中国科学: 信息科学 2014 年 第 44 卷 第 1 期: 142–157
www.scichina.com info.scichina.com
《中国科学》杂志社
SCIENCE CHINA PRESS
软件维护与演化技术专刊 · 论文

基于代码克隆检测技术的 Android 应用重打包检测

王浩宇, 王仲禹, 郭耀*, 陈向群

高可信软件技术教育部重点实验室, 北京大学信息科学技术学院软件所, 北京 100871
* 通信作者. E-mail: yaoguo@sei.pku.edu.cn

收稿日期: 2013–08–13; 接受日期: 2013–11–25
国家重点基础研究发展计划 (批准号: 2011CB302604)、国家高技术研究发展计划 (批准号: 2011AA01A202) 和国家自然科学基金 (批准号: 61103026, 61121063) 资助项目

摘要 随着智能移动设备的流行和普及, 移动应用得到了飞速发展. 这些移动应用带来了丰富的功能和友好的用户体验, 同时也带来一些安全和隐私问题. 恶意的开发者或者剽窃者可以破解已经发布的应用程序, 并且在应用中植入恶意代码或者替换掉原有的广告库之后, 重新打包应用程序并发布在应用市场中. 重打包的应用不仅侵害了原开发者的知识版权和利益, 同时也对移动用户的安全和隐私造成危害. 本文提出了一种基于代码克隆检测技术的 Android 应用重打包检测方法并且实现了该方法的一个原型系统. 实验证明本系统具有很好的准确性和可扩展性, 能够用于应用市场级别的大规模应用重打包检测.

关键词 智能手机 重打包 移动应用 代码克隆 安全 隐私

1 引言

近年来, 移动设备 (例如智能手机和平板电脑) 的发展十分迅速. Android 平台占据了智能移动市场的主要份额, 据统计每天有超过 130 万部搭载 Android 系统的移动设备被激活使用 [1)]. 随着移动设备的流行和普及, 涌现出大量的移动应用. 截止至 2013 年 2 月, 谷歌官方市场 [2)] 中已经有超过 80 万个 Android 移动应用. 这些移动应用不仅增强了移动设备的功能, 还大大丰富了用户体验. 用户逐渐习惯于使用多样的应用来娱乐和办公, 智能移动设备和移动应用已经成为人们生活中不可或缺的一部分.

由于 Android 系统的开放性, 用户不仅能从谷歌官方应用市场下载和安装应用, 也可以从任意的第三方应用市场, 甚至网站和论坛下载和安装应用. 同时, 应用的开发者可以将应用提交到任意的第三方市场来供用户下载. 因此对于应用市场的管理者来说, 只有管理好市场中应用的质量和提供一个良好的市场环境, 才能吸引更多的用户和开发者.

然而, Android 应用很容易被破解, 目前有很多开源的反编译工具可以使用. 因此, 一些恶意的开发者可以很容易的破解应用市场中的合法应用, 修改代码后重新打包并在市场中发布. 付费的应用可以被破解然后免费发布出去, 恶意的开发者也可以将原应用中的广告库替换掉来谋取利益. Gibler 等

1) Http://bgr.com/2013/03/13/android-activation-growth-analysis-373572/.
2) Google Play, http://en.wikipedia.org/wiki/Google_Play.

引用格式: 王浩宇, 王仲禹, 郭耀, 等. 基于代码克隆检测技术的 Android 应用重打包检测. 中国科学: 信息科学, 2014, 44: 142–157, doi: 10.1360/N112013-00130

人 [1] 的研究表明, 重打包应用会给原应用的开发者造成平均 14% 的广告收入的减少. 更为严重的是, 恶意的开发者可以将恶意的代码植入到合法的应用中然后发布出去, 以此来感染更多的用户. 根据网秦在 2013 年 3 月发布的移动安全报告, 应用重打包已经成为恶意软件传播的主要途径之一. 应用重打包的行为不仅侵犯了开发者的利益, 也严重威胁到了用户的安全和隐私.

应用市场的管理者需要控制市场中应用的质量, 检测和移除这些潜在的威胁. 然而, 在应用市场中检测重打包应用是很困难的. 一方面, 管理者大多数时候只能通过手动比较来判断应用是否是重打包应用, 并且很多时候手动比较也很难得到正确的结果. 例如, 重打包的应用可以在功能上包含几个不同的应用, 或者重打包的应用包含恶意软件等等. 另一方面, 考虑到应用市场中海量数目的应用, 手动进行重打包检测不可靠且没有可扩展性. 因此, 在应用市场级别的应用重打包检测需要一个自动化的系统来完成.

实现这样的自动化系统存在着很多的困难和挑战, 尤其是考虑到 Android 市场的开放性和复杂性. 首先, 系统必须要保证准确性, 这就需要提出的检测方法有低误报率和高查全率. 应该考虑到不同层面上的代码更改, 添加和删除. 因为重打包应用的修改可以在 Dalvik 字节码上面进行, 也可以在 Smali 代码上修改, 甚至能够获取源代码的话也可以在 Java 代码层面上进行修改. 其次, 系统必须要有可扩展性，能够快速在海量应用中检测到重打包应用. 现在的应用市场都有数十万数量级的应用, 系统需要保证在有新应用添加时能够增量式的快速检测新应用是否是重打包应用. 再次, Android 应用的一些特性也给重打包检测增大了难度, 比如 Android 应用中大量使用的第三方库以及代码混淆等特性.

本文提出了一种基于代码克隆检测技术的移动应用重打包检测方法, 并且实现了该方法的一个原型系统. 这种检测方法利用基于计数的方法来计算代码块中每个变量的特征计数向量, 计算代码块的特征计数矩阵, 从而将每个应用程序由一系列的特征计数矩阵表示. 两个应用程序之间的相似度可以通过比较它们的特征计数矩阵中相似的比例来进行计算. 论文使用原型系统测试了来自谷歌官方市场以及各种第三方应用市场的超过 14 500 个应用, 比较了超过 13 500 000 个应用对. 实验结果验证了该方法有很好的准确性和可扩展性.

本文主要有以下贡献:

• 使用了基于计数的代码克隆检测技术来检测重打包应用. 该方法有着很好的准确性和扩展性, 并且对于不同层次的代码修改都很有效.

• 对使用的基于计数的代码克隆检测技术进行了修改和优化, 使其更符合 Android 应用程序的特性, 以及更适合移动应用重打包检测. 在保持检测准确性的同时缩短了比较时间, 使得检测速度有很大的提高.

• 研究了目前 Android 平台中第三方库的使用情况, 建立了一个相对完善的第三方库白名单, 通过该白名单过滤第三方库造成的干扰并且缩短比较时间.

• 实现了一套自动化的原型系统, 从应用程序预处理到最后的应用程序相似度比较都能完全自动化执行. 从谷歌官方市场以及各种第三方市场收集了超过 14 500 个应用程序, 对原型系统进行了大规模的实验验证, 并且根据检验结果总结分析了应用重打包的原因.

