译后的二进制资源文件 |

res文件夹是由资源文件夹，由drawable、layout、menu共3个子文件夹组成，分别存放了APK的图片资源、布局文件、菜单文件等。

META-INF文件夹存放的是APK的签名信息，用来保证APK安装包的完整性。在生成APK安装包的时候，打包程序会对所有要打包的文件做一个完整性校验，并把计算结果存放在该本文件夹下；在安装APK文件的时候，Android系统对安装包里面的所有文件按照相同的算法进行校验，并对比META-INF文件下的校验值，如果不一致则拒绝安装该安装包。如果直接替换、修改某一文件，然后直接重新打包则该软件不可能安装成功，这给病毒感染和恶意修改增加了难度，有助于保护系统的安全。

AndroidMainfest.xml是Android应用的全局配置文件，该文件保存了APK的包名、版本信息、SDK版本、应用组件等配置信息，此外应用所需要的权限信息也是在这个文件中申明的。APK文件在安装过程中，系统会解析AndroidMainfest.xml文件中的信息，并存放在/data/system/packages.xml文件中，通过读取该文件可以得到Android系统中全部的应用信息，对恶意软件的检测分析很有帮助。Android应用的组件必须在AndroidMainfest.xml文件中注册后才可以使用。

classes.dex文件是可以直接在Dalvik虚拟机中加载并运行的文件，是APK包中的核心文件。开发者使用Java编写的源代码，通过编译生成.class文件，然后由dx文件Java字节码转换为Dalvik字节码。通过分析Dalvik字节码，可以找到应用的行为，从而判别应用是否为恶意应用。classes.dex文件结构信息在4.2.1中有详细介绍。

### Dalvik虚拟机

虚拟机是指通过软件模拟具有完整硬件系统功能的、运行在一个完全隔离的环境中的计算机系统。Dalvik虚拟机是Google等厂商合作开发的Android移动设备平台的核心组成部分之一，它可以运行.dex(Dalvik Executable)格式的Java应用，是Android应用能够跨平台的基础。大多数虚拟机包括JVM都是一种基于堆栈的虚拟机，而Dalvik是基于寄存器的。

Android应用都是使用Java语言编写的，而Java语言编译完成后会有很多.class文件，有不少冗余信息，比较浪费空间。而Android系统主要是运行在移动设备上，而移动设备会受到电池电量、运行内存的限制，为了满足性能和功耗的限制，Dalvik虚拟机有一些独特的、有别于其他虚拟机的特性：

1. 虚拟机本身很少，使用的空间也很小;
2. 使用特殊的.dex文件格式，该文件经过专门的优化，把全部.class文件都合并到同一个文件中，所有文件共享一个常量池，大大减少了内存的使用；
3. 常量池已被修改为只有32位的索引，以简化解释器，加快执行速度；
4. 基于寄存器实现，虽然单个指令的长度增加(需要指明选择的寄存器)，但是不需要额外的指令从堆栈中获取数据，大大减少了指令的数目。

这些特有的属性使得Android应用包更小、执行速度更快。此外，Android系统还通过共享内存等方式来加快程序执行。

每个Android应用都运行在一个Dalvik虚拟机实例里，每个虚拟机实例都运行在一个单独的进程空间里。Zytoge进程是一个虚拟机进程，也是Dalvik实例的孵化器，是所有Android应用的父进程。Zytoge进程是在系统启动时创建的，它会完成虚拟机的初始化、Android核心库的加载、预制类库的加载和初始化等。每当系统需要执行一个Android应用时，Zytoge通过复制自身，最快速的提供一个进程来执行该应用。此外，对一些只读的系统库，所有的虚拟机实例都和Zytoge共享一块内存区域，这样可以大大节省内存开销。同时，通过对比Zytoge和用户进程中的共享库，可以找到用户进程独有的共享库，对恶意软件的检测也有一定的帮助。

## Android安全机制

Android系统构建与Linux操作系统之上，并有自己的一些特性[3]。Android系统通过继承和创建，对系统架构中各个层次的安全性都有针对性的增强机制，既有传统的Linux安全机制也有Dalvik虚拟机的安全机制。此外，Google针对Android系统本身也开发了一些独特的安全机制。下面对其进行分别介绍。

### Android内核安全机制

Android内核安全机制主要包括程序沙箱机制、文件访问控制等[4,5]。程序沙箱是指系统把每个进程都封闭在一个单独的运行环境中，把各个应用单独分割开单独运行、互不影响。文件访问控制规定了系统中用户的文件访问权限，具有相应权限才可以执行相应操作。

Android系统是一个多进程操作系统，使用沙箱技术来实现不通进程之间的数据隔离和保护。每个应用在安装的时候，都会被分配一个uid，并为其创建一个沙箱(Dalvik虚拟机)。uid在创建的时候确定，并且以后永久不变，Android系统通过uid来唯一标识一个应用。在应用运行时，每个进程都运行自己的地址空间，互不影响，且应用权限分配过后，不管应用如何运行，其对应的权限是固定不变的。Android系统中的uid对应Linux内核中的用户ID，而Linux系统中所有的资源都有用户和组访问属性，不用用户之间的资源不能够相互访问，从而实现了资源访问的隔离。在Android系统中，之后uid为”root”或”system”的进程才可以访问系统资源，对于其他应用，如果要访问系统资源或者其他应用的资源，则必须申请响应的权限或者使用共享uid。

Linux系统中文件系统除用户和用户组的权限控制外还有读写权限，使用RWX三元组合来实现读写权限。对于系统镜像只有只读权限，所有重要的可执行文件或配置文件都存储在固件或系统镜像中，都只有只读权限，这样就防止了恶意软甲对这些文件的修改。对于需要执行的文件，必须具有可执行权限才可以真正的真行成功，这一特性可以在分析可疑进程时使用到。

### Android应用框架层安全机制

Android应用层的安全机制主要包括签名机制和权限机制。签名机制在2.1.2中有简单介绍，可以用来鉴别应用是否完整、是否被篡改等。权限机制(Permission)是Android应用核心的安全机制之一，其本质就是一种访问控制机制，可以用来限制应用的安装和运行。

相比IOS系统严格控制生态系统，只允许通过官方应用商店安装应用，并对每份上传的应用进行仔细而严格的审查，Android系统作为一种开发的手机操作系统，就没有这么严格。因此，在安装Android应用之前，会详细的告诉用户该应用申请的权限，由用户判断是否应该申请这些权限并要求其确认是否安装。如果用户拒绝这些权限的申请，则安装过程会自动结束。用户同意安装后，Android系统会解析应用程序的配置文件，并把配置信息写入到一个文件中，并和分配的uid进行关联。

Android系统提供了大约134个Permission用于保护系统资源的访问，并提供了对应的API接口来访问这些系统资源，只有申请了对应的权限才能够使用这些API。在应用运行时，需要访问指定的API，Android系统会通过对比uid和安装时申请的权限，如果没有对应的权限，则拒绝执行。

## Java平台相关机制

Android应用都是使用Java语言编写的，因此有必要了解Java平台的相关的机制。本节主要介绍Java平台的JNI机制和反射机制。

### JNI标准

JNI全称是Java Native Interface(Java本地接口)，是Java平台的一部分。JNI是本地编程接口，使得Java虚拟机内部运行的Java代码能够调用其他编程语言编写的应用和库函数。本地代码在Android系统中以动态链接库.so文件的形式提供，当Java类需要本地函数组件进行沟通的时候，Dalvik虚拟机会载入相应的组件供Java代码调用。

Java的成功在很大程度上得益于丰富的API和良好的设计，为广大开发者节省了大量的时间提高了开发效率。但API存在局限性，通常在以下情况下会使用JNI：

1. 需要注重处理速度。Java代码的在执行的时候是通过解释器逐行执行的，相对C/C++代码的执行效率要慢很多，在对执行效率要求比较高的情况下，可以通过JNI来调用本地代码以提高执行速度。
2. 需要与平台相关。当应用需要使用到平台相关的属性的时候，Java类库可能并不支持对这些属性的调用，可以使用JNI来直接操作硬件属性。
3. 对既有代码库的复用。当使用C/C++等本地语言编写的软件需要移植到Android系统的时，不需要直接全部使用Java重新实现，而可以使用JNI机制直接调用写好的库函数即可。

Android系统本身就是基于JNI实现的。从本质上说，Android系统是Linux系统的一个子集，使用Dalvik虚拟机对Java平台进行了良好的封装，提供给开发者一个高可用的SKD模板，而SDK函数的底层实现都是通过JNI接口调用本地函数来实现的。

此外，由于逆向C/C++代码比逆向Java代码难很多，恶意应用为了逃避安全软件的查杀，可以通过使用JNI来调用本地代码恶意代码进行加密、加壳等。

### Java反射技术

Android系统源码中很多类中的字段或方法上标有”@hide”注解，其作用是使其在SDK中不可见，即不对开发者提供该API。这种封闭的字段或方法很多，比如ConnectivityManager移动数据连接模块、ServiceManager蓝牙模块等，一般都是一些敏感或底层的API，直接让开发者对其进行操作具有很大的风险，而开发者可以通过Java的反射机制来调用这些API。

Java反射机制允许运行中的Java程序对自身进行检查，并能直接操作程序的内部属性或方法，可以动态生成任意类的实例、变更属性值、调用私有方法等。Java反射机制的实现主要是通过java.lang.Class对象来实现，调用Class.forName()可以获取到任意指定名字的Java类对应的Class对象，而通过Class对象可以创建类实例、调用方法、修改属性等功能。使用Java反射就可以调用到隐藏的API，也可以动态调用其他API，增加静态分析的难度。

## Linux相关机制介绍

从2.1.1的介绍可知，Android系统底层的内存管理、进程管理、网络协议栈和驱动管理等核心服务都是通过Linux来实现的，了解Linux相关技术可以更好的完成对Android系统的恶意软件检测。本节对分析内存镜像需要用到的知识进行简单的介绍，包括虚拟文件系统、LKM内核可加载模块和内核符号表等。

### 虚拟文件系统

虚拟文件系统VFS是Linux内核的基本组成部分。从功能实现角度来说，Linux内核只做了两件事情：执行进程和给进程提供一个与外界交互的窗口。虚拟文件系统即是提供一个交互的窗口。对于虚拟文件系统来说，它一方面要保证快速访问实际文件系统中的数据，一方面还要保证文件和数据能够正确保存。VFS的设计思想是使得不同的文件系统的实现对用户保持透明，从而提供一个统一的、抽象的文件系统接口，使程序使用同一组系统调用，如read()、write()、open()等操作，图2.3展示了虚拟文件系统的架构信息。

proc文件系统就是一个虚拟文件系统，通过文件系统的接口实现，用于输出系统的运行状态，只存在内存当中，而不占用外存空间。它以文件系统的形式为应用进程和操作系统之间的通信提供了一个界面，用户和应用可以通过/proc得到系统的信息，设置可以修改某些系统配置。由于系统的信息(例如进程、CPU使用率等)是动态改变的，所以只有当用户或应用真正读取proc文件时，其才会从内核系统中读取所需要的信息的。



图2.3 虚拟文件系统架构图

Fig 2.3 The Architecture of VFS

/proc目录下的文件都是特殊的文件，目录下的主要内容包括以下几个部分：

1. 进程PID号作为文件名的子目录，其中包含了进程的运行环境和运行状态，包括环境变量、命令行、CPU占用时间、内存映射表、已打开文件的inode等信息。其中PID为文件名中的子目录task包含了该进程下的所有线程信息；
2. 系统中各种资源的管理信息，如/proc/swaps、/proc/slabinfo等；
3. 系统中各种设备的相关信息，位于/proc/devices目录下；
4. 文件系统的相关信息，如/proc/mounts、/proc/filesystem等；
5. 动态可加载模块的相关信息，如/proc/modules、/proc/kallsyms等；
6. 系统版本号以及其他各种统计与状态信息。

用户空间中很多命令的实现都是通过调用/proc虚拟文件系统，查找指定的内核数据结构来实现的，如图2.4所示。



图2.4基于procfs的系统调用

Fig 2.4 System Call base on Procfs

### LKM动态可加载模块

Android操作系统的Linux内核属于单内核体系结构[5]，它把进程管理、内存管理、中断处理、文件系统等编译为一个整体，以实现内核子模块中间的紧密衔接。这种内核体系结构提供良好的运行速度和稳定性，但是也导致系统的可扩展性比较差。为使内核保持较小的体积并能够方便的进行功能扩展，Linux提供了LKM[6]( Loadable Kernel Modules,可加载内核模块)机制。

可加载内核模块是已完成编译并连接可执行文件的程序，能够实现内核的部分功能，在系统需要的时候将其动态的加载到内中并成为内核中的一个组成部分，而不必对内核重新编译或重启系统。在不需要的时候，可以将其从内核中卸载。这种机制在保证了内核可扩展性和灵活性的同时，保证了内核镜像拥有比较小的体积。

