上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

BACHELOR'S THESIS

论文题目：TPL-V Detector

学生姓名：某　某

学生学号：0010900990

专　　业：某某专业

指导教师：某某教授

学 院 (系)：某某系

上海交通大学

学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

上海交通大学

学位论文使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。

本学位论文属于 □**公开论文**

□**内部论文**，□ 1 年/□ 2 年/□ 3 年　解密后适用本授权书。

□**秘密论文**，____ 年（不超过 10 年）解密后适用本授权书。

□**机密论文**，____ 年（不超过 20 年）解密后适用本授权书。

（请在以上方框内打“✓”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

TPL-V Detector

摘　要

中文摘要应该将学位论文的内容要点简短明了地表达出来，应该包含论文中的基本信息，体现科研工作的核心思想。摘要内容应涉及本项科研工作的目的和意义、研究方法、研究成果、结论及意义。注意突出学位论文中具有创新性的成果和新见解的部分。摘要中不宜使用公式、化学结构式、图表和非公知公用的符号和术语，不标注引用文献编号。硕士学位论文中文摘要字数为 500 字左右，博士学位论文中文摘要字数为 800 字左右。英文摘要内容应与中文摘要内容一致。

摘要页的下方注明本文的关键词（4～6 个）。

关键词：上海交大，饮水思源，爱国荣校

ABSTRACT

Shanghai Jiao Tong University (SJTU) is a key university in China. SJTU was founded in 1896. It is one of the oldest universities in China. The University has nurtured large numbers of outstanding figures include JIANG Zemin, DING Guangen, QIAN Xuesen, Wu Wenjun, WANG An, etc.

SJTU has beautiful campuses, Bao Zhaolong Library, Various laboratories. It has been actively involved in international academic exchange programs. It is the center of CERNet in east China region, through computer networks, SJTU has faster and closer connection with the world.

Key words: SJTU, master thesis, XeTeX/LaTeX template

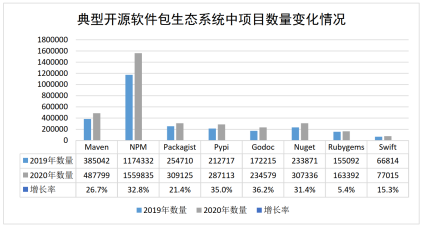
目 录

第一章　绪论

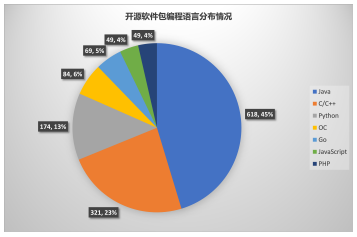
1.1　软件供应链概况

随着容器、微服务等新技术日新月异，开源软件成为业界主流形态，软件行业快速发展。现代软件大多数是被“组装”出来的，不是被“开发”出来的。据 Forrester 统计，软件开发中，80-90% 的代码来自于开源软件。因此，现代软件的源代码绝大多数是混源代码，由企业自主开发的源代码和开源软件代码共同组成。

根据奇安信代码安全实验室的检测与统计[1]，八个典型的开源软件包生态系统发展迅猛，呈现繁荣态势，包括 Maven、NPM、Packagist、Pypi、Godoc、Nuget、Rubygems、Swift。2019 年与 2020 年各开源软件包生态系统增长情况如图1–1a：



a) 2019 和 2020 年八个典型开源软件包生态系统的增长情况



b) 八个典型开源软件包生态系统的 1364 个项目的编程语言分布情况

图 1–1 奇安信对八个典型开源软件包生态系统的增长情况与编程语言分布的统计结果

Maven、Nuget、NPM 包生态系统的开源项目进展速度尤为突出，开源项目的平均版本数超过了 11 个。如表所示1–1，八个典型的开源软件包生态系统的开源项目数量和版本数量如表所示。其中 Maven 包生态系统高居榜首，每个项目的版本数平均值为 18.0，Nuget 包生态系统和 NPM 包生态系统分别位于第二和第三，平均值分别为 11.0 和 9.8[1]。

表 1–1 奇安信实验室统计的八个典型的开源软件包生态系统的项目数量和版本数量

包生态系统	项目数	版本数	平均版本数
Maven	487799	8785416	18.0
NPM	1559835	17148119	11.0
Packagist	309125	3035815	9.8
Pypi	287113	2419533	8.4
Godoc	234579	1109833	4.7
Nuget	307336	3588880	11.7
Rubygems	163392	1094135	6.7
Swift	77015	474314	6.2

如图1–1b所示，在检测的 1364 个开源项目中，所涉及的 7 个编程语言 Java、C/C++、Python、OC、Go、JavaScript、PHP 里 Java 项目数为 618，占据了 45% 的最高比例。

1.2 软件供应链安全现状及研究意义

软件供应链的上游软件可能悄无声息地影响着下游产品。开源软件之间的依赖关系错综复杂，在开发过程中，开发者通常借助包管理程序实现自动管理，因此可能意识不到产品中包含里数量巨大的开源软件。一旦某个上游的开源软件被发现安全漏洞，软件开发者无法立即意识到漏洞同时被引入到了产品中，隐含里巨大的软件供应链安全风险。

2020 年 5 月，GitHub 披露了 Octopus Scanner 漏洞[2]，该漏洞是针对 Apache NetBeans IDE 项目的开源软件供应链攻击，影响到了 26 个开源项目。

2020 年 12 月，安全公司 FireEye 发现全球著名的网络安全管理软件供应商 SolarWinds 遭遇国家级 APT 团伙高度复杂的供应链攻击。该攻击在 SolarWinds 的一个数字签名组件 DLL 中插入后门，该后门通过 HTTP 协议与第三方服务器通信。

安卓软件的供应链安全 Appbrain[3]追踪了 450 个流行的库，统计结果显示它们在安卓生态系统中有着广泛的使用，广告库、社交网络库、以及手机设备分析库尤为受欢迎。如此广泛的第三方库使用在加速开发过程、避免重复造轮子的同时，也吸引着攻击者将目标向软件供应链上游移动，通过利用受欢迎的库的漏洞来达到攻击应用的目的。atvhunter 2-4。来自 Trend Micro 的安全研究团队披露百度提供的 SDK 中的 Moplus 包含的功能可能被恶意使用，以向用户设备植入后门[4]。这一处于软件供应链上游的漏洞已经流入超过 14000 款安卓 APP，可能使得约 1 亿用户处于黑客的攻击风险中。

2022 年 4 月 Google Play 商店内的安卓应用超过 260 万，3 月与 4 月新增应用数量均在 2 万左右，来自其他市场的应用更是不计其数。

如此数量的 APP 包含着不可忽视的供应链风险，但是由于 APP 包含着敏感信息或者具有商业价值的运行逻辑，大部分开发者基于安全和产权的考虑都会将产品进行混淆后再发布。这导致在对 APP 进行安全性检查时更加困难，识别混淆 APP 中引入的上游软件成为了亟待解决的问题。事实上，约 78% 的漏洞都是在间接的依赖中找到，可能带来的安全风险则更加难以发现[1]。

1.3 研究内容和主要贡献

在软件供应链的范畴内，本课题选择安卓软件供应链中 Java 语言的开源仓库组件的识别与分析作为主要目标，设计了应用市场 App 与标准库数据库的特征提取及匹配方法。对来自 App/标准库的包，根据目录信息构建树型结构，根包作为树的根节点，类作为中层节点，方法设置为叶子节点。以树为基础，自下而上提取特征信息，精确度达到方法级别。充分考虑到当前 App 混淆技术的成熟，在特征的提取过程所有的名称常量都被过滤掉，同时目录的结构也被最大程度简化。为了在高层特征中体现出低层特征，本文引入了模糊哈希算法；同时为了加快特征信息的匹配速度，还采用了分级特征匹配的策略以及用于优先匹配的优先级队列。

本文的主要贡献如下：

1. 本文提出识别能力精确到具体版本的第三方库检测系统 TPL-V Detector，实现了高准确率、高召回率、低误报率的检测方法。
2. 本文提出基于描述符类型以及字节码指令序列的特征生成方法，使得 TPL-V Detector 在检测混淆 App 方面仍有较好的表现。

3. 本文引入了粗/细粒度两级特征，并在数据库匹配过程中加入优先级队列策略，有效优化了朴素的匹配方法，节省了时间开销。
4. 本文在 Maven 数据集上进行了评估，能够准确将混淆前后的 SDK 一一匹配，在 6 个库超过 200 个版本的请胯下实现了 100% 的准确率。
5. 本文对自建 App 数据集进行了表现测试，详细分析了不同阶段的时间开销，在综合版本识别与准确率等多个因素考虑下具有较好的性能，准确率高达 98.9%。

1.4 论文内容安排

本文介绍了软件供应链条中，上游 Java 开源仓库的标准组件及其版本在下游混淆 App 产品中的识别和检测。

第一章 简要介绍了软件供应链的蓬勃发展现状，以及当前产品开发流程中对标准组件的高重用；接着通过近年来所发生的较为严重的针对软件供应链的攻击事件，阐述软件供应链安全风险影响范围之广，影响程度之深，表明了该问题的紧迫性和严重性。最后，提出了本课题基于这一背景的研究内容和主要贡献。

第二章 介绍了常用的 App 混淆技术及其对 App 中第三方库识别带来的困难；继续分为混淆库检测、无先验知识检测和有先验知识检测三个类别简述了研究现状，讨论不同类别检测方法的基本特征；最后针对版本检测问题，论述了目前方法的匮乏以及面临的主要问题。

第三章 详细叙述了 TPL-V Detector 的系统设计和实现流程。首先提出了研究方法的概述，然后从 App 的预处理、特征树构建、特征存储与特征匹配几个步骤作了详细介绍。其中还补充了安卓应用各类文件的技术背景，并对特征树的结构、匹配流程与匹配策略等关键技术点增加了流程图与伪代码算法来辅助说明。

第四章 针对准确率、召回率、FPR 等指标，对 TPL-V Detector 的表现进行了评估，此外还深入分析了其各个阶段的时间开销，以及导致各种数据结果的可能原因。最后与现有工具进行了准确率和时间开销的比较。

