

2021 届硕士学位论文

分类号: _____ 学校代码: 10269

密 级: _____ 学 号: **** * * * *



华东师范大学

East China Normal University

硕士 学位 论文
MASTER'S DISSERTATION

论文题目:

基于 Android 仿冒应用的大规模实证研究

院 系: 计算机科学与技术学院

专业名称: *****

研究方向: *****

指导教师: *** *

学位申请人: ***

2021 年 12 月

Thesis for master's degree in 2021 University Code: 10269
 Student ID: ****

EAST CHINA NORMAL UNIVERSITY

A LARGE-SCALE EMPIRICAL STUDY ON ANDROID FAKE APPS

Department: School of Computer Science and Technology

Major: ****

Research Direction: ****

Supervisor: ***

Candidate: ***

January, 2021

*** 硕士学位论文答辩委员会成员名单

姓名	职称	单位	备注
***	***	*****	主席
***	***	*****	
***	***	*****	

摘要

Android 移动操作系统由于其开源、用户广泛等特点，拥有庞大的应用生态环境，其中包含各种良性应用，以及恶意应用、重打包应用和仿冒应用。得益于广泛的前人研究，业界对各类恶意、重打包应用具备充分理解，衍生出了对应检测机制与防护措施；然而，近年来 Android 安全相关研究多聚焦于恶意、重打包应用，学术界对仿冒应用进行的研究较为匮乏，Android 移动应用安全仍有隐患。

目前，针对仿冒应用的研究面临如下三点问题与挑战：(1) 业界尚缺乏一个公开可用的大规模仿冒应用数据集供研究者使用。其难点在于：收集数据时需要同时考虑数据规模、来源多样性与代表性；(2) 业界对仿冒应用的概念较为模糊，仿冒应用的基本特征尚未明晰，进而阻碍后续的检测工作；(3) 仿冒应用作者的行为特征尚不明确，无法为业界预防或拦截仿冒应用提供有效指导。

为获得关于仿冒应用的原始资料，本文采取实证研究的方式，从以下四个方面对仿冒应用进行了深入的分析和研究：

通过收集大规模仿冒应用样本集，改善仿冒应用数据稀缺问题。本研究从应用宝、豌豆荚等多个应用来源入手爬取数据，最终筛选出 5 万个余个仿冒应用，可为后续工作如应用特征提取、应用行为分析等提供数据支持。

针对 Android 仿冒应用特征进行多方面分析研究。利用上述数据集，本文对仿冒应用进行了与原版应用的相似度比对、功能比对等多方面解析，提供了仿冒应用的基本特征解读结果，并提供案例分析。分析结果可为普通用户与应用市场等群体抵御仿冒应用威胁提供指导。

利用证书信息与应用发布时间提供应用开发者行为画像。结合仿冒样本发布时间和从仿冒样本中抽取出的应用证书信息，本文从活跃期、投放特征等角度对仿冒应用开发者开展了一系列行为画像，该画像反映的行为特征可为应用市场提

供监管思路。

提出了可用的仿冒应用排查框架。实证研究表明，国内应用市场尚未有应对仿冒应用的完善措施。针对该现状，本文推出了检测框架 FakeRevealer，帮助应用市场在大规模应用中实现对已知正版应用及其对应仿冒样本的快速鉴别。

综上，本研究收集了较大规模数据集，可为后续工作提供数据支持和探索方向。同时，依托收集到的大量数据，本研究对仿冒应用进行了全面分析，对普通用户和应用市场均提出了对应的实用建议，并提出了可行的检测方案，有助于提升各方对仿冒应用的认识与重视。

关键词: Android 应用程序, 仿冒应用, 实证研究, 数据分析, 排名欺诈

目 录

摘要	i
目录	iii
第一章 绪论	1
1.1 研究背景	1
1.2 研究现状	2
1.2.1 重打包应用的研究现状	3
1.2.2 恶意应用的研究现状	4
1.2.3 其他相关研究	5
1.2.4 研究现状小结	6
1.3 拟采用的研究方法	6
1.4 本文拟解决的关键问题	7
1.5 本文主要工作	8
1.6 本文组织结构	8
第二章 相关技术背景介绍	10
2.1 实证研究相关背景介绍	10
2.1.1 软件工程中的实证研究	10
2.1.2 实证研究方法论	11
2.1.3 有效性验证标准	12
2.2 Android 相关背景介绍	13
2.2.1 Android 应用证书机制	13
2.2.2 重打包技术	17

2.3	本章小结	17
第三章	实证研究设置	18
3.1	研究问题	18
3.2	研究对象	19
3.3	数据收集	19
3.3.1	数据类型与收集方式	20
3.3.2	数据概览	22
3.3.3	相关数据集对比	23
3.4	本章小结	24
第四章	仿冒应用基本特征实证分析	25
4.1	研究方法	25
4.1.1	相似程度度量指标	25
4.1.2	应用受仿冒严重程度相关度量指标	26
4.1.3	关联程度度量指标	27
4.1.4	实验对象与实验数据	28
4.2	实证研究流程与结果解析	28
4.2.1	仿冒样本与原版应用的相似度	28
4.2.2	影响应用被仿冒的严重程度的因素	32
4.2.3	仿冒应用功能与行为	37
4.2.4	研究结果有效性分析	39
4.3	相关工作	39
4.4	本章小结与实用建议	40
第五章	仿冒开发者行为特征实证分析	41
5.1	研究方法	41
5.2	实证研究流程与结果解析	42
5.2.1	应用证书与仿冒应用对应关系	43
5.2.2	仿冒应用证书活跃度	45
5.2.3	仿冒应用上架特征分析	46

5.2.4 研究结果有效性分析	48
5.3 相关工作	49
5.4 本章小结与实用建议	50
第六章 仿冒应用检测框架 FakeRevealer	52
6.1 框架设计与实现	52
6.1.1 整体设计	52
6.1.2 过滤模块	53
6.1.3 鉴别模块模块	55
6.1.4 代码分析模块	56
6.1.5 人工审查	59
6.2 系统实验	60
6.2.1 数据收集	60
6.2.2 实验与结果	61
6.3 本章小结	64
第七章 总结与展望	65
7.1 总结	65
7.2 展望	65
参考文献	67
攻读学位期间发表的学术论文	75
致谢	76

插 图

图 1.1 Google Play 应用商店架上应用总数变化趋势	1
图 2.1 数字签名技术的两个步骤	16
图 3.1 Janus 平台上的数据规模时序图	21
图 3.2 搜索框架整体流程	22
图 4.1 应用的应用名、包名与应用大小的相似性比较	29
图 4.2 各大应用市场应用详情页	31
图 4.3 从不同应用市场中收集到的应用数量以及各市场仿冒率	33
图 4.4 目标应用类别-仿冒率对应图	36
图 4.5 游戏类 App 及其仿冒样本	38
图 5.1 仿冒应用应用证书活跃时间分布	45
图 5.2 每月仿冒样本数量、应用证书数量（2015 年 11 月至 2018 年 9 月）	47
图 5.3 仿冒延迟总体分布	48
图 6.1 FakeRevealer 整体流程	52
图 6.2 Janus 漏洞原理	55
图 6.3 正版与仿冒版 WiFi 万能钥匙图标	62

表 格

表 3.1 目标应用类别与各分类下应用	20
表 4.1 15 款应用于各大市场的仿冒率（数值单位：%）	33
表 4.2 目标应用与其相关统计	34
表 5.1 应用证书/仿冒应用数量对应表	43
表 5.2 应用证书数量/开发者名称数量对应表	44
表 5.3 开发者名称/仿冒样本数量对应表（关联数前十位）	44
表 5.4 仿冒应用证书多市场上架情况统计	47
表 6.1 FakeRevealer 实验样本来源	60
表 6.2 有效性实验结果	61
表 6.3 对比实验结果	63

第一章 绪论

1.1 研究背景

随着移动市场于近年逐渐兴起，Android 系统作为一个主流的移动端操作系统也在蓬勃发展。数据分析机构 StatCounter 资料显示，Android 市场占有率自发布之日起就在逐年稳步增长。截至 2020 年，Android 系统已经占据全球移动端市场份额的 74.3%^[1]。与此同时，Android 应用¹的数量也伴随着 Android 市场的蓬勃发展节节攀高。仅 Android 官方的应用商店 Google Play 就在 2017 年中新上架了近一百万个可供下载的应用程序。虽然因为各种原因，Google Play 上的应用数量在 2018 年有所回落，但如图 1.1 所示，应用市场上目前仍有近三百万个可用的应用程序，Android 应用市场依然充满活力^[2]。

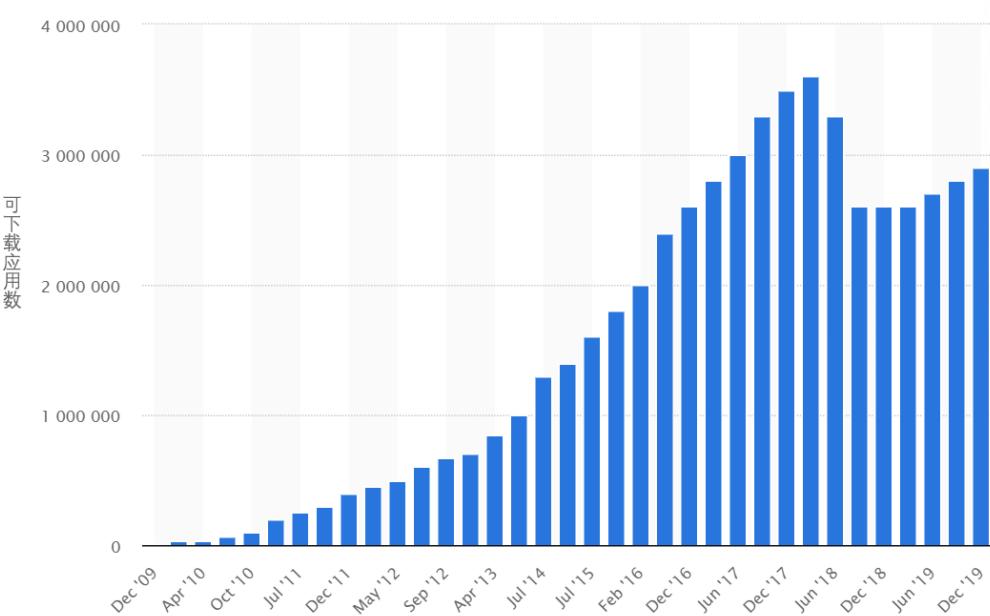


图 1.1: Google Play 应用商店架上应用总数变化趋势

¹应用、App 在本文中均用以指代 Android 应用程序，不再区分

随着 Android 移动应用行业快速发展，移动黑灰产¹也逐渐活跃。黑灰产是以侵害用户、原应用作者或其他第三方利益为手段，凭此或通过其他可疑方式牟利的产业。一方面，随着开发移动应用的门槛逐渐下降，开发一个移动应用的成本已经普遍低于开发一个类似的桌面级应用所需的成本^[3]；另一方面，移动应用功能在实现上的灵活性^[4]增加了移动应用的复杂度，针对黑灰产的分析与拦截面临着多重新挑战。两方面结合，为黑灰色应用进入移动端发展提供了良好的基础。

仿冒应用就是一类广泛存在的移动灰黑产产物。本文研究的仿冒应用指代开发者根据市面上热门应用的外观仿制的 Android 应用。据日常观察，仿冒应用中常有恶意广告弹窗、信息窃取、恶意扣费等行为，开发者可在用户使用仿冒应用时利用广告流量或非法获得的用户信息获利。一方面，仿冒应用损害了原版应用开发者的知识产权，侵害了原开发者的利益；另一方面，仿冒应用中的恶意行为也在威胁普通用户的信息安全。可以联想如下场景：用户尝试通过应用市场搜索安装一个应用，应用市场返回了多个无论是名字或是图标都十分相像的结果，在未有明确指引的情况下，用户安装了仿冒应用。前人针对 Android 恶意应用的研究^[5]显示，恶意行为可被自动触发，若用户下载的仿冒应用中包含此类恶意行为，用户数据安全将受到损害；同时，由于用户不能分辨原版应用与仿冒应用，后者的不良用户体验会导致用户对前者乃至应用市场产生不良评价；即使用户删除仿冒应用、装上原版应用，也浪费了时间和人力成本。

综上所述，仿冒应用影响用户搜索与下载应用时的安全和体验，最终使用户、原版应用开发者和应用市场三方利益均受到损害。为了消除仿冒应用带来的隐患，无论是开发者、应用市场或是普通用户，都应对仿冒应用的特征与行为有所了解。

1.2 研究现状

尽管客观上存在对仿冒应用进行了解的需求，本研究对移动应用领域的前期调研显示，针对仿冒应用进行的实证研究尚缺乏，也未有公开可用的仿冒应用数据集。因此，本节讨论与仿冒应用同属于移动灰黑产的重打包应用和恶意应用、以及其他移动应用领域相关方向的研究现状。

¹即移动端黑色产业与灰色产业，下文同

1.2.1 重打包应用的研究现状

重打包指恶意开发者对原应用解包、篡改内容之后再将应用重新打包的技术^[6]。重打包应用侵害了应用原作者的知识产权，甚至可能具有恶意行为，因此也属于移动黑灰产范畴。目前针对重打包应用的相关工作分为三类，分别为关于重打包的实证研究、针对重打包应用的检测和防止应用被重打包的研究。

相关实证研究方面，Khanmohammadi 等人对重打包应用进行了一次详尽的实证研究^[6]。该实证研究利用已有数据集 AndroZoo^[7]，得出了“重打包应用的主要目的为利用广告获得收入”的结论，并回答了包括“何种类型的应用会被重打包”、“重打包对原应用属性更改如何”在内的五个研究问题，增进了研究者对重打包应用的理解。

在重打包应用检测方面，早期工作通过比对两两应用之间的信息获得应用之间的相似度。有研究者基于应用指令序列的方法使用模糊哈希的方法提取出应用的摘要信息，如 DroidMOSS 和 DroidAnalytics^[8,9]；也有使用静态数据依赖构建程序依赖图 (Program Dependency Graph, PDG) 比对应用的方法，如 DNADroid^[10] 和 AnDar-win^[11]，Centroid^[12] 甚至为应用中的每个函数构建了三维控制流图 (3-Dimensional Control Flow Graph, 3D-CFG)，将三维控制流图聚合后，通过检测不同应用在控制图聚合后的质心位置判断应用间的相似程度；还有算法通过比对抽象语义树检测重打包应用，如 Hanna 等人的 Juxtap^[13] 利用 K-gram 模型对应用的二进制操作码序列进行比对，CLANdroid^[14] 通过分析五种语义特征点（比如代码中的标识符和调用到的 Android API 等），以检测相似应用。上述算法一来因为需要使用静态分析方法对代码进行解析，在遇到使用过代码混淆处理（如反射和代码加密）的应用时性能往往较差；二来需要基于两两比对进行检测，复杂度较高，在应用规模增大时会导致庞大的计算量。为处理静态分析带来的问题，有学者提出了利用动态分析比对应用的方法，如 Wu 等人使用 HTTP 流量对应用行为进行建模^[15]。亦有研究使用信息可视化方法检测重打包应用：ViewDroid^[16] 通过重建和比对应用的视图筛选出重打包应用，DroidEagle^[17] 则利用 UI 布局相似性检测，Kywe 等人比对应用中的文本与图像相似性以寻找相似应用^[18]，Soh 等人分析应用运行时收集的 UI 信息以检测重打包行为^[19]。而从避免两两比对、减小算法复杂度方面考虑，有学者提出

利用第三方库进行检测的办法，如 CodeMatch^[20] 筛选出应用中使用的第三方库代码后，计算并比对剩余部分代码的哈希值，获取不同应用的相似程度；Wukong^[21] 也分两步检测重打包应用，但与 CodeMatch 相比，其第二步使用了基于计数的代码克隆检测手段，而非基于哈希的技术。

防止应用被重打包的研究多基于检测原有代码是否被更改而开展。Zhou 等人提出了两个基于改版 Dalvik 虚拟机的对抗工具：其中，AppInk^[22] 利用一种“水印”技术（Watermarking）对抗重打包，但需要利用一个改版的 Dalvik 虚拟机进行文件检查；另一工具 DIVILAR^[23] 为应用的 Dalvik 代码提出了一种新编码形式，同时也需要修改用户设备 Android 系统中的 Dalvik 虚拟机以对新的编码进行解码。Luo 等人提出了在代码中注入大量检测节点的方法对抗重打包，并推荐开发者进行代码混淆，以避免令重打包开发者发现注入的检测节点^[24]。与之相似，Zeng 等人也提出了在代码中注入逻辑炸弹的方法，并在研究中讨论了多种避免重打包开发者发现代码注入的策略^[25]。

1.2.2 恶意应用的研究现状

针对恶意应用的研究可分为两个方面，分别是面向恶意应用进行实证研究工作与面向恶意应用检测的研究。

面向恶意应用的实证研究众多，较有代表性的有以下两篇文献：Felt 等人的研究^[26] 仔细剖析了来自多个不同平台的 46 个恶意程序样本以了解这些样本的激励机制，揭示了这些样本的运行机制和行为策略，为后人抵御此类恶意行为提供参考；Zhou 和 Jiang^[5] 搜集了来自多个主要恶意应用家族的、超过 1,200 个恶意应用样本，并系统性地描绘了这批样本的不同性质，包括其安装手段、激活机制和其如何执行有效负载（实现恶意行为）。虽然恶意应用与仿冒应用有重合之处，但两者有一定区别，相关理论知识直接套用并不合适。

以上述实证研究与数据分析作为基础，研究者对恶意应用有了较好的了解，从而得以有针对性地进行恶意应用检测。相关方法可大致分为静态分析、动态分析与机器学习三个方向。早期研究通过静态分析手段对恶意应用进行检测，包括 Enck 等人提出的 Kirin，在应用安装时检测应用权限，从而拦截恶意应用的安装^[27]；Felt 等人

提出的 Stowaway^[28] 利用 API 调用分析和权限分析, 寻找权限过大(Overprivileged)的应用; RiskRanker^[29] 则提出了一个两阶恶意应用检测手段, 先筛选出具有 root 行为和可能导致隐私泄露的应用, 再找出其中具有危险 Dalvik 代码加载行为的应用。然而, 受静态分析方法本身的限制, 此类研究在遇到代码混淆时多被影响性能。因此, 有研究者利用动态分析技术对恶意应用进行检测。TaintDroid^[30] 和 DroidScope^[31] 均在沙箱中对应用行为进行监视, TaintDroid 利用污点分析技术寻找应用的信息泄露行为, DroidScope 则从 Android 系统的不同层面分析应用的可疑行为。Zhou 等人^[32] 提出了 DroidRanger, 利用应用行为比对与动态加载分析的方式, 成功地从 5 个应用市场的 204,040 个应用中找出了 211 个恶意应用。ParanoidAndroid^[33] 提出了将恶意检测方法迁移到云端的概念, 将设备上的所有操作同步到云端服务器, 从而在不影响设备本地功能的情况下进行恶意行为检测。无论是静态分析方法或是动态分析方法, 都需要人工对恶意行为的模式进行描绘, 而用人工手段提取行为模式, 不仅对领域知识有较高要求, 还往往是费力费时的。针对此问题, 研究者开始利用机器学习技术检测恶意应用。Chen 等人从大规模数据集中总结出了恶意应用的两类特征, 再将其用于分类器模型训练, 提出了检测工具 StormDroid^[34], 与之类似的方法还有 CrowDroid^[35], DroidMat^[36], Adagio^[37], MAST^[38], DroidAPIMiner^[39] 与 Drebin^[40]。

1.2.3 其他相关研究

Andow 等人针对灰色应用的研究^[41] 从 Google Play 应用商店中采集了多个应用样本。该研究将样本分类, 定义了高仿应用(Imposter)、移动广告应用(Madware)等 9 种不同的灰色应用。灰色应用指不具有明显恶意行为, 但应用意图存疑、又或是会向系统申请过多权限的应用程序。因此, 这类应用不是恶意应用, 但由于其盈利方式可疑, 可将其归入移动黑灰产内。

另一方面, Hu 等人针对市面上的欺诈约会应用进行了一次实证研究^[42]。该研究中的欺诈约会应用声称自己为约会应用, 实则诱导用户开通会员服务, 也应归入黑灰产应用范畴。分析发现, 用户在此类应用中遇到的其他“用户”多数不由真人控制, 而是具有预设对话模板的聊天机器人。此外, 研究还分析了该类应用的商业

模式与分发渠道，评估了受害者的数量，对此类应用的生态进行了详尽解读。该研究还通过直接检测相同评论和标记可疑用户的方式对此类应用进行排名欺诈检测。

Chen 等人发布了漏洞检测工具 Ausera^[43]，该工具通过静态分析技术与敏感词识别，对手机银行应用进行自动化三段式风险评估。同期，他们在实证研究中利用 Ausera 从 693 个手机银行应用中检出超过两千个漏洞，向对应的 60 家银行发送了漏洞报告，并检查了新发行的手机银行应用以跟进漏洞修补进程^[44]。在被检出漏洞的 60 家银行中，21 家确认了该研究反馈的漏洞，进行了对应修复，作者随后根据修复状况进行案例研究，分别面向银行、学者与政府，共给出了 9 条用于改善手机银行应用安全性与市场环境的实用建议。与之类似，本研究也从多个方面出发，提出了防范仿冒应用、净化市场环境的实用建议。

1.2.4 研究现状小结

综合上述分析，与仿冒应用同属黑灰产的重打包应用和恶意应用，甚至是日常生活中常用的手机银行类应用，均有对应研究可提供参考。前人的实证研究收集了较为大量的重打包应用、恶意应用和手机银行应用数据，并提供了重打包应用与原版差异、恶意应用传播方式与负载执行方式、手机银行应用常见漏洞等指导性信息，帮助专业人员改善现有移动应用环境。在丰富的前序研究提供的领域知识与数据支持下，各界才得以对重打包应用、恶意应用开展后续的检测研究。相比之下，仿冒应用的相关数据和研究成果都乏善可陈。

