SPRD：基于应用 UI 和程序依赖图的 Android 重打包应用快速检测方法

汪润[1,2,3]，王丽娜[1,2,3]，唐奔宵[1,2,3]，赵磊[1,2,3]

（1. 武汉大学空天信息安全与可信计算教育部重点实验室，湖北 武汉 430072；
2. 武汉大学计算机学院，湖北 武汉 430072；3. 武汉大学国家网络安全学院，湖北 武汉 430072）

摘 要：研究发现重打包应用通常不修改应用用户交互界面（UI, user interface）的结构，提出一种基于应用 UI 和程序代码的两阶段检测方法。首先，设计了一种基于 UI 抽象表示的散列快速相似性检测方法，识别 UI 相似的可疑重打包应用；然后，使用程序依赖图作为应用特征表示，实现细粒度、精准的代码克隆检测。基于所提方法实现了一种原型系统——SPRD（scalable and precise repacking detection），实验验证所提方法具有良好的可扩展性和准确性，可以有效地应用于百万级应用和亿万级代码的大规模应用市场。

关键词：重打包；代码克隆；用户界面；程序依赖图；安全与隐私

中图分类号：TP309.1

文献标识码：A

doi: 10.11959/j.issn.1000-436x.2018045

SPRD: fast application repackaging detection approach in Android based on application's UI and program dependency graph

WANG Run[1,2,3], WANG Li'na[1,2,3], TANG Benxiao[1,2,3], ZHAO Lei[1,2,3]

1. Key Laboratory of Aerospace Information Security and Trusted Computing Ministry of Education, Wuhan University, Wuhan 430072, China
2. School of Computer, Wuhan University, Wuhan 430072, China
3. School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China

Abstract: A two stage detection approach which combine application's UI and program code based on the observation that repackaging applications merely modify the structure of their user interface was proposed. Firstly, a fast hash similarity detection technique based on an abstracted representation of UI to identify the potential visual-similar repackaging applications was designed. Secondly, program dependency graph is used to represent as the feature of app to achieve fine-grained and precise code clone detection. A prototype system, SPRD, was implemented based on the proposed approach. Experimental results show that the proposed approach achieves a good performance in both scalability and accuracy, and can be effectively applied in millions of applications and billions of code detection.

Key words: repackaging, code clone, user interface, program dependency graph, security and privacy

收稿日期：2017-08-09；**修回日期**：2017-12-21

通信作者：王丽娜，lnwang@whu.edu.cn

基金项目：国家自然科学基金资助项目（No.U1536204, No.61672394, No.61373169, No.61672393）；国家高技术研究发展计划（"863"计划）基金资助项目（No.2015AA016004）

Foundation Items: The National Natural Science Foundation of China (No.U1536204, No.61672394, No.61373169, No.61672393), The National High Technology Research and Development Program of China (863 Program) (No.2015AA016004)

1 引言

移动互联网的快速发展使一些智能设备如智能手机、智能手表等广泛普及并带来了移动应用数量的剧增。根据全球移动通信系统协会（GSMA, Global System for Mobile Communications Association）的一份报告显示，从 2016 年开始移动设备的数量已经超过了人类的总人数。目前，两大主流移动平台，Android 平台的官方商店 Google Play 和 iOS 平台的 App Store 上均有超过百万的移动应用。移动应用不仅能够提供给用户丰富的功能体验，而且也给人们的生活带来了很多便利。在移动市场中由于恶意应用的普遍存在，对用户的安全和隐私已经造成了严重的威胁。

报告显示，Android 平台已经成为恶意软件泛滥的重灾区，其中，98%以上的恶意应用都是发现在 Android 平台。在这些恶意应用中，有超过 86%的应用都是重打包应用[1]。重打包应用是由恶意开发者反编译修改已经发布在应用商店中的合法应用程序，在程序代码中插入恶意的代码片段或修改部分代码片段，引诱用户下载安装使用，达到谋取利益或传播恶意软件的目的。由于 Android 应用程序具有易反编译修改等特点，重打包应用在 Android 平台上更为普遍。从恶意开发者制造重打包应用的动机来看，主要有以下 2 种。

1) 谋取非法利益。恶意开发者替换应用中的广告库，重打包后发布到应用市场，赚取广告收益或汉化侵权谋取利益等。

2) 传播恶意软件。恶意开发者反编译修改应用程序，插入恶意代码，窃取用户隐私信息如读取用户短信、通信录等，上传到远程服务器或利用恶意代码实施远程攻击等。

由于 Android 平台上重打包应用广泛存在且重打包应用的严重危害性，已经引起了研究人员的广泛关注并展开了一系列的相关研究工作[2,3]。这些研究工作多是通过计算应用之间的相似性，利用应用之间的相似度完成重打包应用的检测。具体来说，概括为以下 2 种方法。第一种方法是基于代码克隆的重打包应用检测[4~8]。这种方法主要是比较应用程序代码的相似度，提取应用程序的代码模式如控制流图和数据流图等作为应用代码的特征表示。但是这种方法不能有效应对代码混淆的问题，同时由于其检测效率低下并不适用于百万级应用市场的亿万级代码检测。另外一种方法是基于 Android 应用资源文件的重打包检测[9~11]。这种方法利用 Android 应用交互性强的特点，在 Android 应用的安装文件中存在大量的资源文件，如图片、音频以及视频文件等，而重打包应用不修改或较少地修改安装文件中的资源文件，通过比较资源文件的相似性实现重打包应用检测，这种方法可以有效缓解代码混淆的攻击，但是存在较高的误报率。Android 应用中存在大用户界面（UI），应用 UI 是一种特殊的资源文件。有研究发现重打包应用为了引诱用户下载通常不修改应用（UI）的结构，因此，有研究将 UI 作为应用相似性比较的特征[9,10]。这种方法也属于基于 Android 应用资源文件的重打包应用检测方法。但是这类方法的不足在于有些应用使用的是同一套模板，导致在检测的时候存在较高的误报率。

研究发现，目前，针对 Android 平台重打包应用检测方法存在的不足体现在以下 3 个方面：1) 能够实现高检测准确率，但检测速度低下，不能有效应用于百万级应用市场的重打包检测；2) 能够实现快速检测，但不能保证检测的准确率；3) 一些方法在检测的速度和准确率之间做到很好的平衡，但是在预处理阶段需要消耗大量的计算资源。因此，如何在检测的速度、准确率和计算资源开销这 3 个方面进行平衡是目前重打包应用检测中亟待解决的问题。在面对百万级应用市场的重打包应用检测中，存在以下 3 个挑战。

1) 程序代码量巨大。在百万级的应用市场中存在近 10 亿行代码，如果仅在代码层进行两两比较需要进行超过 10^{16} 次比较。在有新的应用上架时，同样需要重新进行一次比较，计算资源开销巨大，无法做到实时的在线检测。

2) 源代码获取困难。安全分析人员仅能从应用市场上获取应用的安装文件（apk），反编译获取 smali 代码，其可读性差，导致基于源码的代码克隆检测等技术并不能直接应用。

3) 攻击方式多样。国内市场发布的应用多经过代码混淆处理，导致基于控制流图等特征表示的代码克隆检测方法效果不理想。恶意开发者还会猜测检测系统的工作原理，精心构造重打包应用绕过检测。

