软件抄袭检测研究综述

田振洲[1], 刘 烃[1], 郑庆华[1], 佟菲菲[1], 吴定豪[2], 朱森存[2,3], 陈 恺[4]

[1]西安交通大学 智能网络与网络安全教育部重点实验室 西安 中国 710049
[2]宾夕法尼亚州立大学帕克分校 计算机科学与工程系 美国 16802
[3]宾夕法尼亚州立大学帕克分校 信息科学与技术学院 美国 16802
[4]中国科学院信息工程研究所 北京 中国 100093

摘要 随着开源软件项目的蓬勃发展, 软件抄袭俨然已成为软件生态环境健康发展的威胁之一, 其得到越来越多的研究人员、教育人员、开源社区及软件企业的关注, 软件抄袭检测对于软件知识产权保护具有重要意义。本文对软件抄袭检测的研究现状和进展进行综述。首先介绍软件抄袭检测的意义和威胁模型; 然后, 根据应用场景和技术手段, 从源代码抄袭检测、无源码场景下基于软件水印和基于软件胎记的抄袭检测三个方面, 对现有软件抄袭检测技术进行阐述和比较; 最后, 通过分析软件抄袭检测研究存在的问题及其面临的挑战和实际需求, 对未来研究方向进行了展望。

关键词 知识产权保护; 软件保护; 软件抄袭; 软件抄袭检测; 软件胎记; 代码相似性分析; 软件水印; 代码混淆
中图法分类号 TP309.2 DOI 号 10.19363/j.cnki.cn10-1380/tn.2016.03.005

Software Plagiarism Detection: A Survey

TIAN Zhenzhou[1], LIU Ting[1], ZHENG Qinghua[1], TONG Feifei[1], WU Dinghao[2], ZHU Sencun[2,3], CHEN Kai[4]

[1] Ministry of Education Key Lab For Intelligent Networks and Network Security, Xi'an Jiaotong University, Xi'an 710049, China
[2] Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802, USA
[3] College of Information Sciences and Technology, Pennsylvania State University, University Park, PA 16802, USA
[4]Institute of Information Engineering, Chinese Academy of Science, Beijing 100093, China

Abstract With the burst of free and open source software projects, software plagiarism has become a serious threat to the healthy development of the software ecosystem. Researchers, educators, open source developers, and software company managers are paying more and more attention to the problem. Software plagiarism detection is critical to the protection of software intellectual property. This paper provides a review of the state-of-the-art software plagiarism detection techniques. First, the significance and threat models of plagiarism detection are presented, followed by the description and comparison of existing techniques on plagiarism detection. We classify the existing methods into three major categories, including source-code plagiarism detection, software watermark based plagiarism detection and software birthmark based plagiarism detection, according to the scenarios they are designed for and applicable to as well as different principles adopted. Finally, through analyzing the limitations of the existing plagiarism detection techniques, the emerging challenges and practical requirements, we discuss several possible future research directions.

Key words intellectual property protection; software protection; software plagiarism; software plagiarism detection; software birthmark; code similarity analysis; software watermarking; code obfuscation

1 引言

软件抄袭是一种为达到特定目的而非法拷贝并使用他人代码的行为, 从学生上机作业的抄袭到软件产品的抄袭、移动端 App 的重打包, 都属于软件抄袭的范畴。软件抄袭在教育界一直饱为诟病, 学生上机作业的抄袭非常普遍, 这严重影响学生能力的培养和计算机教学的效果。国外最近的一项调研中约有 33%学生承认曾经抄袭[1]; 另一项研究中, 教学人员对学生从 1993 到 2006 年的 4 门编程作业进行分析, 发现 1%-10%比例的抄袭, 而且呈逐年增长趋势[2]; 另外, MOSS(一款软件抄袭检测服务)的统计结果显示, 提交其检测的所有学生作业都会有 10%左右的抄袭比例, 而且这还是在学生提前被告知其上机作

通讯作者: 刘烃, 博士, 副教授, Email: tingliu@mail.xjtu.edu.cn。
本课题得到国家自然科学基金(91118005, 91218301, 61221063, 61428206, 61203174, 91418205, 61472318, 1500365); 教育部创新团队(IRT13035); 国家科技支撑计划(2013BAK09B01)资助。
收稿日期: 2016-06-26; 修改日期: 2016-07-07; 定稿日期: 2016-07-08

业会被检查的情况下的结果[3]。

近些年, 软件抄袭也已成为正规的软件公司和开源软件组织所迫切关心的问题, 其对软件知识产权的保护构成了严重威胁。现在有非常多的开源软件项目和社区, 比如 SourceForge.net 上有 43 万个开源项目, 平均每天有高于 480 万的下载量。开源项目的蓬勃发展一方面促进了软件行业的繁荣, 同时也给不法的抄袭者提供了更多以及更便捷的抄袭他人代码的机会, 因为源码相比可执行的二进制代码而言更容易理解和修改, 抄袭者还可能会利用自动化的混淆工具进行简单包装来盗用全部代码。通常这些开源项目允许用户在遵守特定的 License(如众所周知的 GPL、BSD 等)的前提下使用、修改以及对其进行重新发布, 然而受经济利益的驱使, 有些软件公司或个人会在其产品中集成第三方代码而不遵守相应的 License。另外, 很多下游公司会将其软件项目的部分功能模块外包给上游小公司, 通常这些模块会以二进制代码的形式交付, 上游公司无从得知这些模块是否存在抄袭的情况。这些有意或无意的抄袭也导致了很多案件的发生[4, 5], 比如电信巨头 Verzion 曾因在其 FIOS 无线路由器中使用 Busybox 的代码而被自由软件基金会起诉, 而问题模块是 Actiontec Electronics 交付的; 再比如 Skype 曾因违背 Joltid 的条款险些被迫终止其 voice-over-IP 服务。

此外, 随着智能手机的普及和手机应用的涌现, 移动端 App 的抄袭问题也得到工业界和学术界越来越多的重视[6]。抄袭者通过重打包(将 App 进行拆卸, 对其代码、数据等进行增删改或不进行任何改动, 然后重新组装打包成新 App 的行为)修改软件的拥有者、植入广告达到非法获取经济利益的目的。研究者对多个第三方 App 市场的研究表明, 5%-13%的 App 是重打包发布而来的[7]; 甚至 Google Play 自身中有 1.2%的 App 是重打包的[8]。此外, 通过重打包植入恶意 payloads 可以快速生成并传播大量恶意软件, 对 App 市场生态环境构成严重威胁。最近一项研究显示[8], 来自 33 个 Android 市场共计 120 万个 App 中有 10.93%是可疑的, 即使 Google Play 中也有 7.61%的应用是恶意的, 而且其中有些应用已被安装 100 万次以上; 另一项研究表明[7], Android 恶意软件中约有 86%是通过对合法软件进行重打包而来的。

软件抄袭检测的研究工作已经有 40 多年历史, 最初研究主要集中在学生上机作业的抄袭检测, 应用在源码存在这种场景; 后来随着工业界对软件知识产权保护的重视, 很大一部分关注点转移到商业产品的抄袭检测上, 提出了基于软件水印和软件胎记的检测方法, 这些方法可用于源码不存在的场景; 后来随着手机应用市场的繁荣, 移动应用的抄袭检测研究也获得越来越多的关注, 主要还是结合手机应用的自身特点, 利用软件水印或软件胎记技术实现抄袭检测。

软件抄袭检测的研究成果也在许多顶级的国际会议和期刊上进行了发表, 如 ICSE 会议[9, 10]、CCS 会议[11]、FSE 会议[12]、ASE 会议[13, 14]、POPL 会议[15]、ACSAC 会议[16, 17]、ISSTA 会议[18, 19]、TSE 期刊[20-22]、TOC 期刊[23, 24]、KDD 会议[25, 26]、TIFS 期刊[27]等。目前并没有系统性的工作对软件抄袭检测技术进行综述, 比较相关的是 2010 年发表在《计算机科学》上熊浩等关于代码相似性检测技术的综述[28]以及 2003 年发表在《软件学报》上张立和等关于软件水印的综述[87]。前者仅对源码存在情况下的抄袭检测技术进行了总结; 后者介绍了当时的几种水印技术和水印的攻击方法, 而至今水印技术已取得非常多的进展, 需要重新归纳总结。本文综述了软件抄袭检测研究的最新进展, 不仅囊括了最新的源码抄袭检测技术和软件水印技术, 还对软件胎记这类主流技术进行了归纳, 同时介绍了移动端抄袭检测的研究进展, 全面比较和分析了不同技术的优缺点及适用场景, 并结合我们自身在这方面的研究基础对未来的研究方向和面临的挑战进行了探讨。

本文结构如下: 第 2 节对软件抄袭检测的一些关键问题进行介绍, 包括对抄袭问题的描述、威胁模型以及抄袭检测技术的归类法; 第 3 节介绍针对源码的抄袭检测技术; 第 4 节和第 5 节分别介绍了无源码场景下基于软件水印和软件胎记的抄袭检测技术; 第 6 节探讨抄袭检测研究的一些新问题、面临的挑战和未来的研究方向; 最后总结全文。

2 软件抄袭检测的关键问题

2.1 理解软件抄袭和抄袭检测

对于软件抄袭, 各个领域有不同的定义。综合其他文献的描述, 广义而言本文将其描述为一种非法使用他人软件的各种基本要素的行为, 其中非法是指未得到软件拥有者的允许或未能遵守既定的条款规约, 软件基本要素包括软件代码、设计思想、用户界面、软件文档、输入数据等, 这种行为可以是有意的也可以的无意的。这样的描述相当宽泛, 简单的软件拷贝再发布属于抄袭, 花费大量时间和精力逆向分析出他人代码的核心思想然后用其他语言实现也属于抄袭, 而且很多时候抄袭与否难以界定。2012 年的一项研究显示[1], 大学教学人员和学生对软件

抄袭的界定就存在很大差异, 比如 76%的教学人员认为用户界面的抄袭属于软件抄袭, 而仅有 53%的学生持相同观点。

鉴于此, 本文对要解决的抄袭问题进行限定, 将其定义为攻击者(抄袭者)在对软件没有任何或很少了解的情况下花费很少的精力对软件代码实施的抄袭; 也就说抄袭者不会花时间破解或理解代码, 甚至完全没有任何代码知识, 而是通过专业工具或者很少的人工修改实施抄袭, 这也是目前研究人员关注的抄袭问题。为了准确描述本文讨论的软件抄袭(如无特别说明, 下文的软件抄袭均限定在此范畴, 不再赘述), 下面给出其形式化定义。

定义 1. 软件抄袭。对于两个程序 *P* 和 *Q*, 只要满足以下任一条件, 就说 *Q* 抄袭了 *P*:

C1) *Q* 是 P 的直接拷贝;

C2) *Q* 是攻击者通过对 *P* 应用语义保留的代码变换 T(如标识符修改、优化、混淆)生成的;

C3) *Q* 集成了程序 *P* 的所有代码;

C4) *Q* 集成了程序 *P* 的部分代码;

C5) *Q* 是通过对 *P* 反复应用上述修改得来的。

定义 1 描述了攻击者实施软件抄袭的手段。C1 最为直观, 攻击者对整个软件实施完整的拷贝, 通过直接发布或简单修改软件所有者信息后再发布的方式完成抄袭, 这种方式实施的抄袭非常普遍同时相较其他方式也更易检测。C2 是攻击者利用自动化的代码变换技术(如采用不同的编译器及编译优化选项、单独的优化器、各种混淆工具、加壳工具等)或者较少的人工修改(如修改标识符、增减注释、代码重布局等), 可以说目前绝大部分的检测技术都是针对这种抄袭方式的。C3 是攻击者在原程序 *P*(为方便描述, 下文将原程序 *P* 称为原告, 将抄袭生成的程序 *Q* 称为被告)的基础上通过添加新的功能模块来实施抄袭, 这种抄袭方式在 Android 平台最为常见, 攻击者从市场抓取已发布的原告 *P*, 通过重打包技术在 *P* 中植入广告或恶意模块, 然后再次发布到该应用市场或其他市场。C4 是攻击者只是利用原告 *P* 的部分代码(如某个功能库), 在此基础上进行再次开发的方式, 有可能被告 *Q* 中只包含 *P* 很小一部分代码。C5 是综合运用 C1-C4 的手段实施的抄袭, 比如 Android 平台 App 的抄袭, 攻击者可以在重打包植入恶意代码的同时对原代码及恶意代码实施混淆, 进一步使被告 *Q* 与原告 *P* 的区别增大。此外, 很多情况下被告 *Q* 的源代码是无法获取的, 攻击者为躲避检测, 其抄袭生成的程序 *Q* 通常会以二进制或 Java 字节码等形式发布, 这使得软件抄袭检测难度大大增加, 因为可执行程序的分析要比源代码分析难得多。

因此, 软件抄袭检测需要解决以下几个关键问题:

1) 如何实现检测过程的自动化或较少的人工参与;

2) 源码缺失情况下, 如何开展分析;

3) 如何应对各式各样的代码混淆技术;

4) 如何解决部分抄袭检测的问题。

目前的软件抄袭检测研究都以自动化检测为目标, 提出了很多以二进制代码为分析对象的动静态检测技术, 同时通过融合语义分析也提出了不少对代码混淆具备很好抵抗力的方法。然而对于部分抄袭, 尽管这类问题在实际情况下非常普遍, 但因为更多时候需要人为参与审查, 而研究者主要还是关注自动化的检测技术, 因此目前的研究涉及极少, 这也是软件抄袭检测亟待解决的关键问题。

2.2 代码混淆

软件抄袭属于主机攻击(host attack)范畴, 也就是在不计代价和开销的情况下, 抄袭者可以利用任何当下可行的技术和计算资源达到抄袭的目的; 因而, 实现软件抄袭的手段不胜枚举。下面着重介绍一下通过代码混淆实现抄袭的手段, 也是目前几乎所有抄袭检测研究关注的重点。

代码混淆是通过的特定代码变换手段将程序 *P* 转换成另一个程序 *P'*的一种技术, 并且 *P* 与 *P'*至少在可观测行为上保持语义等价, 其目的是有意地隐藏程序逻辑, 使混淆后的程序 *P'*更加难以理解和分析。代码混淆被广泛用于增加逆向分析的难度, 保护核心代码; 但类似于恶意软件制造者用其隐藏恶意代码, 抄袭者同样利用代码混淆来隐藏抄袭的意图, 加大分析难度以躲避检测。

代码混淆可以在源代码、字节码或者机器码上进行, 通过人工或采用自动化的混淆器实现, 后者需要实现特定的代码混淆技术, 但比人工混淆能够产生更不容易出错同时更加晦涩难懂的代码。简单而言, 可以将现有的混淆技术分为外形混淆、控制混淆、数据混淆三类。

外形混淆主要通过词法变换手段, 如标识符重命名、注释删除、格式及布局调整等, 降低代码的可读性; 此类混淆技术实现容易, 同时不会给程序带来额外开销, 但混淆程度较弱, 仅可以对抗部分基于文本和词法分析的检测技术。控制混淆是通过改变程序的控制流结构进行的代码变换, 如代码内联和外联、循环条件扩充、基于不透明谓词的死码及垃圾代码植入、控制流扁平化等; 控制混淆会对程序的控制流结构造成较大改变, 因而基于控制流分析的某些检测技术可能会失效。数据混淆是对程序的

数据结构进行改变的代码变换, 常采用的手段包括变量分割、参数重排、标量合并、数组重构等; 数据混淆会影响数据的存储、编码等, 因而会干扰单纯基于数据流分析的抄袭检测技术。

现有代码混淆工具也有很多, 典型的如Sandmark[29]、ZelixKlassmaster[30]、DashO[31]、DexGuard[32]、Loco[33]、Upx[34]、ProGuard[35]等。代码混淆技术本身是一个非常活跃也很重要的研究话题, 这里不再做深入探讨, 感兴趣的读者可参阅文献[21, 36-38]。

2.3 软件抄袭检测技术分类

本文按照应用场景和技术手段的不同, 将现有的软件抄袭检测方法归为三类: 源码抄袭检测技术、基于软件水印的抄袭检测技术以及基于软件胎记的抄袭检测技术。

2.3.1 源码抄袭检测技术

这类抄袭检测技术适用于被告源码能够获取的场景, 其通过衡量原告与被告源代码的相似性来判断是否存在抄袭。通常这类技术是将代码当做纯文本进行分析或者采用简单的词法语法分析, 很少涉及语义分析, 因而抗混淆能力较弱。不过在特定的场景, 如学生上机作业的抄袭检测, 大多数情况下学生是采用直接地人工修改(如修改变量名、操作符变换等)的方式实现抄袭, 抄袭手段不会太复杂, 因而也出现了不少实用的检测工具诸如 MOSS[39]、JPlag[40]、Sherlock[41]等, 并在不少国外高校得到推广应用。