换言之，在上述研究中，无论是对恶意应用还是重打包应用进行检测，都需要有一定领域知识或洞见（Insight）作为理论支撑。对仿冒应用而言，学术界与工业界对其认知均较贫乏，“基于新见解进行技术拓展改良”式的研究手段尚未适用于仿冒应用。因此，着眼于领域知识梳理的实证研究更适用于仿冒应用在当下的研究场景。

1.3 拟采用的研究方法

上述研究现状说明，实证研究在多个研究方向上提供了研究数据或领域知识，有一定实用价值。当下的仿冒应用研究，既缺乏可用的数据集，也缺乏对应的基本

特征知识。作为基于数据采集和数据分析的技术，实证研究方法正好可缓解仿冒应用领域知识和数据集的缺失，是适用于本课题的技术手段。本研究将采用实证研究方法，对仿冒应用进行研究。

实证研究是一种基本研究技术，旨在针对特定方法在实际应用场景下的真实状况或对应产物进行数据收集、调查与分析，以让研究者了解事物的工作原理或使理论得以验证，其中数据收集是十分重要的一部分。前人总结^[45] 对实证研究的数据收集方法给出了严格指引，指出数据收集时需要确保数据的准确性与全面性。因此，科学地进行数据收集是本研究面临的一个关键问题。

1.4 本文拟解决的关键问题

上一节提及，科学地进行数据收集是本研究面临的一个关键问题。此外，本研究还有两个关键问题，一并列举如下：

- 如何科学地获取针对仿冒应用的数据？ 由于仿冒应用数据缺失，本文只能自行收集数据。数据搜集是科研工作中公认的难点，为提供一次全面的研究结果，除了数据需要有一定规模以外，搜集的目标应用也需要具备多样性，此外还必须获得不同应用市场上的数据，增加研究的代表性。为此，如何获得大量真实、有效、全面而有代表性的数据将是本文研究的一个关键点。
- 如何对仿冒应用进行理解？ 仿冒应用虽然普遍存在于各大应用市场中，但尚未有研究对其进行分析，无论是应用市场、研究者还是用户，都缺乏对仿冒应用的理解。为能较全面地认识仿冒应用，需要从不同角度对其进行解读。因此，从应用基本属性、应用作者行为等多角度对仿冒应用进行解析，是本文研究的重点之一。
- 是否能有效地筛选仿冒应用 根据日常经验不难发现，在现阶段，用户在实际使用中经常能搜索到仿冒应用，应用市场对仿冒应用的筛选排除并未完善。恶意应用、重打包应用的研究发展表明，充分认识移动黑灰产应用特征对后续的对应检测、发现有明显促进作用。同理，作者希望可基于对仿冒应用进行的解读与画像，设计有效流程帮助各大市场梳理、排除仿冒应用。

1.5 本文主要工作

为解决上述三个关键问题，本研究完成了如下主要工作：

- **全面的数据集整理** 借助自行实现的仿冒应用收集框架，本研究尽可能全面地收集了来自 29 个应用来源的约 14 万个应用样本，并从中筛选出 5 万个仿冒样本，整理成已知的首个仿冒应用数据集，以供其后的相关研究使用。
- **仿冒应用数据画像与现实案例分析** 本文从基本信息与开发者行为两个方向入手分别进行了实证研究，前者对仿冒引用进行了较为全面的数据画像，后者根据仿冒应用开发者行为推断出现有应用市场监管不足之处。两组研究除了从较高维度进行数据分析外，还从数据集中挑选出应用和证书实例，分别展开案例讨论，以深化在数据画像和行为分析中得到的领域知识。
- **提供实用建议** 基于在实证研究中的发现，本文分析了仿冒应用特征的可能成因与现有应用市场的不足之处，由此分别面向普通用户、应用市场和相关从业者提供了实用建议。
- **仿冒应用检测框架** 依靠前序研究中对仿冒应用的理解与解读，作者设计并实现了仿冒应用检测框架 FakeRevealer。

1.6 本文组织结构

本文共分为七章，环绕着本研究的数据搜集和不同的分析方向展开，各章节内容如下：

第一章 主要提供了本文的研究背景、相关工作、本文拟采用的研究方法、拟解决的问题与本文的主要工作。

第二章 介绍相关技术背景，从软件工程领域的实证研究出发，介绍实证研究的常用方法和数据收集、验证标准等理论背景，再着眼于本研究的主体——Android 应用，介绍了 Android 安全证书机制和与仿冒应用相关的重打包技术。

第三章 介绍整个研究的设置，包括根据研究现状提出研究问题、确立研究对象，阐述数据收集的相关考量，并与其他研究收集的数据进行对比。

第四章 从仿冒应用与原版应用相似度、影响应用被仿冒的严重程度的因素及仿冒应用行为三点入手，对与仿冒应用基本特征相关的三个方面进行实证研究，最后针对研究结果分别向用户、应用市场和应用开发者提供实用建议。

第五章 则结合应用证书视角与结合时间维度视角进行探索，分析现有应用市场监管策略存在的不足，针对性地提出改善措施。

以前两章实证研究所得的经验作参考，作者于**第六章** 提出并实现了一个仿冒应用检测框架，并基于新采集的应用数据对框架进行了有效性检验。

第七章 对本文工作进行总结，并对下一步工作进行展望。

第二章 相关技术背景介绍

本章对本研究涉及的相关知识作背景介绍，包括与研究方法相关的实证研究相关知识以及与研究主体相关的 Android 背景知识介绍。

2.1 实证研究相关背景介绍

实证研究是本研究主要采用的研究方法，因此首先介绍实证研究的相关背景与常用方法。

2.1.1 软件工程中的实证研究

在软件工程领域，由于理论研究与工程实践结合较为紧密，学者采用需要检验理论在实践中的落地情况（如程序语言提供的特性是否被良好地理解与使用^[46]、是否在工程实践中体现优势等^[47] 问题）或是调研现实应用场景中产生的现象（如恶意应用、重打包应用等在移动端市场出现的实际问题^[5,26,41,48]），实证研究技术是十分适合的方法。因此，有越来越多的研究者使用实证研究技术对研究主体进行探索^[5,6,26,41–44,46–54]，并为相关研究领域贡献了宝贵的知识。随着实证研究技术在软件工程领域中的认可度逐渐提高，软件工程顶级会议 FSE 与 ICSE 近年分别新设立了面向工业界的投稿分区，鼓励学者多基于应用场景进行实证研究，探明业界现状。

理论方面，为向软件工程领域进行实证研究的学者提供方法论支持，Kitchenham 等人基于自身在评阅软件工程项目的经验和一份针对医学研究者的研究指引，总结出了一份对于软件工程实证研究的初步准则（Preliminary guideline）^[45]，对实证研究的数据收集方法、实验设置方法等各个步骤均给出严格指引，如在数据收集时需要确保数据的准确性与全面性，需要保证实验的可重复性等。Seaman 则结合实例，提供定性分析的方法建议^[55]。在实证研究可采取的具体方式方面，Easterbrook 等人提出了面向软件工程领域的实证研究方法建议^[56]，为软件工程实证研究提供

方法论参考，文中将实证研究方法论分为受控实验、案例分析、调查研究、社会学意义研究与混合方法途径等多类，以适用于不同场景。

2.1.2 实证研究方法论

上一小节提及，软件工程的实证研究方法有前人文献可供参考^[56]。本小节将对介绍文献中总结可用的实证研究方法。

1) 受控实验（Controlled Experiments）

受控实验是用于验证假设的一种研究方法。研究者通过对实验中的自变量（Independent variable）进行小心控制，以测量自变量对一个或多个因变量（Dependent variable）产生的影响。受控实验可使研究者精确地确认各个变量之间如何关联，并特别适用于探究各变量间是否存在因果关系。为使结果具有说服力，受控实验中除自变量以外的其他变量不应对实验结果产生影响。

2) 案例分析（Case studies）

案例分析是软件工程领域最常用的实证研究方法，完整的案例分析通过确立研究问题、选择研究案例和收集数据三步研究真实场景中出现的现象，适用于真实环境对研究主体产生影响的因素之一、又或是实验数据时间跨度较大的场景。对于针对某些现象的初步调查，可使用探索性案例分析（Exploratory case studies）以提出新猜想和构建理论；而验证性案例分析（Confirmatory case studies）则用于验证现存理论。本文的案例分析有用于提出新猜想的探索性案例分析，也有为研究发现提供案例实证的验证性案例分析。

3) 调查研究（Survey Research）

研究调查常用于需要从广泛个体中提取出普遍特征的场景，在数据收集上常以问卷调查为形式进行。然而，其所用的数据也可以通过数据日志技术或机构化访谈获取。对调查研究而言，最大的挑战是对采样偏差（Sampling bias）的控制。若采样出现偏差，所获数据无法代表目标群体，对应的结论也会失去代表性。根据以上定义，本研究可归入调查研究范畴之中，针对缓解采样偏差采取的具体措施，可见下一小节中外部有效性部分的讨论。

4) 社会学意义研究（Ethnographies）

社会学意义研究通过实地考察的形式，对社群（Community）中的人们进行的社会活动进行理解。在软件工程领域中，这样的方法可帮助研究者了解应如何在社群中建立实践与交流文化，从而促进社群中的技术合作。社会学意义研究是把研究重点聚焦于人而非技术的一种研究方法，有助于揭示特定社群（在软件工程场景下为技术社群）的真实状况，从而为实践工作提供微妙而重要的见解。

5) 混合方法途径 (Mixed-method approaches)

混合方法途径指结合定量分析与定性分析，对研究对象进行系统数据解读的实证研究方法。按照实施的方式，混合方法途径可分为顺序解释策略（Sequential explanatory strategy）、顺序探索策略（Sequential exploratory strategy）与并发三角策略（Concurrent triangular strategy）三类。顺序解释策略先收集与分析定量数据，再收集和分析定性数据，以定性数据结果帮助解释定量结果；与之相反，顺序探索策略先收集与分析定性数据，再收集和分析定量数据，以定量数据结果帮助解释定性结果；并发三角策略则会同时采用不同方法，以试图确认、交叉验证或证实已有发现。

2.1.3 有效性验证标准

除上述方法外，文献^[56]还指出研究报告中应以一定篇幅分析实验设计或采取方法中可能使结论有效性受影响的部分（即有效性威胁，Threats to validity），以表示研究者在研究过程中已经考虑到了这些因素的影响，亦曾试图将此类因素对结论有效性产生的影响最小化。分析可从四个有效性验证标准出发进行，本研究在进行时对各标准进行了一定参考。

1) 结构有效性 (Construct validity)

结构有效性关注理论结构是否被正确解释以及相关指标是否被正确测量。为保证本研究的结构有效性，本文将在从各角度分析仿冒应用时，解释该维度用到的测量标准，以提供参考。

2) 内部有效性 (Internal validity)

内部有效性关注研究设计，要求研究者排除非自变量因素带来的影响。本研究的研究主体为 Android 中的仿冒应用，各实验中的自变量均为与仿冒应用相关的数

据。从此角度分析，确保收集到的仿冒样本真实有效，即可在一定程度上确定本研究的内部有效性。使用应用证书中的信息鉴别仿冒应用是行之有效的，因此能保护本研究的内部有效性。

3) 外部有效性 (External validity)

外部有效性关注研究得出的结论是否具有普遍性，通常与研究中数据收集时的采样相关。考虑到结论的普遍性问题，本次研究中进行数据采集时，已尽可能广泛地进行采样。本研究的数据来自 29 个应用来源，50 种目标应用涵盖 11 个类别，并一共采集到近 14 万个应用样本，由此可认为本研究的结论具有一定普遍性与代表性。

4) 可靠性 (Reliability)

可靠性关注其他研究者是否能通过重复报告中的实验复现报告中的结果与结论。本研究的结果与结论均通过对数据进行分析获得，分析方法亦将于后文细述，其他研究者可利用文末提供的信息下载对应数据，再采用相同方法进行分析，可获得同样的结果。

2.2 Android 相关背景介绍

本研究在收集、筛选仿冒应用样本时，主要使用了应用证书进行筛查，因此先在本节对 Android 的应用证书机制作介绍。此外，鉴于本研究在实证研究中讨论了仿冒应用与重打包应用的联系，本节也将相关知识作为 Android 背景知识一并介绍。

2.2.1 Android 应用证书机制

1) 加密散列函数与消息摘要技术

消息摘要 (Message Digest) 又称 hash 值或 hash，是将数据输入加密散列函数 (Cryptographic hash function) 后所得的输出结果。对消息采用加密散列函数得到消息摘要的行为被称为提取摘要或者提取 hash 值。加密散列函数是散列函数的一种，可理解为一类加密算法，有如下特性^[57]：

- 对任何给定的信息，都能算出一个 hash 值 无论输入的信息长度如何，加

密散列函数均能在多项式时间内给出一个具有固定长度的 hash 值。该长度具体由加密散列函数采取的加密算法决定，如被广泛使用的 MD5 算法返回结果的长度为 128bit（可被表示为 32 位十六进制数字），SHA-1 算法返回长度为 160bit（40 位十六进制数字）等。一般来说，可认为返回的结果越长，采用的加密算法越安全。

- 难以由函数的输出值反推出函数的输入值 加密散列函数是单向函数，即易于凭借一定输入获得一个输出，但难以利用该输出反推出原本的输入。基于此特性，想要通过一个 hash 知道原本的数据几乎是不可能的。理论上，基于后文描述的无碰撞特性，可采用暴力爆破的方法，对所有可能的输入计算 hash，再比对所得结果，找出某个 hash 对应的原输入。然而，由于前文提及的特性，加密散列函数的输入空间是无穷大的，在一般情况下要通过暴力破解法反推 hash 的原输入几乎不可能。
- 理想的加密散列函数具有确定性与无碰撞特性 对于理想的加密散列函数，任何输入均只对应唯一输出，即不会发生结果碰撞（不同输入得到相同输出）的情形，且对同样的输入也总会得到同样的输出。受制于现实条件，目前的加密散列函数都有可能发生碰撞。然而，好的加密散列函数应该只有极低的概率会发生碰撞事件。基于此特性，可使用通过较安全的加密散列函数处理的 hash 进行安全性验证。
- 对输入的细微改变会导致输出变得截然不同 该特性被称为雪崩效应，是密码学上的概念，具体指由输入值产生细微变化（如一个 bit 的翻转）导致的输出的不可区分性改变（即输出中每个二进制位有 50% 几率发生翻转）^[58]。基于这种特性，加密散列函数的结果充满随机性，使得攻击者难以推测利用函数输出反推函数输入。

基于以上特点，由各类加密散列函数处理得到的信息摘要具有一定安全性，常用于数据完整性检查、数字证书签名和数据校验等场景中。

2) Android 中的数字签名

数字签名技术是一种用电子信息对数据进行签名，从而保证数据完整性与可信度的技术，是不对称加密算法与消息摘要技术结合的一种应用场景。不对称加

密算法中，信息发布者有两把密钥，分别是公开的公钥和不能公开的私钥。由于公钥加密的信息仅可由私钥解密，私钥加密的信息也仅可用公钥验证，信息发布者可利用不对称加密算法确保发布的数据不被篡改。然而，非对称加密算法在加解密过程中效率并不高，当信息中心的数据量较大时会产生性能问题。因此，数字签名技术还结合了消息摘要技术确保性能。

具体而言，数字签名技术在实际应用中可分为两个步骤（见图 2.1），分别为消息发送方的签署与消息接收方的验证。在签署步骤中，发送方先使用加密散列函数处理待发送的数据，得到待发送数据的 hash 值 H_A 。其后，发送方使用自己的私钥对该 H_A 进行加密，此处称加密的结果为签名。签名随后将与一份证书（其中附有发送方的公钥和前述的加密散列函数算法信息）一起附于数据中，组成图 2.1 中经过数字签名的内容。而验证步骤中，接收方在收到上述经过数字签名的内容后的处理可分为两部分：一部分为取出其中的数据，根据证书中提供的加密散列函数处理该数据，得到 hash 值 H_B ；另一部分为取出签名与证书中的公钥信息，用公钥对签名进行解密，即可得到原数据的 hash 值 H_A 。只要 H_A 和 H_B 相同，就能确认数据在传递过程中没有被损坏或篡改。

在 Android 系统中，数据签名技术的重要应用之一是对应用安装文件（APK 文件，其本质为一个压缩文件）的验证，APK 文件构建尾声和 Android 系统对 APK 文件的安装前验证两个节点分别对应数字签名技术中的签署和验证步骤。

在 APK 文件构建步骤尾声，APK 打包器开始如下签名过程，并生成三个文件：(1) 遍历 APK 中所有文件，对每个文件都提取摘要（常用 SHA-1 或 SHA-256 算法），并将每个文件的路径和 hash 值分别编成条目，把所有条目写入一个新文件 *MANIFEST.MF* 中；(2) 使用同样的加密散列函数对 *MANIFEST.MF* 提取摘要，写入第二个新文件 *CERT.SF* 中，再对 *MANIFEST.MF* 中的每个条目分别再进行一次摘要提取（二次摘要），每个摘要结果连同条目对应的文件路径也写入 *CERT.SF*；(3) 对 *CERT.SF* 的内容使用 APK 发布者的私钥加密，加密结果为一个数字签名，将该签名、签名流程使用的加密散列函数算法以及包含公钥信息的数字证书一同写入第三个新文件 *CERT.RSA* 中。三个新文件与 APK 中原有的内容组成了经过数字签名的内容。

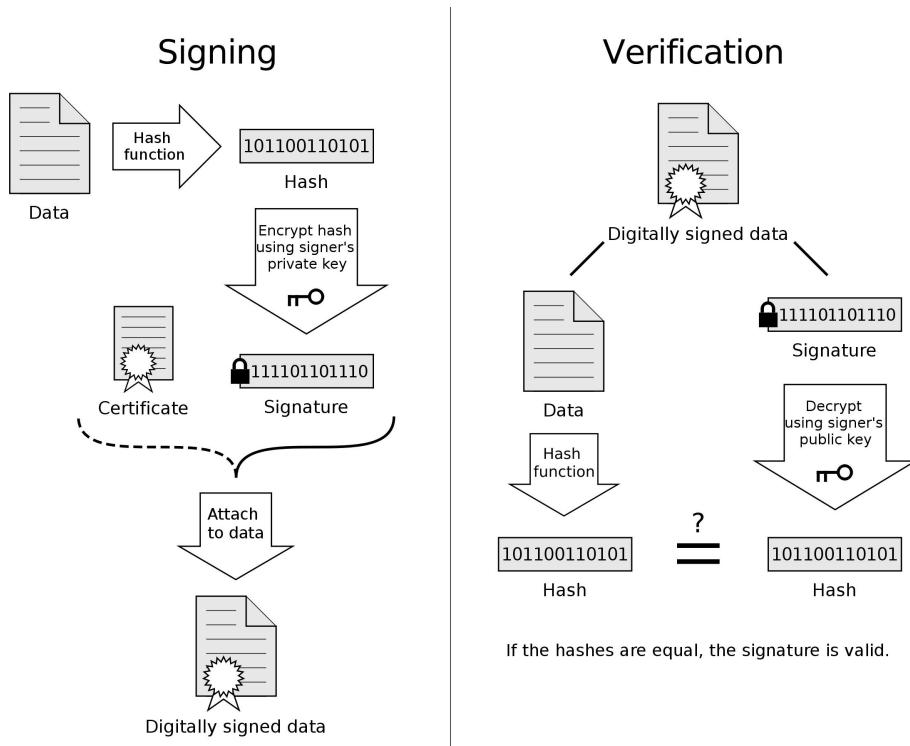


图 2.1: 数字签名技术的两个步骤

Android 系统对 APK 文件的安装前验证则是与上述流程近似的过程：（1）扫描 APK 中的所有文件，按 *CERT.RSA* 提供的加密散列函数算法对每个文件提取摘要，和 *MANIFEST.MF* 中的每个条目比对；（2）用 *CERT.RSA* 提供的公钥对签名进行解密，将解密得到的内容与 *CERT.SF* 的内容进行比对；（3）对 *MANIFEST.MF* 每个条目进行二次摘要，将二次摘要与 *CERT.SF* 对应条目中的摘要进行比对。原理上，若 APK 中文件被第三方篡改，被篡改文件的摘要就与 *MANIFEST.MF* 中的对应摘要不符；若进一步篡改 *MANIFEST.MF* 的对应摘要，则会导致其二次摘要与 *CERT.SF* 中的对应内容不符；篡改方没有 APK 发布者的私钥信息，无法产生新的签名将 *CERT.RSA* 中的原有签名替换，即使其再篡改 *CERT.SF* 中的二次摘要，也无法令上述第（2）步中的解密结果与被篡改后的 *CERT.SF* 内容相吻合。因此，验证流程中，任何一处产生不相符均会导致验证失败，从而令 Android 系统终止安装流程。

在 2017 年，Android 披露了名为“Janus”关于签名机制的重大漏洞。若 APK 仅使用 V1 版本签名机制对应用签署，恶意开发者可通过直接在 APK 文件后拼接恶

意代码对 APK 进行篡改，无需经过解包和重打包步骤，即绕开了签名的步骤。换言之，基于 V1 机制签名的 APK 文件，即使具有正版开发者证书，也可能是被篡改过的版本。面向该问题，本文第六章的鉴别模块中采用了针对 Janus 漏洞的检测机制。