为了解决面向百万级应用市场的快速重打包应用检测问题，同时保证检测的准确率和降低计算资源的开销，在一定程度上对抗代码混淆的攻击。

本文提出了一种基于应用 UI 和程序代码的两阶段重打包应用检测方法。首先，利用重打包应用通常不修改应用 UI 结构的特点，设计了一种基于应用 UI 抽象表示的散列快速相似性检测方法，识别出可疑的重打包应用；然后，提取这些可疑重打包应用的程序依赖图（PDG, program dependency graph）作为应用的特征表示，在代码层进行细粒度的代码克隆分析，检测相似的代码片段，有效地解决应用 UI 作为特征表示带来检测误报率高的问题。本文提出的应用 UI 抽象表示方法能够对抗应用资源文件的插入、修改等攻击。利用 PDG 作为应用的代码特征表示可以有效抵抗代码插入、修改等攻击行为[6]。实验表明，上述方法能够应用于百万级应用市场的快速准确重打包应用检测，同时本文方法不需要在预处理阶段反编译所有的待测应用，可以节约大量的计算资源开销。本文的主要贡献如下。

1) 设计并实现了一套面向百万级应用市场的两阶段重打包应用快速检测（SPRD, scalable and precise repacking detection）系统，利用应用 UI 和程序代码作为应用相似性检测的特征，实现快速准确的重打包应用检测。

2) 提出了一种基于 Android 应用 UI 抽象表示的散列快速相似性检测方法，可以在秒级时间内实现百万级应用的快速相似性检测。

3) 利用应用程序依赖图作为应用程序代码的特征表示，实现应用程序代码细粒度、高精度的克隆检测，降低仅使用应用 UI 作为应用特征表示带来的高误报率。

4) 实验中从 Google Play 官方商店和多个第三方移动应用市场收集了超过 100 万个应用，在大规模的实验数据集中验证了本文方法的有效性，实验结果显示 SPRD 可以达到 93.3%的检测准确率。

2　相关工作

Android 平台已经成为移动恶意软件的重灾区，对用户的安全和隐私造成了严重的威胁。因此，建立健康的移动应用市场有着重要的意义。近年来，学术界和工业界为此做了许多重要的工作[11~24]。由于 Android 平台的开放性以及 Android 应用易反编译等特点，重打包应用成为 Android 平台最常见的恶意软件类型之一[25~29]。目前，针对 Android 重打包应用的检测主要分为基于代码克隆[4~8,30~33]和基于应用资源文件相似性比较[9,10,34~36]这 2 类方法，它们都是通过计算应用之间的相似性检测重打包应用。接下来，详细介绍这 2 类检测方法。

2.1　基于代码克隆的检测

基于代码克隆的重打包应用检测是一种最常见的代码相似性检测方法并被广泛应用于 Android 重打包应用检测。DroidMOSS[4]在操作码上使用了一种模糊散列技术生成表示应用的指纹信息，然后，使用编辑距离计算 2 个应用的相似性。Juxtapp[5]首先利用 k-gram 对应用的操作码进行处理，然后，使用一种基于 Bloom-filter 的特征散列算法生成应用的向量表示，再使用 Jaccard 相似性距离计算 2 个应用的相似性。文献[4]和文献[5]的 2 种方法可以实现大规模的应用相似性比较，但是如果在代码中插入或调整代码位置等，会导致检测方法失效，也就是说不能有效地检测 3.2 节中的类型 2 重打包应用。DNADroid[6]使用程序依赖图作为应用的特征表示，借助 WALA 为应用中每个类的每个方法生成一个 PDG，利用图的相似性匹配来检测相似的应用，这种方法有较高的检测准确率，但是由于图匹配无法在多项式时间内完成，因此该方法难以扩展到百万级的应用市场检测。文献[7,8]利用一种基于计数的代码克隆检测技术实现重打包应用的检测，同时该方法有效地降低 Android 平台中第三方库在应用相似性比较中造成的影响，但是文献中的方法难以应用于存在严重代码混淆和百万级市场的程序代码相似性比较情况。Chen 等[30,31]利用应用程序控制流图的质心表示移动应用的特征，实现大规模快速的应用相似性检测。但是该方法在处理之前要抽取所有应用的控制流图，需要消耗大量的计算资源，而且在代码混淆严重时，会影响控制流图的生成，直接导致该方法失效。

2.2　基于应用资源文件的检测

由于 Android 应用的事件触发机制设计，在 Android 应用中存在丰富的用户交互界面 UI。有研究发现恶意开发者为了迷惑用户下载安装应用，通常不修改或较少修改应用的 UI 等资源文件。ViewDroid[9]提出了一种将用户界面作为应用特征表示的重打包应用检测方法，利用有向图表示应用，其中，有向图的顶点是应用的视图（view），有向边表示 2 个视图之间可以通过事件进行切换，然后利用 VF2 算法[37]比较图的相似性

实现应用的相似性比较。如果攻击者在应用中恶意地插入多余的视图以及应用的视图很少的情况下，该方法并不能取得较理想的效果。DroidEagle[10]使用应用布局作为特征检测应用的相似性，该系统由用于部署大规模应用市场检测子系统 RepoEagle 和轻量级的用于移动端检测的子系统 HostEagle 这 2 个部分组成，帮助用户实现快速检测。如果应用的布局被恶意开发者做了少量的修改会导致客户端检测失效，造成 DroidEagle 不能应用于大规模应用市场的快速检测。以上基于应用 UI 等资源文件的检测方法均可以有效地对抗代码混淆的攻击，但是应用 UI 的树型特征表示难以应对大规模应用市场的应用快速相似性比较。另外，存在一些应用从同一套模板的基础上开发而来，其布局和风格相似，但并不是重打包应用，导致这类方法在此类应用的分析上会存在较高的误报率。

综上所述，现有的重打包应用检测方法没有在检测的速度和准确率之间做到很好的平衡，难以应对百万级应用市场快速、准确检测的要求。本文提出了一种基于应用 UI 和程序代码的两阶段检测方法，旨在解决海量应用市场中，快速准确的重打包应用检测问题，降低分析检测过程中计算资源的开销。

3 相关背景知识

3.1 Android 应用程序

应用集中分发机制是 Android、iOS 等移动平台区别于 PC 平台软件生态系统的重要特点，移动应用市场在保护用户安全和隐私方面扮演了重要的角色。Android 应用程序主要由程序代码和资源文件构成，资源文件包括 UI、多类型资源的 xml 文件（如颜色、字符串等）、图片、音频和视频等。程序代码主要包括由 Java 语言编写编译生成的 Dalvik 字节码和由 C/C++语言编写编译生成的 so 文件。本文仅考虑修改由 Java 语言编写生成的重打包应用，修改由 C/C++语言编写的本地代码不在本文的研究范围内，而且鲜有报告指出恶意开发者通过修改本地代码生成重打包应用。

在 Android 应用程序静态分析中，通常会受到第三方的库文件和代码混淆的影响。在 Android 应用程序中第三方的库文件被大量使用如功能扩展库、工具类库等。这些库文件会占用一定比例的代码，它们的存在也严重影响基于程序分析的应用相似性比较[38]。应用开发者从保护版权或应用安全的角度出发，通常会对 Android 应用进行一些代码混淆处理操作，这会直接导致程序分析技术应用困难。应用 UI 是 Android 应用 4 个组件之一 Activity 的重要表现形式，重打包应用为了诱导用户下载安装使用，通常不修改应用 UI 的结构。利用应用 UI 作为应用相似性特征表示，可以降低在程序分析阶段由于第三方的库文件、代码混淆等对分析造成的影响。