2 背景

2.1 Android 应用程序

Android 应用一般是由开发者将所有的源码和资源打包成 APK (Android package) 文件, 然后发

布供用户下载. APK 文件是一个压缩包, 解压缩之后的文件夹内主要包含该应用的 Dalvik 字节码、UI 资源、UI 布局、配置文件以及签名信息等. Android 应用程序通常用 Java 语言实现, 然后被编译成 Dalvik 字节码. Android 应用程序同样可以包含本地代码 (native code), 但是本地代码不在本文的考虑范围, 因为很少有应用程序包含本地代码, 并且对于应用重打包, 恶意的开发者也很少会去改本地代码. Android 应用的发布是需要签名的, Android 应用的签名是由 Android 应用的开发者在发布前打包时通过 Android SDK 进行签名的, 每一个开发者拥有自己的一个签名密钥, 对于自己开发的各个应用都可以应用同一个签名. 由于密钥颁发机制, 不同的开发者对于应用的签名不同.

相比其他平台的应用程序, Android 应用程序有着一些特性: (1) 与其他桌面应用相比, Android 应用有很多的入口点. Android 应用由很多组件构成, 其中每个组件都有自己的入口, 组件之间通过 intent 进行通信. (2) 大部分的 Android 应用程序都使用了第三方库, 比如广告库 (例如 Admob)、社交网络库 (例如 Facebook) 和开发工具库 (例如 Google Analytics), 这些外部的库文件占据了代码的很大一部分比重. (3) 很多 Android 应用程序在发布之前都经过了代码混淆. 代码混淆的目的是为了增大逆向工程的难度. Proguard[3)] 是集成在 Android 系统中的一个工具, 开发者被鼓励在发布应用之前通过 Proguard 进行代码混淆. Proguard 会通过更改类名, 方法名, 去掉无用的代码等这样的做法来进行混淆. Android 的这些特性都增加了重打包检测的难度, 因此对 Android 应用进行程序分析和检查重打包的时候必须要考虑到这些问题.

2.2 应用重打包

本文中所提及的应用重打包是指一个应用的非授权开发者将该应用的非核心代码增删改或不进行任何改动, 并由非授权密钥签名打包的行为. 应用重打包后的应用称为重打包应用. 重打包应用包含两个特征: (1) 重打包应用与原应用核心代码相似, 核心代码是指去除第三方库等外部通用代码之后的部分. 由于重打包应用需要保持原应用的功能, 因此对于核心代码不会做很大的改动. (2) 重打包应用的签名与原应用签名不同, 签名代表着应用的作者信息. 破解一个应用之后, 必然要对应用重新签名, 而这个签名一般无法与原签名保持一致. 本文中假设开发者的签名是唯一的, 不能被别人获取. 同一个应用的不同版本之间由于签名一致所以不能算是应用的重打包. 本文提出的检测方法, 主要根据重打包应用的这两个特征进行检测.

3 应用重打包检测机制

如图 1 所示, 应用重打包的检测过程包括应用的预处理阶段, 特征提取阶段和相似度分析阶段. 预处理阶段对应用程序进行反编译和过滤, 得到应用程序的核心代码. 然后在特征提取阶段, 使用基于计数的代码克隆检测技术, 计算出每个变量的特征计数向量, 得到每个代码块的特征计数矩阵, 从而将每个应用程序由一系列的特征计数矩阵来表示. 在相似度分析阶段, 通过比较应用程序之间特征计数矩阵的相似度来判断它们是否是重打包关系.

3.1 应用预处理

本文中的方法工作在 Java 代码层, 因此应用预处理这一阶段主要是将应用程序反编译得到 Java 代码, 获取应用程序的签名, 以及经过过滤得到应用的核心代码.

3) Proguard, http://developer.android.com/tools/help/proguard.html.

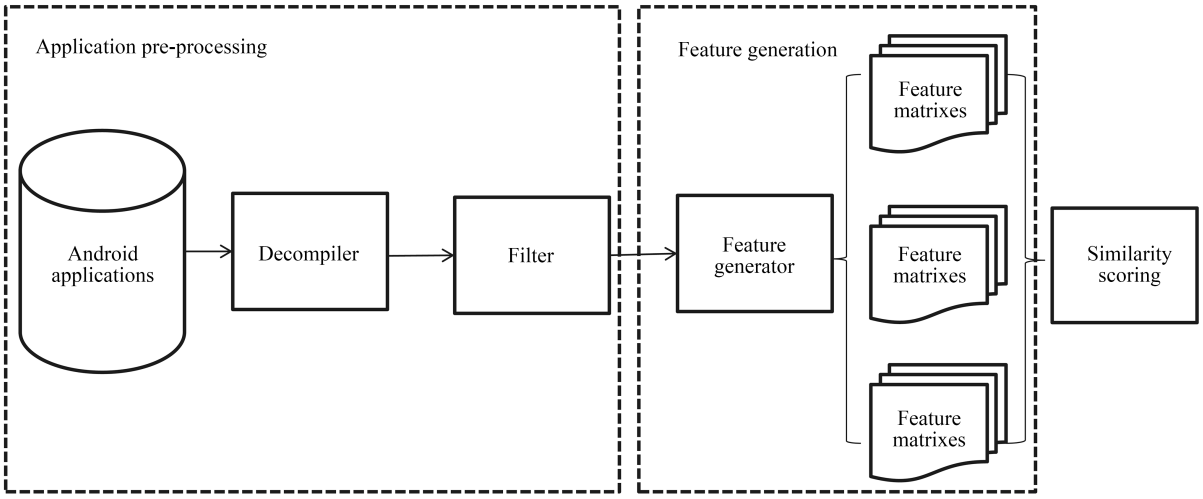

图 1 应用重打包检测系统的工作流程图

Figure 1 The workflow of repackaging detection system

本文将每个应用的代码从 DEX 格式转为 JAR 包, 然后使用 Java 反编译工具来得到 Java 代码. 尽管从 DEX 到 Java 的转换过程不是可逆的, 但是根据之前的研究 [2], 从 DEX 转换到 Java 代码可以达到 95% 的成功转换率, 足够用来做应用重打包检测.

如上所述, 很多 Android 应用都使用了第三方库, 包括广告库, 社交网络库和开发库. 为了得到应用的核心代码, 必须要过滤掉这些第三方库. 这些外部的库不仅会影响重打包检测的速度, 同时也会影响检测的准确度. 这些外部的库影响检测的准确度主要体现在两个方面: (1) 两个应用不是重打包关系, 却因为使用了同样的 Android 第三方库, 而被误判成重打包关系. 这种情况是由于非重打包应用可能使用了相同的第三方库, 导致代码相似比例过大. 例如, 应用 A 和 B 是完全不同的应用, 但是由于应用 A 和 B 都使用了共同的广告库 L_1, 而 L_1 的代码数量很大, 甚至远超应用 A 和 B 本身核心代码的大小. 这就导致应用 A 和 B 检测出来的相似代码的比例很大, 它们就会被误判为是重打包关系. (2) 两个应用是重打包关系, 却因为使用了不同的 Android 第三方库, 而被漏判. 这种情况是由于重打包应用使用了不同的第三方库, 导致代码相似比例过小. 例如, 应用 B 是 A 的重打包应用, 但是由于应用 B 去掉了应用 A 原本使用的广告库, 并且大量添加了其他的广告库, 这样就会导致应用 B 和 A 之间代码相似的比例过小, 可能会在重打包检测中被漏判.

本文调研了 Android 应用中第三方库的使用情况, 发现第三方库的数量其实是相当有限的. 因此本文建立了一个第三方库的白名单, 大约包括 100 多个第三方库. 通过建立的第三方库白名单中的包名称来过滤第三方库. 经过过滤后的代码被认为是一个应用的核心代码, 用作下一阶段的特征提取.

