文章编号：1007-757X(2017)04-0048-05

Android 平台恶意代码检测通用脱壳机的设计

黄灿，　邱卫东，　王力

（上海交通大学 电子信息与电气工程学院，上海 200240）

摘　要：随着移动互联网的普及，以及 Android 平台所占份额的逐步提高，Android 平台应用程序的安全性尤为重要。因此设计了一个基于 Android 平台恶意代码检测的高效通用脱壳机，并辅以静态恶意代码的检测。详细介绍了 Android 平台主流加固技术以及该通用脱壳机的实现原理和方法，结果表明，该脱壳机能够应对市面上几乎所有的主流加固厂商。

关键词：恶意代码检测；脱壳机；加固；静态分析

中图分类号：TP311　　**文献标志码**：A

Design of the Universal Unpacker for Malicious Code Detection on the Android Platform

Huang Can, Qiu Weidong, Wang Li

(Shanghai JiaoTong University, School of Electronic Information and Electrical Engineering, Shanghai 200240, China)

Abstract: With the popularity of mobile Internet and the growing share of the Android platform, the safety of the application on the Android platform has become an important issue. Hence, this paper has designed an efficient universal unpacker for the malicious code detection based on the Android platform in aid of the static analysis. The mainstream reinforcement technologies for the Android as well as the principle and method of realizing the universal unpacker have been illustrated. It is found that the unpacker can satisfy the needs of almost all mainstream reinforcement companies in the market.

Key words: Malicious code detection; Unpacker; Reinforcement; Static analysis

0　引言

近年来智能移动终端和移动互联网的发展如火如荼，由于 Android 系统的开放性，现今越来越多的智能终端上都运行着 Android 系统，Android 系统是当前占有市场份额最大的智能移动终端操作系统，安全研究人员亦逐渐聚焦到 Android 系统应用程序的安全研究上来。对于恶意代码检测而言，目前学术界将其分为静态检测和动态检测两个方向。在静态检测领域，安全研究人员开发出了基于代码相似度检测的 ViewDroid[1]、基于组件的 Chex[2]、基于数据流的 SCanDroid[3] 和基于控制流图搜索和数据标记的 DroidChecker[4] 等工具；在动态检测方面，也有 TaintDroid[5] 之类的基于污点跟踪技术的隐私泄露监控系统和 VetDroid[6]、DroidScope[7]、CopperDroid[8] 之类的细粒度动态行为跟踪系统等。其中在 Android 平台静态恶意代码检测中最知名的就是 Androguard[9]，很多优秀的恶意代码检测系统都是基于它进行开发的，同时提供了很多辅助分析模块供分析人员使用。之后安全研究人员陆续开发了 FlowDroid[10]、RiskRanker[11]、DroidMOSS[12] 等等静态分析系统。这些静态分析系统都利用了 Android 平台使用的 Java 代码较为容易反编译的特点，从代码片段入手进行高效快速的恶意代码检测。但是随着 Android 平台软件保护技术的发展和普及，越来越多的恶意代码对自身的加固保护进一步增强，且同时由于混淆、花指令和代码加密等加固技术的不断加强，导致静态工具检测面临诸多困难，难以实现大规模部署的应用，所以当前的静态分析系统往往需要一个功能强大的通用脱壳机加以辅助。

本文结合当前 Android 平台主流加固技术设计并实现了一个针对应用层的高效率通用脱壳机，本文的组织架构如下：

第二章详细介绍了 Android 平台应用层加固的背景知识，具体包括 Android DEX 文件的加载机制，应用层加固的主要类别以及编写脱壳机所面临的主要困难。

第三章设计并实现了该通用脱壳机，并详细阐述了反-反模拟器模块、DEX 应用层脱壳的工程实现细节和相关原理等。

第四章对脱壳机的覆盖面和时间效率方面进行了全方位的测试，对实验结果进行分析，验证系统性能。

本文最后指出了当前脱壳机的优缺点，提出了改进方法，并说明了下一步的研究方向。

1　加固背景知识

Android 应用程序是 APK 文件，本质是一个 ZIP 的压缩

作者简介：黄　灿(1991-)，男，硕士研究生，研究方向：移动安全。

邱卫东(1973-)，男，教授，博士，研究方向：计算机取证、密码分析破解、密钥防护及电子信息对抗。

王　力(1993-)，男，硕士研究生，研究方向：移动安全。

文件，开发人员编写的应用层代码会被编译成 Dex 文件并打包进 APK 文件中。由于 Java 应用层代码容易被反编译，被攻击者破解，所以开发者为了保护应用程序，对抗逆向分析，而对应用进行加固混淆等处理，后文的分析均是基于 Android 4.1.2 的 Android 源码。