第五章 总结本课题的主要贡献，分析 TPL-V Detector 的优越性与不足之处，提出可能的原因以及现存的改进点，探讨值得后续深入研究的问题，提出展望。

第二章　技术背景

2.1　App 混淆技术

App 的混淆技术通常指代码混淆（Code Obfuscation），是将程序中的代码以某种规则转换为难以阅读和理解的代码的一种行为。为了有效保护开发人员的知识产权，代码混淆技术被广泛地应用于各类 App 产品中。混淆的好处在于：它令 Apk 的逆向难度变大，增加了反编译的成本，此外 Android 当中的混淆还可以在打包时移除无用资源，显著减小 Apk 体积[5]。

2.1.1　混淆工具 Proguard

Java 平台提供了混淆工具 Proguard[6]来帮助开发者快速对代码进行混淆，对 Proguard 的描述如下：

1. 它是一个包含代码文件压缩、优化、混淆和校验等功能的工具。
2. 它能够检测并删除无用的类、变量、方法和属性。
3. 它能够优化字节码并删除未使用的指令。
4. 它能够将类、变量和方法的名字重命名为无意义的名称从而达到混淆效果。
5. 它会校验处理后的代码，主要针对 Java 6 以上的版本和 Java ME。

Proguard 的工作流程由 shrink、optimize、obfuscate 和 preverify 四个步骤组成，如图2–1所示，每个步骤都是可选的，用户可以通过配置脚本来决定执行其中的哪几个步骤。

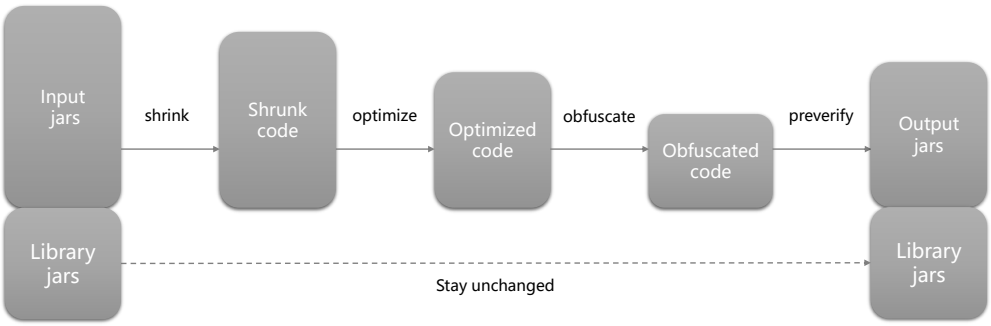


图 2–1 流行混淆工具 Proguard 的工作流程

2.1.2 标识符混淆

标识符混淆通常指将类、变量和方法的名字重命名为无意义的名称。从一款来自 360 的经过混淆处理的 Apk 得到的 class 文件如图2–2所示，文件名被重命名为简短、无意义的字母组合，相应的源代码中引入这些类的 *import* 语句也被混淆处理。从名称中无法获取任何信息。在静态链接方法下，开发人员将第三方标准库代码下载到本地，统一打包为 Jar 文件，代码混淆不仅可以应用于开发者代码，也可以应用于第三方代码，达到隐藏依赖的目的。对于攻击者而言，可以通过恶意引入被披露包含漏洞的一些标注库的旧版本，来达到向 App 中秘密植入漏洞的目的。

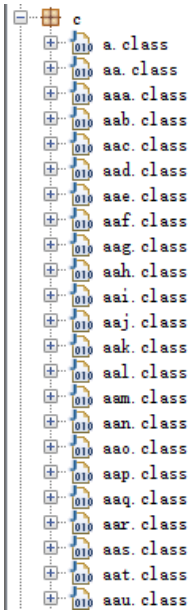


图 2–2 一款 360 软件的 apk 解压后得到的经过重命名混淆的 class 文件

2.1.3 花指令

花指令也叫垃圾指令，是指在原始程序中插入一组无用的字节，同时保持程序的原始逻辑不变，程序仍然可以正常运行，然而反汇编工具在反汇编这些字节时会出错，由此造成反汇编工具失效，提高破解难度。

花指令的主要思想是，当花指令连同正常指令的前几个字节被反汇编工具识别为一条指令时，引起反汇编工具报错。因此插入的花指令都是一些随机的但不完整的指令。不论花指令形式如何，其必须满足两个基本条件：

1. 在程序运行时，花指令位于一个永远不会被执行的路径中。
2. 花指令也是合法指令的一部分，只不过它们是不完整的指令。

据此，需要在每个要保护的代码块之前插入无条件分支语句和花指令，无条件分支保证了程序在运行时不会运行到花指令的位置，从而引起反汇编工具在进行反汇编时报错。目前的反汇编工具主要分为两类：线性扫描算法和递归扫描算法。线性扫描算法按顺序逐个将每一条指令反汇编成汇编指令，因此会把花指令错误识别，导致程序出错[7]。递归扫描逐个反汇编指令，如果某个地方出现了分支，就会把这个分支地址记录下来，然后对反汇编过程中碰到的分支进行反汇编，因此反汇编能力更强。

2.1.4 控制流平坦化

控制流平坦化，就是在不改变源代码的功能前提下，将代码中的 if、while、for、do 等控制语句转换成 switch 分支语句。这样做的好处是可以模糊 switch 中 case 代码块之间的关系，从而增加分析难度[8]。

这种技术的思想是，首先将要实现平坦化的方法分成多个基本块——即 case 代码块，和一个入口块，为每个基本块编号，并让这些基本块都有共同的前驱模块和后继模块。前驱模块主要是进行基本块的分发，分发通过改变 switch 变量来实现。后继模块也可用于更新 switch 变量的值，并跳转到 switch 开始处。

2.2 库检测研究现状

2.2.1 检测混淆库

随着 APP 混淆技术的成熟，以第三方库能够被容易地区分为前提的方法已不适用，标识符被混淆为无意义的简短的字母组合，比如 *com.google* 可能被混淆为 *a.c*，无法提供关于库的任何信息。

T. Book 等人的工作通过白名单的方法检测 APP 内的第三方库[9]，但这类方法显然无法解决标识符重命名的问题。PEDAL[10]借助机器学习方法，从 SDK 和 Apk 中提取代码特征，训练分类器来识别广告库。PEDAL 首先将原始 Apk 转换为字节码，对每一个包的目录，计算其内字节码文件的特征，包括安卓基本组件、可选安卓权限、用户界面元素、运行时权限检查的 API 等方面。与训练的模型用基于这些特征生成的特征向量作为输入，为每一个包打上应用（App）或广告（Ad）的标签。最终 PEDAL 重写权限逻辑，重装 Apk，得到无广告的纯净应用。基于字节码的方法使得 PEDAL 不受到重命名混淆的影响。

2.2.2 检测未知库

一些研究工作提出了在没有已知第三方代码的数据库知识情况下检测 APP 组成成分的方法。此类方法通常首先把开发者代码与第三方代码进行区分，再将第三方代码聚类成不同的组件，组件即一个可能的库的候选，进一步评估候选之间的相似度，当超过相似度阈值的候选的出现次数足够多时就认为找到了一个库。

如 Chen 等人[11]从大量的 APP 中获取库，进行聚类和检测。这一方法在混淆的情况下表现不佳，因为其假设不同 APP 中包含的库的相同实例拥有相同的包名，而混淆打破了这一基本的假设。

Li 等人[12]提出了在缺少先验知识的情况下从手机 App 中提取库的方法。首先综合关系、继承关系及调用关系三个因素将 App 切分成多个部分，两个具有包含或者继承关系的包被定义为同质包（Homogeny Package）。再根据包内函数对包外函数的调用关系，将不同的包放入一个集合当中，此集合就作为一个第三方库的候选。对每一个方法，按照其控制流图中的基本块的操作码序列的哈希值生成基本块的特征。接着将基本块的特征排序和连接，生成方法的特征，并以此类推直到生成库级别的特征。用特征来计算库之间的相似度，当某个集合内的候选库满足相互之间的相似度超过预定义的阈值，且集合的规模的足够大时，就认为找到了一个真正的第三方库。这一方法不要求先验知识，能够有效发现新的第三方库，但是只有当数据集中包含足够多的 App 时才能够使得相似候选库高频率出，一些不流行的库，即便是来自开源仓库的标准库，也难以检测出来。

为解决包名混淆问题，LibRadar[13]使用特征哈希的方法，不需要基于包名的聚类，而是借助包中的目录结构来识别库的候选，具体来说是将一个候选表示为一个目录树的结构。这引入一个新的假设，即包的结构在混淆过程中不改变。但混淆工具可以将不同的包合并为一个包，很容易打破这一假设。WuKong[14]和 AnDarwin[15]用控制流图和 API 数量来定义哈希特征，用来计算各候选库的相似度。考虑到混淆工具可能修改一个方法的控制流图，或者移除在 APP 运行中未真正使用的方法，哈希的质量影响着这两类方法的表现性能。

2.2.3 检测已知库

基于已知库的检测要求关于现存库的知识，如库的基本信息、哈希特征等，在混淆 APP 第三方库识别的场景下，用构建知识数据库的代价换取了更好的表现。

LibRoad[16]使用了典型的基于先验知识数据库的检测混淆 App 第三方库的模型。如图2–3所示，LibRoad 首先对 App 目录进行解析，分出经过混淆处理的部分以及未经过混淆处理的部分。对于未经混淆的包，通过包名匹配策略可以轻松从数据库中找到对应的标准库。对于混淆处理的包，LibRoad 在类粒度上生成了类签名，同样数据库中的各类也预先生成和存储好了签名。首先进行本地数据库匹配，如果匹配失败则进行在线匹配，若再次失败则暂定为找到了新的三方库，输出到 TPL（Third-Party Library）list 中。

