软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(7):2242−2259 [doi: 10.13328/j.cnki.jos.006271]

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

# 面向代码相似性检测的相似哈希改进方法*

李 玫[1,2,3], 高 庆[1,2], 马 森[1,2], 张世琨[1,2], 胡文蕙[1,2], 张兴明[4]

[1](高可信软件技术教育部重点实验室(北京大学),北京 100871)

[2](北京大学 软件工程国家工程研究中心,北京 100871)

[3](北京大学 软件与微电子学院,北京 102600)

[4](之江实验室,浙江 杭州 311121)

通讯作者: 高庆, E-mail: gaoqing@pku.edu.cn, 马森, E-mail: masen@pku.edu.cn

**摘 要**: 代码相似性检测(code similarity detection)是软件工程领域的基本任务之一,其在剽窃检测、许可证违反检测、软件复用分析以及漏洞发现等方向均起着重要作用.随着软件开源化的普及以及开源代码量的高速增长,开源代码在各个领域的应用日益频繁,给传统的代码相似性检测方法带来了新的挑战.现有的一些基于词法、语法、语义的检测方法存在算法较为复杂、对解析工具有依赖性、消耗资源高、可移植性差、候选对比项数量较多等问题,在大规模代码库上有一定的局限性.基于相似哈希(simhash)指纹的代码相似性检测算法将代码降维至 1 个指纹,能够在数据集规模较大的情况下实现快速相似文件检索,并通过海明距离阈值控制匹配结果的相似度范围.通过实验对现有的基于代码行粒度的相似哈希算法进行验证,发现其在大规模数据集下存在行覆盖问题,即高频行特征对低频行特征的覆盖现象,导致结果精确度较低.受 TF-IDF 算法思想启发,针对上述问题创新性地提出了分语言行筛选优化方法,通过各种语言的行筛选器对代码文件行序列进行筛选,从而消除高频出现但语义信息包含较少的行对结果的影响.对改进前后方法进行一系列对比实验,结果表明,改进后的方法在海明距离阈值为 0~8 的情况下都能够实现高精确度的相似文件对检索,当阈值为 8 时在两个数据集下的精确度较改进前的方法分别提升了 98.6%和 52.2%.在所建立的 130 万个开源项目、386 486 112 个项目文件的大规模代码库上进行了实验,结果表明所提方法能够快速检测出待测文件的相似文件结果,平均单个文件检测时间为 0.43s,并取得了 97%以上的检测精度.

**关键词**: 代码相似性检测;代码同源分析;大数据;相似哈希;代码指纹生成

**中图法分类号**: TP311

中文引用格式: 李玫,高庆,马森,张世琨,胡文蕙,张兴明.面向代码相似性检测的相似哈希改进方法.软件学报,2021,32(7):2242−2259. http://www.jos.org.cn/1000-9825/6271.htm

英文引用格式: Li M, Gao Q, Ma S, Zhang SK, Hu WH, Zhang XM. Enhanced simhash algorithm for code similarity detection. Ruan Jian Xue Bao/Journal of Software, 2021,32(7):2242−2259 (in Chinese). http://www.jos.org.cn/1000-9825/6271.htm

## Enhanced Simhash Algorithm for Code Similarity Detection

LI Mei[1,2,3], GAO Qing[1,2], MA Sen[1,2], ZHANG Shi-Kun[1,2], HU Wen-Hui[1,2], ZHANG Xing-Ming[4]

[1](Key Laboratory of High Confidence Software Technologies of Ministry of Education (Peking University), Beijing 100871, China)

[2](National Engineering Research Center for Software Engineering, Peking University, Beijing 100871, China)

* 基金项目: 2019 年工业互联网创新发展工程-工业软件源代码安全检测工具项目; 之江实验室“先进工业互联网安全平台”项目(2018FD0ZX01)

Foundation item: 2019 Industrial Internet Innovation Development Project-industrial Software Source Code Security Detection Tool Project; “Advanced Industrial Internet Security Platform” Project of Zhijiang Laborator (2018FD0ZX01)

本文由“面向非确定性的软件质量保障方法与技术”专题特约编辑陈俊洁副教授、汤恩义副教授、何啸副教授以及马晓星教授推荐.

