第 43 卷 第 11 期 计 算 机 工 程 2017 年 11 月
Vol. 43 No. 11 Computer Engineering November 2017

·安全技术· 文章编号：1000-3428(2017)11-0122-06 文献标志码：A 中图分类号：TP309

基于函数调用图的 Android 重打包应用检测

吴兴茹，何永忠

（北京交通大学 计算机与信息技术学院，北京 100044）

摘 要： 针对 Android 第三方市场中重打包应用日益增多的现象，提出一种利用函数调用图检测 Android 重打包应用的方法。对应用进行反编译，提取并分析 Smali 代码生成函数调用图，同时将函数中的操作码作为结点的属性对函数调用图进行处理，实现第三方库过滤并保留与界面相关的应用程序接口。在此基础上，用 Motif 子图结构表示函数调用图，根据子图的相似度计算应用的相似度，从而判断是否为重打包应用。通过对市场中 1 630 个应用的检测结果表明，该方法具有较高的准确性和良好的可扩展性。

关键词： Android 系统；重打包应用；函数调用图；相似度；子图

中文引用格式： 吴兴茹，何永忠. 基于函数调用图的 Android 重打包应用检测[J]. 计算机工程，2017，43(11)：122-127，139.

英文引用格式： WU Xingru，HE Yongzhong. Android Repackaged Application Detection Based on Function Call Graph[J]. Computer Engineering，2017，43(11)：122-127，139.

Android Repackaged Application Detection Based on Function Call Graph

WU Xingru，HE Yongzhong

（School of Computer and Information Technology，Beijing Jiaotong University，Beijing 100044，China）

【Abstract】 Aiming at the phenomenon that there is an increasing number of repackaged applications in the Android third-party application market，this paper proposes a method of detecting Android repackaged applications by using function call graph. It decompiles the application to gain the Smali code，analyzes the Smali code to generate a function call graph，processes the function call graph by using theoperationcode as the attribute of the node，filters the third-party library，and saves the Application Program Interface(API) associated with the interface. On this basis，it uses the Motifs' substructure to representthe function call graph. According to the similarity of the subgraph，it computes the similarity of the application，so as to determine whether it is a repackaged application. Detection results in 1 630 applications in the market show that the proposed method has higher accuracy and better expansibility.

【Key words】 Android system；repackaged application；function call graph；similarity；subgraph

DOI： 10.3969/j.issn.1000-3428.2017.11.020

0 概述

Android 是一种基于 Linux 内核的操作系统，其以开放和自由的特性深受广大厂商、开发人员以及用户的青睐。目前，该系统已拥有巨大的用户数量、市场份额和应用数量。随着 Android 操作系统占有的市场份额越来越大，应用数量越来越多，恶意应用应运而生并且逐渐增多。360 互联网安全中心发布的《2015 年第三季度中国手机安全状况报告》显示，2015 年第三季度，360 互联网安全中心共截获 Android 移动平台新增恶意程序样本 558 万个，平均每天截获新增手机恶意程序样本近 6.07 万个，累计监测到移动端用户感染恶意程序 7 722 万人次，平均每天恶意程序感染量达到了 83.9 万人次[1]。Android 平台的开放性和自由性，使其在快速发展的同时也吸引了恶意程序开发者的目光，一些恶意开发者从官方市场下载安装包，在应用中插入或者更换广告库，然后利用自己的签名重新发布应用，以此来获取收益；另一些恶意开发者在应用中插入恶意模块，然后重新将应用发布。这些行为破坏了市场秩序，侵犯开发者知识产权和利益，最重要的是泄露了用户隐私，使移动用户面临越来越多的安全问题。重打包应用是生成恶意应用的主要方式，近期研究显示，86% 的恶意应用是重打包应用[2]。

为快速有效地检测重打包应用，本文结合此类应用的特点，提出一种基于函数调用图的检测方

基金项目： 国家自然科学基金(61402035)。

作者简介： 吴兴茹(1992—)，女，硕士研究生，主研方向为移动安全、信息系统安全；何永忠，副教授、博士。

收稿日期： 2016-08-31 **修回日期：** 2016-10-28 **E-mail：** XingruWu@bjtu.edu.cn

法。该方法在生成函数调用图时,把函数中所有的操作码按顺序排列作为结点的属性,在比较结构子图相似度时,同时比较结点的属性以及结点之间的调用关系。此外,目前市场中的应用在发布之前都会经过混淆技术的处理,对于混淆过的应用,只比较函数名可能会出现误报和漏报率高、检测率低的情况。考虑到目前多数的应用只是利用 proguard 提供的混淆工具对函数名进行混淆,函数中关键的操作码不会被混淆,本文提出一个基于函数调用图检测重打包应用的方法,通过比较结点的属性提高检测效率。