1.1 Android 文件加载机制

在 Android 系统中加载 APK 一般有两种方式，一种是通过文件的方式加载，另一种是通过字节流的方式加载，这两种方式本质上是一致的，只是数据的存储方式不同。在 Android 系统中，每个类加载创建进程时都会通过类 PathClassLoader 进行加载，同时开发者也可以通过 DexClassLoader 动态加载额外的 dex 文件。PathClassLoader 和 DexClassLoader 都继承自 BaseDexClassLoader 这个基类，二者会生成一个 DexPathList 对象，在 DexPathList 的构造函数中调用 makeDexElements()函数，继而在 makeDexElement()函数中调用 LoadDexFile()对 dex 文件进行处理，LoadDexFile()最终返回 DexFile 对象。LoadDexFile()函数的执行细节是通过 JNI 进入原生层，调用原生层的相关函数进行处理的。

进入原生层之后，针对 DEX 文件和二进制字节流，系统分别提供了 DvmDexFileOpenFromFd 和 DvmDexFileOpenFromPartial 这两个函数进行处理，这两个函数的最终目的都是构造一个 DexOrJar 结构体，并通过 JNI 把该结构体的地址保存到 DexFile 的私有成员变量 mCookie 中，这一过程结束之后退回到应用层，基本上完成了一个 Android 类的加载过程，后续系统模块中的 Dex 监控模块和 DexDump 模块都是基于此加载流程实现的。

1.2 加固技术

应用层加固一般采用采取自保护变形技术[13]对自身进行保护，目前加固变形技术包括文件整体加密、字节流加密、关键信息破坏、字节码抽取、字节码变形和混合方案等。

文件整体加密是针对 classes.dex 文件进行加密，并以资源文件的方式保存在系统的某处，当进程加完载壳程序后，壳程序会从资源文件处解密 dex 文件，再将解密后的 dex 文件保存在文件系统某处，最终动态加载进内存。

字节流加密所通过二进制字节流的方式加载 dex 文件，其解密过程是在内存中完成，原始的 dex 文件不会保留在文件系统。

关键信息破坏则是同时使用上面两种加壳方案，解压后的 dex 文件最终会完整的存放于内存的某个连续区域，因此只要找到其起始地址和长度，分析人员就能从内存中完整的 dump 出 classes.dex。但对于 Dalvik 虚拟机而言，要正确执行一个 dex 文件并不需要该文件的结构信息，所以可对 dex 文件的结构信息进行破坏，从而影响分析人员对 dump 下来的文件进行静态分析。

字节码抽取是关键信息破坏方案的进一步发展，主要是把 classes.dex 的内容分散存放。这个技术利用了 dex 文件的数据是使用偏移值进行读取的原理，针对这种情况，分析人员必须对当前进程中分布在不同区域的数据进行重建，重新构造出一个连续存储的 dex 文件，并保存到文件系统。

字节码变形的主要目的有两个，一个是隐藏字节码，二是提高静态分析的难度。各家加固服务技术实现细节千差万别，但技术原理类似，其一是修改 Encode Method 结构的 access_flags 和 code_off 字段，使原来的方法变成 native 方法。当该方法被执行时，通过 JNI 机制，就会先执行壳的逻辑，壳会还原 Method 属性，然后通过 JNI 所提供的接口，重新执行原来的字节码。对于这种方式，如果用静态分析工具分析，会发现那些被标记为 native 的方法只剩下方法声明，从而达到隐藏字节码的目的。

其二是在加固时，先为选中的方法添加一个跳转方法，这个跳转方法的 DexCode 和原方法完全相同，而原方法按方式一处理成 native 方法。当执行到壳逻辑时，壳再通过 JNI 所提供的接口，调用跳转方法。这种方式改变了原程序逻辑流程，增加了分析的困难。

