计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.190100117

一种易部署的 Android 应用程序动态监测方案

苏 祥 胡建伟 崔艳鹏
西安电子科技大学网络与信息安全学院 西安 710071

摘 要 Android 应用程序动态监测方案通常有 3 种实现形式:1)定制 ROM 镜像;2)在获取设备 Root 权限的情况下,修改系统文件或者利用 ptrace 技术对目标进程注入代码;3)重打包 APK。这 3 种方式都是以侵入式方式实现,依赖于系统环境,难以部署到不同的设备上。针对上述问题,文中提出了一种基于插件化技术的非侵入式动态监测方案。该方案将监测系统以宿主 App 形式发布并安装到目标设备上;将待监测应用以插件形式加载到宿主 App 环境中运行,同时由宿主 App 加载相应的监控模块,完成对待监测 App 应用行为的动态监测。在待监测应用作为插件运行前,预先启动一个进程,通过动态代理方式对该进程中的 Binder 服务代理对象进行替换,将 Binder 服务请求重定向到虚拟服务进程中的虚拟服务进行处理,从而使待监测应用中的四大组件能在预先启动的进程中运行。然后,在待监测应用 Application 的初始化过程中加载 Java 层和 Native 层监控模块,完成监控。根据该思想,在 VirtualApp 沙箱基础上实现了原型系统 AndroidMonitor,并在 Nexus5 设备上对其进行测试。实验结果表明,与其他方案相比,该方案虽然会使待监测应用的启动时间增加 1.4 s 左右,但不需要获取设备系统 Root 权限,能够同时对 Java 层和 Native 层的敏感 API 进行监控;同时,引入了设备信息防护模块,以防止 App 监控过程中设备信息发生泄露。系统以 App 形式发布,容易部署到不同设备上,同时适应多种应用场景。

关键词:动态监测;插件化;挂钩子;动态代理;沙箱;非 Root
中图法分类号 TP311.5

Easy-to-deploy Dynamic Monitoring Scheme for Android Applications

SU Xiang, HU Jian-wei and CUI Yan-peng
School of Cyber Engineering, Xidian University, Xi'an 710071, China

Abstract Android application dynamic monitoring scheme is usually implemented in three ways: 1) custom ROM; 2) after obtaining the device root permission, modify the system file or use ptrace technology to inject code into the target process; 3) repackage APK to add monitoring code. All three methods are implemented in an intrusive manner, which depends on the system environment and is difficult to deploy to different devices. In order to solve the above problems, a non-intrusive dynamic monitoring scheme based on plug-in technology was proposed. The scheme releases the monitoring system in the form of host App and installs it on the target device. The application to be monitored is loaded by host App environment in the form of a plug-in for operation, and the host App loads the corresponding monitoring module when loading the plug-in, so the App is monited. Start a process ahead of time before the application to be monitored runs as a plugin. The Binder proxy object in the process is replaced by a dynamic proxy method, and the Binder service request in the process is redirected to the virtual service in the virtual service process for processing, so that the components in the application to be monitored can run in the pre-started process. When the Application object in the application to be monitored is initialized, the Java layer and the Native layer monitoring module are loaded to complete the monitoring. According to this scheme, the prototype system AndroidMonitor is implemented on the VirtualApp sandbox and tested on the Nexus5 device. The experimental results show that compared with other schemes, although the startup time of the application to be monitored is increased by about 1.4 s, the scheme does not need to acquire the root authority of the device system, and can simultaneously monitor the Java layer and the native layer sensitive API. The system introduces a device information protection module to prevent device information from leaking when monitoring applications. The system is distributed in the form of an app, which is easy to deploy to different devices and has multiple application scenarios.

Keywords Dynamic monitoring, Plug-in, Hook, Dynamic proxy, Sandbox, Non-root

1 引言

动态监测能够有效地分析安卓应用行为。本文对已有动态监测工具进行了调研,将其分为 3 种实现方式:1)定制 ROM 镜像,对安卓运行时或系统 Framework 层框架进行修改[1];2)Root 手机,在获取系统最高权限的情况下对系统文件进行修改[2],或者使用 ptrace 进程注入技术对待监测应用进程注入监控代码;3)重打包待测 APK,在目标 APK 中插入监控代码[3]。这 3 种方式均是以侵入式方式实现的,以修改 Android 系统或 APK 文件为前提。侵入式的动态监测方案存在以下弊端:1)定制 ROM 镜像的监测系统往往依赖于具体系统版本甚至具体机型,不易部署,所以以模拟器镜像发

到稿日期:2019-01-15 返修日期:2019-04-15 本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
通信作者:苏祥(suxiang234@163.com)

布，导致模拟镜像运行依赖于具体主机的 SDK 环境，在模拟器中运行应用也存在兼容性问题；2）以获取 Root 权限为前提的动态监测方法，随着手机系统安全性的提高而难以获取系统 Root 权限，不能通过修改系统文件或进程注入方式完成部署；3）重打包监控 App 技术在 Android 加固环境下难以实现，很多 App 都设计有防止重打包模块来禁止重打包。