2.2.2 重打包技术

上述的数字签名技术可保证原 APK 被解包篡改后可被发现，但并不能防止对 APK 的篡改行为。篡改方对 APK 进行修改之后，重新对修改后的内容进行构建的行为被称为重打包。重打包行为的危害与针对重打包应用的研究现状已在第一章有所介绍，在此不再赘述。

从技术层面分析，重打包采用反编译手段对原 APK 进行篡改，整体过程可被拆分为以下三步：(1) 利用 Apktool^[59], dex2jar^[60], jd-gui^[61] 等开源工具拆解 APK 包，反编译出其中包括 java 源码在内的内容；(2) 对反编译得到的内容进行修改，包括增删代码，替换资源文件等操作；(3) 对修改后的内容重新进行打包、签名（使用篡改方的私钥进行签名），完成构建。经过重打包的应用 APK 文件可通过 Android 系统对 APK 的安装前验证，进而安装至用户设备中。受限于前文的 Android 应用证书机制，由于重打包应用开发者不具备原版开发者的密钥信息，解包后必须使用篡改方的密钥进行重新签名，因此重打包 APK 的 CERT.RSA 中文件包含的公钥是篡改方的公钥，而非原开发者的公钥信息。按此原理，可直接使用 APK 包内 CERT.RSA 包含的公钥信息辨认开发者，分辨出正版应用与重打包仿冒应用。

2.3 本章小结

本章首先介绍了实证研究的相关背景与前人提出的方法与建议。这些方法在本研究中起到了一定指导作用。之后，本章详细地阐述了 Android 的应用证书机制，简要介绍了重打包技术，为下文对仿冒应用的收集与筛选和后续的分析做铺垫。

下一章中，本文将介绍本研究的设置，包括本文的主要研究问题，研究对象、本研究使用的数据集来源和数据收集方式等。

第三章 实证研究设置

仿冒应用的生态是移动黑灰色产业研究中尚未被探明的一环。为了补上这一研究空白，本文先提出了一系列针对仿冒应用的研究问题，再使用实证研究技术对这些问题一一解答。实证研究需要数据支撑，在缺乏前人研究提供数据或分析的基础上，本研究从工业界直接获取真实世界的样本以完成调研。本章介绍本研究的整体设置：先提出本文的研究问题，再介绍研究对象的选取（即从什么对象中获取研究数据），最后描述数据搜集思路方法与所得数据集和其他已有数据集的对比。

3.1 研究问题

一方面，探明仿冒应用生态的重要一步是了解其本身特征，以协助后续研究中对仿冒应用的自动化辨识和指导一般消费者鉴别仿冒应用。从直觉上看，仿冒应用与某一正版应用越相似，则越可能误导用户下载应用，使仿冒开发者获利。在这一方面，为了解、验证仿冒应用特征，本文从相似度、影响应用被仿冒程度的因素及仿冒应用行为三点出发，提出了以下问题：**RQ1：**仿冒应用和与之相对的正版应用的相似程度如何？**RQ2：**是否存在显著影响应用被仿冒的严重程度的单一因素？**RQ3：**仿冒应用作者制作出了怎样的仿冒应用？是否依然能提供原版应用的功能？

另一方面，仿冒应用的发布行为特征与仿冒应用本身特征同样重要，了解仿冒应用作者的发布特征有利于使市场方强化筛选、审查方案，防止仿冒应用流入市场。由于应用开发者与应用证书的一对多关系（详见第二章），本研究借助仿冒应用的证书信息探究仿冒应用作者的行为。在这一方面，为了解仿冒应用开发者行为特征，本文提出了以下研究问题：**RQ4：**仿冒应用与开发者的对应关系如何？**RQ5：**仿冒应用证书可以活跃多长时间？**RQ6：**仿冒应用的上架行为是否有特征？

上述研究问题从三个不同角度，即仿冒应用基本特征、仿冒应用开发者发布行为的特征和仿冒应用的市场反馈对其进行探究。这些问题的答案不仅可为后续研究提供仿冒应用的基础认知与检测思路，还能同时为普通消费者和应用市场在鉴

别仿冒应用方面提供一定线索。

3.2 研究对象

仿冒应用是一个跟正版应用相对的概念，所以需要先定义正版应用，再对仿冒应用进行探究。因此，本研究先选取了一部分正版应用作为目标应用，再根据正版应用的信息搜寻仿冒应用，对搜集到的样本进行研究。

文献^[45]指出，实证研究需要确保数据的准确性与全面性。若在市场直接中随机挑选应用，将难以解释搜集到的应用是正版或是仿冒应用，进而不能保证数据的准确性，随机选取法并不适用于正版应用的选取。因此，本研究参考了数据平台易观千帆的月度 App 排行榜¹。该排行榜对超过 1200 个主流应用中的用户分享数据进行分析，汇总成应用热度（月度活跃用户数），按热度对应用进行排行。一方面，该排行榜的数据来源是各应用“应用体验计划”中搜集所得的数据，数据本身具有一定真实性，可满足文献的准确性要求；另一方面，排行榜中的应用均为主流应用，由知名度较高的开发者（如腾讯、百度、阿里巴巴等）开发，可认为排行榜上的应用均为正版应用，免去需要对搜集到的目标应用进行甄别的步骤；此外，排行榜中的应用涵盖了多个领域，包括社交网络、资讯、移动购物、视频等，能满足数据收集中的全面性要求。综上，从易观千帆平台选择目标应用是较为合适的方式。

该平台月度 App 排行榜的排名前 50 的热门应用被挑选作为本文的目标应用。从功能角度入手，50 个 App 被归入 11 个类别，分别为社交网络、视频、生活、移动购物等，详情可见表 3.1。作者手动从这 50 款 App 的官方网站上下载了这批应用的最新版本，作为正版应用的参考版本，寻找与之相关的仿冒应用进行后续研究。

3.3 数据收集

鉴于目前学术界中与仿冒应用直接相关的数据集较为稀缺，从工业界中系统地收集所需的仿冒应用数据来完成本次调研是最为直观合适的方法。本节先根据上述的研究问题划定本研究所需的研究数据类型，再介绍如何对数据进行收集。

¹<https://qianfan.analysys.cn/refine/view/rankApp/rankApp.html>

表 3.1: 目标应用类别与各分类下应用

应用类别	应用名
社交网络	微信、QQ、新浪微博
视频	爱奇艺、腾讯视频、优酷、快手、抖音、火山小视频、西瓜视频
生活	支付宝、高德地图、百度地图、美团、墨迹天气、滴滴出行
移动购物	淘宝、京东、拼多多
系统工具	WiFi 万能钥匙、搜狗输入法、腾讯手机管家、360 手机卫士、猎豹清理大师、360 清理大师、WiFi 管家、讯飞输入法
资讯	百度、腾讯新闻、QQ 浏览器、今日头条、UC 浏览器、网易新闻
应用市场	应用宝、华为应用市场、OPPO 应用市场、360 手机助手、百度手机助手、小米应用市场
音乐	酷狗音乐、QQ 音乐、酷我音乐盒、全民 K 歌、网易云音乐
游戏	王者荣耀、开心消消乐
摄影录像	美图秀秀、美颜相机
商务办公	WPS Office、QQ 邮箱

3.3.1 数据类型与收集方式

综合研究问题，本研究需获取的数据类型可分为以下两部分：一、分析仿冒应用特征所需的应用基本信息；二、分析应用发布特征所需的应用证书信息、上架时间等元数据。

确定目标应用后，可在市场中寻找与目标应用相似的应用，提取其数据整理成集，满足一、二部分的数据需求。但要获取足量的仿冒应用数据以组成数据集是一个较有挑战性的任务，有难点如下：

- 研究者要从多个不同的应用市场中爬取 App 样本，每个应用市场都有不同的网页编码，甚至有反爬虫策略，不存在一个爬虫脚本对所有应用市场数据都通用的场景；
- 各个应用市场架上的 App 数量庞大，研究者需要有效地筛选出与目标应用有关的样本；
- 对于收集到的大量数据，研究者需要一个轻量级的解决方案快速判断获得的

App 样本为正版应用或是仿冒应用。

为应对第一个挑战，作者与工业界合作，利用舜众信息公司的 Janus 平台¹对各大应用商店进行样本爬取，获取应用样本信息。该平台自 2017 年起就开始对各大应用商店的 App 进行样本收集，可向用户提供各项 App 信息。如图 3.1 显示，Janus 平台至今已收集到上千万个 App 样本，可满足本研究数据采集需求。除样本信息外，Janus 平台也提供按规则搜索功能，用户可创建规则过滤平台中的应用数据，以获取所需应用 App 样本。因此，视 Janus 平台为从各大应用市场中收集应用样本的代理，借助 Janus 平台获取各市场中的应用信息。

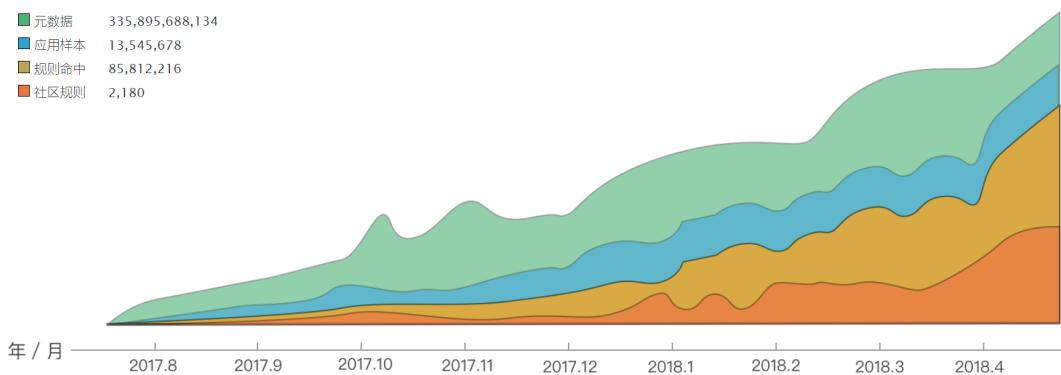


图 3.1: Janus 平台上的数据规模时序图

为应对第二个挑战，本研究提出了基于广度优先搜索（Breadth-First Search，简称 BFS）算法的搜索框架图 3.2。具体地，对于每一款目标应用的收集流程分为三个部分：先以正版应用样本的包名和应用名作为查询条件放入迭代搜索器，迭代搜索器发出查询请求，从 Janus 平台中收集与目标应用相关的所有 App 样本，提取出样本的相关数据返回至应用收集器；其后，应用收集器将数据转移至应用过滤器，利用过滤器判断收集到的应用数据源于正版应用或仿冒应用，保存仿冒应用数据，提取出正版应用的包名和应用名；最后，将上一步中获得的新的正版应用信息（包名和应用名）再次放入迭代搜索器，开始下一次循环，从 Janus 平台中进行查询，获取更多与正版应用相关的样本。

第三个挑战，可利用 Android 签名证书机制（详见第二章）解决。对某个 APK

¹此处平台与第 2.2.1 节中提及的 Android 漏洞同名，但两者无直接关联，请注意区分。后文的“Janus 平台”均指本平台，而“Janus 漏洞”则特指 Android 漏洞。

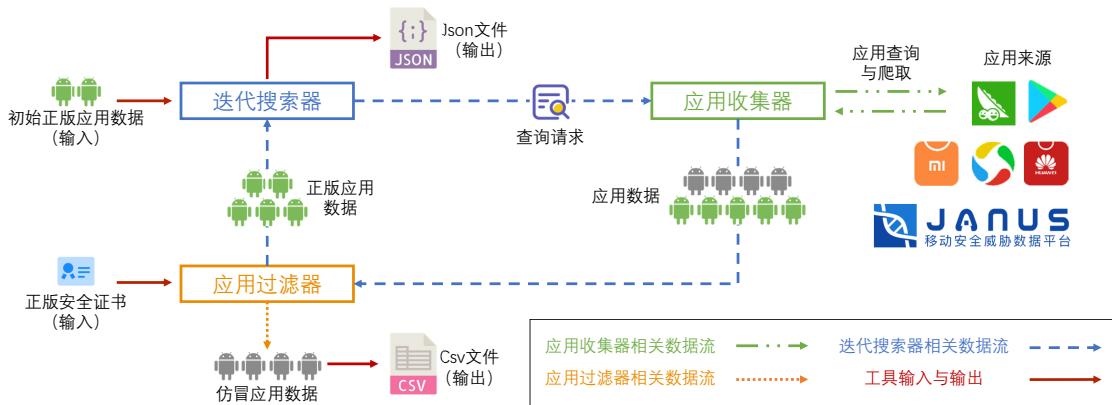


图 3.2: 搜索框架整体流程

安装包而言，该机制提供了其被最后一次修改时的修改人信息。因此，对于任一个被搜索框架收集到的 APK，均可归入以下四种情况：

第一，该 APK 为正版应用，由正版开发者开发，也并未被篡改，应用证书与正版应用证书一致，是目标应用的某一版本；第二，该 APK 原本为正版应用，由正版开发者开发，使用旧版签名机制签署。由于 APK 采用了旧版签名，仿冒应用开发者利用漏洞篡改了部分数据，使样本成为仿冒应用，但应用证书并未被替换。第三，该 APK 原本为正版应用，被仿冒应用开发者解包篡改成为仿冒应用，但篡改是在解包后进行的，仿冒应用开发者无法通过原版证书为该 APK 重新打包，因此该 APK 的应用证书被替换为仿冒应用开发者的证书；第四，该 APK 由仿冒应用开发者开发，由于仿冒应用开发者不具有正版开发者的私钥，不能为该 APK 签署正版应用证书，因此该 APK 具有仿冒应用开发者的证书。

由于旧版签名机制存在漏洞，作者无法直接判断使用旧版签名机制签署、同时具备正版证书信息的 APK 文件是否曾被篡改。然而，只要其应用证书信息与正版应用信息不符，无论其为被解包篡改的原正版应用，或是仿冒应用开发者从新开发的仿冒应用，均可划入仿冒应用范畴。

3.3.2 数据概览

从易观千帆提供的数据榜单中，本研究选择了分属 11 个不同的应用类别的 50 个最热门的 App 作为目标应用，。由于 App 的应用名可能会在 App 更新迭代的时候随之变更，作者用近似 BFS 的策略，从 50 种 App 中一共记录了 198 个不同的应

用名以挖掘仿冒样本。

在这 50 款 App 中，以下三款 App 的样本并不能在市面上找到：*OPPO* 应用商店，*华为应用商店*和*小米应用商店*。因为这三款 App 都是由手机设备厂商开发和预装在对应品牌的手机中的，仅供这些品牌的用户使用，并不在其他应用市场上提供下载。当然，这也是这三款 App 热度高的原因——这几款 App 都被预装到了对应手机品牌厂商的每一部 Android 设备中，而 *OPPO*、*华为*和*小米*又是国内最大的几家手机厂商，这几款 App 自然也会有庞大的用户基数。因此，本研究最后的目标应用只有 47 款。

对这 47 款目标应用，搜索框架共检索到 138,106 个应用样本，样本源于 29 个不同渠道。其中，69,614 个应用样本持有官方开发者证书，52,638 个应用样本并不具有官方证书。余下部分应用样本为某些应用的发布在不同应用市场同一版本，在对各样本计算 SHA1 码去重筛选后被排除（共计 15,854 个）。样本 SHA1 码是使用 SHA1 哈希算法对整个 APK 文件进行数据摘要之后获取到的编码串，可认为每个样本都有独一无二的 SHA1 码。

3.3.3 相关数据集对比

作为软件工程方向的研究热点，Android 应用研究一直广受关注。在 Android 应用数据集方面，为推进 Android 恶意应用研究，一些安全厂商整理出了恶意应用数据集，使研究者免于数据收集之苦；部分论文中亦有公开的 Android 应用集可供参考。较为著名的数据集有 *Android Malware Genome Project*（Genome 数据集），*Drebin* 数据集，*AMD* 数据集和 *AndroZoo* 数据集。

Genome 数据集^[5] 中的样本由 *Yajin Zhou* 等人持续收集一年余所得，其中包含了超过 1,200 个 Android 恶意应用，涵盖了收集时市面上存有的大部分恶意应用家族。通过对数据集中恶意样本的详尽分析，*Yajin Zhou* 等人总结了重打包、更新攻击和路过式下载（Drive-by Download）三种恶意应用的安装途径，并揭示了特权提升，远程控制，恶意扣费和信息收集四种恶意行为的方式，该数据集在后人研究中多被用于训练自动化恶意应用分类器。该数据集收集的时间为 2010 年 8 月至 2011 年 10 月间，但已不再被维护。Android 系统在其后不断更新迭代，故其中的样本已

有部分因兼容性问题不能再被安装到具有较新版本 Android 系统的设备上。与该数据集类似的还有与 2010 年 8 月至 2012 年 10 月间收集的 Drebin 数据集^[40]，该数据集为 Genome 的超集，共包含涵盖了源于 179 个不同恶意应用家族的 5,560 个恶意样本。AMD 数据集^[62]则包含了从 2010 年至 2016 年间收集的 24,553 个样本，样本被分类至 71 个恶意应用家族的 135 个变种中，与 Genome 数据集和 Drebin 数据集相比，提供了更接近现状的恶意应用数据画像。除去时间导致的兼容性问题，虽然 Genome、Drebin 和 AMD 三个数据集中的样本覆盖较为全面，但其收集对象已明确为产生恶意行为的“恶意应用”，而本文研究对象为模仿热门应用的仿冒应用，两者有所区别，因此直接使用三个数据集中的样本进行分析并不合适。

AndroZoo 是一个仍在不断收集应用的数据集^[7]。该数据集从包括 Google Play 应用商店、小米应用商店、安智网等多个数据源进行数据收集，现已包含超过一千两百万个应用样本。一方面，AndroZoo 的数据比前述数据集更新，更能反映应用市场现状；另一方面，比起 Genome、Drebin 和 AMD 三个针对恶意应用进行收集的数据集，AndroZoo 收集的对象不限于恶意应用。该应用集利用 VirusTotal 对所得的应用进行扫描，标记出被报告为恶意应用的样本；同时，维护者也整理了数据集中的重打包应用信息，为学者提供翔实的重打包应用相关的样本数据：该数据集的重打包应用目录共提供了 2,776 个原版应用样本对应的 15,297 个重打包样本。与只针对恶意应用进行收集的数据集相比，同时收录重打包应用数据的 AndroZoo 数据集可用于更广泛的 Android 应用研究。然而，重打包应用只是仿冒应用的其中一种形式，仿冒应用还包括仿照正版应用外观制作的应用程序，该数据集的重打包应用列表未能满足本文研究的数据需求。因此，在对上述四个数据集进行调研对比后，作者选择重新构建仿冒应用数据集，以完成本文研究。

3.4 本章小结

本章主要对本研究进行了概览，提出了本文的主要研究问题，阐述了本研究的对象，并简述了数据的收集方式与收集结果。最后，本章将数据集与其他已有数据集进行比较，解释为何重新收集数据，而非使用已有数据集。

第四章 仿冒应用基本特征实证分析

上一章节介绍了本研究的相关设置，提出了 7 个与仿冒应用相关的研究问题。本章从仿冒应用与原版应用相似度、影响应用被仿冒的严重程度的因素及仿冒应用行为三点入手，对与仿冒应用基本特征相关的三个方面进行实证研究，对应三个研究问题，即：

- RQ1：仿冒应用和与之相对的正版应用的相似程度如何？
- RQ2：是否存在显著影响应用被仿冒的严重程度的单一因素？
- RQ3：仿冒应用作者制作出了怎样的仿冒应用？是否依然能提供原版应用的功能？

4.1 研究方法

本节介绍本章用到的研究方法以讨论本研究的结构有效性，先介绍各度量指标，再讨论实验对象与实验数据。为回答 RQ1，本研究需要选择一种相似度指标衡量仿冒应用与原版应用的相似程度；为回答 RQ2，需要先选择一种指标衡量某款应用在受仿冒的严重程度程度，再将其与可代表应用热度、更新频率的指标相关联，以计算两者间的关联性。

4.1.1 相似程度度量指标

应用之间相似度可从多个层面定义，包括应用行为相似度，应用代码相似度以及应用外观相似度。应用行为相似度量度需要案例与一定方式驱动样例（常用的为手动方式驱动，或利用 UI Automator 或 Monkeys 等工具预先录制好操作脚本再回放）。考虑到采集到的应用规模较大，且种类繁多，要对逐个应用设计、驱动样例的可行性并不高，因此本研究不对所有应用进行行为相似度比对。应用代码

相似度比对常用于重打包检测相关研究，核心为对两个应用间的代码进行比较，计算两者间重合范围。然而，仿冒应用的意图为诱导用户下载安装，而代码层面的内容不在普通用户的可感知范围内。因此，仿冒应用代码并不需要与原版应用相似，代码相似度不是在本研究场景下最为重要的相似度指标。进一步推断可得，外观相似度是最符合本研究场景的相似度类别，仿冒应用开发者甚至可以重新开发与原版应用外观相似的应用骗取用户下载。根据日常经验，用户对应用外观的可感知的因素有以下四点：应用图标、应用 GUI 界面、应用标题（包括包名与应用名）和应用大小。基于采集到的数据，本研究利用应用标题与应用大小进行相似度比较。

在文本相似度比对方面，本研究采用编辑距离^[63]作为具体度量指标。编辑距离是在自然语言处理（Natural Language Processing，简称 NLP）领域被广泛应用的距离定义，其定义如下：

定义 1 编辑距离

给定两个字符串 a 与 b ，其间的编辑距离 $d(a, b)$ 为将 a 和 b 相互转换的最小编辑操作数。其中，每次添加、删除或将一个字符转换成另一个字符均算作一次编辑操作。