收稿时间: 2020-09-16; 修改时间: 2020-10-26; 采用时间: 2020-12-14; jos 在线出版时间: 2021-01-22

[3](School of Software and Microelectronics, Peking University, Beijing 102600, China)
[4](Zhejiang Laboratory, Hangzhou 311121, China)

**Abstract**: Code similarity detection is one of the basic tasks in software engineering. It plays an effective and fundamental role in plagiarism, software licensing violation, software reuse analysis, and vulnerability discovery. With the popularization of open source software, open source code has been frequently applied to multiple areas, bringing new challenges to traditional code similarity detection methods.Some existing detection methods based on lexical, grammar, and semantics have problems such as high computational complexity, dependence on analytical tools, high resource consumption, poor portability, having a large number of comparison candidates, and so on. Simhash-based code similarity detection algorithm reduces the dimension of the code to a fingerprint, which can realize fast near-duplicate file retrieval on a large dataset. It controls the similarity of matched results through the Hamming distance threshold. This study verifies existed simhash algorithm with line granularity through experiments, and discovers the line coverage problem in large-scale datasets. Inspired by the idea of TF-IDF algorithm, a language-based line-filtering optimization method is proposed to deal with it. Line sequences of code files is filtered through line filters in various languages to eliminate the impact of lines that appear frequently but contain less semantic information on the results. After a series of comparative experiments, this study verifies that the enhanced method always achieves high precision with Hamming distance threshold set from 0 to 8. Compared to the method before enhancement, the proposed method improves the precision by 98.6% and 52.2% on two different datasets with threshold set to 8. Based on the large-scale code database built from 386 486 112 files in 1.3 million open source projects, it is verified that the proposed method can, keeping the high precision of 97%, efficiently detect similar files with an average speed of 0.43s per file.

**Key words**: code similarity detection; code homology analyze; big data; simhash; code fingerprint generation

现如今,随着软件代码开源化的日益普及,开源代码量正以光速增长.无论是在企业还是科研单位,越来越多的开发者选择复制粘贴已有代码以提高软件开发效率.各大开源社区的蓬勃发展,吸引了无数用户共享共建开源项目,也有无数衍生软件随之产生,作用于生产生活与科研工作中.开源代码的引入主要有以下两点优势:一方面,对现有代码的直接使用大幅度提高了生产效率,降低了成本;另一方面,著名的开源项目通常由国内外优秀开发者、公司进行开发和维护,软件质量高且结构规范.

然而,随着软件的不断更新,软件功能不断增加,这些重复代码和克隆代码对软件质量、可用性和可维护性的负面影响愈发突显.从开源项目引入的代码降低了软件开发人员对软件系统整体的理解和把控,外来代码与软件系统本身代码之间可能会出现冲突,开源代码中的漏洞也可能会随着代码的复制被引入到项目中,带来安全隐患.针对这一问题,研究人员通常使用代码相似性检测技术来检测软件工程中的相似代码.

自 20 世纪 70 年代起,学术界涌现了大量的代码相似性检测工具和方法,广泛应用于代码克隆检测、软件许可证违反检测、软件抄袭检测、漏洞缺陷发现等方面.随着开源代码量的不断增加,代码相似性检测的大规模化已成为必然趋势.大规模相似检测除了可以帮助开发人员管理项目中的开源代码外,还可用于发现软件系统中组件来源、进行代码搜索、研究组件引用频率等,有助于保证软件质量,降低开发与维护的成本.目前常用的代码相似性检测方法包括:基于指标(metrics-based)、基于文本(text-based)、基于词法(token-based)、基于树(tree-based)和基于图(graph-based)这 5 个层次[1].出于对规模和效率上的考量,本文在基于文本的行粒度相似哈希算法的基础上进行了优化,提出了分语言行筛选的相似哈希检测方法.该方法一方面延续了相似哈希高效相似检索的特性,将源代码预处理后映射为一个二进制数(即代码的指纹),以实现对代码文件的降维和索引,加速大规模库的构建;另一方面结合不同语言代码的特性,排除了常见行对指纹生成的影响,使得指纹结果更能够体现代码特性,以大幅度地提高相似检测的精确度.该方法具有易于实现、部署简单、无其他词法或语法分析器依赖、构建效率高以及语言迁移成本低等特点,与此同时,其高精确度的特点保证了结果中错误匹配对数量较低,降低了后续人工核验的成本.

本文的主要贡献可以简要总结如下.

(1) 对 Zhu 等人[2]提出的基于行粒度的相似哈希算法进行实验分析,发现现有方法中大量存在的行覆盖现象.受 TF-IDF[3]思想启发,本文针对代码数据集的独有特点创新性地提出了基于分语言行筛选的优化方法,在现有方案上引入了常见行筛选器以处理行覆盖问题.通过两轮实验验证改进后算法的有效性,精确度分别为 99%和 97%,对比现有方法分别提高了 98.6%和 52.2%.

(2) 将改进后的相似哈希算法与 MD 5 算法进行对比,证实其能够覆盖 MD 5 方法 99%以上的匹配结果,比使用 MD 5 的方法多获取到 40%以上的相似文件对,并进一步匹配到 44%以上的包含相似成分的工程对.