3.2 特征提取

重打包检测技术的关键是对应用程序提取特征, 产生正确的抽象描述. 提取特征之后, 就可以使用各种比较技术来比较应用之间的相似度. 在本文中, 应用的特征提取使用了基于计数的代码克隆检测技术, 是本文作者在已有的代码克隆检测工作基础上进行优化修改 [3], 使其更符合 Android 应用的特性. 将每个变量在不同计数环境下出现的次数作为特征, 为每个变量计算得到一个特征计数向量,

对于每个代码片段计算得到一个特征计数矩阵. 每个应用程序的特征由一系列的特征计数矩阵表示, 从而两个应用程序之间的相似度就是比较它们的特征计数矩阵中有多少是相似的.

计数环境. 计数环境用来描述代码片段中变量的行为特征, 通过统计变量在不同计数环境下的次数可以得到变量的特征向量. 本文将计数环境分为 3 个阶段, 每个阶段中的计数环境对变量提供不同的特征描述. 第一阶段的计数环境是简单计数, 包括变量的使用, 以及变量的定义. 这一阶段的计数环境比较容易计算, 直接从代码块中查找变量并且加上简单的分析就可以得到. 变量出现在 "=" (或者 "+ =", "− =") 等符号的左边都被认为是变量被定义. 第二阶段的计数环境是语句中计数, 这一阶段的计数环境体现了变量所在语句的语义信息. 例如, 如果一条语句以 "if" 开始, 那么这是一个条件判断语句, 这条语句中的变量在该计数环境下面的值会增加 1. 第二阶段中的计数环境包括在条件判断语句中出现, 在加/减运算中出现, 在乘/除运算中出现, 作为数组下标, 被常量表达式定义. 第三阶段的计数环境是语句内计数, 这一阶段的计数环境需要多条语句的语义信息才能得到. 一个典型的例子是变量在嵌套循环中的层数, 需要判断变量在第一层循环出现, 在第二层循环出现, 还是出现在更深层次的循环中. 因此, 在这 3 个阶段中, 一共使用了 10 个不同的计数环境来描述变量的特征. 计数环境具有易扩展性, 在将来任意被定义好的计数环境都可以被加进来对变量的行为特征进行更细致的描述.

特征计数矩阵. 特征计数矩阵用来描述代码片段的特征. 对于每个变量, 通过计算都能得到它在 m 个计数环境下面的一个 m 维度的计数向量, 其中第 i 位代表变量在第 i 个计数环境特征下出现的次数. 对于包含 n 个变量的代码片段, 该方法计算出其中所有变量的计数向量, 可以得到一个 $n \times m$ 维的特征计数矩阵 **CM**, 将该矩阵作为代码片段的特征. 特征计数矩阵 **CM** 的计算复杂度为 $O(L + knm)$, 其中 L 代表代码块的长度, k 代表代码块中子块的数目. 在计算一个代码块的特征计数矩阵时, 首先计算该代码块中子块的特征计数矩阵, 然后将它们合并. 对于子块的特征矩阵合并不是简单的相加, 例如和循环相关的计数环境在合并特征矩阵时需要重新计算, 因为子块可能在父代码块的更深一层循环中. 除此之外, 如果在子块中有新声明的变量, 那么在合并之后 **CM** 的大小会增加.

保留字和符号的计数向量. 为了提高精确度, 该方法同时计算了保留字和符号的计数向量. 保留字和符号的名字都是唯一的, 因此可以直接将保留字和符号出现的次数作为特征. 例如 while 和 if 等保留字在代码片段中出现的次数作为代码段保留字的特征, 加/减号等符号在代码片段中出现的次数作为代码段符号的特征. 最后为保留字和符号分别生成一个特征计数向量.

3.3 相似度分析

为了分析应用程序之间的相似度, 需要计算两个应用程序之间相似代码块所占的比重. 判断两个代码块是否相似途径是比较两个代码块的特征计数矩阵. 而特征矩阵的比较是通过计算特征向量之间的相似度得到的. 整个相似度分析的过程如图 2 所示, 包括 3 个部分: 特征向量的相似度比较、代码块的相似度比较, 以及应用程序的相似度比较.

3.3.1 特征向量的相似度

变量的特征计数向量相似度可以通过计算两个特征向量的余弦相似度得到, 即两个向量的空间夹角的余弦数值. 对于任意的特征向量 a 和 b, 它们之间的余弦相似度定义为

$$\mathrm{CosSim} = \frac{a \cdot b}{\|a\| \, \|b\|} = \frac{\sum_{i=1}^{m} a_i \times b_i}{\sqrt{\sum_{i=1}^{m} {a_i}^2} \times \sqrt{\sum_{i=1}^{m} {b_i}^2}}.$$

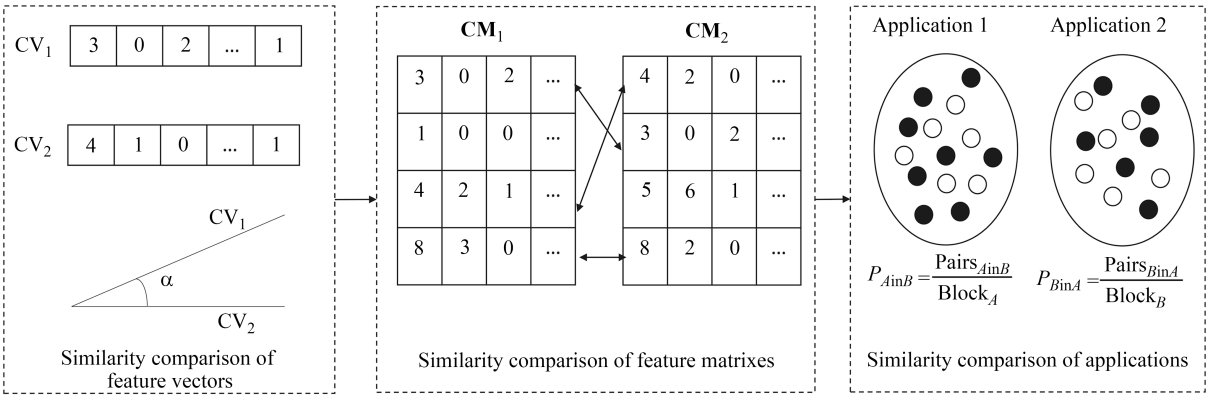

图 2 相似度比较的流程图

Figure 2 The workflow of similarity scoring

对于关键字和符号的特征向量相似度比较, 本文没有使用计算余弦相似度的方法, 而是通过一个改进的比例相似函数得到. 对于某个关键字或者符号, 它在两个向量中出现的次数分别为 a 和 b $(a \geqslant b)$, 那么它们的相似度定义为

$$\text{ProSim} = \frac{1}{a+1} + \frac{b}{a+1}.$$

关键字的个数为 k, 对于关键字特征向量 CVK_1, CVK_2, 它们的相似度为

$$\text{Sim}_k = \prod_{i=1}^{k} \frac{1+\min\left(\text{CVK}_1[i], \text{CVK}_2[i]\right)}{1+\max\left(\text{CVK}_1[i], \text{CVK}_2[i]\right)}.$$

符号的个数为 s, 对于符号特征向量 CVS_1, CVS_2, 它们的相似度为

$$Sim_s = \prod_{i=1}^{s} \frac{1+\min\left(\text{CVS}_1[i], \text{CVS}_2[i]\right)}{1+\max\left(\text{CVS}_1[i], \text{CVS}_2[i]\right)}.$$

3.3.2 代码块相似度

对于代码块 A 与代码块 B, 它们的特征计数矩阵分别为 $\mathbf{CM}_a$ 和 $\mathbf{CM}_b$. 首先需要对 $\mathbf{CM}_a$ 中的每个特征向量 CV_i 找到在 $\mathbf{CM}_b$ 中与之相似度最大的匹配对 Match_i. 最大相似度匹配对可以通过二分图匹配算法来查找, 但是算法的时间复杂度为 $O(n^3)$, 这对于应用中大量代码块的比较是不能接受的. 本文使用了一个快速并且准确的启发式算法.