2.3.2 基于软件水印的抄袭检测技术

很多情况下被告的源码是无法获取的, 特别是商业软件的抄袭, 被告通常以可执行文件的形式发布, 在搜集到足够证据之前, 抄袭者不太可能给出其源码。软件水印技术就是针对这种情况设计的, 并被很多软件公司所采用。该类技术的基本原理是在软件发布之前, 在代码中植入一个特殊的标识符(水印), 来标识软件的所有者; 该水印事后可以通过特殊的提取器被识别或抽取出来, 其可以作为证据达到检测的目的。因为这类技术需要在代码中植入额外的数据(或代码), 水印算法过于复杂会影响程序的性能和稳定性, 过于简单又影响其隐蔽性, 因此在设计水印算法时需要综合考量其性能、花销、隐蔽性、抗毁性等。

2.3.3 基于软件胎记的抄袭检测技术

不管是静态还是动态的软件水印技术, 都较容易被语义保留的代码混淆破坏掉, 研究者进一步提出了软件胎记方法。不同于水印技术, 软件胎记技术是静态或动态地从软件中抽取出一些不易改变的特征(胎记), 这些特征可以唯一地对该软件进行标识, 然后基于胎记的相似性判断抄袭是否存在。由于不需要事先植入额外信息, 胎记技术不会对程序引入额外的负载; 然而该类技术的检测效果很大程度上依赖于高质量的软件胎记的抽取, 软件胎记越贴近程序语义检测效果越好。已有的研究表明, 融合了语义分析的软件胎记技术特别是动态胎记技术对代码混淆具备非常好的抵抗力。

3 源码抄袭检测技术

提及源码抄袭检测, 人们最容易联想到同时也容易混为一谈的是代码克隆检测问题, 尽管克隆和抄袭都会产生高度相似的代码, 但二者有着本质不同。

首先, 目的和基本假设不完全一致: 代码克隆之所以产生是因为代码本身不可以或不方便直接重用, 比如难以满足程序员想要的功能, 需要稍微修改, 克隆代码的行为与原代码完全一致或略有差异, 并且程序员需要理解被克隆代码片段的语义行为; 而抄袭则是攻击者为了获取个人利益, 其通常是在不理解代码的前提下, 企图对代码进行大量以及各种各样的转换、伪装以减少被检测的可能性, 同时要尽可能保证程序语义不被改变, 因为是有意的伪装, 其比代码克隆更难检测。

其次, 克隆和抄袭所产生的相似代码的尺寸不同: 克隆片段通常较小, 而抄袭代码则通常是原代码的大规模或全部重用; 同时, 克隆检测旨在实现软件自身代码重复片段的检测, 返回的是大量候选克隆片段, 而抄袭检测则是软件间的相似性比较, 其要给出软件抄袭与否的判定。

再次, 克隆检测和抄袭检测的关注点也不一致: 克隆检测更重视准确率, 因为太多的误报会引入过多的候选, 时间及人力开销不允许; 而抄袭检测更看重召回率, 特别是对于商业软件而言, 法律诉讼过程漫长, 证据收集的越全面越好。

当然, 克隆检测和源码抄袭检测之间也存在一定关联, 有研究[42-45]将克隆检测技术进行扩展用于抄袭检测, 但通常需要引入深层次的转换和归一化处理, 这容易导致过多的误报, 比如 CCFinder[43]曾用于代码抄袭检测, 但其实际效果还需进一步验证。

源码抄袭检测技术成功应用的前提是被告源码可获取, 而很多情况下如商业软件的源码是不公开的, 所以源码抄袭检测研究的一个典型应用场景是高校学生上机作业的抄袭。由于学生群体的特殊性, 这类抄袭通常是以人工修改方式来实现, 而且通常

不会进行大量复杂的修改。一般来讲, 抄袭手段限定在词法、语法、语义三方面的变换, 而且大部分学生作业的抄袭集中在前两种[46-48]。源码抄袭检测技术由于语义分析的复杂性, 同时考虑到效率问题(通常要分析上千份的编程作业), 很少涉及语义分析, 学生作业抄袭手段简单直接的特点使得源码抄袭检测技术在这类抄袭问题上得到了普遍应用。

简单而言, 源码抄袭检测技术通过匹配相似代码判定抄袭, 其将程序代码当做纯文本分析或对代码采用简单的词法、语法分析, 为方便相似匹配同时提高检测效果, 通常还会对代码进行抽象。依据抽象方式的不同, 本文将现有的源码抄袭检测技术分为基于属性统计和基于结构分析两大类。

3.1 基于属性统计的抄袭检测

基于属性统计(Attribute-Counting)的抄袭检测[49]是最早一批出现的检测方法, 其基本思想是通过对软件代码的各项度量指标进行统计, 将软件转换成一个 n 维向量, 每个维度指定了软件的一项度量指标, 从而将其映射成 n 维笛卡尔空间中的一个点, 最后基于该空间中不同点(软件)间的距离实现抄袭的判定。

最初的方法[50]只使用了基本的 Halstead 度量指标[51, 52], 包括不同操作符个数 n_1、不同操作数个数 n_2、操作符总数 N_1 和操作数总数 N_2 四个度量指标, 从而将程序 P 抽象成一个四元组(n_1, n_2, N_1, N_2), 并且当且仅当两个程序的四元组一致的时候才认为它们存在抄袭。显然这种方法会产生很多漏报, 为此后续工作[48, 53-55]中引入了更多的度量指标, 同时加入了对于度量指标相似性的衡量, 以期待有更好的检测效果。Grier 等[54]在分析 Fortran 程序时引入了变量总数、输入语句、条件语句、循环语句、赋值语句及调用个数等度量指标; Faidhi 等[48]引入了一个含有 23 个不同度量指标的最小集, 这些度量指标间不存在任何关联, 所用指标包括平均标识符长度、保留字个数、条件语句比例, 以及更复杂的结构性指标如 McCabe 圈复杂度、扇入扇出系数等。然而这些努力收效甚微, 属性统计的检测方法因是对整个软件的度量, 其抽象过程抛弃了太多的软件结构信息, 单纯地增加属性的维度并不会有实质性的提升[56], 所以这样的检测系统要么非常不灵敏要么过度灵敏, 会存在很多的漏报和误报, 简单的代码混淆就可以躲避检测。

此外, 也有一些研究[57-61]通过使用其他领域(如数据挖掘、人工智能)的技术或结合其他抄袭检测技术来加强基于属性统计的抄袭检测方法的效果。Moussiades 等[58]利用学生编程作业抄袭时呈现出簇的特点, 提出使用聚类方法来改善基于属性统计的抄袭检测效果。他们首先使用传统的基于属性统计的检测方法计算程序的两两相似性, 并基于一个阈值给出抄袭与否的判定, 存在抄袭的两个程序形成一个抄袭对; 然后将所有的抄袭对转换成一个无向带权图, 图中顶点代表每一个程序, 边代表程序间的相似性; 最后通过聚类识别出抄袭簇, 原本看似无联系的两个程序聚类后有可能会划归到同一个簇中, 从而检测潜在的抄袭。Plague Doctor[57]采用群体学习的思想, 在软件度量指标基础上将其他检测技术(如将要提到的基于 token 的检测方法)的分析结果考虑进来, 将它们作为 BP 神经网络的输入训练分类器达到检测的目的。但该方法限定条件较多, 如需要专业人员事先标定大量的训练数据, 且存在数据稀疏性问题。

3.2 基于结构分析的抄袭检测

软件结构分析需要将代码结构用其他可比较的形式表达出来, 依据表达形式的不同, 将其分为基于 token、基于树和基于图的检测方法来总结。这类技术相比基于属性统计的检测方法而言检测效果更好, 也出现很多实用的检测工具[39-41]。

3.2.1 基于 token 的检测

这类方法是将软件代码转换成一系列 token, 然后通过 token 比较衡量代码相似性。token 可以是编程语言无关的字符串, 也可以是编程语言中的基本元素。

前者是将程序代码当做文本进行分析, 典型代表是 MOSS(Measure of Software Similarity)[39]工具。它利用字符串匹配算法将程序代码划分为一系列 k-gram, 每个 k-gram 是一个长度为 k 的子串(也就是 token); 然后对每个 k-gram 进行 Hash 运算, 通过 Winnowing 算法筛选出部分 Hash 值作为程序指纹, 两个程序共享的指纹越多就越有可能存在抄袭。Brixtel 等[42]提出分段思想将代码首先按行进行切分, 构建两个程序文档的段相似性矩阵, 然后用 Munkres 算法查找段相似性的最大匹配以构建匹配矩阵, 最后通过计算文档间距离来衡量相似性。不同于 MOSS 和 Brixtel 对程序代码进行两两比较的方法, Cosma 等[23]将一批程序作为分析对象, 他们基于所有代码文件构建项-文档矩阵, 其中项是在从代码文件中构造的一系列 token, 然后利用 LSA(Latent Semantic Analysis)计算文档间的相似性以实现抄袭检测; 此外, 他们还提出 PGQT, 将代码段转换为 token 序列后构造查询, 计算该代码段与项-文档矩阵中所有文档的相

似性, 进而依据相似性的分布规律, 评估该代码段作为抄袭证据的贡献度。将程序当做文本分析的一个好处是不受制于分析对象所使用的编程语言, 但也正因为其没有考虑代码的语言特性, 这类方法对代码混淆的抵抗力普遍较弱。

另一类技术[43, 62-64]在对代码token化时考虑了编程语言的基本特性, 其基本思路是对代码进行词法扫描构建 token 字串, 然后通过 token 字串的比较决定程序对的相似性。这类方法的关键在于 token 的选择和比较算法的设计, 筛选的 token 应当尽可能不太容易被简单的代码混淆破坏掉。YAP3[65]将程序转换成 token 字串, 并使用 RKR-GST(Running-Karp-Rabin Greedy-String-Tilling)[66]字串匹配算法实现 token 字串的最大非重叠匹配; 为提高应对代码混淆的能力, 其在 token 化时采取了一系列优化, 包括移除字符串常量和注释、将字符全部转为小写、将函数映射成等价形式(比如 *strncmp* 统一为 *strcmp*)等。Sim[67]通过 flex 词法分析器, 将程序解析成 token 序列, 然后应用字串对齐算法并设计对齐得分机制实现相似性计算; 其 token 由预定义的关键字、特殊符号以及动态指定的标识符组成, token 流的对齐采用分段思想按程序模块实施。JPlag[68]是另一款比较著名的检测系统, 其使用了与 YAP3 同样的字串匹配算法, 但与前两者不同其被设计成基于 web 服务的检测系统。此外, 为减少偶然相似, JPlag 会将特定的程序结构转换成 token, 比如其使用“*BEGIN_METHOD*”这个 token 表示一个方法的开始大括号, 而用“*OPEN_BRACE*”表示其他的开始大括号。

JPlag 等需要针对每门编程语言单独设计前端以实现代码的解析和 token 化, 为此有研究[69-71]将代码转换成中间代码再进行分析。XPlag[69]利用 GCC 编译套件将软件代码转换成 RTL(Register Transfer Language)中间代码, 在中间代码上进行 token 化; 同时为提高检测效率, 其将所有代码 token 化后应用倒排索引实现 token 信息的存储, 并通过构造查询, 利用搜索引擎查找最相似的程序; 此外不同的编程语言转换成中间代码后会具有相同形式, 其在 Java 和 C 程序上的实验初步表明 XPlag 可以检测跨语言的抄袭。该方法很大程度上依赖于编译套件的可靠性, 同时跨语言抄袭检测也主要在小部分 Java 和 C 程序上进行了验证, 其实用性有待进一步研究。

对于 token 流的比较, 除了普遍采用的字符串匹配、比较或对齐算法外, 也有研究[59, 61, 72, 73]结合其他领域知识实现 token 流相似性的衡量。Chen 等[72]从信息论的角度, 提出一项具有 Kolmogorov 复杂度[74]的普适性距离度量指标来衡量两个token流的信息共享量, 并基于归一化后的信息共享量判定抄袭。而 Kolmogorov 复杂度具有不可计算性, 为此设计 SID (Software Integrity Diagnosis system), 通过启发式的压缩算法来逼近这个度量。Zhang 等[59]也提出了一项具有 Kolmogorov 复杂度的普适性信息距离度量指标, 并基于 LZ77 压缩算法解决 Kolmogorov 复杂度的不可计算问题; 此外通过对学生作业多对多分析得到相似性矩阵后, 他们利用 Shared Near Neighbors 聚类算法[24]进行抄袭群体的挖掘。Ciesielski 等[61]则从演化计算的角度, 将粒子群算法用于改善已有的 token 流相似性度量方法, 同时基于遗传算法提出了更优的 token 流相似性度量方法。

基于token的抄袭检测方法通过文本结构分析和词法分析度量程序代码的相似性, 时空复杂度较低, 适用于大规模软件代码的抄袭检测或代码的批量检测。此外其对代码混淆如轻微的代码重排、变量重命名等具备一定的抵抗力, 但仍旧难以应对稍微复杂的代码变换, 如冗余代码植入、控制流混淆等。

3.2.2 基于树的检测

这类技术是将软件代码转换成树型结构[75]进行分析。Son 等[76]利用 ANTLR 将软件代码转换为解析树, 通过子树划分及频数统计构造解析树内核, 并基于解析树内核的内积运算和归一化实现相似性计算。解析树通常包含过多的冗余节点, 为此很多研究[77, 78]在解析树基础上进一步通过约减, 构建程序的抽象语法树(Abstract Syntax Tree, AST)。AST 可以有不同的形式, 其比较方法也有很多种。Guo 等[79]借助 lex 和 yacc 构造程序的 AST, 通过自底向上的累积运算和 AST 遍历为每个节点计算一个 Hash 值, 并通过节点的 Hash 值匹配和匹配的节点比例衡量相似性。这种子树搜索或节点间匹配的方法难以应对代码重排及控制流混淆, 为此更多的研究是将 AST 转换成 token 序列或字符串来处理。

BUAA_AntiPlagiarism[80]在构建 AST 后, 通过线性化处理将 AST 转换成 token 序列比较; 其用节点名称标识每个节点, 通过前序遍历将 AST 转换成字符串, 并利用 k-gram 算法切分成一系列 token, 最后利用 Jaccard 系数计算相似性。此外, 该方法对 AST 进行了一系列优化, 比如将代码转换成 CIL 中间代码后再生成 AST 来简化分析, 通过冗余及无关节点的移除、关联节点的合并来提高检测效果。Resmi 等[81]在构造 AST 时使用了修改的语法, 他们对 AST 进行前序遍历生成节点序列, 然后通过 Needleman-Wunsch 算法和最长公共子序列算法衡量相似性。Ji

等[82]则提出 PST(Program Static Tracing)方法, 其通过词法语法分析构造程序的解析树和符号表, 进行程序语法层面的静态执行, 按照函数的执行次序对预定义关键字进行抽取, 将解析树转换成token序列, 最后利用自适应局部对齐算法实现相似性计算。

XPDec[83]则是首先将代码转换成XML文档形式, 然后通过特定的映射关系提取XML的树结构以构造类型矩阵(Decimal Frame Matrix, DFM), 并通过对XML执行XQuery查询构造控制矩阵(Decimal Control Matrix, DCM), 最后通过矩阵运算计算相似性。EXPDec[84]是该方法的扩展, 它克服了 XPDec 难以处理迭代的控制结构的短板, 并且使用 Levenshtein 距离实现相似性计算。利用XML的树形结构建模程序语法, 可以有效应对代码重排, 然而因其要求编程语言具备较好的结构, 目前只能有效建模 C 或 Pascal 之类的过程化语言。

3.2.3 基于图的检测

基于图的抄袭检测方法[25, 85]相对较少, GPLAG[25]从软件代码中构造程序依赖图(Program Dependency Graph, PDG), 通过子图同构匹配实现相似性计算; 此外, 为提高检测效率, 文章提出有损过滤机制和基于 G-test 假设检验的过滤机制约减子图搜索空间。PDG 由于捕获了程序的数据和控制依赖关系、编码了程序逻辑, 抄袭者很难在不理解代码的前提下对PDG作出修改, 实验证实其相比其他方法能更有效应对多种人工混淆手段。然而自动化混淆工具的混淆能力更强, 抄袭隐藏手段更高明; 此外, PDG 的构建过程代价很高, 而且子图匹配是NP问题, 难以分析较大规模的软件。总的来说, GPLAG 是从学术抄袭到软件产品抄袭的初步实践, 它考虑了较为复杂的人工混淆手段, 但它对源码的需求及较高的时空花销, 使其不足以应对软件产品的抄袭。