3.2 重打包应用

谋取非法收益和传播恶意代码收集用户的隐私数据，实施非法攻击威胁用户的安全是恶意开发者制造重打包应用的主要动机。Android 平台重打包应用的最大特点是恶意开发者在保持原有合法应用核心功能不变的基础上，通过增加、删除和修改程序代码以及改变应用资源文件等方法生成重打包应用。根据恶意开发者修改 apk 文件的内容以及重打包应用构造的精细程度，本文对重打包应用主要概括为以下 4 种类型。

类型 1 不修改 UI 和程序代码。一些恶意开发者在生成重打包应用时保持应用 UI 和程序代码不变，仅仅修改签名或替换一些广告库等。

类型 2 不修改 UI，修改部分程序代码。一些重打包应用仅仅修改 UI 组件的属性如替换 UI 中的字符显示、图片等，不修改 UI 的结构，但是增加、删除和修改部分程序代码。

类型 3 不修改 UI，修改程序代码但保持功能语义相同。保持 UI 结构不变，恶意开发者在理解应用功能的基础上，重新编写代码实现相似的应用功能。

类型 4 UI 不同，仅复制或修改部分功能代码片段。该类型重打包应用主要是开发者抄袭了其他应用的部分功能，复制部分代码作为重打包应用的子功能。

4 总体设计

本节主要介绍所提方法的基本思想以及系统的总体设计等内容。目前，针对 Android 平台的重打包应用检测方法难以在检测的速度、检测的准确率和计算资源开销等三方面做到有效的平衡，导致在百万级应用市场的大规模检测中存在困难。研究发现重打包应用通常不修改或较少修改应用 UI 的

结构，本文基于此发现提出了一种两阶段的检测方案，实现应用快速准确的相似性比较。

本文重打包应用系统 SPRD 的框架如图 1 所示。首先从各大应用市场收集大量的 Android 应用程序，其中，包括 Android 的官方应用市场 Google Play。第一阶段应用 UI 的快速相似性比较步骤主要分为预处理和相似性检测这 2 个部分，在预处理中，首先抽取 Android 应用的 UI，将 UI 中每一个视图（view）生成一种抽象化的表示，并使用散列算法生成 view 的唯一指纹值；在相似性检测部分将每一个应用的 view 指纹值进行比对，并将指纹值相同的 view 做聚类处理，找出 UI 相似的应用，输出可疑的重打包应用集合。在第二阶段程序代码的细粒度克隆检测中，首先，反编译在第一阶段中输出的可疑重打包应用，生成每一个应用的 PDG 作为其特征表示，利用图的相似性比较算法实现程序代码的克隆检测。最后，根据上述 2 个阶段计算得出的相似值进行综合判定完成重打包应用的检测。

本文方法能够有效检测类型 1 和类型 2 的重打包应用。针对类型 3 的重打包应用，可以运用本文中第一阶段的方法获得可疑重打包应用，降低应用在程序分析阶段的规模，显著提高分析和检测的效率。针对类型 4 的重打包应用，可以运用本文第二阶段的代码克隆检测方法，实现有效检测。但是会存在检测速度慢、难以拓展至百万级应用市场规模的问题。

5　SPRD 方法

5.1　基本定义

本文的两阶段重打包应用检测方法主要从应用 UI 和程序代码 2 个部分展开比较，在详细描述本文的检测方法之前，这里先给出一些基本的符号解释和定义。

定义 1　Android 应用程序。Android 应用程序表示为 $A=(I, D)$。其中，I 代表应用的用户交互界面 UI，D 代表应用程序代码。

定义 2　Android 应用 UI。Android 应用 UI 表示为 $I=\{V_i\}, i=1,\cdots,N$。其中，V_i 表示应用的视图 view，N 表示应用中 view 的个数。Android 应用 UI 的 view 表示为 $V=<L, C, \Phi>$。其中，L 和 C 分别表示布局和组件集合，Φ 表示布局和组件之间的包含关系。Android 应用 UI 的多个 view 之间通过事件进行切换，view 中每个组件（如 Button、ImageView 等）都包含在布局（如 LinearLayout、ScrollView 等）中控制显示。

定义 3　Android 应用程序代码。Android 应用程序代码表示为 $D=\{G_i\}, i=1,\cdots,M$。其中，G_i 表示 Android 应用 Dalvik 字节码生成的程序依赖图，M 表示应用程序中函数的个数。应用程序依赖图表示为 $G=<S, E>$，其中，S 表示函数的语句集，E 表示边集，当语句之间有数据或控制依赖时，语句之间存在有向边连接。

5.2　应用 UI 相似性比较

本文中第一阶段利用应用 UI 作为特征进行应用的相似性比较，发现 UI 相似的可疑重打包应用，主要分为离线和在线 2 种分析比较方式。在离线分析中，从各大 Android 应用市场中收集大量的应用样本，通过在收集的待测应用中做 UI 的相似性比较识别可疑的重打包应用。离线分析检测过程中，本文中构建应用 view 的指纹数据库，每个应用 view 都有相应的指纹标识。在线分析中，针对刚上架的移动应用，根据待检测应用生成的 view 指纹，在指纹数据库中按照指纹的属性字段进行线性查找，实现从移动应用市场中快速地匹配出 UI 相似的应用。利用 UI 进行相似性检测能够大幅度减少程序代码的比较次数和分析时间，降低计算资源的开销，提高检测的速度。

本文提出了一种应用 UI 的抽象表示方法，实

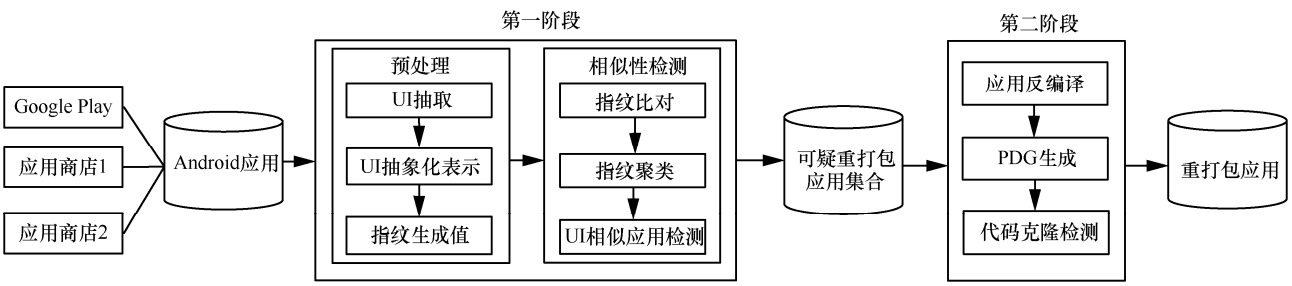


图 1　Android 重打包应用检测系统 SPRD 框架

现应用 UI 的快速检测，识别检测可疑的重打包应用，如图 1 中第一阶段所示。在 UI 相似的可疑重打包应用检测中，主要分为预处理和相似性检测等两部分。在预处理中，本文提出了一种应用 UI 抽象表示方法，能够有效抵御如修改 view 和组件的属性、插入以及修改部分 view 和组件等攻击。同时，设计了一种基于应用 UI 抽象表示的指纹生成方法，用于实现相似应用 view 的快速比较、匹配和查找等操作。在相似性检测中，主要是通过比较应用 UI 的 view 指纹。将相同的 view 指纹放入到同一个类别中，完成指纹的聚类操作。然后，通过比较 2 个应用中相同 view 比例来分析应用的相似性。最后，输出 UI 相似的可疑重打包应用到下一阶段进行细粒度的程序代码相似性检测。