混合类型是以上所述方法的综合体，文件结构更加碎片化、字节码变形更加多样化、加密更加精细化，因此分析难度更高。

1.3 脱壳机面临的困难

目前加固服务提供商一般还会结合其他防御手段，比如防注入、反调试、反模拟器、防 dump 等，这些防御手段的混合使用，进一步提高了脱壳的难度。

1）防注入技术[14]可以有效避免内存 dump，关键函数被挂钩等，比如 Apkprotect[15]会遍历检查/proc/self/maps 的加载列表，如果发现未知的 dex 文件被加载，则直接退出进程。

2）反调试[16]可以勘察出当前进程是否被调试，一旦发现被调试则直接退出进程。主要分为：对调试器监测，进程运行状态检测以及多个进程相互 ptrace[17]实现反调试。

3）反 dump 技术主要有如下两种方法。一种是利用 Inotify 机制，该机制提供了监视文件系统的事件机制，可用于监视个别文件，或者监控目录。二是对 read 函数等进行 hook，检测 read 函数当前读取的数据是否属于关键内存区域，从而阻止内存 dump。

4）反模拟器技术主要用于检测当前运行环境是否为模拟器，常见的模拟器检测技术主要分为基于特殊文件的模拟器检测技术以及基于系统特定属性的模拟器检测技术两种。在 Android 模拟器中存在一些独属于模拟器的特殊文件，应用程序可以通过检测当前系统中是否含有这些特殊文件来确定应用是否处于模拟器的运行环境，同时模拟器中 Android 系统的一些特定属性与实体机不同，因此也可以利用系统中的这些特殊属性进行检测，详细的特殊文件和属性，如表 1 所示。

表 1　模拟器检测相关文件和属性列表

模拟器系统特殊文件	模拟器系统特殊属性
/dev/socket/qemud	ro.hardware == goldfish
/dev/qemu_pip	ro.product.device == generic
/system/lib/libc_malloc_debug_qemu.so	ro.product.model == sdk
/sys/qemu_trace	ro/product.model == sdk
/system/bin/qemu-props	

2　脱壳机设计与实现

综合之前所述，本文设计了一个针对应用层的通用脱壳机，从上而下分为 DEX Dump 应用层、反-反模拟器模块和 DEX 加载监控模块、API HOOK 框架层和 QEMU Android 模拟器层五个层次，脱壳机的整体架构，如图 1 所示。

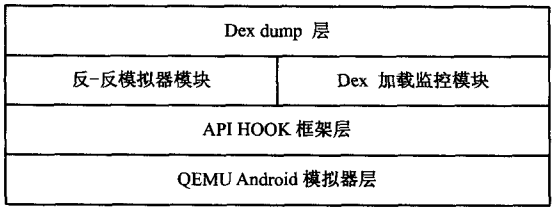

图 1　脱壳机架构

对于 DEX Dump 应用层而言，当壳程序完整解固后，在当前进程中必然保存着完整的 dex 信息(或者部分信息被篡改，但依然可以推断出原始信息)，因此只要成功注入到目标进程，通过收集运行时的关键信息，再利用 Memory Dump 或重建的技术手段，就可以把原始的 dex 数据提取出来。本文基于 Cydia Substrate 框架，在 dex 加载的合适时机添加监控点，收集脱壳所需的关键信息。同时考虑到某些加固方案可能会在加载后做一些内容恢复或替换的操作，因此脱壳机会在整个应用都正常运行起来，待“壳”的修复动作达到稳定之后再进行脱壳，这样就完成了应用层的脱壳。

同时为了方便在模拟器上进行后续工业级大规模的部署，本文开发了反-反模拟器模块，该模块主要用于隐藏模拟器的特殊文件和属性，以应对模拟器的检测，整体基于 API HOOK 框架层而实现。

API HOOK 框架层主要是为上层模块提供服务，涉及 Java Hook 和 Inline Hook 等技术，该模块整体基于 Cydia Substrate 框架实现。另外在这个模块额外添加了过滤策略，过滤掉系统内置的应用，把作用范围限制在本文关注的样本进程中，该过滤策略作用于 Hook Layer 以上的所有模块。

QEMU Android 模拟器层主要是为后续大规模部署、反-反模拟器模块等服务。

由于该脱壳机是面向海量样本的，且要求达到完全自动化的并发执行任务，脱壳机采用 Google 原生 Android 模拟器作为工作环境，模拟器硬件配置，如表 2 所示。