本文提出了一种基于 Android 插件化技术的易部署动态监测方案。在该方案中，监测系统主体为具有插件化功能的宿主 App，监测模块位于宿主 App 中。监测 App 时，宿主 App 将待监测 App 以插件形式加载到宿主环境中运行；同时，宿主 App 将监控模块加载到插件 App 运行的进程中，从而完成对待监测 App 运行时行为的监测。该系统以 App 形式发布，对设备的兼容性由宿主 App 完成，因此容易部署到不同设备上。基于此，本文在插件化框架 VirtualApp 基础上实现了一个监测 App 运行时行为的应用 AndroidMonitor，其能够同时对应用 Java API 和 Native API 进行监控，相比其他动态监测工具具有易部署的特点。

2　相关研究

Android 应用程序行为检测主要分为静态检测和动态检测两种。静态检测通过提取 APK 压缩包中的字节码和 AndroidManifest. xml 等文件进行分析，不需要安装运行待检测 APK 文件。而动态检测则需要监控程序运行过程中的行为、数据流向，并以此分析 App。

Flowdroid[4] 是静态分析领域最具代表性的工具，其核心思想为静态污点分析。首先，它将待测 APK 反编译后，解析其中的 DEX 文件、AndroidManifest. xml 等相关文件，获取其 Source，Sink 以及入口点，得到相应的生命周期和回调函数列表；然后，根据生命周期和回调函数列表生成 dummyMainMethod()函数，并将其作为虚拟的 main 函数来模拟组件的生命周期执行，生成程序调用图（CG）和过程间控制流程图（ICFG）。通过分析 ICFG，追踪从 source 到 sink 的数据流，获得敏感信息流的泄露路径。

动态分析领域也取得了大量研究成果。Taintdroid[5] 是一款基于 Dalvik 虚拟机的动态污点分析系统，它将隐私数据标记成污点数据，如果污点数据在程序运行过程中被截取、拼装、加密、传递，那么新生产的数据也会被污染。在污点泄露处对数据进行检测，如果是污点数据，则表示数据发生了泄露。该系统只能跟踪信息流，而不能跟踪控制流和第三方动态库。CopperDroid[6] 通过分析并直接检测 App 执行过程中的系统调用来重构出 App 的恶意应为，能够抵抗应用层代码的混淆并判断系统行为和进程行为，最终产生详细的有语义的行为信息。DroidInjecter[7] 是一款基于 ptrace 进程注入实现的 Android 应用运行行为动态监测工具。它将具有 Hook 功能的共享库注入到目标进程中，通过 Hook 系统 API，统计系统 API 调用频率，对 App 行为进行评估。DroidMiner[8] 提出利用 Android 系统提供的 API 进行非 Root 权限下敏感 API 的监控，但该方式依赖于系统，只能对部分敏感 API 进行监控。

3　相关背景

Android 系统是基于 linux 内核开发的。虽然 linux 已经提供了管道、Socket 等 IPC 机制，但考虑到数据传输效率和安全性，Android 系统采用了新的进程间通信机制 Binder。Binder 是典型的 Client-Server 通信模式，由服务端向客户端提供服务。Binder 框架在设计中定义了 4 个对象：Client、Server、ServiceManager、Binder 驱动[9]。其中，Binder 驱动工作在内核层；Client，Server，ServiceManager 运行在用户层。用户层的 3 个对象在不同进程进行 IPC 通信时，均通过内核层的 Binder 驱动传递数据。Binder 服务启动后，需要将自身服务的 Binder 引用注册到 ServiceManager 中，而 ServiceManager 内部维护着一个 Binder 引用列表；客户端使用 Binder 服务时，向 ServiceManager 查询对应的 Binder 引用，得到对应的 Binder 引用对象后，生成代理服务对象。客户端在与服务端进行通信时，均由代理服务对象将数据封装成 Parcel 对象并通过 Binder 驱动传递。

Binder 通信框架中，Binder 驱动、直接与驱动交互的 Binder 代理对象及 Binder 服务对象都由 C/C++开发，它们运行在 Native 层，而 Framework 层则以 Java 对象的方式为用户提供服务。考虑到编程的简洁性和维护性，Android 通过 JNI 的方式在 Java 层给 Native 层的 Binder 系统做了一个映射，每个 Java 层的 Binder 对象都持有 Native 层对应 Binder 对象的引用，数据传输时亦是 Java 层流经 Native 层到达驱动，再由驱动流经 Native 层到达 Java 层。Binder 服务在 Java 上以 AIDL 的形式实现。通过 AIDL 文件定义服务接口，会生成相应的 Stub 类和 Proxy 类，服务端只须实现 Stub 类中的服务即可。客户端在获取另一进程的服务时，会得到 Binder 代理对象，由 Binder 代理与服务通信。

4　系统设计

4.1　系统原理及总体架构

侵入式的 Android 应用行为动态监测方案是以修改系统文件或者获取手机设备 Root 权限为前提，来完成监测环境的部署，但难以部署到不同设备上。以 Xposed 框架为例，为了安装 Xposed 框架环境，必须获取设备的 Root 权限，才能替换系统 app_process 文件[10]。

在 Android 应用开发发展中，插件化技术[11] 的出现使模块化编程成为可能。四大组件被打包在独立插件模块中，由宿主 App 加载运行插件中的组件。如果将完整的待监测 App 作为一个插件来进行安装和运行，那么在宿主 App 加载运行待监测 App 前，可以添加一个窗口，用于加载监测模块，这样就能监控待监测 App。而窗口中加载监控模块的逻辑都是宿主 App 完成的，并不需要对 Android 系统和 App 做出修改。面对机型差异时，只需要在宿主 App 适配，该种监测方式就具有非侵入、易部署特点。