对于代码块中的变量, 首先根据变量的使用频率对它们进行排序. 然后对于代码块 A 中的变量 a, 从代码块 B 中找到与 a 排序最接近的变量进行比较. 在该算法中允许重复匹配, 也就是说代码块 A 中的每个变量都能在代码块 B 中找到一个匹配对, 但是对于代码块 B 没有限制. 该算法大大减少了比较的复杂度. 根据实验结果, 该算法是快速并且准确的. 具体见算法 1.

对于每个变量找到相似度最大的匹配对之后, 代码块的相似度 $\text{Sim}_{\mathbf{CM}}$ 由变量匹配对相似度的乘积得到,

$$\text{Sim}_{\mathbf{CM}} = \prod_{i=1}^{n} \text{Sim}\left(\mathbf{CM}_A[i], \mathbf{CM}_B[\text{match}(i)]\right).$$

考虑到保留字计数向量相似度和符号计数向量相似度之后, 代码块的相似度为 $\text{Sim} = \text{Sim}_{\mathbf{CM}} \cdot \text{Sim}_K \cdot \text{Sim}_S$, 其中 Sim_K 和 Sim_S 分别代表保留字和符号的相似度.

最后, 需要计算一个阈值来判断两个代码块是否相似. 随着代码块特征矩阵维数的增加, 按照上述公式中计算得到的 Sim 值可能会越来越小, 因此在判断代码块相似的时候需要考虑到这一点. 计算阈值的公式为 $\text{balance} = \text{SimThres}^{n+k+s}$, 其中 k 代表保留字的个数, s 代表符号的个数, SimThres 为一个可以调节的常量. 在系统实现中, 通过调节 SimThres 的值发现, 将其设置成 0.95 是比较合理的. 随着维度的增加, 阈值也相应的降低. 若 Sim>balance, 那么认为这两个代码块是相似的.

算法 1 Find matching pairs in block b for variables in block a

```
1: function FindMatching(a)
2:     for i = 1 to variableTotal
3:         best ← 0
4:         for j = i − range to i + range do
5:             if (validNum(j)) ⋀(VecSim(CM[a][i], CM[b][j]) > best) then
6:                 best ← VecSim(CM[a][i], CM[b][j])
7:                 match(i) = j
8:             end if
9:         end for
10:    end for
11: end function
```

3.3.3 应用程序相似度

两个应用中代码块相似的比例代表着应用程序的相似度, 因此在比较应用程序相似度的时候, 需要两两比较两个应用中的代码块. 对于两个应用程序 A 和 B, 在计算它们相似程度的时候, 考虑两个值 $P_{A\text{in}B}$ 和 $P_{B\text{in}A}$, 分别代表应用 A 代码块在应用 B 中被克隆的比例, 以及应用 B 中代码块在应用 A 中被克隆的比例. 取其中较大者作为判断这个应用对之间是否是重打包关系. 这样的做法主要是考虑到有些重打包应用可能添加大量的无用代码来干扰相似度的计算, 或者一个大的应用重打包进去很多小应用的功能. 在这种重打包情况下, 由于原应用的核心代码没有发生改变, 这两个值中至少有一个应该比较大. 设定一个阈值 P_{thres}, 当 $P_{A\text{in}B} > P_{\text{thres}}$ 或者 $P_{B\text{in}A} > P_{\text{thres}}$ 时, 则判断出应用 A 和应用 B 是重打包关系. 通过实验发现, 将 P_{thres} 设置为 60% 是比较合理的值.

3.4 优化技术

特征提取与相似度计算相分离. 为了系统的可扩展性, 每次计算完应用的特征矩阵之后, 都将应用的特征保存在文件中, 以便后续比较使用. 虽然每个应用要与多个应用进行相似度比较, 但是应用的特征矩阵只需计算一次. 考虑到应用在大规模应用市场的场景, 市场中现有的应用都可以先计算好特征矩阵保存起来, 每次市场中有新添加的应用时, 将新应用的特征与已有应用的特征进行比较即可. 这项优化技术减少了特征提取的冗余计算, 使得系统在实际应用的时候具有很好的扩展性.

代码块的过滤. 代码块的数量影响着比较的时间和空间效率. 行数很少的代码块包含的信息很少, 对于这些代码块计算它们的特征计数矩阵意义不大. 因此, 本文在系统的实现中过滤掉行数不足 5 行的代码块. 实验结果表明, 代码块的过滤会在不影响比较准确性的情况下大大提高系统的性能.

代码块的比较优化. 代码块的两两比较是很耗时的过程, 严重影响比较的效率. 实验中发现, 在代码块过滤之前很多应用的代码块数量在 10000 以上, 那么两个这样的应用比较, 它们之间代码块的

两两比较会超过 1 亿次, 时间开销会很大. 即使对代码块进行了过滤, 也有很多应用的代码块数量在 1000 左右, 代码块两两之间都进行比较, 矩阵的相似度计算也需要一百万次以上. 实验中发现很多代码块的比较是无用的, 因为很多代码块差异很大, 明显是不相似的. 因此, 本文在实现中对每个代码块提取它的元信息, 包括代码块的行数, 变量的个数, 变量的最大使用次数, 关键字总共出现的次数等. 代码块的元信息代表代码块的总体属性, 如果两个代码块的元信息差别很大, 可以认为这两个代码块是不相似的, 因此也不需要对代码块的特征计数矩阵进行比较. 元信息属性的差异程度体现在两方面: 一方面是两个属性的数值差别比较大, 比如代码行数 20 行与代码行数 35 行相比; 另一方面是两个属性的比例差别比较大, 比如变量个数为 5 与变量个数为 10. 因此, 通过计算元信息中属性的数值差别与比例差别, 来判断是否需要进行更进一步的代码块比较. 代码块的元信息比较见算法 2.

算法 2 Optimization for code block comparison

```
1: function MetaFeatureCompare(a, b, attr, minSep): boolean
2:     for i = 1 to attributeSize
3:         if (Abs(attr[a][i] − attr[b][i]) > minOpt[i][0]) ⋁(attr[a][i]/attr[b][i] > minOpt[i][1]) ⋁(attr[b][i]/attr[a][i] >
           minOpt[i][1])
4:             return false
5:         end if
6:     end for
7:     return true
8: end function
```

延迟比较. 每个应用的特征计数矩阵保存在文件中, 在实验中发现从文件中读取保存的代码块的特征计数矩阵的时间占据了很大一部分比重. 因此, 本文对文件读写时间进行优化, 减少无用的读写计数矩阵时间. 在系统实现中, 将代码块的元信息和计数矩阵信息分开保存, 只有两个代码块的元信息接近的时候才读取计数矩阵信息. 实验结果表明, 延迟比较对于读写性能的提升效果是十分显著的.