应用的 UI 相似性检测主要包括 UI 组件的抽取、UI 抽象表示、应用特征指纹的生成以及相似性计算等部分，本节将详细介绍这些组成部分的细节内容。

1) UI 抽取

在应用 UI 的相似性检测中首先需要抽取应用 UI 的特征表示，获取应用 UI 的基本组成单元。在 Android 应用中，UI 的基本组成单元是 view。view 中包含有多个与用户直接进行交互的组件，如按钮 Button，每个组件都属于一个布局（ViewGroup），如线性布局 LinearLayout。图 2 是从 Google Play 中下载解析微信应用中的一个 view 得到的树型结构图，每一个节点均表示组件或布局。每个组件都有属性标识，本文不抽取属性“visibility= "invisible"”的组件，因为通过研究分析发现大量的恶意应用会在 view 中添加这类不可见的组件，绕过检测。

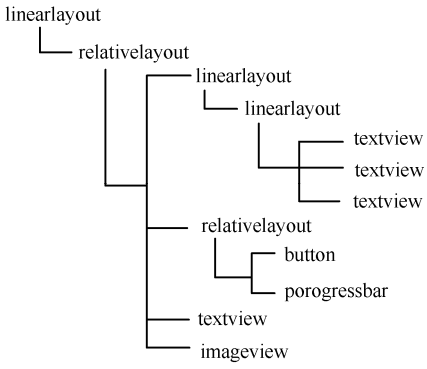


图 2 view 树型结构

本文直接反编译应用获取 apk 文件中的布局文件，解析应用 UI 的 view 结构。本文的 view 解析方法，相比于 ViewDroid[9]等不需要通过反编译代码获取动态生成的布局结构，节省了大量的计算资源开销。同时，本文增加了细粒度的代码克隆检测，用于克服应用 UI 作为特征表示带来的漏报。

2) UI 抽象表示

将应用 UI 的 view 树型结构进行特征抽象表示，用于实现快速的相似性检测。恶意开发者在制造重打包应用时，通常会修改 view 中组件的属性，如位置、大小、文本的文字内容、图片的链接等。但是，应用中 UI 的 view 结构通常不会被修改。本文使用一种抽象化表示规则来转变 view 的结构表示，然后将应用中所有的 view 组合形成应用 UI 的抽象表示集合。基于文中的 UI 抽象表示方法，将图 2 的树型 view 结构表示成线性文本，本文的抽象表示规则具体如下。

元素表示。针对 view 的基本组成元素组件和布局，不考虑它们的属性，直接表示成其名称字符串小写的形式，如布局 LinearLayout 表示成字符串“linearlayout”，组件 Button 表示成“button”，组件“CheckBox”表示成“checkbox”等。

视图表示。在 UI 抽取中，view 表示成树型结构如图 2 所示。本文将树型结构按广度优先遍历将每一层的节点逐层进行线性表示，同一层的兄弟节点按照字符串字典排序，针对同一层中值相同的兄弟节点，比较以该节点为顶点的子树大小。表 1 是图 1 的树型结构描述，并按照节点的抽象表示进行排序。

表 1 view 树型结构描述

ID	层次	父节点	节点抽象表示
a	1	#	linearlayout
b	2	a	relativelayout
c	3	b	imageview
d	3	b	linearlayout
e	3	b	relativelayout
f	3	b	textview
g	4	d	linearlayout
h	4	e	button
i	4	e	progressbar
j	5	g	textview
k	5	g	textview
l	5	g	textview

根据表 1 和 view 树型结构线性抽象表示规则，图 2 中的应用 view 树型结构线性抽象化表示为“(a(b(c,d(g(j,k,l)),e(h,i),f)))”，然后将表 1 中的 ID 分

别用相应的节点抽象表示字符串做替换，得出最终的线性表示，如图 3 所示。view 树型结构线性抽象表示算法设计如算法 1 所示。

view 线性抽象表示	(linearlayout(relativelayout(imageview,linearlayout(linearlayout(textview,textview,textview)),relativelayout(button,progressbar),textview)))
散列值	7c3e342aab2cdb8d1e088a7f60e3c7af
指纹三元组	{140, 7c3e342aab2cdb8d1e088a7f60e3c7af, com.tencent.mm }

图 3　view 指纹生成

算法 1　view 树型结构线性抽象表示

输入　UI 的 view 结构树 *V-tree*

输出　UI 的 view 线性抽象表示 ***V-linear***

① *level*=*getHeight*(*V-tree*); /*获取树的高度*/

② for *m* in range(0, *level*)

③ *NS*← *getNode*(*V-tree*, *m*); /* 将 *V-tree* 第 *m* 层的节点放入集合 *NS* 中*/

④ *NS*← *sort*(*NS*, *order*=*asc*); /* 将集合 *NS* 中节点按照字符串字典升序排序*/

⑤ *flag*=*existSameItem*(*NS*); /* 判断集合 *NS* 中是否存在值相同的节点*/

⑥ if *flag*=true

⑦ *SN*← *getSameNode*(*NS*); /* 将集合 *NS* 中值相同的节点值放入集合 *SN* 中*/

⑧ *sortSubTree*(*NS*, *SN*, *V-tree*); /* 比较相同节点的子树，并调整集合 *NS* 节点的顺序*/

⑨ end if

⑩ ***V-linear***← *output*(*NS*, *m*) /* 将第 *m* 层排序后的节点放入 ***V-linear*** 中*/

⑪ end for

3) 指纹生成

对每一个 view 进行抽象表示之后，生成相应的指纹对其进行表示，实现应用相似 view 的快速比较、匹配和查找等操作。本文中应用 UI 的 view 指纹表示成一个三元组的形式，用于快速地识别可疑的重打包应用。其指纹的三元组表示为<*length*, *hash_value*, *app_id* >。其中，*length* 为线性抽象表示字符串的长度， *hash_value* 表示字符串的散列值，*app_id* 表示为应用程序的 ID。图 3 为图 2 中 view 生成的指纹，图 2 是从 Google Play 中获取的微信应用（*app_id*: *com.tencent.mm*）。

在生成 view 的指纹后，SPRD 以键值对的形式保存所有应用 view 指纹的三元组在字典 ***DIC*** 中。其中，键 key 是字符串长度 *length*，值 value 是散列值 *hash_value* 和应用 ID 组成的二元组。在字典 ***DIC*** 中，相同键的值以列表的形式连续存放。图 4 为存放 view 的字典。

```
140 : {
{'7c3e342aab2cdb8d1e088a7f60e3c7af', 'com.tencent.mm'}
}
```

图 4　存放应用 view 指纹的字典 ***DIC***

4) UI 相似应用检测

在 UI 相似应用检测中，通过比较应用的 UI 相似性生成可疑的重打包应用集合。然后将 UI 相似的可疑重打包应用在本文方法的第二阶段做细粒度的程序代码相似性分析，检测可疑的重打包应用。SPRD 的应用 UI 相似性检测分为离线检测和在线检测 2 种情况。其中，离线检测是为了从已收集的应用中找出 UI 相似的可疑重打包应用；在线检测是针对新上架的应用，在已收集的应用集合中，找出 UI 相似的应用。

在离线的 UI 相似应用检测中，依次遍历字典 ***DIC*** 的键，针对 ***DIC*** 中相同键值的列表，将列表中抽象表示字符串的散列值进行两两相似性比较。如果值相同，则检测出相似的应用 view。在得到存在 view 相似的应用后，通过分析 2 个应用中所有的 view 相似比例来度量应用 UI 的相似性，从而判定其是否为一对可疑的重打包应用。

