软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software [doi: 10.13328/j.cnki.jos.006918]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

LibPass: 基于包结构和签名的第三方库检测方法*

徐 建, 袁倩婷

(南京理工大学 计算机科学与工程学院, 江苏 南京 210094)

通信作者: 徐建, E-mail: dolphin.xu@njust.edu.cn

摘 要: 第三方库检测是 Android 应用安全分析领域的上游任务, 其检测精度对于恶意应用检测、重打包检测、隐私泄露等下游任务有显著影响. 为了提升检测精度和效率, 采用相似性比较的思想, 提出一种基于包结构和签名的第三方库检测方法, 命名为 LibPass. LibPass 以流水线式模式组合主模块识别、第三方库候选识别和细粒度检测等 3 个组件. 主模块识别方法区分主程序二进制代码与引入的第三方库二进制代码, 旨在提升方法检测效率. 在此基础上, 提出由第三方库候选识别和细粒度检测构成的两阶段检测方法. 前者利用包结构特征的稳定性来应对应用程序的混淆行为以提升混淆情形下的检测精度, 并利用包结构签名完成快速比对以识别候选第三方库, 达到显著降低成对比较次数、提升检测效率的目的; 后者在前者涮选出的候选中, 通过更细粒度但代价更高的相似性分析精确地识别第三方库及其对应的版本. 为了验证方法的性能和效率, 构建 3 个评估不同检测能力的基准数据集, 在这些基准数据集上开展实验验证, 从检测性能、检测效率和抗混淆性等方面对实验结果进行深入分析, 结果表明 LibPass 具备较高的检测精度, 检测效率, 以及应对多种常用混淆操作的能力.

关键词: 第三方库; 代码混淆; 安全分析; 签名

中图法分类号: TP311

中文引用格式: 徐建, 袁倩婷. LibPass: 基于包结构和签名的第三方库检测方法. 软件学报. http://www.jos.org.cn/1000-9825/6918.htm

英文引用格式: Xu J, Yuan QT. LibPass: Third-party Library Detection Method Based on Package Structure and Signature. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/6918.htm

LibPass: Third-party Library Detection Method Based on Package Structure and Signature

XU Jian, YUAN Qian-Ting

(School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing 210094, China)

Abstract: Third-party library (TPL) detection is an upstream task in the domain of Android application security analysis, and its detection accuracy has a significant impact on its downstream tasks including malware detection, repackaged application detection, and privacy leakage detection. To improve detection accuracy and efficiency, this study proposes a package structure and signature-based TPL detection method, named LibPass, by leveraging the idea of pairwise comparison. LibPass combines primary module identification, TPL candidate identification, and fine-grained detection in a streamlined way. The primary module identification aims at improving detection efficiency by distinguishing the binary code of the main program from that of the introduced TPL. On this basis, a two-stage detection method consisting of TPL candidate identification and fine-grained detection is proposed. The TPL candidate identification leverages the stability of package structure features to deal with obfuscation of applications to improve detection accuracy and identifies candidate TPLs by rapidly comparing package structure signatures to reduce the number of pairwise comparisons, so as to improve the detection efficiency. The fine-grained detection accurately identifies the TPL of a specific version by a finer-grained but more costly pairwise comparison among candidate TPLs. In order to validate the performance and the efficiency of the detection method, three benchmark datasets are built to evaluate different detection capabilities, and experiments are conducted on these datasets. The experimental results are deeply analyzed

* 基金项目: 国家自然科学基金 (61872186, 61802205)
收稿时间: 2021-02-25; 修改时间: 2021-06-09, 2022-07-08, 2022-08-21, 2023-01-13; 采用时间: 2023-02-08; jos 在线出版时间: 2023-07-26

in terms of detection performance, detection efficiency, and obfuscation resistance, and it is found that LibPass has high detection accuracy and efficiency and can deal with various common obfuscation operations.

Key words: third-party library; code obfuscation; security analysis; signature

市场调研机构 IDC 公布的研报显示 2019 年 Android 智能手机占据 85.9% 的市场份额. 大量开发人员借助于软件复用技术便捷地在 Android 应用 (application, APP) 中集成所需的多样化的功能, 实现快速开发, 使得第三方库 (third-party library, TPL) 俨然成为 APP 开发中必不可少的组成部分. 目前, 开发者使用的 TPL 要么来自于开源平台, 要么来自于特定的企业发布, 缺乏一个统一的 TPL 安全认证和管理平台, 这使得 TPL 的广泛应用带来了很多安全隐患. 例如, TPL 被用于收集位置信息和 UDID 来追踪用户[1], 收集用户的电子邮件地址[2], 读取联系人信息[3]等来谋取商业利益. 更有甚者, 恶意攻击者在原始 TPL 中植入恶意代码[4], 将其重新打包伪装成正常的 TPL, 最后被开发者集成到应用中, 对 APP 和 TPL 开发者的名誉造成损害, 对用户造成隐私泄露. 因此, TPL 检测已经成为 Android 应用安全分析领域的热点问题之一. Android 应用安全分析人员在分析过程中, 借助于 TPL 检测方法识别 APP 中集成的 TPL 并进而对其开展深入分析, 或移除, 或确定是否为恶意第三方库, 从而协助应用安全分析人员完成下游的安全分析任务, 如恶意应用检测、重打包检测等. 所以, 有必要开展 TPL 检测工作, 从 APP 中分离出引入的 TPL, 并对其进行安全性分析.

目前, TPL 检测面临着的诸多挑战. 第 1 个挑战是在字节码级别无法区分实现应用业务功能的主程序和引入的 TPL. 虽然在 APP 开发阶段主程序与 TPL 之间有明显的界限, 但是将源代码和引入的 TPL 编译成 Dalvik 字节码后则难以确定它们之间的界限. 第 2 个挑战是混淆技术在 APP 发布阶段被广泛应用, 这增加了 TPL 检测的难度, 并限制了检测精度. 表现在常见的标识符混淆技术使得基于包名、类名等的匹配方式失效; 代码压缩和优化、包重组等混淆技术使得 APP 中引入的 TPL 的代码结构与原始 TPL 的结构大相径庭, 导致误检、漏检等现象激增. 第 3 个挑战是尚缺乏完备的 TPL 原型库. 若 APP 中引入的 TPL 没有出现在 TPL 原型库, 则该 TPL 不可能被正确识别出. 最后一个挑战是低的检测效率. 特别是在采用成对比较思路的检测方法中, 需要将 APP 与 TPL 原型库中的每个 TPL 进行比较, 而 TPL 包含包、类等不同粒度的比较对象, 且粒度越小所需的计算代价增加, 难以满足快速检测需求.

研究人员提出了多种类型的方法来应对上述挑战. 最早提出的是基于白名单的检测方法[5−8], 代表性的工作有 DroidMoss[5]和 Juxtapp[6]. 该方法通过标识符来识别 TPL, 虽然不需要建立 TPL 原型库, 但是仍然需要预先构建尽可能完备的 TPL 白名单, 因此不能识别未在名单上的 TPL. 此外, 该方法也无法应对常用的标识符混淆技术. 针对上述不足, 研究人员随后提出了基于机器学习的检测方法[9−13], 代表性的工作有 LibRadar[9]以及 LibD[10]. 该方法通过特征工程提取刻画 TPL 的特征, 进而使用分类或聚类算法建立识别模型发现 TPL, 具备一定的抗混淆能力和发现新 TPL 的能力. 但是, 该方法需要建立在收集大量的 APP 作为训练样本的基础上, 且对于使用频率较低的 TPL 误检率较高. 此外, 大部分工作都是面向广告类型的 TPL 检测, 特征工程提取的特征不能直接迁移用于其它类型 TPL 检测. 最近的工作是基于签名的检测方法[14−16], 代表性的方法有 LibScout[14]、LibPecker[15]和 LibID[16]. 该方法为待分析的 APP 和 TPL 中的比较对象, 如包、类、方法等生成签名, 通过成对比较签名相似性识别 TPL. 基于成对比较的方法不可避免面临着检测效率低的困扰, 且会随着 TPL 规模扩充而日益严重, 进而影响了检测方法的实用性. 同时, 应用发布前的混淆操作, 可能使得导入的 TPL 与其原型有很大差异, 虽然在签名设计时融入了抗混淆的元素使得方法能抵御常用的标识符混淆和优化操作, 但对于面对更复杂的混淆操作, 如代码压缩和包重组等, 还是容易出现较高的误检和漏检, 从而影响了检测方法的精度.

为了提升检测效率和精度, 本文以待分析的 APP 和 TPL 对的多粒度签名为研究对象, 采用成对比较的思想, 提出了一种基于包结构树和签名的 TPL 检测方法, 命名为 LibPass. 首先, 提出主模块识别方法准确地区分主程序和引入的 TPL, 降低成对比较的次数以提升检测效率. 在此基础上, 提出了一种基于签名的两阶段检测方法, 即通过基于包结构树的快速检测和基于多级签名的细粒度检测相结合的方法实现 TPL 检测. 前者利用包结构特征的稳定性来应对应用程序的混淆行为以提升混淆情形下的检测精度, 并利用包结构签名完成快速比对以识别候选

TPL, 同样达到降低成对比较次数、提升检测效率的目的; 后者在前者涮选出的候选 TPL 中, 通过细粒度的多级签名相似性分析精确地识别第三方库及其对应的版本. 为了验证方法的性能和效率, 构建了 3 个基准数据集, 并在这些基准数据集上开展了实验验证, 从检测方法的性能、效率和抗混淆性等方面对 LibPass 进行了评价. LibPass 公开发布在 GitHub 社区 (https://github.com/njustbdag/LibPass) 供研究人员使用.

本文第 1 节阐述 Android 平台第三方库检测所需的背景知识以及领域工作现状. 第 2 节详细阐述提出的基于包结构树和签名的检测方法. 第 3 节在基准数据集上进行充分的实验验证, 并与领域先进方法进行的比较分析, 验证本文提出方法的有效性, 并对值得进一步关注的问题进行初步探讨. 最后总结全文.

1 背景知识和相关工作

1.1 常用混淆策略

混淆是增加逆向工程难度的最为常用的技术手段之一, Android 应用领域广泛采用这一技术保护知识产权, 对抗恶意第三方. 常用的混淆器主要有 ProGuard[17], Allatori[18]和 DashO[19], 其中 ProGuard 集成在谷歌的 Android Studio 开发平台, 是最为常用的混淆器. 按照是否改变编译后的 APP 字节码, 可以将混淆技术分为两类[20]: 平凡混淆和非平凡混淆. 鉴于本文采用的常用混淆器都仅支持非平凡混淆, 故仅描述这些非平凡的混淆技术以及与混淆器的对应关系, 如表 1 所示. 标识符混淆 (identifier renaming, IDR) 是将 APP 中包名、类名、方法名和属性名变形为无意义的字符串, 变形后的签名与原来的签名相比产生很大变化, 使得基于标志符或签名匹配的检测方式失效. 字符串加密 (string encryption, SE) 是对 APP 编译打包后的 classes.dex 中的字符串采用特定的加密算法进行加密, 并且在添加进解密算法用于运行时解密. 优化操作 (optimization operation, OO) 通过改变代码顺序、添加额外的流程控制条件和迭代条件改变方法的控制流和数据流等, 使得基于控制流、数据流等进行特征抽取而后采用学习方式进行分析的检测方法失效. 压缩操作 (shrinkage operation, SP) 通过移除 TPL 中没有在 APP 中使用的包、类、方法和属性来降低代码量, 导致 APP 中引入的 TPL 代码结构与原先的不同, 使得基于结构签名、成员成对比较方式的检测方法失效. 类重组 (class repackaging, CR) 通过将编译后的类移动到指定的包中实现重组, 导致包结构发生巨大变化, 使得基于包结构的检测方式失效.

表 1 Android 应用混淆器特征比较

混淆器	IDR	SE	OO	SP	CR
ProGuard[17]	√	×	√	√	√
Allatori[18]	√	√	√	√	×
DashO[19]	√	√	√	√	×

1.2 APP 开发和发布行为

APP 发布前通常应用混淆技术增强逆向工程难度, 同时也显著增加了 TPL 检测和分析难度. 这里简单回顾 APP 的代码组织结构、引入 TPL 的开发行为和混淆过程, 如图 1 所示. 一般说来, Android 应用和 TPL 采用 Java 语言编写的, 都以应用程序包 (application package, AP) 的形式组织代码. 按照代码来源划分, APP 中的包要么隶属于主模块, 要么隶属于非主模块, 其中非主模块是开发者植入的、供主模块调用的 TPLs, 而主模块则是开发者实现 APP 业务功能的代码部分. 所以, 非主模块与第三方库的关系是非主模块包含了 1 个或多个第三方库. 按照组织逻辑划分, 主模块和非主模块中的 AP 均可以划分为多个独立的逻辑单元, 分别称之为应用程序根包 (application root package, ARP) 和库根包 (library root package, LRP). 相应地, 隶属于 ARP 和 LRP 的 AP, 称之为应用程序子包 (application subpackage, ASP) 和库子包 LSP (library subpackage). 根包和子包的定义将在第 2.1 节给出.

在 APP 开发和发布过程中, 存在 3 种行为: 导入、混淆和优化, 其中混淆和优化是通过混淆器实现的. 具体地, 关于导入, 存在完整导入和定制导入两种情形, 其中完整导入是指 TPL 的所有 LRP 都出现在 APP 中, 定制导入是指 TPL 的部分 LRP、部分 LSP, 甚至是部分类出现在 APP 中, 又或者部分类中的方法和属性被改变. 将 TPL 导

入 APP 后, 在正式发布前采用的混淆器根据混淆配置对 APP 进行混淆, 改变标志符名称、移除未使用的包和类、将类移动到指定包、改变控制流等.