3.3 源码抄袭检测技术分析比较

本节对各类源码抄袭检测技术进行总结和比较, 见表 1. 从表中可以看出, 每类技术都有各自的技术特点及优缺点, 在检测精确性、抗混淆能力、计算的时空复杂度等方面有不同的表现, 特别的基于 token 的检测方法出现很多实用化的工具和系统, 如 Moss、JPlag、YAP3 等, 是目前实际应用中被普遍采用的方法。不过总体而言, 目前源码抄袭检测技术对代码混淆的抵抗力不够, 特别是难以对抗自动化的语义保留的代码混淆, 同时对源码的依赖使其难以用于软件产品的抄袭检测。

表 1 各类源码抄袭检测技术的比较

分类	检测精确性	抗混淆能力	时空复杂度	代表技术和工具
基于属性统计	低; 过度抽象, 误报漏报高	很低; 抛弃了太多的软件结构信息, 基本不抗混淆	低; 统计特性提取开销小	Ottenstein's system[50], Fadihi-Robinson's system[48], Accuse[54], PDetect[58], Plague Doctor[57]
基于token	中低; 依赖于token的选择和提取	低; 能抵抗变量重命名、代码重排等混淆, 但难应对冗余代码植入、控制及数据流混淆等稍复杂的混淆手段	低; 以文本结构及词法分析为主, 解析速度快	Moss[39], JPlag[40], Sherlock[62], SIM[67], YAP3[65], Plague[63], XPlag[70], PGDT[23], SID[72]
基于树	中; 依赖于树的精炼程度	中; 考虑了语法及结构特征, 但难应对代码重排、语句拆分、控制流混淆等	中高; 树的构建成本高	Parse Tree Kernel[76], AST[77], PST[82], XPDec[83]
基于图	高; 依赖于图的精炼程度	高; 综合了考虑程序的语法和语义特征, 有效抵御布局混淆, 但难抵御部分数据及控制混淆, 如控制流扁平化、变量拆分等	高; PDG 构建代价很高, 子图匹配是 NP 问题	GPLAG[25]

4 基于软件水印的抄袭检测技术

很多情况下被告是以可执行文件的形式存在, 在搜集到足够证据前, 抄袭者不太可能给出其源码。软件水印技术可以处理这种情况, 其基本原理是在软件发布之前, 在代码中植入一个特殊的标识符(水印), 该水印可以承载软件作者、版权等信息, 事后可以通过特殊的提取器将其从被告软件中识别或抽取出来作为证据以达到检测的目的。软件水印技术的基本原理如图 1 所示。

已有研究对软件水印技术的分类法很多, Collberg 等[86]按照水印功能的不同划分为身份水印、指纹水印、校验水印及许可水印四类, 张等[87]按照水印加入位置的不同将其分为代码水印和数据水印, 其他研究[88]也有按照水印强弱、是否可见等进行划分。考虑到水印技术的关键在于植入和提取两个阶段, 本文按照提取技术的不同划分为静态和动态水印, 进而依据具体植入方法的不同分类总结。

4.1 静态水印技术

静态水印技术[89]是将水印植入到可执行程序的

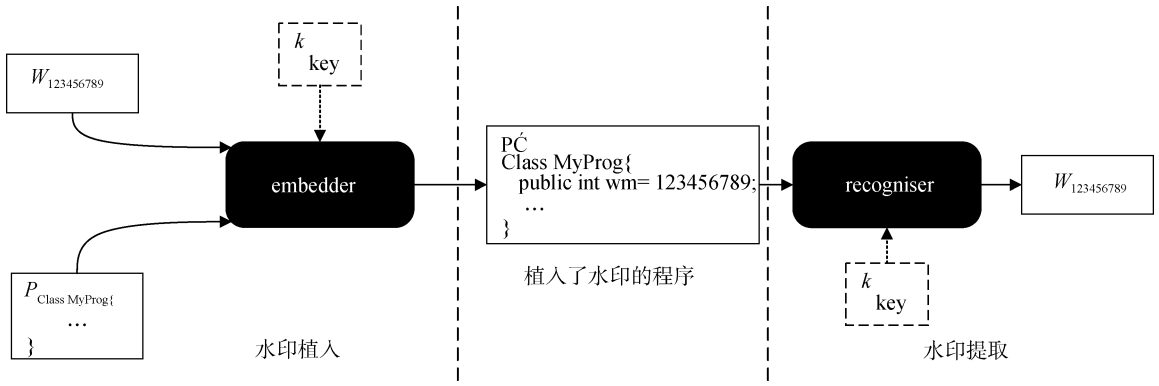


图 1　软件水印技术的基本原理

代码或数据中, 其提取过程不需要运行程序, 通过静态分析完成识别或提取。下面依据植入策略的不同对静态水印技术分类介绍。

4.1.1　代码替换法

这类方法[90-95]的基本思路是将程序中特定的代码或数据片段用软件水印进行替换, 从而嵌入水印。DMI[92]是这类技术的代表, 其通过在虚假函数中替换字节码指令从而在 Java 程序中植入水印, 其中虚假函数是人为在代码中添加的, 同时利用不透明谓词保证这些虚假函数永不会被执行。DMI 首先将水印信息先转换为 bit 序列, 并为特定的字节码指令分配对应的 bit 码, 从而将水印用字节码指令编码出来, 最后用这些字节码指令替换虚假函数中原有的指令完成水印的植入。提取水印时, 只要知道字节码指令与 bit 码的对应关系, 以及 bit 码跟水印信息的对应关系, 采用与水印植入逆向的流程即可。DMI_{SM}[96]是 DMI 的改进版, 其虚假函数构建过程实现了自动化。然而, 它们都容易遭受合谋攻击, 也就是攻击者通过对多份植入水印的程序进行统计分析就可以定位水印的位置, 因为这种方法难以生成与原程序相同的代码。

为此, Fukushima 等[93]将 DMI 跟混淆技术相结合, 他们在对植入水印后会对每份软件再单独进行混淆, 攻击者在实施共谋攻击时, 就会发现软件中呈现出多处差异, 从而难以定位水印位置。但总体而言, 代码替换法植入的水印隐蔽性不好, 而且简单的代码混淆就可以将水印破坏掉。

4.1.2　代码重排法

这类方法[91, 96]的基本原理是将水印转换成数字, 然后通过代码重排编码这个数字。DM 算法[96]利用基本块重排植入水印, 其首先将水印转换成一个数字 w, 对某个(或某些)方法的控制流图(Control Flow Graph, CFG)中的基本块集合 B 作 w 次排列形成新的基本块集合 B', 然后将 B' 的基本块重新链接到原程序中从而替换掉原有基本块 B。水印的提取过程则是通过比较原有基本块的次序和 B' 的次序来获取排列次数 w, 然后将 w 转换回水印。Anckaert 等[97]实现了 DM 算法的一个变种, 不同于 DM 是对多个基本块进行重排, 他们是对多个基本块链进行重排, 这使得其植入的水印比 DM 算法植入的水印具备更好的隐蔽性。

除了利用基本块重排植入水印外, Shirali-Shahreza[98]提出基于数学方程式中对称的数学运算进行重排, Sha 等[99]利用操作数系数重排来编码水印, Gong 等[100]则利用 Java 程序常量池的特点, 通过对 Java 类中常量池的重排植入水印。以 DM 为代表的这类方法植入的水印对代码混淆的抗毁性很弱, 一旦攻击者实施混淆, DM 就难以找到植入了水印的方法, 而水印提取的前提是必须找到实施了重排的基本块所在的方法。一种可能的方案是检查原始程序和植入水印后程序的所有方法对, 但这一方面会导致过高的开销, 另一方面会导致水印提取程序生成很多错误的水印。

VEP[101]将程序当做一个完整的统计对象, 对指令组进行统计分析构建特征向量 C, 将水印构造成同维特征向量 W, 并利用指令重排对程序进行受控的修改, 使得修改后程序的统计特征向量 $\tilde{C}$ 满足 $\tilde{C} = C + W$。VEP 植入的水印无法提取, 只能验证水印是否存在。

4.1.3　寄存器分配法

寄存器分配技术[102]是实现软件水印植入的另一类技术, QP 算法[103]构建程序的冲突图(Interference Graph), 并依据水印转换成的 bit 码值向冲突图中相应节点间追加虚边, 然后利用图着色(Graph Coloring, GC)技术对冲突图着色以实现寄存器分配。不过 QP 算法有一个最大问题是植入的水印并不总是可以提取的, 不同的水印植入后也可能会生成完全相同的冲突图。Myles 等对 QP 进一步改进提出 QPS 算法[104],

它查找冲突图中的 Triples(不影响其他顶点且相对孤立的顶点三元组), 并保证植入水印时只会对这些 Triples 顶点添加虚边。实验表明, QPS 算法植入的水印具备更优的隐蔽性和可提取性, 然而由于它只能利用部分顶点, 植入的水印的信息量有限, Thomborson 等通过对虚边添加过程进一步优化提出了 QPI 算法[105]。

除了对冲突图进行改变外, CC(Color Change)和 CP(Color Permutation)[106]算法则设计着色函数, 可以不改变冲突图结构而是只改变顶点颜色, 这使得它们相比 QPS 以及 QPI 能够植入更大的水印。SCC[107](Selected Color Change)进一步优化了 CC 算法的效率, 其只有当 bit 码为 1 时才改变颜色。

4.1.4 静态图法

GTW[108](Graph Theoretic Watermarking)将水印编码到程序 CFG 的拓扑结构中, 其基本思想是将水印值用一个控制流图 G(简便起见称作水印图 G)表示出来, 然后将水印图 G 合并到原程序 P 的 CFG 中; 为保证水印的隐蔽性, 水印图 G 应该尽可能接近真实的 CFG, 此外 GTW 通过随机游走在图 G 节点和原程序 P 的节点间添加虚假边使 G 与 P 紧耦合, 从而进一步提高水印的隐蔽性。为了事后能够提取 GTW 植入的水印, 需对水印图 G 的节点进行标记, 以从水印图 G 还原出水印信息。

图 2 给出了 GTW 方法的示意图。Collberg 等对 GTW 在 Java 字节码上进行了实现, 称作 GTW_{SM}[109, 110]。GTW_{SM} 提供了两种算法将水印进行分割从而可以编码到多个图 G 中, 然后自动地为每个图 G 生成相应代码并合并到原有程序中。实验表明 GTW_{SM} 可以编码任意大的水印, 但植入水印后程序尺寸增大了 40%-75%, 性能下降了 0%-36%; GTW_{SM} 编码水印用的是 RPG(Reducible Permutation Graph), 尽管 RPG 跟程序真实的 CFG 非常相似, 但攻击者还是比较容易找到这些 RPG 图, 因而隐蔽性并不好, 同时简单的混淆攻击如基本块划分等就可以破坏水印, 导致水印无法提取。

4.1.5 不透明谓词法

不透明谓词是诸如 $x^2 \geqslant 0$, $-3y^2-1==x^2$ 这类结果已知的一种谓词, 它们较难通过自动化分析识别出来, 前边提到的 MON 算法就是利用不透明谓词向代码中添加虚假函数来提高水印的隐蔽性。Arboit[111]提出了两种基于不透明谓词的水印技术, 其基本思想是将水印编码到不透明谓词中, 然后在特定的分支点将这些谓词添加到原程序代码中。为了能够编码较大的水印, 两种方法均将水印值切分成 k 个

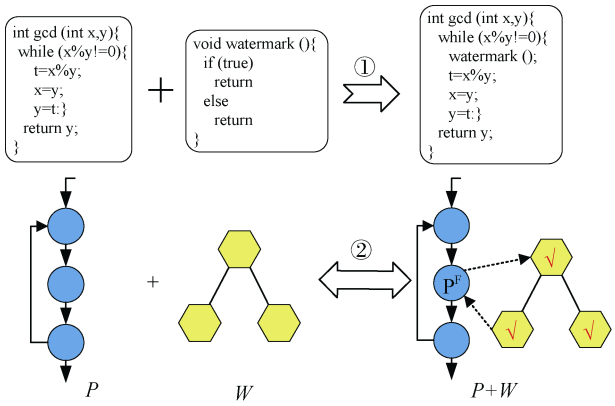


图 2 GTW 水印植入示意图

整数, 然后用不透明谓词中的常量值分别对这 k 个整数进行编码, 比如 $-3y^2-1==x^2$ 编码了整数 4; 两种方法的区别是一个采用不透明谓词编码水印(称作 GA1), 一个则采用不透明函数进行编码(称作 GA2)。对于水印的提取, 需要查找所有不透明谓词/方法, 然后解码成水印。Myles 等[112]对这两种方法进行了实现和评估, 发现 GA1 在性能、抗毁性、隐蔽性方面均优于 GA2, 但两种方法植入的水印还是较容易被混淆攻击破坏掉。

4.1.6 抽象解释法

Patrick 等[113]提出了基于抽象解释的软件水印技术, 其原理是使得植入的水印只有在知晓所采用的具体抽象及抽取策略前提下进行的抽象解释才能提取。具体而言, 该方法将水印分割以编码成简短的代码段, 然后紧密植入到程序原有函数的声明、初始化及函数体代码中; 提取时则采用了常量传播分析的抽象解释方法来提取水印。此外, 该方法还结合混淆及参数抽象手段进一步提高水印的隐蔽性。

综上所述, 静态水印技术是关注如何将水印隐藏到程序代码或数据中, 同时保证通过静态分析能提取出来。对于上述水印技术, 按照植入原理的不同可以进一步概括为两类: 通过代码排序植入(代码重排法)和通过追加代码植入(代码替换、寄存器分配、图水印、不透明谓词、抽象解释法)。

4.2 动态水印技术

动态水印技术[114]是将水印植入到程序的执行过程或运行状态中, 也就是说其并不是直接植入水印本身, 而是植入编码了水印信息的额外代码或数据, 在软件执行过程中可以将水印动态表达或构建出来。本文将现有主流动态水印技术归纳为以下几类。

4.2.1 数据结构法

Collberg 等提出 CT 算法[15, 115, 116], 将水印植入到动态构建的图结构中。具体流程如图 3 所示, 首先

将水印编码到一个图 G 中, 为提高隐蔽性切分成多个子图, 然后在程序的某条路径上插入能够动态构建这些子图的代码; 水印的抽取则是给定特定输入执行该路径时, 图G会从堆数据中被构建出来, 最后再转换回水印。

对于水印到图G的编码方式, CT 提出了三种[117]: 枚举图、基数图和排列图。

枚举编码是将水印值W编码成图G(在某个可枚举图结构中)对应的索引值, 具体的可枚举图结构可以使用有向父指针树(Directed Parent-Pointer Tree, DPPT)、平面立体种植树(Plantted Planar Cubic Tree, PPCT)等。图 4(a)给出了 7 个节点的 DPPT 的第 1、2、22 及 48 次枚举树, 假定要编码的水印值为 22, 则将能动态构造第 22 次枚举树的代码植入原程序即可。Zhu 等[118]提出了两种改进的枚举图结构 PIPPCT 及 MIPPCT, 其水印植入相比 PPCT 具有更好的性能。

排列编码是将水印值 W 转换成一系列整数的排列$\pi=\langle p_1,\cdots,p_n\rangle$, 然后将这个排列编码进单项循环链表中, 具体地其通过向该链表添加节点i到节点p_i的边形成排列图。Venkatesan 等[119]利用可约减排列图 RPG(图 4(b)), RPG 更接近程序的控制流图且具备纠错能力, 所以 RPG 植入的水印对 edge-flip 等混淆攻击具备更好的抵抗力。

基数编码同样利用了循环链表, 具体而言每个节点的右指针指向下一个节点, 左指针指向节点到该节点的距离则编码了基数为k的整数, 这样长度为k的链表可以编码$0\cdots(k+1)^k-1$的任意整数, 这种方式生成的图称作基数图。图 4(c)所示的基数 6 编码图编码了水印值为 4453 的水印。基数图的左指针指向相比排列图的指针具备更好的灵活性, 所以基数编码植入的水印会具备较弱的纠错能力, 但会拥有更高的码率。此外, 也有研究[120]将基数图转换成类似于 PPCT 的结构以改善纠错能力。