1　相关工作

目前国内外已经有较多检测重打包应用的方法,如基于模糊哈希的检测算法、基于特征哈希的检测算法、基于程序依赖图的检测算法等。

DroidMOSS 方法[3]使用了基于模糊哈希的检测算法,首先提取 Dalvik 字节码中的操作码序列,采用模糊散列技术生成操作码的指纹,模糊散列技术可以有效地对应用进行定位并检测重打包应用的变化,通过比较应用程序的指纹之间的编辑距离来检测重打包应用。Juxtapp[4]是一个检测 Android 应用中相似代码的可扩展性基础架构,使用基于特征哈希的检测算法,可以用来检测应用中是否包含恶意代码、是否涉及盗版的代码重用以及是否具有恶意软件的实例,主要通过操作码的 K-Grams 和散列特征表示应用的特征,然后将相关位向量聚集在一起来识别相似代码。DroidMOSS 和 Juxtapp 都是从字节码中提取特征信息,DroidMOSS 采用模糊散列技术生成特征信息的指纹,而 Juxtapp 则生成了特征信息的 K-Grams[5],它们基于不同的散列技术实现了相似性比较。这 2 种方法的优势是简单快速,但是重打包作者可以轻松地避开此类检测技术。

PiggyApp[6]是一种快速的、具有可扩展性的方法,利用模块解耦技术,将一个应用分为主要模块和非主要模块,在主要模块中抽取不同的语义特征,将其转换成特征向量,然后创建一个向量空间,利用排序查找算法来检测应用,通过一种线性的搜索算法来降低检测方法的时间复杂度,该方法主要是用来检测 Piggybacked 应用。ViewDroid[7]基于 Android 应用的 UI 特点,给出了特征视图的定义。ViewDroid 根据 Activity 中涉及的应用程序接口(Application Program Interface,API)以及 Activity 的启动方式等信息来构建应用的特征视图,通过比较特征视图来检测重打包应用。DNADroid[8]是一种基于程序依赖图(Program Dependency Graph,PDG)的检测技术,是代码克隆检测技术中经常使用的方法,其作者收集来自官方 Android 市场以及非官方市场的75 000 个Android 应用程序,从中随机选择 9 400 对(即 18 800 个)潜在的可能被重打包的应用程序。实验结果表明,DNADroid 检测出了至少 141 个被重打包的应用,手动测试也证明了该结论。然而经过代码混淆,在指令之间的数据依赖关系发生变化时,可能导致重打包应用躲避重构图的检测。

针对上述方法所存在的不足,本文提出一个基于函数调用图检测重打包应用的方法。首先生成函数调用图,函数调用图中每个节点代表一个函数,将函数中所有的操作码作为结点的属性;然后对函数调用图进行处理,过滤掉第三方库,实验研究发现,与 UI 相关的 API 几乎不被修改,如一些应用的界面十分相似,仅仅修改图片以及文字,因此,保留与界面相关的 API;最后再进行图相似度的比较。在比较图相似时,先用 Motifs[9]模式中的结构子图[10]将图表示出来,然后比较子图结构的相似性,最后通过比较结构子图相同的个数计算 2 个应用之间的相似度。

2　研究背景

2.1　Android 应用程序

Android 应用程序是系统和用户之间交互的一个接口,应用程序通常都是用 Java 语言开发,然后在 Dalvik 虚拟机中运行,使用 Android SDK 工具对源码进行编译。Dalvik 虚拟机是一种特殊的 Java 虚拟机,它基于寄存器架构,指令集是 Dalvik 字节码[11]。应用开发者一般是将所有的源代码和资源文件一起打包成 APK(Android Packaged)文件,经过签名后发布到市场,用户可以从市场下载并安装到自己的手机中,所以,每个安装到手机中的应用程序都是一个以 APK 结尾的文件。一个 APK 文件包括应用程序的 Dalvik 代码、资源、数据、签名信息及 xml 配置文件等内容。Android 应用程序包括 4 个基本组件:Activity,Service,BroadcastReceiver,ContentProvider。

2.2　重打包应用

重打包又称植入,是指非法开发者对原开发者的应用程序进行修改,加入自己的代码后重新发布到应用市场的行为,重打包应用就是通过重打包后的应用程序。检测技术主要有静态和动态检测。目前来说,重打包应用主要包括 3 种情况:1)恶意开发者一般会利用一些常用的混淆工具,如 ProGuard、DexGuard、APKprotect 等对代码进行混淆,然后用 Keytool 和 jarsigner 签名工具对应用重新签名并发布到市场中;2)在官方应用中嵌入广告库,然后用自己的签名对应用进行重新签名并将其发布到其他第三方市场中,通过用户点击广告或者后台下载软件获取非法收入[12];3)与恶意程序相关,非法开发者下载目标应用,对其反编译,植入恶意代码,再重新打包和签名,然后发布到官方市场或者第三方市场。