例如，“jingdong” 和 “jindeng” 之间的编辑距离是 2，由前者转换为后者的其中一种编辑次数最小方法为将第一个“g”删除，再将“e”转成“o”。同理，字符串“fake” 和 “official” 之间的距离是 7，其中一种方案为在“f”前添加“o”，在“f”和“a”之间添加“fici”（此处包含 4 步操作），将“k”替换作“l”，最后删去“e”。

4.1.2 应用受仿冒严重程度相关度量指标

本研究在衡量某款应用在仿冒应用开发者眼中受欢迎程度时，采用较为直观的指标：从直觉上看，市面上存在越多仿冒个体的应用，越受仿冒应用开发者欢迎（受仿冒程度越严重）。但是，每款目标应用都有不同的样本数（官方样本数与仿冒样本数均有区别），不能直接以仿冒样本的数量作比较。为消除偏差，本文将样本数量归一化，使用仿冒率对比每个目标应用在仿冒应用开发者中的受欢迎程度。

定义 2 仿冒率

某款 *App a* 的仿冒率 $fake\ sample\ rate_a$ 指与其关联的仿冒样本的数量 $fake_a$ 与该 *App* 正版样本的数量 $total_a$ 的商; 某应用市场 *AS* 的仿冒率 $fake\ sample\ rate_{AS}$ 则是其中包含的所有目标应用的均值。两者可分别计算如下:

$$fake\ sample\ rate_a = \frac{fake_a}{total_a}, \quad (4.1)$$

$$fake\ sample\ rate_{AS} = Avg(fake\ sample\ rate_a, \forall a \in \{\text{目标应用}\}). \quad (4.2)$$

应用热度方面, 研究直接取用从易观千帆平台取得的数据作为热度指标。易观千帆平台为国内领先的应用数据收集平台, 可认为其数据来源较为可靠。更新频率指标方面, 成熟的应用通常有较为固定的更新频率。为评估目标应用的更新频率, 本研究标记了每个官方应用样本被发行时的时间, 精确到日, 找到该应用的最新发布样本和最早发布样本, 两者发行时间差值与中间版本数的商即为平均更新频率(单位: 天/版本)。

4.1.3 关联程度度量指标

为探究应用受仿冒应用开发者的欢迎程度与其他因素的相关性, 本研究利用皮尔逊积矩相关系数(Pearson product-moment correlation coefficient, 简称 PPMCC)计算一款 *App* 被仿冒的严重程度与其热度、更新频率是否具有相关性。

定义 3 皮尔逊积矩相关系数

两个变量之间的皮尔逊相关系数定义为两个变量之间的协方差和标准差的商, 计算公式如下:

$$p_{x,y} = corr(X, Y) = \frac{cov(X, Y)}{\sigma_x \sigma_y} = \frac{E[(X - u_x)(Y - u_y)]}{\sigma_x \sigma_y}. \quad (4.3)$$

式 4.3 的值域为 $[-1, 1]$ 。该值越接近 1, 表示两个变量间存在正相关关系越强; 越接近 -1 , 表示两变量间存在的负相关关系越强; 接近 0, 表示两个变量之间的相关关系弱。

4.1.4 实验对象与实验数据

本章实验对象为 subsection 3.3.2 中的应用。为进行对比，本章研究从 69,414 个正版样本与 52,638 个仿冒样本中获取数据进行分析。研究利用 AAPT 脚本提取样本的应用名，包名与版本号，该脚本为 Android SDK 中自带脚本，可相信其提取结果无误；样本大小由 Python 标准库自带函数获取；搜集时间由 Janus 平台获取，为样本从应用市场被爬取到 Janus 数据库的时间点，准确到日，由于 Janus 平台不断对各大应用市场爬取新应用，可将该时间视作该样本的发布日期；APK 包来源同样从 Janus 平台获取，表示该样本的来源市场，因应用可于多个市场上架，所以一个样本可对应多个 APK 包来源。

4.2 实证研究流程与结果解析

本节分为四个部分，前三个部分分别针对章节起始的三个研究问题进行研究，最后一部分对实证研究结果进行有效性分析。

4.2.1 仿冒样本与原版应用的相似度

RQ1：仿冒应用和与之相对的正版应用的相似程度如何？

先从统计数据入手探索数据：包名方面，应用数据集统计结果显示，在所有的 52,638 个仿冒样本中，只有 243 个（少于 0.5%）使用了正版应用的包名，余下的所有仿冒样本都使用了自定义包名。自定义包名的 52,395 个样本中，包含了 14,089 个不同的包名。应用名方面则有截然不同的结果，大部分样本（41,863 个，约 79.5%）使用了与原版应用一致的应用名，余下 10,775 个仿冒应用样本包含应用名共 10,610 个。绝大部分仿冒应用并未保留原版应用的包名，但只有较少仿冒应用采用了新应用名。

作者推断，仿冒样本极少直接沿用正版应用的包名与 Android 系统机制有关。根据 Android 官方文档^[64]描述，每个 Android 应用都具备一个名为 Application ID 的属性，该属性为 Android 系统用于识别应用程序的唯一标识。通常情况下，该属性与包名一致。如果系统在安装 App 时发现系统中已经有具有相同 Application ID 的 App，将检查两个应用的应用证书是否一致，证书不一致会导致安装失败。因此

大部分仿冒应用不会直接使用正版 App 的包名。而应用名只有向用户展示的用途，不用于系统内的应用校验，系统也允许出现重复的应用名，所以大部分仿冒应用采用了与原版应用一致的签名以更好地诱导用户下载。

之后，研究采用如上节所述的外观相似度衡量指标，比对原版应用与仿冒应用之间的相似程度。对于从仿冒样本中获取到的每个应用名，均计算与其对应的官方发布 App 的原版应用名的编辑距离；同理，也计算出每个仿冒样本包名与原版包名的编辑距离，结果可见下图 图 4.1。

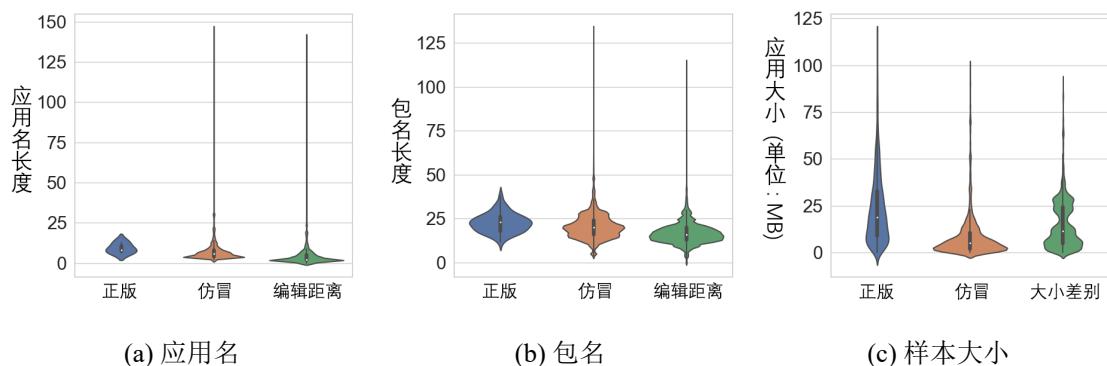


图 4.1: 应用的应用名、包名与应用大小的相似性比较

图 4.1由三个小提琴图^[65]组成，分别表示了本文在应用名、包名和 APK 包大小上的统计信息。每个“小提琴”的外部形状为数据的密度分布情况，小提琴中间的黑色粗条表示四分位数范围，粗条中小白点表示数据的中位数，黑色细条表示 95% 置信空间范围。

4.1a从左到右的三个图例分别展示官方样本、仿冒样本的应用名和两者间编辑距离的统计数据。正版样本和仿冒样本的应用名的平均长度较为相近（接近数值“6”），“仿冒”图例整体分布具有集中性，较多仿冒样本应用名长度落于 y 轴数值为 4 到 5 的区间中。“编辑距离”图例外部形状与“仿冒”图例大致类似，中位值较低（在 y 轴上为“2”），意味着过半数仿冒应用通过从正版样本的应用名中修改少于 3 个字符来获得其应用名。上述结果表示大部分仿冒应用正在使用与正版样本非常相似的应用名。将三个图例与前文统计数据结合，可推断具备不同应用名的 10,775 个仿冒样本多数在原版应用名的基础上删减少量字符以获得新应用名。同时，少量仿冒应用具有长名称（11 个仿冒样本的应用名长度大于 50 个字符，最长

的应用名包含 146 个字符)。作者对部分具有长应用名的仿冒样本进行人工检查,一些长应用名拼合了多个热门 App 的名称(比如“潮流女装-美丽说蘑菇街淘宝天猫京东美团精选”或“老黄历万年历日历-农历天气预报知乎倒数日记账事本闹钟备忘录优步滴滴打车同花顺大智慧微博大众点评小说壁纸”)。作者推测仿冒应用开发者采取该策略以令应用更易于出现在搜索结果中。

4.1b 显示了针对包名的结果。正版样本的包名长度中位值和仿冒样本的包名长度中位值较为接近(双方的值分别为“23”和“20”),两者外形较为近似,说明原版样本与仿冒样本在包名长度上有类似分布。然而,原版包名与仿冒样本包名之间编辑距离的中位数明显较高(在 y 轴上“16”的位置),这意味着将一个仿冒应用的包名转换为一个正版样本的包名平均需要进行 16 次编辑。换言之,正版样本的包名和仿冒应用的包名有明显差异,仿冒应用更倾向于使用自定义且与原版应用有较大区别的包名。

4.1c 则显示 APK 包大小的相关对比。为了能更好地显示结果,部分极端样本在作图前被剔除出数据集:该部分样本为大于 150MB 的 APK 包,在所有 69,614 个正版样本的样本中占 851 个(约为 1%),在 52,638 个仿冒样本中占 447 个(少于 1%),大部分来源于“游戏”类别下。图表显示,仿冒样本大小的中位数约为 5MB,约半数的正版 App 大小大于 18MB。“正版”图例中,应用大小样本分布相对均匀,密度曲线变化平缓,大部分样本分布于 (0, 60)MB 区间内,多数正版应用大小小于 25MB,但在大于 25MB 的区间内仍有少量样本分布;与之相比,“仿冒”图例的样本分布集中于 3 到 4MB 处,几乎没有样本大小大于 25MB;从该角度看,仿冒应用更有可能:

1) 由仿冒开发者自行开发,而非使用重打包技术制作。因为重打包之后的应用通常不会在大小上与原版有太大差距;

2) 是恶意应用。单独的恶意应用不需要包括较大的资源文件,只需少量具备恶意代码即可产生恶意行为;

简而言之,可从图 4.1 中获得两点总结,回答 RQ1:

1) 更倾向于使用与正版 App 相似(甚至相同)的应用名,但包名与正版应用相差较大;

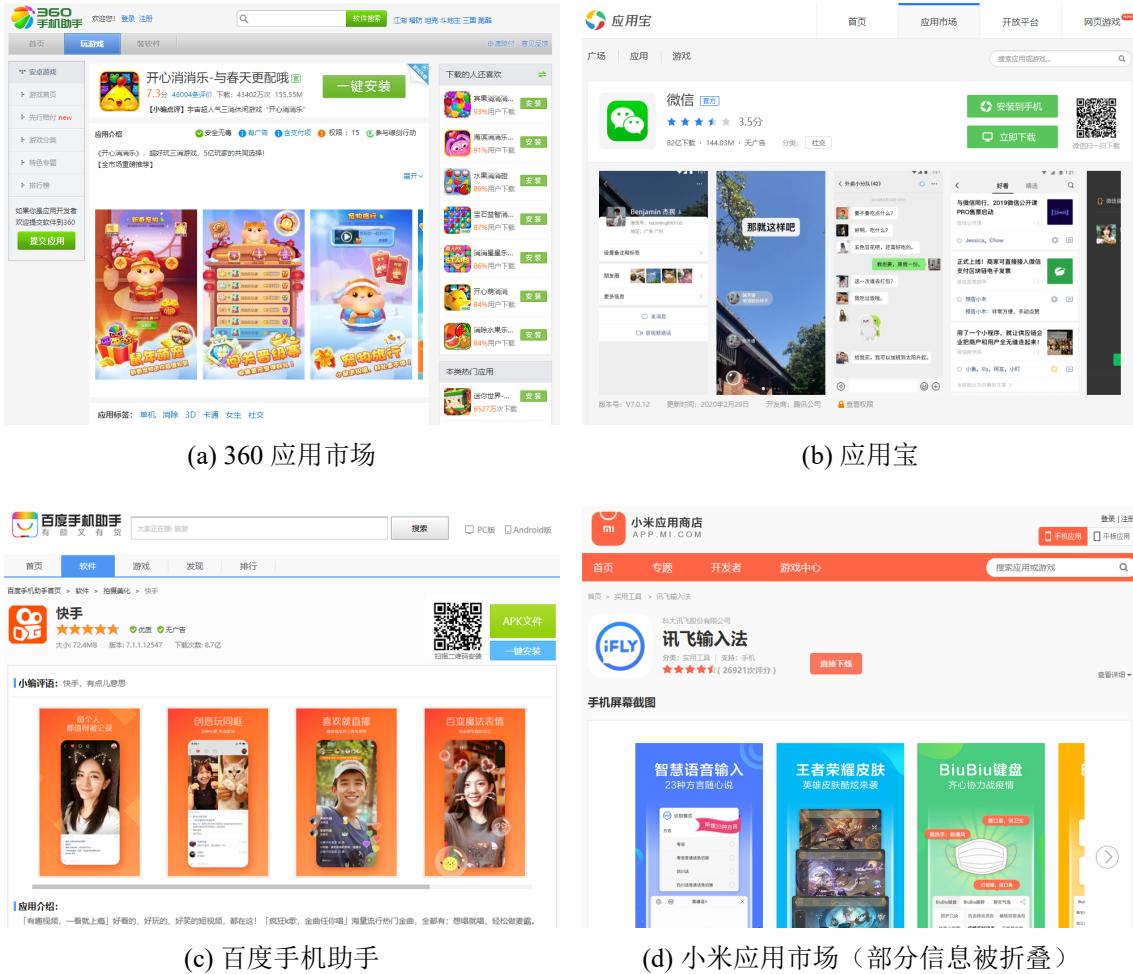


图 4.2: 各大应用市场应用详情页

2) APK 大小通常较小, 很可能为仿冒应用开发者重新开发的恶意应用。

作者认为, 造成上述第一点总结的原因是应用市场上提供的应用信息的不完备和用户对 Android App 了解的缺乏。图 4.2 显示了 4 个国内主流应用市场的详情页。当用户浏览应用详情页时, 各应用市场均显示对应应用的应用名、下载量、应用描述、其他用户对应用的评论和评分等信息。但是, 页面上较为醒目的条目是应用名、应用评分和图标信息, 应用的技术参数(比如应用大小、版本号等)信息或是在不显眼处标记, 或是被折叠, 甚至不被展示。在 4.2d 所示的小米应用市场上, 用户不能直接看到应用大小信息, 需要点开折叠页才能获得相关数据。上述四个应用市场应用详情页均不显示应用的包名信息。由于普通用户不了解 Android App 中包名和应用名的区别和关联, 展示相关信息对市场方引导用户下载安装应用也

没有促进作用，市场对应用的技术参数展示并不重视。因此，用户在应用市场上无法感知应用包名甚至应用大小，仿冒应用开发者无需在该两点上花费精力模仿正版应用。相反，一方面，应用名在市场详情页上处于显眼位置，仿冒应用名称与正版越接近，越容易误导用户下载；另一方面，应用名重名不会在技术上造成阻碍。因此，仿冒应用的应用名与正版应用十分类似，但包名、应用大小与正版应用相距较大。

4.2.2 影响应用被仿冒的严重程度的因素

RQ2：是否存在显著影响应用被仿冒的严重程度的单一因素？

影响应用被仿冒的严重程度的因素众多，如能找到影响应用被仿冒的严重程度的主要因素，可针对性地制订策略防范仿冒应用。根据经验，监管严格的应用市场上，仿冒应用更难通过审核；一个应用的热度越高，越可能被仿冒应用开发者抄袭；不同类别的应用具有不同目标人群，也可能影响仿冒应用开发者对其的态度。因此，作者假设仿冒应用的数目与其来源市场相关，也受原版应用的热度、应用分类等因素影响。App 的更新频率也被视作影响仿冒数量的潜在因素，因为频繁更新的 App 或许可阻碍仿冒应用开发者对其进行仿冒。根据上述因素，分别从应用所在市场与应用自身出发，作者将 RQ2 再细分为以下子问题：

RQ 2.1：仿冒应用主要源于什么市场？仿冒率与市场规模是否有关系？

RQ 2.2：某应用的热度/更新频率/所在类别是否影响其对应仿冒率？

RQ 2.1：仿冒应用主要源于什么市场？仿冒率与市场规模是否有关系？

本研究中搜集的应用样本来源于多个不同的应用市场，不同市场规模不一，其审核、监管力度也并不一致。根据各个样本的来源，有统计结果如图 4.3。

左图显示，在本研究收集数据的所有 29 个应用来源中，从样本总数与仿冒样本数两个角度看，百度手机助手均提供了最多样本。由于本研究选取的目标应用为市面上最受欢迎的应用，较有代表性，可将各渠道提供的样本量与国内市场规模相联系。各个应用市场的仿冒率在右图呈现。

百度手机助手^[66]和安卓市场^[67]的仿冒率均约为 40%，在所有的 29 个渠道中处于中等水平，但由于源于该两个渠道样本基数最大，两者也提供了最多仿冒样本

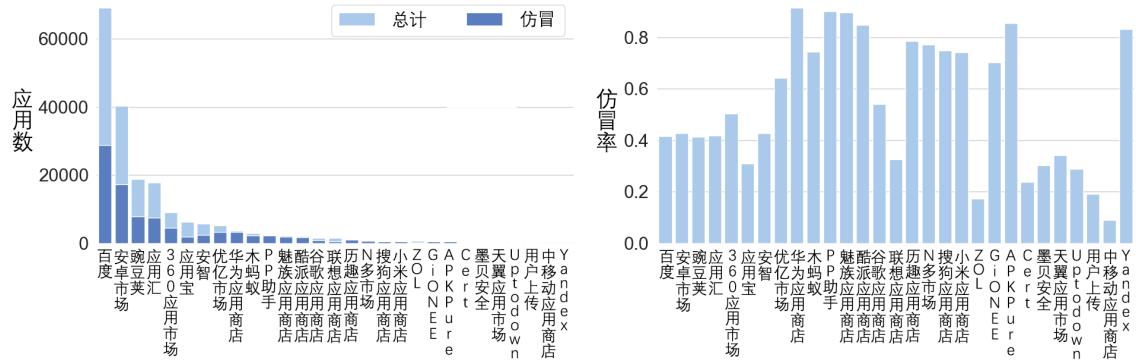


图 4.3: 从不同应用市场中收集到的应用数量以及各市场仿冒率

数。由图中数据可得，应用市场的样本数量与仿冒率不直接相关。然而，应用和市场的关系有可能对应用仿冒率产生影响：本研究的 50 款目标应用中，有 12 款由腾讯公司开发，3 款由 360 开发。针对该 15 款应用进行仿冒率计算，得到各应用于规模较大的 9 个应用市场中仿冒率如下：

表 4.1: 15 款应用于各大市场的仿冒率（数值单位：%）

	百度	安卓市场	豌豆荚	应用汇	360	应用宝	安智	优亿	华为
QQ	54.86	52.73	49.74	46.12	49.45	29.72	26.36	66.55	95.73
微信	89.94	90.47	93.6	89.05	92.34	53.19	94.58	95.0	99.81
腾讯视频	6.25	4.55	13.27	10.84	4.55	0.0	2.0	0.0	66.67
腾讯手机管家	62.9	78.94	72.14	67.29	82.67	56.32	74.74	87.1	98.04
QQ 音乐	5.09	2.04	12.87	5.71	2.63	1.96	0.0	19.35	84.21
开心消消乐	89.66	80.19	87.5	47.89	70.23	63.54	66.67	93.7	96.36
王者荣耀	83.91	78.84	66.19	55.88	77.42	28.92	67.21	93.15	88.89
QQ 邮箱	2.37	1.72	0.0	6.06	0.0	0.0	0.0	15.38	66.67
QQ 浏览器	2.18	2.47	10.78	1.98	6.52	0.0	2.7	15.79	90.0
腾讯新闻	0.82	1.31	0.0	5.56	0.0	0.0	0.0	0.0	0.0
应用宝	39.63	56.12	22.01	28.39	44.68	52.38	92.86	80.0	91.43
全民 K 歌	42.86	43.01	35.78	55.77	60.38	23.81	49.06	35.29	66.67
360 手机卫士	65.92	46.2	79.56	74.78	67.27	91.46	82.05	83.08	96.1
360 清理大师	4.26	6.67	0.0	13.04	0.0	0.0	0.0	9.09	0.0
360 手机助手	33.89	68.0	17.89	45.21	19.48	30.0	37.5	15.79	87.5

腾讯系的 12 款应用分别为 QQ，微信，腾讯视频，腾讯手机管家，QQ 音乐，开心消消乐，王者荣耀，QQ 邮箱，QQ 浏览器，腾讯新闻，应用宝和全民 K 歌；360 系的 3 款应用为 360 手机卫士，360 清理大师和 360 手机助手。应用宝为腾讯旗下

应用市场，表 4.1 中结果表明，应用宝中腾讯系应用仿冒率明显比其他市场低；同样地，360 系应用在 360 市场中的仿冒率也比其他市场中明显降低。作者猜测，应用宝对 12 款腾讯系应用相关的应用有较为严格的审核流程，有效遏制了该市场中与上述 12 款腾讯系应用相关的仿冒应用。同理，360 系应用在 360 市场中仿冒率较低。