由于模块在插入后作为Linux内核的一部分来运行的，可以在模块中使用一些由内核导出的资源如sys\_call\_table等，根据该地址直接修改系统调用的入口，实现改变系统调用。因此LKM可以当作后门来使用，一旦攻击者能够将恶意内核模块加载到系统内核，就可以肆意的对系统进行各种，如隐藏文件、进程、网络连接等。而这些攻击处在系统的内核层面，所以能够逃过一般的应用级安全工具的检测。此外，LKM也可以当作取证工具来使用，通过加载进入内核中，可以访问到内核中的一些信息，比如本文用到LiMe内存镜像获取工具。

### 内核符号表

符号是程序建立的块，是程序中的变量或者函数。Linux内核是一个很复杂的程序集合，包含很多全局符号，且Linux内核不使用符号名而是通过符号地址来识别变量或函数名，调用符号的时候需要找到对应的地址。因此需要一个文件来做映射，System.map[7](内核符号表)完成了这个映射关系，它列出了所有内核中用到的符号和对应的地址。System.map在编译的时候产生，并且每次内核改变的时候，内核符号表也会被改变。

在Linux2.6内核中引入了kallsyms[8]机制,其功能就是把内核中用到的所有内核符号地址和名称链接写入内核文件，并且在Linux内核启动后把kallsyms信息加载到内存中方便以方便内核的调试。kallsyms抽取了内核中用到的所有的函数地址(全局的、静态的)和非栈数据变量地址，生成数据块并作为只读数据链接到vmlinux中，通过proc虚拟文件系统下的/proc/kallsyms文件可以查看。因此Linux2.6及以后的内核不再依赖System.map文件，而是依赖kallsyms机制。

只有在内核中使用EXPORT\_SYMBOL或EXPORT\_SYMBOL\_GPL导出的符号才能在内核模块中直接使用。而在Linux2.6.37以后，为了保护Linux内核安全，内核符号不再允许被直接导出，以root权限去导出符号表也是全0，需要编译内核时指定编译项或者修改内核变量控制文件才可以获取到内核符号表。

## 本章小结

本章首先介绍了Android系统的基本概念，包括其分层体系结构、应用包的结构信息以及Android Dalvik虚拟机的基本知识。在体系结构的介绍中，分层对Android系统的各个层次进行了介绍，然后介绍了Android系统的安全机制和Java平台上的特殊机制，最后对Linux内核的相关机制进行了详细介绍，包括虚拟文件系统、可加载内核模块以及在内存镜像分析中需要用到的内核符号表信息。通过对这些技术的简单介绍，建立对Android系统的整体认识以及对基于内存镜像分析的恶意软件检测需要用到的技术有一个直观的认识。

# Android内存镜像分析

第2章介绍了Android的结构体系，其进程管理、内存管理等都是通过处于最底层的Linux内核来实现的，分析Android的内存镜像需要熟悉Linux的内核数据结构、进程和内存管理方式。本章首先介绍了Linux内存和进程管理方式，然后提出了一种通过对比进程号列表的方式来查找隐藏进程的方法，以及根据网络端口、动态链接库和Android的权限机制来查找可疑的进程的方法。

## Linux进程管理

进程是程序在一个数据集上的一次运行，是软件在操作系统中运行表现形式。基于内存镜像的恶意软件检测，就是对内存镜像中存在的进程信息进行分析。本节对Linux内核的进程管理进行简单的介绍，包括进程基本概念和进程调度等。

### 进程和线程

进程(process)是Unix操作系统最基本的抽象之一[10]，是一个程序在一个数据集上的一次运行。进程定义了一个执行环境，除了可执行程序代码外还包含其他资源，包括打开的文件、挂起的信号、内核内部数据等，是系统资源分配的基本单位。进程有四个基本的组成部分：程序，专属的堆栈，进程的控制块，系统资源(CPU、内存)等，是系统资源分配的基本单位。

线程是共享在地址空间中的控制流，除了调用栈以外没有其他的资源，线程栈记录了线程执行历史。由于进程的所有线程共享进程的资源，在CPU调度同一进程不同线程的时候就不需要切换进程的上下文状态，使得调度更快的完成，因而线程是CPU调度的基本单位。

Linux内核中，通常把进程称作任务(task)，并使用task\_struct来描述进程，称作进程描述符(process descriptor)[11]。task\_struct包含了进程相关的全部内容，如进程的属性、进程的状态、打开的文件等，进程描述符和进程之间的关系是一对一的，并且使用标识符PID来标识进程。而对线程的处理上，Linux内核把它当作了进程处理，没有特殊的数据结构和调度策略专门为线程服务。线程同样使用task\_struct来描述，和进程的区别就是线程需要共享父进程的其他资源。这样做简化了线程的设计和调度方式，所有的调度都是通过对task\_struct的调度来完成。

### 进程调度

进程是程序在运行时的表示形式，而程序的运行需要CPU的支持。CPU在同一个时刻只能执行一个任务，而Linux是典型的多任务操作系统，因而需要特殊的数据结构和调度策略来组织多个任务的有序执行。Linux的进程调度是基于分时技术的，允许多个进程并发意味着CPU的时间被分为多个时间片，给每个可运行的进程分配一片，然后每次选择一个进程并分配指定时间片长的CPU时间给该进程执行。

进程从创建到销毁会经历几种状态的改变，task\_struct表示了第一个字段state表示了进程目前的状态。state的值可以包含0(可运行&运行中)、1(可中断睡眠状态)、2(不可中断睡眠状态)、4(暂停状态)、8(追踪状态)、16(僵死状态)等，各个状态之间可以相互转换，转换过程如图3.1所示。

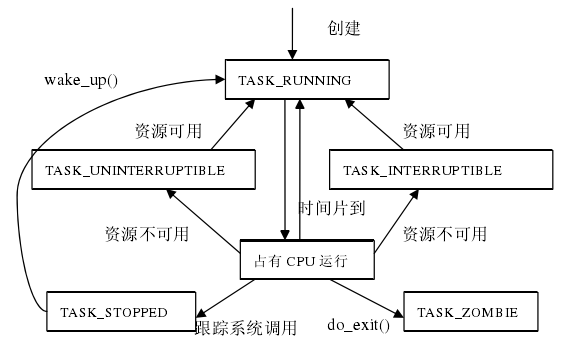


图3.1 进程状态转换图

Fig 3.1 The Transition Diagram of Process State

当state值为0的时候，表示进程处于可执行状态，可以参与CPU的调度。进程调度是操作系统的核心功能，CPU需要从所有的就绪进程中选择一个优先级最高的进程来执行。

Linux2.4之前是采用的O(n)调度器。在task\_struct中通过两个指针把所有的可运行状态(state=0)的进程连接成为一个循环双链表，CPU调度的时候每次需要遍历整个双链表来找到优先级最高的进程，为其分配CPU的执行权。采用静态优先级和动态优先级相结合的方法来确定进程的最终优先级，实时进程具有高于所有普通进程的优先级，普通进程可以设置基本优先级，然后根据执行时间的划分来确定动态优先级。O(n)调度器的性能是与就绪队列长度相关，因而进程数目受到限制，并且由于所有的CPU都共享同一个就绪队列，为了防止因为竞争产生的错误需要额外的加锁，效率更为底下。

为了改进O(n)调度器，在Linux2.6.0中实现了O(1)调度器，调度时间恒定，和就绪队列的任务数没有关系。O(1)借助非常复杂的公式在常量的时间内获取到优先级最高的进程，但是有些程序会让该调度器的性能显著下降，导致交互式进程反映缓慢。此外，O(1)调度器需要确定优先级的公式太多，使用了大量难以维护和阅读的复杂代码，虽然性能得到了解决，但是由于代码过于复杂，后续的维护非常困难。

因此在Linux2.6.36中采用了完全公平调度器CFS(Completely Fair Scheduler),不在使用各种复杂的公式来完成对进程进行划分，而是对所有的进程平等对待，所有任务的执行时间都按照同一规则来决定。在全局上依旧将进程划分为实时进程和普通进程，只改进了对普通进程的调度策略，不再使用优先级队列而是采用了红黑树的实现，红黑树的每个节点的键值是定义的虚拟运行时间也是有优先级来决定的。由2.1.1讨论可知，Android2.2以后的发行版本都是采用的2.6.3+的Linux内核版本，使用的调度器均是CFS调度器。

## Linux内存管理

在计算机科学领域中，内存通常指处理器通过总线可以直接访问且不属于处理器的物理存储器。内存镜像分析，就是把系统当前的物理内存中的数据镜像复制出来，然后针对次镜像文件进行分析并提取出有用的信息。内存镜像分析的一个关键点是操作系统使用的是逻辑地址，需要把它转换为物理地址。Android系统是跨平台的系统，可以支持多种处理器，智能手机上一般使用的是Arm处理器。本节首先介绍了基于Arm处理器的内核虚拟地址转换，然后介绍了一些在恶意软件检测过程中需要用到的内核数据结构。

### 内核虚拟地址转换

进程地址空间由该进程可寻址的虚拟内存组成，进程虚拟内存空间包括内核态空间和用户态空间，其中内核态空间是所有进程共享页表页目录。Linux 简化了分段机制，使得虚拟地址与线性地址总是一致，因此， Linux 的虚拟地址空间也为0～4G 字节。Linux 内核将这4G 字节的空间分为两部分：将最高 的1G 字节，供内核使用，称为“内核空间”；将低位的3G空间，提供给各个进程使用，称为“用户空 间”，如图3.2所示。因为每个进程可以通过系统调用进入内核，因此，Linux 内核由系统内的所有进程共享。内核符号表枚举的内核变量和函数的地址是虚拟地址，而内存镜像中的地址都是物理地址，在分析内存地址的时候需要把虚拟地址映射成物理地址。



图3.2 虚拟地址空间分配示意

Fig 3.2 The virtual address space allocation

内核维持着一组自己使用的页表，驻留在内核全局也目录中。该目录存放在全局变量swapper\_pg\_dir中，大小为PAGE\_SIZE(在32位环境中为4K，64位环境中为8k,android大多数都是32位环境)。

在ARM Linux系统中有两种页表结构：一种是3G以上一级页表，用于内核态寻址，是静态的；一种是ARM MMU的两级页表结构，用与虚拟地址转换为物理地址，是动态的。Linux操作系统从3G开始，把虚拟地址沿着物理内存开始，一直映射到物理内存的边界，这部分页表只起到虚拟地址到物理地址的转换作用。3G以下的地址采用二级页表模型，不过为了使得ARM MMU能够在Linux的虚拟内存框架下工作，做了一些设计：在Linux虚拟内存模型下，需要记录页面的dirty、可读、可写等属性，但是ARM L2的页表项并不能跟Linux的要求一一对应，而且L2页表项又被硬件全部占满，所以对于每个L2页表Linux准备了两份，一份给硬件MMU使用一份给Linux本身使用。

实际上在物理内存小于4G以及没有其他映射方式的时候，主内核全局页目录从768至991是从物理内存0处开始映射的。从页目录768到1023都采取直接映射最多也只能同时覆盖1GB物理内存，因此内核将最后的32项不参与直接映射而是可以多次重复映射，每次映射都是暂时的。前面的224个页表直接映射(可以同时覆盖896M物理内存)，后面的32项页表通过重复映射，能覆盖的地址就不止128M。一个计算机系统的物理内存大小都是固定的，所以前面224个内核页表（对应页表目录项768-991）的内容可以确定。如果物理内存小于896M，则可以只使用768~991的页表就可以全部覆盖，页表项的内容在整个系统运行期间固定不管；如果物理内存为2G，那么Linux在启动的时候先为前面的896M物理内存建立内核页表，剩下的1G+128M的物理内存不得不在使用某个页框的时候用992到1023项页目录的临时映射。

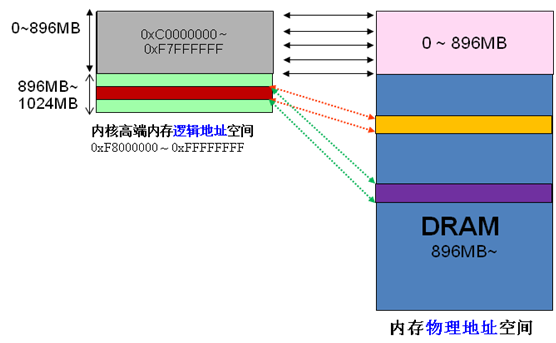


图3.3 内核高端地址映射方式

Fig 3.3 The Mapping of High Address

但是如果出现固定映射、永久内核映射以及非连续内存映射的时候，这种线性规则被打破。因此，尽管Linux提供宏操作\_pa()来实现内核虚拟地址向物理地址转换的方法，在实际应用中还是要根据主内核也全局目录来检查转换的合理性，检查方式如(3.1)所示。

\*(pgd + i) == (i – 768)\*4M + 0x1e3 (3.1)

如果第i(991>= i >=768)页目录项不满足该式，那么就不能通过\_pa()进行地址地址转换，而要用模拟硬件的MMU进行地址转换，转换过程在文献12中有详细描述。