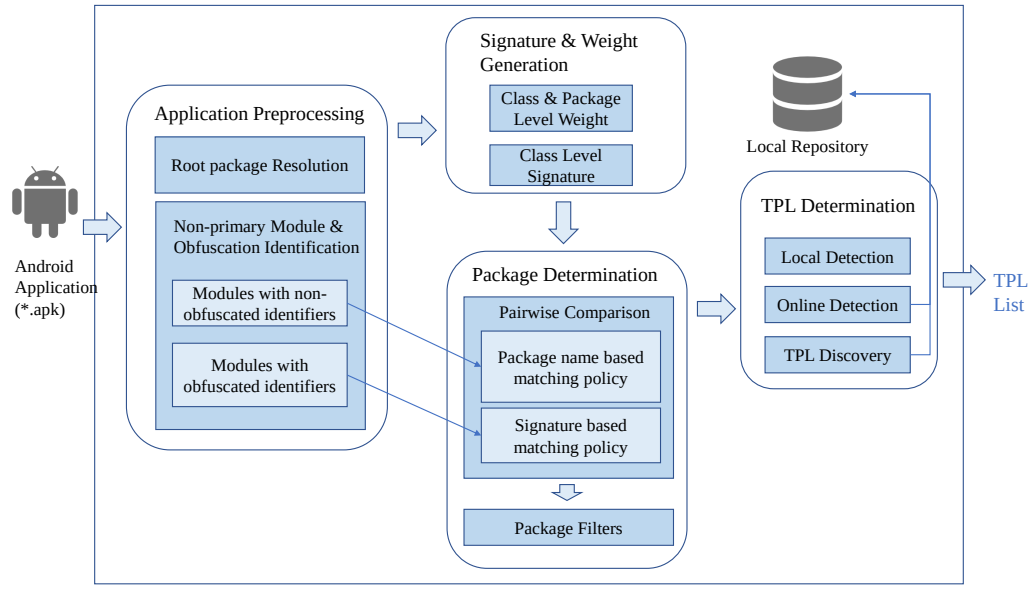


图 2–3 LibRoad 的设计框架和工作流程

另外一个具有代表性的工具是 LibScout[17]，将一个包构建为一个 Merkel 树，根包作为树的根节点，依次往下，方法作为叶子节点。使用 Java 字节码来计算方法的哈希值作为特征，进一步生成类的哈希作为特征，进行 APP 与数据库中第三方库的匹配。定义来自标准库的包 lp 与来自 App 的包 ap 的匹配的得分为

$$score_c(lp, ap) = \frac{\#\ classes\ in\ lp\ that\ match\ in\ ap}{\#\ classes\ in\ lp} \in [0, 1] \tag{2–1}$$

超过得分阈值的 App 的包作为标准库的包的一个匹配候选。为了加速匹配过程，对一个标准库的包 lp 与它的一个匹配候选 ap，二者拥有相同的根目录结构，因此对于根目录结构不相同的其它候选可以直接剔除。标准库中包与子包的关系也被用来剔除不可能的匹配，以

降低计算代价，用 $a\ candidateOf\ b$ 表示 a 是 b 的一个匹配候选，$relationship(a,b)$ 表示 a 是 b 的父/兄弟/子节点，则剔除不满足以下关系的候选：

$$\begin{aligned} &\forall\, ap_i, ap_j, lp_x, lp_y, \\ &ap_i\ candidateOf\ lp_x, ap_j\ candidateOf\ lp_y, \\ &relationship(ap_i, ap_j) = relationship(lp_x, lp_y) \end{aligned} \tag{2–2}$$

以上表达式说明，若 ap_i、ap_j 分别是 lp_x、lp_y 的匹配候选，而对于来自标准库的 lp_x 和 lp_y 所拥有的已知的关系，ap_i 和 ap_j 之间并不满足同样的关系，则此匹配候选是无效的，不需进一步计算。最终每个标准库的包在经过筛选后的全局范围内计算得分 $simScore$

$$simScore_c = \frac{sum\ of\ matched\ classes}{\#\ of\ classes\ in\ library} \tag{2–3}$$

最大者作为最终匹配结果，即认为该包所属的 App 中使用了此标准库的包。LibScout 生成树形结构，充分利用各个方法特征的特性能够抵抗控制流篡改，同时以字节码和返回值、参数类型为特征，在包/类/描述符重命名情况下依然有效。

2.3 标准库的版本检测

现有工作中以版本为目标实现精确检测的并不多，AdDetect[18]仅能够区分广告和非广告的库，基于聚类的方法如 LibRadar[13]，LibD[19]等都没有声明能够检测库的特定版本。

实现版本的检测仍面临着很多问题：

1. 需要处理庞大的数据集。第三方库本身就纷繁复杂，如果再将各个版本考虑进去，将导致需要处理的数据成倍增长。
2. 缺乏精确的表示。一个库的不同版本可能差异微小，如何找到合适的特征来区分这一差别非常关键。
3. 代码混淆的干扰。代码混淆同样会导致库的代码发生改变，这种改变是由不同库引起还是由同一库的不同版本引起，需要被准确的区分。

第三章　TPL-V Detector 系统设计

在参考了多篇文献后，我提出了一种适用于包的结构混淆、包/类/标识符重命名场景的，基于已知标准库的数据库，利用两类信息生成粗粒度/细粒度两级哈希特征的安卓应用第三方库及其特定版本的检测方法。

3.1　方法概述

此方法不依赖于包中的目录结构以及各级名称，因此可以抵抗结构混淆以及重命名混淆，包括了四个步骤：

1. 预处理 jar、aar 和 apk。将来自 Maven 仓库的 jar 包、aar 包以及待检测 apk 处理成便于构建树结构的形式。
2. 构建特征树。根据上一阶段输出，将每个包作为根节点构建特征树，该包内的所有类，不论是根包的类还是子包的类，一律作为树的中间层节点，各类的方法作为叶子节点。特征分为粗粒度、细粒度两级特征。粗粒度特征为方法的描述符的返回值以及参数类型，细粒度特征为该方法的字节码，首先生成叶子节点的两级特征，再利用叶子节点生成中间层节点即类节点的特征。
3. 构建数据库与匹配。根据以上特征生成方法，计算 Maven 仓库中的标准库的特征，并存储到数据库中。对待检测 APP，首先生成粗粒度特征，确定所包含的库，再根据细粒度特征，确定各库的具体版本。

3.2　包的预处理

3.2.1　Dex 与 Class 简介

DEX 文件　是 Android 系统中的一种文件，是一种特殊的数据格式，能够被 Dalvik 虚拟机识别并加载执行，类似于 Windows 上的 EXE 可执行文件。将 APK 安装包解压后得到的文件就包含了 DEX 文件，它记载了应用程序的全部操作指令以及运行时数据。当 java 程序编译成 class 文件后，还需要使用 dx 工具将所有的 class 文件整合到一个 DEX 文件里，目的是其中各个类能够共享数据，在一定程度上降低了冗余，同时也使文件结构更加紧凑。DEX 文件大小通常是传统 jar 包的 50% 左右。

CLASS 文件　是能够被 java 虚拟机识别，加载并执行的文件格式，通过 javac 程序可以从 java 源文件生成 class 文件。class 文件记录了一个类文件的所有信息，不仅包含了 java 源代码中的信息，还包括了 this、super 等关键字的信息。作为一种 8 位字节的二进制六文件，class 中的数据按顺序紧密排列，没有间隙，从而让 JVM 加载更加迅速，每一个类、接口或者枚举都单独占据一个 class 文件。

3.2.2　Apk 与 jar 的预处理

预处理流程如图3–1所示。

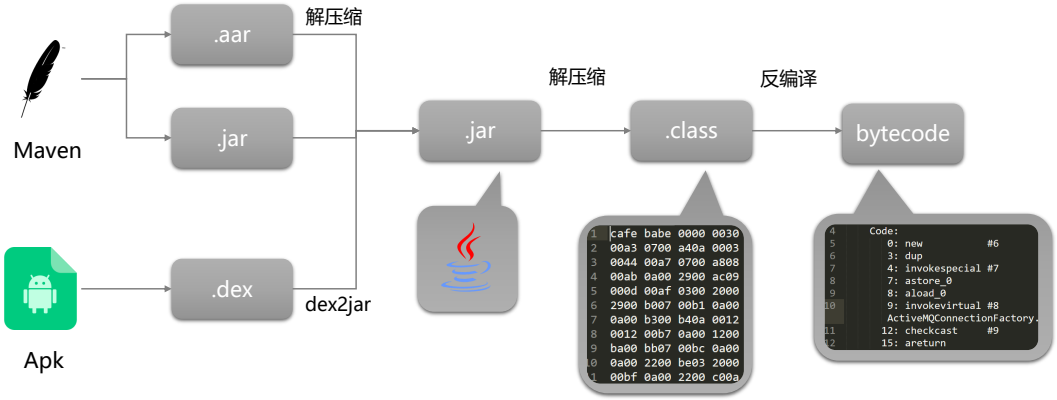


图 3–1 Apk 与标准库转换到 java 字节码的预处理流程

Apk 预处理： 从应用市场获取待检测的应用的安装包，即 apk 文件。Apk 文件本质上为 zip 格式的压缩文件，解压后可以得到 dex 文件。借助 dex 到 jar 的转换工具 dex2jar[20]将 DEX 文件转换为包含多个 class 文件。此处的 class 文件通常是经过软件开发者混淆的，大部分类的名称和函数的名称经过了混淆处理，不能提供有效信息，转换得到的包的目录结构也不可靠。

Jar 预处理： Maven 社区提供的中央仓库[21]包含了大量常用的库，包括绝大多数流行的 Java 开源构件，源码，许可证信息等，一般来说简单的 Java 项目依赖的构建都可以满足，因此选择 Maven 仓库中的部分 jar 包作为构建数据库的基础。我利用爬虫从 Maven 中央仓库中获取了大量的 jar 包或 aar 包，aar 包需要先解压一次获得其中的 jar 包，将 jar 包解压就可以获得该标准库的各个 class 文件。在爬取标准库的过程中，需要跳过一些带有 javadoc 或者 sources 字样的链接，这表示该包为一个说明文档包或者源代码包，在此次工作中用不到。