合法应用程序到重打包的具体步骤如下：

步骤 1 合法开发者开发出一个应用程序并将其更新至官方 Android 市场。

步骤 2 恶意开发者下载获得合法的应用程序，并打包进恶意代码。

步骤 3 恶意开发者将含有恶意代码的应用程序更新至第三方应用市场，等待不知情的用户下载。

当用户下载该含有恶意代码的应用程序至手机后，恶意开发者便可通过重打包应用中的恶意代码窃取用户位置、联系人信息 SMS 信息，或在未经用户允许的情况下下载文件、打开未知网页等。重打包技术一方面侵犯了开发者的知识产权，损害了开发者的经济利益；另一方面用户进行下载和授权安装后，这些重打包应用会通过通知栏、悬浮窗提醒、广告栏等多种形式通知用户，吸引用户点击，窃取用户的隐私信息或者在后台下载各种软件，这些重打包应用严重威胁用户信息安全，扰乱市场秩序。

3 本文检测方法

3.1 检测原理

如图 1 所示，Android 重打包应用的检测包括函数调用图生成、函数调用图处理和相似度比较 3 个过程。

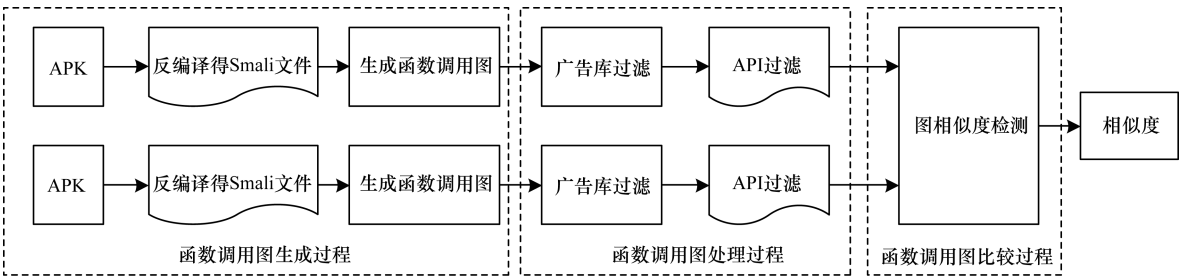

图 1 Android 应用重打包检测流程

1）函数调用图生成过程。此过程首先用反编译工具对应用程序进行反编译，获得应用程序的 Smali 代码，然后通过自己编写的 Java 程序对 Smali 代码进行分析处理，生成应用程序的函数调用图并为节点添加属性。

2）函数调用图处理过程。为缩短比较时间，提高检测的准确率和效率，先过滤掉第三方库，因为与 UI 相关的 API 几乎不被修改，所以保留与界面相关的 API，本文是通过创建了一个第三方库白名单和与界面相关 API 的白名单，然后利用自己编写的 Python 代码依次对第 1）步生成的函数调用图进行过滤和处理。

3）函数调用图比较过程。此过程首先用 Motif 子图结构表示函数调用图，然后把每个子图结构按照子图中的结点、结点属性以及调用关系的形式进行存储，通过比较子图结构计算函数调用图的相似度，根据此相似度来判断是否为重打包关系。

3.2 函数调用图生成

定义 1（函数调用图） 用有向图 $G = <V, E>$ 来表示函数调用图，其中，V 表示图 G 中所有顶点的集合，每个顶点代表一个函数，包括 Android 系统库函数以及用户自定义的函数，每个函数由其所在类名、方法名和参数类型组成。E 代表图 G 中所有有向边的集合，即 $E \subseteq V \times V$，且对 $\forall (u,v) \in V, \exists (u,v) \in E$，每条有向边代表了图 G 中节点（函数）之间存在的先后调用关系，即如果节点 u 在节点 v 之前被调用，则存在从节点 u 指向节点 v 的一条有向边，即 u 是调用节点，v 是被调用节点。

因为生成函数调用图的方法是基于 Smali 代码层进行处理的，所以在生成函数调用图这一阶段先要对应用程序进行反编译得到 Smali 代码，然后用 Java 程序分析 Smali 代码，生成应用的函数调用图，并为结点添加属性。