4 实验与结果分析

本文实验系统在 Linux 平台实现, 并且添加了对 OS X 平台的支持. 在应用程序预处理模块, 使用了 3 个开源工具 Keytool[4], Dex2jar[5] 和 JD-Core-Java[6]. Keytool 被用来提取应用程序的签名信息. Dex2jar 用来将得到的 Dalvik 字节码文件反编译成 JAR 包. JD-Core-Java 用于分析和反编译 JAR 包. 其中, 系统对 JD-Core-Java 进行了改写, 使其可以在命令行批量执行并且添加了对于 OS X 平台的支持. 通过编写脚本, 将这些工具组合成一个自动化的工具链, 可以批量对应用程序进行预处理. 第三方库的收集参考了 AppBrain[7] 的统计数据, 分别收集了广告库、社交网络库、开发工具库 3 类第三方库的信息, 同时调研了国内 Android 应用常用的第三方库信息, 对第三方库的白名单进行完善.

在特征提取和相似度分析阶段, 主要使用 C++ 来实现. 应用的特征提取后保存在本地文件中, 所有应用的预处理和特征提取只需要进行一次.

4) Keytool, http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html.
5) Dex2jar, https://code.google.com/p/dex2jar/.
6) JD-Core-Java, https://github.com/nviennot/jd-core-java.
7) AppBrain, http://www.appbrain.com/.

4.1 实验环境与数据来源

本文共进行了两个阶段的实验, 第一阶段是小规模实验, 主要用来验证代码块过滤的效率以及本文方法的准确性; 第二阶段是大规模实验, 主要用来测试系统的可扩展性和性能.

在第一阶段共测试了 116 个 Android 应用. 这 116 个应用的来源分为两部分, 一部分是从应用市场中手动下载的应用, 另一部分是人工重打包应用而得到的. 实验室的志愿者从知名 Android 第三方应用市场以及各种 Android 论坛下载有可能是重打包的应用 (通过名字、描述来判断, 比如破解版、修改版等), 并且从谷歌官方市场寻找可能对应的原版应用. 另外, 实验室的一名志愿者从谷歌官方应用市场下载应用, 通过重打包的常用方式来反编译并且对应用做一些简单的修改 (更改变量和函数名, 添加删除代码和第三方库等), 最后将修改后的应用重新签名并且打包. 这 116 个应用的 APK 文件大小在 50 KB 到 28 MB 之间, 基本能代表 Android 市场中一般应用程序的大小. 所有 116 个 Android 应用在实验的时候被重新编号命名, 在实验之后手动的安装和检查应用代码来验证结果的准确性. 实验在一台四核心的苹果 MacBook Pro 笔记本电脑上面进行, 操作系统是 OS X 10.8.3.

在第二阶段, 使用大规模应用集进行测试. 实验数据的来源分为两部分, 一部分是谷歌官方应用市场中的应用, 另一部分是五个国内 Android 第三方市场中的应用. 谷歌官方应用市场中的应用包括 27 个大类中每个类排名前 500 的应用, 一共是 13 500 个应用. 五个国内 Android 第三方市场中的应用包括每个应用市场中最热门的前两百个应用, 一共 1000 个应用. 假定谷歌官方应用市场中全部都是原版应用, 即没有重打包的应用. 实验中将这 1000 个第三方市场中的应用与官方的 13 500 个应用分别进行相似度比较, 一共比较 13 500 000 个应用对. 实验运行在 10 台 64 核心的 linux 服务器, 每个机器节点同时开启 50 个进程并行的对应用对进行比较.

4.2 代码块的过滤

代码块的数量代表应用中特征矩阵的数量, 也代表应用程序之间相似性比较的复杂度. 第一阶段的实验统计了这 116 个应用中代码块的数量, 结果如图 3 所示. 横坐标代表应用中代码块的数量, 对应的柱状体表示代码块数在对应区间的应用所占的比例. 可以看到, 在保留所有代码块的情况下, 很多应用的代码块数量超过了 10 000. 这意味着对于这样的两个应用之间比较相似度, 需要超过 1 亿次的特征矩阵的比较, 会很影响系统的时间和空间效率. 如前所述, 行数很少的代码块本身包含的信息很少, 而且有可能实际有效的代码基本没有, 这对于检测应用的重打包毫无意义. 因此过滤掉行数小于等于 5 的代码块. 过滤代码块之后, 应用的代码块数量分布如图 3 所示, 可以看到代码块的数量大大减少.

尽管过滤掉行数很少的代码块进行优化, 但是剩余的代码块中很大一部分都属于使用的第三方库, 对于检测的准确率和效率都是一个干扰. 因此, 通过之前所建立的白名单来过滤第三方库, 过滤之后的不同代码块数量的应用分布如图 4 所示. 可以看到, 在过滤第三方库之后, 应用的代码块数显著的减少, 大部分应用的代码块数在 1000 块之内.

4.3 准确性分析

实验对这 116 个应用进行两两比较检测, 总共比较了 $C_{116}^2 = 6670$ 个应用对. 对于应用 A 和 B, 计算两个值 $P_{A\text{in}B}$ 和 $P_{B\text{in}A}$, 分别代表应用 A 中的代码块在应用 B 中被克隆的比例, 以及应用 B 中