表 2　模拟器硬件配置

1.	Device: Nexus One (3.7", 480 * 800:hdpi)
2.	Target: Android 4.1.2 - API Level 16
3.	CPU/ABI: ARM(armeabi-v7a)
4.	Memory Options: 512MB
5.	Internal Storage: 1GB
6.	SD Card: 512MB
7.	Emulator Options: Snapshot

2.1　反-反模拟器模块

加固软件会检测模拟器系统特殊文件和系统特殊属性，而这些操作都需要通过调用 libc.so 库中的 fopen 等 API 函数读取系统文件属性来做判断，考虑到 API 函数执行的正常流程是先从 API 到 ABI 再到内核层，本模块采取的方案是 HOOK ABI 层次，由于 ABI 是所有函数执行必须经过的层次，通过 HOOK ABI 调用接口，使其先执行自定义的系统函数，再返回正常的后续逻辑，这样可以避免诸多限制和制约。

本模块最后通过 LKM 实现对 Linux 内核功能的扩展，通过在 ABI 层面 HOOK 模拟器常用的系统库函数而得以实现该模块。

2.2　DEX 应用层

针对 DEX 加载监控模块，本系统主要是在 APK 加载流程中添加了两个监控点：BaseDexClassLoader 的构造函数和 dvmDexFileOpenPartial 函数。通过 HOOK BaseDexClassLoader 的构造函数，可以得到所有继承于此类的 BaseDexClassLoader 以及子类实例引用。通过这些实例引用，可以获得 DexOrJar 指针，为此本脱壳器设计了类 ClassLoaderDumper，该类负责存储这些实例引用对象。

正常的加载流程只能加载 Dex 文件，而通过 dvmDexFileOpenPartial 可以加载 Dex 二进制字节流，因此通过这个函数，可以获取保存 Dex 数据的内存地址 Addr，数据长度 Len 和构造 DvmDex 结构的地址，本脱壳器设计了类 HookDexDumper 负责存储这些数据应用层脱壳所涉及到的几个关键类之间的联系，如图 3 所示。

Dex Dump 是应用层 DEX 脱壳机的核心部分，一共包含 5 个处理流程，并封装成 5 类，它们分别是 HookDexDumper、DexOrJarDumper、ClassLoaderDumper、DirectlyDumper 和 RebuildDexDumper，且都继承自类 BaseDexDumper。这 5 类所完成的流程互相依赖，而且每个流程都会有相应的输出，当所有的流程运行完毕之后，会得到一个或多个 dex 文件，紧接着脱壳机会进行分组，并从每组中提取出最优的文件。

HookDexDumper 类，当监控到 dvmDexFileOpenPartial 函数被调用时，脱壳机会马上生成 HookDexDumper 对象，HookDexDumper 根据传入的 addr 和 len，把 dex 的所有数据复制保存到 copyData_，这样做主要是考虑到后面 dex 数据有被篡改的可能，然后 HookDexDumper 会先把“壳”过滤掉，然后将 copyData_写入文件系统。

DexOrJarDumper 类负责处理 DexOrJar 结构。其中 dump 的过程相比于其他的几个流程会复杂一点，先通过 pDvmDex 获取 dex 在内存的中开始地址 addr 和 size。如果是 dex 文件，则检查 dex 文件头是否合法，主要是检查是否被篡改，如果合法则启用 DirectlyDumper 流程。考虑到有字节码抽离的情况，因此紧接着再启用 RebuildDexDumper 流程。如果是 odex 文件，过程与 dex 文件类似。

ClassLoaderDumper 类。任何继承自 BaseDexClassLoader 的子类在创建实例时都要触发 ClassLoaderDumper 类的创建。该类主要保存 Java 实例的引用。ClassLoaderDumper 的 Dump 的逻辑很简单，先从 classloader_成员变量中获取所有 DexOrJar 的地址，然后调用 DexOrJarDumper 的 Dump 方法。