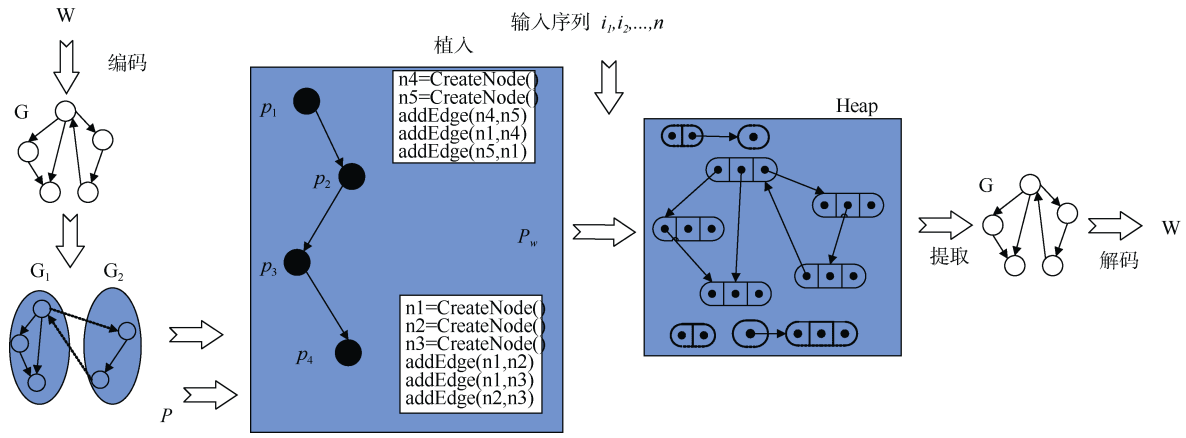


图 3　CT 动态水印技术的基本流程

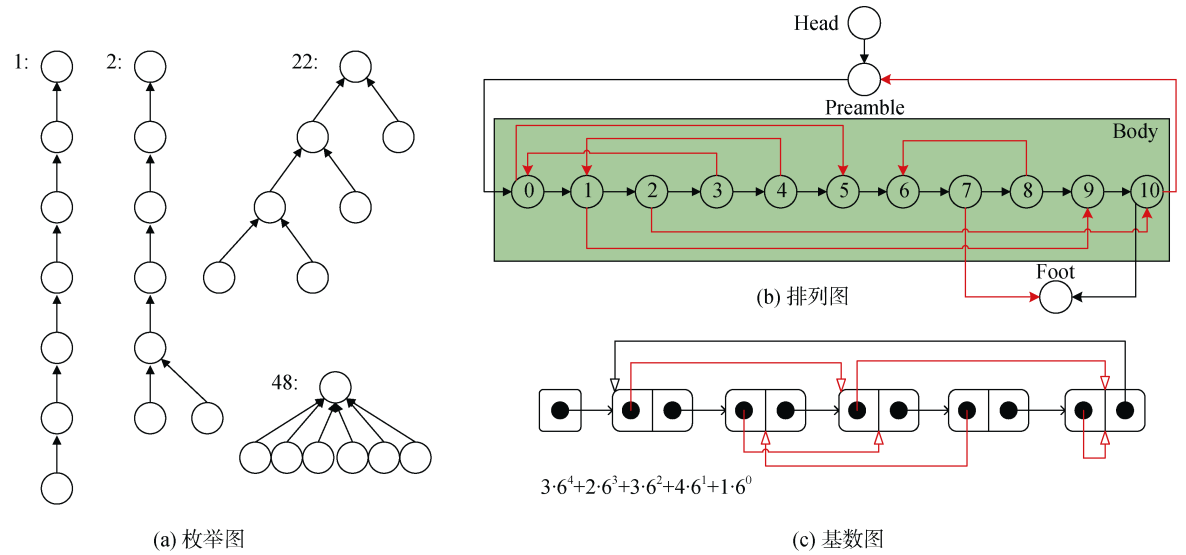


图 4　水印的图编码

除了利用堆数据结构以外, Kamel 等[121]提出利用基于磁盘的数据结构 R-tree 编码水印, 其利用 R-tree 节点内词条存在冗余及无序限制的特点, 通过节点内词条的重排编码水印值。

4.2.2 条件分支法

CBW[122]将水印编码到程序的动态分支结构中, 其基本思路是用条件分支的动态行为(取 False 或 Ture 分支)编码水印值, 捕获程序在特定输入(或输入序列)下的执行轨迹, 并在该轨迹的恰当位置植入编码了水印值的条件分支语句; 水印提取过程则是用相同的特定输入(或输入序列)执行程序, 捕获编码了水印值的执行轨迹, 通过分析其条件分支的动态行为识别出水印。为提高隐蔽性和效率, Collberg 采取了一系列优化措施, 如利用中国剩余定理(Chinese Remainder Theorem)将水印值分解为多份散布到程序多个位置植入, 通过轨迹分析将条件分支语句植入到非频繁执行点等。Myles 等[123]结合代码混淆和防篡改技术的部分思想, 将非条件分支和条件分支均转换成分支函数, 分支函数一方面实现原有控制转移功能, 另一方面负责水印编码。

4.2.3 线程争用法

TCW[124]是基于线程竞争实现的水印技术, 其在原程序中的单线程部分引入新的线程, 并使用锁机制保证线程切换始终受控, 从而将水印编码到线程切换行为中。比如对于线程化部分被执行的两个基本块, 如果是在不同线程中被执行, 则用这种行为编码比特 0; 如果是在相同线程被执行, 则用这种行为编码比特 1。通过将水印转换成比特码, 就可以编码到基本块序列中。此外, 对于线程化部分的选择, Thomborson 等通过动态轨迹追踪避开频繁执行代码, 同时对于水印的植入也只选用不会被频繁执行的基本块进行编码, 这样可以大大减少水印植入所带来的性能开销。但新引入的线程代码需要进行非常精心的设计以保证程序正确性, 此外初步试验表明植入水印后软件的性能下降 2~8 倍, 尺寸也相比原来有很大的增加。

4.2.4 运算法

前几类方法需要先将水印用图结构或者指令模式等进行编码, 然后向原程序中添加能够动态生成这些图或指令模式的代码。运算法与之不同, 其忽略了将水印转化为其他结构或模式这一过程, 而是直接向原程序中添加代码, 这些代码在特定输入下通过运算直接将水印输出。

LSW[125]利用循环结构植入水印, 其在既有循环体代码语义分析基础上增加额外代码, 这些代码对原循环体代码存在数据或控制依赖关系, 且在特定输入(对应特定的循环展开)下会计算出水印值。HFW[126]通过构造 Hash 函数对水印值进行编码, 并通过替换特定输入下执行路径上的常量将 Hash 函数添加到到原代码中, 当执行相同的特定输入时, Hash 函数就会被执行并输出水印值。ROPW[127]利用 ROP(Return-Oriented Programming)技术从既有代码中构造水印代码, 链接到 ROP 的执行路径同时隐藏到数据区中; 只有在特定输入下才会动态地恢复出隐藏在堆数据中的 ROP 执行路径, ROP 路径的执行最终输出水印。ROPW 采用数据区而非代码区隐藏水印, 使其具备非常好的隐蔽性。

运算法由于不使用第三方表示编码水印, 其在植入较大水印的同时带来的性能开销要低于其他方法; 然而正是由于忽略水印编码的过程, 这类技术在设计时必须使水印代码跟原代码紧密耦合以保证隐蔽性。

4.3 基于软件水印的抄袭检测技术分析比较

本文从隐蔽性、抗毁性、码率及性能开销四个维度对各类水印技术进行对比分析(见表 2)。隐蔽性刻画了水印躲避攻击者识别和定位的能力; 抗毁性刻画了水印对各种攻击如代码混淆等的抵抗能力, 也就是要保证在各种攻击下水印不会被破坏, 依旧可以可靠地识别和提取; 码率刻画了水印技术植入大型水印的能力; 性能开销刻画了水印技术植入和提取水印的复杂程度, 以及水印植入后所带来的性能降级、尺寸增加的程度。四个特性间通常存在矛盾关系, 任何水印技术都需要根据实际需求对四个特性进行权衡。

5 基于软件胎记的抄袭检测技术

软件水印技术通过识别事先植入的水印来检测抄袭, 然而水印的植入不仅会引入额外的时空开销, 导致程序性能下降, 甚至会引入新的软件缺陷和漏洞, 影响软件的正确性和安全性。此外, 当前的水印技术对语义保留的代码混淆攻击的抵抗力都较弱, 经过适当的努力攻击者始终可以破坏任何水印[122]。为解决上述问题, 研究者提出了基于软件胎记的抄袭检测技术。

相比水印是事先植入到软件中的额外数据, 软件胎记是指从软件中可提取的一系列特征, 这些特征刻画了软件自身的固有属性, 并且可以唯一地标识这个软件。给定待检测的被告软件, 胎记技术通过判断其与原告软件的胎记是否一致判定抄袭。然而真实的情况是, 即使软件 Q 抄袭了 P, 它们的胎记也

表 2 各种软件水印技术的比较分析

动/静态	植入策略	隐蔽性	抗毁性	码率	性能开销	代表技术
静态	代码替换法	弱, 易被合谋攻击定位	弱, 非常容易被消减攻击或代码重排等混淆攻击破坏	高, 可植入任意尺寸的水印	对程序的尺寸及性能影响很小, 植入和提取都较容易	DMI[92], DMI_{SM}[96]
	代码重排法	弱, 重排引入过多 GOTO 指令, 易被检测	弱, 简单的再编译攻击或混淆可使水印难以提取	中, 受可重排基本块或指令数目限制	对程序的尺寸及性能影响小, 植入和提取的复杂度一般	DM[91], VEP[101]
	寄存器分配法	高, 极难定位	弱, 对变量拆分、方法合并等混淆无抵抗力; 同时算法的设计决定水印并不总可以提取	低, 代码越复杂能够植入的水印越小	对程序尺寸及性能影响很小, 但植入和提取的复杂度高	QP, [103] QPS[104], QPI[105], CC&CP[106], SCC[107]
	静态图法	中, RPG 与实际 CFG 存在一定差距, 通过静态分析易对水印进行定位	中, 基本块划分、方法合并等混淆可破坏水印, 导致无法提取	高, 可植入任意尺寸的水印	对程序尺寸及性能影响大, 植入和提取的复杂度一般	GTW[108], GTW_{SM}[110]
	不透明谓词法	中, 与正常程序存在统计指标上的差异, 但难具体定位	中, 不透明谓词混淆、布尔表达式拆分等将导致水印无法提取	中, 依赖于程序中判断语句的个数	对程序尺寸及性能影响低, 植入和提取的复杂度低	OPW[111], OMW[112]
	抽象解释法	高, 与程序语义紧密耦合, 难以定位	中, 可抵抗大部分混淆攻击	高, 可植入任意尺寸的水印	对程序尺寸及性能影响一般, 植入复杂度一般, 提取开销近似于编译时间	AIW[113]
	数据结构法	中, 与正常程序存在统计指标上的差异, 但难具体定位	中, 节点拆分混淆等可破坏水印	高, 可植入任意尺寸的水印	对程序尺寸影响一般, 性能影响大, 植入和提取的复杂度一般	CT[15, 116], R-Tree[121]
动态	条件分支法	中, 与原程序代码紧耦合, 静态合谋攻击无效, 但易被动态合谋攻击定位	中, 对分支注入混淆抵抗力低	高, 可植入任意尺寸的水印	对程序尺寸影响一般, 性能影响较小, 植入和提取复杂度一般	CBW[122], BFW[123]
	线程争用法	高, 极难定位	高, 随机线程切换注入可能会破坏水印	高, 可植入任意尺寸的水印	对程序尺寸及性能影响大, 植入复杂度很高, 提取复杂度一般	TCW[124]
	运算法	中高, 隐藏到数据区或散布到整个程序, 较难定位	中低, 数据混淆易破坏水印	高, 可植入任意尺寸的水印	对程序尺寸及性能影响极小, 植入复杂度一般, 提取复杂度很低	LSW[125], HFW[126], ROPW[127]

并不一定会完全一致; 因为软件胎记是个理想化的概念, 实际中很难保证抽取的胎记切实地刻画软件自身固有的难以改变的属性。为此, 在实际应用时, 胎记技术是通过衡量胎记的相似性实现抄袭的判定, 这类似于源码抄袭检测技术, 但胎记技术的操作情景是默认被告源码无法获取, 同时应对的是更复杂更苛刻的抄袭手段。图 5 给出了胎记技术的基本流程。

软件胎记技术的关键在于软件胎记的抽取和胎记相似性的衡量。为构造高质量胎记, 应使抽取的胎记尽可能接近程序的语义行为, 从而不容易被代码混淆破坏掉, 按照胎记抽取是否需要程序运行, 现有的软件胎记可以归纳为静态胎记和动态胎记两类; 胎记相似性的衡量主要依赖于胎记的具体形式, 可以划分为序列、集合和图形式胎记。本文将根据胎记构建方法的不同对其进行分类介绍。

5.1 静态胎记技术

5.1.1 属性分析法

最初的软件胎记主要是通过分析软件的词法及结构特性抽取出来的, 其利用构成软件的基本要素如指令、API、系统调用等, 而较少考虑语法语义信息。

Tamada 等[128, 129]率先提出软件胎记的概念, 并针对 Java 程序实现了四种不同的静态胎记, 包括 CVFV, SMC, IS 以及 UC; 它们分别从类常量字段、标准 API、继承结构以及标准类的使用情况四个角度对 Java 类进行特征化, 将每种胎记表示成对应基本属性的序列, 比如某个类的 UC 胎记就是由该类所使

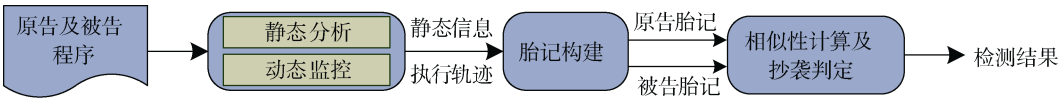


图 5　基于软件胎记的抄袭检测方法的基本流程

用的所有标准类按字母序排列成的序列。Myles 等[130]则使用整个指令序列标识软件, 鉴于序列太长难以直接比较, 其利用 k-gram 算法对软件指令序列进行切分, 然后将所有 gram 构成的集合作为胎记, 原告与被告胎记通过 Containment 集合运算实现相似性度量。Myles 的胎记没有考虑 gram 的频率特性, Xie 等[131]在此基础上将 gram 的频率引入胎记生成中, 提出了带权 k-gram 胎记, 但检测效果的提升幅度并不明显。

AVSB[132]使用函数参数及局部变量个数作为函数胎记, 然后将所有函数胎记构成的序列作为程序胎记; AVSB 使用半全局比对(semi-global alignment)算法衡量序列的相似性, 或者将序列转换成 k-gram 后基于集合运算实现胎记的相似性计算。Choi 等[133, 134]提出了基于系统调用的静态胎记 WSCB, 其构建程序的调用图, 然后将每个函数中使用的以及调用图中距离该函数深度 k 步以内的函数中使用的所有系统调用构成的集合作为该函数的胎记, 并将所有函数胎记的并集作为软件胎记 WSCB; Jaccard 系数被用来衡量函数胎记的相似性, 最后通过最大双边图匹配实现 WSCB 胎记相似性的计算。

此类胎记技术单纯利用构成程序且较难改变的基本元素, 抛弃了过多的语法语义信息, 因此难以对抗甚至十分简单的代码混淆, 如指令重排、替换、垃圾代码植入、函数内联外联等。

5.1.2　静态控制流法

Taisook 等[135-140]提出了一系列基于控制流分析的静态胎记, 其基本思想是在控制流分析引导下尽可能模拟软件真实的运行时行为, 以使生成的胎记更贴近软件的行为特征, 从而不容易被混淆等破坏掉。

FPB[135](Flow Path Birthmark)构建每个方法的 CFG, 其定义一条 Flow Path 为 CFG 中从任意节点出发 k 步可达的基本块构成的路径, 将从程序所有方法中提取出的所有 Flow Path 构成的集合作为该程序的 FPB 胎记; 对于 FPB 的相似性计算, 其用某 Flow Path 上所有指令构成的序列表示该 Flow Path 的行为, 并利用用半全局比对算法实现 Flow Path 行为的比较, 得到原告与被告所有 Flow Path 的两两相似性矩阵, 最后基于双边图匹配实现原告与被告 FPB 的相似性计算。CFEB[136]则将 CFG 中所有控制流边构成的集合作为胎记, 并用每条边所连接的两个基本块中所有指令构成的序列刻画该控制流边的行为; CFEB 采用了最长公共子序列 LCS 衡量控制流边行为的相似性, 并采用双边图匹配实现原告与被告软件 CFEB 胎记的相似性计算。