根据上述基本原理，本文在插件化沙箱环境 VirtualApp 的基础上实现了 AndroidMonitor 动态监测系统，如图 1 所示。

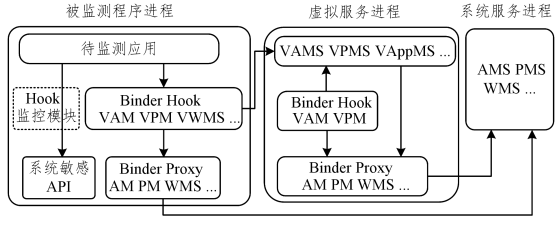

图 1　AndroidMonitor 动态监测系统架构

Fig. 1　Architecture of AndroidMonitor dynamic monitoring system

正常情况下,一个 App 进程与系统服务进程通信时,是通过 Binder 代理对象进行的。为了使待监测 App 中的组件能够在事先启动的进程中运行,VirtualApp 仿照系统服务进程构造了一个虚拟服务进程,对系统服务进程 Framework 层中的诸多 Binder 服务对象如 AMS(Activity ManagerService)和 PMS(PackageManager Service)进行模拟,构造出虚拟的 VPMS(VPackageManager Service)等 Binder 服务对象。虚拟的 Binder 服务对象和系统的 Binder 服务对象具有几乎一样的服务方法,但逻辑不完全相同,它们只是模拟了系统的 Binder 服务功能。一个正常 App 进程内部缓存有一部分系统服务的 Binder 代理对象,在 App 运行过程中,组件是通过这些代理对象与系统服务进程通信的。为了使组件能够插件化运行,VirtualApp 以动态代理方式对待监测进程中的 Binder 代理对象进行 Hook,将缓存 Binder 代理对象替换成自定义的动态代理对象,使得待监测进程访问系统服务时,自定义的动态代理对象会将 Binder 服务请求重定向到虚拟服务进程中。虚拟服务进程作为一个中间人,模拟系统服务处理 Binder 服务请求,或者将服务请求中的上下文信息替换成宿主 App 中的上下信息发送到系统服务进程处理。在这种方式下,对待监测 App 中的四大组件进行插件化,使其能够在事先启动的进程中运行。而 AndroidMonitor 监测系统,在事先启动的进程启动后而待监测 App 组件启动前这个窗口时机,加载自定义的监控模块,从而对 App 进行监测。

4.2 初始化插件运行环境

插件运行环境的主体为宿主 App,具有将待监测应用以插件形式加载到自身运行环境中运行的能力,从而在待监测应用启动时将监测模块加载到待监测应用运行的进程中。该宿主 App 的实现依赖于 Android 插件化开发技术,本文实现的 Android 应用行为动态监测系统 AndroidMonitor 的插件运行环境由插件化沙箱框架 VirtualApp 提供。

4.2.1 插件进程环境的初始化

AndroidMonitor 监测系统的插件功能主要由 VirtualApp 提供。VirtualApp 运行时,会存在 4 种进程:宿主进程、虚拟服务进程、客户端进程、子进程[12]。

(1)宿主进程:用于管理客户端 App 的安装、启动和卸载。

(2)虚拟服务进程:模仿系统 Framework 层的服务功能创建一套虚拟的 Binder 服务,其中服务命名与系统服务命名类似,如系统 Framework 层的 AMS 对应虚拟服务进程的 VAMS。

(3)客户端进程:插件 App 运行的进程,也就是待监测 App 运行的进程。

(4)子进程:宿主进程的子进程。

由于存在多个进程,因此每个进程中的 Application 都会调用 VirtualCore 类的 startup 方法初始化一次。VirtualCore 使用 InvocationStubManager 类的 injectAll 方法,根据当前进程类型,以动态代理的方式进行 Hook,用对应的 Stub 类替换相应的类。Application 不对宿主进程进行 Hook,对虚拟服务进程只 Hook AM(ActivityManager)和 PM(PackagerManager)两个代理对象;对客户端进程,即待监测进程,需要 Hook,整个 Framework 层的 Binder 服务代理对象和其他相关对象。而对 Binder 代理的 Hook 会将当前进程的 Binder 服务请求重定向到虚拟服务进程中的虚拟服务中。

4.2.2 待监测应用的安装

由于待监测 App 作为客户端 App 运行,因此其在运行前必须由宿主 App 进行一次安装。安装并不是由系统 PMS 进行的,而是由虚拟服务进程中的虚拟服务 VAppManagerService 进行的,该服务模拟了 PMS 安装 App 的流程。

(1)通过 PackageParser 解析待监测 APK 中的 AndroidManifest.xml 文件,生成 VPackage 对象来储存解析过程中生成的组件信息和权限信息。VPackage 对象是可以序列化的,它最终会被序列化到私有安装目录下的 package.ini 文件中。