DirectlyDumper 主要根据传入的 addr 和 size 把 dex 写入文件系统，其过程跟 HookDexDumper 流程相比 主要是多了 dex 和 odex 的判断，影响最终写入文件的命名后缀。Re-

buildDexDumper 类比较复杂，除了解决字节码抽离的情况，同时在重建的过程中能够修正被某些刻意篡改的数值。

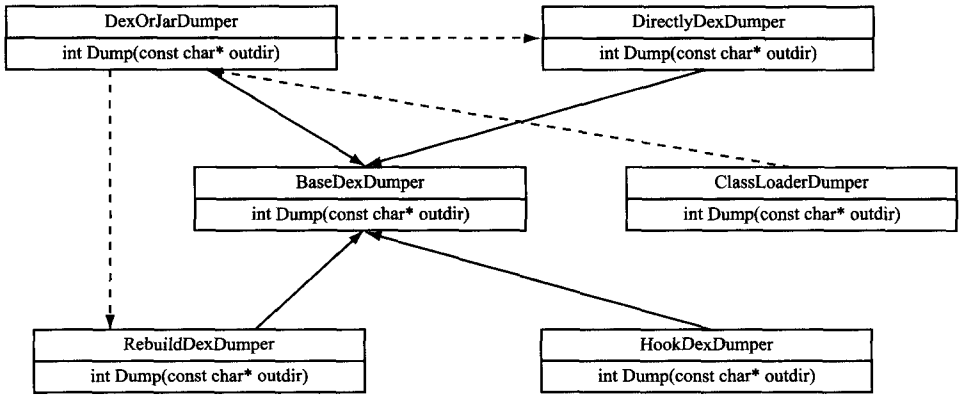

图 3　应用层关键类关系

这 5 类所完成的流程相互依赖，并且每个流程都会产生相应的输出，当所有的流程运行完毕之后，便会得到多个 dex 文件，这边是 DEX 应用层所输出的结果，DEX 应用层类协作脱壳的完整流程图，如图 4 所示。

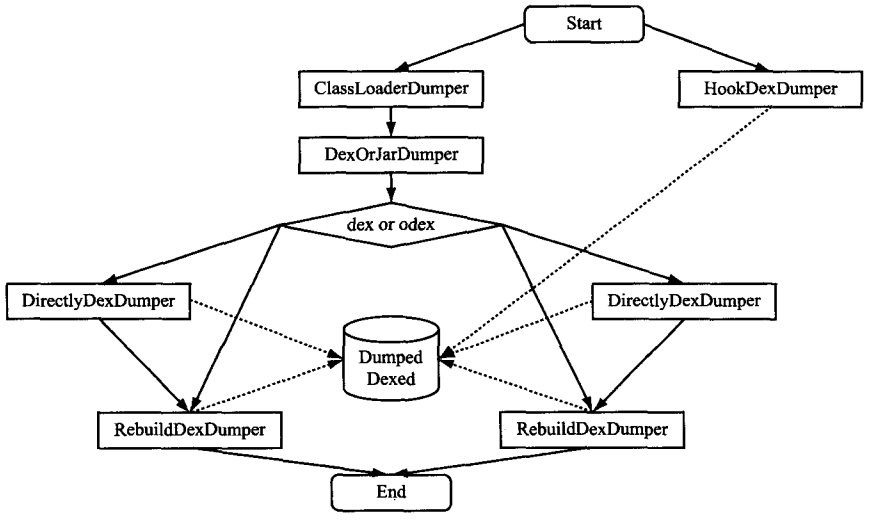

图 4　应用层类协作脱壳流程

3　试验与讨论

3.1　脱壳机有效性分析

使用此脱壳机对市场上绝大部分加固厂商的普通版加固产品进行脱壳测试，测试结果，如表 3 所示。

表 3　脱壳机效果展示

加固厂家	能够脱壳成功
阿里聚安全	能
百度加固	能
腾讯加固	能
360 加固	能
梆梆加固	能
爱加密	能
通付盾	能
其他	不确定

从上面的结果可以看出，本文设计的脱壳机能够应对市面上所有主流的加固厂商。

3.2　脱壳机效率

脱壳机的效率主要参考单个加固样本的平均脱壳时间。脱壳时间是指从脱壳机获取样本后开始准备脱壳任务开始，但脱壳完成输出脱壳后样本文件为止。中间主要包括启动脱壳机时间，安装待脱壳样本事件以及脱壳时间。其中启动脱壳机时间一般固定不变，而后两者会根据具体样本的不同变化而变化。通过对随机抽取 10 个加固样本，分别对它们进行脱壳处理，各个样本的脱壳时间，如图 5 所示。