字节码的生成： Java 字节码是一种程序的低级表示，能够直接被 java 虚拟机所理解和翻译，以实现 java 跨平台的特性。具有二进制格式的 class 文件，实质上就是 java 的字节码，属于独立于平台和操作系统的指令集。为了删除 class 文件中的常量信息，我使用 javap 程序将 class 文件反编译为具有可读性的助记符形式的字节码，可以清楚地区分开描述符，代码，常量池等部分，并存储为文本文件供后续使用。

3.3 构建特征树

3.3.1 两级特征

如果特征的计算方法过于复杂，将会导致在获取 apk 特征时用时过长，因此本方法中采用了两级特征——粗粒度特征与细粒度特征，来解决这一问题，同时实现第三方库的版本检测。粗粒度特征计算方法简短、生成速度快、相应的精准度有所下降，用于确定待测 apk 中

的包是数据库中的哪一个标准库。细粒度特征计算方法复杂、生成速度慢、但可以精确到单条字节码操作指令的层面。对于细粒度特征而言，即便同一个库的版本不同所导致的细微代码差异也能够体现出来，因此可以在包成功匹配的情况下进一步匹配具体版本。

(1) 粗粒度特征

粗粒度特征由函数的描述符生成，由于函数描述符包含了函数名、参数名，极易受到重命名混淆影响，因此我将名称部分删去，以*返回值类型（参数 1 类型，参数 2 类型，...，参数 n 类型）*的字符串作为描述符，计算其 md5 哈希值，得到该方法的签名，一个例子如表3–1所示。尽管此签名可能在多个包中，乃至一个包中的不同类中出现，但是结合该类下的各个方法的签名，方法的序列，可以减少碰撞的概率，用于初步表示一个类。如表3–4所示，标准库 *org.codehaus.avtivemq* 的两个类 *ActiveMQMessageConsumer* 和 *BrokerClientImpl* 在方法“*public java.lang.String toString()*”上发生了碰撞，但是在其他方法上有很大差异，方法的总数也不同，从而类的层面的特征有所区别。

表 3–1 方法描述符的处理

原始描述符	public static ActiveMQConnection makeConnection(String user, String password, String uri)
删除名称后的描述符	public static org.codehaus.activemq.ActiveMQConnection makeConnection(java.lang.String, java.lang.String, java.lang.String)
md5 哈希值	befc542005082b1940176d89035826ab

表 3–2 标准库 org.codehaus.activemq 中的两个类包含方法（部分）的情况

描述符	ActiveMQMessageConsumer	BrokerClientImpl
public java.lang.String toString();	✓	✗
protected long getStartTime();	✓	✗
protected void setBrowser(boolean);	✓	✗
public void updateBrokerCapacity(int);	✗	✓

(2) 细粒度特征

细粒度特征基于函数的字节码生成。*ActiveMQConnection* 类的一个方法 *makeConnection* 的字节码如下所示：

```
public static org.codehaus.activemq.ActiveMQConnection makeConnection(
java.lang.String) throws javax.jms.JMSException;
  Code:
     0: new           #6
     3: dup
     4: aload_0
     5: invokespecial #10
```

```
 8: astore_1
 9: aload_1
10: invokevirtual #8
13: checkcast     #9
16: areturn
```

Java 字节码是基于堆栈结构的，上面字节码“Code”部分的每一行对应于一条操作指令，仅仅表示对堆栈的操作，而不包含操作数。源代码中所定义的常量、变量名等静态成员存放在常量池中，因而避免了名称混淆的问题。每一条操作指令对应于一个十六进制的操作码，表3–3展示了部分对应关系及其说明。

表 3–3 Java 字节码助记符与十六进制操作码对应关系（部分）

助记符	操作码	说明
new	0xbb	创建一个对象，并将其引用值压入栈顶
dup	0x59	复制栈顶数值，并将复制值压入栈顶
aload_0	0x2a	将第一个引用类型本地变量推送至栈顶
invokespecial	0xb7	调用超类构造方法，实例初始化方法，私有方法
astore_1	0x4c	将栈顶引用型数值存入第二个本地变量
aload_1	0x2b	将第一个引用类型本地变量推送至栈顶
invokevirtual	0xb6	调用实例方法
checkcast	0xc0	检验类型转换，检验未通过将抛出异常
areturn	0xb0	从当前方法返回对象引用

3.3.2 特征树的实现

(1) 字节码特征提取

如图3–2所示，字节码解析器布置在树的下方，其接收字节码文件，从文件头开始读入，利用正则表达式匹配每一个函数的描述符。当匹配成功时，表明一个函数的读入即将开始，则解析器依次读取函数描述符和操作指令序列，并将原始特征记录为一个特征对。当读至文件尾时，该 class 文件代表的类的所有成员函数全部读取完毕，将各特征对传递给上方的树。

(2) 方法层

方法实现为 *mNode*（Method Node）类。根据提取出来的字节码的原始特征，对于每一对粗/细粒度原始特征，生成一个方法节点，对原始特征进行处理。方法节点记录了原始的函数描述符，文件路径等基本信息，同时生成了处理后的特征的 MD5 哈希值，作为此方法的两级特征。表为 *ActiveMQConnection* 类中方法 *createSession* 的各个属性值。

(3) 类层

类实现为 *cNode*（Class Node）类，类层根据该 class 文件中包含的方法，将各个方法对应的 *mNode* 节点作为自己的子节点。不同类的方法数大相径庭，有的类没有方法，只有数据成员，有的类则包含几十甚至上百个函数成员。再考虑到类的不同版本之间可能差异很

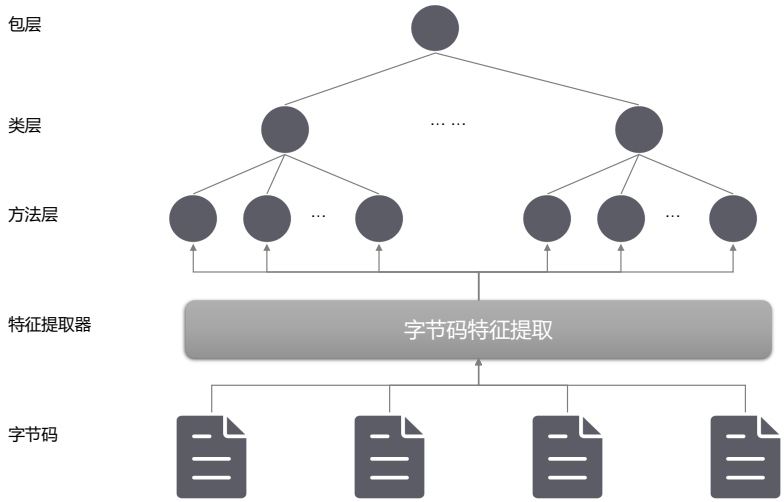


图 3–2 使用特征提取器从字节码文件中获取特征并存储为树型数据结构

小，仅仅体现在个别方法的几个函数上，类的特征必须能体现函数成员的特征。因此我采用了模糊哈希的方法，算法如3–1所示，将类下各方法的特征进行排序、连接，计算模糊哈希值作为类的粗/细粒度特征。

此处模糊哈希的用法与 ATVHunter[22]有相似之处，都是为了降低代码混淆对最后特征带来的影响，但是 ATVHunter 的目标是减少部分指令改变对函数签名的影响，而此处是减少部分函数改变对类的特征的影响。如图3–3所示，模糊哈希的应用步骤为：

1. 将类下各方法的特征连接成一个序列。
2. 使用滑动窗口（滚动哈希）将序列分割成不同的切片。
3. 对每个切片，计算其 MD5 哈希值，取最后 6 位作为压缩映射值。
4. 最后将所有映射值连接起来作为最终值。

表 3–4 标准库 org.codehaus.activemq 中的两个类包含方法（部分）的情况

方法	org.codehaus.activemq.ActiveMQConnection$createSession
描述符特征	public javax.jms.Session(boolean, int)
字节码指令序列	2a b6 2a b6 bb 59 2a 1b 99 03 a7 1c b7 b0
粗粒度特征	d8a58a75cf9b3272e8736cd74256fdc7
细粒度特征	45422e23d690d166bc6dd144f226076f

(4) 包层

包实现为 *pNode*（Package Node）类，是一个特征树的根节点。包层不产生特征，用于管理属于该包下的各个类。为了解决 APK 产生的包中可能存在目录结构的混淆问题，每一个包都采用根包作为唯一的根节点，其下包含的子包以及子包中的二级子包等所拥有的类

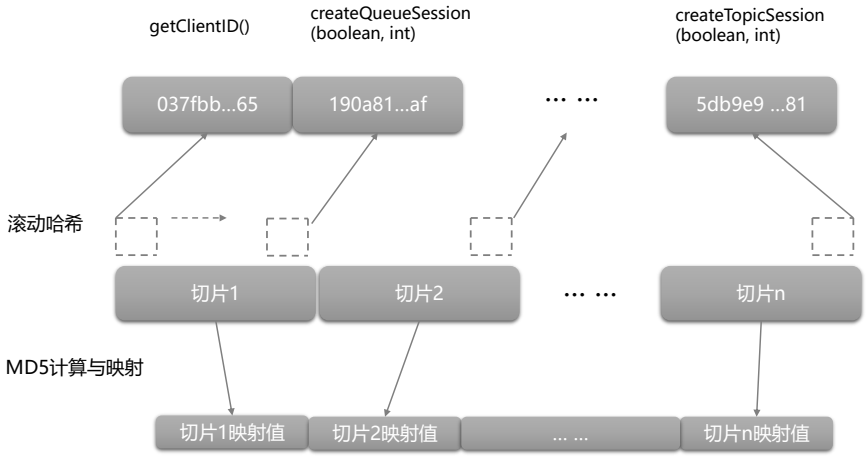


图 3–3 使用特征提取器从字节码文件中获取特征并存储为树型数据结构

算法 3–1 类的模糊哈希值计算

Data: 类节点 cNode
Result: 类的模糊哈希值 Fuzzy_hash

```
initialization;
sort(cNode.getMethods());
Sequence←null;
for mNode ∈ cNode.getMethods() do
    Sequence ← Sequence cascade mNode.getFeature();
end
Slices←rolling_hash(Sequence);
Fuzzy_hash←null;
for slice ∈ Slices do
    F_hash_slice←mapping_function(MD5(slice));
    Fuzzy_hash←Fuzzy_hash cascade F_hash_slice;
end
```