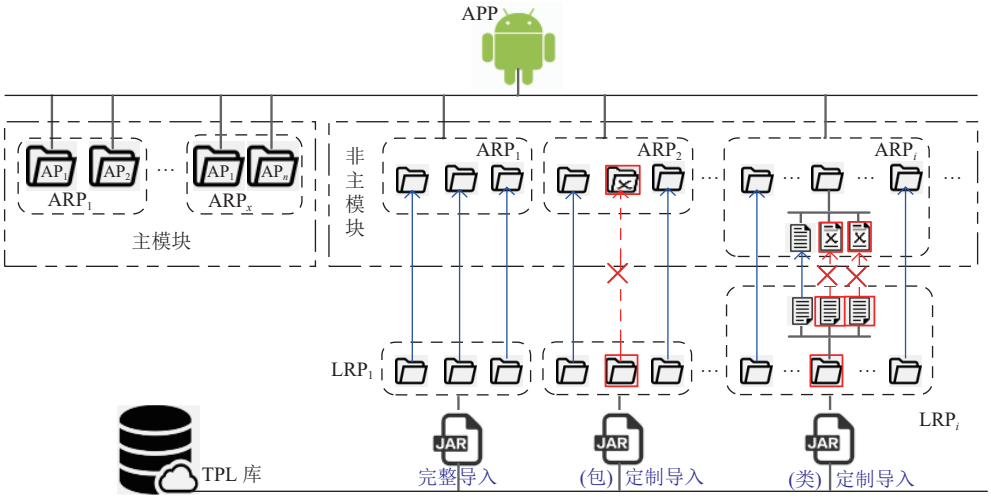


图 1 APP 组成以及与引入 TPL 的对应关系

1.3 相关工作

作为 Android 应用安全分析的重要前置任务之一, TPL 检测方法得到了很多的关注. 现有的检测方法大致可以分为 3 类: 基于白名单的方法[5,6,21−24], 基于机器学习的方法[9−13,25−27]和基于相似性比较的检测方法[14−18,28−33]. 下面逐一阐述代表性的方法, 总结其基本思路和优缺点, 归纳对比结果如表 2 所示.

基于白名单的方法是 TPL 检测最初使用的方法. 该方法的基本思想是预先建立一个知名 TPL 的白名单, 列出每个 TPL 的名称以及库中包含的包名, 将从待检测的应用中提取的包名与白名单进行比较, 识别应用中引入的 TPL. DroidMoss[5]是第 1 个基于白名单方式的检测方法, 并用于 Android 应用重打包检测中. Chen 等人[21]构造了 73 个流行 TPL 的白名单用于 Android 应用克隆检测. 最大规模的白名单是由 Li 等人[22]采集的, 包含 5 000 个不同的 TPL. 此外, Juxtapp[6]、AdRisk[7]和 Lin 等人[24]也应用该方法识别应用中的广告库、社交库、金融库等, 进而进行恶意应用分析. 该方法的一个显著特点是简单、高效, 但是缺点同样明显. 首先检测效果极大依赖于白名单中 TPL 覆盖率, 而建立一个完备的白名单是困难的, 在持续有新的 TPL 出现的现实情境下保持白名单的完备性就显得愈发困难. 其次, 该方法不能用于识别混淆后的 TPL. 上述不足严重影响了该方法的广泛应用.

为了能从混淆后的 Android 应用中识别第三方库, 机器学习技术被应用于第三方库检测. Narayanan 等人首次提出了基于机器学习的第三方库检测方法—AdDetect[12], 用于识别第三方广告库. 该方法的基本思想是通过特征工程将待分析的 Android 应用转为特定的特征向量, 且确保每个特征在混淆情形下具备稳定性, 进而应用传统的机器学习方法建立 TPL 识别器用于第三方库. 该方法具有抗混淆, 准确率高的优点, 主要原因在于借助于领域专家的经验开展的特征工程很好地捕获了广告库易于区分的、抗混淆的特征, 如 View 组件, 权限和 API 等. 类似的方法还有 PEDAL[13]等, 取得了不错的应用效果. 然而, 这一方法也存在不足之处. 首先, 方法仅局限于检测广告类的 TPL, 不能直接用于检测任意 TPL, 除非重新针对性的人工建立有效的特征刻画 TPL. 其次, 方法在构建识别器的过程中依赖于应用与 TPL 之间关系标注信息, 而本领域尚缺乏大规模的带标注的基准数据集, 且人工标注一个 Android 应用引入的 TPL 是一个耗时且极易产生漏报、误报的工作. 针对上述问题, 研究人员深入分析了 APP 引入 TPL 的开发过程, 发现了两个比较突出的特点: 一是特定的 TPL 会被多个应用集成使用, 且该 TPL 越流行, 在应用中出现的频率越高; 二是绝大多数的应用开发者在引入 TPL 时基本不做修改和裁剪, 于是不同应用中的同一个 TPL 具备相同的特征. 利用上述特点, 研究人员提出了基于聚类的检测方法, 该方法通过对大量的应用进行聚

类使得不同应用中引入的相同 TPL 会聚集到同一个簇, 从而摆脱对人工标注的 APP 与 TPL 对的依赖. 代表性的方法有: AnDarwin[25]、WuKong[12]、LibRadar[21]和 LibD[22,27]. AnDarwin 和 WuKong 是最早的基于该思路的方法, 它们以应用中包含的所有子包为单位进行聚类, 将函数调用的 API 作为每个子包的特征向量, 对子包进行聚类识别 TPL. 这一方法依赖于假设 TPL 被众多 APP 使用且未经改变. 然而, 大部分情况下引入的 TPL 会通过混淆技术进行优化, 使得方法的检测精度不高. 此外, 该方法不能识别 APP 中具体使用的 TPL 版本. 针对上述不足, LibRadar 和 LibD 提出了多层级聚类的概念, 考虑了子包路径之间的包含关系来提升聚类准确性. 基于聚类的检测方法与之前的方法相比, 只要选取了合适的与 TPL 相关的特征, 则同样具备抗混淆, 准确率高的优点. 此外, 该检测方法能发现新出现的 TPL, 能用于识别各种类型的 TPL, 而无需第三方库类型的先验知识. 当然, 该方法也同样存在一些不足之处. 首先, 基于聚类的检测方法需要收集大量的 Android 应用; 其次, 对于使用频率低的 TPL 识别能力依然有限, 越流行的 TPL 越容易识别; 最后, 在使用复杂混淆操作如代码优化和包结构平坦化的应用中人工设计的特征不够稳定, 导致检测方法的抗混淆能力下降, 不能准确识别 TPL.

目前, 最新的方法是基于相似性比较的检测方法, 其基本思路是采用比较方式, 先为 APP 和 TPL 中的包、类生成签名作为比较对象, 而后度量签名相似性, 当相似性超过阈值时识别出引入的 TPL. 代表性的方法有 LibScout[14]、LibPecker[15]、LibID[16]和 Orlis[28]. APK 开发和发布过程中引入的混淆操作导致的标志符名称改变、包结构改变、控制流改变等是造成基于相似性比较的 TPL 检测方法误检和漏检的主要原因. 因此, 现有的方法都致力于探寻合适的表示方法刻画 TPL, 设计混淆后仍然保持不变的特征, 以提升检测精度.

LibScout 首次提出了 Merkle 树表示方法刻画待比较的 APP 和 TPL, 最顶层存储 TPL 或 APP 签名、第 2 层存储包签名、第 3 层为类签名、最底层为方法签名, 签名的生成方式采用自底向上方式实现, 即先产生方法签名, 而后有类中的所有方法签名产生类签名, 依此类推. 在匹配阶段, 则采用自顶向下的方式进行匹配, 若 TPL 的 Merkle 树和 APP 中特定包的 Merkle 树之间的相似度超过阈值, 则认为该 APP 中引入了该 TPL. LibScout 采用的签名机制确保在引入的 TPL 结构不变的情形下能有效应对标识符混淆操作, 然而在使用优化操作、压缩操作或类重组混淆技术情形下则对抗能力不足, 尤其是在较多无用的类被删除的情况下, 会产生较高的漏检率. 此外, LibScout 在生成类签名时, 仅考虑了类的方法签名容易产生误报. 针对 LibScout 的签名机制存在的问题, LibPecker 在沿用 Merkle 树表示方法的基础上, 对签名机制进行了改进, 不仅考虑类自身信息, 如类内的函数信息和类的重要性, 还考虑类与类之间的依赖关系, 设计了基于加权方式的签名机制. 与 LibScout 相比, 虽然 LibPecker 提高了 TPL 检测的召回率和准确率, 但是仍然仅能有效应对标志符混淆而无法有效对抗优化操作、压缩操作或类重组等混淆.

不同于 Merkle 树表示方法, LibDetect[31]和 Orlis 设计基于方法级字节码特征的表示方法, 该类方法的基本思路是采用不同混淆操作后主要的代码改变体现在方法层面, 因此从方法层面捕获混淆后不变特征是应对混淆的有效方法. 例如, LibDetect 设计了一组由 5 种不同的方法抽象表示构成的特征用于刻画 TPL 或 APP 的方法, 包括字节代码、无地址表示、匿名表示、结构保持表示以及模糊结构保持表示, 且随着抽象程度加深, 更抽象的表示能应对更多种混淆操作, 同时也可能导致更高的误匹配. 在 TPL 检测过程中, 给定 APK 中的方法 m, 借助于方法级的 5 种不同抽象表示从 TPL 库中查找一个包含与方法 m 最为相似的方法的 TPL 作为候选, 依据方法-类隶属关系将方法相似累积到类相似, 依据类-包隶属关系将类相似累积到包相似, 从而确定最终的 TPL. 然而, 在类重组混淆情形下, 因为类隶属包会的随机分配使得从类级别相似累积计算包相似失效, 从而导致 TPL 检测不准确, 同样无法有效应对类重组混淆.

LibID 引入了类签名字典的概念用于表示一个 APK 或 TPL, 其基本思想是为在签名生成阶段, 为 APK 或 TPL 中的每个类生成一组类签名, 由所有类的类签名共同刻画一个 APK 或 TPL, 进而在检测阶段, 以类作为比较对象, 兼顾类相似和类依赖关系相似性确定最终的 TPL. 每个类包含多个基本块, 通过构造控制流图可以从图上抽取所有的基本块, 将基本块连同 4 个类特征, 包括类访问标识、父类名、类接口和方法描述一起生成类签名, 能抵御标志符混淆、代码压缩、控制流改变和包扁平化. 在真实数据集上的实验验证结果表明在 ProGuard 缺省混淆配置下, LibID 的检测精度约为 92%, 而采用扁平化混淆情形下, 检测精度下降到约 55%, 仍然不能有效应对扁平化混淆.

表 2　与现有的基于相似性的方法比较

检测方法(年份)	表示形式	采用的相似性度量	相似性比较方式	方法优点或贡献	本文工作的针对性改进	检测效率(s)	*F*1-score (%)		是否公开基准数据集	源码可用
							无混淆	有混淆		
LibScout[14] (2016)	固定深度Merkle树	类库相似性、包相似性、类相似性和方法相似性	树自顶向下逐层比较	首次提出Merkle树结构表示APP或TPL中的包, 按照函数、类、包、TPL (或APP)等层级提取特征	1) 引入包结构相似度量进行候选TPL筛选, 解决人工设定相似性阈值的问题; 2) 提出基于多级签名的细粒度检测方法, 解决签名机制缺陷导致的漏报; 3) 设计了一种新的基于包结构树的表示方法应对方法移除、类移除、包扁平化等混淆技术的干扰	~2.8	90.64	24.39	×	√
LibDetect[31] (2017)	一组方法级字节码特征	方法相似性、类相似性	方法相似性快筛、而后类相似性	设计了5种方法字节码抽象表示以应对方法和类混淆	1) 提出基于多级签名的细粒度检测方法, 解决签名机制缺陷导致的漏报和误报; 2) 设计了一种新的基于包结构树的表示方法应对方法移除、类移除、包扁平化等混淆技术的干扰	<7	83.55	17.00	×	×
Orlis[28] (2018)	一组方法级字节码特征	APP-Lib 相似性、类签名相似性	APP-Lib 相似性快筛, 而后类相似性精确查找	考虑方法及其间调用关系构建方法特征映射为字符串作为APP或TPL摘要用于相似性比较, 能抵御代码变更	引入包结构相似度量进行候选TPL筛选, 解决人工设定相似性阈值的问题	>800	83.00	24.19	×	√
LibPecker[15] (2018)	固定深度Merkle树	加权模糊类签名相似性	逐对比较	改进LibScout签名机制, 考虑依赖关系提出基于加权的签名机制, 以提升检测精度	1) 引入包结构相似度量进行候选TPL筛选, 解决人工设定相似性阈值的问题; 2) 设计了一种新的基于包结构树的表示方法应对方法移除、类移除、包扁平化等混淆技术的干扰. 3) 设计主模块识别和基于包结构的快速检测组件改进检测效率	>500	91.37	23.82	×	×
LibID-A[16] (2019)	增强的类签名字典	类签名相似性、依赖相似性、库相似性	类签名相似性快筛, 而后依赖相似性, 最后库相似性	设计基本块连同类特征的签名机制提升扁平化、代码优化和控制流改变混淆情形下的检测精度; LSH哈希方式避免成对比较, 提高效率; 能区分导入的TPL版本	引入包结构相似度量进行候选TPL筛选, 解决人工设定相似性阈值的问题	< 50	92.05	55.36	√	√