首先用 Apktool 反编译工具将应用程序反编译，得到对应的 Smali 代码、资源文件和 XML 配置文件，然后利用 Java 程序遍历 Smali 文件夹下的每个 Smali 文件，每个 Smali 文件是一个类，Smali 文件中的每个函数是一个结点，结点的 id 为该方法所在类的类名 + 该方法名 + 该方法的参数类型，然后为函数调用图中的结点添加属性，遍历每个函数中的操作码，将函数代码块中的所有操作码作为结点的属性，本文主要对 Smali 代码中常用的 221 个操作码进行分析。最后根据 Smali 文件中 invoke 关键字确定函数结点之间的调用关系，即确定函数调用图的边。具体步骤如下：

步骤 1 遍历所有 Smali 文件，根据 Smali 文件中的 method 关键字，获得所有的函数名，函数名即作为函数调用图的节点。图 2 是 Smali 文件中 onCreate( )方法的函数调用图，该方法即为一个结点，结点 id 为：com/example/wuxingru _ ui/LoginActivity; -> onCreate (Landroid/os/Bundle;)。

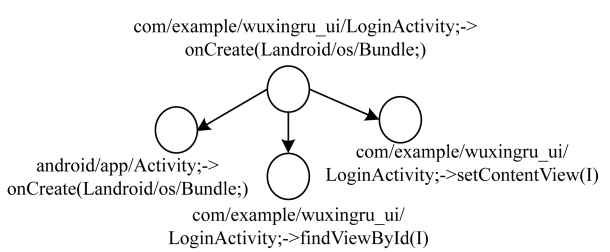

图 2 Smali 文件的函数调用图

步骤 2　遍历函数中的操作码,根据操作码与属性之间的对应关系,为节点添加属性;遍历每个方法中的操作码,如 onCreate( ) 方法中有 invoke-super、const、invoke-virtual、move-result-object、return-void 等 8 个操作码。根据操作码和属性之间的对应关系,为结点添加属性 ICICIVNR。

步骤 3　遍历所有的函数,在函数里寻找 invoke 关键字,若 invoke 的函数已经存在,则直接连一条边,若不存在,则先创建该节点,然后再进行连接;在 onCreate( ) 方法中共有 3 个 invoke 系列的操作码,因此,存在 3 条以该结点作为源结点的边。

3.3　函数调用图处理

函数调用图处理过程包括 2 个步骤:1)第三方库的过滤;2)保留与界面相关的 API。

3.3.1　广告库的过滤

目前很多 Android 应用程序中都添加了第三方库,如广告库、开发库、固定模板等。在生成图之后,首先应将第三方库过滤掉,因为这些第三方库会对实验结果会产生以下影响:

1)漏报。一个本身为重打包的应用,因为使用了不同的第三方库而没有检测出来,这种情况是因为该应用的核心代码比较少,绝大多数代码都是第三方库的代码,如果重打包的应用更换第三方库,使用和官方应用不同的第三方库,这时进行重打包检测时,会因为相似代码的比例过小,导致计算出来的应用的相似度比较低,这样就可能会出现漏报的情况。

2)误报。一个非重打包应用,因为使用了和官方应用相同的第三方库而被误判为重打包应用,这是因为非重打包应用使用了第三方库,由于第三方库的代码数量过大,甚至远远超过了本身核心代码的数量,这就使得该非重打包应用和官方应用之间的相似代码的比例过大,从而被检测为重打包应用。

为得到更准确的对比结果,需要先过滤掉第三方库,消除第三方库对实验结果的干扰。通过调研分析和检测,本文建立了一个第三方库白名单,该白名单中包含 100 多个第三方广告库和常用的开发库,生成的函数调用图以. gexf 文件的形式保存,在生成函数调用图之后,首先用此第三方库白名单过滤掉函数调用图中的第三方库,经过过滤后的图,仍以 gexf 的格式保存,用于下一阶段的分析。

3.3.2　界面相关 API 的提取

API 是操作系统留给应用程序的一个调用接口,不需要访问源码或理解内部工作机制的细节,应用程序可以通过调用操作系统的 API 使操作系统去执行应用程序的动作。Android API 是 Android 系统提供的,所有 Android 应用程序的行为都必须通过 API 和硬件联系在一起,在不修改原始行为的情况下,很难对 API 进行替换或者修改。重打包一般是更改第三方库或者修改一些代码,很少对一个应用程序的界面进行修改,因为更改应用的界面需要非法开发者对应用的整体逻辑结构十分清楚,这会耗费开发者更多的精力;另一方面,更改界面结构也会影响用户体验,所以,重打包的应用很少对应用程序的界面进行修改。因此,将和界面相关的 API 作为图的特征,预先过滤掉不相关的结点,可以提高图比较的效率和准确性,并且对于混淆技术具有更好的强壮性。保留和界面相关的 API 结点,这一过程主要用 python 脚本实现,在处理过程中使用复杂网络分析工具 Networkx。