### 重要内核结构

内存镜像的数据是直接从内存中复制出来的，只有熟悉相关的数据结构才可以从内存镜像获取有用的信息。下面对本文用到的Linux内核中的重要数据结构进行简单的介绍，包括进程描述符、内存描述符、进程就绪队列以及多出地方用到的内核链表等。

1. 进程描述符task\_struct。
2. 内存描述符mm\_struct。
3. 进程就绪队列runqueue。
4. 内核链表结构。

## 隐藏进程检测

系统层面的内核级恶意软件为了防止被安全软件检测出来，可能会采用各种手段来隐藏自己，其中最常用的就是进程隐藏和文件隐藏。与此对应的是，正常软件并不会有这些隐藏操作。因此，通过检测内存镜像中的隐藏进程可以很好的发现系统中正在执行的内核级恶意软件。本节首先介绍了常用的进程隐藏方法，并基于这些方法的缺点和Linux本身的一些机制提出了基于进程列表对比的内存镜像隐藏进程检测方法。

### 隐藏进程常用方法



图3.4 系统调用劫持进程隐藏示例

Fig 3.4 Hide Process by Hijacking System Call

所有的操作系统都会把内核中的运行状态(进程信息、系统内核状态等)以对象的形式保存在特定的数据结构中(结构体、数组、链表等)。这些结构体称为内核对象，3.2.2中介绍的内核结构都是内核对象。很多系统提供的API都是通过访问内核对象读取响应的信息，返回需要的结果。通过替换API函数的入口点，使其指向自定义函数，劫持对内核对象的访问，从而可以实现需要的操作，这就是KOH。在系统运行时，内核对象都是存储在内核中的，所以可以修改这些内核对象。直接修改API函数访问的内核对象，从而改变API的返回结果，实现某些隐藏功能，就是DKOM的实质。通过修改task\_struct中的list\_head指针，将其从全局进程链表中移除，就可以实现进程的隐藏，详细过程如图3.5所示。



图3.5直接修改内核对象实现进程隐藏示例

Fig 3.5 Hide Process by DKOM

### 隐藏进程检测方法

在Linux系统中，通过ps或top命令可以查看系统中正在运行的进程，而ps命令的原理是通过/proc虚拟文件系统实现的，相关过程如图2.4所示。无论采用何种方法来隐藏进程，都需遵循一个原则：一个进程需要被执行，必须通过CPU调度。根据3.1.2的介绍可知，Android系统都是基于CFS调度器的，该调度器对所有进程公平对待，通过基于红黑树的就绪队列runqueue来完成进程的调度的。因此，本文基于内存镜像的隐藏进程检测方法如下：

1. PID出现在展示队列中，而没有出现在ps命令结果ps\_tasks中，即
2. PID出现在进程直接互联的全局进程链表all\_tasks(以init\_task为循环双链表的头部)中，但是并没有出现在展示进程队列中,即；
3. PID出现在runqueue\_tasks链表中，而没有出现在全局链表all\_tasks中，即。

其中ps\_tasks是使用ps命令获取的所有进程列表，list\_tasks是通过ls /proc目录然后抓取数字目录而获取的进程列表，all\_tasks是3.2.2中讨论的struct list\_head tasks的循环双链表，runqueue\_tasks是所有CPU调度时的就绪队列总和。通过对比以上几种tasks中的进程链表，如果满足三条中任意一条则认为是该进程有很大可能是隐藏进程，可以认为该进程所对应的软件或模块为恶意软件。



图3.6 基于进程号对比的隐藏进程检测方法

Fig 3.6 The way of Hidden Process Dected base on Contrasting PID

由3.1.1的讨论可知，Linux中进程和线程都是使用task\_struct的结构体来表示，区别只是是否共享父进程的资源，而ps\_tasks只会包含进程、all\_tasks会包含进程和线程，所以list\_tasks会有两个不同的展示，一个是只包含进程的，一个是包含了线程的，的获取方式是/proc/${pid}/task目录下的所有数字目录。隐藏进程检测的详细过程如图3.6所示。

## 可疑进程检测

由2.1讨论可知，Android系统是一个上层操作系统，其对外接口是通过Android应用APK来完成，因而恶意应用比内核级的恶意软件更多。内核级恶意软件需要通过感染Rom然后让用户刷机才能感染，而应用层的恶意应用只需要诱导用户安装就可以实现。本节首先介绍恶意应用的典型行为，然后根据恶意应用的典型行为提出了可疑进程的检测方法。

### 恶意应用的典型行为

根据安全厂商Symantec在2013年发布的一份安全报告显示，Android恶意应用带给用户的危害主要包括以下四个方面：1)恶意扣费，在用户不知情的情况下订购付费业务，并屏蔽设置自动回复确认短信；2)隐私窃取，窃取用户的短信、通话、联网、位置等信息并上传到指定的服务器。3)消耗资费，在后台自动联网，不断下载插件、视频等垃圾信息，恶意消耗用户流浪，对用户造成损失。

通过分析已知恶意应用的代码，可以发现恶意应用主要是通过申请敏感权限甚至利用漏洞获取到root权限，然后利用短信、电话、网络、推广其他垃圾应用、读取系统敏感信息等方式来达到非法目的。恶意软件在获取短信发送权限后，在用户不知情的情况下，后台发送付费业务的短信，拦截并自动回复订购业务的确认短信。拦截原理是通过Android应用的广播机制的漏洞，设定高优先级的广播接收器(Broadcast Receiver)，当接收到短信回执的时候判断是否需要中断广播即可[14]。恶意程序一般通过两种方式来使用网络：一种是和隐私信息泄漏结合，在后台自动联网并自动上传用户隐私信息；一种是不断下载垃圾软件消耗用户流量同时可以推广其他应用，依靠广告费来获利。

### 可疑进程检测方法

从3.4.1的介绍可知，恶意应用的一般会使用网络来上传或下载信息，需要网络通信的支持。使用加密、加壳等方法进行恶意代码隐藏的时候需要通过JNI机制来调用动态链接库.so文件。当访问短信、网络、电话的时候需要特殊的权限，需要在AndroidMainfest.xml文件中申明。因此可以从网络端口、私有动态链接库以及APK申请的权限来检测进程或应用是否可疑。

1. 网络链接检测
2. 动态链接库检测

Android应用可以使用JNI机制来调用动态链接库，可以用来完成一些需要效率或和硬件相关的功能。恶意应用通常会使用动态链接库来完成一些加密或其他的恶意行文。动态链接库需要加载到内存中才能被调用，在/proc/${pid}/maps目录下可以看到进程已经加载的所有动态链接库。从2.2的介绍可知，Linux中使用RWX3中权限来限制资源的读写属性，而动态链接库以函数的形式存在，则必须具有执行权限才可以。

在内存镜像中，task\_struct存储了进程的所有信息，也包括加载的动态链接库信息。从3.2.2的介绍中可知，进程描述符task\_struct包含了该进程对应的内存描述符mm\_struct信息，而mm\_struct中包含了内存区域描述符(vm\_area\_struct)，该结构体描述了指定内存地址空间内连续区间上的一个独立内存范围。Linux内核将每个内存区域当作一个单独的内存对象管理，每个内存区域都拥有一致的属性，每个内存描述符都对应于进程地址空间中的唯一地址。vm\_area\_struct的主要字段如下所示：

struct vm\_area\_struct {

struct mm\_struct \* vm\_mm; //相关的mm\_struct对象  
struct vm\_area\_struct \*vm\_next, \*vm\_prev; //所有的VMA组成链表的链表指针

unsigned long vm\_flags /VMA/标志

}；

其中vm\_flags标了该内存区域的访问属性,1表示可读、2表示可写、4表示可执行。通过链表指针可以找到指定进程所有的内存区域描述符，查找具有执行权限的内存区域，就可以找到该进程加载的所有的动态链接库。

从2.1.3的介绍中可知，Android应用进程都是以Dalvik虚拟机进程表示的，而Dalvik虚拟机进程是从Zygote进程孵化出来的。对一些可以共享的动态链接库，所有的Dalvik进程都是共享Zygote进程的链接库地址。通过对比指定进程的动态链接库和Zygote进程的动态链接库，可以找到所有进程私有的动态链接库。从前面的介绍可知，具有私有的动态链接库则可以认为该进程为可疑进程。

1. APK权限检测

权限机制是Android系统安全机制中很重要的一个组成部分。权限机制要求开发者在发布的应用中申请其需要用到的权限，当应用安装的时候需要用户决定该应用是否应该具有这些权限，如果用户拒绝则停止安装该应用。Android系统总共定义了134个权限信息[15]，并对每种权限的含义做了说明，根据权限的危险级别又分为4个大类：normal,默认值，可以自动授予而不需要用户同意；dangerous，较高风险的权限，必须经过用户的授权；signature，签名权限；sinatureOrSystem,系统只会把该权限授予Android系统固件中的应用或具有相同数字签名的应用。

如果仅因为Android软件申请了某个单独的权限而被判定为恶意软件，势必会导致极高的误报率。然后Android应用都会申请多种权限，不同权限的组合蕴含了该应用固有的特性，正常软件和恶意软件之间的权限组合也是存在差异的。虽然具有实施恶意行为能力的软件并不一定会表现出恶意行为，但并不妨碍当作可疑应用进行深入的检测。

在检测过程中，由于是直接对待检测智能手机进行操作，有可能得不到全部处于运行状态的应用安装包APK，造成得不到AndroidMainfest.xml文件从而没有办法提取到对应的权限信息。分析Android系统源码可知，在应用安装过程中，安装程序会调用PackageManagerService.java类中parsePackage ()方法来解析APK中包含的AndroidMainfest.xml文件，获取到其对应的权限、属性、Linux用户ID名称等信息，并写入到文件/data/system/packages.xml中。packages.xml文件结构如表3.1所示。通过对packages.xml文件的分析，并对比前面获取到的进程列表，获取到所有运行进程对应的应用权限信息，通过权限对权限组合的分析，就可以判断该应用是否是可疑应用。

表3.1 packages.xml文件标签描述

Table 3.1 The Tags of packages.xml

|  |  |  |  |
| --- | --- | --- | --- |
| packages  根节点 | permission  定义了系统中所有定义的权限 | name | 定义的权限名 |
| package | 所属的包名，主要分类两类：系统定义的，package为android；APK自定义的，package为APK的包名 |
| package 代表一个安装的APK | name | 对应APK的包名 |
| codePath | APK安装路径 |
| system | 如果安装在/system/app下则为true，否则false |
| userID | 分配的UID,如果为共享的，则为shared-user ID |
| perms | 该APK申请的权限，每有一项则增加一项<item> |
| shared-user  代表一个共享UID | name | 共享UID的名称 |
| perms | 代表了共享UID的权限  每有一个权限则增加一项<item> |

## 本章小结

本章是对Android内存镜像的分析，首先介绍了Android系统的基础——Linux系统的进程和内存管理方式，包括进程调度、内核虚拟地址的转换以及在后续分析中需要使用到的一些内核数据结构。然后在此基础之上，提出了基于进程列表对比的隐藏进程检测技术，以此来发现系统级的恶意软件。对Android应用层的恶意软件，通过分析网络端口、私有的动态链接库以及Android应用本身的权限组合信息来查找可疑进程及对应的可疑应用，以供后续的深入分析。

# Android恶意应用分类检测

## 引言

按照3.4介绍的方法检测出可疑进程后，需要更深入准确的需要进一步检测进程所对应的应用是否为恶意软件。目前恶意代码检测技术有很多，如基于特征码的检测、基于行为的检测、沙箱技术等。基于特征码检测的方法通过匹配已有的特征规律来鉴别恶意软件，对未知的恶意代码检测率比较低。因此基于行为的检测又是恶意软件检测的主要方法，这种方法通过分析软件的行为，来判断待测软件是否具有恶意性。根据检测时机的不同，基于软件行为检测又可以分为动态行为监测和静态行为检测。

动态行为检测又被称为外部行为检测，通过把待测软件运行在沙箱、虚拟机甚至实体机里面，检测软件行为，包括系统调用、状态对比、行为跟踪等，进而判断该软件是否为恶意代码。动态检测方法的效率比较高，结果也比较准确，能够检测到未知恶意软件以及通过混淆、加密等技术变种而来的恶意软件。但是动态检测也有自己的不足，比如实施起来比较复杂，需要特定的运行环境才能监控到软件行为。此外由于一般的动态的行为检测都是在沙箱或者模拟器中进行，受运行环境测限制，一次能够测试的样本数量比较少，增大了恶意样本学习难度，也难以支持大量样本的批量检测。

静态行为检测是指不需要运行待检测软件，而是直接分析软件的安装包所包含的文件、软件的执行代码、依赖的外部资源等，从中提取出程序的运行流程、系统资源调用情况、软件的函数API序列等静态行为，并与已知恶意软件进行对比，从而判断出待测软件是否是恶意软件。静态行为检测实施起来比较简单方便，并且在恶意软件感染前就能检测出来，对于恶意软件的预防有很好的作用。但是对于一些使用了特殊方法来逃避静态检测的恶意软件，则是无能为力。例如有些Android恶意应用采用加壳等方法使.dex文件无法反编译为smali文件，从而无法提取API调用序列得到静态行为，更没有办法进行行为分析。