SFB[137]是模拟 Java 程序操作数栈的运行情况提出的静态胎记, 其基于 CFG 扫描所有可能的执行路径, 然后对于每条路径, 分析该路径上每条指令执行前后的操作数栈的状态(栈深度), 将指令及其执行后的栈状态共同构成的序列称作一条 Stack Flow, 并将所有 Stack Flow 的集合作为软件胎记; SFB 也是利用半全局比对获得 Stack Flow 的两两相似性矩阵, 再利用最大双边图匹配计算原告与被告 SFB 的相似性。WSPB[138]也是模拟 Java 操作数栈的运行情况, 其将整个指令序列按照栈深为 0 切分成一系列 Stack Pattern, 同时其并不平等对待 Stack Pattern 中的所有指令, 而是利用 TF-IDF 计算每条指令的权值, 并将所有的带权 Stack Pattern 构成的集合作为软件胎记。

WSPB 为不同指令赋权的思想体现了不同指令对于刻画程序行为贡献度不一的问题, OTB[139]仅利用 Java 程序中与对象操作相关的 11 条特定指令生成胎记, OTB 在 CFG 基础上通过移除所有与对象操作无关的指令构建对象流图(Object Flow Graph, OFG), 其挖掘 OFG 中所有长度为 k 的序列, 并将整个程序中挖掘出的所有序列作为软件胎记; OTB 采用局部比对实现序列的两两比较, 并同样采用双边图匹配实现原告与被告胎记的相似性计算。除了考虑指令对于刻画软件行为贡献不一外, SMPB[140]只考虑可以反映程序行为且较难修改的关键路径, 其分析构成 CFG 的各项分支、循环等结构, 寻找最可能的执行路径, 并将该执行路径中出现的所有系统调用构成的序列作为软件胎记; SMPB 的相似性计算采用 Smith-Waterman 序列对齐算法。

5.1.3　静态语义分析法

Taisook 提出的这些基于控制流分析的静态胎记, 通过 CFG 蕴含的指令流、操作符栈状态流、标准 API 流等从不同角度刻画程序行为, 本质上都是通过尽可能模拟程序真实的运行状态以保证抽取的胎记与程序的语义行为更贴近, 从而提高胎记的抗毁性; 然而, 这些基于控制流分析的静态胎记仍然容易被

简单的代码混淆手段如不透明分支植入、基本块拆分、垃圾指令植入等破坏掉。

Cop[12]是目前唯一一种基于严格语义分析的静态胎记技术, 其基于 CFG 获取所有的线性独立路径, 并用路径上蕴含的基本块序列刻画程序行为; 对于每个基本块, Cop 通过符号执行将其表示成蕴含了输入输出关系的一系列逻辑公式, 并利用自动定理证明器来衡量两个基本块的语义相似性, 然后基于最长语义等价基本块子序列衡量原告与被告中任意两条线性独立路径的相似性, 进一步实现整个软件胎记的相似性计算。Cop 方法采取了严格的语义分析, 因而相比其他静态胎记而言对各类语义保留的代码混淆技术具备很好的抵抗力。然而符号执行及理论证明引入的巨大开销使 Cop 难以处理大规模的软件, 同时 Cop 也受制于目前理论证明在处理不透明谓词、未解决猜想等方面的局限性; 此外 Cop 作为静态胎记的一种, 其受制于静态分析难以处理间接分支等的局限性。

5.2 动态胎记技术

5.2.1 轨迹元素法

动态胎记是通过分析软件的执行过程提取出来的胎记, 相比静态胎记其能更好地刻画程序的语义和行为特征, 目前研究也普遍认为其相比静态胎记能更有效地应对各类代码混淆攻击。

Myles 等[141]首次提出动态胎记的概念, 并通过分析程序的动态控制流路径实现了一种胎记 WPP(Whole Program Path), 不过其很容易被循环展开、函数内联等混淆破坏掉。Tamda 等[142, 143]提出了两种适用于 Windows 程序的动态胎记 EXESEQ 和 EXEFREQ, 它们利用 API 序列及其频率分布特性来刻画软件行为, 并分别利用最长子序列和 Cosine 距离计算胎记相似性。Schuler[13]则利用 Java 标准 API 的动态调用序列生成胎记, 以检测 Java 软件的抄袭。基于 API 的胎记只适用于特定的编程语言, Wang 等[16]提出基于系统调用的两种胎记 SCSSB 和 IDSCSB 解决这个问题。

除此之外, DIB[144]及 ISB[145]使用构成执行轨迹的指令生成胎记, 然而整个指令序列过于庞大, 难以分析大规模软件, 而且采用所有指令会包含很多噪声, 难以抵抗代码混淆。对此, Tian 等[20, 146]结合动态污点分析技术识别与程序语义行为紧密相关的关键指令序列, 利用 k-gram 算法生成 DYKIS 胎记并基于 Cosine 距离衡量相似性。Jhi[9]则利用动态数据流分析识别程序执行过程中难以更改且与程序语义紧密相关的核心值, 提出核心值胎记 CVB 并基于 LCS 计算相似性。

5.2.2 依赖分析法

单纯利用软件执行轨迹所体现的基本元素生成胎记, 会抛弃了较多的语法语义信息, 难以对抗复杂的混淆手段; 为此不少研究通过挖掘执行轨迹中基本元素间的依赖关系构造胎记, 以进一步提高对代码混淆的抵抗力。

Wang 等[11]挖掘系统调用间的数据依赖和控制依赖关系, 提出系统调用依赖图胎记 SCDGB; Patrick[27, 147]通过对 JavaScript 程序的动态堆数据进行分析, 构建基于堆图的动态胎记 HGB 刻画程序行为。SCDGB 和 HGB 均采用子图同构实现相似性计算, 然而子图同构是 NP 问题, 在实际应用时面临计算复杂度过高的局限性。

为此对于图形式的胎记, 有研究将图转换成序列或向量形式进行运算。DAAV[148, 149]构建程序的动态调用图(包含系统调用及自身函数)DACG, 利用随机游走为图中每个 API 计算一个权威值, 进而将 DACG 转换成向量形式, 利用 Cosine 距离衡量胎记相似性。SUPB[150]基于程序执行轨迹构建动态调用序列图 DCSG, DCSG 的边反映了函数调用关系, 同时记录了函数调用后的栈深度, SUPB 通过将 DCSG 转换成栈深浅变化序列后利用 LCS 衡量相似性。Jhi[22]在核心值胎记 CVB[9]基础上, 通过分析核心值之间的数据依赖关系, 提出核心值依赖图胎记 VDGB; VDGB 搜索 VDG 中存在偏序依赖的代表性路径, 通过衡量原告代表性路径在被告中所占的比例实现相似性计算, VDGB 相比 CVB 能更有效应对指令重排混淆。

5.2.3 语义分析法

相比已有动态胎记通过分析软件运行时信息和状态来尽可能贴切地描述程序语义行为, LoPD[155, 156]采用严格的形式化逻辑分析捕捉程序语义行为; 此外, 不同于已有胎记技术通过衡量原告与被告相似性判定抄袭, LoPD 通过查找二者可能存在的任何不相似来检测抄袭。LoPD 利用最弱前置条件推理将程序执行路径转换成逻辑公式, 并利用约束求解判断原告及被告执行路径的等价性; 符号执行被用来产生新的输入和驱动新路径的执行, 路径背离检测技术被用来加速抄袭的检测过程。LoPD 由于采用严格的语义等价性证明判定抄袭, 能有效抵抗各种语义保留的代码混淆; 然而其要求存在抄袭的软件具备完全一致的语义, 这极大地限制了 LoPD 的应用范围, 因为语义上丝毫的改变就会使 LoPD 判定为不存在抄袭。此外, LoPD 受制于符号执行和约束求解的局

限性及其带来的巨大开销, 难以处理大规模软件。

5.3 基于软件胎记的抄袭检测技术分析比较

一种软件胎记技术的优劣是通过以下两方面进行衡量: 一是识别抄袭的能力, 也就是胎记抵抗各类语义保留的代码混淆的能力, 要保证混淆前后通过胎记依然可以识别出原程序及混淆版本的同源性, 称之为弹性或抗毁性; 二是区分独立开发软件的能力, 要保证其能有效地将功能相似但独立实现的软件区分开来, 称之为可靠性。

表 3 对各类胎记技术进行总结和比较。从表中可以看出, 现有胎记技术普遍具备较高的可靠性, 而在弹性及性能开销上存在较大差异。静态胎记普遍弹性较低, 但具备开销小的优点; 动态胎记弹性普遍较高, 但面临开销较大的问题。特别是基于语义分析的软件胎记具备非常高的弹性, 但其带来的巨额开销给其实际应用带来很大局限性。基于软件胎记的抄袭检测研究仍处于起始阶段, 有很多挑战性问题需要进一步解决, 也是未来研究的重点。此外, 尽管性能问题一直不是胎记技术关注的重点, 但如何在提高弹性的同时尽可能降低性能开销, 是将来设计并开发实用化工具或系统应当遵守的原则。

表 3 基于软件胎记的抄袭检测技术对比

胎记名称	胎记形式	动/静态	胎记生成方法	弹性	可靠性	性能开销	可用的工具及系统
CVFV/SMC/IS/UC[128, 129]	序列	静态	属性分析法	低	高	小, 胎记抽取及比较相较其他方法开销均小	Sandmark[29] Stigmata[151]
SKB[130]	集合						
W-SKB[131]	集合						
AVSB[132]	序列						
WSCB[133]	集合						
FPB[135]	集合						
CFEB[136]	集合						
SFB[137]	集合		静态控制流法	中	高	中, 控制流引导的胎记构建过程复杂度高, 相似性比较开销一般	
WSPB[138]	集合						
OTB[139]	集合						
SMPB[140]	集合						
Cop[12]	序列		静态语义分析法	高	高	大; 基于理论证明的语义等价证明开销巨大	
WPP[141]	图						
EXESEQ/EXEFREQ[142, 143]	序列	动态	轨迹元素法	中高	高	大; 执行轨迹获取开销大, 具体开销依赖于监控的粒度	JBirth[152] DBPD[153] TAB-PD[154]
JBirth[13]	集合						
SCSSB[16]	集合						
DIB[144]	集合						
ISB[145]	集合						
DYKIS[20]	集合						
CVB[9]	序列						
SCDGB[11]	图		依赖分析法	中高	高	大; 依赖关系分析开销大, 胎记构建过程复杂, 相似性计算开销大	
HGB[27]	图						
DAAV[148]	图						
SUPB[150]	图						
VDGB[22]	图						
LoPD[155]	序列		语义分析法	高	高	大; 受制于符号执行和约束求解的巨大开销	

6 抄袭检测面临的挑战及未来研究方向

前文对当下主流的软件抄袭检测研究进行了总结, 可以看出尽管软件抄袭检测的研究工作取得了很大的进展, 但现有方法在抗混淆能力、性能等方面依然存在很大的局限性, 不少技术还处于理论研究阶段, 难以在实际中得到推广应用; 此外新平台(如移动平台、云平台)的出现、软件发展趋势的变化(多

线程编程)等, 也给抄袭检测带来新的问题和挑战。本节对这些问题进行梳理, 并对未来的研究方向进行展望。

6.1 新平台下的抄袭检测研究

随着移动应用市场的繁荣, 移动应用的抄袭问题日益严峻, 特别是 Android 平台, 其应用程序数量众多, 难以一一人工核查; 此外 App 存在易被反编译的问题, 加之各种成熟的混淆工具的存在, 抄袭者可以很容易获得其代码, 通过重打包等方式修改持有者名字或植入广告再发布的方式达到盈利目的。研究者对多个第三方 App 市场的研究表明, 5%-13%的 App 是重打包发布而来的[7]; 甚至 Google Play 自身中有 1.2%的 App 是重打包的[8]。此外, 通过重打包植入恶意 payloads 可以快速生成并传播大量恶意软件[157]。最近一项研究[8]显示, 来自 33 个 Android 市场共计 120 万个 App 中有 10.93%是可疑的, 即使 Google Play 中也有 7.61%的应用是恶意的, 而且其中有些应用已被安装 100 万次以上; 另一项研究[7]表明, Android 恶意软件中大约有 86%是通过对合法软件进行重打包而来的。因而, 移动应用的抄袭检测研究也获得越来越多的关注, 主流的技术还是通过结合移动应用的自身特点, 设计适用于移动端 App 的软件水印或胎记技术。

目前针对移动 App 的水印技术研究非常少, AppInk[158]简单地将传统的动态图水印技术用于移动端 App 中, 其采用排列图实现水印的编码和植入。Ren 等[14]设计了一种针对移动平台的水印技术 Droidmarking, 其利用自加密代码(Self-Decrypting Code, SDC)段植入水印; SDC 可以使得水印跟原代码间存在紧耦合关系, 简单的解耦会破坏程序完整性而导致无法运行。这样的好处在于不必再关注水印的隐蔽性, 同时 SDC 植入的水印提取开销不大, 使得 Droidmarking 可以部署到应用市场或用户终端上进行实时在线的检测。

移动端 App 具有数量庞大, 更新速度快等特点, 类似于 AppInk 将传统的水印技术简单移植到 App 上的方法显然并不能满足移动平台对其在抗毁性、泛化能力以及性能等方面的需求; Droidmarking 结合移动平台的特点作出了初步的尝试, 但其目前要求 Dalvik 虚拟机运行在非优化模式以保证 SDC 段的格式不会发生改变, 此外其应对动态攻击的能力一般, 并且水印植入给原程序带来的开销也需要进一步优化。因此, 综合考虑移动平台的特点, 通过对传统的动静态水印技术进行扩展, 或者设计专门的水印技术, 来满足移动平台的需求将会是一个非常重要的研究方向。

对于移动平台, 现有方法大部分是利用胎记技术实现 App 的抄袭检测[7, 10, 17, 19, 159-168], 主要是通过分析 App 的词法、语法及结构特征来构造胎记。DroidMoss[7]通过反编译获取 App 的字节码, 利用模糊哈希技术将字节码序列切分成不等长片段并计算 Hash 值, 将所有片段的 Hash 值作为软件胎记, 最后通过计算胎记间的编辑距离衡量相似性。同样地利用字节码序列, Ko 等[159]将整个序列切分成一系列 k-gram 作为胎记, Juxapp[160]则以基本块为单位利用 k-gram 及特征哈希构造胎记, Juxapp 还借助 Hadoop 框架提高检测效率。WuKong[19]提出了两阶段的抄袭检测应对时间复杂度高的问题, 其首先构造粗粒度的 API 胎记筛选出相似性较高的可疑程序, 然后再进行细粒度的指令级分析。不难理解, 这些基于词法分析的胎记很容易被简单的代码混淆如垃圾指令植入、代码重排破坏掉。

Potharaju[161]构造方法的抽象语法树, 将树中所有的水平及纵向路径作为胎记; 该方法考虑更多的语法及结构特征, 但依然难以抵御复杂的混淆手段。通过挖掘控制及数据依赖关系, 也出现一些基于图的软件胎记[10, 17, 162-165]。DroidSim[162]通过分析 API 间的控制流依赖关系, 构建基于组件的控制流图胎记 CB-CFG 并通过子图同构实现相似性计算; Chen 等[10]为 CFG 中每个顶点定义一个坐标, 将 CFG 视作几何图形, 提出基于几何特征的胎记, 该方法极大地提高相似性计算效率, 可以应用于大规模的抄袭检测。这两种基于 CFG 的方法可以有效地检测语法相似的程序, 但难以应对诸如方法拆分、内联、控制流混淆等代码变换手段。DNADroid[163]使用基于程序依赖图 PDG 的软件胎记提高抗混淆能力, 并利用子图同构实现相似性计算, 但性能开销导致其难以大规模应用; AnDarwin[164]对其在性能上进行了优化, 将 PDG 转成语义向量并借助 LSH 实现相似性计算。基于 PDG 的胎记方法可以有效地应对控制流混淆、垃圾指令植入等手段, 但采用特定的数据混淆可以躲避检测。

除了采用经典的 CFG 和 PDG 外, 也有研究[17, 165, 169-171]利用 App 用户界面及其交互丰富的特点, 通过分析 UI 接口及其布局特点等设计新的软件胎记。Yang 等[170]提出 Attribute UI Graph(AUIG)胎记并通过子图同构计算相似性。ViewDroid[165]通过分析可操作的用户接口及接口间的跳转关系构建 View Graph, 进行特征抽取提出 FVG(Feature View Graph)胎记; ResDroid[17]除了利用用户接口的跳转关

系外, 还利用接口的布局信息来构建胎记。不同于前面的方法, DVFB[169]通过动态执行 App 实现 View 信息的收集和胎记构建, 其采用 LSH 和最大双边匹配实现相似性衡量, 是目前唯一已知的针对移动应用抄袭检测的动态胎记。基于 UI 接口的胎记不容易被代码混淆破坏掉, 但抄袭者可以通过虚拟视图植入改变 FVG 的结构, 同时这些胎记不适用于视图较少的 App。