本文首先解析函数调用图,提取 ID 和 Attvalue 标签的值,然后遍历与界面相关的 API,只把与界面相关的 API 之间有路径相连的点保存到新图中,最后将新图以 gexf 的格式保存,用于图相似度的比较。

3.4　函数调用图相似度比较

定义 2(图同构)　设 $G_1=(V_1,E_1)$ 和 $G_2=(V_2,E_2)$ 是 2 个简单图,若存在一个从 V_1 到 V_2 的双射函数 f,且 f 具有以下性质:对 V_1 中的所有 x 和 y,x 和 y 在 G_1 中相邻,当且仅当 $f(x)$ 和 $f(y)$ 在 G_2 中相邻,那么称 G_1 与 G_2 是同构的,这样的函数 f 称为同构函数。

定义 3(子图)　设图 $G_1=(V_1,E_1)$ 和 $G_2=(V_2,E_2)$,如果 $V_1\subseteq V_2$,且 $E_1\subseteq E_2$,则将 G_1 称为 G_2 的子图。若节点子集或边子集是真子集,则称这个子图为真子图。

研究表明,所有的图都可以用 Motif 图的子图结构来表示图的结构特征,软件网络图也可以用 Motif 的子图结构模式来表示图的结构特征。因此,本文在比较图相似度时,采用了 motif 子图结构模式,首先用 motif 子图结构先将函数调用图表示出来,在判断 2 个图之间相似时,通过比较两图中的 Motif 子图结构,确定子图相同的个数来计算 2 个应用之间的相似度。算法 1 描述了用 motifs 子图结构表示函数调用图以及图相似度的比较过程。

算法 1

```
1. Vs←G. nodes( )
2. NodeLinkAttr←new Array( )
3. for m←0 to Vs. length( ) - 1 do
4.     VNs←G. neighbors( Vs[ m ] )
5.     for i←0 to VNs. length( ) do
6.         VNNs←G. neighbors( VNs[ i ] )
7.         for j←0 to VNNs. length( ) do
8.             attr←Vs[ m ]. attr( ) + "→" + VNs[ i ]. attr( ) +
"→" + VNNs[ j ]. attr( )
9.             NodeLinkAttr. add( attr)
10.        end
11.    end
12.    if VNs. length( ) > 1 do
13.        for i←0 to VNs. length( ) - 1 do
14.            for j←i + 1 to VNs. elngth( ) do
15.                attr←Vs[ i ]. attr( ) + "←" + Vs[ m ]. attr( ) +
"→" + VNs[ j ]. attr( )
16.                NodeLinkAttr. add( attr)
17.            end
```

```
18.            end
19.        end
20.        for n←m + 1 to Vs. length( ) do
21.            VnNs←G. nodes( Vs[ n] )
22.            for j←0 to VnNs. length( ) do
23.                if VnNs[ j] in VNs do
24.                    attr←Vs[ n]. attr( ) + "→" VnNs[ j]. attr +
"←" + Vs[ m]. attr( )
25.                    NodeLinkAttr. add( attr)
26.                end
27.            end
28.        end
29. end return NodeLinkAttr
```

具体步骤如下：

1）用 Motifs 的子图结构将第 2 阶段处理过后的函数调用图表示出来，Motifs 图的子图结构共有 13 种，本文选用出现频率较高的 3 种子图结构来表示函数调用图，如图 3 所示。

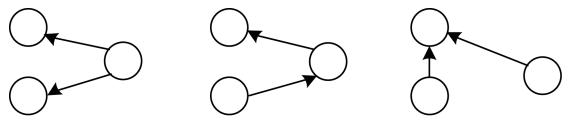

图 3 Motifs 子图结构

2）每一个子图结构都用一个三元组 $G=(V,S,E)$ 来存储，其中，V 代表子图中的节点，S 表示每个节点的属性，E 是每个子图结构中的调用关系，即边。

3）存储所有的三元组存储，在比较 2 个结构子图的相似性时即可转化为比较相同的三元组结构，在此比较过程中，不仅比较节点的 id、节点之间的调用关系，还比较函数中代码块的操作码，这具有一定的抗混淆的作用，因为即使包名被混淆，但是为了代码能正常运行，函数中的操作码不会被混淆，所以利用该方法，可以提高检测率，然后根据相同结构的个数，利用式（1）计算函数调用图的相似度。

$$Sim = len(s)/(len(a) + len(b) - len(s)) \quad (1)$$

其中，a 表示应用程序 a 的函数调用图中 motif 子图结构，b 表示应用程序 b 的函数调用图中 motifs 子图结构，s 表示 2 个应用程序中相同的子图结构模式，$len(a)$ 表示应用程序 a 中所有子图结构模式的个数，$len(b)$ 表示应用程序 b 中所有子图结构模式的个数，$len(s)$ 表示 2 个应用程序中相同的子图结构模式的个数。