应用 UI 的相似性度量如式(1)所示，通过计算 2 个应用 UI 的 view 相似比例来判定是否为可疑的重打包应用。在式(1)中，I_1 和 I_2 表示应用的 UI（如定义 2 所示），$I_1 \times I_2$ 表示应用 I_1 和应用 I_2 的 view 笛卡尔集。在笛卡尔集 $I_1 \times I_2$ 中，集合元素表示为(V_i, V_j')，其中，V_i 和 V_j'分别为应用 I_1 和应用 I_2 的 view。函数 *compare*($I_1 \times I_2$, *i*)用于检测集合 $I_1 \times I_2$ 中第 *i* 个元素的 view 对是否相似，如果相似则函数返回 1，否则返回 0。$|I_1 \times I_2|$表示集合中元素的个数，$|I_1|$和$|I_2|$分别表示应用 I_1 和应用 I_2 中 UI 的 view 个数。

$$sim(I_1, I_2) = \frac{\sum_{i=1}^{|I_1 \times I_2|} compare(I_1 \times I_2, i)}{\min(|I_1|, |I_2|)} \quad (1)$$

如果 $sim(I_1, I_2)$大于阈值 0.8，判定这 2 个应用的 UI 相似，下一步将 UI 相似的应用做细粒度的程序代码克隆检测。其中，通过大量的测试样本调节发现，阈值为 0.8 是比较合理的。

5.3 程序代码相似性比较

相同模板开发出来的应用，UI 总是保持相似，但它们并不是重打包应用。因此，针对这类重打包应用需要在应用程序的代码层进行细粒度的代码相似性检测，识别相似的代码片段，降低使用应用 UI 作为特征检测带来的误报率。相同应用的不同版本，UI 有时也会保持较高的相似性。但是它们的签名相同，可以通过提取应用的签名信息来判定应用是否为不同版本的应用。

本文使用程序依赖图 PDG 作为应用程序代码的特征表示，实现细粒度的代码克隆检测。基于 PDG 的代码特征表示可以有效地抵御攻击者在应用程序代码中插入、删除语句以及修改语句顺序等操作的攻击。PDG 主要用于表示程序函数体中语句之间的依赖关系，包括数据依赖和控制依赖等 2 种[39]。由定义 3 可知，PDG 的顶点是函数体中的语句，边是指顶点的语句之间存在数据依赖或控制依赖。其中，数据依赖是指如果语句 S1 中数据依赖语句 S2，则 S1 中有变量的值取决于 S2；控制依赖是指如果语句 S1 与语句 S2 存在控制依赖关系，那么语句 S1 可以控制语句 S2 执行与否。

本文程序代码克隆检测如图 5 所示，在得到 apk 文件之后，对每一个 apk 文件做反编译处理，抽取应用程序中每个函数的 PDG。由于第三方库在应用程序开发中被大量使用，使用白名单过滤第三方库对程序代码克隆检测造成的影响。另外，一些代码行数少的函数存在功能通用以及代码模板相似的特点，这些函数的 PDG 更可能出现相似或相同的问题，因此，本文过滤掉 PDG 的节点小于特定值（本文特定值设为 8）的函数。程序代码克隆检测是通过 2 个程序代码形成的 PDG 中找出相同的节点，该问题可以转化成子图同构的问题，也即发现 2 个 PDG 中语句的映射关系，应用的程序代码相似性计算如式(2)所示。

$$sim(D_1,D_2)=\frac{\sum_{G\in D_1}|C(G)|}{|D_1|} \tag{2}$$

在式(2)中，程序代码 D_1 和 D_2 如定义 3 所示，$|D_1|$表示程序代码 D_1 的函数个数（过滤掉通用的第三方库），$C(G)$表示 D_1 在 D_2 中找到最佳匹配子图的节点个数。

5.4 重打包应用检测

本文利用恶意开发者为了诱导用户下载安装使用重打包应用，通常不修改应用 UI 结构的特点。提出了一种基于应用 UI 抽象表示的散列快速可疑重打包应用检测方法以及通过细粒度的程序代码分析实现准确的代码克隆检测。细粒度的代码克隆检测用于降低使用应用 UI 作为特征检测带来的误报率，提高重打包应用检测的准确率。

在本文两阶段重打包应用检测中，需要统计和分析有多少应用 UI 和应用程序代码被恶意开发者在合法应用基础上做了多大程度的修改。在研究过程中，发现重打包应用具有以下 2 个特点：1) 重打包应用的 UI 常保持不变，恶意开发者会插入或修改程序代码，因此，相比于应用 UI，程序代码更能表示应用的特征；2) 在 Android 应用程序中，应用 UI 中 view 的个数明显少于应用中函数的个数。

因此，在应用的相似性计算中，需要考虑应用 UI 和应用程序代码的相似度以及它们被修改的数量或比例等因素。由上述的重打包应用特点可知，应用的相似性受程序代码的影响更大。在本文中，应用 A_1 和 A_2 相似性计算表示为：$sim(A_1, A_2)=\alpha\cdot sim(I_1, I_2)+(1-\alpha)\cdot sim(D_1,D_2)$。其中，系数 α 为权重系数，根据应用中 view 和应用程序代码中函数的个数计算得出，用于反映有多少应用的 view 和函数被修改。具体的计算如式(3)~式(5)所示。

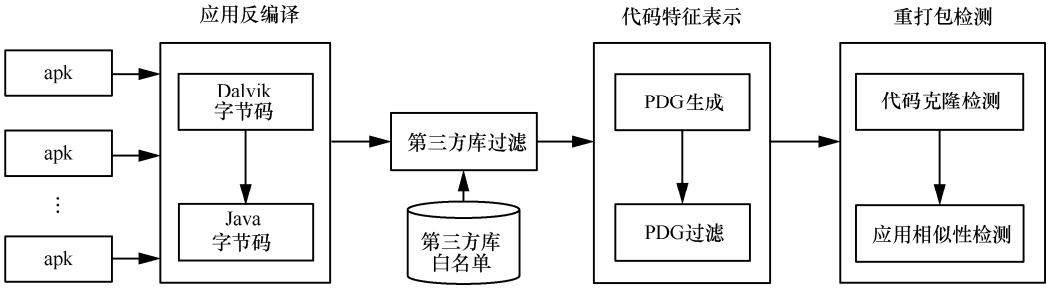


图 5 程序代码克隆检测流程

$$I_m = \max(|I_1|,|I_2|) \quad (3)$$

$$D_m = \max(|D_1|,|D_2|) \quad (4)$$

$$sim(A_1,A_2) = \frac{I_m \cdot sim(I_1,I_2) + D_m \cdot sim(D_1,D_2)}{I_m + D_m} \quad (5)$$

其中，$|I_1|$、$|I_2|$和$|D_1|$、$|D_2|$分别表示应用 UI 的 view 和应用程序代码的函数个数，$sim(I_1, I_2)$和 $sim(D_1, D_2)$如式(1)和式(2)所示，如果 $sim(A_1, A_2)$大于阈值 0.85，判定应用 A_1 和 A_2 是一对重打包应用。通过大量的测试样本调节发现，阈值为 0.85 是比较合理的。

6　实验与结果分析

6.1　方法实现

Android 平台重打包应用检测实验部分主要有以下 6 个步骤。

1) apk 文件反编译。使用 Keytool 提取应用的签名信息，作为唯一标识区分应用。Apktool 用来反编译应用程序，获取 apk 的资源文件和 smali 代码等，解析应用 UI 的 xml 文件获取应用 UI 的 view 结构。