(2)为安装的待检测 App 分配私有安装目录。该安装目录并非是系统的安装目录,而是在宿主 App 的 Data 目录下以安装应用包名创建的一个私有安装目录。私有安装目录内部的目录结构与系统安装目录内部的目录结构类似,但多了一些安装应用相关信息的序列化存储文件,如 package.ini 等。

(3)将 APK 文件及其相关文件复制到私有安装目录下,如将 so 文件复制到对应的 lib 目录下。

(4)创建 PackageSetting 对象,在 PackageSetting 对象中存储待监测应用的安装信息,如安装目录、分配用户 ID 等,将 PackageSetting 对象保存到 VPackage 对象中。

(5)向虚拟服务进程内定义的广播系统注册待监测 APK 的静态广播。

4.2.3 待监测应用的启动

一个 Android 应用通常由四大组件组成。若该应用仍未启动,访问任意一个组件提供的功能都会导致该应用进程的创建启动,这个过程由 AMS 控制。AMS 内部维持一个 ProcessRecord 对象列表,ProcessRecord 对象储存了对应进程的相关信息。当组件启动时,AMS 会查找该组件所属进程对应的 ProcessRecord 对象是否存在于列表中,若不存在,则通知 Zygote 进程 fork 一个进程,并将进程对应的 ProcessRecord 对象保存在列表中,然后在该进程中创建并启动该组件。

在 AndroidMonitor 中,虚拟服务进程仿照系统服务实现了诸多虚拟服务,其中 VAMS 服务仿照了系统的 AMS。VAMS 内部有 ActivityStack,ActivityRecorder 等数据结构,模拟实现了对待监测进程中 Activity 的栈管理。当待监测 App 启动时,VAMS 并没有像原生 AMS 那样通过 Socket 通信告知 Zygote 去创建一个进程,而是在应用层通过一个 StubContentProvider 组件的启动来创建待监测应用进程。在 StubContentProvider 组件启动过程中,会创建一个 VClientImpl Binder 服务对象,并将该服务的 Binder 引用返回给 VAMS 保存。VAMS 可通过 VClientImpl 与待监测应用通信,执行后续的进程初始化工作。

正常应用启动过程中,在创建当前应用进程后,系统会调用当前进程 ActivityThread 类的 handleBindApplication 方法,完成 Application 对象的初始化。在实例化当前进程的 Activity 或 Service 组件前,系统会再次检测当前进程的 Application 是否初始化,若没有则进行初始化。

由 StubContentProvider 组件启动创建的待监测进程启动后,待监测进程的 Application 仍是原来的 Application,其完成了对当前进程中各种 Binder 服务代理对象和其他关键对象的 Hook 替换,使 Binder 服务请求重定向到虚拟服务进程中。在启动插件化的 Activity 或 Service 组件前,VirtualApp 会调用 VClientImpl 类的 bindApplicationNoCheck 函数

方法。该方法模拟了 ActivityThread 类的 handleBindApplication 方法，创建并初始化待监测应用中的 Application 对象，并且对待监测应用的运行环境进行了配置。

待监测进程的启动会调用 bindApplicationNoCheck 函数进行待监测应用的 Application 切换，这时插件化的四大组件尚未启动执行，这相当于提供了一个窗口，AndroidMonitor 在 bindApplicationNoCheck 函数的窗口处添加监控模块，完成对当前进程的插桩，从而对当前进程进行监控。

4.3　监控行为的选取

4.3.1　系统敏感行为

Android 应用行为监测系统需要对系统函数进行插桩才能完成监测，这里列出部分常见系统敏感函数。

(1)网络流量。Android 应用通常会把本地数据上传到服务器，我们可以追踪 URL 类的构造方法来获取 URL 链接，追踪 HttpUrlConection 类的相关方法和域对象来获取网络链接和流量信息。如果使用第三方网络链接库，如 Okhttp3，就需要有针性地 Hook 第三方网络链接库中的函数。

(2)文件系统。Android 在 Java 层主要使用 File 类进行文件操作，因此可考虑 Hook File 类相关构造函数来监控文件对象。ContextImpl 类的 openFileOutput 函数也有文件读写功能。另外，由于 Android 支持 NDK 编程，因此也有可能通过 fopen 等 C/C++函数进行读写操作。

(3)数据库存储。Android 使用 SQLite 数据库进行数据存储操作，并提供了 SQLiteOpenHelper 工具类来管理数据库。SQLiteOpenHelper 的 getWritableDataBase 会创建数据库。而数据类 SQLiteDataBase 用于描述创建的数据库对象，它的 execSQL 方法可以执行 SQL 语句。SQLiteDataBase 类也支持 insert，update，delete，query 等操作，这些都可以作为监控数据库存储的监控点。如果待监测应用使用了具有加密功能的第三方 SQLCipher 数据库，就需要有针对性地 Hook 第三方库中的相关函数。

(4)组件间通信和进程间通信。Android 组件间通信主要依赖于 Binder 机制。Intent 对象作为媒介，携带数据从一个组件传递到另一个组件，可以通过 Intent 对象的 putXXX 方法设置不同类型的数据以及 getXXX 方法获取数据。Activity 对象使用 startActivity 从一个 Activity 切换到另一个 Activity。Activity 启动服务时，使用 startService/bindService 方法；Activity 发送广播时，会使用 sendBroadcast 方法。这些方法会将 Intent 对象作为参数传递到另一个组件中，所以组件的这些方法都可以作为组件间通信的监控点。应用程序在使用 Binder 服务时，会打开 Binder 驱动文件/dev/binder，使用 IO 控制函数 ioctl 与驱动文件进行交互，因此可以考虑 Hook libBinder.so 或 libc.so 共享库中的 ioctl 函数[13]来监控 Binder 间的进程通信。