结果表明，应用市场的规模与仿冒率无明显关联，但应用本身与市场的关系对仿冒率有影响。

RQ 2.2：某应用的热度/更新频率/所在类别是否影响其对应仿冒率？

通常，某款 App 越受欢迎，其对应的仿冒应用越有可能被用户误下载，令仿冒应用开发者获取利润，吸引更多仿冒应用开发者对其仿冒；应用更新可分为功能性更新与安全性更新两类，频繁更新的应用功能迭代速度快，复杂度高，也可能有更好的安全性，使仿冒开发者难以复刻；不同类别的应用具有不同目标人群与功能，也可能影响对应的应用被仿冒的严重程度。针对上述前两个因素，本节使用皮尔逊积矩相关系数衡量仿冒率与因素对应维度指标的关联性；由于类别无法用连续变量表示，相关系数不适用于该因素，本研究将以图表形式表示两者之间的关联程度。

热度数据源于易观千帆平台；应用更新频率取自某应用最早版本与最新版之间的平均更新时长，即两者发行时间差值与中间版本数的商；根据应用功能划分，本研究的 50 款目标应用被分为 11 个类别，分别为应用市场，摄影录像，游戏，资讯，生活，音乐，移动购物，商务办公，社交网络，系统工具，视频。

表 4.2: 目标应用与其相关统计

应用名	类别	月度热度指数	更新频率(天/版本)	样本总数	仿冒样本数	仿冒率	平均仿冒延迟(天)
微信	社交网络	91.2K	6.4	9248	6447	69.7%	12.1
QQ	社交网络	54.6K	10.7	11167	3780	33.8%	9.2
爱奇艺	视频	53.5K	6.4	7586	3481	45.9%	9.3
支付宝	生活	48.1K	10.2	983	231	23.5%	10.1
淘宝	移动购物	47.5K	7.0	6003	3010	50.1%	8.1
腾讯视频	视频	47.3K	6.3	1429	68	4.8%	10.7
优酷	视频	40.9K	7.3	2058	262	12.7%	6.7

新浪微博	社交网络	39.2K	5.3	5947	2715	45.7%	5.7
WiFi 万能钥匙	系统工具	36.4K	3.1	4808	2999	62.4%	3.0
搜狗输入法	系统工具	33.3K	11.0	898	40	4.5%	21.8
百度	资讯	32.4K	11.1	15651	3514	22.5%	12.8
腾讯新闻	资讯	28.7K	8.5	1051	11	1.0%	8.9
QQ 浏览器	资讯	27.8K	5.6	1369	43	3.1%	11.6
今日头条	资讯	27.4K	4.4	3538	179	5.1%	5.6
应用宝	应用市场	27K	11.4	2419	266	11.0%	11.6
快手	视频	24.4K	3.2	8273	4270	51.6%	3.5
腾讯手机管家	系统工具	24.2K	8.7	2463	1340	54.4%	8.7
高德地图	生活	24K	6.5	1225	51	4.2%	13.1
酷狗音乐	音乐	23K	8.6	1313	122	9.3%	12.2
QQ 音乐	音乐	21.7K	9.4	1132	65	5.7%	14.6
百度地图	生活	21.3K	8.8	2609	1438	55.1%	15.3
抖音	视频	19.4K	11.1	317	12	3.8%	8.3
京东	移动购物	18.5K	10.9	5000	2377	47.5%	12.3
UC 浏览器	资讯	16.7K	7.4	4232	1624	38.4%	7.0
360 手机卫士	系统工具	15.4K	12.4	3670	1423	38.8%	19.1
全民 K 歌	音乐	14.7K	21.1	618	215	34.8%	17.3
美团	生活	13K	8.0	4752	1415	29.8%	6.9
拼多多	移动购物	12.9K	6.6	2327	551	23.7%	7.8
王者荣耀	游戏	12.5K	15.5	2350	1319	56.1%	12.3
美图秀秀	摄影录像	12.4K	5.4	1705	784	46.0%	5.8
火山小视频	视频	12.2K	11.9	410	16	3.9%	9.6
墨迹天气	生活	12K	4.2	10081	7093	70.4%	4.7
滴滴出行	生活	11.8K	8.6	943	117	12.4%	7.0
华为应用市场	应用市场	11.8K	N/A	0	0	0.0%	N/A
开心消消乐	游戏	11.2K	19.7	2406	1738	72.2%	20.6
酷我音乐盒	音乐	11K	2.9	3778	69	1.8%	4.2
西瓜视频	视频	11K	11.5	866	100	11.5%	8.8
OPPO 应用商店	应用市场	10.8K	N/A	0	0	0.0%	N/A
猎豹清理大师	系统工具	9.9K	10.3	1803	388	21.5%	13.5
360 清理大师	系统工具	9.6K	17.3	327	8	2.4%	8.5
360 手机助手	应用市场	9.2K	7.6	1616	137	8.5%	8.4
WiFi 管家	系统工具	8.8K	19.5	1636	658	40.2%	15.7
讯飞输入法	系统工具	8.6K	6.0	1451	8	0.6%	10.1
百度手机助手	应用市场	8.2K	11.4	3849	437	11.4%	14.5
小米应用市场	应用市场	7.8K	N/A	0	0	0.0%	N/A
WPS Office	商务办公	7.4K	6.0	1152	69	6.0%	7.8
美颜相机	摄影录像	7.1K	5.3	1600	691	43.2%	6.3

网易云音乐	音乐	7K	10.5	616	6	1.0%	12.2
网易新闻	资讯	6.7K	7.0	1441	93	6.5%	5.0
QQ 邮箱	商务办公	6.6K	16.4	520	11	2.1%	10.4

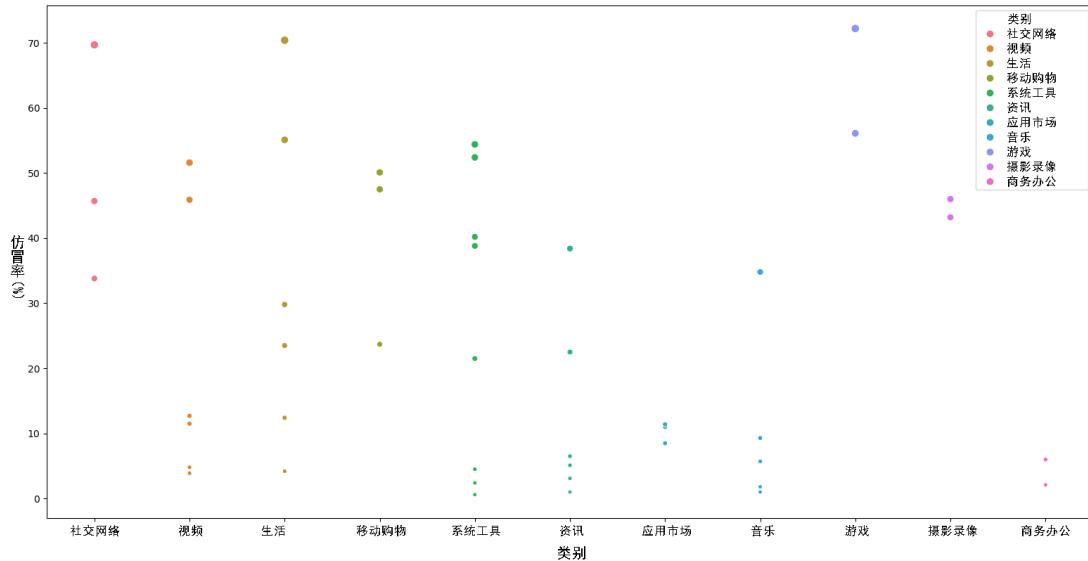


图 4.4: 目标应用类别-仿冒率对应图

表 4.2 按每款 App 的热度排序，展示了每款目标应用的类别、仿冒率、更新频率、关联样本总数等数据。将数据代入式 4.3 可分别获得应用热度、更新频率与仿冒率之间的关联程度。图 4.4 显示每款目标应用所在类别及其仿冒率的对应关系，如 x 轴社交网络上对应的三个粉色圆点为三个社交网络类别下目标应用的仿冒率。

结果表示，仿冒率与应用热度之间的相关系数为 0.246，处于较弱水平；更新频率和仿冒率之间相关系数为 0.084，两者之间几乎没有关联；图 4.4 中，除应用市场、商务办公、摄影录像、游戏四类仿冒率相对集中外，其他各类别对应应用的仿冒率较为分散。

综上，应用热度、更新频率与应用类别都不是影响应用是否会被仿冒的决定性因素，由上述结果可得：1) 应用被仿冒的严重程度并非由单一因素决定，未来的分析应尝试从多角度同时入手分析；2) 应用更新频率与被仿冒的严重程度几乎不相关，进一步巩固了上一节的结论，即数据集中的大部分仿冒样本都不是重打包应用，而是仿冒应用开发者自行开发的。无论官方版本受到的保护程度如何，仿冒应用开发者都可以制作对应的仿冒应用；3) 某些应用类别下的应用仿冒率相

对集中，表示该类应用的确更受（或更不受）仿冒应用开发者喜爱，后续探索可从应用市场、商务办公、摄影录像、游戏四类类别入手。

4.2.3 仿冒应用功能与行为

RQ3：仿冒应用作者制作出了怎样的仿冒应用？是否依然能提供原版应用的功能？

出于性能考虑，本研究并未对每个仿冒 APK 包进行详尽的拆包解析，为了解仿冒应用在功能、行为上与原版应用的差异，本研究采用案例分析的实证研究方法进行研究。由图 4.4 可知，游戏类应用（王者荣耀和开心消消乐）吸引了大批的仿冒应用样本。本研究随机从这两款游戏应用的仿冒样本中各选择了 7 个仿冒样本与 1 个正版样本，将样本安装至实验设备上运行。

测试使用的设备为高配版小米 5 手机，搭载的 CPU 为最高主频 2.15GHz 的骁龙 820 处理器，3GB 内存，64GB 机身存储，安装的 Android 系统版本为 Android 6.0 (Android Marshmallow, API 23)。4.5a 展示了这些样本在真实的 Android 设备上安装之后的实际外观。官方渠道下载的正版 App 在图片中由绿色边框标记出。可见仿冒应用的应用名与原版应用名十分相似，符合前文发现；仿冒应用的图标设计也与原版雷同。本研究在测试设备上实际运行了上述安装的 14 个仿冒样本，将仿冒应用的界面、行为与原版应用对比，同时将样本上传至 VIRUSTOTAL^[68] 进行恶意行为扫描。

4.5b 和 4.5c 分别是在测试设备上运行官方版本的开心消消乐和其中一个仿冒版的开心消消乐时的系统截屏。从 UI 布局上看，两款应用的外观十分相像。同时，仿冒版的确实现了完整的游戏功能，与正版的玩法、实际操作逻辑一模一样，有较强迷惑性。应用开发者未必可判别两个应用的真伪，应用市场的普通用户更难以区分两者。

7 款开心消消乐的仿冒样本中，4 款与官方样本十分相似（其中之一可能是经过重打包技术处理的应用），2 款声称自己是“系统攻略”，1 款在运行时闪退，无法在测试设备上实际运行。在 4 款仿冒游戏中，3 款都在游戏中不时自动弹出游戏内购窗口，要求玩家购买道具，十分可能导致玩家不想要的花费。所有 7 个仿冒样本都在 VIRUSTOTAL 中被报告为恶意应用。



图 4.5: 游戏类 App 及其仿冒样本

另一方面，王者荣耀的仿冒样本功能截然不同。7 款仿冒样本中，3 款是壁纸浏览器，内含游戏人物插画，可在应用内将插画设置为系统壁纸；余下 4 款是简单的拼图游戏，同样包含游戏人物的插画，应用内容为把被打乱的插画拼图恢复原状。VIRUSTOTAL 显示，7 款仿冒样本中，有 6 款是恶意软件，涵盖了木马病毒、广告软件等类型，而余下的一款则被报告为 PUP (Potentially Unwanted Program, 潜在有害程序)。PUP 通常在用户不知情或者不愿意的情况下，通过静默安装或者捆绑安装的形式被安装在系统中。该类软件不一定包含恶意代码，但动机存疑。

造成该现象的原因为模仿正版应用功能难易程度不一。从技术角度看，王者荣耀的设计、实现难度均比单机益智类游戏如开心消消乐高许多。多人在线竞技游戏物理引擎之外，还要实现聊天系统、在线匹配、负载均衡等业务模块；相比之下，益智类游戏的核心逻辑较为简单，无需设计多个复杂模块。

因此，即使不考虑维护问题，开发复杂游戏的成本对仿冒应用开发者而言也明显过高。但由于王者荣耀的高热度可能带来巨大收益，仿冒应用开发者会为了蹭上热度而开发外观相似、内容完全不符的仿冒样本。相比之下，开心消消乐由于开发难度相对较小，仿冒应用开发者可开发一个内容相似的应用，再通过内购陷阱等

手段收取效益。这两款应用的仿冒样本透露出了仿冒应用开发者在仿冒方面两个截然不同的思路。

结合前文结果分析，第 4.2.2 节中显示商务办公类别仿冒率较低也可能是类似原因导致的结果。一方面，商务办公类的工具核心逻辑比较复杂，仿冒应用开发者不会制作与其功能接近的仿冒应用；另一方面，该类应用也不像游戏类应用可衍生出周边产品（如王者荣耀的游戏人物插画），仿冒应用开发者无法从周边入手蹭热度。结合两个原因，商务办公类应用相对不受仿冒应用开发者欢迎。

4.2.4 研究结果有效性分析

结构有效性已于第 4.1 节解释，不再赘述。内部有效性方面，一个可能的威胁是应用与市场之间的关系对该应用在市场上仿冒应用数量的影响。本章在研究中提出了该可能性，并对相关应用单独进行了进一步检查以消去该因素可能带来的干扰。外部有效性威胁主要在于实验中仅使用了 50 款目标应用作为数据，结论可能存在偏差。的确，50 款应用并不能代表 Android 应用生态的全貌，但市面上的 Android 应用无穷无尽，收集全量应用进行分析并不可行。本研究采用的 50 款应用为国内最受欢迎的 50 款应用，分别源于 11 个类别，有一定市场影响力与较大用户群体，本身具有一定代表性与普遍性。在衡量实验开销与结果有效性后，作者认为选用该组数据进行分析是可行的。

4.3 相关工作

仿冒应用在移动应用行业中早已存在，但少有学者关注仿冒应用的特性。在已知文献中，与本章研究较为类似的是 Khanmohammadi 等人对重打包应用进行的实证研究^[6]，文献中采用了 2,776 款原版应用和与其对应的 15,296 个重打包样本作为实验数据，来源为 AndroZoo 数据集^[7]。AndroZoo 数据集应用来源是以 Google Play 市场为主的 18 个数据源，其中包含 4 个国内市场。一方面，本研究的数据来源共有 29 个，以国内市场为主，可更好地反映国内市场现状。另一方面，本研究虽然只选取了 50 款原版应用作为对比目标，但收集到的对应仿冒样本有 62,638 个，对比该文献，每款原版应用都有更多仿冒样本作为研究对象提供数

据。因此，作者认为本文的分析结果更为准确。再者，该文献仅对重打包应用进行探索，忽视了仿冒应用开发者重新开发相似应用的可能性。结合前文结果可知，重打包应用在仿冒应用中占比并不高。从仿冒应用研究角度看，本文提供的结果更具参考性。

4.4 本章小结与实用建议

本章从三个不同角度对仿冒应用的基本特征进行了实证研究，研究结果可概括如下：从相似度角度分析，由于仿冒应用开发者目的为诱导用户从应用市场中下载仿冒应用，在用户可感知的方面（应用名），仿冒应用与原版应用相似度较高；在用户较难感知的方面（应用包名与 APK 大小），仿冒应用与原版应用相差较大。从影响应用受仿冒程度角度分析，从应用市场规模并不能与应用商店的仿冒率相对应，仿冒应用普遍存在，但应用本身与市场的关系会影响市场中对应的该应用仿冒样本数量；同时，应用的更新频率、热度、类别三个因素均不对应用受仿冒的严重程度起决定性影响，说明应用受仿冒的严重程度是较为复杂的、多因素共同作用的结果。从仿冒应用可提供功能的角度分析，研究利用案例分析方法，从游戏类别的仿冒应用入手，揭示了仿冒应用开发者对不同应用会采用不同仿冒策略。对结果进行推断，可以得出“仿冒应用作者更倾向重新制作仿冒应用，而非通过重打包方式制作仿冒应用”的结论。

针对本章实证研究过程的发现，有三点实用建议如下：1) 应用市场方应加强审核流程，除审核应用内容外，也应从外观、应用名、应用内 UI 等方面对应用进行审核，从源头抵御仿冒应用，保护原版应用作者利益；基于案例分析可得，仿冒应用包含恶意行为的可能性较大，应用市场方可对该方面加强管理，拦截仿冒应用。2) 应用开发者可更积极地与应用市场方联系，督促应用市场对自身应用的监管。3) 普通用户在搜索、下载应用时，应小心从搜索结果中选择，可结合其他信息（如应用截图、开发者信息）对搜索结果进行判断，避免下载仿冒应用；也可参考本研究获取正版样本的方法，直接从应用官网下载原版应用安装。

第五章 仿冒开发者行为特征实证分析

本章针对仿冒应用开发者的行为特征进行实证研究，了解该类行为有助于了解应用市场现有监管机制是否存在缺陷，从而可针对性地提出改善措施。研究内容主要为分析仿冒应用与开发者的对应关系、仿冒应用证书活跃期、仿冒应用上架速度等与开发者行为相关的方面。具体分析从三部分展开，结合应用证书视角与结合时间维度视角进行探索，对应以下三个研究问题：

- RQ4：仿冒应用与开发者的对应关系如何？
- RQ5：仿冒应用证书可以活跃多长时间？
- RQ6：仿冒应用的上架行为是否有特征？

5.1 研究方法

本节介绍本章研究中整理研究数据的方法，以讨论研究的结构有效性。对 RQ4，根据 第 2.2.1 节的介绍，可利用样本的应用证书构建其与应用开发者的对应关系，RQ5 亦涉及应用证书相关信息使用，因此需要阐明应用证书数据的获取方式；同时，RQ5 有对应用证书活跃时间的需求，本章研究引入了一个应用证书活跃时间的近似计算方法；对 RQ6，为研究仿冒引用的上架特征，本研究整理复原了各款正版应用各版本上架的时间线，本节将介绍时间线的整理方法。本章研究对象为与上一章相同、由 69,414 个正版样本与 52,638 个仿冒样本组成的应用集。

由 第 2.2.1 节，可知每个 APK 文件内都有包含应用开发者信息的 *CERT.RSA* 文件。每个仿冒样本，作者将其解压，取出/META-INF 文件夹中的应用证书文件，再利用 Keytool^[69] 工具获取证书信息（如应用证书 *SHA1* 码与应用开发者名）。Keytool 工具为 Java 自带工具，被广泛地用于管理密钥和证书，因此可相信使用 Keytool 获取的应用证书信息无误。

在获取应用证书活跃度方面，由于 Janus 类似一个爬取各大市场应用的平台，并不能感知某样本的下架时间或某样本是否已下架，因此只能对仿冒应用证书的活跃时间作近似计算。计算的具体方法为将所有仿冒样本遍历一次，取出样本的应用证书 SHA1 码，凭借该样本的搜集时间更新该应用证书的最早出现时间（或最晚出现时间），取各证书最早出现时间与最晚出现时间为证书的活跃期，活跃期长度则为该证书的活跃时长。

为了解仿冒应用的上架速度，研究需要获取各款目标应用每个版本的更新时间和与该版本对应的仿冒样本的上架时间。各个目标应用多年来发布版本众多，已无法从大部分应用官网中寻回完整更新日志，因此本研究借助样本集中的正版样本搜集时间整理复原各版原版应用的上架时间线。本文对所有收集到的正版样本按目标应用进行分类，再按照版本号对同一目标应用下所有样本排序。由于同一应用可在多个市场上架，同一版应用在不同市场可能存在细小差别（如在华为市场上架的应用需要接入华为钱包作为支付途径），导致同一版本的 APK 安装包具有不同 SHA1 码。因此，此处以应用版本而非应用 SHA1 码为粒度分割样本，以消除上述情况影响。另外，基于各市场审核流程、其他商业因素考虑，可能存在应用同一版本在不同市场有不同上架时间的情况。针对该情况，如某版原版样本存在多个上架时间，本研究只取最小值，该值表示仿冒应用开发者可接触该版原版应用的最早时间。之后，将每款应用中每个版本上架时间串联，即可大致重现每款应用的更新时间线。

从仿冒样本角度考虑，仿冒样本大多不是重打包应用，无法与原版应用的版本号直接对应。为求出仿冒样本的仿冒延迟（即原版应用上架时间与仿冒样本上架的时间差），本文提取出每个仿冒样本的上架时间，以不晚于这个仿冒样本发布的原版应用作为其对应的仿冒对象，取两者上架时间差作为仿冒延迟。

5.2 实证研究流程与结果解析

本节分为四个部分，前三个部分分别对应本章三个研究问题的研究流程，最后一部分对本章研究进行有效性分析。

5.2.1 应用证书与仿冒应用对应关系

RQ4：仿冒应用与开发者的对应关系如何？

由于每个仿冒样本均有对应的应用证书，本研究构造了仿冒样本与应用证书的二分图，其中两类节点分别为仿冒样本和应用证书，若两节点之间存在边，则表示该样本由该证书签署。在仿冒应用持有的所有应用证书中，多数证书（76%）仅仅关联了一个或两个仿冒样本，与最多仿冒应用的证书签署了 1,374 个仿冒样本。表 5.1 统计了应用证书和他们对应的仿冒样本数量。表格第一栏为仿冒样本的数量区间，第二栏为关联的仿冒样本数量该落于区间的应用证书数，如第一列数据指有 8252 个应用证书签署的样本数为 1 到 5 个。大部分应用证书都只关联了 1 到 5 个应用样本，但也有少量应用证书与大量仿冒样本有关联关系。