表 2　与现有的基于相似性的方法比较 (续)

检测方法 (年份)	表示形式	采用的相似性度量	相似性比较方式	方法优点或贡献	本文工作的针对性改进	检测效率 (s)	*F*1-score (%)		是否公开基准数据集	源码可用
							无混淆	有混淆		
LibSeeker[33] (2019)	固定深度Merkle树	包相似性、类相似性	树自底向上逐层比较	改进LibScout, 新增函数特征向量降低误匹配, 通过参数自整定方式自动调优, 无需人工设定	—	—	—	—	×	√
PanGuard[32] (2019)	API调用图	图相似性	逐对比较	考虑包结构和内容信息刻画TPL或APP应对多种混淆, 包括代码压缩、优化和标志符混淆; 通过边缘计算提升检测效率	—	—	—	—	×	√
LibPass	包结构树	包结构树相似性、加权类签名相似性	包结构树签名快筛, 而后类相似性精确查找	设计了基于包结构树的表示方法, 提出了包结构树整形方法改变包排序方式和重命名以应对标志符混淆、压缩操作; 能区分导入的TPL版本; 无需人工设定相似性阈值	—	<15	94.92	54.22	√	√

现有方法的归纳和对比分析如表 2 所示. 本文研究工作的目标是从给定的 Android 应用 APK 文件中识别集成的特定版本的第三方库, 并不进一步区分植入的第三方库是否为广告库或者恶意库. 现有的基于白名单的方法和基于机器学习的方法可以达成与本文工作相同的目标. 然而, 基于白名单的方法和基于机器学习的方法除了可以达成与本文研究工作相同的目标外, 还可以用于识别植入的广告库和恶意库. 以基于白名单的 TPL 检测方法为例, 可以通过在白名单中设定已知的广告库或者恶意库清单, 一旦识别到出现在清单上的第三方库, 则能在识别植入的第三方库的同时给出诸如该第三方库是否为广告库或恶意库的标注信息. 从这个角度上来说, 基于白名单的方法在给出白名单中每个库的标注信息前提下, 可以提供更丰富的功能. 因此, 表 2 重点比较了基于相似性比较的 TPL 检测方法. 表 2 中的比较项“表示形式”是一种 APK 或 TPL 的抽象表达, 刻画了有利于比较的 APK 或者第三方库的变形; “采用的相似性度量”则表示基于特定的表示形式所设计的具体度量; “比较方式”刻画了所设计的相似性度量的应用方法和过程. 现有的基于相似性比较的 TPL 检测方法在上述 3 个比较项上做了精心设计以满足检测需求. 这 3 个方面是有着紧密联系的. 表示形式往往影响着检测能力, 如类库区分能力、混淆对抗能力. 相似性度量则依赖于表示形式, 而比较方式依赖于表示形式和相似性度量. 此外, 值得注意的是表中列出的不同方法的检测效率和精度来自于对应文献, 并不是在统一的基准数据集上的实验结果, 鉴于方法评价实验中采用的混淆策略、数据集规模等上存在显著不同, 因此结果并不具备可比性. 这里, 检测效率是指给定一个待分析 APK (以.apk 文件存在的 APP), 输出 APK 引入的 TPL 清单的时间开销. 基于白名单的方式, 时间开销是包名匹配时间. 基于机器学习的方式和基于相似性比较的方式, 时间开销有 APK 特征抽取和匹配两阶段构成. 因为基于相似性比较的方式建立在 TPL 本地库上, 而本地库规模大于基于机器学习方式建立的分类器或聚类个数, 所以需要更多的时间开销.

与现有的工作相比, 本文的创新之处在于采用了基于包结构树的表示方式, 并提出包结构树整形方法来有效应对标志符混淆和代码压缩, 并一定程度上缓解扁平化混淆带来的检测精度下降. 包结构树是刻画结构信息的一种天然表示, 缺省的包结构树同一层上隶属于相同父节点的子包以字典序排序. 然而, 集成到应用程序中的 TPL 在应用正式发布前会进行混淆, 可能删除部分库中未被使用的子包、改变类的隶属关系, 以及混淆包名等, 这些混淆操作会导致包结构树显著地不同于缺省包结构树, 导致基于包结构树相似性比较方式识别 TPL 的方法准确性低. 因此, 引入包结构树整形方法, 先改变子包排序方式, 取代字典序以子包重要性作为包排序方式, 包重要性越高则在混淆阶段被移除的可能性更小, 将其放置在同一层上的更左侧位置, 而后对所有包名进行重命名, 进一步地借助于包结构树签名匹配筛选可能引入的 TPL. 在包结构树签名生成阶段, 类似于 LibID, 本文同样考虑了基于依赖关系的类签名方法, 主要动机在于类依赖关系是在混淆情形下仍能得以保持的稳定特性能提升类签名抗混淆能力. 在类签名生成时考虑类的方法数量和类的依赖关系作为类重要性评价的两个因素计算类权重, 构建了基于加权机制的类签名方法, 这与 LibPecker 中权重度量方法是不相同的. 此外, 与最为相近的工作 LibScout 和 LibPecker 相比, 还进行了效率和性能方面的改进和优化形成了新的解决方案 LibPass. 针对检测效率低的问题, 设计主模块识别和基于包结构的快速检测组件, 前者用于排除 APP 中开发人员自行开发的功能模块, 后者快速从大规模第三方库本地库中筛选出候选第三方库, 经过验证显著提升了检测效率. 针对检测中的误检率和漏报率高的问题, LibPass 提出了一种基于签名的两阶段检测方法, 即通过基于包结构树的快速检测和基于多级签名的细粒度检测相结合的方法实现 TPL 检测. 前者利用包结构特征的稳定性来应对应用程序的混淆行为以提升混淆情形下的检测精度, 并利用包结构签名完成快速比对以识别候选 TPL, 同样达到降低成对比较次数、提升检测效率的目的; 后者在前者涮选出的候选 TPL 中, 通过细粒度的多级签名相似性分析精确地识别第三方库及其对应的版本.

2 LibPass

给定任意的安卓应用 APK 文件和一组本地第三方库 TPLs 作为 TPL 检测方法的输入, 通过检测方法输出为该 APP 中植入的带版本号的 TPL 清单, 且每个 TPL 保存为一个对应的 jar 文件. 对于采用成对比较思路的检测方法而言, 给定一个待分析的 APK, 理论上需要与 TPL 本地库中每个 TPL 进行逐一比较, 进而确定该 APK 中引入

的 TPL. 假设 APK 有 m 个包, TPL 本地库的规模 n, 且每个 TPL 最多有 1 个包, 在非混淆情形下, 可以通过包名匹配确定 APK 中引入的 TPL, 其计算复杂度为 $m \times n$. 当 n 较大时, 产生较高的比较代价, 尤其是在混淆情形下, 无法借助于包名实现 TPL 检测, 不得不采用更细粒度的对象进行比较, 如以类为比较对象, 这进一步增加了计算复杂度, 进而影响实用性. 本文旨在确保检测精度的前提下, 提升检测效率, 试图通过缩小 m 和 n 来达成这一目标, 为此提出的一种基于包结构树和签名的检测方法, 命名为 LibPass, 其框架如图 2 所示, 整个框架由为 3 个关键组件构成, 包括主模块识别、基于包结构树的候选 TPL 筛选和基于多级签名的检测方法.

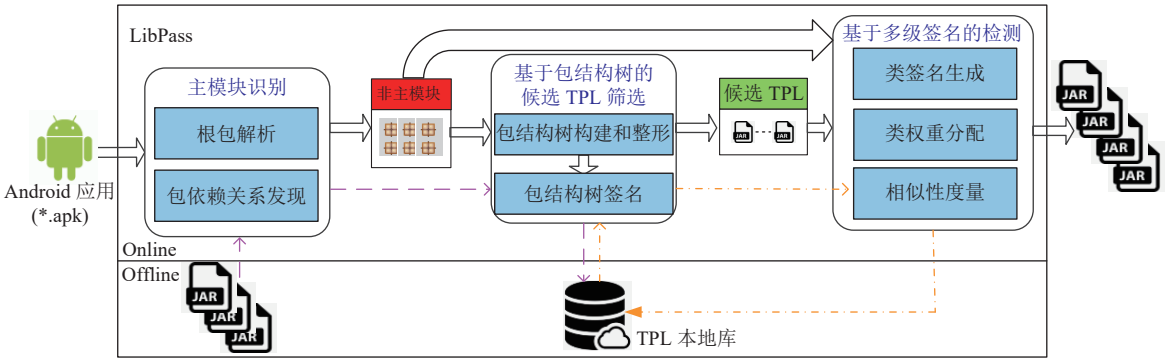


图 2 LibPass 的组成框架

主模块识别组件负责解耦待分析的 APK, 将其拆分为主模块和非主模块. 该组件引入根包的概念, 将 APK 的包以更粗粒度的根包形式加以组织, 建立包依赖关系图, 通过依赖强度分析识别主模块. 假设主模块和非主模块分别包含 m_1 和 m_2 个包, $m_1 + m_2 = m$, 通过主模块识别可以过滤掉 m_1 个包, 成对比较计算复杂度降为 $m_2 \times n$. 对于本地库中的 TPL, 以离线方式对其预处理, 识别 TPL 的根包.

基于包结构树的候选 TPL 筛选组件负责从庞大的 TPL 本地库中快速搜索可能引入的 TPL. 对待分析 APK 的根包, 将其表示为一棵包结构树. 为了对抗混淆操作以提升检测精度, 对包结构树进行了整形和重命名, 在此基础上提出了一种包结构树签名方法. 对于 TPL 的根包, 同样为其生成包结构树、进行整形和重命名, 最后进行签名, 并存储在 TPL 本地库中. 于是, TPL 候选筛选问题转变为 APK 中根包签名与 TPL 中根包签名的哈希值比较问题, 有很多现有算法可以快速实现这一目标. 假设筛选出的 TPL 数目为 n', 有 $n' \ll n$, 因此, 细粒度比较对象上进行成对比较的计算复杂度降为 $m_2 \times n'$.

基于多级签名的细粒度检测组件负责为 APK 非主模块中的根包从候选 TPL 中确定引入的唯一一个 TPL. 该检测过程不再以包作为比较对象, 而是以类为比较对象. 鉴于影响检测精度的主要因素是混淆, 提出了基于类内部信息和类间依赖关系的类签名机制, 同时以 APK 为参照对象, 考虑加权成对比较方法, 以降低混淆对于检测精度的影响. 对于给定的 APK, 通过 LibPass 最终输出该 APK 引入的所有 TPL.

2.1 主模块识别

Android 平台采用 Java 作为开发语言, 因此无论是主模块, 还是非主模块, 都是以层次包结构形式组织代码的. 通常, 每个模块都有自己特有的命名空间, 用于区分不同功能意图的模块. 然而, 在混淆情形下, 主模块和非主模块的包名、类名被混淆为无意义的短字符, 编译后的类通过扁平化操作移动到其他包中进行重组, 使得主模块与非主模块的边界模糊, 显著增加了 TPL 检测的难度. 因此, 有必要研究主模块识别方法实现主模块与引入的 TPL 的分离, 从而减少待分析的包, 从而有利于提高检测性能和效率. 本文提出了一种基于元信息和包依赖关系的主模块识别方法, 其基本思路是对于没有混淆的 APK, 采用基于元信息的方式, 而对于混淆的 APK 采用基于包依赖关系的方法实现模块解耦. 采用这一思路的事实依据是研究报告[34]表明不少于 55% 的 APK 是混淆的. 因此, 针对混淆和非混淆的情形, 采用不同的解决方法是直截了当的想法.

在非混淆的情形下, 每个 APK 解压后的 AndroidManifest.xml 文件中包含的关于“package”的元信息, 该信息

可以用于识别 APK 的主模块. 在混淆情形下, 虽然 APK 中关于“package”的元信息仍然可以获取, 但是代码中包名被混淆, 无法直接通过与元信息中的包名进行比较来辨别. 因此, 基于元信息的方式不再适用. 在混淆情形下, 尽管标识符被混淆、类通过扁平化被重组, 但是包之间依赖关系仍然得以保持, 于是提出了基于包依赖关系的主模块识别方法, 该方法将 APK 转换为一个包依赖关系图, 节点表示包, 边表示节点之间的依赖关系, 赋予边权重刻画不同依赖关系的强度, 计算每个包的依赖强度, 将依赖强度最大的包作为主模块. 基于元信息和包依赖关系的主模块识别过程如算法 1 所示.

算法 1. 主模块识别 (primary module identification, PMI).

输入: *apk*, 依赖强度阈值 δ_d ;
输出: list.

```
1. boolean obfuscated = isSymbolObfuscated(apk);
2. ARPs arps = getAllRootPackages(apk, obfuscated);
3. if (!obfuscated)
4.    getPrimaryModulesFromMetaData();
5. else
6.    pdg = constructPDG(arps);
7.    list=sortModulesbyDependencyStrength(arps);
8. end if
9. return list;
```

算法 1 第 1 行函数调用 isSymbolObfuscated() 用于判断给定的 APK 是否经过了混淆. 尽管不同的混淆器支持不同类型的混淆策略, 但是标志符混淆却是任何混淆器都支持的基本配置, 同时也是在绝大部分混淆 APK 中都加以应用的混淆技术. 标志符混淆后的代码符号与程序员书写的代码符号相比是极其不自然的, 因此, 从符号的不自然性角度来识别一个 APK 是否经过了混淆是可行的. 鉴于本文的重点是研究 TPL 检测方法, 而不是 APK 混淆检测方法, 故直接应用基于 NLP 的混淆检测方法[35], 该方法使用交叉熵来捕获标识符的不自然性, 具有高准确性.