可以看出, 尽管目前也出现了许多基于胎记的 App 抄袭检测方法, 但都还处于起始阶段。大部分方法停留在词法及语法分析层面, 对代码混淆的抵抗力不强; 少数方法尽管可以具备较高的抗混淆能力, 但胎记构建及比较所带来的时间及性能开销导致是其难以得到大规模应用。此外, 鉴于移动平台 App 数量非常庞大, 目前几乎所有方法均采用静态分析, 以牺牲抗混淆能力为代价换取可扩展性。然而静态胎记存在先天的局限性, 比如难以应对诸如加壳、采用动态加载技术等的混淆手段; 动态胎记可以处理这些情况, 然而又面临时空开销大的问题。因而, 如何在保证可扩展性的同时, 结合动静态分析的优势设计抗混淆能力更强的胎记技术, 将会是非常重要同时也非常有趣的研究方向。

此外, 随着云平台的兴起, 有研究[172, 173]开始关注云平台上的软件抄袭问题, Yu 等[173]给出了云端软件抄袭的威胁模型, 并提出了一种适用于云架构的水印技术, 其依然采用传统的水印算法, 但利用 Hadoop 的 MapReduce 编程框架实现并行的水印植入和检测。该研究还比较初步, 但云平台上的软件抄袭检测不失为一个非常有趣的研究点。

6.2 多线程程序的抄袭检测研究

从个人 PC 到智能手机再到服务器, 普遍都采用多核处理器, 为了更充分地利用多核带来的性能提升, 运行的软件也必须是多线程的; 多线程编程变得越来越流行, 也必然是将来软件的发展趋势。前面提到动态软件胎记技术是实现抄袭检测的一种行之有效的方法, 其相比静态胎记而言能更有效地应对代码混淆攻击; 然而动态胎记技术本质上是基于程序执行行为的相似性判定抄袭, 而多线程程序线程交织的不确定性会使得执行行为发生很大的改变, 比如对于一个具有 n 个线程, 每个线程执行 k 步的程序, 其线程交织的可能性高达 $(nk)!/(k!)^n > (n!)^k$ 种, 这必然会大大影响传统动态胎记的检测效果。Tian 等[174]的研究工作也证实即使对于同一个多线程程序, 用相同输入执行多次, 采用传统的动态胎记技术如 SCSSB、DYKIS 等为每次执行抽取胎记并进行比较, 会将同一个程序判定为不存在抄袭。

为此, Tian 等[174]提出了线程感知胎记(Thread-Aware Birthmark)的概念, 并提出 TAB 框架减小线程交织对传统动态胎记的影响。他们的基本假设是尽管线程交织可以非常复杂, 但每个线程内部事件(指令、API 调用等等)发生的次序是相对固定有序的。图 6 简要描述了 TAB 框架: 给定一条执行轨迹, TAB 首先将该轨迹中的每一个事件按照其所在线程进行投影, 形成一系列的线程切片(一个切片就是一条子序列); 然后为每个线程切片单独构建切片胎记, 再将所有的切片胎记进行融合以构建软件胎记。对于切片胎记的构建, 可以沿用其他经典的动态胎记算法; 对于切片胎记的融合, TAB 提供了切片聚合(Slice Aggregation, SA)和切片集(Slice Set, SS)模型生成线程感知胎记。

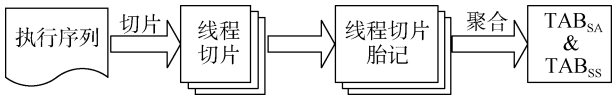


图 6 TAB 框架

这样的好处在于, TAB 作为一个框架, 可以很容易地将传统动态胎记扩展成线程感知胎记。Tian 等利用 TAB 框架对 SCSSB、DYKIS 及 JBirth 进行了扩展, 实验表明扩展后的胎记在处理多线程程序时相比原来检测效果得到大幅度提升。随后, Tian 等又提出 TreSB[175]和 TreCxtB[176], 其利用线程相关的系统调用构造胎记; 其基本假设是线程相关的系统调用是实施及影响程序交织行为而非被影响的元素, 因而利用这些系统调用构建的胎记也更不易被交织影响, 实验表明 TreSB 及 TreCxtB 的检测效果均优于 SCSSB 及 TAB 扩展后的 SCSSB。

然而目前的方法还比较初步。TAB 的假设不完全正确, 线程间的交互和相互影响可能会导致每个线程内发生的事件产生变化; 此外, 进行线程切片尽管可以减小交织带来的影响, 但同时使得胎记难以捕捉程序的整体行为, 特别是对于线程间交互特别复杂的程序将会产生很多漏报。TreSB 及 TreCxtB 对当下主流的代码混淆具备很强的抵抗力, 但不难通过诸如随机植入 lock/unlock 或增加单独线程引入额外线程相关系统调用等方式挫败 TreSB 和 TreCxtB。目前没有专门针对多线程程序设计的混淆手段, 然而一旦类似 TAB、TreSB 以及 TreCxtB 的线程感知胎记得到关注, 必然会出现针对性的混淆手段。多线程编程必然是未来软件开发的主流, 多线程程序的抄袭检测也必然是未来非常重要的一个研究方向。

6.3 部分抄袭问题

目前绝大多数的抄袭检测研究都是针对整体抄袭问题, 也就是抄袭者花费很少的精力通过直接拷贝、自动化混淆或较少人工修改将他人软件拥为己有的情况。整体抄袭很常见也是非常重要的问题, 比如学生作业的抄袭、Android 重打包等均属于整体抄袭的范畴。

然而, 另一种普遍存在的情况是抄袭者仅仅将其他软件的一部分如某个模块或库代码集成到自己软件中, 也就是部分抄袭问题。对于部分抄袭问题, 最先想到的会是采取静态分析来实现, 因为静态分析可以随意地提取可疑的程序部分展开分析; 水印技术显然并不适用, 现有源码抄袭检测技术以及静态胎记技术中有不少针对函数层面的分析方法, 可适用于程序局部模块的分析, 通过适当调整应当可以很容易用于部分抄袭检测。不过正如前文提到的静态分析存在不少局限性, 如不抗混淆、难以处理加壳程序等。

那么另一种选择是采用动态胎记技术, 然而应用动态分析的一个难点在于很难单独抽取出原告以及被告可疑的部分, 因为动态分析需要运行程序, 而很多情况下可疑模块无法有效地从程序中简单剥离出来单独运行。SCDG[11]及 HGB[27]这两个动态胎记技术做了初步尝试, 尽管可以实现部分抄袭检测, 但需要较大的人为参与, 比如需要人为地事先从原告代码中抽取出可疑部分并构造成可执行程序, 或者通过标定来人为地选择监控程序的特定部分。因而, 实现部分抄袭检测, 特别是抗混淆、自动化的部分抄袭检测依然是一个非常具有挑战性的问题。

6.4 抄袭定位及证据生成研究

现有抄袭检测研究普遍忽视的另一个重要问题是抄袭的定位问题, 相比于简单地输出原告及被告的相似值或事先植入的水印信息, 终端用户更加关心的是原告及被告中哪些部分存在抄袭以及存在抄袭部分之间的的对应关系。源码抄袭检测中基于 token 方法的 JPlag[68]和 Moss[39]除了给出原告及被告的相似性外, 还可以将相似的对应代码块的匹配关系展示给用户; 然而基于 token 的方法难以有效对抗混淆攻击, 因而如何对其进行有效地借鉴, 使其他抄袭检测方法特别是高度抗混淆的动态胎记技术也能实现原告及被告匹配模块的对照将会是非常有趣也很重要的研究内容。

一个更具挑战性的问题是抄袭证据的生成。抄袭检测技术原本设计的目的就是搜集并提供尽可能多的证据, 从而在进行法律诉讼时提供充分的技术支持。然而将水印或胎记作为抄袭证据粒度过粗, 即使像 JPlag 和 Moss 给出相似代码块的匹配关系, 也不足以到达作为证据的要求。Cosma 等[23]提出的 PGQT 方法通过衡量匹配的代码块对最终判定抄袭的关键程度, 评估其作为证据的贡献度, 这是证据生成的一个很好思路, 起到了抛砖引玉的作用。此外, 在定位给出匹配的代码块基础上, 对高贡献度的代码块如能证明其语义等价性, 甚至通过分析能够挖掘出原告代码块到被告代码块的变换手段, 实现变换过程的重放, 将会是非常强有力的证据, 当然这将会是一项非常有挑战性的研究, 但也是抄袭检测技术得到实际应用必须要面临和解决的问题。

7 结束语

软件抄袭作为一种非法拷贝并使用他人代码的行为, 已然对软件生态环境的健康发展构成严重威胁, 其得到越来越多的研究人员、教育者、开源社区以及软件企业等的学术界和工业界的普遍关注, 也必然会得到包括知识产权局等在内的政府监管部门的重视。目前, 已提出了大量的软件抄袭检测技术, 并出现了一些相对成熟的检测工具和系统。本文对该领域的研究成果进行回顾: 首先介绍了软件抄袭检测的一些关键问题; 然后依据应用场景和具体技术的不同, 将主流的抄袭检测分为三类分别进行总结, 详细介绍了针对源码的抄袭检测技术, 以及无源码场景下基于软件水印和软件胎记的抄袭检测技术, 并对各类抄袭检测技术进行了系统的比较和分析。最后, 梳理了随着新平台的出现、软件发展趋势的变化等, 抄袭检测面临的新问题和挑战, 探讨了可能的应对之策, 并对未来的研究方向进行了展望。

软件抄袭检测是一个比较复杂的问题, 其涉及教育学、法律学和技术等多个方面。检测技术仅仅是从技术的角度辅助决策者作出决策, 比如通过预筛选存在潜在抄袭的学生作业, 减轻教育人员对上机作业的人工审查压力, 提高教学效果; 识别存在潜在抄袭的移动 App, 减轻移动平台审查员的审查工作, 提升移动平台 App 的质量和安全性; 从技术角度尽可能地搜集证据, 为抄袭案件的法律诉讼提供技术支持等。总体而言, 目前的抄袭检测研究还比较初步, 在应对复杂多样的抄袭手段及实际应用方面还存在很大的局限性, 同时软件发展的新趋势如移动及云平台软件开发、多线程编程等也给抄袭检测研究带来新的问题和挑战。因此, 迫切需要学术界、企业界乃至社会监管部门一起对抄袭问题进行

深入的理解和探讨, 这是帮助软件抄袭检测技术得到更好地发展与成功应用的关键。

参考文献

[1] D. Chuda, P. Navrat, B. Kovacova, and P. Humay, "The Issue of (Software) Plagiarism: A Student View, " *IEEE Transactions on Education*, vol. 55, no. 1, pp. 22-28, 2012.

[2] F. Rosales, A. Garcia, S. Rodriguez, J.L. Pedraza, R. Mendez, and M.M. Nieto, "Detection of Plagiarism in Programming Assignments, " *IEEE Transactions on Education*, vol. 51, no. 2, pp. 174-183, 2008.

[3] T. Lancaster and F. Culwin, "A Comparison of Source Code Plagiarism Detection Engines, " Computer Science Education, vol. 14, no. 2, pp. 101-112, 2004.

[4] "Lawsuits On Open Source, " http://sourceauditor.com/blog/open-source- compliance-trend/.

[5] "Skype-Jotlid Dispute, " http://www.martinsuter.net/blog/2009/08/skype- joltidlicensing-dispute-epic-ma-screwup.html.

[6] L.N. Luo, F. Yu, D.H. Wu, S.C. Zhu, and P. Liu, "Repackage-Proofing Android Apps, " in *The IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'16)*.

[7] W. Zhou, Y. J. Zhou, X.X. Jiang, and P. Ning, "Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces, " in *Proceedings of the second ACM conference on Data and Application Security and Privacy (CODASPY '12)*, pp. 317-326, 2012.

[8] K. Chen, P. Wang, Y. Lee, X.F. Wang, N. Zhang, H.Q. Huang, W. Zou, and P. Liu, "Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale, " in *USENIX Security Symposium (USENIX Security'15)*, pp. 659-674, 2015.

[9] Y.-C. Jhi, X.R. Wang, X.Q. Jia, S.C. Zhu, P. Liu, and D.H. Wu, "Value-Based Program Characterization and its Application to Software Plagiarism Detection, " in *Proc. Int. Conf. Softw. Eng. (ICSE'11)*, pp. 756-765, 2011.

[10] K. Chen, P. Liu, and Y. Zhang, "Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones On Android Markets, " in *Proc. Int. Conf. Sof. Eng.(ICSE'14)*, pp. 175-186, 2014.

[11] X. R. Wang, Y.-C. Jhi, S.C. Zhu, and P. Liu, "Behavior Based Software Theft Detection, " in *Proc. ACM Conf. Computer and Communications Security (CCS'09)*, pp. 280-290, 2009.

[12] L.N. Luo, J. Ming, D.H. Wu, P. Liu, and S.C. Zhu, "Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection, " in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng. (FSE'14)*, pp. 389-400, 2014.

[13] D. Schuler, V. Dallmeier, and C. Lindig, "A Dynamic Birthmark for Java, " in *Proc. IEEE/ACM Int. Conf. Automated Software Engineering (ASE'07)*, pp. 274-283, 2007.

[14] C.G. Ren, K. Chen, and P. Liu, "Droidmarking: Resilient Software Watermarking for Impeding Android Application Repackaging, " in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*, pp. 635-646, 2014.

[15] C. Collberg and C. Thomborson, "Software Watermarking: Models and Dynamic Embeddings, " in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pp. 311-324, 1999.

[16] X.R. Wang, Y.-C. Jhi, S.C. Zhu, and P. Liu, "Detecting Software Theft Via System Call Based Birthmarks, " in *Annual Computer Security Applications Conference (ACSAC'09)*, pp. 149-158, 2009.

[17] Y.R. Shao, X.P. Luo, C.X. Qian, P.F. Zhu, and L. Zhang, "Towards a Scalable Resource-Driven Approach for Detecting Repackaged Android Applications, " in *Annual Computer Security Applications Conference (ACSAC'14)*, pp. 56-65, 2014.

[18] F.F. Zhang, Y.-C. Jhi, D.H. Wu, P. Liu, and S.C. Zhu, "A First Step Towards Algorithm Plagiarism Detection, " in *Proc. Int. Symp. Software Testing and Analysis (ISSTA'12)*, pp. 111-121, 2012.

[19] H.Y. Wang, Y. Guo, Z. Ma, and X.Q. Chen, "WuKong: A Scalable and Accurate Two-Phase Approach to Android App Clone Detection, " in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)*, pp. 71-82, 2015.

[20] Z.Z. Tian, Q.H. Zheng, T. Liu, M. Fan, E.Y. Zhuang, and Z.J. Yang, "Software Plagiarism Detection with Birthmarks Based On Dynamic Key Instruction Sequences, " *IEEE Trans. Software Engineering*, vol. PP, no. 99, pp. 1, 2015.

[21] C.S. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection, " *IEEE Trans. Software Engineering*, vol. 28, no. 8, pp. 735-746, 2002.

[22] Y.-C. Jhi, X.Q. Jia, X.R. Wang, S.C. Zhu, P. Liu, and D.H. Wu, "Program Characterization Using Runtime Values and its Application to Software Plagiarism Detection, " I*EEE Trans. Software Engineering*, vol. 41, no. 9, pp. 925-943, 2015.

[23] G. Cosma and M. Joy, "An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis, " *IEEE Trans. Computers*, vol. 61, pp. 379-394, 2012.

[24] R.A. Jarvis and E.A. Patrick, "Clustering Using a Similarity Measure Based On Shared Near Neighbors, " *IEEE Trans. Computers*, vol. 100, no. 11, pp. 1025-1034, 1973.

[25] C. Liu, C. Chen, J.W. Han, and P.S. Yu, "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis, " in *Proc. ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD'06)*, pp. 872-881, 2006.

[26] S. Chaki, C. Cohen, and A. Gurfinkel, "Supervised Learning for

Provenance-Similarity of Binaries, " in *Proc. ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD '11)*, pp. 15-23, 2011.

[27] P.P.F. Chan, L.C.K. Hui, and S.M. Yiu, "Heap Graph Based Software Theft Detection, " *IEEE Trans. Information Forensics and Security*, vol. 8, no. 1, pp. 101-110, 2013.

[28] H. Xiong, H.H. Yan, T. Guo, Y.G. Huang, Y.L. Hao, and Z.J. Li, "Code Similarity Detection: A Survey, " *Computer Science*, vol. 37, no. 8, pp. 9-14, 2010.
(李舟军, 熊浩, 晏海华, “代码相似性检测技术: 研究综述”, 计算机科学, 2010, 37(08): 9-14。)

[29] "Sandmark, " http: //sandmark.cs.arizona.edu/.

[30] "Zelix Klassmaster, " http: //www.zelix.com/klassmaster/.