此外，SharedPreferences、动态库加载、多进程、加解密函数、资源加载、序列化等都是 Android 应用运行时应该监控的关注点。而 AndroidMonitor 有针对性地对其中的 Java 层和 Native 层敏感 API 进行了 Hook。

4.3.2　待监测应用中的敏感行为

对于待监测应用中的 API，只需要 Hook 其中的关键 API 即可。为了选取待监测 App 中的敏感 API，需要对待监测 App 进行逆向分析。对于 DEX 格式文件，通常使用 JEB 工具进行逆向分析即可获取相关敏感 API。对于加固的 DEX 文件，使用 Android SDK 中的性能分析软件 TraceView，记录待监测应用运行过程中的函数调用流程关系图进行分析，获取敏感 API。对于待监测应用中的 so 共享库文件，使用逆向分析工具 IDA 进行逆向分析，获取 Native 层的敏感 API。

4.4　监控模块的设计

这里加载的监控模块的主体为 Hook 模块，主要功能为对当前插桩函数的信息进行日志输出。当待监测应用调用被监测函数时，我们能够从 log 日志中清晰看到被调用函数的名称、参数信息、调用时间。

4.4.1　加载时机

分析 4.2.3 节中指出的 VClientImpl 类的 bindApplicationNoCheck 函数方法后，发现该函数运行在待监测进程内，内部会创建待监测 App 中的 Application 对象并替换当前进程中的 Application，回调创建的 Application 对象中的 onCreate 方法等操作。图 2 给出了部分关键流程，待监测 App 启动过程中有 3 个重要的窗口。执行至窗口 1 时，待监测 App 中的 Application 类尚未初始化，由于 Application 是 App 启动过程中第一个加载的类，在初始化 Application 类之前，应提前完成系统函数的插桩，因此在窗口 1 处，AndroidMonitor 会加载系统函数插桩模块；在窗口 2 处完成了 Application 的创建，创建使用的是 LoadApk 的 newApplication 方法，创建过程中会回调 Application 的 attachBaseContext 方法；窗口 3 处会回调 Application 的 onCreate 方法。执行至窗口 2 或窗口 3 处，由于 Application 已创建，此时待监测 App 中的类已经被某个类加载器所加载，因此窗口 2 或窗口 3 适合加载对待监测 App 中敏感 API 进行监控的模块。

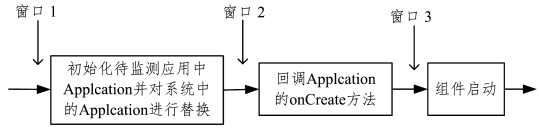

图 2　窗口图

Fig. 2　Window diagram

4.4.2　Hook 模块的设计

对 4.3 节的监控行为进行分析后，可以把监控的敏感 API 归为 3 类：系统 Binder 服务类 API，Java API，Native API。其他动态监测工具都是单一对 Java 层或 Native 层进行监控。而本文实现的 AndroidMonitor 动态监测系统可以同时对 Java 层和 Native 层进行监控，并针对这 3 类方法，分别采用不同的 Hook 方法。

(1)Binder Hook

在构造插件环境时，待监测进程启动过程中，已经通过动态代理的方式对 AMS，PMS 等 Binder 代理对象进行了 Hook。如果监控的敏感 API 所属对象为已经 Hook 的 Binder 代理对象，则通过动态代理的方式直接进行 Hook，这样能提高效率。如果代理对象尚未被 Hook，则把敏感 API 视为普通 Java 类。

(2)Java Hook

Android 系统从 5.0 版本开始，正式使用 ART 虚拟机作为安卓运行时环境。而 ART 虚拟机会在安装 Android 应用时，将 Dalvik 字节码翻译成本地机器指令存储在 OAT 格式

文件中。在执行过程中，每个 Java 方法对应一个 ArtMethod 对象。ART 虚拟机加载 Class 类时，会将 Java 方法对应机器指令的起始地址设置到对应 ArtMethod 对象的 compiled_code_entry_point 域。Java 方法执行时，会根据 compiled_code_entry_point 的地址，跳转执行对应的机器指令，从而加快方法的执行。与 ART 模式不同，Dalvik 模式下的 Hook 方法在 ART 模式上已经完全无法使用。WiBfeld 提出了一种 callee-side method hook injection 技术[14]，实现了在 ART 虚拟机上 Hook。其核心思想是在 Java 方法运行时，通过 ArtMethod 对象的 compiled_code_entry_point 域获取被 Hook 方法和目标方法对应的 Native 代码的地址，并对被 Hook 方法的 Native 代码进行备份；然后，创建一个有跳转指令的 ShellCode，其中的跳转指令会使 PC 指针跳转到目标方法的 Native 代码执行；最后，将 ShellCode 写入被 Hook 方法的 Native 代码首地址处，从而改变被 Hook 方法的执行流程，达到执行目标方法的目的。为了在执行目标方法后再执行原方法，需要在目标方法中通过反射去回调备份的原方法。基于该思想，epic 框架[15]较好地实现了 ART 模式下的 Hook。出于性能考虑，本文采用 epic 框架来完成对 Java 方法的 Hook。