表 5.1: 应用证书/仿冒应用数量对应表

关联仿冒样本数量	1-5	6-10	11-50	51-100	大于 100
应用证书数量	8252	525	531	71	80

鉴于应用证书与应用开发者存在多对一关系（第 2.2.1 节 Android App 签名机制），作者认为每个应用证书只签署少量样本是仿冒应用开发者规避应用市场监管机制的策略。如果仿冒应用开发者只使用一个应用证书上传多个仿冒应用，万一其中一个应用被投诉下架，使用同一证书签署的其他的应用很可能会受到牵连。但一个仿冒应用开发者可持有多个应用证书，即仿冒应用开发者可利用多个身份在应用市场中上传应用，继而令应用市场就难以找到仿冒应用之间的关联。即使其中一个证书签署的仿冒应用被举报下架，由其他证书签署的余下应用也得以被保全。

另一方面，有部分仿冒应用应用证书与多个仿冒样本关联。其中，SHA1 码为“*61ed377e85d386a8dfee6b864bd85b0bfaa5af81*”的应用证书是所有证书中关联仿冒样本最多的，有 1,374 个样本由该应用证书签署，样本涵盖了本研究数据集 47 个目标应用中的 37 个（79%）。根据 Keytool 从该证书抽取出的开发者信息，其开发者名为“Android”。Android 官方文档^[70]显示，该证书为 Android Studio 自带的 Android 调试证书。此结果表明国内大部分应用市场在审核阶段并未对该类证书进

行有效拦截，开发者可凭借该类证书将仿冒应用上传至市场中，增加用户风险。

进一步，作者抽取所有证书的开发者名称进行匹配，得到表 5.2。与下文表 5.1 相似，表格第一栏为某开发者名对应应用证书的数量区间，第二栏为对应应用证书数量该落于区间的开发者名称数，如第一列数据指有 5355 个开发者名称仅与 1 个应用证书对应。

表 5.2: 应用证书数量/开发者名称数量对应表

对应应用证书数量	1	2-5	6-10	11-50	大于 50
开发者名称数量	5355	419	38	23	9

对应证书最多的前 10 个开发者名称与其对应证书数量依次为：*cn* (873), *Android Debug* (441), (空字符串) (279), *Unknown* (207), *mobcent* (195), *addone* (125), *Java* (102), *Android* (63), *I* (56), *a* (43)，其中 *Android Debug*、*Android* 为 Android 调试证书的开发者名，*Java* 很可能为利用 Keytool 生成证书的默认开发者名，空字符串、*cn* 与 *Unknown* 为其他工具生成证书的默认开发者名（CN 对应生成证书时需输入的开发者昵称，Common Name），*I*、*a* 也不是有意义的开发者名称。因此，除 *addone* 与 *mobcent* 两个开发者名称有具体含义外，余下 8 个开发者名称无明显意义，即其对应的应用证书很可能为调试证书。

表 5.3: 开发者名称/仿冒样本数量对应表（关联数前十位）

开发者名称	对应仿冒样本数量
<i>cn</i>	1888
<i>Android</i>	1661
<i>Android Debug</i>	1285
(空字符串)	1278
<i>zh</i>	1176
http://soft.sj.91.com	1033
<i>yk</i>	982
<i>Gavin Van</i>	698
<i>Unknown</i>	653
<i>android</i>	560

将开发者名称–应用证书–仿冒样本关联，可得到开发者名称与其对应的仿冒

样本数量。与最多样本关联的前 10 个开发者名称如表 5.3 所示。综合表 5.3、表 5.1，可确定国内各大应用市场并未对调试证书签署的应用进行拦截，从而产生十分大的安全风险。此外，虽然表 5.2 显示多数开发者名称与应用证书呈一对关系，但该现象不与前文推论（仿冒应用开发者利用多个证书发布仿冒应用，每个证书只对应少量样本）矛盾，因为开发者在生成证书时可随意编写开发者名称，开发者名称与仿冒应用开发者之间对应关系并不明确。相反，表 5.2 中的 *addone* 与 *mobcent* 两个开发者名称分别对应证书 125 个和 195 个，对应样本数则分别为 149 个和 263 个，可为上述推论提供数据支持。

5.2.2 仿冒应用证书活跃度

RQ5：仿冒应用证书可以活跃多长时间？

仿冒应用证书活跃时间可用于评估应用市场防御机制强度。假定仿冒应用开发者会利用同一应用证书不断上传仿冒应用，证书的活跃时间越短代表市场方封禁处理越及时；相反，证书的活跃时间越长代表市场方检测、处理速度越慢。

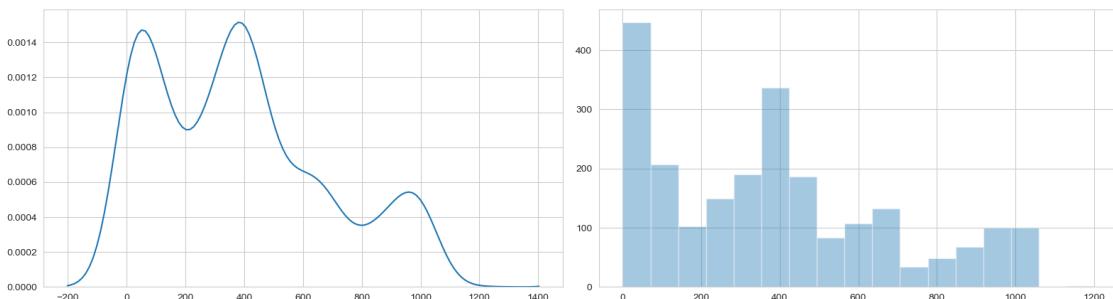


图 5.1：仿冒应用应用证书活跃时间分布

按第 5.1 节所述方法计算证书活跃时间，图 5.1 展示了不同仿冒应用应用证书在应用市场里活跃时间的总体分布。考虑到上节结果显示 76% 应用证书仅关联一至两个仿冒样本，对活跃时间计算有较大干扰（仅发布一个样本的证书活跃天数为 0 天），图中数据去除了该部分数据。左图为概率密度分布图，*x* 轴表示活跃时长，*y* 轴表示了与 *x* 轴上的值对应的概率密度。右图为数量分布直方图，*x* 轴表示活跃时长，*y* 轴对应区间内证书数量。

经计算，图 5.1 左图中 *x* 轴值大于 109 对应的 *y* 轴曲线下方区域面积接近 0.75，

表示约 75% 仿冒应用应用证书活跃时间大于 109 天，数据的 50 分位数与 75 分位数分别为 366 与 579.5，表示 50% 仿冒应用证书活跃时间超过一年，约 25% 仿冒证书活跃时间超过 580 日。前文提及的证书“`61ed377e85d386a8dfee6b864bd85b0bfaa5af81`”也是活跃市场最长的应用证书。本研究采用的仿冒样本上架时间范围为 2015 年 5 月 29 日至 2018 年 9 月 15 日之间，使用上述证书签署的样本最早于 2015 年 11 月 12 日在百度手机助手上线，最晚于 2018 年 9 月 15 日在百度手机助手上线。2015 年至 2018 年间，该证书一直处于活跃状态，29 个渠道中，除 Uptodown，天翼应用市场、Apkpure、GiONEE、中移动应用商店、Google Play 应用商店 6 个应用市场中未有发现由该证书签署的样本外，其他 23 个应用来源均有发现与该证书关联的样本。然而，三年期间发布 1,374 个样本是有违常理的，以一年 365 天计算，需在每 1.25 日开发、上线一个新应用样本，该开发速度与日常认知并不一致。而且，该证书为 Android Studio 内置调试证书，因此作者认为证书对应的 1,374 个样本并非由同一个人或组织上传。进一步推断，各市场除了接收应用开发者上传的应用外，还会自行搜集应用上线，但在搜集过程中并未制订严格安全规范，扩大了安全风险。

以上结果进一步表明国内应用市场防御机制存在较大缺陷，国内应用市场的利益未能被有效保障。

5.2.3 仿冒应用上架特征分析

RQ6：仿冒应用的上架是否有特征？

仿冒应用上架特征可分为不同角度观察：从宏观角度，仿冒应用上架数是否有一定趋势？从微观角度看，仿冒应用开发者上架应用时是否有偏好？本节研究将 RQ6 细分为以下子问题：

RQ 6.1：仿冒应用总体上架量是否呈现一定趋势？

RQ 6.2：仿冒应用开发者是否会同时在多个市场上架应用？

RQ 6.3：仿冒应用开发者偏向在新应用版本推出多久后发布仿冒应用？

RQ 6.1：仿冒应用总体上架量是否呈现一定趋势？

5.2a 展现了自 2015 年 11 月起，仿冒应用开发者每个月在 29 个数据来源中上架的仿冒样本数量， x 轴代表月份， y 轴表示某月对应仿冒样本数量。数据表明，

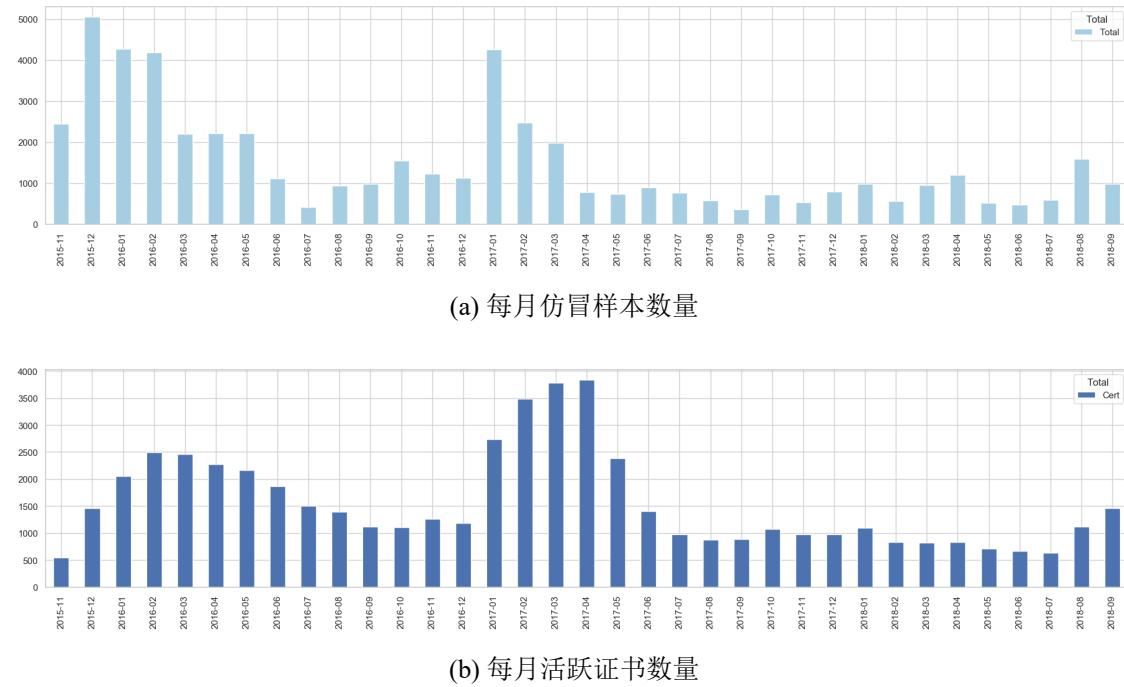


图 5.2: 每月仿冒样本数量、应用证书数量 (2015 年 11 月至 2018 年 9 月)

2015 年年末、2016 年第一、第二季度与 2017 年第一季度间，有大量仿冒应用被上架到市场中，其他时段仿冒应用上架量较为稳定，约为每月 1000 个，仿冒应用上架量有逐年递减的趋势。5.2b 则为同时期在三个月内发布过仿冒样本的应用证书数量。2016 年与 2017 年第一、二季度为仿冒应用开发者最活跃的阶段，其后仿冒应用开发者数量骤减并趋向稳定，从 2018 年第三季度开始有复苏迹象。结合两图，作者认为仿冒应用高峰已过，但其威胁依然存在，各大应用市场方仍需加强审查保障用户安全。

RQ 6.2: 仿冒应用开发者是否会同时在多个市场上架应用？

表 5.4: 仿冒应用证书多市场上架情况统计

同时上架市场数量	1	2	3	4	5	6	7	8	9	大于等于 10
应用证书数量	5063	2368	983	488	237	130	91	45	24	30
占总量比例 (%)	53.53	25.03	10.39	5.16	2.51	1.37	0.96	0.48	0.25	0.32

作者对所有仿冒样本的上架情况与其应用证书关联，得到如表 5.4 所示的仿冒应用证书多市场上架情况。逾半数仿冒证书（53.53%）仅会发布在一个市场中，近

半数仿冒应用证书对应的应用会发布在多个市场。结果表明，仿冒应用开发者会在多个应用市场中上架仿冒应用，各应用市场之间并未采取策略应对此类行为。

RQ 6.2：仿冒应用开发者偏向在新应用版本推出多久后发布仿冒应用？

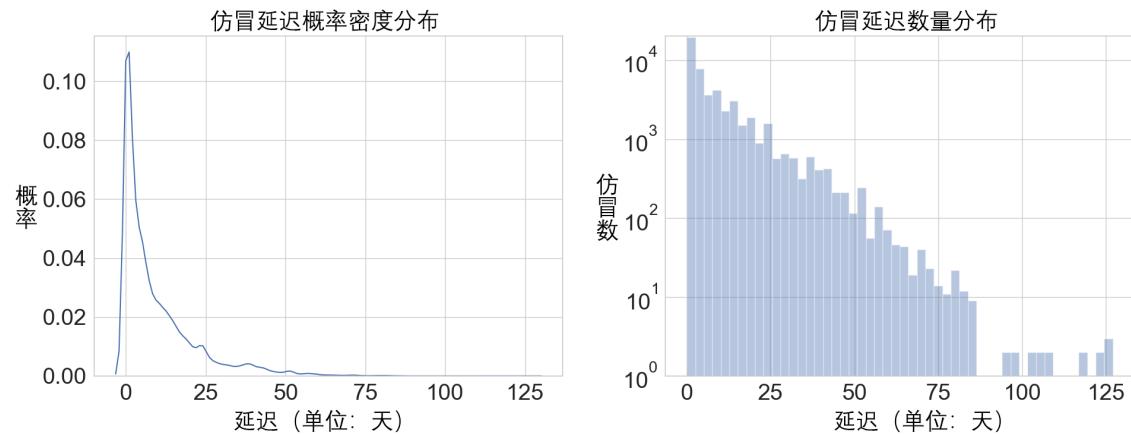


图 5.3: 仿冒延迟总体分布

本研究按第 5.1 节中的方法计算了每版应用被仿冒的延迟时间和延迟的分布情况，结果显示如图 5.3。绝大多数（超过 90%）仿冒样本在官方新版推出后的 20 天内被发布，60% 仿冒样本在正版被推出的 6 天内发布，该结果表明仿冒开发者的行动十分迅速。由于个人开发者的开发速度相对缓慢，作者推测制作仿冒应用已在移动黑灰产业中已规模化、组织化，以在新版应用推出后迅速推出对应的仿冒应用。同时，根据第四章结论，多数仿冒应用为重新开发的简单应用，而非重打包版本。无需对原版应用进行破解、重打包也可加速仿冒应用开发流程，有利于仿冒应用开发者快速完成新版仿冒应用。

5.2.4 研究结果有效性分析

结构有效性已在上一节讨论过，不再赘述。

内部有效性威胁在于研究未能将仿冒应用证书与仿冒应用开发者精确匹配。应用证书与开发者之间存在多对一关系，开发者创建证书时可输入名称等信息，但该信息不足以唯一定位开发者。本章研究利用证书中的开发者名称信息匹配证书与开发者，此方式有效性有限（仿冒应用开发者可在创建不同证书时输入不同虚假信息），但有一定效果（如 *addone* 与 *mobcent* 两个开发者名称分别对应证书 125

个和 195 个，为仿冒应用开发者持有多个证书的论断提供数据支持）。另外，上述多对一关系对 RQ6.1 研究的准确性也产生影响，每月实际活跃的仿冒应用开发者数量低于 5.2b 所示的每月活跃证书数量。然而，因数据显示仿冒应用证书活跃数下降趋势较为明显，5.2a 也呈现仿冒样本骤减现象，可认为 RQ6.1 的研究结果依然有效。

外部有效性威胁源于两方面：其一，由于本章研究的目标应用为最受欢迎的 50 款 Android 应用对应的仿冒应用，未包含较小众的 Android 应用，未能探明针对小众 Android 应用的仿冒应用开发者具备何种行为特征；其二，因开发者可任意发布 Android 应用程序，为降低收集、整理样本的成本，研究数据来源大部分源于国内应用市场，研究结果未必适用于整个互联网环境。因此，尽管样本数据较丰富、来源多样，本章研究的结论普遍性仍未完善。作者将在未来工作中对两方面入手进行更多研究，进一步提升研究普适性。

5.3 相关工作

与本节工作相关的研究分为两类，一类的研究方法与本章类似，另一类的研究目的与本章类似。

Zhang 等人针对 Android 应用中的第三方库进行实证研究^[71]，其中采用了与本文类似的研究方法，探究应用作者与潜在恶意第三方库（potentially malicious libraries, PMLs）的关联。该文献提取应用中的证书信息，与应用中的 PML 进行配对，寻找与 PML 存在一一对多关系的应用证书，并推断某些流行的 PML 可能由同一开发者制作。在探究 PML 与应用证书的关联时，Zhang 等人也发现了其应用样本包含开发者名为空或为 Android Debug 的证书。由于该类证书无法提供更多开发者信息，Zhang 等人将携带该类证书的应用从研究中排除。与之相比，本研究重点为通过分析仿冒应用作者的行为揭示应用市场现存的监管漏洞，此类证书的出现表示应用市场在应用证书审核流程仍有缺陷，与研究方向相符，因此本章研究并未筛去此类证书及相关样本。

与本文研究目的类似的工作有 Zhang 等人针对 Android 恶意应用的实证研究^[5]。该文献收集了源于 49 个家族的 1260 个 Android 恶意应用，结合数据统计与

样例分析，总结出与恶意应用相关的领域知识，如恶意应用常通过重打包、更新攻击和路过式下载（Drive-by Download）三种方式进行传播。更新攻击指恶意应用开发者为了躲避应用市场的监管审查，有意在应用市场上上架不包含恶意代码的应用，然后在用户安装应用之后，提示用户升级，进而绕开应用市场，将带有恶意代码的新版本安装到用户设备上的行为。路过式下载则指用户已安装的恶意软件，在用户不知晓的情况下，静默下载和安装更多恶意软件的行为。通过揭示恶意应用的传播方式、恶意行为等信息，该研究揭示了改良已有恶意应用检测方法的重要性与优化方向。类似地，本文通过对大量仿冒应用数据分析挖掘，点明现阶段应用市场的监管机制、策略缺陷，也可为应用市场方提供风险管控参考方向。

5.4 本章小结与实用建议

本章分别从仿冒应用与开发者的对应关系、仿冒应用证书活跃时间与仿冒应用上架特征三个不同视角对数据进行分析，结果可总结如下：从仿冒应用与开发者的对应关系看，一个仿冒应用作者可持有多个应用证书，每个证书仅用于上架少量应用，这可被视作绕开应用市场监管机制采取的策略；同时，部分仿冒应用作者采用 Android 调试证书在应用市场上架仿冒应用，但应用市场并未拦截此类行为。从仿冒应用证书活跃时间看，去除仅上架了少量应用的证书后，数据显示仿冒应用证书能在较长时间保持活跃。该结果表示，国内应用市场未能有效发现、处理上架了仿冒应用的开发者，监管机制存在较大缺陷。从仿冒应用上架特征角度看，数据表示仿冒应用行业在 2016、2017 年发展迅速，有大量仿冒应用上架、大量证书活跃。行业热度在 2018 年有回落趋势，但在 2018 年第三季度起有复苏迹象；数据显示近半数仿冒应用于多个应用市场中上架，表示各应用市场间未有交流机制以共同抵御仿冒应用；仿冒应用可在正版应用新版发布后迅速推出，表示仿冒应用产业链已较为成熟。

针对本章实证研究结果，对应用市场方有三点实用建议如下：1) 应用市场在审核开发者资质时，应加强对应用证书的检验，阻止开发者利用调试证书上传应用，并要求开发者上传具备完善、真实开发者信息的证书；2) 为对抗仿冒应用威胁，应用市场应完善监管机制，加快对仿冒应用开发者的处理，禁止曾上传过仿冒

应用的开发者账号利用同一证书再上传应用；3) 恶意开发者/可疑开发者信息互通平台缺失，导致仿冒应用开发者可在多个应用市场上架仿冒应用，各应用市场应尽快合作建立类似平台，联合监管国内 Android 生态环境。

第六章 仿冒应用检测框架 FakeRevealer

前两章分别从仿冒应用基本特征与仿冒应用开发者的行为特征入手，对仿冒应用开展实证研究，获取了仿冒应用命名、大小、投放偏好等特性，也说明了当下的应用市场并未对仿冒应用进行有效拦截。基于实证研究的发现，作者设计了仿冒应用检测框架 FakeRevealer。通过使用 FakeRevealer，应用市场方可在大规模应用中实现对已知正版应用及其对应仿冒样本的快速鉴别，有效检查出开发者上传的仿冒或正版应用，提高应用市场方的审核速度。本章将先阐述 FakeRevealer 的设计与实现，再进行系统实验并解析实验结果。