[31] "DashO, " https: //www.preemptive.com/products/dasho.

[32] "DexGuard, " https: //www.guardsquare.com/dexguard.

[33] M. Madou, L. Van Put, and K. De Bosschere, "LOCO: An Interactive Code (De)Obfuscation Tool, " in *Proc. ACM SIGPLAN Symp. Partial Evaluation and Semantics-based Program Manipulation (PEPM'06)*, pp. 140-144, 2006.

[34] "Upx, " http: //upx.sourceforge.net/.

[35] "ProGuard, " http: //proguard.sourceforge.net/.

[36] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations, " Technical Report #148, Department of Computer Science, University of Auckland, 1997.

[37] Z. Wu, S. Gianvecchio, M. Xie, and H. Wang, "Mimimorphism: A New Approach to Binary Code Obfuscation, " in *Proc. ACM Conf. Computer and Communications Security (CCS'10)*, pp. 536-546, 2010.

[38] K.A. Roundy and B.P. Miller, "Binary-Code Obfuscations in Prevalent Packer Tools, " *ACM Computing Surveys*, vol. 46, no. 4, 2013.

[39] "Moss-a System for Detecting Software Plagiarism, " http://theory.stan ford.edu/~aiken/moss/.

[40] "Jplag, " https: //jplag.ipd.kit.edu/.

[41] "Sherlock, " http://www2.warwick.ac.uk/fac/sci/dcs/research/ias/software/ sherlock/.

[42] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes, "Language-Independent Clone Detection Applied to Plagiarism Detection, " in *IEEE Working Conference on Source Code Analysis and Manipulation (SCAM'10)*, pp. 77-86, 2010.

[43] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, " *IEEE Trans. Software Engineering*, vol. 28, no. 7, pp. 654-670, 2002.

[44] H.J. Kim, Y.B. Jung, S.H. Kim, and K.K. Yi, "MeCC: Memory Comparison-Based Clone Detector, " in *Proc. Int. Conf. Softw. Eng. (ICSE'11)*, pp. 301-310, 2011.

[45] C.K. Roy and J.R. Cordy, "A Survey On Software Clone Detection Research, " SCHOOL OF COMPUTING TR 2007-541, QUEEN'S UNIVERSITY, vol. 115, 2007.

[46] G. Whale, "Plague: Plagiarism Detection Using Program Structure, " Comput. Sci., Univ. New South, 1988.

[47] E.L. Jones, "Metrics Based Plagiarism Monitoring, " *Journal of Computing Sciences in Colleges*, vol. 16, no. 4, pp. 253-261, 2001.

[48] J.A.W. Faidhi and S.K. Robinson, "An Empirical Approach for Detecting Program Similarity and Plagiarism within a University Programming Environment, " *Computers & Education*, vol. 11, no. 1, 1987.

[49] G. Whale, "Software Metrics and Plagiarism Detection, " *Journal of Systems and Software*, vol. 13, no. 2, pp. 131-138, 1990.

[50] K.J. Ottenstein, "An Algorithmic Approach to the Detection and Prevention of Plagiarism, " *ACM Sigcse Bulletin*, vol. 8, no. 4, pp. 30-41, 1976.

[51] M.H. Halstead, "Elements of Software Science, " Elsevier New York, 1977.

[52] M.H. Halstead, "Natural Laws Controlling Algorithm Structure?" *ACM Sigplan Notices*, vol. 7, no. 2, pp. 19-26, 1972.

[53] A.L.P.H. John L Donaldson, "A Plagiarism Detection System, " *Computer Science Education*, vol. 13, no. 1, pp. 21-25, 1981.

[54] S.L. Grier, "A Tool that Detects Plagiarism in Pascal Programs, " *Computer Science Education*, vol. 13, no. 1, pp. 15-20, 1981.

[55] H. Berghel and D.L. Sallach, "Measurements of Program Similarity in Identical Task Environments, " *Sigplan Notices*, pp. 65-76, 1984.

[56] M.J.W. Kristina L Verco, "Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting Systems, "in *Australasian Conference on Computer Science Education (ACSE'96)*, pp. 81-88, 1996.

[57] S. Engels, V. Lakshmanan, and M. Craig, "Plagiarism Detection Using Feature-Based Neural Networks, " in *SIGCSE Technical Symposium on Computer Science Education (SICCSE'07)*, pp. 34-38, 2007.

[58] L. Moussiades and A. Vakali, "PDetect: A Clustering Approach for Detecting Plagiarism in Source Code Datasets, " *The Computer Journal*, vol. 48, no. 6, pp. 651-661, 2005.

[59] L. Zhang, Y.T. Zhuang, and Z.M. Yuan, "A Program Plagiarism Detection Model Based On Information Distance and Clustering, " in *International Conference on Intelligent Pervasive Computing (IPC'07)*, pp. 431-436, 2007.

[60] E. Merlo, "Detection of Plagiarism in University Projects Using Metrics-Based Spectral Similarity, " in *Dagstuhl Seminar Proceedings (DSP'07)*, 2007.

[61] V. Ciesielski, N. Wu, and S. Tahaghoghi, "Evolving Similarity Functions for Code Plagiarism Detection, " in *Proceedings of the*

10th Annual Conference on Genetic and Evolutionary Computation (GECCO'08), pp. 1453-1460, 2008.

[62] M. Joy and M. Luck, "Plagiarism in Programming Assignments, " *IEEE Transactions on Education*, vol. 42, no. 2, pp. 129-133, 1999.

[63] G. Whale, "Identification of Program Similarity in Large Populations, " *The Computer Journal*, vol. 33, no. 2, pp. 140-146, 1990.

[64] M.J. Wise, "Detection of Similarities in Student Programs: YAP'ing May be Preferable to Plague'ing, " in *SIGCSE Technical Symposium on Computer Science Education (SIGCSE'92)*, pp. 268-271, 1992.

[65] M.J. Wise, "YAP3: Improved Detection of Similarities in Computer Program and Other Texts, " in *SIGCSE Technical Symposium on Computer Science Education (SIGCSE'96)*, pp. 130-134, 1996.

[66] R.M. Karp and M.O. Rabin, "Efficient Randomized Pattern-Matching Algorithms, " *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249-260, 1987.

[67] D. Gitchell and N. Tran, "SIM: A Utility for Detecting Similarity in Computer Programs, " in *SIGCSE technical symposium on Computer science education (SIGCSE'99)*, pp. 266-270, 1999.

[68] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding Plagiarisms Among a Set of Programs with Jplag, " *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016-1038, 2002.

[69] C.H. Zhao, H.H. Yan, and M.Z. Jin, "Approach based on Compiling Optimization and Disassembling to Detect Program Similarity, " *Journal of Beijing University of Aeronautics and Astronautics*, vol. 32, no. 6, pp. 711-715, 2008.
(金茂忠, 赵长海, 晏海华, “基于编译优化和反汇编的程序相似性检测方法”, 北京航空航天大学学报, 2008, 32(06): 711-715。)

[70] C. Arwin and S.M. Tahaghoghi, "Plagiarism Detection Across Programming Languages, " in *Proceedings of the 29th Australasian Computer Science Conference (ACSC'06)*, pp. 277-286, 2006.

[71] V. Juricic, "Detecting Source Code Similarity Using Low-Level Languages, " in *International Conference on Information Technology Interfaces (ITI'11)*, pp. 597-602, 2011.

[72] X. Chen, B. Francia, and M. Li, "Shared Information and Program Plagiarism Detection, " *IEEE Transactions on Information Theory*, vol. 50, no. 7, pp. 1545-1551, 2004.

[73] K.L. Pandey, S. Agarwal, S. Misra, and R. Prasad, "Plagiarism Detection in Software Using Efficient String Matching, " in *International Conference on Computational Science and Its Applications (ICCSA'12)*, pp. 147-156, 2012.

[74] M. Li and P. Vitanyi, "An Introduction to Kolmogorov Complexity and its Applications, " *Springer-Verlag New York*, 1997.

[75] J.W. Son, T.G. Noh, H.J. Song, and S.B. Park, "An Application for Plagiarized Source Code Detection Based On a Parse Tree Kernel, " *Engineering Applications of Artificial Intelligence*, vol. 26, no. 8, pp. 1911-1918, 2013.

[76] J.W. Son, S.B. Park, and S.Y. Park, "Program Plagiarism Detection Using Parse Tree Kernels, " in *Pacific Rim international conference on Artificial intelligence (PRICAI'06)*, pp. 1000-1004, 2006.

[77] C.L. Liu, S.Y. Jia, L.P. Zhang, and D.S. Liu, "AST-Based Plagiarism Detection Method, " *Computer Engineering and Design*, vol. 33, no. 4, pp. 1660-1664, 2012.

[78] H. Kikuchi, T. Goto, M. Wakatsuki, and T. Nishino, "A Source Code Plagiarism Detecting Method Using Alignment with Abstract Syntax Tree Elements, " in *IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'14)*, pp. 1-6, 2014.

[79] T. Guo, G.W. Dong, H. Qin, and B.J. Cui, "Improved Plagiarism Detection Algorithm Based On Abstract Syntax Tree, " in *International Conference on Emerging Intelligent Data and Web Technologies (EIDWT'13)*, pp. 714-719, 2013.

[80] H. Xiong, H.H. Yan, Z.J. Li, and H. Li, "BuAA_AntiPlagiarism: A System to Detect Plagiarism for C Source Code, " in *International Conference on Computational Intelligence and Software Engineering (CiSE'09)*, pp. 1-5, 2009.

[81] N.G. Resmi and K.P. Soman, "Abstract Syntax Tree Generation Using Modified Grammar for Source Code Plagiarism Detection, " *International Journal of Computing and Technology*, vol. 1, no. 6, 2014.

[82] J.H. Ji, G. Woo, and H.G. Cho, "A Source Code Linearization Technique for Detecting Plagiarized Programs, " in *Proceedings of the Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'07)*, pp. 73-77, 2007.

[83] S.Y. Noh, "An Xml Plagiarism Detection Model for Procedural Programming Languages, " Iowa State University: Technical Report TR03-14, 2003.

[84] S.Y. Noh, S.W. Kim, and C.Y. Jung, "A Lightweight Program Similarity Detection Model Using Xml and Levenshtein Distance, " in *International Conference on Frontiers in Education: Computer Science & Computer Engineering (ICFE-CSCE'06)*, pp. 3-9, 2006.

[85] J. Krinke, "Identifying Similar Code with Program Dependence Graphs, " in *Working Conference on Reverse Engineering (WCRE'01)*, pp. 301-309, 2001.

[86] J. Nagra, C. Thomborson, and C. Collberg, "A Functional Taxonomy for Software Watermarking, " in Proceedings of the Australasian Conference on Computer Science (ACSC'02), pp. 177-186, 2002.

[87] L.H. Zhang, Y.X. Yang, X.X. Niu, and S.Z. Niu, "A Survey on Software Watermarking, " *Journal of Software*, vol. 14, no. 2, pp. 268-277, 2003.
(张立和, 杨义先, 纽心忻, 牛少彰“软件水印综述”, *软件学报*,

2003, 14(02): 268-277。)

[88] W. Zhu, C. Thomborson, and F.Y. Wang, "A Survey of Software Watermarking, " in *IEEE International Conference on Intelligence and Security Informatics (ISI'05)*, 2005.

[89] J. Hamilton and S. Danicic, "A Survey of Static Software Watermarking, " in *World Congress on Internet Security (WorldCIS'11)*, pp. 100-107, 2011.

[90] P.R. Samson, "Apparatus and Method for Serializing and Validating Copies of Computer Software, " no. US 07/938, 278 1994.

[91] R.I. Davidson and N. Myhrvold, "Method and System for Generating and Auditing a Signature for a Computer Program, " no. US 08/268, 967 1996.

[92] A. Monden, H. Iida, K.I. Matsumoto, K. Inoue, and K. Torii, "A Practical Method for Watermarking Java Programs, " in *the IEEE Computer Society International Conference on Computers, Software & Applications (COMPSAC'00)*, pp. 191-197, 2000.

[93] K. Fukushima and K. Sakurai, "A Software Fingerprinting Scheme for Java Using Classfiles Obfuscation, " *Springer*, 2003.

[94] S. Thaker, "Software Watermarking Via Assembly Code Transformations [Ph.D.dissertation], " San Jose State University, 2004.

[95] C. Collberg and T.R. Sahoo, "Software Watermarking in the Frequency Domain: Implementation, Analysis, and Attacks, " *Journal of Computer Security*, vol. 13, no. 5, pp. 721-755, 2005.

[96] G. Myles, C. Collberg, Z. Heidepriem, and A. Navabi, "The Evaluation of Two Software Watermarking Algorithms, " *Software: Practice and Experience*, vol. 35, no. 10, pp. 923-938, 2005.

[97] B. Anckaert, B. De Sutter, and K. De Bosschere, "Covert Communication through Executables, " pp. 83-85.

[98] M. Shirali-Shahreza and S. Shirali-Shahreza, "Software Watermarking by Equation Reordering, " in *International Conference on Information and Communication Technologies: From Theory to Applications (ICTTA'08)*, pp. 1-4, 2008.

[99] Z.L. Sha, H. Jiang, and A.C. Xuan, "Software Watermarking Algorithm by Coefficients of Equation, " in *International Conference on Genetic and Evolutionary Computing (WGEC'09)*, pp. 410-413, 2009.

[100] D.F. Gong, F.L. Liu, B. Lu, P. Wang, and L. Ding, "Hiding Information in Java Class File, " in *Computer Science and Computational Technology (ISCSCT'08)*, pp. 160-164, 2008.

[101] J.P. Stern, G. Hachez, F. Koeune, and J. Quisquater, "Robust Object Watermarking: Application to Code, " in *Information Hiding (IH'99)*, pp. 368-378, 1999.

[102] F. Koushanfar, G. Qu, and M. Potkonjak, "Intellectual Property Metering, " in *Information Hiding (IH'01)*, 2001.

[103] G. Qu and M. Potkonjak, "Hiding Signatures in Graph Coloring Solutions, " in *Information Hiding (IH'99)*, pp. 348-367, 1999.

[104] G. Myles and C. Collberg, "Software Watermarking through Register Allocation: Implementation, Analysis, and Attacks, " in *Conference: Information Security and Cryptology (ICISC '03)*, pp. 274-293, 2003.

[105] W. Zhu and C. Thomborson, "Algorithms to Watermark Software through Register Allocation, " in *Internatinal Conference on Digital Rights Management: Technologies, Issues, Challenges and Systems (DRMTICS'05)*, pp. 180-191, 2005.

[106] H. Lee and K. Kaneko, "New Approaches for Software Watermarking by Register Allocation, " in *ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'08)*, pp. 63-68, 2008.

[107] X.C. Lu and Z.M. Chen, "Software Watermarking Algorithm Based On Register Allocation, " in *International Symposium on Distributed Computing and Applications to Business Engineering and Science (DCABES'10)*, pp. 539-543, 2010.

[108] R. Venkatesan, V. Vazirani, and S. Sinha, "A Graph Theoretic Approach to Software Watermarking, " in *Information Hiding (IH'01)*, pp. 157-168, 2001.

[109] C. Collberg, A. Huntwork, E. Carter, and G. Townsend, "Graph Theoretic Software Watermarks: Implementation, Analysis, and Attacks, " in *Information Hiding (IH'04)*, pp. 192-207, 2004.

[110] C. Collberg, A. Huntwork, E. Carter, G. Townsend, and M. Stepp, "More On Graph Theoretic Software Watermarks: Implementation, Analysis, and Attacks, " *Information and Software Technology,* vol. 51, no. 1, pp. 56-67, 2009.

[111] G. Arboit, "A Method for Watermarking Java Programs Via Opaque Predicates, " in *the Fifth International Conference on Electronic Commerce Research (ICECR'02)*, pp. 102-110, 2002.

[112] G. Myles and C. Collberg, "Software Watermarking Via Opaque Predicates: Implementation, Analysis, and Attacks, " *ELECTRONIC COMMERCE RESEARCH*, vol. 6, no. 2, pp. 155-171, 2006.

[113] P. Cousot and R. Cousot, "An Abstract Interpretation-Based Framework for Software Watermarking, " in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pp. 173-185, 2004.

[114] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q.Y. Shao, and Y. Zhang, "Experience with Software Watermarking, " in *Annual Conference on Computer Security Applications (ACSAC'00)*, pp. 308-316, 2000.

[115] C. Collberg, S. Kobourov, E. Carter, and C. Thomborson, "Error-Correcting Graphs for Software Watermarking, " in *Workshop on Graph Theoretic Concepts in Computer Science (WGTCCS'03)*, pp. 156-167, 2003.