本文是对恶意软件的检测研究，主要是对待检测Android系统的恶意软件进行分析检测。由第3章的介绍可知，对于系统级的恶意软件是检测隐藏进程等来实现，而对于Android应用层面的Android应用是在可疑进程检测基础之上进行深入分析来完成。本章通过获取可疑进程的内存dump文件中提取dex文件，反编译后提取API调用序列。同时引入集成概率神经网络，先使用大量样本进行学习，用来检测从内存中提取出的dex文件是否为恶意代码，从而完成对恶意应用的检测工作。

## APK逆向分析

从上节的介绍可知，静态行为检测和动态行为检测在恶意软件检测过程中都有不足。结合具体的检测环境和Android系统的特性，通过逆向分析获取已经脱壳、自修改过等正常的dex文件，为后续的静态行为检测提供比较正确的数据。本节首先介绍.dex文件的结构，其次介绍Dalvik虚拟机的类加载机制，最后结合.dex文件的结构提出了一种通过暴力搜索core dump来提取.dex文件的方法。

本节涉及到部分Android源码，使用的AOSP源码[16]是android-4.4\_r1.2，build版本是KRT16S。

### DEX文件结构

classes.dex文件是Android系统运行于Dalvik虚拟机上的可执行文件，也是Android应用的核心所在。从Java源文件生成.dex文件的基本映射关系如图4.1所示。Java源文件通过Java编译器生成class文件，然后通过dx工具转换为classes.dex文件。.dex文件和class文件一个最大的不同为.dex文件将索引结构，类型、方法名、字段名等信息都存储在一个常量池中，而不是每个.class都对应一个常量池，最大可能减少了存储空间。

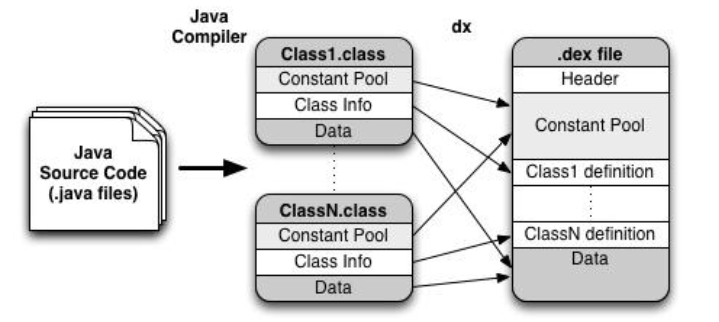


图4.1 Java和.dex文件映射关系

Fig 4.1 The Mapping Relationship between Java and DEX File

dex文件的结构很简单，实际上就是有不同的结构体以首尾相连的方式拼接在一起，一个.dex文件的基本结构如图4.2所示。Dalvik虚拟机在加载.dex文件的时候会验证header信息，所以重点对其header进行介绍。header是.dex文件的文件头，简单的记录了.dex文件的一些基本信息以及大致的数据分布，包含magic字段、校验值、SHA-1哈希值、string\_ids的个数以及file\_size等，其结构固定占用0x70个字节。header的主要字段如表4.1所示。

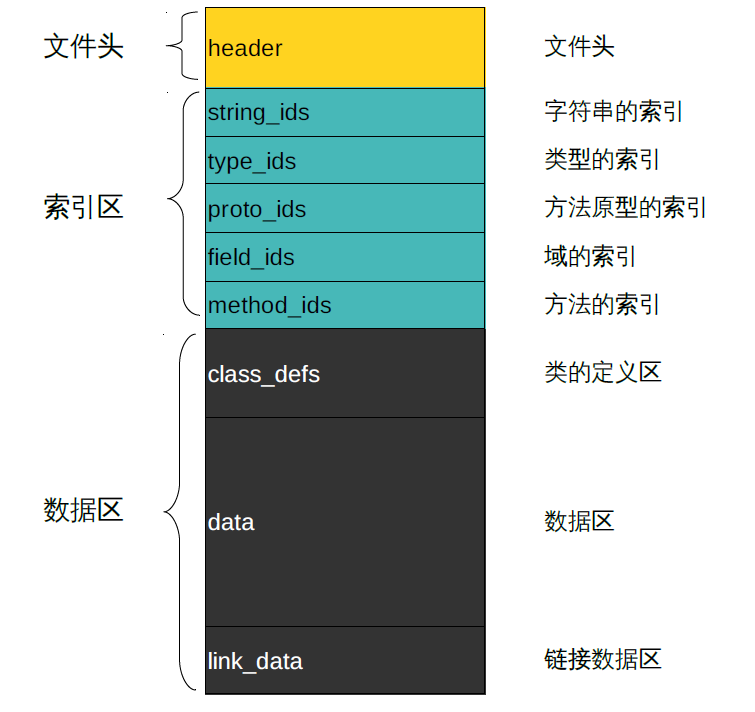


图4.2 .dex文件结构

Fig 4.2 The Structure of DEX File

表4.1 .dex文件header的主要成员

Table 4.1 The Main Member of DEX File Header

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名称 | 偏移值 | 长度 | 描述 |
| magic | 0x0 | 8 | 魔术字段，格式如”dex/n/035/0” |
| checksum | 0x8 | 4 | 校验码 |
| signature | 0xC | 20 | SHA-1签名 |
| file\_size | 0x20 | 4 | .dex文件总长度 |
| string\_ids\_size | 0x38 | 4 | 字符串列表中字符串个数 |
| string\_ids\_off | 0x3C | 4 | 字符串列表基址 |
| class\_defs\_size | 0x60 | 4 | 类定义表中类的个数 |
| class\_defs\_off | 0x64 | 4 | 类定义表中列表基址 |

### Dalvik类加载机制

对于Java虚拟机而言，将类数据加载到虚拟机运行时环境的过程叫做类加载。类加载机制的主要功能是将应用中的操作码以及程序数据提取并加载到虚拟机内部，以保证程序的正确执行。具体到Dalvik虚拟机就是将.dex文件中的指令与数据提取并加载到Dalvik虚拟机与一个ClassObject数据结构进行关联，实现在虚拟机中执行字节码。类加载机制在应用和执行模块之间建立了一个桥梁，处在一个承上启下的位置。图4.3反映了类加载机制在Dalvik虚拟机执行过程中所承担的重要作用。类加载机制主要包括.dex文件的优化与验证、解析与装载以及最后的加载数据时的映射关系，下面分别对这些过程进行介绍。



图4.3 Dalvik虚拟机程序执行流程

Fig 4.3 The Execution Process of Dalvik VM

1. .dex文件验证与优化

当虚拟机获取程序中的classes.dex文件后，首先将这个.dex文件进行初步的验证工作，主要包括：验证Magic，校验SHA-a签名，计算.dex校验和等几个方面，当校验完成后，则开始进行优化。.dex文件优化的目的是根据当前平台特性对应用中的字节码进行部分替换，并为.dex文件增加辅助信息以提高Dalvik运行效率，最后输出一个优化文件.odex。



图4.4 .dex文件和.odex文件的结构对比

Fig 4.4 Structure Comparison of DEX files and Odex Files

直观上.odex文件在.dex文件上进行了扩充，增加了头部信息以及在尾部拼接了依赖库、寄存器映射关系以及哈希索引等辅助信息，.dex和.odex文件的对应关系如图4.4所示。头部结构DexOptHeader整体的描述了.odex的文件结构和各部分数据含义，还包含了.dex文件的描述信息，固定长度为40个字节。DexOptHeader的详细描述见表4.2。

表4.2 DexOptHeader主要成员

Table 4.2 The Main Member of DexOptHeader

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名称 | 偏移值 | 长度 | 描述 |
| magic | 0x0 | 8 | 魔术字段，格式如”dex/n/036/0” |
| dexOffset | 0x8 | 4 | .dex文件偏移量，固定值0x28 |
| dexLength | 0xC | 4 | .dex文件总长度 |
| depsOffset | 0x10 | 4 | 依赖库偏移量 |
| depsLength | 0x14 | 4 | 依赖库信息总长度 |
| optOffset | 0x18 | 4 | 优化数据偏移量 |
| optLength | 0x1C | 4 | 优化数据总长度 |
| flags | 0x20 | 4 | 标志位，标志优化与验证选项 |
| checksum | 0x24 | 4 | 文件校验和 |

为了更大程度的保证原.dex文件的数据的安全性以及优化机制的独立性，优化机制并没有直接改写原.dex文件，而是从新创建了一个odex文件并以严格的机制要求优化信息写入该文件，主要包括依赖库关系、寄存器映射关系以及类的索引关系，这些管理可以提高类加载机制的执行效率。同时在优化过程中还根据平台特性对原.dex文件进行部分字节码替换（如对字段的替换，把查找改为直接引用），最后将修改过后的.dex文件写入.odex文件中。

1. Dex文件解析

.dex文件与由多个Class文件整合而成，这是的Dalvik在加载一个目标类之前需要对.dex文件进行一系列的处理。.dex文件解析的主要目的就是对.dex文件进行读取分析，并建立一个DexFile结构体实例来描述该.dex文件，使得类加载函数可以通过该结构对目标类进行全部的数据进行索引并提取，完成类的实际加载工作。

在应用启动的时候，Dalvik虚拟机把.odex文件直接加载到内存中，使得普通的内存读取操作也可以访问DexFile中的内容。.odex文件在解析过程中比较快的原因就是可以把类索引信息、依赖库信息、寄存器关系信息等提前与DexFile进行关联。完成优化数据的处理之后，Dalvik对写入.odex文件的.dex数据进行解析，将DexFile数据结构中各个成员变量与.dex文件的各个数据部分进行关联，使得虚拟机能够更加高效的对.dex文件中的数据进行查找并获取。

因为关联只是建立了指针链接而不是直接复制的数据，.dex文件在内存中还是连续存在的，这使得从进程的core dump文件中提取.dex文件成为可能。

### Dex文件获取

从Dalvik虚拟机的类加载机制可以看出，APK应用的入口都是通过.dex文件的,因此不管如果对APK本身加密、加壳、混淆等，最后执行时一定会把.dex文件分离、解密、还原出来。

DexFile中各个引用都是直接指向.dex在映射区的位置，并没有自己复制一份数据过去。说明.odex在内存中是连续存在的，.dex文件在内存中也应该是连续存在的。如果能够获取到这个Dalvik虚拟机对应的进程core dump，那么通过4.2.1介绍的.dex文件结构，找到magic魔数字段和file\_size文件长度字段，就可以整个的获取到脱壳的.dex文件。从4.2.2的分析中可知，core dump中实际存在的文件是.odex文件，是Dalvik虚拟机对dex文件进行优化后的文件。结合dex文件格式和.odex文件格式的介绍可知，这两个文件里面有几个固定字段，可以用来判定.odex文件的位置，固定字段见表4.3。

表4.3 .odex和.dex文件的固定值

Table 4.3 The Fixed Value in ODEX files and DEX files

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 字段 | 位置 | 长度 | 值 | 十六进制表示 | 相对偏移 |
| magic | DexOptHeader | 8 | dey/n036/0 | 64 65 78 0A 30 33 35 00 | 0x00 |
| dexOffset | DexOptHeader | 4 | 0x28 | 28 00 00 00 | 0x08 |
| magic | DexHeader | 8 | dex/n035/0 | 64 65 79 0A 30 33 36 00 | 0x28 |
| header\_size | DexHeader | 4 | 0x70 | 70 00 00 00 | 0x4C |

综上，通过暴力搜索core dump文件，获取odex文件的流程如所示，odex文件包含了.dex文件的内容，同样可以反编译得到smali文件。



图4.5 从core dump中提取.odex文件过程

Fig 4.5 The Extraction Process of ODEX file from core dump

## 敏感API提取

API(应用结构，Application Programming Interface)是一些预先定义的函数，目的是提供应用与开发人员基于某软件或硬件得以访问一组例程的能力，而又无需访问源码。Android API位于Android体系结构的第三层，是一套系统调用的接口，为上层应用提供了丰富的框架支持。这些应用利用Android API完成各种软件行为，包括发送短信、访问网络、订阅收费服务等敏感行为，较好的反映了应用的正式行为，可以软件静态行为来进行恶意倾向分析。

### Smali语法介绍

Dalvik字节码(使用Smali语言描述)是Dalvik虚拟机设计的一种指令集，从Java字节码演化过来的，是一种跨平台的中间代码，可以运行在支持Dalvik虚拟机的各种平台上。.dex文件最后的CodeData中的insns字段即是存放了该Dex文件的全部执行指令。和Java虚拟机不同的是，Dalvik虚拟机是基于寄存器的，可以使用的虚拟寄存器达65536个，每个寄存器的大小是32位，两个相邻的寄存器可以表示64位的数据。在实际操作中，Java源码需要先生成Java字节码(class文件)，在经过dx工具转换为符合Smali语法的Dalvik字节码。