2) UI 指纹生成。本文使用 MD5 算法生成 UI 线性抽象表示的散列值。

3) 指纹字典存储。本文选择使用 Python 自带的数据字典结构，能够在 $O(1)$时间复杂度内实现键的查找。

4) 第三方库过滤。本文使用文献[40]中提供的 1 113 个第三方功能库和 240 个广告库作为白名单，过滤待测应用中的第三方库。

5) 应用程序代码的 PDG 生成。本文使用 soot[41] 为程序中所有类的每种方法生成 PDG。在本文设置当语句之间存在数据依赖时，语句之间建立有向边连接，有研究发现数据依赖更加有效的对抗（如程序代码中修改、插入等攻击行为）。

6) 程序代码相似性比较。本文使用 VF2 算法[37] 找出 PDG 的同构子图，识别相似的程序代码片段。

6.2　实验环境与数据来源

基于所提方法，本文实现了面向大规模的 Android 重打包应用快速检测方法原型系统 SPRD，具体的实验环境 CPU 为 Intel(R) Core(TM) i7-6700K 4 GHz，32 GB 内存，操作系统为 Ubuntu14.04。

本文中实验数据来源于 Android 官方应用商店 Google Play，国内应用商店（Baidu、Anzhi）、美国应用商店（Pandaapp）和欧洲应用商店（Opera）等，具体的各应用商店的实验数据采集情况如表 2 所示。实验数据中 apk 文件大小分布如图 6 所示，收集的 apk 文件最小的为 8 KB，最大的为 49.5 MB，超过 76%的 apk 文件大小超过 1 MB，有超过 60% 的 apk 文件大小为 1~10 MB，这些实验数据基本上可以代表 Android 应用市场中应用程序的普遍大小。采集的所有 apk 文件超过 7 TB，代码总量超过 10 亿行。

表 2　　实验数据采集

应用市场	应用数量	比例
Google Play	279 888	27.7%
Baidu	163 756	16.2%
Anzhi	147 578	14.6%
Pandaapp	228 760	22.7%
Opera	189 422	18.8%
总计	1 009 404	100%

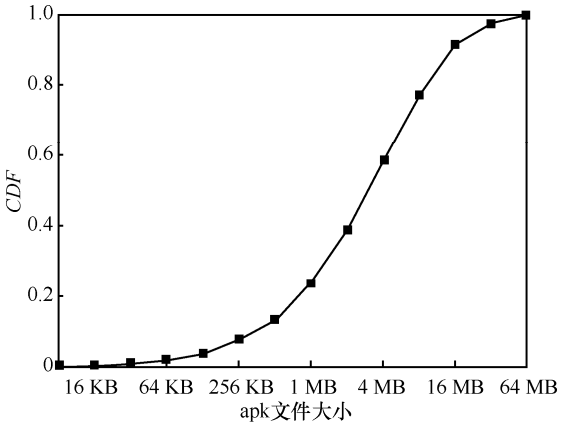


图 6　实验数据中 apk 文件大小分布

6.3　实验结果分析

在本文的实验结果评价中，主要从本文方法在大规模应用市场重打包应用检测的速度和检测的准确率 2 个部分来分析实验结果。

本文方法分为 2 个阶段，第一阶段利用重打包应用不修改应用 UI 的结构，它们的 UI 保持相似的特征，识别出 UI 相似的可疑重打包应用。图 7 表示在第一阶段的 UI 相似性检测中，应用数量的增长与 UI 相似性检测时间的关系。从图 7 可以看出，在离线分析中，随着应用数量的增加，应用 UI 相似性检测的时间并不发生显著变化。在生成应用指纹之后，本文的应用 view 相似性比较方法，依然

可以在 13 s 内从百万级应用中快速地检测出 view 相似的应用。因此，本文所提 UI 相似性检测方法可以有效应用于大规模移动应用市场。

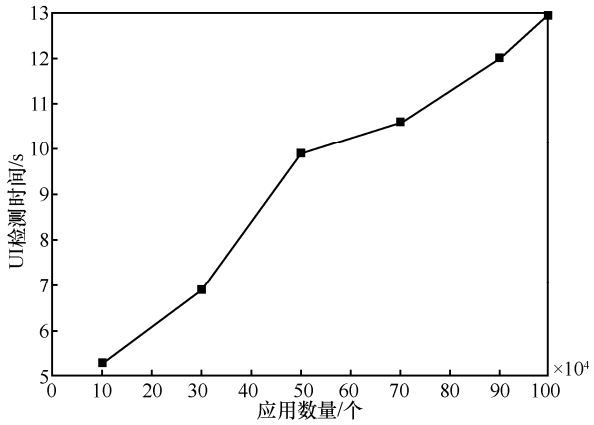


图 7　应用数量与应用 UI 相似性检测时间关系

应用 UI 的相似性检测用于提高第二阶段中细粒度的程序代码克隆检测效率，降低程序代码克隆检测中比较的次数。图 8 表示在第二阶段的程序代码克隆检测中，应用数量的增长与代码对比较次数的关系。从图 8 可以发现，随着应用数量的增长，如果不通过第一阶段的 UI 相似性分析找出可疑的重打包应用，仅仅通过应用程序代码两两比较检测应用的相似性，其代码比较的次数呈现指数爆炸式增长。通过对比发现，本文利用第一阶段的 UI 相似性比较，其程序代码的比较次数依然可以保持在线性增长空间中。

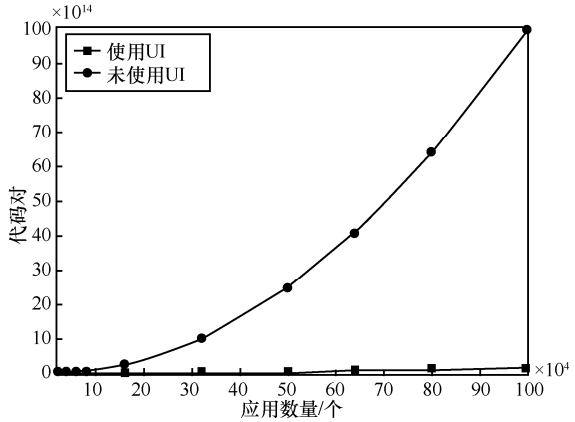


图 8　程序代码对比较次数与应用数量关系

在检测时间方面，SPRD 完整检测时间包括第一阶段的应用 UI 相似性检测和第二阶段的细粒度代码克隆检测。由于本文第一阶段 UI 相似性检测中能够将应用分析的规模压缩到原规模的 11.3%左右，而在第一阶段中应用 UI 的分析与检测平均时间只是第二阶段代码克隆检测时间的 10%左右。因此，SPRD 的完整检测时间明显快于文献[6]、文献[33]、文献[7]等基于代码克隆检测的方法。本文方法 SPRD 能够在 15.2 s 完成完整百万级的重打包应用检测，按照 Centroid[30]实验评价检测时间线性增长规律计算可知，其处理百万级应用时的平均检测时间在 25.6 s。在特征抽取中，SPRD 相比于 Centroid 仅有 11.3%的待测应用需要进行耗时的代码分析，但是 Centroid 需要利用程序分析方法抽取所有待分析应用的控制流图。

图 9 表示在 5.2 节应用 UI 相似性检测和 5.4 节重打包应用检测中，阈值与准确率的关系。实验中通过多次测试调整阈值发现，阈值设为 0.8 时，UI 的相似性检测准确率最高达到 87.1%；阈值设为 0.85 时，应用的相似性检测准确率最高达到 93.3%。