[116] C.S. Collberg, C. Thomborson, and G.M. Townsend, "Dynamic Graph-Based Software Fingerprinting, " *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 6, pp. 35, 2007.

[117] C. Collberg, C. Thomborson, and G.M. Townsend, "Dynamic Graph-Based Software Watermarking, " TR04-08, Department of Computer Science, 2004.

[118] J.Q. Zhu, Y.H. Liu, and K.X. Yin, "A Novel Planar Ippct Tree Structure and Characteristics Analysis, " *Journal of Software*, vol. 5, no. 3, pp. 344-351, 2010.

[119] R. Venkatesan and V. Vazirani, "Technique for Producing through Watermarking Highly Tamper-Resistant Executable Code and Resulting “Watermarked” Code so Formed, " no. US 10/970, 425 2006.

[120] P. Zhou, X. Chen, and X.G. Yang, "The Software Watermarking for Tamper Resistant Radix Dynamic Graph Coding, " *Information Technology Journal*, vol. 9, no. 6, pp. 1236-1240, 2010.

[121] I. Kamel and Q. Albluwi, "A Robust Software Watermarking for Copyright Protection, " *Computers & Security*, vol. 28, no. 6, pp. 395-409, 2009.

[122] C.S. Collberg, E. Carter, S.K. Debray, A. Huntwork, J.D. Kececioglu, C. Linn, and M. Stepp, "Dynamic Path-Based Software Watermarking, " in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI'04)*, pp. 107-118, 2004.

[123] G. Myles and H.X. Jin, "Self-Validating Branch-Based Software Watermarking, " in *Information Hiding (IH'05)*, pp. 342-356, 2005.

[124] J. Nagra and C. Thomborson, "Threading Software Watermarks, " in *Information Hiding (IH'04)*, pp. 208-223, 2004.

[125] M. Dalla Preda, R. Giacobazzi, and E. Visentini, "Hiding Software Watermarks in Loop Structures, " in *International Symposium on Static Analysis (SAS'08)*, pp. 174-188, 2008.

[126] X.S. Zhang, F.L. He, and W.L. Zuo, "Hash Function Based Software Watermarking, " in *Advanced Software Engineering and Its Applications (ASEA'08)*, pp. 95-98, 2008.

[127] H.Y. Ma, K.J. Lu, X.J. Ma, H.N. Zhang, C.F. Jia, and D.B. Gao, "Software Watermarking Using Return-Oriented Programming, " in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIA CCS'15)*, 2015.

[128] H. Tamada, M. Nakamura, and A. Monden, "Design and Evaluation of Birthmarks for Detecting Theft of Java Programs, " in *IASTED Conf. on Software Engineering (IASTEDSE'04)*, pp. 569-574, 2004.

[129] H. Tamada, M. Nakamura, A. Monden, and K.I. Matsumoto, "Java Birthmarks--Detecting the Software Theft--, " *IEICE Transactions on Information and Systems*, vol. 88, no. 9, pp. 2148-2158, 2005.

[130] G. Myles and C. Collberg, "K-Gram Based Software Birthmarks, " in *Proc. ACM Symp. Applied Computing (SAC'05)*, pp. 314-318, 2005.

[131] X. Xie, F.L. Liu, B. Lu, and L. Chen, "A Software Birthmark Based On Weighted K-Gram, " in *IEEE Int. Conf. Intelligent Computing and Intelligent Systems (ICIS'10)*, pp. 400-405, 2010.

[132] D.J. Kim, S.J. Cho, S.C. Han, M. Park, and I. You, "Open Source Software Detection Using Function-Level Static Software Birthmark, " *Journal of Internet Services and Information Security (JISIS)*, vol. 4, no. 4, pp. 25-37, 2014.

[133] S. Choi, H. Park, H.I. Lim, and T. Han, "A Static API Birthmark for Windows Binary Executables, " *Journal of Systems and Software*, vol. 82, no. 5, pp. 862-873, 2009.

[134] S. Choi, H. Park, H. Lim, and T. Han, "A Static Birthmark of Binary Executables Based On API Call Structure, " in *Proceedings of the Asian Computing Science Conference on Advances in Computer Science: Computer and Network Security (ASIAN'07)*, pp. 2-16, 2007.

[135] H. Lim, H. Park, S. Choi, and T. Han, "A Method for Detecting the Theft of Java Programs through Analysis of the Control Flow Information, " *Information and Software Technology*, vol. 51, no. 9, pp. 1338-1350, 2009.

[136] H.I. Lim, H. Park, S. Choi, and T. Han, "A Static Java Birthmark Based On Control Flow Edges, " in *Annual IEEE International Computer Software and Applications Conference, (COMPSAC'09)*, pp. 413-420, 2009.

[137] H.I. Lim and T. Han, "Analyzing Stack Flows to Compare Java Programs, " *IEICE Trans. Information and Systems*, vol. 95-D, no. 2, pp. 565-576, 2012.

[138] H.I. Lim, P. Heewan, and H. Taisook, "Detecting Theft of Java Applications Via a Static Birthmark Based On Weighted Stack Patterns, " *IEICE Transactions on Information and Systems*, vol. 91, no. 9, pp. 2323-2332, 2008.

[139] H. Park, H.I. Lim, S. Choi, and T. Han, "Detecting Common Modules in Java Packages Based On Static Object Trace Birthmark, " *Computer Journal*, vol. 54, no. 1, pp. 108-124, 2011.

[140] S. Park, H. Kim, J. Kim, and H. Han, "Detecting Binary Theft Via Static Major-Path Birthmarks, " in *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems (RACS'14)*, pp. 224-229, 2014.

[141] G. Myles and C. Collberg, "Detecting Software Theft Via Whole Program Path Birthmarks, " in *Proc. Int. Conf. Information Security (ISC'04)*, pp. 404-415, 2004.

[142] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K.I. Matsumoto, "Dynamic Software Birthmarks to Detect the Theft of Windows Applications, " in *Int. Symp. Future Software Technology (ISFST'04)*, 2004.

[143] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. Ichi Matsumoto, "Design and Evaluation of Dynamic Software Birthmarks Based On API Calls, " Nara Institute of Science & Technology, pp. 1751-1763, 2007.

[144] B. Lu, F. Liu, X. Ge, B. Liu, and X. Luo, "A Software Birthmark

Based On Dynamic Opcode N-Gram, " in *International Conference on Semantic Computing (ICSC'07)*, pp. 37-44, 2007.

[145] Y.M. Bai, X.M. Sun, G. Sun, X.H. Deng, and X.M. Zhou, "Dynamic K-Gram Based Software Birthmark, " in *Australian Conference on Software Engineering (ASWEC'08)*, pp. 644-649, 2008.

[146] Z.Z. Tian, Q.H. Zheng, T. Liu, and M. Fan, "DKISB: Dynamic Key Instruction Sequence Birthmark for Software Plagiarism Detection, " in *IEEE Int. Conf. High Performance Computing and Communications (HPCC'13)*, pp. 619-627, 2013.

[147] P.P. Chan, L.C. Hui, and S.M. Yiu, "JSBiRTH: Dynamic Javascript Birthmark Based On the Run-Time Heap, " in *IEEE Annual Computer Software and Applications Conference (COMPSAC'11)*, pp. 407-412, 2011.

[148] D. Chae, S. Kim, S. Cho, and Y. Kim, "DAAV: Dynamic Api Authority Vectors for Detecting Software Theft, " in *ACM International on Conference on Information and Knowledge Management (CIKM'15)*, pp. 1819-1822, 2015.

[149] D.K. Chae, S.W. Kim, S.J. Cho, and Y. Kim, "Effective and Efficient Detection of Software Theft Via Dynamic API Authority Vectors, " *Journal of Systems and Software*, vol. 110, pp. 1-9, 2015.

[150] J. Park, D. Son, D. Kang, J. Choi, and G. Jeon, "Software Similarity Analysis Based On Dynamic Stack Usage Patterns, " in *Conference on Research in Adaptive and Convergent Systems (RACS'15)*, pp. 285-290, 2015.

[151] "Stigmata, " http: //stigmata.osdn.jp/implementation.html.

[152] "JBirth, " https: //www.st.cs.uni-saarland.de/birthmarking/.

[153] "DBPD, " Z.Z. Tian, http://labs.xjtudlc.com/labs/wlaq/dbpd/site.

[154] "TAB-PD, " http://labs.xjtudlc.com/labs/wlaq/TAB-PD/site/index.html.

[155] J. Ming, F.F. Zhang, D.H. Wu, P. Liu, and S.C. Zhu, "Deviation-Based Obfuscation-Resilient Program Equivalence Checking with Application to Software Plagiarism Detection, " *IEEE Trans. Reliability*, 2016.

[156] F.F. Zhang, D.H. Wu, P. Liu, and S.C. Zhu, "Program Logic Based Software Plagiarism Detection, " in *IEEE Int. Symp. Software Reliability Engineering (ISSRE'14)*, pp. 66-77, 2014.

[157] W. Yang, X.S. Xiao, D.F. Li, H.R. Li, H.Y. Wang, Y. Guo and T. Xie, "Security Analytics for Mobile Apps: Achievements and Challenges," *Journal of Cyber Security*, vol. 1, no. 2, pp. 1-14, 2016.
(杨威, 肖旭生, 李邓锋, 李豁然, 刘譞哲, 王浩宇, 郭耀, 谢涛, “移动应用安全解析学: 成果与挑战”, *信息安全学报*, 2016, 1(2): 1-14。)

[158] W. Zhou, X.W. Zhang, and X.X. Jiang, "AppInk: Watermarking Android Apps for Repackaging Deterrence, " in *Proceedings of ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS'13)*, pp. 1-12, 2013.

[159] J. Ko, H.J. Shim, D.J. Kim, Y.S. Jeong, S.J. Cho, M. Park, S. Han, and S.B. Kim, "Measuring Similarity of Android Applications Via Reversing and K-Gram Birthmarking, " in *Proceedings of the 2013 Research in Adaptive and Convergent Systems (RACS'13)*, pp. 336-341, 2013.

[160] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications, " in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'12)*, pp. 62-81, 2012.

[161] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang, "Plagiarizing Smartphone Applications: Attack Strategies and Defense Techniques, " in *International Conference on Engineering Secure Software and Systems (ESSoS'12)*, pp. 106-120, 2012.

[162] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, "Detecting Code Reuse in Android Applications Using Component-Based Control Flow Graph, " in *IFIP Advances in Information and Communication Technology (IFIP AICT'14)*, pp. 142-155, 2014.

[163] J. Crussell, C. Gibler, and H. Chen, "Attack of the Clones: Detecting Cloned Applications On Android Markets, " in *European Symposium on Research in Computer Security (ESORICS'13)*, pp. 37-54, 2012.

[164] J. Crussell, C. Gibler, and H. Chen, "AnDarwin: Scalable Detection of Semantically Similar Android Applications, " in *European Symposium on Research in Computer Security (ESORICS'13)*, pp. 182-199, 2013.

[165] F.F. Zhang, H.Q. Huang, S.C. Zhu, D.H. Wu, and P. Liu, "ViewDroid: Towards Obfuscation-Resilient Mobile Application Repackaging Detection, " in *Proc. ACM Conf. Security and Privacy in Wireless and Mobile Networks (WiSec'14)*, pp. 25-36, 2014.

[166] H.Q. Huang, S.C. Zhu, P. Liu, and D.H. Wu, "A Framework for Evaluating Mobile App Repackaging Detection Algorithms, " in *International Conference on Trust & Trustwor-thy Computing (TRUST'13)*, pp. 169-186, 2013.

[167] Q.L. Guan, H.Q. Huang, W.Q. Luo, and S.C. Zhu, "Semantics-Based Repackaging Detection for Mobile Apps, " Springer International Publishing, 2016.

[168] I. Gurulian, K. Markantonakis, L. Cavalaro, and K. Mayes, "You Can't Touch this: Consumer-Centric Android Application Repackaging Detection, " *Future Generation Computer Systems*, vol. 65, pp. 1-9, 2016.

[169] C. Soh, H.B.K. Tan, Y.L. Arnatovich, and L.P. Wang, "Detecting Clones in Android Applications through Analyzing User Interfaces, "in *International Conference on Program Comprehension (ICPC'15)*, pp. 163-173, 2015.

[170] C.X. Yang, C.S. Zuo, S.Q. Guo, C.Y. Hu, and L.Z. Cui, "UI Ripping in Android: Reverse Engineering of Graphical User Interfaces

and its Application, " in *IEEE Conference on Collaboration and Internet Computing (CCIC'15)*, pp. 160-167, 2015.

[171] M. Sun, M. Li, and J.C.S. Lui, "DroidEagle: Seamless Detection of Visually Similar Android Apps, " in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec'15)*, pp. 1-9, 2015.

[172] C.W. Wang, M.C.Y. Cho, C.W. Wang, and S.W. Shieh, "Combating Software Piracy in Public Clouds, " *Computer*, vol. 48, no. 10, pp. 88-91, 2015.

[173] Z.W. Yu, C.K. Wang, C. Thomborson, J.M. Wang, S.G. Lian, and A.V. Vasilakos, "A Novel Watermarking Method for Software Protection in the Cloud, " *Software: Practice and Experience*, vol. 42, no. 4, pp. 409-430, 2012.

[174] Z.Z. Tian, Q.H. Zheng, T. Liu, M. Fan, X.D. Zhang, and Z.J. Yang, "Plagiarism Detection for Multithreaded Software Based On Thread-Aware Software Birthmarks, " in *Proc. Int. Conf. Program Comprehension (ICPC'14)*, pp. 304-313, 2014.

[175] Z.Z. Tian, T. Liu, Q.H. Zheng, F.F. Tong, M. Fan, and Z.J. Yang, "A New Thread-Aware Birthmark for Plagiarism Detection of Multithreaded Programs, " in *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE'16)*, pp. 734-736, 2016.

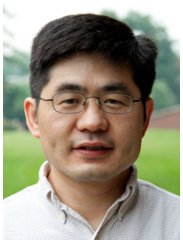
[176] Z.Z. Tian, T. Liu, Q.H. Zheng, M. Fan, E.Y. Zhuang, and Z.J. Yang, "Exploiting Thread-Related System Calls for Plagiarism Detection of Multithreaded Programs, " *Journal of Systems and Software*, 2016.

田振洲 于 2010 年在西安交通大学计算机专业获得工学学士学位。现在西安交通大学计算机专业攻读博士学位。研究领域为可信软件。研究兴趣包括: 软件动态行为分析、软件抄袭检测、恶意软件分析。Email: zztian@stu.xjtu.edu.cn

刘烃 于 2010 年在西安交通大学系统工程专业获得博士学位。现任西安交通大学系统工程研究所副教授。研究领域包括: 网络安全、智能电网、可信软件。研究兴趣包括: 智能电网漏洞挖掘与入侵检测、软件行为建模及分析。Email: tingliu@mail.xjtu.edu.cn

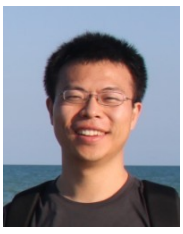
郑庆华 于 1997 年在西安交通大学系统工程专业获得博士学位。现任西安交通大学副校长、计算机系主任。为国家杰出青年基金获得者, 教育部长江学者特聘教授, 国家“新世纪百千万人才工程”人选, 首批“万人计划”科技创新领军人才, 教育部科技创新团队、陕西省重点科技创新团队负责人。研究领域包括: 计算机网络安全、智能 e-Learning 环境的理论与技术、网络舆情及有害信息监控、可信软件。Email: qhzheng@xjtu.edu.cn

佟菲菲 于 2015 年在西安交通大学计算机科学与技术专业获得工学学士学位。现在西安交通大学计算机科学与技术专业攻读硕士学位。研究领域为可信软件。研究兴趣包括: 软件抄袭检测、软件分析。Email: tongfeifei@stu.xjtu.edu.cn



吴定豪 于 2005 年在普林斯顿大学计算机专业获得博士学位, 现任宾夕法尼亚州立大学信息科学与技术学院助理教授, 研究兴趣包括: 软件安全、程序分析与验证、软件工程和程序设计语言。Email: dwu@ist.psu.edu

朱森存 于 2004 年在乔治梅森大学信息技术专业获得博士学位, 现任宾夕法尼亚州立大学副教授。研究领域为安全与隐私, 研究兴趣包括: 传感器网络安全、智能手机及蜂窝网络安全、软件安全、在线隐私和安全。Email: szhu@cse.psu.edu



陈恺 于 2010 年在中国科学院大学获得博士学位, 现在中国科学院信息工程研究所担任研究员, 研究兴趣包括软件分析与测试、智能手机、隐私安全。Email: chenkai@iie.ac.cn