对 Java 方法 Hook 时，需要找到方法所属的 Class 类对象。AndroidMonitor 中对于 Android 系统类 Hook 的 Java 监控模块加载时机在窗口 1，由于系统类对应的 DEX 文件在 ART 虚拟机启动时就被加载到了内存中，并以 DexFile 类形式保存在 ClassLinker 中，BootClassLoader 是能准确加载系统类的，因此 AndroidMonitor 可以直接使用 Android 系统的 Class 类并对其中的方法进行 Hook。而对于待监测 App 中的类，系统类加载器是无法正确加载的。在 Android 系统中，一个加载到内存的 App 文件通常用 LoadApk 对象表示，通过调用 LoadApk 的 getClassLoader 即可生成 App 对应的类加载器。待监测 App 中的 Application 类由 LoadApk 生成 ClassLoader 完成加载。因此，在窗口 2 或窗口 3 处，AndroidMonitor 通过调用待监测 App 的 Application 对象的 getClassLoader 方法来获取 App 的类加载器，然后通过类加载器的 loadClass 方法加载待监测类以得到对应的 Class 类对象，完成其中方法的 Hook。

在设计 Java Hook 模块时，为了降低代码的耦合度，采用了命令模式。将 4.3 节中的 Java 层监控点分成若干类后，每一个分类对应一个 HookItem 类，在 HookItem 类内完成该分类的所有 Hook，而 HookItemManager 类管理维护所有的 HookItem 类。从效率出发，在编译生成 AndroidMonitor.apk 时，让 HookItemManager 只加载感兴趣的 HookItem 即可。

(3)Native Hook

在 Android 平台，对系统共享库或 App 中的共享库的 Hook 通常需要获取 Root 权限，然后通过进程注入技术将 so 库注入到目标进程完成 Hook。本文提出的基于插件化沙箱的动态监测方法，在待监测应用组件启动前，就已经启动了待监测进程，在不同窗口时机加载相应的 Native 监控模块，并不需要 Root 权限和进程注入，具有易部署的特点。在 Hook Native API 方法时，根据不同的 Hook 目标，采取不同的 Hook 策略。

将系统共享库 Hook 的自定义的 Hook 模块加载至窗口 1，也就是待监测组件启动前。该 Hook 模块的 Hook 原理流程为：

1)找到定义原函数的 so 共享库文件，根据共享库符号表找到原函数的首地址，并将首地址处的原始指令进行备份，大小为跳转指令大小与新函数地址大小之和，大约 8 个字节；

2)在内存中分配一段空间，在内存空间中构造 ShellCode，其作用为跳转到新函数执行并返回，执行备份的原函数指令，跳转到原函数被备份指令后面的指令地址处执行；

3)在原函数首地址处写入跳转指令和 ShellCode 地址，使原函数跳至 ShellCode 处执行。

由于这种方式的 Hook 发生在原函数定义处，因此任何依赖该共享库的其他可执行文件使用被 Hook 函数时都将生效。Linux 系统调用函数被封装在系统库的 libc.so 中，如文件操作函数 open、设备驱动管理函数 ioctl 等。AndroidMonitor 在窗口 1 处通过 JNI 方式调用 Native 层的 Hook 逻辑对 libc.so 的函数进行了 Hook，能够实时监控系统调用函数的执行状况。

对于待监测 App 中 Native 层的 so 共享库，由于无法确定 Java 层 System.loadLibrary 的执行时机，因此在窗口 2 和窗口 3 处就无法完成对监测 App 中 so 库的 Hook。但 System.loadLibray 底层调用 linker 的 dlopen 函数，AndroidMonitor 在窗口 1 处对系统 linker 的 dlopen 函数进行 Hook，可以得到待监测 App 中 so 的加载时机。对 dlopen 进行 Hook 后，跳转执行的新函数 new_dlopen 的伪码如图 3 所示，在 new_dlopen 函数中，调用原函数 old_dlopen 加载 so 库后，再将 so 库名和返回的 handle 传入 hasLoadSo 函数中，在 hasLoadSo 函数中对 so 库名称进行判断后，再根据 handle 对待监测应用中的 so 库进行 Hook。

```
void * new_dlopen(const char * name,int flag){
    void * ret=old_dlopen(name,flag);
    hasLoadSo(name,ret);
    return ret;}
void hasLoadSo(const char * name,void * handle)
{
  if( name==待监测应用中的 so 库名称){
根据 handle 对待监测应用中的 so 进行 Hook;
    }
    }
```

图 3 new_dlopen 和 hasLoadSo 函数图

Fig. 3 new_dlopen and hasLoadSo method diagram

4.5 设备信息防护模块

其他类型的侵入式动态监测方案的监测过程中没有考虑到设备信息的泄露，本文提出的易部署的动态监测方案具有设备信息保护功能，能够用自定义的虚拟设备信息来代替主机的设备信息，防止监控过程中设备信息的泄露。