都归为根包的类。

最后将从包到方法的各个节点封装为一个 *javaTree* 类，负责统筹管理从文件的筛选到各级节点的生成等诸多工作。

3.4 特征存储

为了便于特征的查找与管理，我将哈希树输出的特征存放在 MySQL 数据库中，步骤如下：

1. 设计数据库 scheme：tpl 如表3–5所示，在 tpl 下创建表 maven，存储从 Maven 仓库获取的标准库。

2. 遍历本地爬取到的 Maven 仓库，当文件夹中包含 *META-INF* 时表明当前目录为一个根包的目录。
3. 在根包的目录上构建哈希树，解析目录中的文件，形成该包下各类、各方法的特征。
4. 用插入语句将特征存储到数据库中

表 3–5 MySQL 数据库中 tpl 的 scheme

字段	package	class	method	coarse feature	fine feature
数据类型	varchar(255)	varchar(255)	varchar(1024)	varchar(255)	varchar(255)

3.5 特征匹配

表 3–6 标准库与安卓应用的相关符号及说明

符号	说明
T_{sim}	相似度阈值，相似度超过此值的两个特征认为互相匹配
$feature_{lib}$	来源于标准库的一个类或者方法的特征
$feature_{app}$	来源于安卓应用的一个类或者方法的特征
C_{lib}	标准库所包含的类的集合
C_{app}	安卓应用所包含的类的集合
M_{class}	类所包含的方法的集合
$Similarity_p$	两个包之间的相似度
$Similarity_c$	两个类之间的相似度

(1) 相似度

为了表征标准库与来自 Apk 的包的相似程度，我引入了相似度来量化这一概念。相似度适用于不同来源（Maven/APP）的类或方法的匹配，是一个介于 0 和 1 之间的数，1 表示完全匹配。若要判断 APP 所包含的包是否存在于数据库中，需要查看数据库中是否存在足够数量的类，使得这些类的特征值与 APP 中类的特征值的相似度都超过了一定阈值。由于类的特征值是根据方法的特征值生成的，因此类的相似程度能过说明其内各个方法的相似程度。类和方法的特征均为字符串序列表示的哈希值，因此用编辑距离来计算两个序列之间的相似度：

$$Similarity(feature_{lib}, feature_{app}) = \frac{edit\_distance(feature_{lib}, feature_{app})}{max\{length(feature_{lib}), length(feature_{app})\}} \quad (3\text{–}1)$$

对于待定的阈值 T_{sim}，定义来自标准库的特征 $feature_{lib}$ 与来自 APP 的特征 $feature_{app}$ 匹配，如果满足：

$$Similarity(feature_{lib}, featureapp) \geq T_{sim} \quad (3\text{–}2)$$

同时，标准库的规模也应该纳入考虑范畴。一些大规模的类、或者相似的类，可能在部分方法上有相似之处，在实现逻辑上有相同点，因此不能仅仅因为成功匹配其下的一部分类就将其所谓候选库。而一些小规模的类，方法数可能很少，一定数量的类匹配就说明其有很大概率就是 APP 所使用到的库。一种简明的相似度计算方法如下：

$$Similarity_p(lib, app) = \frac{|\{c_1, c_2, \ldots, c_n\}|}{|C_{lib}|} \in [0, 1] \tag{3–3}$$

其中，c_i 满足：

$$(a)\ c_i \in C_{lib} \tag{3–4a}$$

$$(b)\ \exists\, c' \in C_{app} Similarity(c_i, c') \geq T_{sim} \tag{3–4b}$$

类似地，用 $Similarity_c$ 表示方法相似度，也可以在方法粒度上实现更为精确的匹配：

$$Similarity_c(class_{lib}, calss_{app}) = \frac{|\{m_1, m_2, \ldots, m_n\}|}{|M_{class_{lib}}|} \in [0, 1] \tag{3–5}$$

其中，m_i 满足：

$$(a)\ m_i \in M_{class_{lib}} \tag{3–6a}$$

$$(b)\ \exists\, c' \in M_{class_{app}} Similarity(m_i, m') \geq T'_{sim} \tag{3–6b}$$

(2)　匹配策略

来自 Maven 中央仓库的标准库数量巨大，一个包中可能包含多个子包，包的版本数可能超过 50 个。一种朴素的匹配方法是：将来自 Maven 的标准库与来自 App 的包两两匹配，计算其下类的相似度，但这会产生以亿为单位的配对数，耗时严重。因此我引入了如图3–4所示的两种策略来加速匹配的过程，分别是两级特征与优先级队列。

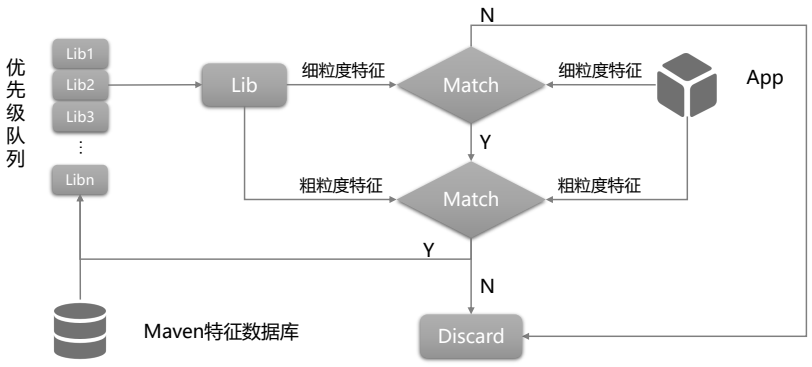


图 3–4　数据库匹配的两种加速策略——两级特征与优先级队列

两级特征策略 包含粗粒度、细粒度两种计算方法复杂度不同、精确度不同的特征，思想为首先利用粗粒度特征初步缩小匹配范围，即 App 中使用到了 Maven 的哪些标准库，再用细粒度特征确定这些库的具体版本。一个库如果没有通过粗粒度筛选，说明其方法的数量、各方法的参数数量、返回值及参数的类型这些特征与 App 存在较大差异，因此它几乎不可能在 App 中使用到，对这样的库进行具体版本的匹配也是没有意义的。对于通过粗粒度筛选的库，说明它在方法层面与 App 中的包有一定相似程度，此时再通过基于字节码生成的细粒度特征来精准匹配，不同版本所带来的细微差异也能够体现在特征中。

优先级队列策略 基于一个“局部性”假设：前一个来自 App 的类匹配完毕时，新到来的类有较大可能与前一个类处于同一个包中。

这一假设在直观上是合理的，因为在匹配的过程中，我按照 App 目录中的文件顺序进行处理，相邻的类处于同一个目录下。尽管 App 可能经过了目录结构的混淆处理，即将来自不同包的类混入到一个目录下，但除非所有的包都做了混淆且每一个类都被单独打乱路径而非以多个类为批次操作，否则此假设在大多数情况下都成立。

为了充分利用这一假设，对待匹配的 App 设置一个优先级队列。初始时优先级队列为空，从数据库中按序取出 K 个类的记录加入到队列当中。队列中元素的优先级 $Priority$ 定义为与当前待匹配 App 的类的相似度，其中细粒度特征的相似程度由于包含了更多的原始信息而被赋予更大的权重：

$$\begin{aligned} Priority(c_{lib}, c_{app}) = 0.3 \times Similarity(feature_{c_{lib}}^{coarse}, feature_{c_{app}}^{coarse}) \\ + 0.7 \times Similarity(feature_{c_{lib}}^{fine}, feature_{c_{app}}^{fine}), \\ c_{lib} \in C_{lib}, c_{app} \in C_{app} \end{aligned} \tag{3–7}$$

匹配算法如3–2所示，依次将优先级队列中的记录取出，计算与待匹配类的相似度，进一步计算优先级，按照优先级从大到小重排队列。队列中的记录全部使用完毕后，从数据库中取出一条新的记录，计算相似度与优先级，并插入到队列的合适位置，如果优先级小于队列尾部记录的优先级则不进入队列。以此类推，将数据库中的所有记录进行匹配。当前待测类匹配流程完毕后，从 App 中取出下一个待测类，首先与优先级队列中的记录匹配，队列匹配完毕后再从数据库中取得记录，按照此方法将 App 中的所有类进行匹配。

对于某次匹配结果：

1. 如果完全匹配，即相似度为 1，则中止当前流程，直接记录此类属于数据库中类所在的标准库。
2. 如果相似度小于 1，则以优先级队列中优先级最高的记录为最佳匹配，并将该记录来自的标准库作为当前待测类的识别结果。

算法 3–2 标准库与 App 的匹配算法

Data: 标准库特征数据库 C_{lib}，App 特征数据库 C_{app}
Result: 匹配结果 $Package$

Initialize priority queue $Queue_p$;
Initialize mapping relation $Package \leftarrow empty\ dictionary$;
for $c_{app} \in C_{app}$ **do**
  if $isEmpty(Queue_p)$ **then**
    for $c_{lib} \in C_{lib}$ **do**
      $priority \leftarrow Priority(c_{lib}, c_{app})$;
      $Queue_p$ėn$Queue(c_{lib}, priority)$;
      if *priority=1* **then**
        $break$
      else
        $continue$;
      end
    end
  else
    for $c_{lib} \in Queue_p$ **do**
      $priority \leftarrow Priority(c_{lib}, c_{app})$;
      if *priority=1* **then**
        $break$
      else
        $continue$;
      end
    end
    for $c_{lib} \in C_{lib}$ **do**
      $priority \leftarrow Priority(c_{lib}, c_{app})$;
      $Queue_p.enQueue(c_{lib}, priority)$;
      if *priority=1* **then**
        $break$
      else
        $continue$;
      end
    end
  end
  $c_{lib}^{match} \leftarrow Queue_p.first()$;
  $Package[c_{app}] \leftarrow c_{lib}^{match}.get\_package()$;
end