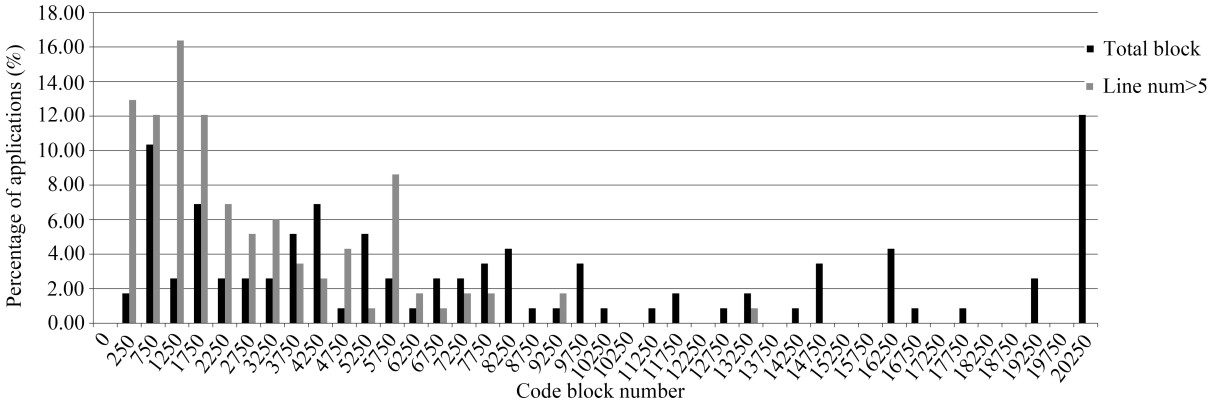

图 3 不同代码块数量的分布图

Figure 3 The distribution of different code block size

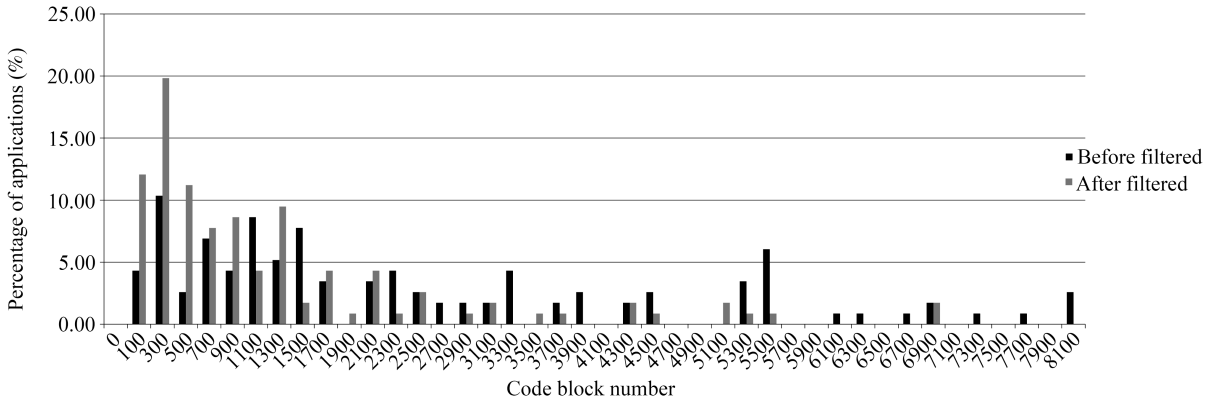

图 4 过滤第三方库前/后不同代码块数量的分布

Figure 4 The distribution of different code block size before/after libraries filtered

的代码块在应用 A 中被克隆的比例. 取其中较大者作为判断这个应用对之间是否是重打包关系. 这样的做法主要是考虑到有些重打包应用可能添加大量的无用代码来干扰相似度的计算. 由于重打包应用与原应用核心功能不变, 因此计算的两个值中至少有一个能够判断出应用重打包. 对 6670 个应用对计算出的相似度的分布如图 5 所示.

通过安装并运行这 116 个应用程序并且手动的检查代码, 实验发现在这 6670 对应用程序组合中有 43 个应用对是重打包关系, 其中有多个应用之间互为重打包关系. 根据已经计算好的应用对相似度的分布, 调节阈值大小, 检查在不同阈值下的误报率和漏报率, 结果如图 6 所示. 误报率指的是实际非应用重打包关系但是被判为应用重打包关系的应用对的百分比. 漏报率指的是实际应用重打包关系但是被判为非应用重打包关系的应用对的百分比. 可以看出, 随着阈值的增加, 漏报率在增加, 误报率在减少. 这个结果是符合预期的, 因为阈值的增加会使得一些计算出来克隆比例低于阈值的重打包应用被判为非重打包应用, 从而造成漏报率增加; 而阈值的增加会使得一些计算出来克隆比例高于先前阈值的被误判的重打包应用重新判为非重打包应用, 从而误报率会减少.

在阈值为 60% 的时候, 漏报率为 6.97%, 误报率为 2.33%, 43 对重打包关系的应用检测到了 40 对, 并且误报了一对不是重打包关系的应用. 检查误报和漏报的应用对, 本文通过检查代码来分析了误报

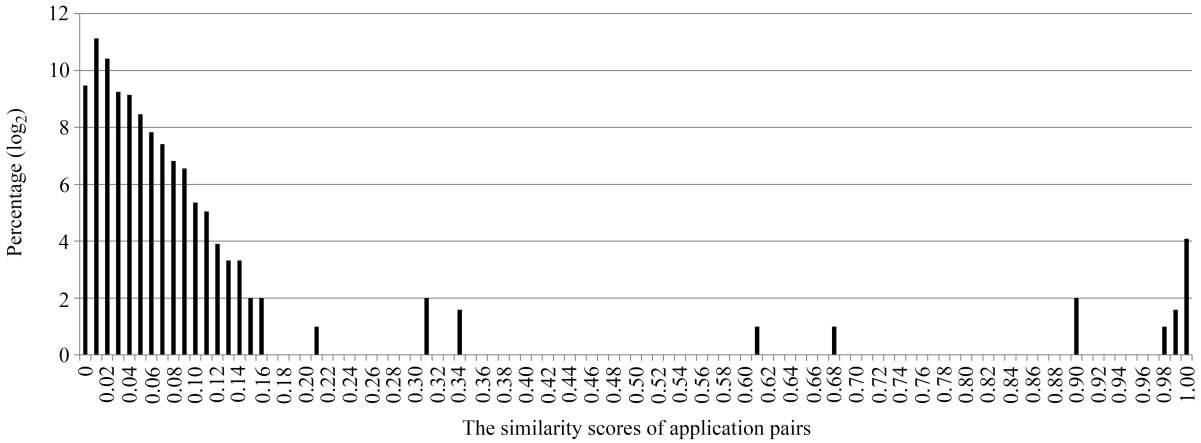

图 5　应用对相似度的分布

Figure 5　The distribution of different similarity scores

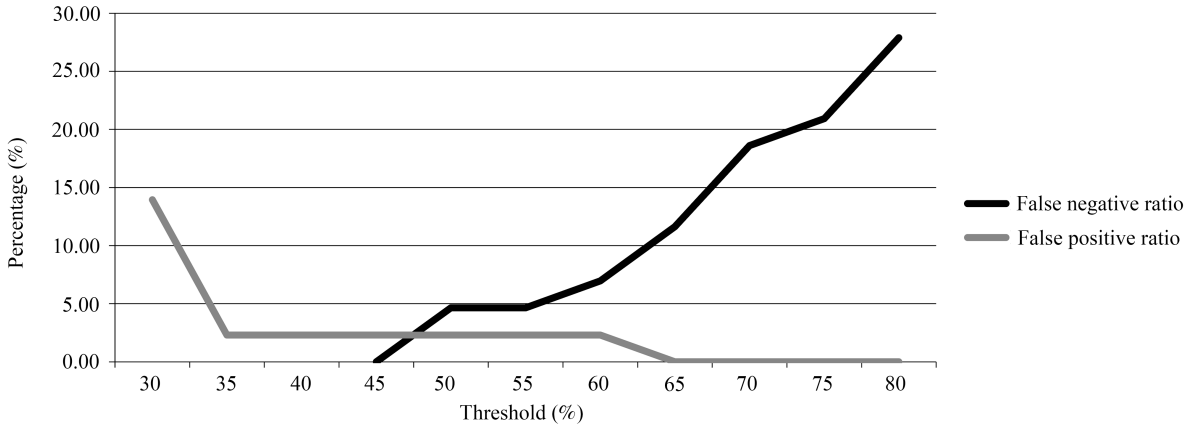

图 6　不同阈值下的误报率和漏报率

Figure 6　The false-positive and false-negative ratios in different thresholds

和漏报的原因: (1) 误报的应用是一对游戏, 它们都使用了相同的游戏引擎, 因此很大一部分代码是相同的. 过滤第三方库的白名单中没有这个游戏引擎库, 所以造成了这对应用的误判. (2) 漏报的三个应用对, 它们的代码中使用的第三方库的代码也被混淆了. 在本文系统中, 核心代码的混淆对于系统检测没有影响, 系统依然可以提取核心代码中的特征矩阵. 但是对于第三方库代码的混淆会造成通过包名来过滤第三方库的方法失效, 因此导致对计算应用程序的相似度造成干扰. 在对第三方库白名单进行完善之后, 误报率为 0, 即检测到的重打包应用全部是正确的.