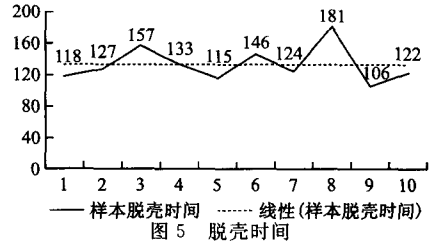

图 5　脱壳时间

从图 5 可以看出，每个样本的脱壳时间并不相同，这主要跟样本本身的大小有关。一般情况下，样本越大，其安装时间就越长，脱壳时间也越长，其最终所需要的脱壳时间就越大。估算出平均需要的脱壳时间为 132.9 秒，那么一台模拟器一天可以脱壳 650 个样本，假设每台主机可同时运行 6 台模拟器，共 10 台主机，那么一天一共可以完成 39 000 的脱壳量，足够覆盖每天的新增样本数。

4　总结

本文设计了一个基于 Android 平台恶意代码检测静态分析的通用脱壳机，以辅助静态分析能够分析加固后的应用程序，该脱壳机能够应对市面上绝大部分代码在应用层采取的加固技术，且脱壳迅速，从而极大的提高了静态检测的准确率和效率。

由于本系统未深入探讨原生层的脱壳问题，但考虑到双层协作脱壳是后续的主流，因此原生层的脱壳问题值得进一步研究和关注。后期可以借鉴 Windows 平台 PE 文件格式的一些成熟检测技术，拓宽对原生层脱壳的处理。

其次脱壳与恶意代码检测息息相关，未来可以考虑将本系统与恶意代码检测相结合，脱壳后运用大数据和机器学习的方法建立恶意代码行为模式，进行漏洞探测等研究，以便建立更加智能化的恶意代码检测机制，提供更加精确的查杀结果。

参考文献

[1] Zhang F, Huang H, Zhu S, et al. ViewDroid: towards obfuscation-resilient mobile application repackaging detection[C]//Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks, 2014: 25-36.

[2] Lu L, Li Z, Wu Z, et al. Chex: statically vetting android apps for component hijack-ing vulnerabilities [C]//Proceedings of the 2012 ACM Conference on Computer and Communications Security, 2012: 229-240.

[3] Fuchs A P, Chaudhuri A, Foster J S. SCanDroid: Automated security certification of Android applications[R]. CS-TR-4991 of University of Maryland Tech, 2009:1-16.

[4] Chan P P F, Hui L C K, Yiu S M. Droidchecker: analyzing android applications for capability leak[C]// Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks, 2012: 125-136.

[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.

[6] Zhang Y, Yang M, Xu B, et al. Vetting undesirable behaviors in android apps with permission use analysis [C]//Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, 2013: 611-622.

[7] Yan L K, Yin H. Droidscope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis[C]//the 21st USENIX Security Symposium (USENIX Security 12), 2012: 569-584.

[8] Reina A, Fattori A, Cavallaro L. A system Call-centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors[J]. EuroSec, 2013: 1-6.

[9] Desnos A. Androguard-Reverse Engineering, Malware and Goodware Analysis of Android Applications [EB/OL]. http://code. google. com/p/androguard, 2013-03-26/2016-12-22.

[10] 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.

[11] Grace M, Zhou Y, Zhang Q, et al. Riskranker: Scalable and accurate zero-day an-droid malware detection [C]//Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, 2012: 281-294.

[12] Huang H, Zhu S, Liu P, et al. A Framework for Evaluating Mobile APP Repackaging Detection Algorithms[C]//International Conference on Trust and Trustworthy Compu-ting. Springer Berlin Heidelberg, 2013: 169-186.

[13] Schlegel R, Zhang K, Zhou X, et al. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones[C]//Proceedings of the 18th Annual Symposium on Network and Distributed System Security (NDSS), 2011, 11: 17-33.

[14] Lin J C, Chen J M. An automatic revised tool for anti-malicious injection[C]// Proceedings of the 6th IEEE International Conference on Computer and Information Technology, 2006: 164-164.

[15] Love R. Kernel korner: Intro to inotify[J]. Linux Journal, 2005, 2005(139): 8.

[16] Gagnon M N, Taylor S, Ghosh A K. Software protection through anti-debugging[J]. Security & Privacy, 2007, 5(3): 82-84.

[17] Dike J. A user-mode port of the Linux kernel[C]// Proceedings of the 2000 Linux Showcase and Conference. 2000, 2(1): 2.1.

(收稿日期：2016.09.06)