第四章　混淆第三方库检测实验

4.1　系统实现与实验概述

4.1.1　系统实现

TPL-V Detector 系统的控制流程部分用 Python 实现，负责 jar 包、apk 等文件的处理，调用命令行工具 javap 对 class 文件进行反编译，以及对字节码文件的筛选。特征树的构建用 C++ 实现，负责从字节码中提取特征，建立树状数据结构，以及生成各类的特征。其中 Python 通过 pymsql 库连接 MySQL 数据库，实现特征的存储和管理、特征的查询与匹配。系统实现包含脚本文件共 19 个，代码行数约 2500 行。其他配置见表4–2。

表 4–1 系统实现与实验环境配置

配置	详情
CPU	Intel(R) E5-2630 v4
OS	Ubuntu 16.04.5 LTS
Core	8
Python	3.6.5
GCC	5.4.0
MySQL	8.0.29
Memory	8GB

4.1.2　实验流程

实验使用自建数据集，包含了 8 个来自 Maven 的标准库，共 40 个版本，以及 10 款来自第三方市场的 App。实验首先在 SDK 数据集上进行，测试类的相似度阈值的最佳取值，然后使用混淆工具 Proguard 对 Maven 标准库进行混淆，将混淆前的标准库和经过混淆的标准库分别交由 TPL-V Detector 系统处理和存储，以未混淆的标准库作为先验知识，检测混淆过后的标准库，最后对其各项指标进行评估。接下来在真实数据上即来自第三方市场的 App 上进行实验，首先获取 App 中包含的第三方库，作为先验知识数据库，随后由 TPL-V Detector 进行检测，将结果与实际情况进行对比，评估其准确率及其它指标。此外，本文还将系统与现有的一些其他检测方法在准确率和耗时上进行了对比，以分析系统的优越性与不足之处。

4.1.3　评价指标

系统评价指标包括准确率（Accuracy）、精确率（Precision）、召回率（Recall）、F1 得分（F1 score）和误报率（FPR），各自定义如下：

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4–1a}$$

$$Precision = \frac{TP}{TP + FP} \tag{4–1b}$$

$$Recall = \frac{TP}{TP + FN} \tag{4–1c}$$

$$F1\_score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{4–1d}$$

$$FPR = \frac{FP}{FP + TN} \tag{4–1e}$$

其中 TP、FP、TN、FN 分别代表检测结果中真阳性、假阳性、真阴性、假阴性的数量。

4.2 混淆 SDK 检测实验

4.2.1 Maven 数据集

Maven 仓库未 Java 开发者提供了丰富的开源组件，是目前最流行的开源软件包生态系统之一。本文从 Maven 中央仓库中选取了 6 个开源标准库，各库及其版本情况如表**??**所示。

表 4–2 实验所选 Maven 标准库及其版本数

库	版本数
net.sf.ezmorph	11
com.alibaba.fastjson	15
com.github.xujiaji.happy-bubble	2
com.squareup.okhttp3.okhttp	42
com.squareup.okio	20
com.tencent.mm.opensdk	2

4.2.2 阈值 T_c 的选取

阈值 T_c 用于判断两个类是否匹配，如果类 $class_a$ 和类 $class_b$ 满足特征的相似度超过阈值，定义两者具有匹配关系 $match$，即：

$$\begin{aligned} & class_a \ match \ class_b, \\ if \ similarity(class_a, class_b) = 1 - & \frac{edit\_distance(feature_a, feature_b}{length(feature_a) + length(feature_b)} \\ & > T_c \end{aligned} \tag{4–2}$$

阈值 T_c 的选取非常关键，对实验结果影响较大。阈值过高会导致匹配对细微的差异过于敏感，导致准确率降低；阈值过低会导致匹配门槛过于宽松，准确率虽然有所升高，但是误报率也会相应升高。因此本文对随机字符串的编辑距离，即 Levenshtein 距离的分布情况进行了描绘以确定合适的阈值选取范围。对于所含字符随机、长度随机的 1000 个字符串，分成 500 对分别计算其 Levenshtein 距离，进一步按照本文方法计算相似度得分，并将 [0,1) 平均分为 10 个区间，统计落入各区间的字符串对的数量。

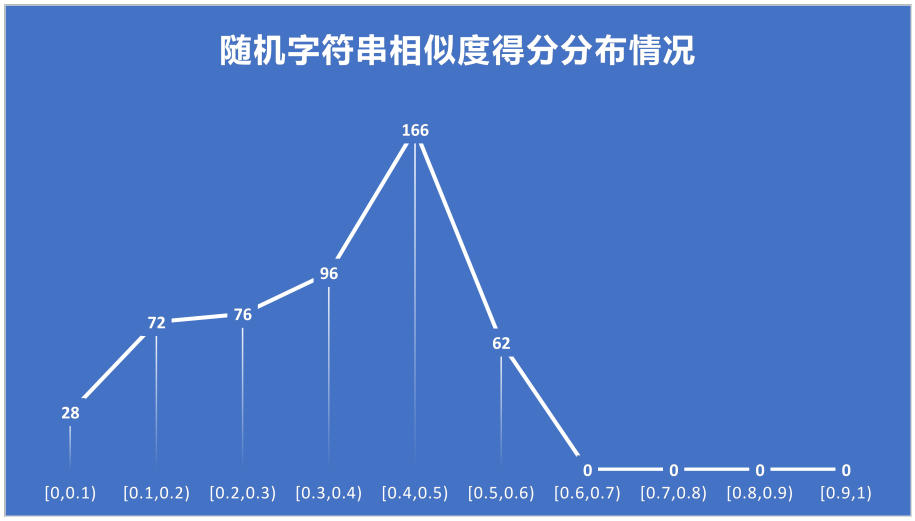


图 4–1 随机生成的字符串的相似度得分分布情况

结果的分布情况如图4–1所示。根据图片可以看出，完全随机的两个字符串的相似度得分超过 0.6 的概率很小，因此选择 $T_c = 0.6$，以将尽可能多的库识别出来。

4.2.3 TPL-V Detector 性能评估

(1) 粗粒度-库匹配

实验所选的六个库的各版本数之和为 92，即数据库中包含了 92 个独立的库。根据两级特征匹配策略，首先使用粗粒度特征来识别混淆的六个库分别对应与哪一个标准库。将混淆库从 1 至 6 依次编号，对于一个混淆库（例如混淆库 1）与未混淆库（例如 com.squareup.okio），首先将其下的类进行两两匹配，计算每一个配对的相似度得分。包含方法数多的类的匹配难度更高，对与不包含方法的类，其特征值是空字符串生成的哈希值，可以与任意另一个方法数为 0 的类匹配。即便在最初特征生成的过程中，TPL-V Detector 将文件大小不超过 1KB 的字节码文件删去，但上面的极端情况仍然可以表明方法数不同的类的匹配难度不同，因此引入方法数占总方法数的比例作为该类的权重，即相似度得分前的系数项。

具体算法如4–1所示，对每一个混淆库，计算其与 6 个未混淆库的得分，选择得分最高者作为它对应的标准库，实验结果表4–3，加粗的单元格表示所在行列对应的混淆库与未混淆库匹配。

根据表4–3，混淆库从 1 至 6 分别对应于 ezmorph、okio、fastjson、happy-bubble、opensdk、okhttp，与实际上混淆的对应关系相同，混淆的第三发放标准库被全部检测出来。

(2) 细粒度-库匹配

在确定了各混淆库对应的标准库后，需要使用基于字节码的细粒度特征确定具体版本。实验算法基本与算法4–1相同，对于两个库，计算各个类的配对的相似度，以方法数为权重，计算两个库的总得分。对每个混淆库的得分列表，选取得分最高的未混淆库作为它的匹配结果。

各库版本匹配结果如图4–2所示，对每个库列出了排名前三的版本得分，由于 opensdk 和 happy-bubble 只包含两个版本，因此列出了前两名的版本得分。

算法 4–1 粗粒度匹配实验算法

Data: 未混淆的标准库集合 L，混淆后的标准库 L^{obfus}
Result: 混淆库与未混淆库映射关系 $obfus\_from$

Initialize obfuscated libraries C_{lib}^{obfus};
Initialize score list $score\_list \leftarrow empty\ list$;
for $l^{obfus} \in L^{obfus}$ **do**
  Initialize $score\_list[l^{obfus}] \leftarrow empty\ dictionary$;
  for $l \in L$ **do**
    $score(l^{obfus}, l) \leftarrow 0$;
    for $c^{obfus} \in C_{l^{obfus}}$ **do**
      for $c \in C_l$ **do**
        if c^{obfus} *match* c **then**
          $score \leftarrow score + similarity(c, c^{obfus}) \times \frac{|c.get\_methods()|}{|l.get\_methods()|}$;
        else
          continue;
        end
      end
    end
    $score\_list[l^{obfus}] \leftarrow score\_list[l^{obfus}] \cup \{< l, score >\}$;
  end
  $obfus\_from[l^{obfus}] \leftarrow l, max\{score\}\ for\ < l, score > \in score\_list[l^{obfus}]$;
end

表 4–3 标准库混淆前后在粗粒度特征上的匹配结果

	混淆库 1	混淆库 2	混淆库 3	混淆库 4	混淆库 5	混淆库 6
ezmorph	**0.930**	0.493	0.465	0.518	0.494	0.462
fastjson	0.377	0.446	**0.852**	0.397	0.446	0.463
happy-bubble	0.529	0.486	0.486	**0.929**	0.500	0.486
okhttp	0.388	0.444	0.442	0.379	0.426	**0.808**
okio	0.400	**0.842**	0.527	0.420	0.439	0.473
opensdk	0.414	0.456	0.491	0.435	**0.842**	0.455