Dalvik虚拟机基于寄存器设计，Java字节码转换为Dalvik字节码时，调用栈已经确定，其中明确指出了使用的寄存器个数以及额外的其他数据。Smali语言中，操作码是1字节，操作数是以16字节为一个单元，除了Java对象和数据是引用类型，其他都是原始类型。基本类型被一个简单的字符描述，类型描述符如表4.4所示。

表4.4 Smali类型描述符

Table 4.4 The Type Descriptor for Smali

|  |  |
| --- | --- |
| 语法 | 含义 |
| V | void,只能用于返回值 |
| Z | boolean，布尔值 |
| B | byte |
| S | short |
| C | char |
| I | int |
| J | long(64位) |
| F | float |
| D | double(64位) |
| L | Java对象 |
| [ | 数组类型 |

除基本类型外，对象、数组、字段、方法等都是通过直接引用字符串常量池来实现的，具体表示如下：

1. 对象以”Lpackage/name/ObjectName;”的形式表示， L表示这是一个Java对象，package/name是该对象所在的包名，ObjcetName表示对象名,如”Ljava/lang/String;”相当于java.lang.String。
2. 数组的表示是通过”[“来实现，多维数据就是多个”[“,如”[I”表示int[]、”[[I”表示int[][];对象数组的表示是通过”[”和”L”结合实现，如”[Ljava/lang/String;”表示String对象数组。
3. 字段的表示是通过类型结合类型来实现,由对象名、字段名、字段类型组成，其中字段名和字段类型用”:”隔开。如”Lct/ax$c;->a: Ljava/lang/String;”表示所属ct.a类、类型为java.lang.String、名字为a的一个字段。
4. 方法的表示比字段的表示更加复杂，因为Java可以支持静态多态，Dalvik虚拟机通过方法名、类型参数、返回值来详细描述一个方法，具体为” L对象;->方法名(参数序列;)返回类型;”。如”Ljava/util/Map;->put(Ljava/lang/Object; Ljava/lang/Object;)Ljava/lang/Object;”表示Map对象中的put方法，转换为对应的Java源码形式的格式为 Object java.util.Map.put(Object, Object)。

### 逆向提取敏感API

.dex文件是class文件(jar包)通过dx工具编译完成的，dx工具调用Smali汇编器将Java字节码转换为符合Smali语法的Dalvik字节码。由于字节码是编译好的二进制代码，不能直观展示和提取API，需要反汇编成为.smali文件。反汇编Dalvik字节码可以使用Baksmai工具实现，反编译后每个.class都对应一个.smali文件。

在.smali文件中，以4.3.1介绍的Dalvik指令的方式显示，可以直接提取需要的代码。提取API即是查找方法调用，因此只需要查找方法调用的指令即可。在Dalvik指令中，方法调用的基础指令是invoke，根据方法类型的不同，共有5条方法调用指令，详情见表4.5。

表4.5 Dalvik中方法调用指令

Table 4.5 Method call Instruction in Dalvik

|  |  |
| --- | --- |
| 指令 | 描述 |
| invoke-virtual | 调用实例的虚方法(最常用形式，动态绑定) |
| invoke-direct | 调用实例的直接方法(final/private) |
| invoke-static | 调用实例的静态方法(static) |
| invoke-super | 调用父类方法(super) |
| invoke-interface | 调用接口方法 |

Java代码可以通过混淆来来保护代码，但是有几个地方并不能混淆：使用反射、依赖系统接口、依赖配置文件。API都是系统接口，所以并不能被混淆。因此通过扫描invoke指令后面的方法来获取到我们需要的API。通过3.2.2的介绍可知，涉及到网络、通信、本地信息、系统安全、地理位置等行为的API，可以认定为敏感API。按照文献17和18给出统计结果，Android API中敏感API共计41个，表4.6列出了其中恶意软件最常用的几个。

表4.6 常见敏感API

Table 4.6 Common Sensitive API

|  |  |  |  |
| --- | --- | --- | --- |
| 包名 | 类名 | 方法 | 说明 |
| android.telephony | SmsManager | sendDataMessage | 发送彩信 |
| sendTextMessage | 发送短信 |
| TelephonyManager | getLine1Number | 获取手机号 |
| getCellLocation | 获取地理位置 |
| PhoneStateListener | onCallStateChanged | 监听手机状态 |
| android.content | BroadcastReceiver | abortBroadcast | 拦截短信 |
| ContentResolver | query update | 获取/修改其他应用共享的数据 |
| Intent | setAction | 设置事件 |
| setData | 设置对应数据 |

在.smali文件中，查找API调用时需要将函数的调用转换为对应的符合Dalvik字节码表示的指令形式，如java.lang.Class类中的forName方法的调用，转为的Invoke指令格式为”invoke-static {v0}, Ljava/lang/Class;->forName (Ljava/lang/String;)Ljava/lang/Class;”。

此外，通过Java反射机制可以跳过直接匹配的方法，因此需要增加对反射机制的检查。由2.3.2的介绍可知，Java反射机制一般使用Class.forName()方法来加载其他类，并使用该类对应的Class对象来调用其包含的API。根据Invoke指令的形式，寄存器{v0}表示局部变量当作Class.forName()的参数，向上查找对{v0}寄存器的赋值语句找到对应的类来判断是否使用了反射机制来调用敏感API。为了简单简化程序设计，如果对应的类中有敏感的API，则可以认为该文件有一次敏感API调用，该类所属的所有敏感API均增加一次计数。

## Android恶意应用分类检测

静态行为检测在提取待测软件的行为特征后，需要使用分类器进行类别的分类检测。本节介绍了概率神经网络和分类器集成，并使用集成概率神经网络Bagging\_PNN进程恶意应用的分类检测。

### 概率神经网络PNN介绍

获取到敏感API调用序列过后，需要进行分类检测，而分类检测需要分类器，本文选取概率神经网络(Probabilistic Neural Network，PNN)作为恶意应用检测的分类器。概率神经网络是Specht根据Bayes分类规则及Paren窗的概率密度函数提出的一种层次模型[19]，不仅具有神经网络固有的样本数据适应性强、学习能力好的优点，同时相比BP神经网络等，又具有训练迅速、参数调节少、分类能力强、网络结构清晰等有点[20]，广泛应用于模式识别和模式分类领域。在面对输入层参数数目更新时，PNN只需要相应调解对应的模式层单元，不用重建整个网络，可以很好的应对Android 系统不同版本中敏感API对应特征向量改变的情况。

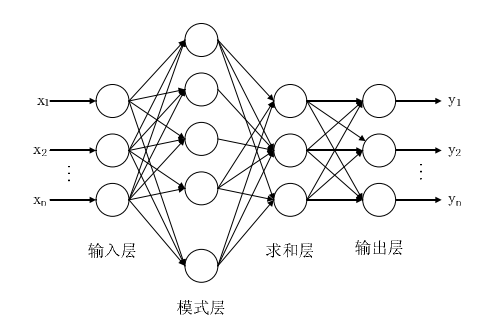


图4.6 概率神经网络结构示意图

Fig 4.6 The Structure of PNN

PNN是统计方法和前馈神经网络相结合的一种神经网络模型，由输入单元(输入XI,输出YI)、模式单元(输入XP=YI,输出YP)、求和单元(输入XT=YP(XP),输出YT)、输出单元(输入XE=YT，输出YE)共4层组成，基本结构如图4.6所示，下面对每个层次进行分别介绍。

1. 输入层将特征向量传递给网络，其神经元个数与样本特征向量维数相同，并不做任何计算。输入样本为：

,   
式中和 分别是输入层第*i*(*i=1,2,…,d，d*为样本向量的维数)个神经元的输入和输出。

1. 模式层计算输入特征向量与训练集中各个模式的匹配关系，其神经元数量和训练样本数量相同，第*i*类模式的第*j*个模式层神经元所确定的输入输出关系为：

( 4.1)

式中*i=1,2…,M，M*为样本分类类别数，*j=1，2…，*为第*i*类训练样本数;为高斯函数标准差，可以当作平滑因子使用，选择不同的可以得到不同的分类器，如时为线性分类器、时为临近分类器。

1. 求和层单元根据多元正态核函数的混合，估计出条件概率密度，其计算公式为：
2. 输出层，需要满足Bayes分类规则具有最小”期望风险”的优化决策规则，选择一个具有最大后验概率的求和层神经元作为输出，其计算公式为：

YE=arg max{}

### 分类器集成介绍

单个分类器的分类结果可能并不准确，因而需要多个分类器集成，最后采用投票的方式来确定分类结果。集成学习是将数据简单的、精度比较低的若分类器以某种方式组成在一起形成一个强分类器，从而提高分类准确率。在分类器集成方面Boosting和Bagging是两种主要的算法[21]。

Boosting算法主要思想是在每轮训练中，样本的提取是根据上一轮的错误率来实现的。每次训练过后，下一轮对训练失败的样例赋予较大的权重，加强对错误样例的学习。最终的预测函数对分类问题采用有权重的投票方式，因而具有较高的分类精度。但正是由于下一轮的训练需要依赖上一轮的结果，导致Boosting的训练必须串行进行而没有办法并发，在样本数量和纬度比较大的时候，耗时比较多。

Bagging算法主要是通过多次重复抽样并训练，形成多个假设，然后将多个假设组成最终的假设，用于具体的分类任务。Bagging要求分类算法“不稳定”，数据集小的变动对分类结果有显著影响，而神经网络算法是典型的不稳定算法[22]。它的理论基础是通过重复取样获取到不同的样本子集，使得在不同的样本子集上训练得到的个体分类器具有较高的泛化性能，不同的分类器具有较大的差异度。该方法通过从原始训练集中抽取若干采样来训练差异度模型，不同的模型之间不存在强的依赖关系，从而可以通过无差别投票来组成最终的分类模型。由于不同模型之间不具有其他关系，因而可以并发进行计算，通过投票选举方式使得分类的正确率也有比较好的保证。

为了提交Android恶意应用检测泛化能力和分类准确率，同时考虑到算法的运行效率问题，本文利用概率神经网络作为分类器，并采用Bagging学习方式来集成多个分类器，最后采用无差别投票方式组成最后的分类器来，并用它来实现恶意应用的检测,大体过程如图4.7所示。



图4.7集成概率神经网络示意图

Fig 4.7 Diagram of the Bagging\_PNN

为了训练分类器，需要有大量的已知状态的样本，包括正常应用和恶意应用。样本来源和3.3.2中类似的方法，为了保证正确率，需要扩大样本库容量，本文从VirusShare收集了7700个Android应用当作恶意应用样本，并从Google Play下载了5500个正常软件当作正常样本，然后分组对PNN进行训练，组成最后的Bagging-PNN分类器。

### 基于API调用的应用分类

综合前面介绍的内容，包括待测.dex文件的获取、敏感API的提取过程，本文的基于Bagging\_PNN的Android可疑应用分类检测大体分为两个阶段：

1. 训练阶段：对样本集进行预处理，解压APK文件，抽取.dex可执行文件；逆向反编译.dex文件为.smali文件，遍历.smali文件抽取敏感API,统计调用频率获取到PNN可接受输入数据，形成训练样本集对应的特征矩阵;进行可重复抽样，并分别训练不同的PNN模型；通过Bagging集成，使用投票选举最后的分类结果，并使用测试样本集进行测试。
2. 检测阶段：从可疑进程中获取到core dump；遍历抽取.odex文件，并反编译为smali文件；抽取敏感API,统计调用频率，形成待测应用的特征矩阵；使用训练阶段得到的Bagging\_PNN进行分类，得到最后的检测结果。

训练阶段主要用于分类器的构建，检测阶段才是完成待测应用的分类检测，基于API调用的应用分类检测模型如图4.8所示。



图4.8 基于API调用的Android可疑应用分类检测模型

Fig 4.8 The Suspicious Application Classification Model for Android based on API Call

## 本章小结

本章在第3章的可疑进程检测基础之上进行更深入的分类检测。首先详细的阐述了APK的逆向分析，包括.dex文件结构、Dalvik类加载机制，为后续的从可疑进程中提取.dex文件和逆向分析获取敏感API调用打下基础。其次介绍了如何从.dex文件中提取API调用并当作应用的静态行为特征，进行后续的分类检测。最后利用概率神经网络的鲁棒性强、学习能力好、训练简单等特点作为分类器进行可疑应用的鉴别，同时针对神经网络的不稳定新而造成的单一网络模型结果不是很准确的缺陷，使用了Bagging进行分类器的集成。最终形成了一个基于API调用的Android可疑应用的分类检测模型，对基于内存镜像分析得出的可疑进程进行深入检测，最终确定应用是为恶意应用，为以后更加深入的分析和查杀减少了分析范围，具有一定的应用意义。