4）根据相似度判断是否为重打包关系。设定一个阈值，若相似度超过该阈值，即可判定为重打包应用；否则，则不是重打包应用。根据实验结果分析，本文采用的阈值是 0.7。

4 实验及结果分析

4.1 实验环境

本文实验在生成函数调用图时，主要用到开源工具 Apktool。Apktool 是对应用程序进行反编译，得到生成图所需要的 Smali 代码和资源文件，函数调用图主要是用 Java 代码编写的程序生成。函数调用图处理过程中，第三方库白名单主要参考实验室和 AppBrain[13-14] 统计的数据，同时对国内第三方市场中应用使用广告库的情况进行统计分析，不断完善白名单；和界面相关的 API 列表是通过搜集和分析官方安卓 API 文档中和界面相关的 API 得到，这一阶段的工作主要通过 python 编写的脚本对图进行处理。函数调用图相似度比较过程，利用 python 代码来实现，比较每个函数调用图中的 3 种子图结构，来检测两图是否相似，其中每个函数调用图的子图结构特征用一个三元结构保存在文本文档中，相同子图的个数是通过相同的三元组结构确定，所有应用程序的子图结构只需生成一次。

4.2 实验结果

4.2.1 实验数据集

实验共测试 1 630 个 Android 应用，这些 Android 应用分成 2 个部分：第一部分为 130 个应用，这130 个应用中一部分是人工重打包应用得到，重打包的应用主要是通过反编译下载的应用，并对应用做一些简单的修改，比如添加删除第三方库、更改变量或函数名、增加删除一些代码等，最后将应用重新签名并打包；另一部分是从应用市场中手动下载，主要从国内第三方市场以及各种 Android 论坛下载可能为重打包的应用（可以通过名字或者描述来判断，比如破解版本、修改版本等），由于谷歌官方市场的限制，根据国内 21 个安卓应用市场的排名[15]，UC 应用市场信任度排名第一，因此从 UC 应用市场寻找可能对应的原版应用。安卓应用程序的大小一般分布在 50 KB ~ 50 MB，这 130 个应用的大小为 50 KB ~ 30 MB，基本上可以代表一般应用程序的大小，此部分应用主要是为了证明该方法的正确性。第二部分的 1 500 个应用，是从国内外市场和各论坛中下载的应用，下载过后通过 Virustotal 扫描之后确认为恶意的应用。恶意应用不同家族应用数目分布情况如表 1所示。

表 1 恶意应用分类

家族	应用数量
Plankton	240
DroidKungFu	230
Penetho	150
Opfake	250
FakeRun	130
FakeInst	300
GinMaster	200
总计	1 500

4.2.2 检测结果

实验对 1 500 个恶意应用程序生成的函数调用图的大小进行了统计，大部分应用的函数调用

图的大小在 500 KB ~ 2 000 KB 之间。生成这些应用程序的函数调用图,由于在比较 2 个应用程序的相似度时,是通过比较 2 个应用生成的函数调用图中相同子图的个数来确定,比较里边的第三方库不仅耗时而且还会影响比较的准确性,又因为由应用程序生成的函数调用图比较大,里边第三方库也占据一定的比例,所以在比较之前,需要对图进行处理,提高比较的效率和准确性。因此,通过对函数调用图进行第三方库和与界面相关的 API 对图进行过滤。经过过滤之后,函数调用图的大小显著变小。

应用程序是两两之间比较的,130 个应用程序,共需要比较 8 385 个应用对。通过对这 130 个应用程序进行安装运行,并手动检查这些应用程序的代码,实验结果显示,这些应用中共有 51 对应用为重打包关系,用本文方法共检测出 48 对重打包关系的应用程序,漏报率为 5.86% ,误报率为 1.96% ,而利用文献[16]方法对第三方市场中不同包名的重打包应用进行检测时,检测出 38 对重打包应用,最后经过人工确认只有 34 个重打包应用,错报率为 10.5% ,这是因为此方法在进行相似度比较时,只比较的是包名,当对包名进行混淆处理后,就会造成错报率较高。本文方法在比较时,不仅比较包名,还比较函数中的操作码,具有移动的抗混淆能力,所以,错报率比较低。