为表示方便省略标准库的 groupID。
粗体数据表示该数据为所在行/列中的最大值。

4.3 混淆安卓 App 第三方库检测实验

4.3.1 Android 数据集

Li[23]等人分析了大量的手机应用及其所使用的第三方库，本文从此数据集中选取了 10 款 App 作为实验的 App 数据集，并从这些 App 所包含的库中选择 9 个作为标准库数据集，各 App 编号及详情见附录 A，包含情况如表4–4所示。

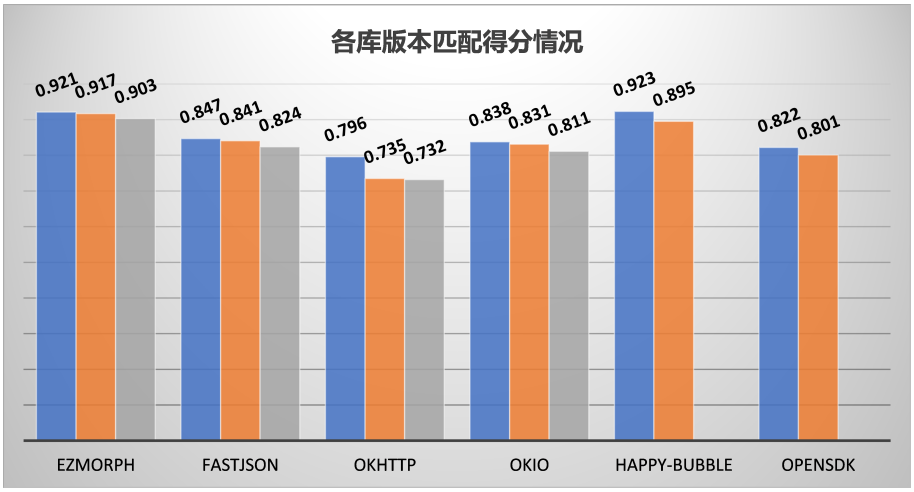


图 4–2 细粒度版本匹配各库的版本得分情况

表 4–4 App 对各标准库的包含情况

App / Lib	1	2	3	4	5	6	7	8	9	10
Google Ads	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
Apache Http	✓	✗	✓	✗	✗	✓	✓	✗	✗	✗
Firebasee	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗
Square Dagger	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗
ChartBoost	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗
JavaX Annotation	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
OKHttp3.0	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Tencent SDKs	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
ksoap2	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

✓表示 app 包含库，✗表示 app 不包含库。

4.3.2 TPL-V Detector 性能评估

(1) 在评价指标上的表现

对于表4–4中的标准库，将每个库的每个版本作为一个独立的库，共得到 529 个标准第三方库作为先验知识。对于十款待测 App，如果 App 的某个目录满足以下两个条件：

1. 该目录下包含 class 文件
2. 该目录的上级目录不包含 class 文件

则将该目录作为一个根包的目录。这样做的依据是：一个 App 所引入的第三方库的根包含有子包（文件夹）、根层调用子包的逻辑以及自身的部分代码实现，根包的上层目录作为组 Id 不包含实现代码，仅为了表明包的所属机构以及创造隔离的命名空间。

因此对于 App 中每个满足上述条件的目录，都以此目录为包的根目录构建特征树，生成特征。由于 App 经过混淆处理，可能包括代码优化、资源压缩、死代码删除等步骤，因此为防止因为细微差别导致结果出现错判，设置阈值得分 T，对于一对分别来自标准库和 App 的包，如果相似度得分阈值超过 T，则认为该 App 使用了此标准库。

对阈值从 0.6 到 0.9，以 0.05 为间隔，刻画准确率与召回率变化曲线，来选择最佳的阈值。如图4–3所示，在 0.6 至 0.8 区间，随着阈值升高，匹配标准更加严格，准确率有所上升，但是出现某些 App 内的混淆库找不到对应的标准库的情况，因此召回率下降严重。当阈值跌过 0.7 时，准确率发生快速下降情况，导致这一情况产生的原因可能是阈值接近了两个不相关库就能达到的得分，大量版本错误的库甚至是不同的库进入到结果列表当中。为了在保持召回率不过低的情况下，尽量提高准确率，本文选择阈值 $T = 0.8$。

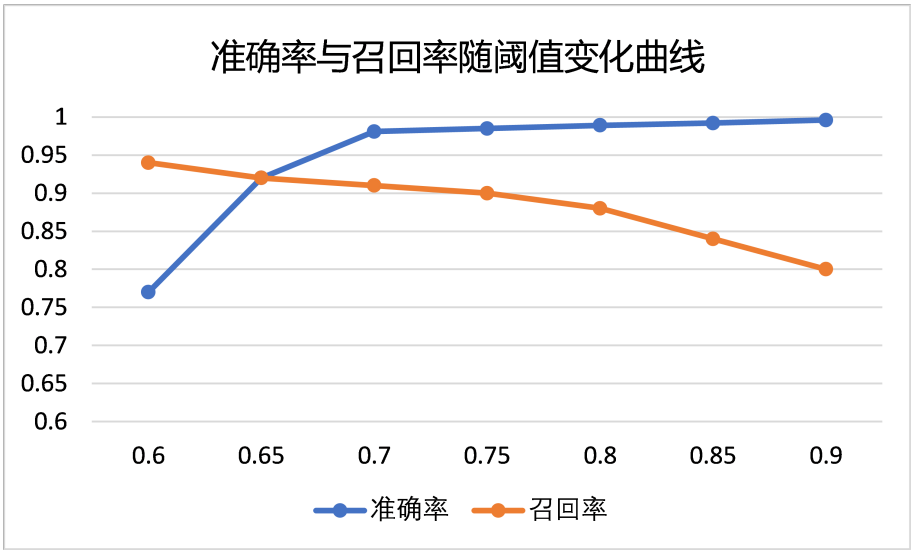


图 4–3 准确率与召回率随阈值变化曲线

阈值选定后，进行 App 与标准库的匹配。对每一个 App 中划分出的每一个包，先采用粗粒度特征匹配算法4–1选出该包所对应的库，再采用细粒度特征匹配算法识别出库的具体版本。由于阈值的设定，每个 App 的包可能在标准库数据库中有多个匹配结果，实验将这些结果都加入到匹配列表中，表示这些库是在现有知识下得出的结论。为了能够有效计算准确度，定义有效样本数为列表长度的倒数，因此若列表中只包含一个库，即系统选出了唯一的最佳匹配结果，那么此样本在评估指标计算的真阳性（TP）样本数中占 1，否则列表越长，表示此次匹配效果越差，即不确定性高，相应的真阳性样本数占据小于 1 的一个分数。

匹配结果如表4–5所示。

表 4–5 App 对各标准库的包含情况

App / Lib	1	2	3	4	5	6	7	8	9	10
Google Ads	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
Apache Http	✓	✗	✓	✗	✗	✓	✓	✗	✗	✗
Firebasee	✓	✓	✓	✗	✗*	✗	✗	✗	✗	✗
Square Dagger	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗
ChartBoost	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗
JavaX Annotation	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
OKHttp3.0	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Tencent SDKs	✗	✗	✗	✗*	✗	✗	✓*	✗	✗	✗
ksoap2	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

* 表示检测结果错误。

系统各评估指标结果如表4–6所示，TPL-V Detector 在准确率上表现较好，误报率很低，但是没有将 App 中使用到的库全部召回。查看各库的版本情况发现，识别出错较多的库——腾讯 SDK（tencentcloud-sdk-java-common）包含了 276 个版本，仅版本 3.1.x 的子版本号就超过了 100 个，本身各版本之间的差异可能不大，而混淆后的 App 中经过了代码优化、压缩等过程，导致部分未用到的代码被删去，有一定可能损失包含关键区别的部分特征，最后使得系统在此库上表现不佳。

表 4–6 TPL-V Detector 各项指标评估结果

指标	评估结果
Accuracy	98.9%
Precision	88.0%
Recall	88.0%
F1_score	88.0%
FPR	0.6%

(2) 时间开销实验

为了细致评估 TPL-V Detector 的时间开销，实验将从取得 App 到完成库匹配的整个流程分成了如图4–5所示的四个部分，分别为：

1. App 预处理
2. 字节码特征提取
3. 特征树构建
4. 特征存储与匹配

其中 App 预处理又包含了 Apk 的解压、Dex 文件的反编译、Class 文件到 Bytecode 文件的转换，以及小于 1KB 文件的过滤。字节码特征提取包含了基于方法描述符的粗粒度特征的提

取和基于字节码指令序列的细粒度特征的提取。特征树的构建自下而上从方法层到类层构建特征。特征的存储与匹配分为特征向 MySQL 的存储，以及特征的粗粒度/细粒度两级特征匹配。