4.4　可扩展性和效率分析

实验运行在 10 台 64 核心的 linux 服务器上面, 将第三方市场中的 1000 个应用分别与官方市场中的 13 500 个应用进行比较. 首先在这 10 台服务器上面计算这 14 500 个应用程序的特征信息 (特征矩阵, 签名信息等), 然后将官方市场中 13 500 个应用程序的特征信息平均分布在 10 台机器中, 同时将 1000 个待检测应用的特征信息副本在每台机器分别放置.

表 1 第三方应用市场中重打包应用的比例

Table 1 The percentage of repackaged application in third-party application markets

Application market	Sample applications	Detected repackaged applications	Percentage (%)
Market 1	Top 200 Apps	9	4.5
Market 2	Top 200 Apps	6	3.0
Market 3	Top 200 Apps	17	8.5
Market 4	Top 200 Apps	21	10.5
Market 5	Top 200 Apps	12	6.0

经统计, 在 10 台机器节点上面比较 13 500 000 个应用对总共耗时将近 3 个小时. 统计每台机器上面应用对的比较时间, 得到每台机器上平均每个进程比较一个应用对耗时 0.36 秒. 在 10 台机器上平均比较一个应用对需要 0.00072 秒. 结果表明, 本文系统在比较大规模应用程序的时候, 效率是完全可以接受的.

本文对 5 个第三方应用市场中的 1000 个应用的检测结果进行分析, 得到第三方市场中重打包应用的比例如表 1 所示. 结果表明, 对于这 5 个第三方应用市场的最热门的应用而言, 应用重打包的比例在 3.0%~10.5% 之间. 之所以关注最热门的应用, 主要是因为这些应用的下载量和流行度最高, 因此这些应用中的重打包应用对原应用开发者的利益和对用户的潜在危害更大. 结果表明, 第三方市场中重打包应用的比例还是很严重的. 相信实际的重打包应用的比例可能会比检测到的结果更高, 主要原因有两点: (1) 只检查了每个应用市场最热门的 200 个应用, 这些应用里面很大一部分都是官方应用, 排名较靠后的一些非官方应用重打包的可能性会更大一些. (2) 只是将第三方市场中的应用跟爬取的谷歌官方市场中的 13 500 个应用相比较. 在本文实验中, 官方市场中的应用爬取的不全面, 可能会导致一些重打包应用由于没有原版应用做比较而没有被检测出来.

4.5 重打包应用的分析

本文对检测出来的重打包应用, 安装应用并分析代码, 试图找出这些重打包应用背后的原因. 将检测到的重打包应用归为以下几类:

通过替换广告库来谋取利益. 很大一部分的重打包应用是通过替换掉原应用的广告库, 或者替换广告库中的客户标识 (Client ID) 来谋取利益. 目前 Android 应用的主要盈利方式是通过广告点击, 因此恶意开发者目的是通过这样的方式来获得收入. 这种行为造成了原应用的广告流量减少, 侵犯了原应用作者的知识产权并且导致收入的损失. 检测中发现, 有些重打包应用直接使用了跟原应用相同的名称, 用户很难区分是否是合法的应用. 而另外有些重打包应用直接换了名字和界面中的图片, 以另外一个新的应用身份出现.

汉化版和去广告版. 一些被检测出来的重打包应用自称是汉化版和去广告版, 但是通过检查代码发现很多汉化版和去广告版的应用中使用的广告库更多, 申请的权限也更多, 给用户带来更大的安全隐患. 这些应用是以汉化版和去广告版为噱头来吸引用户下载, 但是其实很多应用并不可靠.

功能包含. 实验中检测到一款被认为是重打包的应用是一个工具箱, 它包含了很多实用的功能. 经过的检测发现它将一些其他小应用的功能包含进来之后进行重打包. 比如检测结果显示一个测量海拔的小应用中超过 80% 的代码块都在这个工具箱中能找到相似的代码. 在实验中没有找到将病毒

或者恶意程序打包进去的重打包应用案例, 主要原因是, 一方面测试的第三方市场中应用数量不够多, 另一方面是还没有建立起一个病毒样本库来用来检测重打包.

5 相关问题及讨论

5.1 比较的粒度

本文系统在提取特征的时候, 是在代码块的粒度上来计算特征计数矩阵, 最后大部分应用的代码块数量在 1000 以内. 为了提升比较速度, 可以通过增大计数粒度的方式来减少特征计数矩阵的个数, 从而减少应用之间的比较时间. 根据不同的需求, 可以选择函数粒度, 类粒度, 甚至是源文件的粒度来提取特征. 本文系统可以很容易的修改来选择不同的比较粒度.

5.2 代码库的过滤

实验中发现, 系统在进行重打包检测的时候, 误报和漏报的原因基本上都是由于第三方库过滤的不完善导致的. 在第三方库完全过滤的情况下, 本文系统检测的准确性非常高. 为了完整的过滤第三方库文件, 一方面该系统需要不断的完善, 将查找到的第三方库补充添加进白名单. 另一方面, 也可以通过自动聚类的方式来过滤第三方库. 对于一个应用而言, 除了核心代码, 就是使用的第三方库文件. 可以对整个应用都提取特征, 包括核心代码和第三方库, 然后对这些特征进行聚类. 如果很多应用都使用了相同的第三方库, 那么对大量应用的特征进行聚类的时候, 这些相同的第三方库的特征就会被聚在一起. 假定使用相同的第三方库的应用数量远远大于同一应用的重打包版本数量, 那么就可以通过设定阈值来将第三方库文件过滤掉.

5.3 确定原版应用

本文系统能够找到有重打包关系的应用对, 但是很多情况下很难界定哪一个应用是原版应用而另一个是重打包应用. 之前的一些相关工作也对此进行了讨论, 提出了一些直观上的方法, 但是一直没有很好的解决办法. 一种方法是按照应用提交的时间来确定原版应用. 但是由于第三方市场很多, 并且应用的版本很多, 除非对所有的应用市场中提交的所有的应用一直有一个追踪统计, 否则很难做出判断. 另外一种方法是根据应用的流行程度 (下载量) 或者评分来确定原版应用. 但是这种方法也存在着一些问题. 除非是非常知名的应用, 比如 QQ 是腾讯公司的产品, 但是很多其他应用提交到应用市场之后用户不知道哪些是重打包应用并且会把重打包应用当做是原版应用下载. 除此之外, 还有通过代码的大小或者 apk 文件的大小来确定原版应用, 这种方法也不是可靠的, 因为代码大小和 apk 文件的大小对于重打包应用来说都是可控的.

5.4 系统的应用场景

本文系统的目标是在应用市场级别做增量式的应用重打包检测. 假设应用市场会维护一个现有应用的特征库. 每当应用的开发者在市场中准备发布一个新应用的时候, 系统会对该应用进行检查. 首先提取应用的特征, 然后在应用特征库中进行比对. 如果没有发现重打包的可能, 那么应用就可以发布, 新应用的特征被加入到应用特征库中. 如果发现提交的应用跟特征库中的应用相似并且签名信息不同, 那么提交的应用就要被暂时禁止发布并且交给市场的管理者进行进一步审核.