设备保护模块由两部分组成。第一部分位于虚拟服务进程，虚拟服务进程中定义了一个虚拟信息管理服务 VVirtualInfoManagerService，该服务内部储存了自定义的虚拟设备信息。待监测 App 通过 Binder Hook 方式来访问 VVirtualInfoManagerService 中的虚拟设备信息。第二部分位于待监测 App 启动时期即窗口 1 处，通过反射方式对 android.os.Build 类中的设备信息进行替换，从而防止设备信息泄露。

5 应用场景

本文设计的动态监测方案具有多种应用场景。

5.1　改善现有监控系统的易部署性

Android 应用行为动态监测的核心技术为 Hook，以侵入式方式实现的动态监测系统均可移植到本文的动态监测系统中实现其监测功能，从而增加监测系统的易部署性。以 Jiang 提出的动态监测方案[16]为例，该系统利用进程注入技术 Hook libbinder.so 文件 GOT 表中的 ioctl 函数，在 new_ioctl 函数中对 binder_transaction_data 结构体进行解析，实现对系统的监控。移植时，我们在本系统中直接 Hook libc.so 中的 ioctl 函数，定向到 new_ioctl 函数即可，从而省去了跨进程注入等诸多环节，并且容易部署到不同设备上。

5.2　应用行为的实时监控

AndroidMonitor 将插桩的敏感 API 的调用时间、敏感 API 类型、API 方法名、参数信息及返回值以日志形式输出。

监控模块对系统服务敏感 API 进行了 Hook，可全面掌握待监测应用与系统服务的交互情况，因此可用于对待监测应用进行实时防护。本文在此基础上实现了一个 Intent 拦截模块，该模块能够实时拦截监控应用中组件间跳转时 Intent 携带的信息。

5.3　脱壳

恶意应用在运行过程中很可能动态加载 DEX 文件和 so 文件，以绕过静态检测。对此，需要在应用运行过程中提取释放的恶意 DEX 文件和 so 文件，而这些释放的文件被加载到内存中时必须经过 libc.so 文件提供的 open，mmap 和 memcpy 系统函数之一。通过 Hook 这些 Native 函数，并比较这些函数参数中的字符串是否包含相应的文件后缀名或者地址的前 n 个字节中是否包含相应文件的魔数，来获取内存地址和文件大小，以此 Dump 内存中的相应文件，获取源码。这种方式对 DEX 文件和 so 文件的整体加密具有良好的支持性。本文实现的 AndroidMonitor 系统能够在免 Root 环境下实现这些功能。

5.4　隐私保护

本文实现的设备信息防护模块能够自定义应用对应的设备信息，防止监控过程中设备信息泄漏，可以直接用于隐私保护，将应用安装到沙箱内运行即可。该模块的设置界面如图 4 所示。

图 4　虚拟信息设置界面

Fig. 4　Virtual information setting interface

此外，对函数的 Hook 还可以应用到流量监控、文件透明加密等多个方面。

6　实验评估

6.1　敏感 API Hook 测试

AndroidMonitor 对 4.3 节指出的相关监控点进行了大量 Hook，表 1 分析了部分具有代表性的敏感 API。

表 1　Hook 表

Table 1　Hook table

敏感 API	Hook 方式	是否正常 Hook
TelephonyManager.getDeviceId	Binder Hook	是
Cipher.getIV	Java Hook	是
ContextWraper.startActivity	Java Hook	否
ioctl	Native Hook	是

由表 1 可知，AndroidMonitor 对 Binder Hook 和 Native Hook 具有良好的支持性；而 Java Hook 主要受限于 ART 虚拟机下的 dynamic callee-side rewriting 技术，过短的方法有可能被内联到调用方而无法进行 Hook，因此对于 Java 敏感 API，应尽量选择方法体指令较多的方法进行 Hook。

6.2　类监控软件的对比

Inspeckage 是一款安卓应用动态分析工具[17]，通过 Hook 技术来动态监测 Android 应用行为。Inspeckage 运行时，要求目标设备设置安装 Xposed 框架，它能够对网络流量、文件系统等 Java 层敏感 API 进行监控。DroidInjecter 是一款基于 ptrace 进程注入实现的动态监测系统。表 2 对比了 3 种工具。

表 2　软件对比表

Table 2　Softwarte comparison table

	Inspeckage	DroidInjector	AndroidMonitor
Root 手机	需要	需要	不需要
修改系统文件	需要	不需要	不需要
Binder Hook	不支持	不支持	支持
Java Hook	支持	支持	支持
Native Hook	不支持	不支持	支持

通过对比发现，相比其他动态监测工具，AndroidMonitor 具有易部署的特点，在安装部署时，既不需要设备 Root 权限，也不需要对设备系统进行任何修改，不依赖 Xposed 框架。同时，它支持 Binder Hook，Native Hook，Java Hook，能够对系统敏感 API 进行全面的监测。

6.3　性能测试

在 Hook 原理上，几种方案的本质都是向原函数写入跳转代码，改变函数的执行流程；不同的是，本文方案并没有通过获取系统的 Root 权限去修改系统文件来加载 Hook 模块，而是采用插件化技术，因此容易部署到不同设备上。与其他方案相比，本文方案在插件化环境初始化方面的性能消耗较大。表 3 对比了本文方案对 4 种应用正常安装启动和沙箱内安装后的启动时间，表中数据是在 Nexus5 真机上进行测试（系统版本为 5.0），每项测量 10 次后取平均值所得。