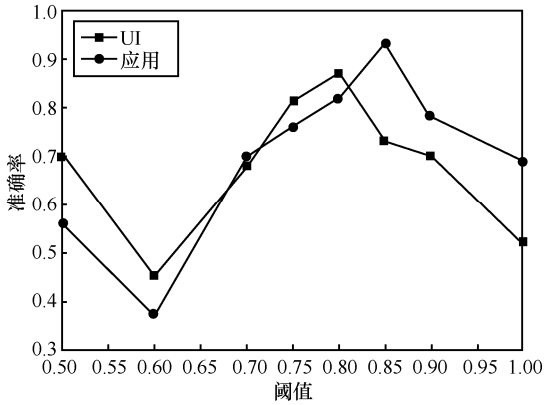


图 9　阈值与准确率的关系

在本文中，重打包应用检测的结果利用机器学习中经常使用到的指标进行评价和度量，分别是：1) 真阳性（TP, true positive）；2) 假阳性（FP, false positive）；3) 真阴性（TN, true negative）；4) 假阴性（FN, false negative）。这 4 个度量指标的具体含义如表 3 所示。

表 3　　度量指标含义

预测值	真实值为 Y	真实值为 N
Y	*TP*	*FP*
N	*FN*	*TN*

根据 4 个度量指标，构成以下 4 个常用指标。具体计算式为

$$precision = \frac{TP}{TP + FP} \tag{6}$$

$$recall = \frac{TP}{TP + FN} \tag{7}$$

$$F\text{-}score = \frac{2 \times precision \times recall}{precision + recall} \tag{8}$$

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{9}$$

Androguard 是一个用于 Android 应用逆向分析和静态分析的开源工具。本文使用 Androguard 测试文中检测出的重打包应用的准确性。实验结果分析发现，本文方法能够实现精准率（precision）为 93.4%，召回率（recall）为 90.5%，准确率和召回率的调和平均值（F-score）为 91.9%，准确率（accuracy）为 93.3%。

利用本文的重打包应用检测系统 SPRD 分析表 2 中收集的测试应用，5 个应用市场中重打包应用分布情况如表 4 所示。

表 4　　应用市场中重打包应用比例

应用市场	重打包应用比例	国家
Google Play	5.23%	美国
Baidu	2.10%	中国
Anzhi	18.90%	中国
Pandaapp	15.20%	美国
Opera	3.70%	欧洲

从实验结果分析看，相比于现有的基于代码克隆的检测和基于应用资源文件的检测方法，本文的两阶段重打包应用检测方法能够实现检测准确率高、检测速度快以及计算资源开销小等。本文方法与这 2 类方法在检测速度、检测准确率和计算资源开销这 3 个方面的比较如表 5 所示。

表 5　　重打包应用检测方法比较

重打包应用检测方法	检测速度	检测准确率	计算资源开销
基于代码克隆检测	慢	高	大
基于资源文件检测	快	低	中
本文方法	快	高	小

6.4　相关讨论

本文使用白名单机制过滤第三方的库文件在代码克隆检测中带来的干扰，由于收集到的第三方库存在不完备等问题，在检测中会造成一些误报率。在未来的工作中，尝试收集更多第三方的库文件减少这方面的误报率。在程序代码克隆检测中，本文主要分析应用程序的 Java 代码，没有考虑应用程序的本地代码。因此，修改本地代码的重打包行为，文中的方法并不奏效。可以应用二进制代码克隆检测相关技术解决本文工作在本地代码克隆检测中的不足，但是少有研究指出恶意的开发者通过修改本地代码生成重打包应用，因此，本文的研究范围依然是合理的。

本文方法不需要反编译所有的待检测应用，抽取它们的程序代码特征表示，节省了大量的计算资源开销。计算的时间复杂度方面，通过第一阶段的 UI 相似性分析，检测出可疑重打包应用集合$|P| \square M$（待检测应用数量）。在使用 PDG 作为特征表示进行代码克隆检测时，其中，VF2 算法在最坏情况的时间复杂度为 $O(n!)$，最好情况为 $O(n^2)$，可以满足大规模环境中快速检测的要求。

7　结束语

本文提出了一种基于应用 UI 和程序代码的两阶段重打包应用检测方法，利用重打包应用不修改应用 UI 结构的特点，设计了一种基于应用 UI 抽象表示的散列快速相似性检测方法。然后，使用 PDG 作为应用的特征表示，实现应用程序代码的克隆检测，提高应用相似性检测的准确率，并降低使用应用 UI 作为特征检测带来的误报率。实验结果表明，本文方法可以应用于百万级应用市场的重打包应用检测与防御，不仅具有较高的检测准确率，而且通过两阶段的检测方法可以节约大量的计算资源开销。本文的方法解决了现有 Android 平台重打包应用检测方法难以在检测的速度、检测的准确率和计算资源开销这 3 个方面进行有效平衡的问题。但本文的重打包应用检测系统也存在一些缺陷，需要做进一步改进。

本文检测方法仍然需要通过分析应用的相似性实现重打包应用检测，未来工作尝试抽取重打包应用程序的内在特征实现检测，例如程序代码的风格、代码间的关联程度等。针对部分程序代码的克隆问题，未来的工作试图抽取一种轻量级的代码特征，实现快速比较并且可以对抗代码插入、修改等攻击。

参考文献：

[1] ZHOU Y, JIANG X. Dissecting Android malware: characterization and evolution[C]//IEEE Symposium on Security and Privacy (SP).2012: 95-109.

[2] ACAR Y, BACKES M, BUGIEL S, et al. Sok: lessons learned from Android security research for appified software platforms[C]// 2016 IEEE Symposium on Security and Privacy (SP). 2016: 433-451.

[3] XU M, SONG C, JI Y, et al. Toward engineering a secure Android ecosystem: a survey of existing techniques[J]. ACM Computing Surveys (CSUR), 2016, 49(2): 38.

[4] ZHOU W, ZHOU Y, JIANG X, et al. Detecting repackaged smartphone applications in third-party Android marketplaces[C]//The second ACM conference on Data and Application Security and Privacy. 2012: 317-326.

[5] HANNA S, HUANG L, WU E, et al. Juxtapp: a scalable system for detecting code reuse among Android applications[C]//International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. 2012: 62-81.

[6] CRUSSELL J, GIBLER C, CHEN H. Attack of the clones: detecting cloned applications on Android markets[C]//European Symposium on Research in Computer Security. 2012: 37-54.

[7] WANG H, GUO Y, MA Z, et al. Wukong: a scalable and accurate two-phase approach to Android app clone detection[C]//The 2015 International Symposium on Software Testing and Analysis. 2015: 71-82.

[8] 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[J]. 中国科学:信息科学, 2014, 44(1): 142-157.
WANG H Y, WANG Z Y, GUO Y, et al. Detecting repackaged Android applications based on code clone detection technique[J]. Science China Information Sciences, 2014, 44(1): 142-157.

[9] ZHANG F, HUANG H, ZHU S, et al. ViewDroid: towards obfuscation-resilient mobile application repackaging detection[C]//The 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks. 2014: 25-36.

[10] SUN M, LI M, LUI J. Droideagle: seamless detection of visually similar Android apps[C]//The 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks. 2015: 9.

[11] 焦四辈, 应凌云, 杨轶, 等. 一种抗混淆的大规模 Android 应用相似性检测方法[J]. 计算机研究与发展, 2014, 51(7): 1446-1457.
JIAO S B, YING L Y, YANG Y, et al. An anti-obfuscation method for detecting similarity among Android applications in large scale[J]. Journal of Computer Research and Development, 2014, 51(7): 1446-1457.