6.1 框架设计与实现

6.1.1 整体设计

FakeRevealer 是一个轻量的仿冒应用检测框架，基于已知的正版特征对输入的应用集进行自定义筛选。应用市场方可将其用于大规模的 Android 应用检测，自动筛选出其中的正版应用与潜在仿冒应用，再结合人工审查，完成对仿冒应用的验证、归类和对可疑行为的提取，感知潜在的威胁。

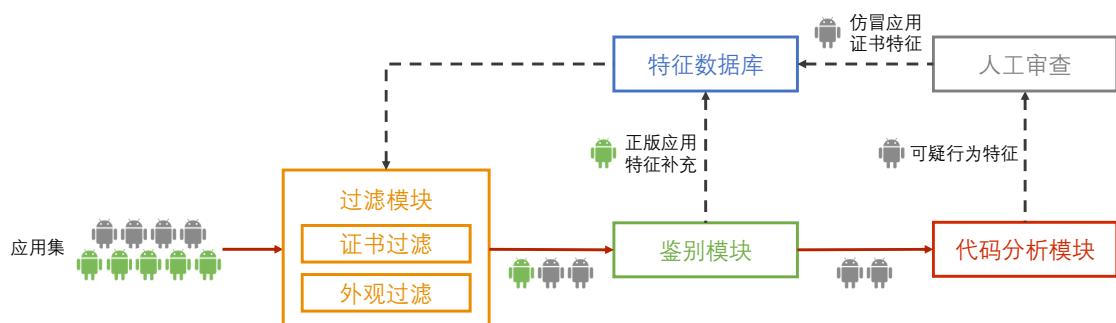


图 6.1: FakeRevealer 整体流程

图 6.1展示了 FakeRevealer 的整体工作流程，框架主要分为过滤模块、鉴别模块、代码分析模块、人工审查和应用特征库五个部分。应用特征库为存储应用

特征的数据库，用户在使用前，需要先在应用特征库中输入部分正版应用特征信息和其他相关信息（如开发者黑名单）作为先验知识。FakeRevealer 以 Android 应用集为输入，先利用过滤模块对应用集进行初筛，过滤与正版应用不相关或相似度低的应用，降低后续检测与人工审核的压力。之后，鉴别模块模块提取应用样本的证书信息进行仿冒鉴别。被判别为正版的应用样本的外观特征将被提取进应用特征库中以加强后续检测；被判别为仿冒的应用样本则会进入代码分析模块被提取行为信息。代码分析模块提取应用中的静态分析数据（如应用代码中的方法信息、类信息等），构造方法调用图，分析应用的行为信息；此外，应用样本的基本信息也会被提取，方便后续展示。之后，代码分析模块分析涉及敏感 API 调用（如打电话、收发短信等）的应用行为，其中的可疑行为会被记录，汇总后分发到人工审查环节复核。Android 应用面对的业务场景是十分复杂的，类似的方法调用行为在不同业务场景下会有截然不同的含义，因此现阶段仍需人工审查确认应用的 API 调用是否可疑或具有恶意。结合框架给出的应用信息和方法调用关系，应用市场审核人员可对仿冒应用进行快速判别，提高审核效率。最后，在人工审查环节确认为仿冒应用或恶意应用的样本的证书特征将被加入应用特征库，由同一开发者证书签署的其他应用在未来的检测中会在过滤模块阶段被直接拦截。

框架基于 Python 3 编写。

6.1.2 过滤模块

过滤模块分为两部分，分别为证书过滤和外观过滤。在面向规模较大的应用集输入时，过滤模块有三个主要功能：其一为通过应用证书信息比对，筛出黑名单开发者的应用，拒绝将其上架；其二为通过将输入的应用与已知正版应用匹配，将与已知正版应用无关的输入过滤，减少后续检测压力；其三为对输入的应用分类，以便在鉴别模块模块中根据分类进行对应的正版检测。

证书过滤指应用进入框架后，将先被提取证书信息，与应用特征库中已有的开发者黑名单比对。若应用证书于黑名单内，该应用将被直接拒绝上架，流程结束。证书信息比对部分的开发者黑名单由应用市场方自行维护，其中应包括已知仿冒开发者的证书信息与用于在 Android Studio 等开发环境调试的 debug 证书信息。应

用市场也应定期与其他各应用市场共享黑名单信息，防止第四章中发现的仿冒应用开发者在多个市场中上传应用、利用 debug 证书上传应用等风险再次产生。

外观过滤则再分为应用名匹配与应用图标匹配两部分。在应用名匹配部分，判断输入应用是否与某一已知正版应用匹配的依据源于人工审查中的正版应用命名模式。一个正版应用的命名模式由若干个特征点通过逻辑运算（与、或运算）连接而成，每个特征点为该正版应用的命名特征。一个特征点可由正则表达式表示，也可以是应用名长度范围。由第四章总结可得，仿冒应用的命名与对应的正版应用十分接近，因此应用市场方也可给出某正版应用名与相似度阈值作为特征点，加入该正版应用的命名模式。比如，由观察可得爱奇艺的应用名常有更改，但总会以“爱奇艺”起始，可用 app_name (“爱奇艺”) 表示该类特征；而 $similarity$ (“开心消消乐”，0.6) 则会将与“开心消消乐”相似度大于等于 0.6 的应用名纳入匹配范围。

应用图标匹配部分，应用市场方需要先为已知正版应用配置一个图标和相似度阈值作为比较标准，过滤模块获取输入应用图标后，对两者进行相似度计算，若相似度大于等于阈值，则认为两者相似。现有的主流图像相似度算法可分为四类，分别为严格判别算法、直方图比较、哈希算法和特征匹配法。严格判别算法即对两图片的像素点作严格一一比对的算法，速度较快，但敏感度太高，鲁棒性并不好，只适用于严格判定图像全等的领域；直方图比较则将两张图片的直方图数据归一化后进行相似度比较；哈希算法会先为每张图片生成一个哈希串，再通过哈希串之间对比实现图像相似度对比，可理解为将图像相似度转化为字符串相似度计算的算法；特征匹配法则通过提取图像中的特征点，计算特征点的特征向量后，比对不同图像的特征向量进行相似度比对。在对比了哈希算法与特征匹配法的性能后，本框架采用了特征匹配算法计算图像相似度。

具体实现方面，证书信息在利用 apktool 对应用拆包后，使用 JDK 的 keytool 工具提取，APK 中的图标文件具有固定路径，其提取也在 apktool 拆包后进行；应用名提取由 Android SDK 中的 aapt 命令行工具对 APK 文件进行解析后，从解析结果中截取获得；字符串匹配的相似度由两者之间的编辑距离与给出应用名的长度之比计算，特征匹配算法使用了图像的 GIST (Generalized Search Tree) 特征^[72]。

匹配之后，应用名和应用图标均被判定为与已知正版应用相似的样本会被打

上对应标签（标记样本与哪个已知应用类似），进行后续的仿冒鉴别。不被过滤模块判定为与已知正版应用相关、且应用证书不在开发者黑名单中的应用样本将进入应用市场方原有审核流程，而非由本框架判定是否仿冒应用或可否上架。

6.1.3 鉴别模块模块

鉴别模块模块根据过滤模块的标签对应用进行仿冒判别，判别依据为样本内的证书指纹信息。由第 2.2.1 节可知，Android 应用的签名信息可在大部分场景下指向应用的最后修改者，但恶意开发者仍可利用在 2017 年被揭露的 Janus 漏洞，对使用 V1 机制签名的 APK 进行篡改。

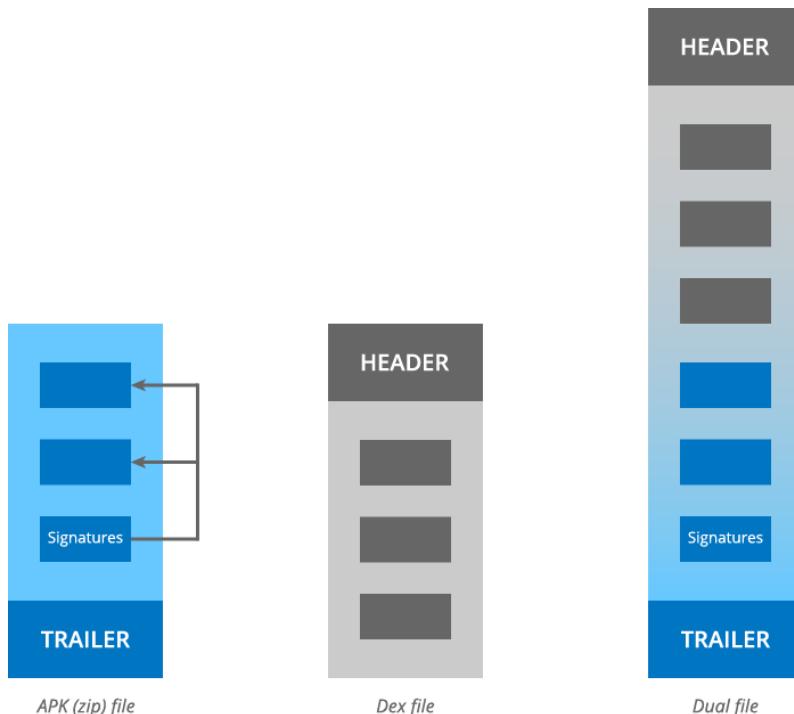


图 6.2: Janus 漏洞原理

图 6.2¹展示了 Janus 漏洞的工作原理。在安装 APK 文件时，虚拟机先从 APK 的尾部（Trailer 一端）读取校验签名信息（图中的 Signature 部分），校验成功后再从文件头部对 APK 文件进行解析和安装。然而，Android 系统的 dex 虚拟机同时支持对 dex 文件和 apk 文件的读取和执行。若在 APK 文件头部拼接一个 dex 文件，

¹ 本图源于网络资料 [73]

虚拟机在完成新文件的尾部校验之后、跳转到文件头部对新文件进行解析安装时，将直接执行拼接于头部的 dex 文件，导致系统被攻击。

因此，对于使用 V2、V3 机制签名的应用，本框架直接利用应用的证书信息判别其是否仿冒应用；对于使用 V1 机制签名的应用，框架除验证证书信息外，还使用另一种手段确保应用未被攻击。根据 dex 文件格式，每个 dex 文件的前 8 个字节为 Magic number 字段，内容为固定值 “dex\n035\0”；而 APK 文件本质上为 ZIP 格式的压缩文件，也具有固定格式，文件头部的前 4 个字节内容为 16 进制的“504b0304”。基于上述格式对文件头部前 4 个字节的内容读取判断，可鉴别应用是否损坏或被使用 Janus 漏洞攻击。

在本模块中被鉴别为正版应用的样本，其图标特征将被提取进应用特征库中以加强后续检测；若应用样本被认定为仿冒应用，则会进入代码分析模块被提取代码信息。利用 Janus 漏洞篡改应用的开发者将被直接加入应用特征库的开发者黑名单中。

6.1.4 代码分析模块

经过过滤模块和鉴别模块对应用筛选鉴别后，代码分析模块对应用进一步提取信息。本模块提取的数据有两类，一类为应用的基本数据，另一类为应用的代码数据。基本数据指应用的包名、构建应用使用的 Android API 版本、应用版本名称、应用版本号、应用大小、应用中声明使用的权限和证书指纹信息，以上信息可通过现有工具直接读取 APK 文件获取。代码数据则指利用代码分析技术获取的数据。借助静态分析技术，代码分析模块可从代码获取应用中的模块结构信息，包括应用中的类信息、各类中包含的方法信息、调用关系和应用写在 *AndroidManifest.xml* 文件中的配置。本模块在提取数据后，进一步分析调用关系，获取声明但未被使用过的敏感权限、敏感 API 调用链等信息，作为审核依据流转至人工审查。

具体来说，代码分析模块借助 Androguard^[74]，对应用中的 dex 文件进行反编译，获取反编译后的代码信息。对 dex 文件反编译得到的 Java 代码中，按类的定义者区分，可将应用代码中包含的类分为三种类型，分别为系统类，用户自定义类和第三方类。系统类指 Android 官方提供的开发框架中的类，如 Android 四大组件之

一 Activity 的基类 *android.app.Activity*; 第三方类指并非由 Java 语言本身提供、也并非开发者在项目中编写的类，通常包含于开发者引入的第三方库代码中，如游戏类应用常用 Unity 提供的第三方库作为游戏引擎；用户自定义类指应用开发者在开发该应用时，于项目中自行编写定义的类（包括继承系统类的自定义类）。Android Studio 对应用项目进行编译打包时，会将部分系统类、项目中的所有第三方类和所有用户自定义类都编译进 dex 中。对 dex 文件反编译出的所有代码进行解析会产生一定性能影响，也不利于应用间区分。因此，代码分析模块在处理时先使用 Libradar^[75] 将系统类剔除和识别第三方类，一则减少后续分析的数据量，二则避免第三方类中的敏感 API 调用对后续分析产生影响。

算法 6.1: 敏感 API 调用关系排查算法

输入: *permissions_API*, 列表, Android 权限与 API 映射关系
 输入: *declared_permissions*, 列表, 包含正版应用及其信息键值对
 输出: *redundant_permission*, 集合, 应用声明的冗余权限
 输出: *calling_sequence*, 集合, 分析后的敏感 API 的调用序列

```

1 Function iterSearcher(permissions_API, declared_permissions):
2     redundant_permission ← ∅;
3     for permission ∈ declared_permissions do
4         flag ← 0;
5         for API ∈ permissions_API[permission] do
6             callers ← getCallers(API);
7             if not caller.is_empty then
8                 flag ← 1;
9                 for caller ∈ callers do
10                    cache = PermissionCallAnalysis(caller, ∅);
11                    calling_sequence.update(cache);
12
13                    if flag = 0 then
14                        redundant_permission.add(permission);
15
16     return redundant_permission, calling_sequence;

```

除了类信息，代码分析模块还提取应用中的方法信息协助之后的人工审核。提取的方法信息包括用户自定义的所有方法以及 Android 框架中较为敏感的方法调用信息。用户自定义方法通过对 Androguard 中获得的用户自定义类进行方法分析

获取：代码分析模块遍历所有自定义类，获取其定义的每个方法，再将类名与方法名组合为二元组 $\langle class_name, method_name \rangle$ ，作为该方法在应用中的唯一标识保存；敏感方法调用信息则以自底向上的方法进行排查。

在基本数据获取阶段，代码分析模块通过 aapt 获得了构建应用的 Android API 版本和应用中声明使用的权限，而 Androguard 附带了各 Android API 版本中 API 与权限的映射关系。结合以上信息可获取在某应用中可能会被调用到的敏感 API，详情可见算法 6.1。对于应用声明的每个权限，先通过映射关系查出其对应的敏感 API 列表；对于每个敏感 API，算法遍历应用中的各个用户自定义类和第三方类，排查应用中是否有对该敏感 API 的调用（第 6 行）。如果有，则对调用该 API 的方法层层回溯，分析出敏感 API 的调用序列（第 11 行），具体流程见后文。对于某个已声明的权限，如果其对应的全部敏感 API 均没有调用记录，则可对应两种情况：一，该权限为由开发者声明的冗余权限，可能会被恶意应用利用，造成额外风险；二，开发者通过 Java 反射机制调用了与权限对应的敏感 API，导致该调用无法被静态分析扫描到。无论是哪种情况，都可能对用户安全产生隐患，也可被视作风险点，因此代码分析模块会记录该权限为冗余权限（第 13 行）。

在实现对敏感 API 调用链的回溯分析时，有两点挑战如下：第一点是方法调用路径中可能会出现调用环；第二点是敏感 API 及其前序方法可能存在多个调用者。前者指同一个方法多次出现在调用链中（如某方法多次递归调用自身后再调用敏感 API），容易导致工具在进行调用链回溯时出现死循环；后者让调用链成为了一棵以敏感 API 为根节点的调用树，树上的每个节点为一个方法，节点之间的有向边表示方法之间的调用关系。方法调用并不是一个线性过程，各方法之间灵活的相互调用导致了以上两点挑战。

为应对以上两点挑战，代码分析模块采用 DFS 算法和集合进行调用链回溯处理，具体可见算法 6.2。

对个某个方法，代码分析模块先借助 Androguard 获取其所有被调用者（第 3 行）和调用者（第 4 行）。如果被调用者中包含与权限相关的 API，则将 $\langle API, permission \rangle$ （即该 API 和其对应的权限）加入现有调用序列中（第 6 到 9 行）。之后，对每个调用者，若其之前未被分析过，且其为用户自定义方法，则递归调

算法 6.2: 调用链回溯算法

输入: *cur_method*, 当前遍历的方法
 输入: *itered_methods*, 集合, 已遍历的方法
 输出: *calling_sequence*, 集合, 分析后的敏感 API 的调用序列

```

1 Function PermissionCallAnalysis(cur_method, itered_methods):
2     calling_sequence ← ∅;
3     callees ← getCallees(cur_method);
4     callers ← getCallers(cur_method);
5     cur_seq ← [];
6     for callee ∈ callees do
7         if callee.is_permission_related_API then
8             permission = callee.get_related_permission();
9             cur_seq.add(<callee, permission>);
10    for caller ∈ callers do
11        if caller.is_self_defined && caller ∉ itered_methods then
12            itered_methods.add(caller);
13            res ← PermissionCallAnalysis(caller, itered_methods);
14            for sequence ∈ res do
15                calling_sequence.add(sequence + cur_seq);
16        if calling_sequence.is_empty && not cur_seq != [] then
17            calling_sequence.add(cur_seq);
18    return calling_sequence;

```

用方法 *PermissionCallAnalysis* 回溯以该方法为起始点的权限相关 API 调用序列（第 10 至 15 行）。对于方法是否被调用过的判断可以保证之前分析过的方法不再被重复分析，避免死循环的产生；遍历调用者和递归调用 *PermissionCallAnalysis* 保证了所有调用者均可被回溯，若某方法没有被其他自定义方法调用、或是其调用者已经全被分析过，则该方法为回溯终点。

分析结束后，代码分析模块将应用信息汇总，流转至人工审查。

6.1.5 人工审查

应用市场的审核人员在本模块对仿冒应用进行风险评估和确定。尽管本框架的前三个模块自动化地进行了仿冒应用的判别和应用行为信息的获取，但应用的

业务场景十分复杂，相同的 API 调用链在不同业务场景下可以有截然不同的含义（如“获取设备位置信息并上传至服务器”在导航软件中是必要的，而其他应用的相同行为可能会侵犯用户隐私），因此人工审查仍然不可避免。此处的人工审查有两个作用，其一是对被鉴定为仿冒的应用进行确认，排除误报信息，并根据误报情况修正过滤模块中不同部分的匹配阈值；其二是对仿冒应用的行为进行风险评估，根据应用行为的风险大小决定对应用和对应开发者的处理。如果一个样本仅在外观上模仿正版应用，但应用行为没有可疑之处，应用市场可驳回该次上架请求，要求开发者修正应用名与图标后重新申请上架；然而，若样本的确包含高风险行为甚至包含恶意代码，应用市场除了拒绝上架该应用，还可以对应用开发者作封禁处理，将应用中的证书指纹信息加入应用特征库的黑名单中，在未来的检测中直接拦截相同证书签署的应用。

6.2 系统实验

上节介绍了框架的设计与实现，本节将介绍针对框架的实验设计和结果分析。

6.2.1 数据收集

为检验 FakeRevealer 的有效性，作者重新从 Janus 平台上收集了 286 个应用样本对 FakeRevealer 进行测试。286 个应用样本包含了采集自 5 个热门应用的 255 个样本和随机采集的 31 个噪声样本，详细分布可见 表 6.1。

表 6.1: FakeRevealer 实验样本来源

来源应用	正版数量	仿冒数量	总计
抖音	32	18	50
开心消消乐	51	27	78
保卫萝卜	21	15	36
贪吃蛇大作战	14	29	43
WiFi 万能钥匙	2	46	48
噪声应用	N/A	N/A	31

噪声样本用于检验 FakeRevealer 是否准确地在不产生应用误报（将无关应用

识别为与已知应用相关的应用)的前提下,找到与已知应用有关的 Android 应用。在过滤模块的图标匹配部分,算法效果可能受已知样本数影响。框架为了消除该影响,实验在不同类别的应用中采用了不同的正版/仿冒数量比例作对比。

6.2.2 实验与结果

进行本组实验的设备搭载了 Intel i5-8250 CPU 与 24GB 内存。由于已知研究中,与仿冒应用直接相关的资料较贫乏,对比实验以近似研究提出的工具(第 1.2.1 节中的 CodeMatch^[20])作为比较对象。CodeMatch 是利用第三方库信息检测重打包应用的工具,用户在使用前需先将已知第三方库信息加入数据库中。工具先扫描 APK 文件获取代码,然后排除其中的已知第三方库,通过计算、比对剩余代码的哈希值进行重打包判断。根据本文定义,重打包应用也属于一种仿冒应用,因而可采用 CodeMatch 进行对比实验。

实验 1: 工具有效性验证

本实验以上节的 286 个应用样本组成测试集。FakeRevealer 在使用前需先配置正版应用相关信息,作者从测试集的 5 种正版应用中各取 2 个正版样本,提取其应用图标中的特征作为先验知识存入数据库中,图标匹配的相似度阈值均设置为 0.8,余下 276 个样本输入 FakeRevealer 进行测试。

表 6.2: 有效性实验结果

来源应用	正版样本识别数 (TN)	仿冒样本识别数 (TP/rbg)	仿冒样本识别数 (TP/gray)	FN	FP (rbg)	FP (gray)
抖音	30	15	17	0	3	1
开心消消乐	49	24	24	0	3	3
保卫萝卜	19	10	11	0	5	4
贪吃蛇大作战	12	29	29	0	0	0
WiFi 万能钥匙	0	32	30	0	14	16
噪声应用	0	0	0	0	0	0
总计	110	110	111	0	25	24