# 实验验证与结果分析

第3章在分析Android系统内存镜像基础之上，介绍了通过查找隐藏进程的方法来检测系统级的恶意软件，及通过查找可疑进程的方法找到应用层面的恶意应用。第4章在第3章的基础上，从查找到的可疑进程中提取完整的应用静态行为，并使用Bagging\_PNN作为分类器来对应用的恶意倾向深入分析。本章通过在Android模拟器上运行多种常见具有隐藏进程功能的系统级恶意软件和已知的恶意应用并对其进行检测，来验证前两章提出的恶意软件检测方法。首先对实验环境进行了介绍，然后介绍内存镜像获取方法，对内存镜像分析过程进行介绍并找出隐藏进程和可疑进程，最后对检测出的可疑进程对应的Android应用进行恶意倾向检测。

## 实验环境搭建

进行实验之前需要搭建实验环境，主要包括四个部分：环境配置、Android源码下载、Linux\_Kernel下载和编译、实验数据配置。

1. 环境配置:

实验硬件环境为[Acer ATC705-N90](http://www.baidu.com/link?url=0lS-xeQrjA6ipZaIJ85L70szBflJ4wjdebJ7bQbI0vYk8sP7uHqiWF9xmB7vPF1HQVYBtYewzMFzMvN6uWQE6q)，8G内存，500G硬盘；实验软件环境为Ubuntu 14.04LTS，64位操作系统；安装JDK版本为JDK1.7, Android版本为Android4.4.2；Linux\_Kernel为[android-goldfish-3.4](https://android.googlesource.com/kernel/goldfish/+/android-goldfish-3.4)，对应的Linux内核版本为3.4。

1. Android源码获取和编译

Android源码使用了Git作为代码管理工具，Git工具是一个开源的分布式版本控制工具，用以有效、高速的处理项目版本管理。Google还开发了Repo工具对Git命令进行封装，用以下载、管理Android项目的软件仓库，下载Android源码首先需要下载Repo工具。

由于网络问题，如果直接从Android下载源码可能会失败，因此可以使用清华大学的TUNA镜像源下载Android源码，具体过程如图5.1所示。下载完成后直接make命令即可完成编译。

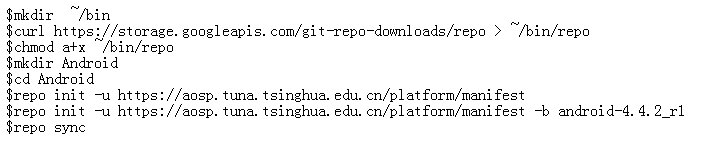


图5.1 Android源码下载过程

Fig 5.1 The Way of Android Source Download

1. Linux\_Kernel获取和编译

Android源码树上下载的Android源码并不包含内核代码，而是使用预先编译好的内核kernel\_qemu[23]。QEMU是一套能够模拟不同处理器的自由软件，可以虚拟出不同架构的虚拟机，如在x86的机器撒上模拟arm的机器。但Android源码自带的kernel\_qemu不支持LKM，不能动态加载LiMe取证工具，kallsyms机制在root用户下也是全0，因此实验过程中需要自己编译Linux内核模块。本文使用的Linux\_Kernel源码使用的是[android-goldfish-3.4](https://android.googlesource.com/kernel/goldfish/+/android-goldfish-3.4)，goldfish是一种虚拟的ARM处理器，可以在Android的仿真环境中使用[24]。

编译前需要配置.config文件，因为需要使用LiMe来获取内存镜像，而LiMe是一个基于LKM机制的，需要配置CONFIG\_MODULES，为了可以直接从模拟器中导出kallsyms文件，所以需要配置CONFIG\_KALLSYMS，配置项选择的详情如所示。

表5.1 内核编译关键配置项

Table 5.1 The Key Configuration for Kernel Compilation

|  |  |
| --- | --- |
| 配置项 | 说明 |
| CONFIG\_MODULES=y  CONFIG\_MODULE\_UNLOAD=y  CONFIG\_MODULE\_FORCE\_UNLO AD=y | 可以加载LKM模块insmod  可以卸载LKM模块rmmod  可以强制卸载LKM模块rmmod –f |
| CONFIG\_KALLSYMS=y  CONFIG\_KALLSYMS\_ALL=y  CONFIG\_KALLSYMS\_EXTRA\_PASS=y | 内核符号表包含全部函数  包含没有EXPORT\_SYMBOL导出的变量  可以直接导出/proc/kallsyms文件内容 |

内核编译完成过后，使用”emulator -kernel ./kernel/arch/arm/boot/zImage “启动基于该内核的Android模拟器。

1. 实验数据配置

验证隐藏进程和可疑进程检测方法需要在Android环境中运行恶意软件。

对于系统调用劫持进行进程隐藏的验证，本章使用文献25提供的MyKit工具，该工具通过劫持sys\_call\_table中关于文件显示显示的系统调用，将/proc文件系统下需要隐藏的进程对应的文件夹隐藏，从而使得ps、top等命令无法查看进程。对于直接修改内核对象的进程隐藏，本章使用文献26提供的linuxfu来实现，linuxfu通过修改指定进程的task\_struct结构体将其移除全局进程链表。

MyKit和linuxfu可以隐藏给定PID的进程。为了验证进程隐藏效果，编写两个简单的APK应用，为了保证其能够一直处于就绪队列，两个APK的功能相同：通过启动一个Service服务来不断计算圆周率PI。将应用下载到模拟器中并运行，进程号分别为869和2537，执行结果如图5.2所示。然后分别使用MyKit和linuxfu对其进行隐藏，效果如图5.3所示，可以看出通过ps命令已经找不到这两个进程了。需要注意的是，本文为了简单起见MyKit和linuxfu本身并没有隐藏，但是恶意软件为了防止被检测一般都会隐藏自己的。

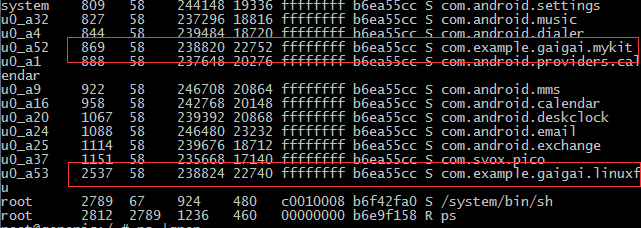


图5.2 进程未隐藏时的进程列表

Fig 5.2 The List of Processes when not are hidden.

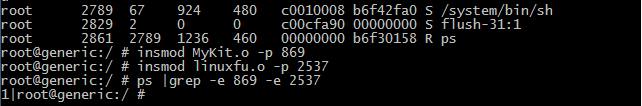


图5.3 查找隐藏后的进程结果

Fig 5.3 The Result of Finding Hidden Processes

对于具有恶意行为的Android应用，使用了具有各种典型行为的恶意软件，包括窃取用户隐私、私自发送短信、私自安装垃圾软件等，详情如表5.2所示。这些应用都需要安装进模拟器，并在内存镜像获取之时都需要处于运行状态，打开的端口、加载的链接库都在内存中存在。此外，为了验证对正常软件的检测，安装了知乎、UC浏览器等应用。

第四章提到，分类器的训练需要使用大量样本文件，本文首先一个共享病毒库VirusShare收集了7700个Android应用当作恶意应用样本，并从Google Play下载了5500个非游戏、安全、支付等应用(这些应用需要支付、网络、短信充值等功能，大量使用了敏感API，当作正常应用会对学习结果造成影响)当作正常样本。对可疑进程检测时，基于权限检测的样本集也从样本库里面随机抽取。恶意应用多余正常应用是因为恶意应用可能采取了混淆、加壳等抗静态检测的方法，导致没有办法提取敏感API，从而表现为正常应用，为了稀释这些应用的对训练结果的影响所以增加了恶意应用的比例。

表5.2 测试用Android恶意应用

Table 5.2 Android Malicious Applications for Testing

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 包名 | 主要恶意行为 | 运行时PID |
| Spyphone | com.sivartech.spyphone | 窃取用户隐私信息 | 2832 |
| 铅笔素描 | com.picture.sketch.free | 窃取用户隐私信息 | 2851 |
| 克里希纳 | com.hddevs.newkrishna | 内含推送广告插件,消耗流量 | 2901 |
| 云墨书院 | com.cloud.ebook | 私自发送短信订购扣费业务 | 3320 |
| 特效相机 | com.beautifulphoto.beta | 私自发送扣费短信 | 3327 |
| 美女壁纸 | com.livewallpaper.mingcheaa | 私自下载安装软件 | 3410 |
| 星座神算 | com.newhua.xingzuoshensuan | 伪造短信推送垃圾广告 | 3515 |

## 内存镜像获取

Android系统动态内存获取只要有两种方式：一种是针对特定的APP，通过Android SDK中DDMS工具获取相应的运行内存；一种是通过加载LiMe模块在系统中，获取到整个运行内存。LiMe[27]是一个可以从Linux设备中提取动态内存的KLM模块，在提取过程中LiMe可以将内核空间和用户空间的交互影响到最小，从而降低对已有内存数据的修改。本节使用LiMe工具进行Android内存镜像的获取，LiMe可以使用github上获取到。

获取LiMe源码过后需要对其进行编译，需要在Makefile文件中设置交叉编译模、指明Linux\_Kernel所在位置。在终端输入make命令进行编译，完成后在主目录下生成lime.ko文件。编译完成后即可进行Android内存镜像获取工作。提取动态内存文件有TCP传输和文件存储两种方式，由于TCP连接需要建立额外的网络连接，对网络中缓存的数据有一定的覆盖，一般使用文件存储的方式获取内存镜像文件。具体命令如表5.3所示，执行成功后则可以在~/Android/ram目录下看到当前运行的Android内存镜像文件ram.dump。

表5.3 LiMe获取内存镜像命令

Table 5.3 The Command for LiMe to Get Memory Mirroring

|  |  |
| --- | --- |
| 位置 | 命令 |
| Host | $adb push lime.ko /sdcard/dump/lime.ko  $adb shell |
| Target | #insmod /sdcard/lime.ko “path=/sdcard/dump/ram.dump format=lime” |
| Host | $adb pull /sdcard/dump/ram.dump ~/Android/ram/ |

## 隐藏进程检测

### 实验设计

按照3.3.2中介绍的基于进程列表对比的隐藏进程检测方法，获取到四种tasks，然后通过对比可以识别出内存镜像获取时所有处于就绪状态的所有隐藏进程和非DKOM类型的恶意软件，其中ps\_tasks和list\_tasks可以直接从模拟器使用命令得到。此外，分析Android内存镜像需要使用到内核符号表，而Android系统中并没有System.map文件的存在，需要从kallsyms文件中获取。编写一个shell脚本来完成该功能，执行过程如图5.4所示，其中list\_task包含两个文件list\_task1.txt代表只有进程的PID和list\_task2.txt代表包含了线程的PID。

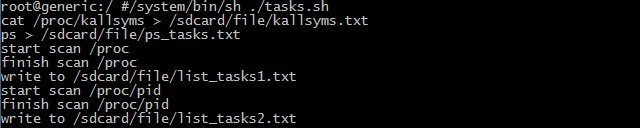


图5.4内核符号表获取过程

Fig 5.4 The Process of Getting System.map

对于全局进程链表队列all\_tasks，可以通过Volatility取证工具中的linux\_pslist插件来实现。Volatility是开源的Windows、Linux、Mac、Android的内存取证分析工具，由python编写而成，命令行操作，通过提供一系列的命令插件来完成各种分析工作，包含的Linux分析工具详情如图5.5所示。

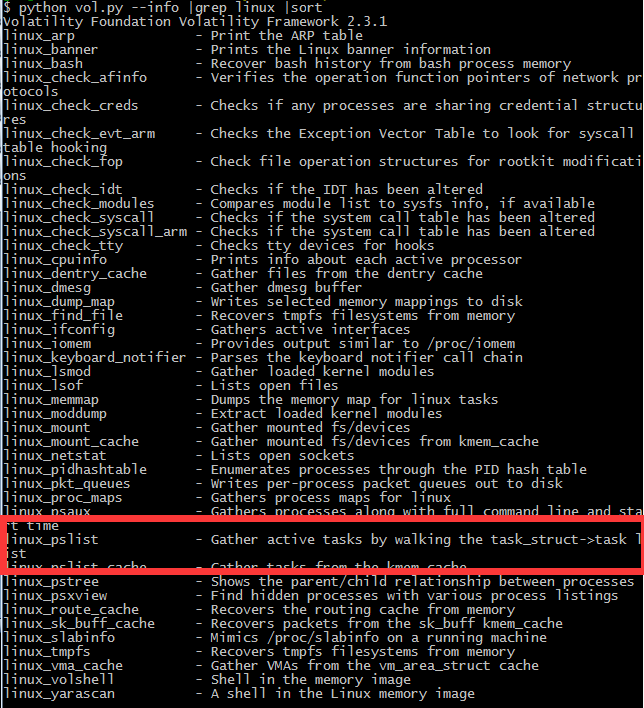


图5.5 Volatility工具包含的插件

Fig 5.5 The plug-ins of Volatility tools

由图5.5可知，linux\_pslist是通过遍历task\_struct来实现的，使用这个插件可以获取到所有的处于全局进程链表中的进程。通过实验发现其对线程进行了处理，没有显示线程的结果而只有进程。而Linux的就绪队列链表runqueue\_tasks是基于线程调度的，显示的均是线程的PID,所以需要对linux\_pslist进行修改，利用修改后的linux\_tasklist来完成all\_tasks文件的获取。