[12] 卿斯汉. Android 安全研究进展[J]. 软件学报, 2016, 27(1): 45-71.
QING S H. Research progress on Android security[J]. Journal of Software, 2016, 27(1): 45-71.

[13] 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): 78-86.
WEN W P, MEI R, NING G, et al. Malware detection technology analysis and applied research of Android platform[J]. Journal on Communications, 2014, 35(8): 78-86.

[14] 张玉清, 王凯, 杨欢, 等. Android 安全综述[J].计算机研究与发展,2014,51(7):1385-1396.
ZHANG Y Q, WANG K, YANG H, et al. Survey of Android OS security[J]. Journal of Computer Research and Development, 2014, 51(7):1385-1396.

[15] 李挺, 董航, 袁春阳, 等. 基于 Dalvik 指令的 Android 恶意代码特征描述及验证[J].计算机研究与发展,2014,51(7): 1458-1466.
LI T, DONG H, YUAN C Y, et al. Description of Android malware feature based on Dalvik instructions[J]. Journal of Computer Research and Development, 2014,51(7):1458-1466.

[16] 张玉清, 方喆君, 王凯, 等. Android 安全漏洞挖掘技术综述[J]. 计算机研究与发展, 2015, 52(10): 2167-2177.
ZHANG Y Q, FANG Z J, WANG K, et al. Survey of Android vulnerability detection[J]. Journal of Computer Research and Development, 2015, 52(10): 2167-2177.

[17] 杨威, 肖旭生, 李邓锋, 等. 移动应用安全解析学: 成果与挑战[J]. 信息安全学报, 2016, 1(2): 1-14.
YANG W, XIAO X S, LI D F, et al. Security analytics for mobile apps: achievements and challenges[J]. Journal of Cyber Security, 2016, 1(2): 1-14.

[18] 刘新宇, 翁健, 张悦, 等. 基于 APK 签名信息反馈的 Android 恶意应用检测[J]. 通信学报, 2017, 38(5): 190-198.
LIU X Y, WENG J, ZHANG Y, et al. Android malware detection based on APK signature information feedback[J]. Journal on Communications, 2017, 38(5): 190-198.

[19] FAN M, LIU J, WANG W, et al. DAPASA: detecting Android piggybacked apps through sensitive subgraph analysis[J]. IEEE Transactions on Information Forensics and Security, 2017, 12(8): 1772-1785.

[20] 杨欢, 张玉清, 胡予濮, 等. 基于多类特征的 Android 应用恶意行为检测系统[J]. 计算机学报, 2014, 37(1): 15-27.
YANG H, ZHANG Y Q, HU Y P, et al. A malware behavior detection system of Android applications based on multi-class features[J]. Chinese Journal of Computers, 2014, 37(1): 15-27.

[21] ARP D, SPREITZENBARTH M, HUBNER M, et al. DREBIN: effective and explainable detection of Android malware in your pocket[C]//NDSS. 2014.

[22] YAN L K, YIN H. DroidScope: seamlessly reconstructing the os and dalvik semantic views for dynamic Android malware analysis[C]//USENIX Security Symposium. 2012: 569-584.

[23] 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.

[24] 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.

[25] 许艳萍, 马兆丰, 王中华, 等. Android 智能终端安全综述[J]. 通信学报, 2016, 37(6): 169-174.
XU Y P, MA Z F, WANG Z H, et al. Survey of security for Android smart terminal[J]. Journal on Communications, 2016, 37(6):169-174.

[26] LI L, LI D, BISSYANDE T F, et al. Understanding Android App piggybacking[C]//The 39th International Conference on Software Engineering Companion. 2017: 359-361.

[27] REAVES B, BOWERS J, GORSKI III S A, et al. Android: assessment and evaluation of Android application analysis tools[J]. ACM Computing Surveys (CSUR), 2016, 49(3): 55.

[28] GONZALEZ H, STAKHANOVA N, GHORBANI A A. Droidkin: lightweight detection of Android apps similarity[C]//International Conference on Security and Privacy in Communication Systems. 2014: 436-453.

[29] KIM D, GOKHALE A, GANAPATHY V, et al. Detecting plagiarized mobile apps using API birthmarks[J]. Automated Software Engineer-

ing, 2016, 23(4): 591-618.

[30] CHEN K, LIU P, ZHANG Y. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets[C]//The 36th International Conference on Software Engineering. 2014: 175-186.

[31] CHEN K, WANG P, LEE Y, et al. Finding unknown malice in 10 seconds: mass vetting for new threats at the Google-Play Scale[C]//USENIX Security. 2015: 15.

[32] ZHOU W, ZHOU Y, GRACE M, et al. Fast, scalable detection of piggybacked mobile applications[C]//The Third ACM Conference on Data and Application Security and Privacy. 2013: 185-196.

[33] CRUSSELL J, GIBLER C, CHEN H. Andarwin: scalable detection of semantically similar Android applications[C]//European Symposium on Research in Computer Security. 2013: 182-199.

[34] SHAO Y, LUO X, QIAN C, et al. Towards a scalable resource-driven approach for detecting repackaged Android applications[C]//The 30th Annual Computer Security Applications Conference. 2014: 56-65.

[35] GADYATSKAYA O, LEZZA A L, ZHAUNIAROVICH Y. Evaluation of resource-based App repackaging detection in Android[C]//Nordic Conference on Secure IT Systems. 2016: 135-151.

[36] SOH C, TAN H B K, ARNATOVICH Y L, et al. Detecting clones in Android applications through analyzing user interfaces[C]//The 2015 IEEE 23rd International Conference on Program Comprehension. 2015: 163-173.

[37] CORDELLA L P, FOGGIA P, SANSONE C, et al. A (sub) graph isomorphism algorithm for matching large graphs[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2004, 26(10): 1367-1372.

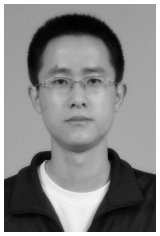
[38] LI M, WANG W, WANG P, et al. Libd: scalable and precise third-party library detection in Android markets[C]//The 39th International Conference on Software Engineering. 2017: 335-346.

[39] LIU C, CHEN C, HAN J, et al. GPLAG: detection of software plagiarism by program dependence graph analysis[C]//The 12th ACM SIGKDD International Conference On Knowledge Discovery And Data Mining. 2006: 872-881.

[40] LI L, BISSYANDÉ T F, KLEIN J, et al. An investigation into the use of common libraries in Android apps[C]//2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). 2016: 403-414.

[41] LAM P, BODDEN E, LHOTAK O, et al. The soot framework for Java program analysis: a retrospective[C]//Cetus Users and Compiler Infrastructure Workshop (CETUS 2011).2011.

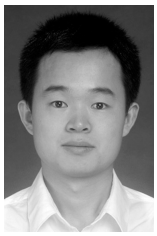
[作者简介]



汪润（1991-），男，安徽安庆人，武汉大学博士生，主要研究方向为 Android 安全与隐私、系统安全等。

王丽娜（1964-），女，辽宁营口人，博士，武汉大学教授、博士生导师，主要研究方向为系统安全、网络安全、信息隐藏等。

唐奔宵（1991-），男，湖北黄石人，武汉大学博士生，主要研究方向为移动安全与隐私、系统安全等。



赵磊（1985-），男，山东菏泽人，博士，武汉大学副教授、硕士生导师，主要研究方向为软件安全、系统安全等。