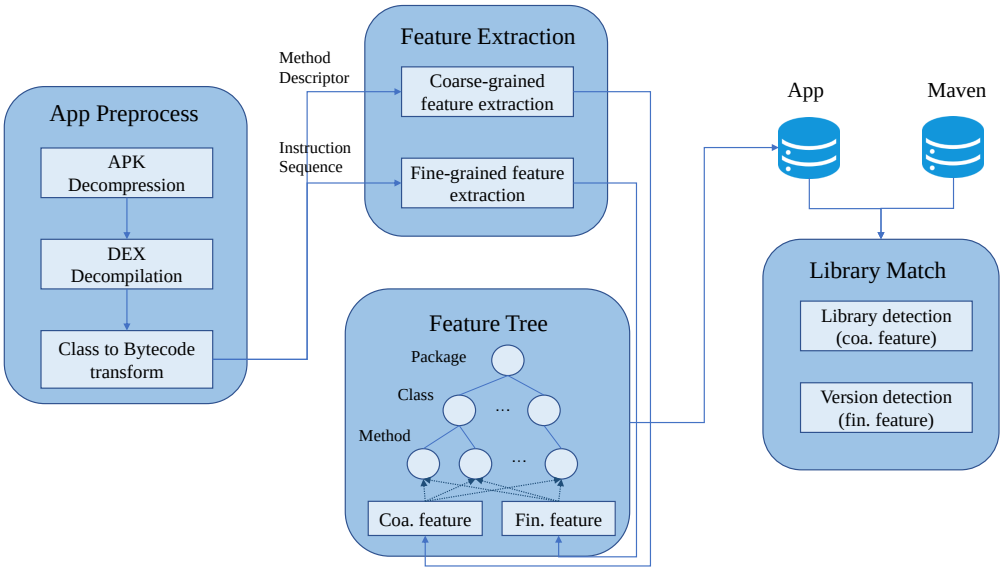


图 4–4 时间开销测试 App 库匹配的四阶段工作流程

对于实验所用十款 App，按照文件大小排序，分别测试其各阶段所用时间，测试结果如图**??**所示。对于一个大小为 15MB 的 App，执行完四个阶段所需耗时仅仅约半分钟。最大的 App 大小为 258MB，处理时间开销已经接近了十分钟。在四个阶段中，阶段一耗时占比最大，基本都超过了总耗时的一半。经过对阶段一进一步分析，我认为可能是调用第三方工具所导致。对 App 的预处理首先需要调用 unzip 解压工具，获取 DEX 文件；对 DEX 文件，需要借助 github 开源工具 dex2jar 来获取 class 文件；最后 class 文件转换成字节码的过程中需要频繁的使用 javap 程序，累积产生了较大的时间开销。阶段二特征提取耗时较少，主要原因可能是此阶段只需要进行文本读取操作，并且为了过滤掉不包含方法的类，TPL-V Detector 还会删除小于 1KB 的字节码文件，进一步节省了时间。阶段三是耗时最少的步骤，此部分由 C++ 代码实现，数据结构较为繁杂，但是处理流程高效，将原始的粗细特征处理成方法的粗细特征，并将包/类/方法组织成树形结构，生成类的特征即可。第四阶段耗时占比在 15% 至 30% 之间，此部分耗时实际上依赖于先验知识数据库的大小，数据库越大，则匹配耗时越长；数据库小，则匹配的时间开销就很少。

Apk 文件越大，包含的 DEX 文件就大，相应的 class 文件、字节码文件、产生的特征就更多，作为根包生成特征树的目录也更多，匹配耗时越长。反之亦然，App 规模小，各阶段耗时也有所下降。根据图4–6，四个阶段的时间开销与 App 大小基本上呈现出线性关系。

4.3.3 与现有工具的对比实验

(1) 检测准确率对比实验

(2) 检测耗时对比实验

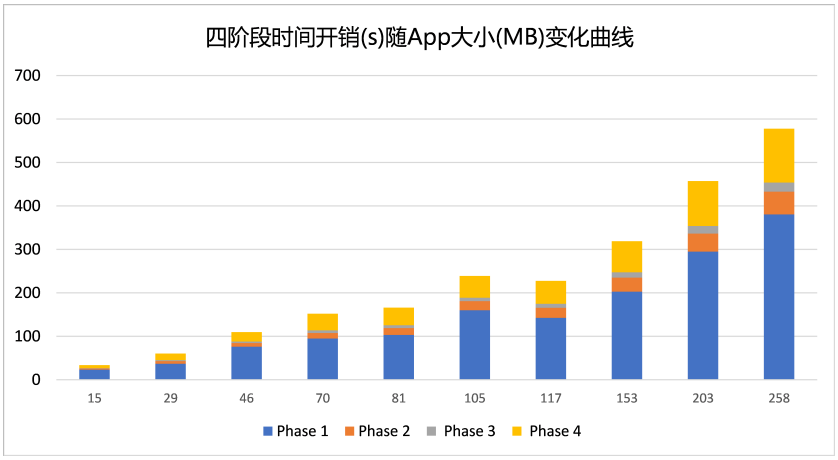
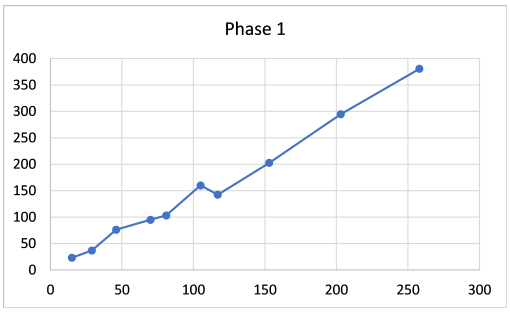
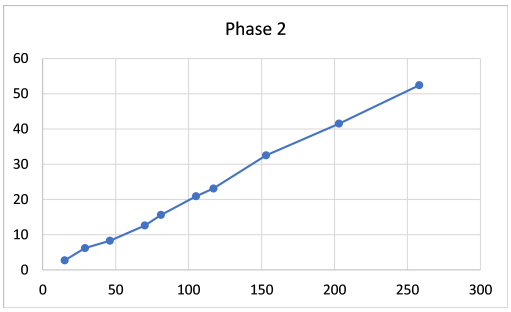


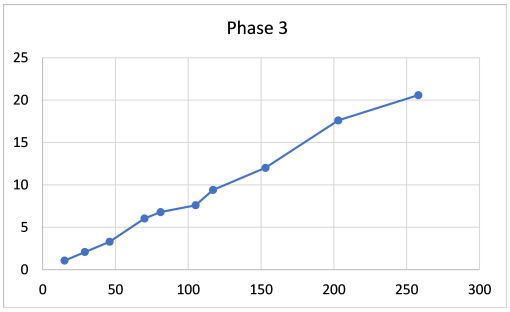
图 4–5 各阶段时间开销随 App 大小变化曲线



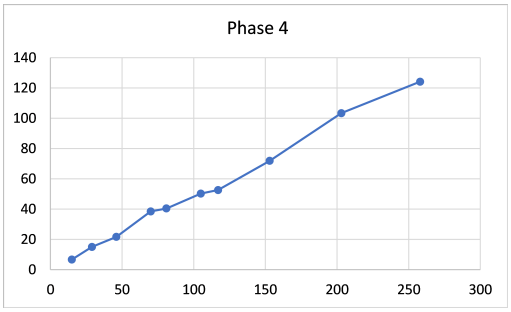
a) App 预处理阶段时间开销



b) 特征提取阶段时间开销



c) 特征树构建阶段时间开销



d) 特征存储与匹配阶段时间开销

图 4–6 各阶段时间开销 (s) 随 App 大小 (MB) 变化曲线

参考文献

[1] 2021 中国软件供应链安全分析报告[Z]. https://www.qianxin.com/news/detail?news_id=1108. Accessed May 9, 2022.

[2] LAB G S. The Octopus Scanner Malware: Attacking the open source supply chain[Z]. https://securitylab.github.com/research/octopus-scanner-malware-open-source-supply-chain/. Accessed May 9, 2022.

[3] AppBrain. Android library statistics[Z]. https://www.appbrain.com/stats/libraries. Accessed May 9, 2022.

[4] Thehackernews.com. Backdoor in Baidu Android SDK Puts 100 Million Devices at Risk[Z]. https://thehackernews.com/2015/11/android-malware-backdoor.html. Accessed May 9, 2022.

[5] DONG S, LI M, DIAO W, et al. Understanding android obfuscation techniques: A large-scale investigation in the wild[C]//International conference on security and privacy in communication systems. 2018: 172-192.

[6] Java Obfuscator and Android App Optimizer[Z]. https://www.guardsquare.com/proguard. Accessed May 12, 2022.

[7] Dalvik Bytecode Obfuscation on Android[Z]. http://www.dexlabs.org/blog/bytecode-obfuscation.

[8] LÁSZLÓ T, KISS Á. Obfuscating C++ programs via control flow flattening[J]. Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica, 2009, 30(1): 3-19.

[9] BOOK T, PRIDGEN A, WALLACH D S. Longitudinal analysis of android ad library permissions[J]. arXiv preprint arXiv:1303.0857, 2013.

[10] LIU B, LIU B, JIN H, et al. Efficient privilege de-escalation for ad libraries in mobile apps[C]//Proceedings of the 13th annual international conference on mobile systems, applications, and services. 2015: 89-103.

[11] CHEN K, WANG X, CHEN Y, et al. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios[C]//2016 IEEE Symposium on Security and Privacy (SP). 2016: 357-376.

[12] LI M, WANG P, WANG W, et al. Large-scale third-party library detection in Android markets [J]. IEEE Transactions on Software Engineering, 2018, 46(9): 981-1003.

[13] MA Z, WANG H, GUO Y, et al. Libradar: fast and accurate detection of third-party libraries in android apps[C]//Proceedings of the 38th international conference on software engineering companion. 2016: 653-656.

[14] WANG H, GUO Y, MA Z, et al. Wukong: A scalable and accurate two-phase approach to android app clone detection[C]//Proceedings of the 2015 International Symposium on Software Testing and Analysis. 2015: 71-82.

[15] CRUSSELL J, GIBLER C, CHEN H. Andarwin: Scalable detection of android application clones based on semantics[J]. IEEE Transactions on Mobile Computing, 2014, 14(10): 2007-

2019.

[16] XU J, YUAN Q. Libroad: Rapid, online, and accurate detection of tpls on android[J]. IEEE Transactions on Mobile Computing, 2020, 21(1): 167-180.

[17] BACKES M, BUGIEL S, DERR E. Reliable third-party library detection in android and its security applications[C]//Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 2016: 356-367.

[18] NARAYANAN A, CHEN L, CHAN C K. Addetect: Automated detection of android ad libraries using semantic analysis[C]//2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP). 2014: 1-6.

[19] LI M, WANG W, WANG P, et al. Libd: Scalable and precise third-party library detection in android markets[C]//2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). 2017: 335-346.

[20] https://github.com/wangzifan184/dex2jar. Accessed May 9, 2022.

[21] https://repo.maven.apache.org/. Accessed May 9, 2022.

[22] ZHAN X, FAN L, CHEN S, et al. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications[C]//2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). 2021: 1695-1707.

[23] LI B, HE Q, CHEN F, et al. Embedding app-library graph for neural third party library recommendation[C]//Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021: 466-477.