算法 1 第 2 行的函数调用 getAllRootPackages() 实现将 APK 划分为若干 ARP. 下面先给出不同类型包的定义, 然后再给出根包划分方法.

定义 1. 应用程序子包 (application subpackage, ASP). 对于任意的包, 如果该包中有类或接口, 则称该包为应用程序子包, $asp=\{C_1,\ldots,C_i,\ldots,C_{n1},I_1,\ldots,I_j,\ldots,I_{n2}\}$, 其中 C_i 和 I_j 分别表示该子包中的类和接口.

借助于子包的概念, 可以将 APK 理解为由若干相互之间存在依赖关系的子包构成的, 这里考虑以下 4 种情形的依赖关系.

- 继承关系, 是指一个类 (称为子类、子接口) 保留并扩展了另外的一个类 (称为父类、父接口) 的功能.
- 实现关系, 是指一个类实现一个接口的功能, 实现是类与接口之间最常见的关系.
- 类属性关系, 是指一个类 B 的实例以类属性的形式出现在类 A 中, 则 B 是 A 的关联类, 体现了 A 与 B 之间相对强的语义依赖性.
- 参数引用关系, 是指类 B 以形参的形式出现在类 A 的一个或多个方法声明中, 体现了相对弱的语义依赖性.

在以往的工作[12]中认为上述 4 种依赖关系的强度依次为继承、实现、类属性、参数引用, 为它们分配一个依赖强度值 w_i, $1\leqslant i\leqslant 4$, 满足 $\sum_{i=1}^{4}w_i=1$. 不失一般性, 设为 $w_1=0.4$, $w_2=0.3$, $w_3=0.2$, $w_4=0.1$. 进一步地, 若引入无依赖关系的情形, 并将其依赖强度设为 0, 记为 $w_5=0$, 因此仍有 $\sum_{i=1}^{5}w_i=1$. 基于上述子包及其依赖关系可以构建包依赖关系图来表征 APK, 如定义 2 所示.

定义 2. 包依赖关系图 (package dependency graph, *PDG*). 包依赖关系图是一个有向图, $PDG=<V,E>$, 其中 V 表示应用程序中包的集合, $V=\{p_1,\ldots,p_n\}$, E 表示任意两个包之间以依赖关系为边的集合, $E=\{p_i\overset{s_{ij}}{\rightarrow}p_j|1\leqslant i\neq$

$j \leqslant n\}$, 其中 s_{ij} 表示包 p_i 与 p_j 之间的依赖强度.

定义 3. 依赖强度. 任意两个包 p_i 与 p_j 之间的依赖强度 s_{ij}, $s_{ij} = \sum_{x=1}^{n}\sum_{y=1}^{m} w_{t(e_x,e_y)} | e_x \in p_i, e_y \in p_j$, 其中 $t(e_x, e_y)$ 获取 e_x 和 e_y 之间的关系类型, 值为 1 到 5.

定义 4. 应用程序根包 ARP. 应用程序根包是由 m 个具有最长共同路径, 且有强依赖关系的子包构成的, $arp = \{asp_1, \ldots, asp_i, \ldots, asp_m\}$, 满足两个约束条件: 1) $path(asp_1) \cap \ldots \cap path(asp_i) \cap \ldots \cap path(asp_m) \neq \varnothing$; 2) 对于任意的子包 asp_i, 至少存在一个子包 asp_j, $j \neq i$, 使得 asp_j 与 asp_i 的依赖强度 s_{ji} 大于依赖强度阈值 δ_d.

第 1 个约束条件表明同一个根包中的所有子包应该具有相同的命名空间, 这符合 Java 语言中包命名机制. 但是仅依赖于相同的路径前缀来判别两个或多个子包是否属于同一个根包是不充分的. 以两个较为广泛使用的第三方库 com.google.common 和 com.google.zxing 为例, 它们具有相同的路径前缀 com.google, 但是不存在依赖关系. 因此, 第 2 个约束条件用于增加同一个根包中子包间存在依赖关系的判定, 仅查找一个子包与该根包中至少一个子包有依赖关系.

一旦判定 APK 没有混淆, 算法第 4 行直接通过解析其 AndroidManifest.xml 文件中包含的关于“package”的元信息获取主模块的名称, 否则算法第 6 行为混淆 APK 构建根包级别的包依赖关系图. 值得注意的是, 这里仍然沿用定义 2 和 3 来构造包依赖关系图, 只不过需要将节点表示的 ASP, 替换为 ARP. 算法第 7 行负责在 *PDG* 上度量每个节点的依赖强度, 并且根据依赖强度对模块进行降序排列; 算法第 8 行将所有的模块按照依赖强度大小逆序排列, 依赖强度最大的模块作为主模块.

2.2 基于包结构树的候选 TPL 识别方法

应用主模块识别方法排除 APK 中的主模块后剩余部分以根包的形式组织, 每个 ARP 都可能源自于特定的 TPL. 为了提升混淆情形下的检测精度, 提高检测效率, 提出了一种基于包结构树的候选 TPL 方法, 该方法利用包结构特征的稳定性来应对混淆, 并借助于包结构签名完成快速比对识别候选 TPL. 该方法由包结构树生成、包结构树签名和基于签名的候选筛选等过程构成.

首先, 构造包结构树. 无论是 ARP, 还是 LRP, 都是由若干个具有层次关系的子包构成, 因此根据包层次关系以点分隔法可以为一个 ARP 或 LRP 构造为一棵缺省的包结构树, 同一层上隶属于相同父节点的子包以字典序排序. 以第三方库“glide-3.8.0”的根包“com.bumptech.glide”为例, 生成的缺省包结构树如图 3(a) 所示. 集成到应用程序中的 TPL 在应用正式发布前会进行混淆, 可能删除部分库中未被使用的子包, 还可能混淆包名, 这些混淆操作会导致包结构树显著地不同于所引入 LRP 的缺省包结构树, 从而使得通过比较包结构树的方式识别 TPL 的方法准确性低. 为了对抗混淆, 提升检测准确率, 引入了包结构树整形概念, 涉及两个操作, 一是改变子包排序方式, 二是对包名进行重命名. 具体地, 考虑以子包重要性排序替换缺省包结构树中同一层上隶属于相同父节点的子包以字典序排序的方式, 同时对所有包名进行重命名. 子包重要性取决于在包结构树中每个子包拥有的孩子节点个数, 即在包结构树中拥有更多孩子节点的子包承载更多功能, 其重要性更高, 在混淆阶段被移除的可能性更小, 因此将其放置在同一层上的更左侧位置. 按照自顶向下方式对包结构树进行整形, 先计算同一层上的节点作为根节点, 子树拥有的节点数, 并按照从左到右降序排列, 即拥有孩子节点数多的节点放置在树的左侧; 然后, 将同一层上有相同的孩子节点数的节点按照节点名字母顺序降序排列; 最后, 对整形后的包结构树上的节点进行重命名, 即将每层隶属于同一个父节点的节点从左到右用大写字母升序进行替换. 图 3(a) 中的包结构树经过整形后得到了如图 3(b) 所示的结果, 其中括号中字符串表示整形操作中重命名后结果, 从图中可以看出第 2 层上节点 load 包含的子包数目最多, 是构成第三方库的重要组成部分, 在混淆阶段被移除的可能性小.

在为待比较的 ARP 和 LRP 构造包结构树后, 直接比较它们对应的包结构树仍然需要较大的计算代价, 为此引入包结构树签名, 如定义 5 所示, 通过深度优先方式遍历包结构树生成包结构树签名.

定义 5. 包结构树签名 (package structure tree signature, PSTS). 一个根包的包结构树签名是一个哈希值, $Sig_{pst}(p) = hash\left(\sum_{i}^{n} strcat(name(p_i), ',')\right)$, $name(p_i)$ 表示子包 p_i 的名称, $strcat(\cdot,\cdot)$ 表示字符串连接操作, $hash(\cdot)$ 表示通过哈希算法计算字符串的的哈希值, 这里采用 16 位的 MD5 算法计算哈希值.

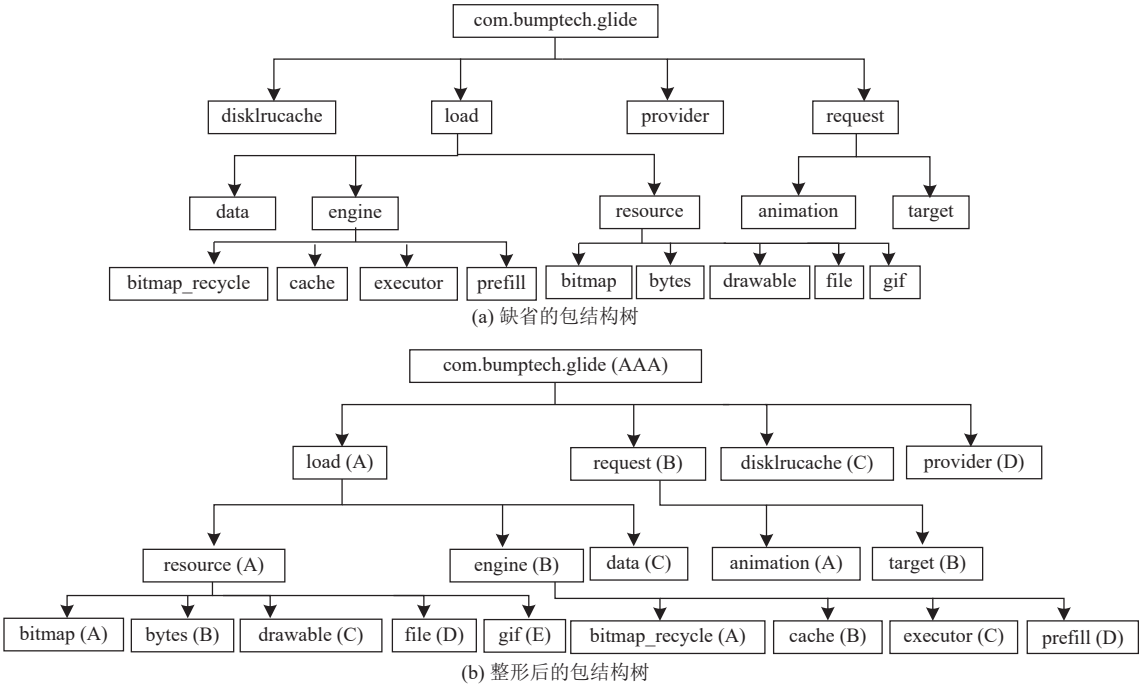


图 3 glide-3.8.0 的 LRP 对应的包结构树

对于给定的根包对 $<arp, lrp>$, 关于 lrp 是否为 arp 对应的 TPL 的判定转化为对它们的包结构树相似性分析问题, 本文提出了一种基于包结构树的候选 TPL 识别方法, 综合利用了根包对应的包结构树蕴含的 3 种信息, 即包名、包结构和包结构树签名, 来快速识别候选 TPL, 如算法 2 所示. 算法第 3 行通过两个根包的包名和包结构树签名进行筛选, 若包名和包结构树签名均相同, 则表明 arp 源自于 lrp, 加入候选列表; 否则算法第 6–8 行引入包结构相似度的概念度量两个根包的包结构树相似性, 如定义 6 所示.

算法 2. 基于包结构树的候选 TPL 识别方法.

输入: arp, TPL = $\{lrp_1, \ldots, lrp_i, \ldots, lrp_n\}$;

输出: candidate TPLs: $list$.

1. sortedList = null;
2. **for** $i \leftarrow 0$ **to** n **do**
3. **if**(isPackageNameMatch(arp, lrp_i) && isSiganatureMatch($Sig_{pst}(arp), Sig_{pst}(lrp_i)$)) // 包名和包结构树签名均匹配
4. $list$ = addLRPtoCandidate(lrp_i, 1); // 1 表示相似性度量值, 取值范围 0–1
5. **else**
6. $sim = sim_{pss}(arp, lrp_i)$;
7. **if** $sim > 0$ **then**
8. $list$ = addLRPtoCandidate(lrp_i, sim); // 包结构树中子包匹配个数
9. **end if**
10. **end if**
11. **end for**
12. sortInDescending($list$);
13. return $list$;

定义 6. 包结构相似度 (package structure similarity). 对于给定的待分析包对 $<arp, lrp>$, 其包结构相似度是指两个根包对应的包结构树中相同子包的个数的两倍与两棵包结构树中所有子包数目之和的比值,

$$sim_{pss}(arp, lrp) = \frac{NumofMatchedSubpackages(arp, lrp) \times 2}{NumofSubpackages(arp) + NumofSubpackages(lrp)} \tag{1}$$

其中, 函数 $NumofSubpackages$ () 统计包结构树中除根节点之外的节点数目, 函数 $NumofMatchedSubpackages$() 统计两棵包结构树中具有相同包名的子包个数.

若两个根包的相似性值大于 0, 则将 lrp 加入候选列表. 没有引入非零阈值作为更严格筛选条件的原因有两个, 一是避免漏检, 故将所有可能的 TPL 都纳入细粒度检测范围, 虽然增加了一定的检测时间开销, 但是实际测试中发现能排除绝大部分 TPL, 设计的相似性度量是具有较好的区分性; 二是无需先验知识来设定阈值. 最后, 算法第 11 行对候选按照相似性降序排序得到最终的候选清单.