误报率是非重打包应用被误判为重打包应用占的百分比,漏报率是本身是重打包应用但是被判定为非重打包应用占的百分比。通过对漏报和误报应用对的分析,得知误报的那对应用程序是一个小的浏览器程序,因为他们使用了相同的框架,造成了绝大部分代码都是相同的,在过滤第三方库时,由于第三方库白名单中没有该框架,因此造成了该应用对的误判。之后,在白名单中添加该框架的名字,更新第三库白名单,重新对此应用对进行检测,完全可以检测出来,所以,需要不断的完善和更新第三方库白名单。对于漏报的应用,是因为对第三方库进行了混淆,第三方库的代码占的比例比较大,因为混淆过了,在过滤时未能将其过滤掉,所以在计算相似度的值会比较小,从而被漏判定为非重打包应用。

在对 1 500 个恶意应用进行检测时发现,对于恶意应用的重打包现象比较严重,共检测到重打包应用 672 个,所占百分比为 44.8% ,其中存在多个互为重打包应用的现象。检测的这几个家族中,Opfake 家族中的重打包应用最多,共有 111 个,占整个应用数量的 7.4% ,Penetho 家族中检测到的重打包应用最少,为 65 个,占 4.35% 。恶意应用检测的结果如表 2所示。

表 2　恶意应用检测结果

家族	恶意应用数量	所占比例/%
Plankton	98	6.6
DroidKungFu	78	5.2
Penetho	65	4.3
Opfake	111	7.4
FakeRun	88	5.9
FakeInst	120	8.0
GinMaster	102	6.8
总计	672	44.8

5　结束语

本文提出一种新的 Android 重打包应用检测方法,根据 Smali 文件生成应用的函数调用图,对函数调用图进行第三方库和 API 的过滤,并利用 Motif 子图结构表示函数调用图,然后比较子图结构的相似度判断 2 个应用之间的重打包关系。该方法在检测应用时,由于对结点添加了属性,比较的是函数中的每个操作码,因此准确性较高,并且具有一定的抗混淆性。通过对 1 680 个应用的检测,验证了本文方法具有较好的检测率和可扩展性。由于本文使用静态检测的方法,具有一定的局限性,因此下一步将结合动态检测方法进行重打包应用的检测。

参考文献

[1] 360 互联网安全中心. 2015 年第三季度中国手机安全状况报告[EB/OL]. (2015-10-22). http://zt.360.cn/1101061855.php?dtid=1101061451&did=1101460794.

[2] ZHOU Yajin, JIANG Xuxian. Dissecting Android Malware: Characterization and Evolution[C]//Proceedings of 2012 IEEE Symposium on Security and Privacy. Washington D.C., USA: IEEE Press, 2012: 95-109.

[3] HANNA S, HUANG Ling, WU E, et al. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications[M]//FLEGEL U, MARKATOS E, ROBERTSON W. Detection of Intrusions and Malware, and Vulnerability Assessment. Berlin, Germany: Springer, 2013: 62-81.

[4] MYLES G, COLLBERG C. K-gram Based Software Birthmarks[C]//Proceedings of 2005 ACM Symposium on Applied Computing. New York, USA: ACM Press, 2005: 314-318.

[5] JANG J, BRUMLEY D, VENKATARAMAN S. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis[C]//Proceedings of the 18th ACM Conference on Computer and Communications Security. New York, USA: ACM Press, 2011: 309-320.

[6] SUN Xin, ZHONGYANG Yibing, XIN Zhi, et al. Detecting Code Reuse in Android Applications Using Component-based Control Flow Graph[C]//Proceedings of 2014 IFIP International Information Security Conference. Berlin, Germany: Springer, 2014: 142-155.

(下转第 139 页)