表 6.2 展示了 FakeRevealer 对 6 种应用的辨识结果。表中的“rgb”和“gray”分别对应图标匹配时采用的不同模式;“TP”,“TN”,“FP”,“FN”分别对应 *True Positive*,

True Negative, *False Positive* 和 *False Negative*, 用于描述实验结果。因 FakeRevealer 用于鉴别仿冒应用, 实验的“Positive”被设置成鉴别为仿冒的样本数目, “Negative”为鉴别为正版的样本数目(即“非仿冒样本”的数目)。

“rgb”模式表示采用包含 rgb 信息的原图像进行特征采集, “gray”模式表示在采集特征前先对图标进行灰度化处理。在 rgb 模式下, 每个像素点的色彩由“r”, “g”, “b”三个值表示, 每个值的值域为 [0, 255], 而在 gray 模式下, 一个像素点的色彩只需用一个值域同为 [0, 255] 的值表示其灰度, 因此在 gray 模式下进行的特征运算和比较速度更快。结果表明, 两种模式下的检测效果相当, gray 模式甚至略优于 rgb 模式。综合效果与运算速度, gray 模式更适合本框架使用。另外, FakeRevealer 在两种模式下均能正确辨认所有正版样本, 因此表 6.2 中的 TN 与 FN 不再分模式列出数据。

从实验数据看, FakeRevealer 的平均准确率 ($Precision = \frac{TP}{TP+FP}$) 分别为 81.48% (rgb 模式) 和 82.22% (gray 模式), 召回率 ($Recall = \frac{TP}{TP+FN}$) 均为 100%, F1 值 ($2 \times \frac{Precision \times Recall}{Precision + Recall}$) 则分别为 89.80% (rgb 模式) 和 90.24% (gray 模式), 无正版应用错判记录, 整体可用性较好, 在个别应用(贪吃蛇大作战)下甚至可以识别出所有仿冒样本。FakeRevealer 不对正版错判的原因有两个, 一个是于第 2.2.1 节中介绍的 Android 签名机制, 另一个是鉴别模块将正版图标特征反馈回特征数据库的机制。前者保证持有正版开发者证书的应用不会被误判为仿冒应用, 后者保证 FakeRevealer 对新图标有一定容错性。



图 6.3: 正版与仿冒版 WiFi 万能钥匙图标

然而, 在 WiFi 万能钥匙应用下, FakeRevealer 的鉴别能力较差, 46 个仿冒样本中有 16 个样本未被拦截。人工确认结果显示, 该部分样本均因为图标匹配未达

阈值而被误判。图 6.3 展示了 WiFi 万能钥匙对应的部分仿冒应用图标，正版图标于右上角由红色边框标出。该结果表明图标匹配算法仍有改进空间。

为确保 FakeRevealer 仅对已知应用样本生效，测试集还包含了 31 个随机搜集的应用样本（噪声应用），该组样本的图标和应用名均与已知的 5 种应用不类似，用于测试 FakeRevealer 是否会对未收录的应用类别产生误报。数据显示，31 个噪声应用均未被判定为正版或仿冒，FakeRevealer 具有一定抗干扰能力。

实验 2：对比实验

本组实验用于比较 CodeMatch 与 FakeRevealer 在分析应用时的有效性，分为时效性对比与准确性对比。实验采用 WiFi 万能钥匙一组的样本作为实验对象。由于 CodeMatch 并不能直接判断应用是否为仿冒应用，因此作者直接以 2 个正版样本与 46 个仿冒样本组成的 92 个应用对作为 CodeMatch 的输入，检查 46 个仿冒应用中是否存在正版应用的重打包样本。FakeRevealer 组以实验 1 中的 gray 模式实验数据作为结果与 CodeMatch 对比。

表 6.3: 对比实验结果

工具名	平均用时	判断出的仿冒应用（或重打包应用）数
CodeMatch	51.86 分钟	0
FakeRevealer	1.13 分钟	30

表 6.3 显示，46 个仿冒应用样本中未有针对 2 个正版应用样本的重打包应用，因此 CodeMatch 并未能检查出异常。重打包应用只是仿冒应用的一种，且商业应用迭代速度较快，相隔数个版本的应用已有较大区别，使用重打包检测方法检查仿冒应用不仅会遗漏非重打包形式的仿冒应用，也缺乏灵活性。仿冒应用凭借外观迷惑用户，检测方应该以应用名和图标鉴别仿冒应用，因而 FakeRevealer 的有效性明显高于 CodeMatch。用时方面，由于重打包应用需要在获取应用代码之后将代码和数据库中的所有已知第三方库逐一比对，难免有较大的时间开销；相比之下，FakeRevealer 直接根据应用名模式与图标特征对应用进行筛选，需处理的数据量远小于重打包检测中逐行代码比对产生的数据，因此也明显在速度上占优。

综上，重打包检测并非检测仿冒应用的有效思路，以应用外观为依据检测仿冒

应用是更有效且快捷的方法。

6.3 本章小结

本章介绍了仿冒应用检测框架 FakeRevealer 的设计与实现，并对其进行了系统实验。FakeRevealer 是设计用于 Android 应用市场的仿冒应用检测框架，有五个主要部分，用于自动化拦截对已知正版应用进行仿冒的仿冒应用和已知恶意开发者的恶意应用，可减缓应用市场在应用审核方面的人力成本，提高应用市场的安全程度。Android 应用市场常有大量应用上传等待上架，为有效处理大批量 Android 应用程序，FakeRevealer 采用过滤模块对应用进行快速筛选，剔除与已知正版应用不相似的应用样本，并为匹配的应用贴上对应标签。鉴别模块模块根据应用标签，鉴别应用真伪。其后，代码分析模块对鉴别模块模块鉴别为仿冒的应用进行拆包反编译处理，采集包括应用包名、版本号、声明权限在内的基本数据与应用中的类信息、方法信息等代码数据传入人工审查，并在人工确认后，将仿冒应用的证书指纹存入应用特征库；由鉴别模块模块判别为正版应用的样本将会被提取图标特征并加入应用特征库中，加强之后的检测。

系统实验显示，FakeRevealer 具有较好的可用性，可利用较少的已知数据快速定位输入应用集中的仿冒应用，每个仿冒应用的判别与分析均可在数分钟内完成。然而，部分仿冒应用因图标相似度未达到阈值未被 FakeRevealer 拦截，该结果表明 FakeRevealer 的图标匹配部分仍有改进空间。

第七章 总结与展望

7.1 总结

本文率先引入了“仿冒应用”这一概念。在搜集了大量数据的前提下，对仿冒应用从两个不同角度先后进行了两次实证研究，获取了仿冒应用的基本特性和仿冒应用开发者的行为特征，并推出了仿冒应用检测框架 FakeRevealer，有一定现实意义。具体地，本文主要工作总结如下：

- 1) 引入了“仿冒应用”的概念，对仿冒应用的危害进行了简要分析，并借助 Janus 对收集了来自现实世界中 29 个应用来源的近 14 万个应用样本，获取了样本的证书信息、应用大小、应用名等信息，组成数据集。
- 2) 利用上述数据集，分别从仿冒应用与原版应用相似度、影响应用被仿冒的严重程度的因素及仿冒应用行为入手开展实证研究，结合实际案例，对仿冒应用的基本特征进行分析，推断出“仿冒应用作者更倾向重新制作仿冒应用，而非通过重打包方式制作仿冒应用”的结论。
- 3) 利用数据集中的另一部分数据，完成面向仿冒应用开发者行为特征的实证研究，透过开发者可利用 Android 调试证书上传应用、仿冒应用开发者证书可长时间活跃等现象，推断出现有应用市场在监管方面的不足。
- 4) 在完成上述两次实证研究后，有针对性地分别向普通用户、开发者与应用市场方三个不同群体提出了切实可行的实用建议。
- 5) 设计并实现了仿冒应用检测框架 FakeRevealer，利用系统实验说明了 FakeRevealer 的可用性。

7.2 展望

本研究致力于展现一个面向仿冒应用的清晰视角，并通过设计 FakeRevealer 协助市场方提升仿冒应用对仿冒应用的拦截能力。然而，在研究中仍有几点问题，可

作为未来工作的方向。其一为对仿冒应用基本特征的实证研究中未涉及与代码、应用行为相关的具体研究，若能分析仿冒应用的行为并总结出一类特征，将十分有助于后续检测。其二为 FakeRevealer 中鉴别模块模块的代码信息获取部分，现时方案还无法追踪用户的 Java 反射调用相关代码；FakeRevealer 的图标匹配部分也有待加强以提高灵敏度。其三为 FakeRevealer 仍对人工有较强依赖，包括对输出的结果进行确认判断和规则的分析提取。尽管人工审核在安全领域难以避免，但在未来工作中或许可为 FakeRevealer 添加一个自动分析模块，减少在特征分析上的所需人力。

参考文献

- [1] STATCOUNTER. Mobile Operating System Market Share Worldwide, 2009 - 2020[EB/OL]. 2019 [February 23, 2020].
<https://gs.statcounter.com/os-market-share/mobile/worldwide/#yearly-2009-2020>.
- [2] CLEMENT J. Number of available applications in the Google Play Store from December 2009 to December 2019[EB/OL]. 2019 [January 4, 2020].
<https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>.
- [3] WASSERMAN A I. Software engineering issues for mobile application development[C] // Proceedings of the FSE/SDP workshop on Future of software engineering research. 2010 : 397 – 400.
- [4] CHEN S, FAN L, CHEN C, et al. StoryDroid: Automated Generation of Storyboard for Android Apps[C] // Proceedings of the 41th ACM/IEEE International Conference on Software Engineering, ICSE 2019. 2019.
- [5] ZHOU Y, JIANG X. Dissecting Android Malware: Characterization and Evolution[J]. 2012 IEEE Symposium on Security and Privacy, 2012 : 95 – 109.
- [6] KHANMOHAMMADI K, EBRAHIMI N, HAMOU-LHADJ A, et al. Empirical study of android repackaged applications[J]. Empirical Software Engineering, 2019, 24(6) : 3587 – 3629.
- [7] LI L, GAO J, HURIER M, et al. Androzoo++: Collecting millions of android apps and their metadata for the research community[J]. arXiv preprint arXiv:1709.05281, 2017.
- [8] ZHOU W, ZHOU Y, JIANG X, et al. Detecting repackaged smartphone applications in third-party android marketplaces[C] // CODASPY. 2012.

- [9] ZHENG M, SUN M, LUI J C S. DroidAnalytics : A Signature Based Analytic System to Collect , Extract , Analyze and Associate Android[C] // . 2013.
- [10] 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.
- [11] CRUSSELL J, GIBLER C, CHEN H. Scalable semantics-based detection of similar Android applications[C] // Proc. of ESORICS : Vol 13. 2013.
- [12] CHEN K, LIU P, ZHANG Y. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets[C] // Proceedings of the 36th International Conference on Software Engineering. 2014 : 175 – 186.
- [13] HANNA S, HUANG L, WU E, et al. Juxtap: 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.
- [14] LINARES-VÁSQUEZ M, HOLTZHAUER A, POSHYVANYK D. On automatically detecting similar Android apps[C] // Program Comprehension (ICPC), 2016 IEEE 24th International Conference on. 2016 : 1 – 10.
- [15] WU X, ZHANG D, SU X, et al. Detect repackaged android application based on http traffic similarity[J]. Security and Communication Networks, 2015, 8(13) : 2257 – 2266.
- [16] ZHANG F, HUANG H, ZHU S, et al. ViewDroid: towards obfuscation-resilient mobile application repackaging detection[C] // WISEC. 2014.
- [17] SUN M, LI M, LUI J C. DroidEagle: seamless detection of visually similar Android apps[C] // Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks. 2015 : 1 – 12.
- [18] KYWE S M, LI Y, DENG R H, et al. Detecting camouflaged applications on mobile application markets[C] // International Conference on Information Security and Cryptology. 2014 : 241 – 254.
- [19] SOH C, TAN H B K, ARNATOVICH Y L, et al. Detecting clones in android appli-

- cations through analyzing user interfaces[C] // 2015 IEEE 23rd international conference on program comprehension. 2015 : 163 – 173.
- [20] GLANZ L, AMANN S, EICHBERG M, et al. CodeMatch: obfuscation won't conceal your repackaged app[C] // Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017 : 638 – 648.
- [21] 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.
- [22] ZHOU W, ZHANG X, JIANG X. AppInk: watermarking android apps for repackaging deterrence[C] // Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. 2013 : 1 – 12.
- [23] ZHOU W, WANG Z, ZHOU Y, et al. Divilar: Diversifying intermediate language for anti-repackaging on android platform[C] // Proceedings of the 4th ACM conference on Data and application security and privacy. 2014 : 199 – 210.
- [24] LUO L, FU Y, WU D, et al. Repackage-proofing android apps[C] // 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 2016 : 550 – 561.
- [25] ZENG Q, LUO L, QIAN Z, et al. Resilient decentralized Android application repackaging detection using logic bombs[C] // Proceedings of the 2018 International Symposium on Code Generation and Optimization. 2018 : 50 – 61.
- [26] FELT A P, FINIFTER M, CHIN E, et al. A survey of mobile malware in the wild[C] // SPSM@CCS. 2011.
- [27] ENCK W, ONGTANG M, MCDANIEL P. On lightweight mobile phone application certification[C] // Proceedings of the 16th ACM conference on Computer and communications security. 2009 : 235 – 245.
- [28] FELT A P, CHIN E, HANNA S, et al. Android permissions demystified[C] // Proceedings of the 18th ACM conference on Computer and communications security. 2011 : 627 – 638.

- [29] GRACE M, ZHOU Y, ZHANG Q, et al. Riskranker: scalable and accurate zero-day android malware detection[C] // Proceedings of the 10th international conference on Mobile systems, applications, and services. 2012 : 281 – 294.
- [30] 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) : 1 – 29.
- [31] YAN L K, YIN H. Droidscope: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis[C] // Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12). 2012 : 569 – 584.
- [32] ZHOU Y, WANG Z, ZHOU W, et al. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets.[C] // NDSS : Vol 25. 2012 : 50 – 52.
- [33] PORTOKALIDIS G, HOMBURG P, ANAGNOSTAKIS K, et al. Paranoid Android: versatile protection for smartphones[C] // Proceedings of the 26th annual computer security applications conference. 2010 : 347 – 356.
- [34] CHEN S, XUE M, TANG Z, et al. Stormdroid: A streaminglized machine learning-based system for detecting android malware[C] // Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. 2016 : 377 – 388.
- [35] BURGUERA I, ZURUTUZA U, NADJM-TEHRANI S. Crowdroid: behavior-based malware detection system for android[C] // Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. 2011 : 15 – 26.
- [36] WU D-J, MAO C-H, WEI T-E, et al. Droidmat: Android malware detection through manifest and api calls tracing[C] // 2012 Seventh Asia Joint Conference on Information Security. 2012 : 62 – 69.
- [37] GASCON H, YAMAGUCHI F, ARP D, et al. Structural detection of android malware using embedded call graphs[C] // Proceedings of the 2013 ACM workshop on Artificial intelligence and security. 2013 : 45 – 54.
- [38] CHAKRADEO S, REAVES B, TRAYNOR P, et al. Mast: Triage for market-scale

- mobile malware analysis[C] // Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks. 2013 : 13–24.
- [39] AAFER Y, DU W, YIN H. Droidapiminer: Mining api-level features for robust malware detection in android[C] // International conference on security and privacy in communication systems. 2013 : 86–103.
- [40] ARP D, SPREITZENBARTH M, HUBNER M, et al. Drebin: Effective and explainable detection of android malware in your pocket.[C] // Ndss : Vol 14. 2014 : 23–26.
- [41] ANDOW B, NADKARNI A, BASSETT B, et al. A Study of Grayware on Google Play[J]. 2016 IEEE Security and Privacy Workshops (SPW), 2016 : 224–233.
- [42] HU Y, WANG H, ZHOU Y, et al. Dating with Scambots: understanding the ecosystem of fraudulent dating applications[J/OL]. IEEE Transactions on Dependable and Secure Computing, 2019.
<http://dx.doi.org/10.1109/TDSC.2019.2908939>.
- [43] CHEN S, MENG G, SU T, et al. AUSERA: Large-Scale Automated Security Risk Assessment of Global Mobile Banking Apps[J]. arXiv preprint arXiv:1805.05236, 2018.
- [44] CHEN S, SU T, FAN L, et al. Are mobile banking apps secure? what can be improved?[C] // Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2018 : 797–802.
- [45] KITCHENHAM B A, PFLEEGER S L, PICKARD L M, et al. Preliminary guidelines for empirical research in software engineering[J]. IEEE Transactions on software engineering, 2002, 28(8) : 721–734.
- [46] BIEMAN J M, ZHAO J X. Reuse through inheritance: A quantitative study of C++ software[J]. ACM SIGSOFT Software Engineering Notes, 1995, 20(SI) : 47–52.
- [47] HARRISON R, COUNSELL S, NITHI R. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems[J]. Journal of Systems and Software, 2000, 52(2-3) : 173–179.

- [48] WANG H, LI H, LI L, et al. Why are android apps removed from google play? a large-scale empirical study[C] // 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR). 2018 : 231 – 242.
- [49] DYBÅ T, DINGSØYR T. Empirical studies of agile software development: A systematic review[J]. Information and software technology, 2008, 50(9-10) : 833 – 859.
- [50] MANOTAS I, BIRD C, ZHANG R, et al. An empirical study of practitioners' perspectives on green software engineering[C] // 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). 2016 : 237 – 248.
- [51] MCINTOSH S, KAMEI Y, ADAMS B, et al. An empirical study of the impact of modern code review practices on software quality[J]. Empirical Software Engineering, 2016, 21(5) : 2146 – 2189.
- [52] MCILROY S, ALI N, HASSAN A E. Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store[J]. Empirical Software Engineering, 2016, 21(3) : 1346 – 1370.
- [53] 吴迪. 基于开源软件的 C++ 关键语言特性实证研究 [D]. 南京大学 2016.
- [54] 杨姗媛. 信息安全风险分析方法与风险感知实证研究 [D]. 中央财经大学 2015.
- [55] SEAMAN C B. Qualitative methods in empirical studies of software engineering[J]. IEEE Transactions on software engineering, 1999, 25(4) : 557 – 572.
- [56] EASTERBROOK S, SINGER J, STOREY M-A, et al. Selecting empirical methods for software engineering research[G] // Guide to advanced empirical software engineering. [S.l.] : Springer, 2008 : 285 – 311.
- [57] Cryptographic hash function[EB/OL]. 2020 [June 02, 2020].
https://en.wikipedia.org/wiki/Cryptographic_hash_function.
- [58] FEISTEL H. Cryptography and computer privacy[J]. Scientific american, 1973, 228(5) : 15 – 23.
- [59] Apktool[EB/OL]. [June 02, 2020].
<https://github.com/iBotPeaches/Apktool>.

- [60] dex2jar[EB/OL]. [June 02, 2020].
<https://github.com/pxb1988/dex2jar>.
- [61] jd-gui[EB/OL]. [June 02, 2020].
<https://github.com/java-decompiler/jd-gui>.
- [62] LI Y, JANG J, HU X, et al. Android malware clustering through malicious payload mining[C] // International Symposium on Research in Attacks, Intrusions, and Defenses. 2017: 192–214.
- [63] LEVENSHTEIN V I. Binary codes capable of correcting deletions, insertions, and reversals[C] // Soviet physics doklady : Vol 10. 1966 : 707–710.
- [64] Set the application ID | Android Developers[EB/OL]. [August 13, 2020].
<https://developer.android.com/studio/build/application-id>.
- [65] Violin plot[EB/OL]. 2020 [March 27, 2020].
https://en.wikipedia.org/wiki/Violin_plot.
- [66] Baidu App Store[EB/OL]. [September 26, 2018].
<https://shouji.baidu.com/>.
- [67] Hiapk[EB/OL]. [September 26, 2018].
<http://apk.hiapk.com/>.
- [68] VirusTotal[EB/OL]. [September 26, 2018].
<https://www.virustotal.com/>.
- [69] Keytool[EB/OL]. [August 25, 2021].
<https://docs.oracle.com/javase/8/docs/technotes/tools/windows/keytool.html>.
- [70] Android 平台 UICC 验证 [EB/OL]. [August 25, 2021].
<https://source.android.google.cn/devices/tech/config/uicc>.
- [71] ZHANG Z, DIAO W, HU C, et al. An empirical study of potentially malicious third-party libraries in Android apps[C] // Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks. 2020 : 144–154.
- [72] TORRALBA A, MURPHY K P, FREEMAN W T, et al. Context-based vision system for place and object recognition.[C] // ICCV : Vol 3. 2003 : 273–280.

- [73] Janus 漏洞 (CVE-2017-13156) : 修改安卓 app 而不影响签名 [EB/OL]. [October 5, 2020].
<https://www.anquanke.com/post/id/89979>.
- [74] Androguard[EB/OL]. [Sep 23, 2020].
<https://github.com/androguard/androguard>.
- [75] 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.

攻读学位期间发表的学术论文

1. ***** ****, *** ****, ***** **, **** * *, **** **, ***** ****, and
***** **, “A large-scale empirical study on industrial fake apps,” in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, IEEE Press, 2019: 183-192. (第一作者, 发表于 CCF A 类推荐学术会议, 软件工程方向顶级会议 ICSE2019)

致 谢

在师大的七年时光转瞬而逝，不知不觉，研究生历程已快要告一段落，我也将迎来下一个人生阶段。

借此机会，想对在师大遇上的各位老师和同学表示最真诚的感谢，尤其是我的导师，在我研究生的各个阶段都给予我鼓励和指导。同样想感谢的还有家里人和身边的朋友，是他们的支持帮助我走过了一路上的各种波折。

二〇二〇年三月