2.3 基于多级签名的细粒度检测方法

基于包结构树的快速检测方法仅能确定候选范围, 并不能直接确定引入的 TPL, 仍然需要对 APP 与每个候选进行细粒度的成对比较, 因此提升比较效率和性能对于 LibPass 的实用性是至关重要的. APP 和 TPL 中根包、子包、类、属性和方法的层次组织方式决定了低层次的信息量丰富, 比较精度高同时比较代价也高, 而高层次的信息粒度粗, 比较精度低且代价低. 为了提升检测精度, 同时兼顾效率, 提出了一种基于多级签名的细粒度检测方法, 如图 4 所示. 在签名构建过程中, 以类为对象, 生成类的基本签名、属性签名和方法签名, 而后生成类签名; 在比较过程中, 以根包作为比较对象, 采用递归的思想将根包的比较分解为根包内子包的比较, 子包的比较分解为子包内类签名相似性度量, 类签名相似性比较分解为基本签名、属性签名和方法签名的相似性比较.

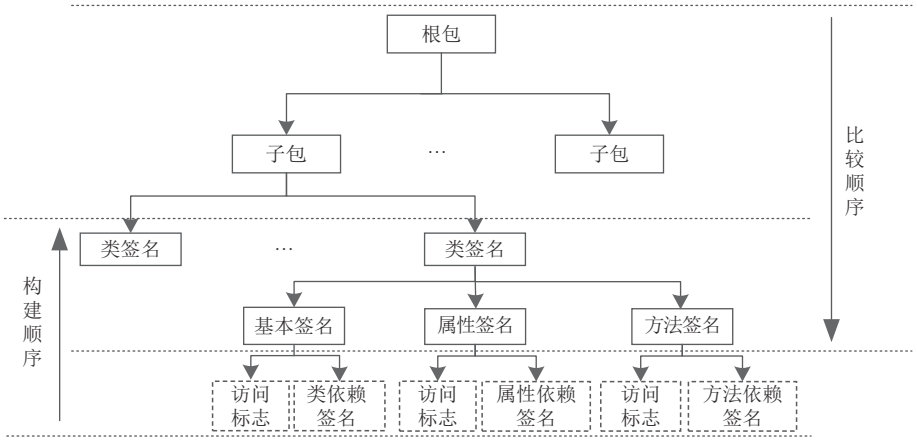


图 4 签名生成和比较模型

在相似性比较中类签名相似性比较是核心, 旨在通过类签名层面的分析提高第三方库及其版本识别的准确性. 现有的基于相似性比较的第三方库检测方法, 如 LibScout 和 LibPecker[24], 采用哈希方式为类中所有方法描述符构成的字符串生成类签名, 具有对抗标志符混淆的能力, 但若在混淆过程中将类中没有使用的方法移除, 则会增加误匹配可能性, 尤其是在类中包含的方法较少的情形下, 导致第三库检测准确率下降. 针对这一问题, 本文提出了一种基于依赖关系的类签名方法, 引入类依赖关系是在混淆情形下仍能得以保持的这一稳定特性提升类签名抗混淆能力. 下面详细描述基于依赖关系的类签名方法, 如算法 3 所示. 给定类 $Class_A$, 其类签名由基本签名、属性签名和方法签名的组合生成, 其中:

- 基本签名, Sig_b, 是类申明中的访问标志和继承依赖拼接成的字符串映射成的哈希值. 这里类继承依赖仅考虑声明中的类继承关系, 若类 $Class_A$ 继承自类 $Class_B$, 则说 $Class_A$ 继承依赖于 $Class_B$.
- 属性签名, Sig_a, 是类中申明的属性的访问标志和属性依赖组成的字符串映射成的哈希值. 若类 $Class_A$ 中定

义的属性其类型为类类型$Class_B$, 则说$Class_A$属性依赖于$Class_B$.

● 方法签名, Sig_m, 是类中方法的访问标志和方法依赖组成的字符串映射成的哈希值. 若类$Class_A$中定义的方法使用了类$Class_B$的作为方法形式参数或者返回值, 则说$Class_A$方法依赖于$Class_B$.

将继承依赖、属性依赖和方法依赖等采用统一的方式进行编码, 生成字符串. 假设类$Class_A$通过上述 3 种依赖方式之一与类$Class_B$形成依赖关系, 编码方法考虑了类$Class_B$的类型和名称, 将两者连接在一起构造字符串. 具体的:

● 类$Class_B$的类型. 考虑 3 种可能的类型: 系统类、同一包中的类、其它包中的类, 使用唯一的整数来表示每种类型, 这里采用了整数 0, 1 和 2 依次表示它们.

● 类$Class_B$的名称. 类的类型决定了编码中采用的类名称. 鉴于在混淆中, 混淆器仅对非系统类进行混淆. 因此, 区分系统类和非系统类, 若类$Class_B$是一个系统类, 则在编码中直接采用类名称, 否则编码中使用一个常量字符作为类名的替代.

算法 3. 基于依赖关系的类签名.

输入: 类 A;

输出: $Sig_b(A)$, $Sig_a(A)$, $Sig_m(A)$, $Sig_c(A)$.

1. Class B = getSuperClass(A); //类 A 的父类
2. Modifer AMC= $getAccessModifier(A)$; //类 A 的访问控制符
3. Type c_type = getClassType(B); // 获取类 B 的类型
4. String str = getEncoding(B, AMC, c_type);
5. $Sig_b(A) = hash(str)$;
6. Class[] F = getAllClassofFields(A); //类 A 的属性对应的类
7. Modifer[] AMF= $getAccessModifierofField(A)$; //类 A 的属性访问控制符
8. Type[] f_type = getClassType(F);
9. **for** $i \leftarrow 0$ **to** $n \leftarrow |F|$ **do**
10. str += getEncoding(F[i], AMF[i], f_type[i]);
11. **end for**
12. $Sig_a(A) = hash(str)$;
13. Class[] M = getAllClassofMethods(A); //类 A 的方法申明中出现的类
14. Modifer[] AMM= $getAccessModifierofMethod(A)$; //类 A 的方法访问控制符
15. Type[] m_type = getClassType(M);
16. **for** $i \leftarrow 0$ **to** $n \leftarrow |F|$ **do**
17. str += getEncoding(M[i], AMM[i], m_type[i]);
18. **end for**
19. $Sig_m(A) = hash(str)$;
20. $Sig_c(A) = Sig_b(A) + Sig_a(A) + Sig_m(A)$;
21. **return** $Sig_b(A), Sig_a(A), Sig_m(A), Sig_c(A)$;

在得到了类的基本签名、属性签名、方法签名和类签名后, 可以度量两个类之间的相似性. 假设类 A 和 B 分别为 APP 和 TPL 中的类, 计算方法如公式 (2) 所示, 其中 $sim_{b\_to\_b}$、 $sim_{a_to_a}$和 $sim_{m\_to\_m}$ 分别表示类基本签名、类属性签名、类方法签名的相似性, 分别如公式 (3)–公式 (5) 所示, w_1、 w_2和w_3分别为各自的权重, $\sum_{i=1}^{3} w_i = 1$. 对于 APP 中引入的 TPL 而言, 如果在混淆阶段没有移除或修改类成员, 则类签名相同, 类相似性为 1, 否则分别度量类的基本签名、属性签名和方法签名的相似性, 并通过加权方式获得类签名相似性. 公式 (3)–公式 (5) 中的函数 $lcs()$表示最长公共子序列.

$$sim_{c\_to\_c}(A,B)=\begin{cases}1, & Sig_c(A)=Sig_c(B)\\ w_1 sim_{b\_to\_b}(A,B)+w_2 sim_{a\_to\_a}(A,B)+w_3 sim_{m\_to\_m}(A,B), & \text{otherwise}\end{cases} \tag{2}$$

$$sim_{b\_to\_b}(A,B)=\begin{cases}1, & Sig_b(A)=Sig_b(B)\\ \left|lcs(Sig_b(A),\ Sig_b(B))\right|/\left|Sig_b(A)\right|, & \text{otherwise}\end{cases} \tag{3}$$

$$sim_{a\_to\_a}(A,B)=\begin{cases}1, & Sig_a(A)=Sig_a(B)\\ \left|lcs(Sig_a(A),Sig_a(B))\right|/\left|Sig_a(A)\right|, & \text{otherwise}\end{cases} \tag{4}$$

$$sim_{m\_to\_m}(A,B)=\begin{cases}1, & Sig_m(A)=Sig_m(B)\\ \left|lcs(Sig_m(A),Sig_m(B))\right|/\left|Sig_m(A)\right|, & \text{otherwise}\end{cases} \tag{5}$$

每个 TPL 都包含很多类, 每个类对 TPL 功能实现的贡献是不尽相同的, 于是使用权重方式来体现类重要性程度的差异. 类权重计算方法如公式 (6), 与 LibPecker 中采用的类权重度量方法是类似的, 其中函数 $NumofMethods()$ 计算类拥有的方法数, 而函数 $NumofDepClasses(C,i)$ 中参数 i 取值为 1 或 2, 分别表示类 C 所在根包中依赖于类 C 的类个数和类 C 依赖的其他类个数. 与 LibPecker 中类权重度量方法相比, 差异之处在于类 C 的依赖类数量度量方式, LibPass 不仅考虑了类 C 依赖的其它类个数, 而且考虑了依赖于类 C 的类个数, 并且将依赖关系范围限定于所在的根包.

$$weight(C)=NumofMethods(C)+\sum_{i=1}^{2}NumofDepClasses(C,i) \tag{6}$$

公式 (6) 表明类的方法数越多, 重要性越高; 另一方面, 类依赖的或被依赖的类越多, 则重要性越高. 进一步的, 假设类 C 所在的子包有 m 个类, 按照公式 (7) 计算得到归一化的类权重.

$$\overline{weight}(C=C_i)=\frac{weight(C_i)}{\sum_{i=1}^{m}weight(C_i)} \tag{7}$$

与类重要性类似, 根包中的子包也存在重要性程度差异, 同样采用权重机制来体现这一差异. 重点考虑子包中类的个数和类权重来设计子包的权重计算方法, 如公式 (8) 所示, 其中函数 $NumofClasses()$ 用于度量子包中类的个数, $C_i \in SP$, $1 \leqslant i \leqslant m$.

$$weight(SP)=NumofClasses(SP)+\sum_{i=1}^{m}weight(C_i) \tag{8}$$

公式 (8) 表明子包拥有的类个数越多, 重要性越高; 子包中类的权重和越大, 则子包重要性越高. 类似地, 对子包的权重进行归一化处理. 假设子包 SP 隶属的根包有 n 个子包, 则归一化后的权重可根据公式 (9) 计算.

$$\overline{weight}\left(SP=SP_j\right)=\frac{weight\left(SP_j\right)}{\sum_{j=1}^{n}weight\left(SP_j\right)} \tag{9}$$

下面阐述基于多级签名的细粒度检测方法识别 APP 中引入的特定版本的 TPL. 正如第 1.2 节分析的那样, TPL 在引入到 APP 中时可能发生较大改变, 例如删除 TPL 中的部分方法、类等使得 APP 中保留的部分与该 TPL 原型差异甚大. 因此, 为了对抗上述混淆行为, 提出的基于多级签名的细粒度检测方法以 APP 中的根包为参照对象以缓解方法、类, 甚至是包缺失带来的影响; 通过成对比较方式以最大相似性原则从本地 TPL 仓库中查找引入的 TPL, 以避免误匹配和 TPL 版本错误.

给定待比较的包对 $<arp,lrp>$, 假设 arp 中 n_1 个子包, lrp 中 n_2 个子包, 以根包 arp 为参照, 根包相似性可以根据公式 (10) 计算, 其中 $sim_{asp\_to\_lrp}(asp_i,lrp)$ 表示根包 lrp 中与子包 asp_i 最为相似的子包相似性值, 可以根据公式 (11) 计算.

$$sim_{arp\_to\_lrp}(arp,lrp)=\frac{\sum_{i=1}^{n_1}sim_{asp\_to\_lrp}(asp_i,lrp)\times\overline{weight}(asp_i)}{n_1} \tag{10}$$

$$sim_{asp\_to\_lrp}(asp_i,lrp)=\max\left\{sim_{asp\_to\_lsp}(asp_i,lsp_j)|_{j=1}^{n_2}\right\} \tag{11}$$

同样以子包 asp_i 为参照, 将包中类相似性度量值以加权方式获得子包相似性. 假设子包 asp_i 中有 m 个类

$\{C_{i1},\ldots,C_{ik},\ldots,C_{im}\}$, 子包相似性可以根据公式 (12) 计算, 其中 $sim_{c\_to\_c}()$ 由公式 (2) 计算, C' 表示 lrp 中与子包 asp_i 中类 C_{im} 相似性最大的类, $I(C')$ 是一个指示函数用于表明类 C' 是否在子包 lsp_j 中, 如公式 (13) 所示.

$$sim_{asp\_to\_lsp}(asp_i,lsp_j)=\frac{\sum_{i=1}^{m}I(C')\times sim_{c\_to\_c}(C_{im},C')\times\overline{weight}(C_{im})}{m} \tag{12}$$

$$I(C')=\begin{cases}1, & C'\in lsp_j\\ 0, & C'\notin lsp_j\end{cases} \tag{13}$$

3 实 验

3.1 实验数据集

在第三方库检测领域, 评价检测方法性能的基准数据集通常以一对多的映射关系存在, 形如 $<apk_i,\{lrp_1,\ldots,lrp_j,\ldots,lrp_n\}>$, 表示应用 apk_i 中使用的第三方库, 且每个 lrp_j 记录了第三方库的名称以及版本, 不仅可用于第三方库检测, 而且还可用于版本识别. 遗憾的是, 目前尚没有一个得到广泛认可的基准数据集, 主要有以下几个原因: 1) 构建基准数据集的方法存在缺陷, 导致评价能力有限, 比如基准中不包含混淆的 TPL; 2) 数据集的规模比较小, 比如 LibD 中使用的基准数据集规模为 1 000 个应用, 包含 2 613 个 TPL, 而 LibID 仅使用 69 个不同 TPL, 包含 1 444 个版本.