关于就绪队列runqueue\_tasks的获取比较复杂，现阶段还没有可以直接使用的工具。由3.2.2中runqueue的介绍可知，Linux在使用CFS调度器时，其进程就绪队列采用红黑树来组织，从实时进程和普通进程的两个红黑树根节点开始遍历可以获取到就绪队列中的所有就绪进程。通过分析android\_goldfish源码，得到进程就绪队列的获取方法如图5.6所示，可以编写一个Python脚本实现runqueue\_tasks的获取。



图5.6 runqueue\_tasks获取过程

Fig 5.6 The Process for getting Runqueue\_tasks

### 结果分析

对比5个文件的包含的进程号，结果如表5.2所示。

表5.4 隐藏进程检测结果

Table 5.4 The Result for Hidden Processes Detection

|  |  |
| --- | --- |
| 比较文件 | 对应PID |
| list\_tasks1 - ps\_tasks | 空 |
| all\_tasks - list\_tasks2 | 869,870,872,873,874,875,876,877,878  1089,1118,2851,3541,3552 |
| runqueue\_tasks - all\_tasks | 2537,2538, 2543,2544 |

从上表可疑看出，全局进程列表all\_tasks比从/proc虚拟文件系统中导出的进程要多得多,除了应该包含的869以外还包含了其他进程号，而就绪队列对比全局进程链表也不止2537这一个 PID。分析进程信息可知，除869、1089,1118, 2537,2851,3541,3552以外都是线程而非进程，这说明2537肯定是隐藏进程，符合3.3.2中提出的第三条规则。而1089,1118,2851, 3541, 3552对应的进程都是/system/bin/sh和flush-31,前者是Android系统中的shell执行器，后者是缓存刷新进程，均是系统必须包含的进程。这些进程产生的原因两者文件内容生成时间有一定的时间间隔，且从可知list\_task2的文件的获取是通过shell脚本来实现的，调用了/system/bin/sh程序。这说明除了869进程以外，其他都不应该是真正的隐藏进程。

以上实验表明，通过对比进程号列表可以找到非直接修改内核对象(DKOM)的其他隐藏进程(MyKit隐藏的进程)以及处于就绪状态中的所有隐藏进程(linuxfu隐藏的进程)，说明本文基于对比进程号列表的隐藏进程检测方式是可行的，具有一定的应用价值。当查找到隐藏进程的PID过后，根据其对应的task\_struct结构体就可以找到全部有关该进程的信息，进而获取到其对应的恶意软件。

## 可疑进程检测

可疑进程检测对象是Android应用，其对应的是Dalvik虚拟机进程。按照3.4.2介绍的方法，基于内存镜像的可疑进程检测主要是检测三个方面：网络端口检测、动态链接库检测、应用权限检测。

### 实验设计

网络端口检测可以通过Volatility工具包中的linux-netstat插件来实现，linux\_nestat的输出和netstat命令类似，包含协议、本地地址、目标地址等，如图5.7所示。

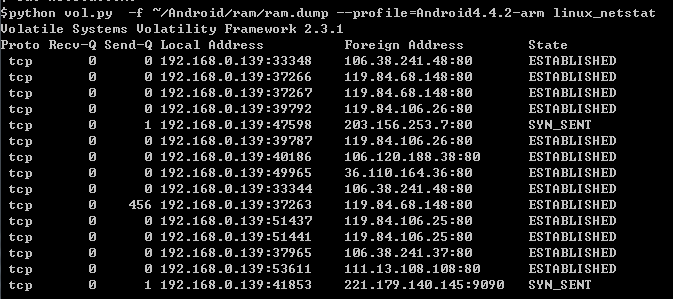


图5.7 linux-netstat输出结果

Fig 5.7 The Result of linux-netstat

图5.7中并没有包含对应的进程信息，找不到进程号和进程名。按照3.3.2的介绍，系统为每个打开网络链接的进程都分配了一个文件描述符，通过对比打开的网络对应的文件描述符和进程打开的所有文件信息，可以找到每个网络链接对应的进程。而列举打开的文件信息可以通过Volatility工具包中的linux\_lsof插件来实现，该工具可以展示所有打开的文件信息，使用-i参数查看端口运行情况进而找到端口对应的进程信息。如对端口33348查找对应的进程信息，执行linux\_lsof命令结果如所示,对应的是进程号为2832、进程名为com.sivartech.spyphone。

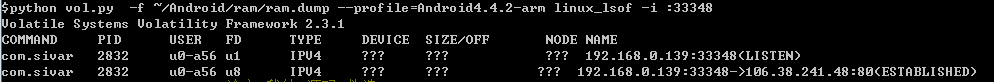


图5.8 指定端口号所对应的进程ID

Fig 5.8 The Specifies PID corresponding to the Port Number

动态链接库检测也可以使用Volatility工具包中的linux\_memmap插件，该插件可以是列出内存镜像中所有的内存映射文件，包含了虚拟内存地址、访问权限(可读、可读写、可执行)、链接的动态库名称等。由3.3.2的介绍可知，链接库检测可疑进程需要对比普通应用进程和zygote进程的maps的不同，在本实验的内存镜像中，zygote的进程号为58，其maps输出(只输出具有执行权限的链接库)如图5.9所示。

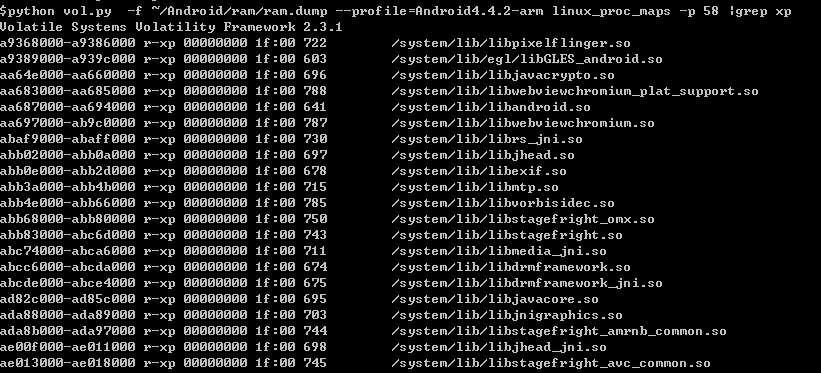


图5.9 指定进程对应的链接库信息

Fig 5.9 The Link Libraries for the Specifies Process

应用权限检测判断可疑应用采用的是文献17所介绍的方法：通过信息熵聚类算法把相关性高的权限聚合在一起形成一个权限簇，通过卡方检验计算计算不同权限簇在分类中权重影响因子，然后使用改进的朴素贝叶斯分类算来来实现对Android应用是否为恶意软件进行初步分类。改进的贝叶斯公式如下所示，其中 ,为单个权限对分类结果的卡方值。

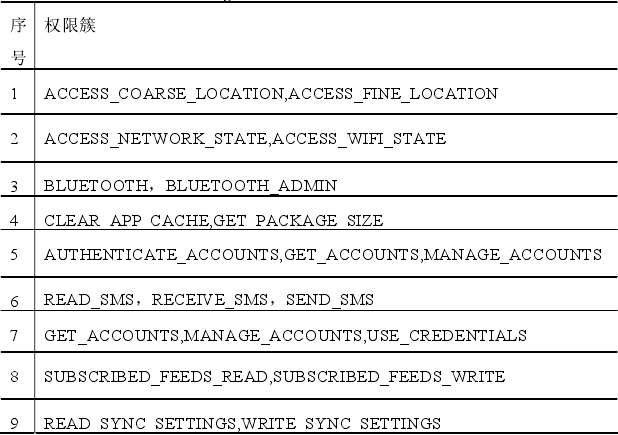
（5.1）

贝叶斯分类是通过先验概率和样本数据计算后验概率，需要计算先验概率和样本集。从前文下载的样本集中随机抽取700个恶意软件和500个正常软件，计算贝叶斯先验概率。样本集应用的权限信息都是在AndroidManifest.xml文件中提取，直接使用ZIP对APK文件解压就可以获取到AndroidManifest.xml文件。待检测样本可能没有对应的APK文件，按照3.4.2介绍可知权限信息可以从/data/system/packages.xml文件中获取。编写脚本分别从AndroidManifest.xml文件和packages.xml文件中提取对应应用的权限信息，并进行卡方计算、使用信息熵聚类算法将强相关性的权限聚类为权限簇、计算贝叶斯先验概率，最后进行可疑应用的检测，将检测出的可疑进程与ps\_tasks进行对比，找出应用对应的进程号。

通过卡方计算，将没有在恶意应用出现或经常同时出现在正常应用和恶意应用权限去除，权限特征从134个减少到68个。通过信息熵聚类算法，在相关距离阈值的时候得到9组权限簇，结果如所示。

表5.5 Android强相关权限簇

Table 5.5 Android High Relative Permission Clusters



### 结果分析

可疑进程的检测包含网络端口检测、动态链接库检测、基于贝叶斯的应用权限检测。由于只需要对Android应用的检测，因此可以排除系统进程的分析结果。分析内存镜像获取到打开的网络端口、进程链接的动态链接库，以及利用5.4.1学习出的贝叶斯分类模型对模拟器上已经安装的应用对应权限进行分类，得到的结果如表5.6所示。注意，如果三项都没有检测出来，则没有在表中出现。

表5.6 可疑进程检测结果

Table 5.6 The Result for Suspicious Process Detection

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| PID | 包名 | 网络端口 | 动态链接库 | 应用权限 | 可疑应用 |
| 1297 | com.baidu.input |  |  |  |  |
| 2832 | com.sivartech.spyphone |  |  |  |  |
| 2851 | com.picture.sketch.free |  |  |  |  |
| 2901 | com.hddevs.newkrishna |  |  |  |  |
| 3320 | com.cloud.ebook |  |  |  |  |
| 3327 | com.beautifulphoto.beta |  |  |  |  |
| 3410 | com.livewallpaper.mingcheaa |  |  |  |  |
| 3515 | com.newhua.xingzuoshensuan |  |  |  |  |
| 3740 | com.zhihu.android |  |  |  |  |
| 3744 | com.UCmobile |  |  |  |  |

从可以看出，恶意软件并不是一直都会打开网络链接，也不是所有的恶意软件都包含有动态链接库。对比表5.2和表5.6可以看出，基于应用权限的检测时，除了克里希纳(com.hddevs.newkrishna)没有被标记为恶意软件，其他都是分类正确，说明其检测结果还是有比较高的可信度。此外，安装的百度输入法也被检测出包含动态链接库，从3.4.2的介绍可知，动态链接库可以使用加壳、加密等方法来防静态检测，因此也可能是一个恶意软件，需要进行后续的检测。UC浏览器由于打开了网络链接，但是并没有使用链接库，基于权限的检测也没有被标记为恶意应用，综合所有选项并没有把它标记为恶意应用。这说明基于链接库、应用权限组合的可疑进程检测方法是可以行的。总共9个可疑进程检测出来后，需要进行更深入的检测。

## 恶意应用分类检测

恶意应用的检测是基于可疑进程的基础上的，找到可疑的进程后需要进行其他分析来判断其是否真的是恶意应用。根据第四章介绍的方法，基于敏感API的恶意应用检测分为三个部分：Bagging\_PNN学习、可疑应用对应dex文件提取、以及使用Bagging\_PNN进行分类检测判断是否为恶意应用。

### 集成概率神经网络学习

集成Bagging\_PNN学习过程需要使用大量样本，从提供的样本库中随机选取7000个恶意样本和5000正常样本。基于敏感API调用的恶意软件检测需要从应用样本中提取敏感API序列。使用Python编写一个脚本dexBaksmali.py来完成批量提取和反编译classes.dex文件，执行过程如所示。完成后会在指定的文件夹下生成一个smali文件夹，包含指定文件夹下所有APK反编译后的smali文件。反编译后每一个Android应用APK文件对应一个文件夹，随机进入一个文件夹，如所示。