表 3　启动时间的对比

Table 3　Comparison of start-up time

（单位：ms）

	版本	正常启动	沙箱内启动
墨迹天气	7.0809.02	1684	2739
WIFI 钥匙	5.6.5	1406	2567
新浪新闻	6.2.2	2944	4318
微信读书	3.5.0	2843	5059

由表3可知，本文方案在沙箱环境初始化过程中会耗时1.4s左右，这也是本文方案相比其他监控方案增加的主要时间。

另外，本文还采用了腾讯发行的Android GT性能测试工具[18]对所提方案的性能进行了测试。仿真环境同上，进程的CPU占用率(PCP)、内存占用大小(PSS)的性能测试结果如表4所列。

表4 性能测试结果

Table 4 Performance test results

	PCP/%	*PSS*/kb
宿主进程	0.24	26490
虚拟服务进程	1.85	20184

6.4 方案优化

本文提出的方案在实际测试过程中仍存在一些不足。

(1)用插件化技术实现的沙箱采用了反射技术，但增加了应用的启动时间，少量加固的应用在沙箱内无法正确安装运行进行监测，需要适配。更接近原生Android系统的沙箱是移动端目前研究的一个方向，这样的环境更接近真实运行环境。

(2)在Hook过程中，备份指令时，并没有考虑相对寻址指令的修复，这可能导致Hook后函数运行过程中程序出现崩溃。因此，指令修复也是未来需要做的工作。

结束语 本文提出了一种非侵入式、易部署的Android应用动态监测方案。该方案利用插件化技术实现了具有监测应用行为的宿主App，能够将待监测应用以插件形式加载到宿主App运行，并监测其动态行为。该监测系统能够同时对Java层和Native层进行监控，并包含设备信息保护模块，可防止在监测过程中设备信息的泄露，具有非侵入式、易部署的特点。未来需要对沙箱环境的实现方式进行改进，并对Hook过程中备份的指令进行修复。

参 考 文 献

[1] PAKW, CHA Y, YEO S. Detecting and tracing leaked private phone number data in Android smartphones[C]// International Conference on Information Networking (ICOIN). IEEE, 2015: 503-508.

[2] ZHENG M, SUN M, LUI J C S. DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability [C]// Wireless Communications and Mobile Computing Conference (IWCMC). IEEE, 2014: 128-133.

[3] SHEN K, YE X J, LIU X N, LI B. Android App behavior-intent inference based on API usage analysis[J]. Journal of Tsinghua University, 2017, 57(11): 1139-1144.

[4] ARZT S, RASTHOFER S, FRITZ C, et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps[J]. Acm Sigplan Notices, 2014, 49(6): 259-269.

[5] ENCK W, GILBERT P, HAN S, et al. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones[J]. ACM Transactions on Computer Systems (TOCS), 2014, 32(2): 5-34.

[6] REINA A, FATTORI A, CAVALLARO L. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors[J]. EuroSec, April, 2013.

[7] FAN W, SANG Y, ZHANG D, et al. DroidInjector: A process injection-based dynamic tracking system for runtime behaviors of Android applications[J]. Computers & Security, 2017, 70: 224-237.

[8] YANG C, XU Z Y, GU G F, et al. DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications[C]// European Symposium on Research in Computer Security. 2014: 163-182.

[9] SCHREIBER T. Android BinderAndroid Interprocess Communication[C]// Seminar thesis, Ruhr-Universität Bochum, 2011.

[10] CONSTANTINESCU A S. Ensuring privacy in the android os by hooking methods in its api[J]. Journal of Mobile, Embedded and Distributed Systems, 2015, 7(3): 107-112.

[11] CHEN X Y, WANG D Q. Research and Implementation of Android Proxy Based on Dynamic Agent [J]. Industrial Control Computer, 2017(7): 99-100.

[12] JI S B. Basic principles of VirtualApp[EB/OL]. http://rk700.github.io/2017/03/15/virtualapp-basic/.

[13] JIA P, HE X, LIU L, et al. A framework for privacy information protection on Android[C]// 2015 International Conference on Computing, Networking and Communications (ICNC). IEEE, 2015: 1127-1131.

[14] WIβFELD M. ArtHook: Callee-side Method Hook Injection on the New Android Runtime ART[D]. Saarbrücken: Saarland University, 2015.

[15] WEI S. AOP implementation on ART [EB/OL]. http://wei-shu.me/2017/11/23/dexposed-on-art/.

[16] JIANG X, ZHANG H X, MU D J A Method for Dynamically Monitoring Android Applications [J]. Journal of Northwestern Polytechnical University, 2016, 34(6): 1074-1081.

[17] vul_wish. Inspeckage-Android Package Inspector[EB/OL]. https://www.freebuf.com/sectool/98607.html.

[18] vul_wish. Inspeckage: 安卓动态分析工具[EB/OL]. https://www.freebuf.com/sectool/98607.html.

[19] Tencent. GT[EB/OL]. https://gt.qq.com/index.html.

SU Xiang, Ph.D, is not member of China Computer Federation. His main research is Android security.

HU Jian-wei, professor, is not member of China Computer Federation. His main research interests include cyber security and cyber confrontation.