为了提升第三方库检测的评价能力, 更好地验证方法有效性, 本文提出了 3 种不同的策略以构建基准数据集, 从而满足不同验证场景的需求.

• 人工构造策略, 即以纯净的 Android 应用为基应用, 通过将一定数量的 TPL 注入到基应用中产生一个新应用, 从而精准地获取该新应用对应的第三方库清单. 具体地, 先选取 10 000 个 Android 应用, 对它们进行反编译生成一组 smali 文件, 通过删除目录中的 smali 文件构造一个基应用, 确保基应用中原有的第三方库被清空. 其次, 为每个应用随机分配一个第三方库注入清单, 包含随机的 8 个到 10 个第三方库. 为了使构建的数据集更贴近真实情况, 在注入前对第三方库进行不同程度的修改. 最后, 根据注入清单将第三方库原型反编译成 smali 文件, 注入到相应的基应用中, 并将 smali 重新打包成一个 APK 文件. 采用上述人工构造策略建立了一个包含 10 000 个应用的基准数据集, 共计 95 434 个 $<apk,lrp>$ 基准对, 将其命名为 GTB-1.

• 开源项目策略, 即从开源的 Android 应用项目发布平台获取项目文件, 进而从项目文件中提取该应用的第三方库清单和主模块. 具体地, 依托开源平台 F-Droid 获取 Android 应用对应的项目文件, 分析第三方库依赖文件获取应用引入的第三方库清单和主模块信息. 基于上述策略分析了 1 000 个开源项目, 共计得到 14 233 个 $<apk,lrp>$ 基准对, 将其命名为 GTB-2.

• 变异策略, 即采用多种主流混淆器提供的混淆功能对注入的第三方库进行变异, 获取混淆后的应用及其第三方库清单. 具体地, 从开源平台 F-Droid 获取 Android 应用项目文件, 为每个应用分别配置 3 种不同的混淆器, 分别为 ProGuard、Allatori 和 DashO; 同时考虑 4 种不同的混淆能力或处理方法, 分别为符号混淆 (包、类、方法等标识符名称转化为无意义字符串, 记为 Type-1)、优化 (使用控制流分析、数据流分析等技术优化程序, 记为 Type-2)、压缩 (把未使用的类、方法等移除, 记为 Type-3)、扁平化 (把混淆后的类移动到指定的包中, 进行类重组, 记为 Type-4). 设计不同混淆器和混淆功能的组合构成混淆配置应用于每个 Android 应用项目产生对应的混淆版本. 基于上述策略分析了 1 000 个开源项目, 仅部分项目顺利完成混淆操作, 得到的结果如表 3 所示, 将其命名为 GTB-3.

表 3 应用不同混淆配置后得到的数据集

Obfuscator	Number of APPs	Number of RPs	Number of APP-TPL pairs	Obfuscation types			
				1	2	3	4
ProGuard	200	1 889	914	√	√	√	√
Allatori	241	1 493	1 309	√	√	√	×
DashO	215	1 041	1 001	√	√	√	×

通过上述 3 种不同策略构建的基准数据集对于第三方库检测能力评价提供了不同的支撑, 差异之处可以概括如表 4 所示.

表 4　不同基准数据集在第三方检测能力评价上的差异

基准数据集	规模	检测能力		抗混淆能力	检测效率	主模块分析
		precision	*recall*			
GTB-1	10 000 APKs	√	√	×	×	×
GTB-2	1 000 APKs	×	√	×	√	√
GTB-3	[200, 282] APKs	√	√	√	×	×

3.2　TPL 本地库

TPL 本地库的完整性也是影响基于相似性比较的 TPL 检测方法准确性和效率的一个重要因素. 然而, Android 社区并没有统一的 TPL 发布平台和规范. 开发者使用的 TPL 主要来源于 GitHub 或者 Maven 等代码托管仓库, 因此存在版本不完整、新 TPL 或新版本上线后不能及时更新等问题, 进而导致 TPL 检测方法精度下降, 检测过程中因发起实时 TPL 搜索、获取和分析而导致检测时间成本显著增加. 为了有效应对上述问题, 本文采用面向代码仓库搜索和 APK 反向分离相结合的方法构建较大规模的 TPL 库, 并借助于 LibPass 中的主模块识别、组件签名等完成对 TPL 的预处理, 以提升检测精度和效率.

首先, 面向 Maven 自动搜索 TPL. Maven 代码托管仓库中不仅存有面向 Android 平台的大量第三方库, 而且还有大量面向其他平台或开发语言的 TPL. 因此, 在自动获取面向 Android 平台的 TPL 之前, 首先需要预先给出一个待收集的 TPL 清单, 以过滤掉其他不相关的 TPL. 为了定制 TPL 清单, 分析了 Android 应用开源社区“F-Droid”中所有的项目, 提取每一个项目的 gradle 文件中所包含的 TPL 清单加以合并去重, 并且采用三元组 GAV=<Groupname, Artifactld, Versionpairs>唯一地标识每一个 TPL. 一旦确定了待收集的 TPL 清单, 即可利用爬虫程序根据清单中的每一个 GAV 爬取相关的第三方库 jar/aar 文件. 值得注意的是, APK 以汇编索引代码 (二进制形式代码) 形式存在, 因此需要借助于转换工具, 如 dextojar 工具等, 将下载的 jar/aar 文件中所有的类打包编译为 dex 文件后存入 TPL 本地库.

其次, 通过 APK 反向分离发现 TPL. 尽管原型库构建可以通过 Maven/Gradle 从网上爬取 TPL 原型, 但是 Maven 上并不能收集到所有的 TPL, 包括一些过期的 TPL 版本. 而每个 Android 应用中都有多个 TPL, 因此, 可以通过分析大量 Android 应用从中抽取其 TPL 原型使原型库更完备. smali 提供反汇编功能的同时, 也提供了打包反汇编代码重新生成 dex 的功能. 基于这一特点, 在抽取 TPL 的过程中, 先将大量 APK 文件反编译成 smali 分析, 随后对其进行模块解析操作, 将 APK 拆分成多个模块, 再把每个非混淆模块重打包生成 DEX 文件存入 TPL 原型库中, 可得到大量的 TPL 原型, Maven 存储库中缺少的 TPL 原型都可以从 APK 中找到. 本文分析了 10 000 个从 Google play 中下载下来的 Android 应用, 把其中所有未被混淆过的、且本地原型库中缺少的 TPL 其 smali 文件提取出来打包成 DEX 文件, 存入本地原型库中.

综上两种 TPL 抽取方式创建了包含 140 000 个 TPL 的本地库, 其中包括从 maven 中爬取的 12 648 个 TPL, 以及从 10 000 个 APK 中分离出的 127 352 个 TPL.

3.3　实验设置

为了验证提出的第三方库检测方法的性能和效率, 并且与其他领域知名的检测方法进行对比分析, 开展了一组实验并进行深入分析. 实验采用的服务器配置为 Windows 10 专业版 64 位操作系统, Intel(R) Xeon(R) CPU E5-2678 v3 处理器, 主频 @ 2.50 GHz , 内存 64 GB. 在实验中采用了前文构建的 3 种不同能力的基准数据集作为实验对象, 选取了本领域先进的、公开源码的方法, 如 LibScout[14]、LibRadar[21]、LibD[22]、LibPecker[15]、LibID-S[16] 和 LibID-A[16]作为对比方法. LibPass 主模块识别模块中使用的依赖强度阈值 δ_d 设置为 0.01, 确保任意两个子包存在依赖关系即可. 实验中使用召回率 (*recall*), 精确率 (*precision*), 假阳率 (false positive ratio, *FPR*) 以及假阴率

(false negative ratio, *FNR*) 等 4 个指标来评估第三方库检测方法性能, 度量方法分别如公式 (14)–公式 (17) 所示:

$$recall = \frac{number\_of\_true\_positives}{number\_of\_pairs\_in\_the\_ground\_truth} \tag{14}$$

$$precision = \frac{number\_of\_true\_positives}{number\_of\_detected\_pairs} \tag{15}$$

$$FPR = \frac{number\_of\_detected\_pairs - number\_of\_true\_positives}{number\_of\_detected\_pairs} \tag{16}$$

$$FNR = \frac{number\_of\_pairs\_in\_ground\_truth - number\_of\_true\_positives}{number\_of\_pairs\_in\_ground\_truth} \tag{17}$$

其中, *true_positives* 指正确识别的 $< apk, lrp >$ 对的个数, *detected_pairs* 指检测方法识别的所有的 $< apk, lrp >$ 对的个数; *pairs_in_the_ground_truth* 指基准中 $< apk, lrp >$ 对的个数 . 此外, 实验中采用运行时间来度量检测方法及其关键组件的效率.

3.4 结果与分析

3.4.1 主模块识别性能和效率分析

本节对 LibPass 的主模块识别组件进行评价, 先评估主模块识别的性能和效率, 而后评估主模块识别组件对于 LibPass 解决方案的贡献.

首先, 评估主模块识别的性能和效率. 鉴于在 3 个基准数据集中仅 GTB-2 支持对主模块识别性能进行评估, 实验中以 GTB-2 作为实验对象, 通过人工分析项目的依赖文件的方式得到了每个项目包含的模块以及主模块, 共计有 16 387 个模块, 其中 1 000 个为主模块. 随机选取 30% 的项目为其配置 ProGuard 作为混淆器, 采用默认配置确保对标识符进行混淆, 得到上述项目对应的 APK 文件. 将本文提出的主模块识别方法应用于这些 APK, 实验重复 10 次, 实验结果取平均值, 如表 5 所示. 从表中可以看出: 1) 模块解析的准确率 98.91%, 存在约 1% 的误检, 对于误检的模块进行人工分析发现, 误检主要原因是代码复用, 即开发者会将其他 TPL 整合到自己开发的 TPL 中. 2) 模块解析方法是抗标识符混淆的, 在 30% APK 混淆情形下, 模块解析和主模块识别的准确率高达 99.2%. 3) 仅考虑元信息的情形下, 主模块识别方法准确率为 70%, 但表明该方法准确识别了所有非混淆 APK 的主模块, 仅通过分析 Android manifest 文件即可实现兼具高效性, 平均运行时间为 17.7 ms. 4) 仅考虑依赖关系的情形下, 主模块识别识别方法准确率为 98.7%, 显著高于基于元信息的方法, 但该方法所需的时间开销是基于元信息的方法的约 49 倍. 5) 融合了元信息和依赖关系的主模块识别方法的准确率较纯粹基于依赖关系的方法有提升, 总的时间开销降低了 29.2%. 上述实验结果表明, 本文提出的融合元信息和依赖关系的识别方法具有较高的主模块识别准确率, 并能降低时间开销.

表 5 模块解析和主模块识别实验结果

类型	方法	模块数目	正确的模块数	准确率 (%)	平均时间 (ms)
模块解析	PDG	16387	16209	98.91	865.6
主模块识别	元信息识别	1 000	700	70.00	17.7
	依赖关系识别	1 000	987	98.70	866.8
	两种方法结合	1 000	992	99.20	613.6

其次, 评估主模块识别组件对于 LibPass 解决方案的贡献. 实验中考虑了是否在 LibPass 中集成主模块识别组件, 实验结果如表 6 所示. 可以看出 *recall* 和 *FPR* 在是否启用主模块识别功能两种情形下没有变化, 而平均检测时间有较大变化. 具体地, 应用主模块识别组件后平均检测时间从 23.68 s 减少到 15.47 s, 节省 28.4% 的检测时间, 这表明主模块识别组件的引入对效率提升有积极影响, 但对性能没有影响. LibPass 检测过程中将主模块排除在外是检测效率提升的一个可能原因. 为了验证这一点, 对数据集中每个 APK 主模块的包分布情况进行了统计, 大于 10 个包的占比 24.2%, 大于 5 个包的占比 37.8%, 多于非主模块包数.

表 6　LibPass 方法中是否应用主模块识别的实验结果

是否集成主模块识别	性能 (%)		平均检测时间 (s)	
	recall	*FPR*	主模块识别时间	两阶段检测时间
√	100	11.39	1.48	15.47
×	100	11.38	—	23.68

最后, 讨论主模块识别中涉及的阈值对于检测结果的影响. LibPass 仅涉及一个阈值“依赖强度阈值δ_d”, 该阈值出现在主模块识别方法中, 用于判定两个子包是否存在依赖关系. 实验中将 LibPass 主模块识别模块中使用的依赖强度阈值δ_d设置为 0.01, 一个比较小的值, 意味着确保任意两个子包容易存在依赖关系. 从表 5 和表 6 中关于主模块对 LibPass 的贡献分析结果可知, 主模块识别对 LibPass 的检测性能无影响, 对于检测效率有一定影响, 主模块识别仅是一个用于优化检测效率的可选组件. 因此, 依赖强度阈值对于 TPL 检测性能是没有影响的, 故这里不再进行阈值变化对检测结果的影响分析.