6 相关工作

Android 应用的重打包是 Android 平台安全和隐私问题的主要来源之一, 学术界已经有了不少检测 Android 应用重打包的工作. DroidMOSS[4] 在 Dalvik 字节码层进行应用重打包检测. DroidMOSS 提取 Dalvik 字节码中的操作码序列, 使用模糊散列的方法对应用程序产生一个指纹签名并作为特征, 通过比较应用程序指纹之间的编辑距离得到应用程序的相似度. 类似的, Juxtapp[5] 使用了特征散列的方法对应用程序产生指纹签名并且以它们之间的 Jaccard 距离作为重打包的判断依据. DroidMOSS 和 Juxtapp 都是从 Dalvik 字节码中提取静态的特征信息, 并且使用不同的散列技术来将这些静态信息表示成向量从而进行比较. 这种比较方法的优点是简单快速, 能够很容易的扩展到大规模应用的比较. 但是重打包应用很容易就能逃避这种检查技术, 比如最简单的交换代码顺序或者增添删除操作码就会导致应用程序的指纹发生改变从而导致检测方法失效.

DNADroid[6] 通过比较应用的程序依赖图 (PDG) 来检测重打包应用. 基于程序依赖图的检测技术是代码克隆检测中经常使用的方法. 它使用了程序的语义信息, 因此检测的准确率应该会比较高. 但是基于程序依赖图的检测方法执行效率是个问题. 在 DNADroid 中, 作者使用 Hadoop Mapreduce 并行计算框架在四台机器上执行应用的重打包检测, 但是平均每分钟只能比较 0.71 个应用对. 因此, 这种方法的扩展性不高, 很难应用到应用市场级别数十万的应用检测.

Zhou 等人 [7] 提出了基于程序依赖图的模块解耦技术来把应用程序的代码划分为核心模块和次要模块. 首先根据 Dalvik 字节码建立程序依赖图, 使用聚类算法来将程序依赖图中的包聚类. 然后根据应用程序配置文件的一些基本信息 (活动、服务、内容提供者、接收者等) 来确定已经聚类的模块中哪些是主要模块和次要模块. 应用程序之间的比较是通过从主模块中提取出来的一些语义的信息, 如应用程序申请的权限和 Android API 的调用等. 作者同时提出了一个线性搜索算法来降低在一堆应用中检测重打包的复杂度, 即不需要两两之间都要比较, 只需比较 $O(n\log_n)$ 个应用对即可.

本文工作与上述工作相比, 在能够确保准确性的前提下保持了良好的可扩展性. 与使用特征散列计算指纹的方法相比, 本文基于计数的代码克隆检测技术准确性更高, 但效率略低. 由于 DroidMOSS[4] 与 Juxtapp[5] 都没有开源版本, 因此没有与这两个系统进行实际的对比. 从理论上来讲, 本文方法的准确性更高是因为比较的是代码块的特征矩阵, 保留的信息更多. 重打包应用很容易就能逃避使用特征散列的检查技术, 比如最简单的交换代码顺序或者增添删除操作码就会导致应用程序的指纹发生改变从而导致检测方法失效, 但是本文方法能够很容易的检测到这种情况. 本文方法比使用特征散列计算指纹的方法效率略低的原因也正是因为前者保留的信息更多. 前者对每个应用程序产生一组特征矩阵, 而后者对每个应用产生一个指纹向量, 因此比较起来花费的时间会比后者多一些. 但由于应用的相似度比较是完全可并行执行的任务, 因此在扩展到多台机器上并行执行时, 本文系统执行效率是完全可以接受的. 根据实验结果, 本文系统可以很容易的扩展到数十万应用数量的应用市场级别.

与基于程序依赖图的检测技术相比, 本文方法在保持准确率的基础上效率更高. 例如, DNADroid[6] 在 4 台机器组成并行计算框架中平均每分钟只能处理 0.71 个应用对, 而本文系统在性能更差的单台机器上单进程处理一个应用对只需要 0.36 秒. 而在实验中, 系统扩展到 10 台机器上之后, 平均比较一个应用程序对只需 0.00072 秒. 因此, 本文的工作与之前的相关工作相比, 在准确性和可扩展性方面都达到了一个很好的平衡.

7 结束语

本文研究了目前 Android 平台中存在的应用重打包问题, 提出了基于计数的代码克隆检测技术来

检测应用重打包的方法, 实现了一个准确并且可扩展的原型系统. 实验研究了国内的 5 个知名 Android 第三方市场, 发现市场中 3.0% ∼ 10.5% 的应用都是重打包应用, 表明了目前 Android 市场中应用重打包问题的严重性. 本文人工分析了检测到的重打包应用, 揭示了应用重打包的原因. 结果表明, 本文系统能够有效的在应用市场级别进行应用重打包的检测与防御.

参考文献

1 Gibler C, Stevens R, Crussell J, et al. AdRob: examining the landscape and impact of Android application plagiarism. In: Proceedings of 11th International Conference on Mobile Systems, Applications and Services, Taipei, 2013

2 Enck W, Octeau D, McDaniel P, et al. A study of Android application security. In: Proceedings of the 20th USENIX Security Symposium. Berkeley: USENIX, 2011

3 Yuan Y, Guo Y. Boreas: an accurate and scalable token-based approach to code clone detection. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. New York: ACM, 2012. 286–289

4 Zhou W, Zhou Y, Jiang X, et al. Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy. New York: ACM, 2012. 317–326

5 Hanna S, Huang L, Wu E, et al. Juxtapp: a scalable system for detecting code reuse among Android applications. In: Flegel U, Markatos E, Robertson W, eds. Detection of Intrusions and Malware, and Vulnerability Assessment. Berlin: Springer, 2013. 62–81

6 Crussell J, Gibler C, Chen H. Attack of the clones: detecting cloned applications on android markets. In: Proceedings of the 17th European Symposium on Research in Computer Security, Pisa, 2012. 37–54

7 Zhou W, Zhou Y, Grace M, et al. Fast, scalable detection of Piggybacked mobile applications. In: Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy. New York: ACM, 2013. 185–196

Detecting repackaged Android applications based on code clone detection technique

WANG HaoYu, WANG ZhongYu, GUO Yao* & CHEN XiangQun

Key Laboratory of High-Confidence Software Technologies (*Ministry of Education*)*, School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871*, China*
*E-mail: yaoguo@sei.pku.edu.cn

Abstract With the popularity of smart phones, mobile applications are growing rapidly, especially the Android applications. These applications not only provide useful features, but also bring security and privacy issues. The plagiarist can easily crack the legitimate applications, add malicious code or change the existing libraries, and advertise them in various third-party markets. The repackaged applications not only compromise the copyright of the original authors, but also weaken the security and privacy of users. In this paper, we propose an accurate and scalable approach to detect repackaged android applications and implement a prototype system. We use the system to study more than 14,500 applications collected from Google Play market and various third-party markets. The results demonstrate the effectiveness and scalability of our approach.

Keywords smartphone, repackage, mobile application, code clone, security, privacy

WANG HaoYu was born in 1991. He received the bachelor degree in computer science from Xiamen University, Xiamen, in 2011. Currently, he is a Ph.D. student in the School of Electronics Engineering and Computer Science at Peking University. His research interest includes operating systems and mobile computing.

GUO Yao was born in 1976. He received the Ph.D. degree in computer engineering from the University of Massachusetts, Amherst, in 2007. Currently, he is an associate professor in the School of Electronics Engineering and Computer Science at Peking University. His research interest includes operating systems, low-power design and mobile computing.

WANG ZhongYu was born in 1991. He received the bachelor degree in computer science from Peking University, Beijing, in 2013. Currently, he is a software engineer at Google Inc. His job includes improving tools of Android Content Review Engineering Team.

CHEN XiangQun was born in 1961. Currently, she is a professor in the School of Electronics Engineering and Computer Science at Peking University. Her main research interest includes system software and software engineering, etc.