[3] 杨　剑,王　珏,钟　宁. 流形上的 Laplacian 半监督回归[J]. 计算机研究与发展,2007,44(7):1121-1127.
[4] DENG C, HE Xiaofei, HAN Jiawei, et al. Graph Regularized Nonnegative Matrix Factorization for Data Representation [J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2011, 33 (8): 1548-1560.
[5] KIM J, RENATO D C M, HAESUN P. Group Sparsity in Nonnegative Matrix Factorization [C]//Proceedings of SIAM International Conference on Data Mining. Anaheim, USA: Society for Industrial and Applied Mathematics, 2012: 851-862.
[6] ZEYNEP A, CHRISTIAN T, CHRISTIAN B. Non-negative Matrix Factorization in Multimodality Data for Segmentation and Label Prediction[C]//Proceedings of the 16th Computer Vision Winter Workshop. Mitterberg, Austria: 2011: 27-34.
[7] LIU Jialu, WANG Chi, GAO Jing, et al. Multi-view Clustering via Joint Nonnegative Matrix Factorization[C]// Proceedings of SIAM International Conference on Data Mining. Austin, USA: Society for Industrial and Applied Mathematics, 2013: 252-260.
[8] 卢桂馥,万鸣华. Hessian 正则化的低秩矩阵分解算法[J]. 小型微型计算机系统, 2016, 37 (10): 2296-2299.
[9] 刘红丽,刘伟锋,王延江,等. Hessian 正则化 Logistic 回归模型[J]. 计算机工程与应用, 2016, 52 (5): 236-240.
[10] DAVID L D, CARRIE G. Hessian Eigenmaps: New Locally Linear Embedding Techniques for High-dimensional Data[J]. Proceedings of National Academy of Sciences of the United States of America, 2003, 100(10): 5591-5596.
[11] KWANG I K, FLORIAN S, MATTHIAS H. Semi-supervised Regression Using Hessian Energy with an Application to Semi-supervised Dimensionality Reduction [C]//Proceedings of Advances in Neural Information Processing Systems. Vancouver, Canada: DBLP, 2009: 979-987.
[12] TAO Dapeng, JIN Lianwen, LIU Weifeng, et al. Hessian Regularized Support Vector Machines for Mobile Image Annotation on the Cloud [J]. IEEE Transactions on Image Processing, 2013, 15(4): 833-844.
[13] LIU Weifeng, TAO Dacheng. Multiview Hessian Regularization for Image Annotation [J]. IEEE Transactions on Image Processing, 2013, 22 (7): 2676-2687.
[14] LIU Weifeng, MA Tengzhou, TAO Dapeng, et al. HSAE: A Hessian Regularized Sparse Auto-encoders [J]. Neurocomputing, 2016, 187: 59-65.
[15] PENTTI P, UNTO T. Positive Matrix Factorization: A Nonnegative Factor Model with Optimal Utilization of Error Estimates of Data Values [J]. Environmetrics, 1994, 5(2): 111-126.
[16] STEFAN W, WRITTEN S W, JAMES C, et al. Seeding Non-negative Matrix Factorizations with the Spherical K-means [EB/OL]. (2010-11-21). https://www.researchgate.net/publication/247469258_Seeding_NonNegative_Matrix_Factorizations_with_the_Spherical_K-Means_Clustering.
[17] PRITHVIRAJ S, GALILEO N, GALILEO N, et al. Collective Classification in Network Data [J]. AI Magazine, 2008, 29: 93-106.
[18] LI Yifeng, ALIOUNE N. The Non-negative Matrix Factorization Toolbox for Biological Data Mining[J]. Source Code for Biology and Medicine, 2013, 8(10): 1-26.

编辑　刘　冰

(上接第 127 页)

[7] ZHANG Fangfang, HUANG Heqing, ZHU Sencun, et al. ViewDroid: Towards Obfuscation-resilient Mobile Application Repackaging Detection [C]//Proceedings of 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks. New York, USA: ACM Press, 2014: 25-36.
[8] CRUSSELL J, GIBLER C, CHEN Hao. Attack of the Clones: Detecting Cloned Applications on Android Markets [M]//FORESTI S, YUNG M, MARTINELLI F. Computer Security-ESORICS 2012. Berlin, Germany: Springer, 2012: 37-54.
[9] MILO R, SHENORR S, ITZKOVITZ S, et al. Network Motifs: Simple Building Blocks of Complex Networks[J]. Science, 2002, 298(5594): 824-827.
[10] VALVERDE S, SOLÉ R V. Network Motifs in Computational Graphs: A Case Study in Software Architecture[J]. Physical Review E, 2005, 72(2).
[11] 杨广亮,龚晓锐,姚　刚,等. 一个面向 Android 的隐私泄露检测系统[J]. 计算机工程, 2012, 38(23): 1-6.
[12] 陈　凯,王　鹏,LEE Y,等. 面向海量软件的未知恶意代码检测方法[J]. 信息安全学报, 2016, 1(1): 24-38.
[13] ZHOU Wu, ZHOU Yajin, JIANG Xuxian, et al. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces[C]//Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy. New York, USA: ACM Press, 2012: 317-326.
[14] AppBrain[EB/OL]. [2016-06-01]. http://www.appbrain.com/stats.
[15] NG Y Y, ZHOU Hucheng, JI Zhiyuan, et al. Which Android App Store Can Be Trusted in China? [C]// Proceedings of the 38th Annual Computer Software and Applications Conference. Washington D. C., USA: IEEE Press, 2014: 509-518.
[16] 吴雪平,张大方,苏　欣,等. 基于流量相似度的 Android 二次打包应用的检测技术研究[J]. 小型微型计算机系统, 2015, 36(5): 954-958.

编辑　金胡考