3.4.2　基于包结构树的检测性能和效率分析

本节对 LibPass 中基于包结构树的检测组件进行评价, 特别是评估其对于 LibPass 解决方案效率方面的贡献. 实验在基准数据集 GTB-2 进行, 结果如表 7 所示. 从表中可以看出, 是否应用基于包结构树的检测方法对性能指标 *recall* 和 *FPR* 也没有影响, 但是应用后的平均检测时间从 525.6 s 下降到 15.47 s, 节省 97.05% 的检测时间. 这同样表明基于包结构树的检测组件的引入对效率提升有显著影响, 但对性能没有影响. 效率提升的原因在于基于包结构树的检测方法提供了包级别匹配快速识别候选第三方库, 而不需要更耗时的类级别检测.

表 7　LibPass 方法中是否集成基于包结构树的检测方法的实验结果

是否集成基于包结构树的检测	性能 (%)		平均检测时间 (s)		
	recall	*FPR*	基于包结构树的	基于多级签名的	两者之和
×	100	11.38	—	525.60	525.60
√	100	11.39	2.74	12.73	15.47

进一步地, 效率提升的空间取决于待分析应用程序是否经过混淆. 为了估计效率提升的空间, 从豌豆荚、Google Store、F-Droid 上收集约 16 900 个 Android 应用, 对应用的混淆情况进行了统计分析, 结果如表 8 所示, 从表中可以看出, 83.46% 的 APK 没有混淆, 这些 APK 包含的 ARP 个数占比 92.63%, 剩余的 7.37% ARP 中仅有 1.77% 应用了包重组混淆技术会导致基于包结构树的检测方法失效, 因此仅需对 1.77% 的 ARP 执行基于多级签名的相似性比较, 而其他的 98.23% 的 ARP 可以通过基于包结构树的检测方法能对抗标识符混淆, 快速过滤到绝大多数 TPL. 实验中统计基于包结构树的检测方法获得候选 TPL 个数平均值约为 70, 而 TPL 本地库规模为 140 000 个, 因此对于 98.23% 的 ARP 检测效率提升了 2 000 倍. 从上述分析结果可以得出基于包结构树的快速检测方法对于 LibPass 的检测效率提升有显著贡献, 即使 TPL 规模的增大也不会对 LibPass 的检测效率带来不良影响.

表 8　收集的 APP 混淆情况

是否混淆	混淆类型	APK数 (占比(%))	ARP数 (占比(%))
×	—	14 104 (83.46)	122 014 (92.63)
√	标志符混淆	1 644 (9.72)	7 389 (5.60)
	标志符混淆 & 包重组	1 152 (6.82)	2 325 (1.77)

3.4.3　检测能力评价

本节在 GTB-1 基准数据集上完成对 LibPass 的检测能力评价, 并与其他第三方库检测方法进行了对比分析, 实验结果如表 9 所示. 从表中可以看出, LibPass 的 precision 为 93.07%, 比最好的 LibD 低 0.61%; LibPass 的 *recall* 略低于 LibPecker 和 LibID-A, 分别低了 3.45% 和 0.09%; LibPass 的综合指标 *F*1-score 上显著优于其他对比方法, 比排名第 2 的 LibID-A 的对应值高 13.71%; 这表明 LibPass 的总体性能优于其他对比方法.

表 9 基于 GTB-1 数据集实验结果 (%)

方法	*precision*	*recall*	*FPR*	*FNR*	*F*1-score
LibRadar	90.74	65.98	9.26	34.02	76.41
LibD	**93.68**	62.91	6.32	37.09	75.27
LibScout	67.44	94.07	32.56	5.93	78.56
LibPecker	44.67	**99.63**	55.33	**0.37**	61.68
LibID-S	67.39	88.30	32.61	11.70	76.44
LibID-A	69.74	96.27	30.26	3.73	80.89
LibPass	93.07	96.18	**6.93**	3.82	**94.60**

与 LibPass 解决思路最为接近的是 LibPecker, LibPass 在 precision 上比 LibPecker 高 48.4%, 主要原因在于较多 TPL 中存在着相同或相似的包, LibPecker 以包为单位进行比较, 容易造成误匹配, 而 LibPass 是以根包为单位进行成对比较, 且每个根包只能匹配一个第三方库, 大大减少了误匹配, 故其准确率较高. 从表 9 中还可以看出, LibPass 的假阳性率为 6.93%, 对检测结果为假阳性的 $<arp,lrp>$ 对进行分析发现 24 个重复出现的 TPL 是造成假阳性的原因. 进一步分析这些 TPL 发现在本地库中存在同一个 TPL 的多个版本, 对版本之间的相似性进行了评估, 获得的相似性度量结果如图 5 所示, 从图 5 可以看出, 同一个 TPL 不同版本间有高相似性, 即 79.2% 的 TPL 其版本之间的相似度超过 0.7, 这验证了多个高相似版本的存在是导致 LibPass 产生假阳性的主要原因. 因此, 尽管 LibPass 可以确定 TPL 的具体版本, 但是仍需在降低版本相似性带来的假阳性方面开展进一步研究.

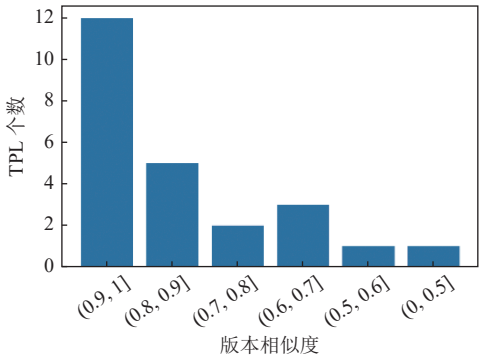


图 5 不同版本的第三方库相似度

3.4.4 抗混淆能力评价

本节在 GTB-3 基准数据集上完成对 LibPass 的抗混淆能力评价, 并与其他第三方库检测方法进行了对比分析, 实验结果如图 6 所示, 图 6(a)–(d) 分别给出了无混淆、应用混淆器 ProGuard、混淆器 Allatori 和 DashO 等 4 种情形下的实验结果. 从图 6(a) 可以看出, 在无混淆的情形下, LibPass 有最高的 *F*1-score, 达 94.92%, 比其他检测方法中最好的 (LibID-A) 高 17.3%, 且有较低的误检率 5.90% 和漏检率 4.24%. 在采用混淆器 ProGuard 进行混淆的情形下, 尽管 LibPass 取得了最高的 *F*1-score, 但仅有 54.22%, 与无混淆的情形比较下降了约 40%. 在采用混淆器 Allatori 进行混淆的情形下, LibPass 同样取得了最高的 *F*1-score, 87.09%, 与无混淆的情形比较下降了 7.83%. 在采用混淆器 DashO 进行混淆的情形下, LibPass 获得的 *F*1-score 为 76.36%, 比 LibID-A 的 *F*1-score 低 3.73%, 与无混淆的情形比较下降了 18.56%. 上述结果表明 LibPass 在抗混淆能力方面优于与现有的检测方法.

当然, 我们也注意到混淆仍是 TPL 检测方法面临的主要挑战之一. 在应用 3 种不同混淆器的情形下, 采用混淆器 ProGuard 进行混淆时 LibPass 及其对比方法获得的结果都是最差的. 一个不容忽视的问题是大多数 Android 开发者选择免费的 ProGuard 作为应用发布前的混淆器[35], 因此本文深入分析了造成检测准确率低的原因. 从表 10 可以推测, 一个可能的原因在于使用了混淆器 ProGuard 的扁平化混淆功能, 而另外两个混淆器不支持该功能. 为了验证这一点, 对基准数据集 GTB-3 生成过程进行了控制, 每次仅启用混淆器的一种混淆能力, 即优化处理、标识符混淆、压缩处理以及混淆后扁平化处理, 产生了 4 个新的实验数据集, 外加一个不混淆的原始数据集.

LibPass 在上述数据集上的实验结果如表 10 所示. 从表中可以看出, 相对于未混淆的情形, 启用优化、标识符混淆和压缩等 3 个混淆功能对于 LibPass 的影响不大, *precision* 最大仅下降了 2.68%, *recall* 最大仅下降了 2.79%, 且 3 种混淆情形下性能指标变化不大; 启用扁平化功能对于 LibPass 有较大影响, *precision* 下降了 17.74%, *recall* 下降了 43.05%, 这一结果验证了扁平化是造成 LibPass 检测方法抗混淆能力下降的主要原因.

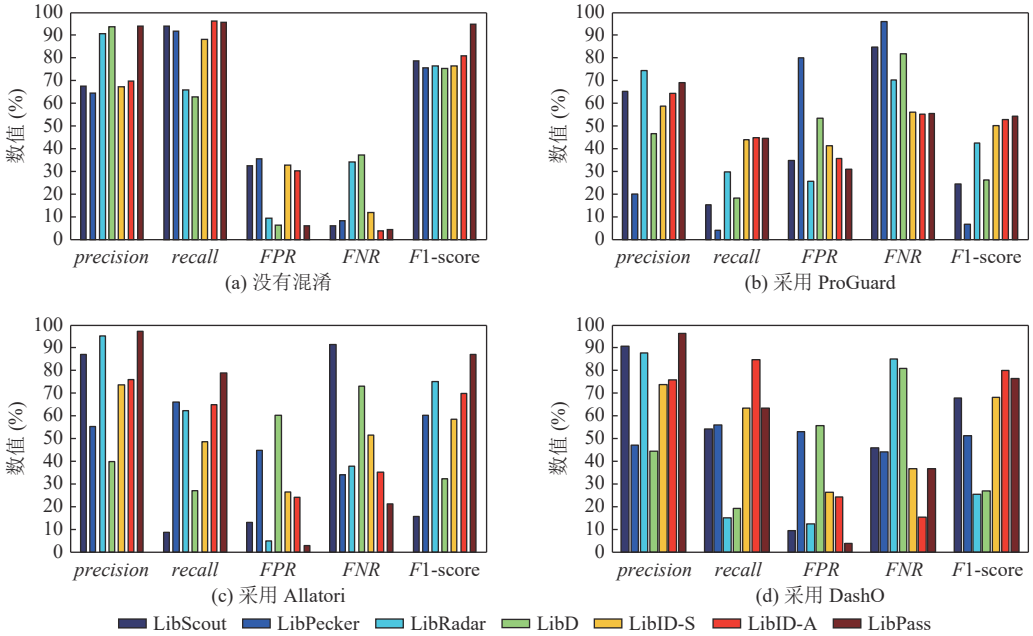


图 6 不同方法的抗混淆能力评价结果

表 10 在开启 ProGuard 不同混淆功能情形下的抗混淆能力 (%)

数据集	混淆功能	默认配置	*precision*	*recall*	*FPR*	*FNR*	*F*1-score
GTB-3-0	无混淆	—	94.89	95.43	5.11	4.57	95.16
GTB-3-1	优化	√	93.17	94.71	6.83	5.29	93.93
GTB-3-2	标识符混淆	√	92.44	93.94	7.56	6.06	93.18
GTB-3-3	压缩	√	92.31	92.64	7.69	7.36	92.47
GTB-3-4	扁平化	×	77.15	52.38	22.85	47.62	62.40

进一步的, 表 11 描述了 LibPass 和其他 6 种 TPL 检测方法在抗扁平化混淆能力方面的对比结果, 其中括号内数值为无混淆情形下的检测结果. 从表 11 的结果可以看出, 基于相似性比较的 TPL 检测方法或工具在扁平化混淆情形下 TPL 检测精度与无混淆情形下相比均有不同程度下降, 其中 LibPecker、LibRadar 和 LibD 下降较为显著, *F*1-score 远低于 50%, 而 LibPass 和 LibID 下降幅度相对较小, *F*1-score 略高于 50%. 虽然 LibPass 在扁平化混淆情形下得到了最高的检测精度, 但 *F*1-score 也仅有 62.4%, 尚不能有效应对扁平化混淆.

表 11 不同 TPL 检测方法的抗扁平化混淆能力评价结果 (%)

检测方法	*precision*	*recall*	*F*1-score
LibRadar	36.23 (69.98)	29.65 (65.98)	32.61 (67.92)
LibD	19.25 (49.23)	26.97 (62.91)	22.47 (55.24)
LibScout	65.22 (69.35)	15.00 (94.09)	24.39 (79.84)
LibPecker	20.00 (65.26)	4.00 (91.68)	6.67 (76.25)
LibID-S	66.67 (93.91)	44.00 (88.35)	53.01 (91.05)
LibID-A	72.58 (88.16)	45.00 (96.30)	55.56 (92.05)
LibPass	77.15 (94.89)	52.38 (95.43)	62.40 (95.16)

不过幸运的是约 70% APK 采用了 ProGuard 的默认配置, 仅启用了优化、标识符混淆和压缩, 原因在于扁平化可能导致发布后的 APP 存在不能在所有版本的 Dalvik 上运行的风险, 因此默认不启用该混淆功能. 从上述分析可以得出结论, LibPass 具备良好的在优化、标识符混淆和压缩等混淆情形下抗混淆能力, 但在对抗扁平化情形的混淆方面还有待提升.

3.5 讨 论

本节讨论研究工作中的局限性以及未来可能的改进工作.

3.5.1 多维度相似性

LibPass 仅依赖于 APP 和 TPL 的包结构信息和类申明信息度量 APP 的根包与 TPL 的根包之间的结构相似性实现 TPL 检测, 而没有考虑具体的代码实现及其语义信息、UI 资源等其他维度的相似性. 代码克隆检测或程序相似性比较中常考虑语法相似性, 可以利用 APK 的 opcode 序列抽取语法层面的特征, 进行设计相似性度量. 又如, 对于那些使用了 Java 注解机制开发的 TPL, 注解信息会被嵌入到字节码中, 这些注解信息可以用于度量注解相似性. 再如, 有些 TPL 提供了 UI 资源, 因此 UI 相似性也可以作为一个比较维度来识别引入的 TPL. 鉴于每个 TPL 都有其各自的隐含特征, 考虑多维度相似性, 而不仅仅是单一维度相似性, 或许对于提升 TPL 检测精度有明显帮助, 需要进一步的工作加以验证.