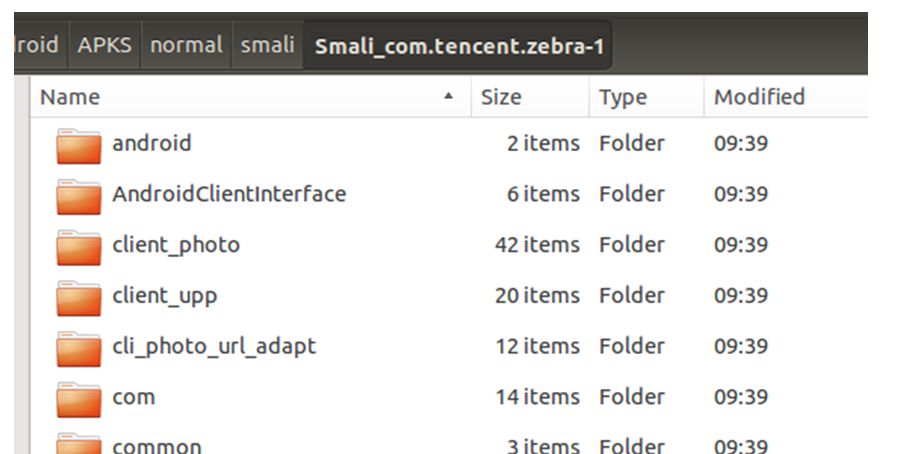


图5.10 应用样本逆向Smali文件集合示例

Fig 5.10 The Example of Application Sample Invers Smali Documents

该文件夹的子文件夹路径和APK对应的Java源码中包名是一致的，且是没有加壳的，能够正常反编译成功。按照4.3.2介绍的方法，遍历该文件夹的下的.smali文件，查找指定的敏感API对应的Dalvik指令码格式的字符串，以及使用Class.forName反射调用的敏感API对应的Dalvik字节码等，抽取出该应用对应的敏感API调用次数。使用一个Python脚本extractAPI.py来完成敏感API的抽取过程，并将结果写入到一个文件中，当作特征向量输入到PNN中。

将提到的敏感API调用序列当作输入向量，对概率神经网络进行学习训练。神经网络的创建可以使用Matlab软件完成。使用newpnn(P,T,SPREAD)函数可以快速创建一个概率神经网络，其中P为输入特征向量矩阵，T为输出向量(目标值)，Spread是径向基函数的扩展系数，默认为1，Spread值越大则输出结果越准确，但是计算困难更大耗时更多。通过多次实验，发现在spread的值为1.8的时候在本次实验机上的准确率和耗时有比较好的平衡。

单个PNN的结果在准确率上并不满意，为了增强PNN的泛化能力和分类准确率，本文对多个单独的PNN进行Bagging集成，形成多个独立的、通过投票选举的集成概率神经网络分类器。从7700个恶意样本和5500个正常样本之间进行10次可重复样本抽样，每次抽取100个，利用这些样本训练出10个不同的概率神经网络模型。由于本文是采用的Bagging算法来集成多个PNN，因此可以并行训练从而节省时间开销。

训练完成后，需要对集成Bagging\_PNN分类器进行验证，将剩下的700个恶意样本和500个正常样本分为5组,分别对其进行验证，验证结果如表5.7所示。

表5.7 Bagging\_PNN 测试结果

Table 5.7 The Result for Bagging\_PNN Testing

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 序号 | 恶意样本正确数 | 正常样本正确数 | 恶意样本正确率 | 错误样本正确率 | 正确率 |
| 1 | 124 | 86 | 88.57% | 86% | 87.5% |
| 2 | 126 | 87 | 90% | 87% | 88.75% |
| 3 | 122 | 89 | 87.14% | 89% | 88.92% |
| 4 | 121 | 88 | 86.43% | 88% | 87.08% |
| 5 | 127 | 90 | 90.71% | 90% | 90.47% |
| 总计 | 620 | 440 | 88.57% | 88% | 88.33% |

从表中数据可以看出，集成概率神经网络的分类正确率平均值达到了88%以上，可以当作分类器来对基于API调用次数的恶意软件进行分类。

### 可疑应用DEX文件获取

进程core dump文件的获取可以使用GDB来完成，GDB是一个由GUN开源组织发布的基于Unix/Linux操作系统下的程序调试工具，通过使用ptrace系统调用，可以使一个程序(追踪者)可以观察和控制另外一个程序的执行，并检查和修改被追踪者的内存及寄存器，主要用于断点调试和追踪系统调用。通过使用gcore命令，可以获取进程的core dump，里面包含了进程当前的执行内容，包含程序运行时内存、寄存器状态、堆栈指针以及内存管理信息等。



图5.11 core dump 文件获取过程

Fig 5.11 The Process of getting core dump files

将5.3.2中9个可疑进程的PID放入一个文件中，然后下载到模拟器中，启动编写好的脚本即可批量获取到所有进程的core dump文件，具体过程如图5.11所示，生成的core dump文件存放在/sdcard/file/corefiles目录下。

获取到core dump文件过后需要从中提取Dex文件，通过介绍的方法，可以从core dump文件中获取到.dex文件。编写一个Python脚本，把cored ump文件当作二进制流读入，按照图4.5指定的查找流程，查找.dex文件位置并截取载出当作.dex文件，执行过程和结果如图5.12所示。

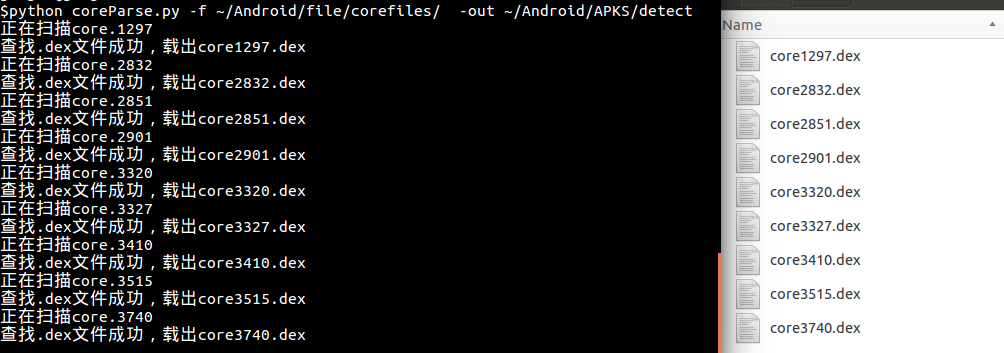


图5.12 .dex文件提取过程和结果

Fig 5.12 The Process and Result of getting DEX file from core dump files

### 恶意应用分类检测

从获取到的.dex文件中抽取敏感API序列，然后使用5.5.1训练得到的集成Bagging\_PNN分类器来完成对.dex文件是否为恶意应用的检测。需要注意一点，按照4.2.3介绍的方法从core dump文件中提取出的是odex文件。从4.2.2的介绍可知，odex文件相比dex文件有一些字节码的替换和头尾文件的添加，优化过程中依赖了Android系统中/system/framework环境。直接使用dexBaksmali.py文件进行反汇编会报错，需要将模拟器中对应的/system/framework的全部文件拷贝回来，并修改调用Baksmali工具包时的参数，修改后的调用方式为”java –jar baksmali.jar –x classes.odex –d framework –o ~/outpath” ,修改完成后即可进行批量的反编译odex文件为.smali文件。

获取到所有可疑进程的对应的.smali文件集合后，继续使用API提取脚本extractAPI.py来完成对API的提取，提取结果同样写入到一个文件中。将该文件导出成为一个二维矩阵，当作集成Bagging\_PNN的输入即可进行软件的分类检测。检测结果如所示，每一行是可疑进程对应的应用在某个PNN中的预测结果，最后一行是根据多数规则投票选举出的最终结果。

从表中可以看出，安装的8个恶意软件除了克里希纳(2901)没有被标记为恶意软件以外，其他软件都被正确检测出。其中Spyphone(2832)被在10个单独的分类器中都被标记为恶意软件，而Spyphone的功能就是泄漏用户隐私，可以监听用户的电话短信等信息、自动发送短信、自动打开网络链接等，对敏感权限的使用很多。克里希纳的恶意行为是在联网状态下推送广告，消耗流量，并不能自动更改网络状态，因而对敏感API的调用并不多。正常应用的百度输入法和知乎都没有被标记为恶意应用，说明它们的行为模式和恶意软件并不相同。

表5.8 待测应用检测结果

Table 5.8 The Result for Applications to be Tested

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 分组 | 1297 | 2832 | 2851 | 2901 | 3320 | 3327 | 3410 | 3515 | 3740 |
| 1 |  |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |  |  |  |
| 8 |  |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |  |
| 结果 |  |  |  |  |  |  |  |  |  |

基于集成神经网络的恶意应用检测中，恶意应用和正常应用能够正确区分，对后续更近一步的分析减少了样本范围。

## 本章小结

本章主要对前面两章提出的基于内存镜像分析的隐藏进程和可疑进程检测以及在可疑进程基础之上的Android恶意应用检测进行了实验。对隐藏进程检测实验上，表明从内存镜像中获取进程号列表并进行对比的检测方法可以识别出所有的非DKOM类型的恶意软件以及所有处于就绪队列的恶意软件，检测出系统级的隐藏进程后，可以通过进程描述符提取出对应软件的相关信息。通过网络端口、动态链接库以及Android应用权限检测出可疑进程，可以减少深入检测可疑应用的样本范围。在基于API调用的集成概率神经网络检测实验中，对正常应用和恶意应用的区分度比较高，缩小了后续的更进一步分析的样本范围，对恶意应用的更深入检测和查杀有一定的帮助。

# 总结与展望

## 本文工作总结

## 未来工作展望

# 致谢

# 参考文献

1. 何桂立. 2011年中国通信产业十大关键词点评文章(四) 智能手机[J]. 数据通信, 2011(1):9-10.
2. IDC.Smartphone OS Market Share [EB/OL]. [http://www.idc.com/prodserv/smartphone-os- market- share.jsp](http://www.idc.com/prodserv/smartphone-os-%20market-%20share.jsp)
3. Yi-Yang F U, Zhou D P. Android’s Security Mechanism Analysis[J]. Netinfo Security, 2011.
4. 符易阳, 周丹平. Android安全机制分析[C]// 第26次全国计算机安全学术交流会论文集. 2011:23-25.
5. 刘磊. Android安全体系的分析[D]. 广东工业大学, 2013.
6. 王亚军, 刘金刚. Linux 运用于嵌入式系统的技术分析[J]. 计算机应用研究, 2005, 22(5): 102-104.
7. de Goyeneche J M, De Sousa E A F. Loadable kernel modules[J]. IEEE software, 1999, 16(1): 65.
8. Surhone L M, Timpledon M T, Marseken S F. System.map[M]. Betascript Publishing, 2010.
9. de Melo A C. The new linux’perf’tools[C]//Slides from Linux Kongress. 2010.
10. 袁志坚, 王春平, 陈融,等. Android平台安全威胁及其应对策略[J]. 计算机技术与发展, 2013(9):110-113.
11. (美)洛夫(Love, R. ). Linux内核设计与实现 : 第3版[M]. 机械工业出版社, 2011.
12. (美)博韦, 西斯特, 陈莉君,等. 深入理解Linux内核[M]. 中国电力出版社, 2001.
13. 李国银. ARM的虚拟内存管理技术的研究[D]. 北京交通大学, 2013.
14. 刘伟, 孙其博. Android平台恶意软件行为模式研究[J]. 软件, 2012(11):150-156.
15. Hamandi K, Chehab A, Elhajj I H, et al. Android SMS Malware: Vulnerability and Mitigation[C]// Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on. IEEE, 2013:1004-1009.
16. Google.Mainfest permission [EB/OL]. http://developer.android.com/guide/topics/manifest/ permission-element.html
17. 姜正禄. Ubuntu系统下Android源码编译Compilation of the Android Source Codes in the Ubuntu System[J]. Software Engineering\s&\applications, 2015, 04:44-50.
18. 张锐. Android环境下恶意软件静态检测方法研究[D]. 重庆大学, 2014.
19. 文伟平, 梅瑞, 宁戈,等. Android恶意软件检测技术分析和应用研究[J]. 通信学报, 2014, 35(8):78-85.
20. Specht D F. Probabilistic neural networks and general regression neural networks [M]// Fuzzy logic and neural network handbook. McGraw-Hill, Inc., 1996:301-344.
21. 蔡曲林. 基于概率神经网络的模式识别[D]. 国防科学技术大学, 2005.
22. Maclin R, Opitz D. An Empirical Evaluation of Bagging and Boosting[C]// In Proceedings of the Fourteenth National Conference on Artificial Intelligence. 2010:546--551.
23. 蒋芸, 陈娜, 明利特,等. 基于Bagging的概率神经网络集成分类算法[J]. 计算机科学, 2013, 40(5):242-246.
24. Charan K V, Sharmila S P, Manjunath A S. Customizing AOSP for different embedded devices[C]// Computing for Sustainable Global Development (INDIACom), 2014 International Conference on. IEEE, 2014:259-264.
25. 罗升阳. Android系统源代码情景分析[M]. 电子工业出版社, 2012.
26. 平程, 蔡皖东. Android系统隐藏技术及检测方法[J]. 计算机技术与发展, 2014(5):142-145.
27. Anjikar, Milind G. Detection and Analysis of Rootkits on Android Smart Phones[J]. Dissertations & Theses - Gradworks, 2015.
28. Sylve J, Case A, Marziale L, et al. Acquisition and analysis of volatile memory from android devices[J]. Digital Investigation, 2012, 8(8):175-184.