3.5.2 相似性阈值

LibPass 虽然也采用了基于相似性比较的方法, 但却没有引入相似性度量相关的阈值, 这是有别于其他基于相似性度量方法的. 例如, LibScout 引入了 Merkle 树相似度阈值. LibPass 中没有引入非零阈值作为更严格筛选条件有两个优势之处: 一是避免漏检, 将所有可能的 TPL 都纳入细粒度检测范围, 虽然增加了一定的检测时间开销, 但是实际测试中发现能排除绝大部分 TPL; 二是无需先验知识来设定相似性阈值.

3.5.3 其他应用场景

提出的 TPL 检测方法不仅可以识别引入的 TPL, 而且可以确定 TPL 的具体版本, 因此作为程序分析的一个前置任务, TPL 检测方法可以被整合到更加复杂的任务中. 例如, 文献 [23] 提供了一种代码补全场景, 在该场景中给定程序片段以及该程序片段所属项目先前已经发布的版本, 建立该程序片段的上下文, 进一步基于上下文借助于诸如 WALA 等工具完成代码补全. 在该任务场景中, 提出的 TPL 检测方法可以被整合到程序片段上下文识别阶段, 用于提升识别准确性. 更具体地说, 将程序片段视为 APP, 而该程序片段所属项目先前已经发布的版本中出现的 JARs 作为 TPLs, 应用 TPL 检测方法可以识别程序片段是否与先前使用的 JARs 相关. 如果待分析程序片段来自于某个 JAR, 并且不同项目版本中存在该 JAR 的多个版本, 则可以进一步确定具体的 JAR 版本, 从而获取更加准确的上下文. 上述 TPL 检测方法的基本应用思路在具体场景下的实现过程和细节尚待进一步斟酌.

4 结 论

本文采用相似性比较的思想, 从结构相似性角度提出了一种基于包结构树和多级签名的 TPL 检测方法, 由主模块识别、候选 TPL 识别和细粒度检测等 3 个关键组件以流水线方式组成. 主模块识别和候选 TPL 识别组件旨在提升检测效率, 分别通过减少移除主模块的根包和识别可能候选 TPL 的方式降低了约 29% 和 97% 的时间开销, 能够在秒级完成 TPL 检测. 候选 TPL 识别组件对检测性能也有影响, 该组件利用包结构树稳定的特性产生签名识别可能的 TPL 候选, 具备应对标志符混淆、包移除、控制流混淆等常用混淆的对抗能力. 细粒度检测组件旨在提升检测性能, 以类级、方法级和属性级签名为比较对象, 充分利用类内的不变特性产生签名, 并以 APP 中的根包为参照, 通过加权方式度量相似性, 识别引入的特定版本的 TPL, 具备对抗类、方法、属性移除等混淆操作的能力, 在基准数据集上获得了 94.6% 的 $F1$-score. 此外, 本文还提出了 3 种不同策略构建基准数据集, 并将其公开供研究人员使用, 对 TPL 检测方法评价有积极贡献.

References:

[1] Hu WH, Octeau D, McDaniel PD, Liu P. Duet: Library integrity verification for Android applications. In: Proc. of the 2014 ACM Conf. on Security and Privacy in Wireless & Mobile Networks. Oxford: Association for Computing Machinery, 2014. 141–152. [doi: 10.1145/

2627393.2627404]

[2] Wang HY, Hong J, Guo Y. Using text mining to infer the purpose of permission use in mobile APPs. In: Proc. of the 2015 ACM Int'l Joint Conf. on Pervasive and Ubiquitous Computing. Osaka: Association for Computing Machinery, 2015. 1107–1118. [doi: 10.1145/2750858.2805833]

[3] Li L, Bissyandé TF, Klein J. SimiDroid: Identifying and explaining similarities in Android APPs. In: Proc. of the 2017 IEEE Trustcom/BigDataSE/ICESS. Sydney: IEEE, 2017. 136–143. [doi: 10.1109/Trustcom/BigDataSE/ICESS.2017.230]

[4] Zheng X, Pan L, Yilmaz E. Security analysis of modern mission critical Android mobile applications. In: Proc. of the 2017 Australasian Computer Science Week Multiconference. Geelong: Association for Computing Machinery, 2017. 2. [doi: 10.1145/3014812.3014814]

[5] Zhou W, Zhou YJ, Jiang XX, Ning P. Detecting repackaged smartphone applications in third-party Android marketplaces. In: Proc. of the 2nd ACM Conf. on Data and Application Security and Privacy. San Antonio: Association for Computing Machinery, 2012. 317–326. [doi: 10.1145/2133601.2133640]

[6] Hanna S, Huang L, Wu E, Li S, Chen C, Song D. Juxtapp: A scalable system for detecting code reuse among Android applications. In: Proc. of the 9th Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment. Heraklion: Springer, 2013. 62–81. [doi: 10.1007/978-3-642-37300-8_4]

[7] Grace MC, Zhou W, Jiang XX, Sadeghi AR. Unsafe exposure analysis of mobile in-APP advertisements. In: Proc. of the 5th ACM Conf. on Security and Privacy in Wireless and Mobile Networks. Tucson: Association for Computing Machinery, 2012. 101–112. [doi: 10.1145/2185448.2185464]

[8] Book T, Pridgen A, Wallach DS. Longitudinal analysis of Android ad library permissions. arXiv:1303.0857, 2013.

[9] Ma ZA, Wang HY, Guo Y, Chen XQ. LibRadar: Fast and accurate detection of third-party libraries in Android APPs. In: Proc. of the 38th Int'l Conf. on Software Engineering Companion. Austin: Association for Computing Machinery, 2016. 653–656. [doi: 10.1145/2889160.2889178]

[10] Li MH, Wang W, Wang P, Wang S, Wu DH, Liu J, Xue R, Huo W. LibD: Scalable and precise third-party library detection in Android markets. In: Proc. of the 39th Int'l Conf. on Software Engineering. Buenos Aires: IEEE, 2017. 335–346. [doi: 10.1109/icse.2017.38]

[11] Wang HY, Guo Y, Ma ZA, Chen XQ. WuKong: A scalable and accurate two-phase approach to Android APP clone detection. In: Proc. of the 2015 Int'l Symp. on Software Testing and Analysis. Baltimore: Association for Computing Machinery, 2015. 71–82. [doi: 10.1145/2771783.2771795]

[12] Narayanan A, Chen LH, Chan CK. AdDetect: Automated detection of Android ad libraries using semantic analysis. In: Proc. of the 9th IEEE Int'l Conf. on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP). Singapore: IEEE, 2014. 1–6. [doi: 10.1109/ISSNIP.2014.6827639]

[13] Liu B, Liu B, Jin HX, Govindan R. Efficient privilege de-escalation for Ad libraries in mobile apps. In: Proc. of the 13th Annual Int'l Conf. on Mobile Systems, Applications, and Services. Florence: Association for Computing Machinery, 2015. 89–103. [doi: 10.1145/2742647.2742668]

[14] Backes M, Bugiel S, Derr E. Reliable third-party library detection in Android and its security applications. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. Vienna: Association for Computing Machinery, 2016. 356–367. [doi: 10.1145/2976749.2978333]

[15] Zhang, Y, Dai JR, Zhang XH, Huang SR, Yang ZM, Yang M, Chen H. Detecting third-party libraries in Android applications with high precision and recall. In: Proc. of the 25th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Campobasso: IEEE, 2018. 141–152. [doi: 10.1109/SANER.2018.8330204]

[16] Zhang JX, Beresford AR, Kollmann SA. LibID: Reliable identification of obfuscated third-party Android libraries. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: Association for Computing Machinery, 2019. 55–65. [doi: 10.1145/3293882.3330563]

[17] ProGuard. 2021. https://www.guardsquare.com/en/products/proguard

[18] Allatori Obfuscator. 2021. http://www.allatori.com/

[19] DashO. 2021. https://www.preemptive.com/products/dasho/overview

[20] Hammad M, Garcia J, Malek S. A large-scale empirical study on the effects of code obfuscations on Android APPs and anti-malware products. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: Association for Computing Machinery, 2018. 421–431. [doi: 10.1145/3180155.3180228]

[21] Chen K, Liu P, Zhang YJ. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In: Proc. of the 36th Int'l Conf. on Software Engineering. Hyderabad: Association for Computing Machinery, 2014. 175–186. [doi: 10.1145/2568225.2568286]

[22] Li L, Bissyandé TF, Klein J, Traon YL. An investigation into the use of common libraries in Android APPs. In: Proc. of the 23rd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER). Osaka: IEEE, 2016. 403–414. [doi: 10.1109/SANER.2016.52]

[23] Zhong H, Wang XY. Boosting complete-code tool for partial program. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Urbana: IEEE, 2017. 671–681. [doi: 10.1109/ASE.2017.8115677]

[24] Lin JL, Amini S, Hong JI, Sadeh N, Lindqvist J, Zhang J. Expectation and purpose: Understanding users' mental models of mobile APP privacy through crowdsourcing. In: Proc. of the 2012 ACM Conf. on Ubiquitous Computing. Pittsburgh: Association for Computing Machinery, 2012. 501–510. [doi: 10.1145/2370216.2370290]

[25] Crussell J, Gibler C, Chen H. AnDarwin: Scalable detection of Android application clones based on semantics. IEEE Trans. on Mobile Computing, 2015, 14(10): 2007–2019. [doi: 10.1109/TMC.2014.2381212]

[26] Wang HY, Guo Y, Ma ZA, Chen XQ. Automated detection and classification of third-party libraries in large scale Android APPs. Ruan Jian Xue Bao/Journal of Software, 2017, 28(6): 1373–1388 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5221.htm [doi: 10.13328/j.cnki.jos.005221]

[27] Li MH, Wang P, Wang W, Wang S, Wu DH, Liu J, Xue R, Huo W, Zou W. Large-scale third-party library detection in Android markets. IEEE Trans. on Software Engineering, 2020, 46(9): 981–1003. [doi: 10.1109/TSE.2018.2872958]

[28] Wang Y, Wu HW, Zhang HL, Rountev A. ORLIS: Obfuscation-resilient library detection for Android. In: Proc. of the 5th Int'l Conf. on Mobile Software Engineering and Systems. Gothenburg: Association for Computing Machinery, 2018. 13–23. [doi: 10.1145/3197231.3197248]

[29] Tang W, Luo P, Fu JL, Zhang D. LibDX: A cross-platform and accurate system to detect third-party libraries in binary code. In: Proc. of the 27th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). London: IEEE, 2020. 104–115. [doi: 10.1109/SANER48275.2020.9054845]

[30] Soh C, Tan HBK, Arnatovich YL, Narayanan A, Wang LP. LibSift: Automated detection of third-party libraries in Android applications. In: Proc. of the 23rd Asia-Pacific Software Engineering Conf. Hamilton: IEEE, 2016. 41–48. [doi: 10.1109/APSEC.2016.017]

[31] Glanz L, Amann S, Eichberg M, Reif M, Hermann B, Lerch J, Mezini M. CodeMatch: Obfuscation won't conceal your repackaged app. In: Proc. of the 11th Joint Meeting on Foundations of Software Engineering. Paderborn: Association for Computing Machinery, 2017. 638–648. [doi: 10.1145/3106237.3106305]

[32] Tang ZS, Xue MH, Meng GZ, Ying CG, Liu YG, He JA, Zhu HJ, Liu Y. Securing Android applications via edge assistant third-party library detection. Computers & Security, 2019, 80: 257–272. [doi: 10.1016/j.cose.2018.07.024]

[33] Huang SR, Tao FF, Zhang Y, Yang M. LibSeeker: Detecting Android third-party libraries using parameter auto-tuning. Journal of Chinese Computer Systems, 2019, 40(2): 332–340 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-1220.2019.02.017]

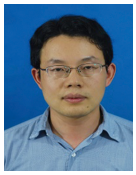
[34] Wang Y, Rountev A. Who Changed You? Obfuscator identification for Android. In: Proc. of the 4th IEEE/ACM Int'l Conf. on Mobile Software Engineering and Systems (MOBILESoft). Buenos Aires: IEEE, 2017. 154–164. [doi: 10.1109/MOBILESoft.2017.18]

[35] Wang P, Bao QK, Wang L, Wang S, Chen ZF, Wei T, Wu DH. Software protection on the go: A large-scale empirical study on mobile APP obfuscation. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: Association for Computing Machinery, 2018. 26–36. [doi: 10.1145/3180155.3180169]

附中文参考文献:

[26] 王浩宇, 郭耀, 马子昂, 陈向群. 大规模移动应用第三方库自动检测和分类方法. 软件学报, 2017, 28(6): 1373–1388. http://www.jos.org.cn/1000-9825/5221.htm [doi: 10.13328/j.cnki.jos.005221]

[33] 黄思荣, 陶非凡, 张源, 杨珉. LibSeeker: 参数自整定的安卓应用第三方库检测方法. 小型微型计算机系统, 2019, 40(2): 332–340. [doi: 10.3969/j.issn.1000-1220.2019.02.017]



徐建(1979－), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为软件分析, 智能运维, 数据挖掘.

袁倩婷(1994－), 女, 硕士, 主要研究领域为软件分析, 数据挖掘.