(3) 基于 Github 开源项目,构建了项目数量达 130 万、文件数量达 3.86 亿的大规模代码指纹库.通过索引和分表,文件平均检测速度达到 0.43s/个,并抽取了 309 对工程对加以验证,结果表明改进后的算法能够在大数据集上保持 97%以上精度的检测效果(实验结果可从 https://Github.com/Nica97/simhash_experimental_results 下载).

本文第 1 节介绍代码相似性检测领域研究背景,对比传统指纹与相似哈希在原理和应用上的不同,并回顾相似哈希在代码相似性检测领域已有的研究成果.第 2 节介绍传统相似哈希方法及行粒度相似哈希筛选方法,深入分析后者在大规模数据集下产生的行覆盖现象及其原因,并提出本文的分语言行筛选的相似哈希优化算法.第 3 节提出 3 个研究问题(research question,简称为 RQ),对实验数据集、本文采用的评测指标以及整体的实验流程进行介绍.第 4 节分别针对 3 个 RQ 的实验结果进行展示和分析.第 5 节总结本文工作并展望下一步研究.

## 1 相关工作

### 1.1 代码相似性检测

在代码相似性检测流程中,如何选取合适的方法对代码进行特征提取和表达是最为关键的步骤,依照提取特征层次的不同,可分为基于指标、基于文本、基于词法、基于树和基于图 5 这个层次[1].早期的检测方法通常基于指标或基于软件量测,如 Ottenstein[4]提出的基于霍尔斯特德复杂度测量指标[5]的检测方法中将源代码表示为不同操作符个数、不同操作数个数、操作符出现总次数、操作数出现总次数构成的 4 维向量并进行比较,虽然该类方法较其他类型方法实现简单但缺乏有效性.基于文本的检测方法只对源代码进行少量的简单处理,主要将源代码看作文本来进行处理,如著名工具 NiCad[6]通过最长公共子序列(longest common subsequence)对比两个从源代码中获取的字符串序列来获取相似代码.基于词法的检测方法则通常利用解析器将源代码分成符号序列,并对符号序列进行比较和分析得出检测结果.比较著名的工具有 CCFinder[7]、CCAligner[8]、SourcererCC[9]等.基于树的检测方法将源代码转化为抽象语法树(AST)来进行分析,如 CloneDR[10]提取了 AST 的信息作为中间表示并通过树匹配技术对获得的有效信息进行克隆分析.基于图的检测方法可以捕捉程序的语义信息,例如基于程序依赖图(PDG)的检测方法 Duplix[11]等等.

在目前代码大数据的情境下,传统的代码相似性检测方法面临着新的挑战[12].采用基于树或图的检测方法通常需要依赖于特定语言的语法分析器以生成抽象语法树、程序依赖图等代码特征信息,处理过程消耗了大量计算资源且可扩展性有限,在大规模库上的应用较少.对于基于词法的检测方法,通常将代码转化为一组 token 序列,再通过各种相似度衡量方法,如后缀树(suffix tree)、Jaccard 相似度、字符串对齐等方法进行相似性度量.相较于基于树和基于图的方法,其效率更高,适用于更大规模的数据集.这些方法中通过各种手段对检测的候选结果进行筛选,如仅对 token 序列子集的每一位进行索引[9]、对 *n*-gram 的部分 token 建立倒排索引等[8],获得了不错的效果,如 SourcererCC[9]可以检测的规模达到 250MLOC(lines of code).但是,随着规模的进一步扩大,候选 token 序列的规模亦将持续扩大,与 token 序列之间的对比会消耗大量时间.对此,本文主要采用文本层次的基于相似哈希指纹的相似检测方法,将源代码转化为一串二进制字符串,两两比较效率更高.通过分段索引和按语言分表的方法,将候选序列范围大幅度降低,以达到快速检索的效果.针对文本检测在代码相似性检测上出现的问题,结合语言的特点进行了优化,语言迁移成本极低,在大规模库上构建的效率也比较高.

### 1.2 传统指纹与相似哈希

在大规模代码库中,为了快速获取相似的代码文件,可以将源代码降维为指纹(fingerprint),再通过指纹之间的比较进行快速匹配.正如人的指纹能够代表个人身份,代码相似性检测中的指纹也常用来代表输入代码的独有特性.指纹计算方法的输入对象可以是一个文件、一个方法,甚至某一行代码,对其进行一系列变换、降维后

将最终生成一个字符串或一个二进制数,也就是计算对象的指纹值.两个指纹之间的比较相当于两个二进制数之间的比较,与两段源代码之间直接两两比较相比,效率得到了大幅提升.在构建大规模代码库时,采用将输入代码转化为指纹后入库的方法也能够节省存储空间,便于索引的建立.如图 1 所示,左侧为输入的待测代码文件,右侧分别展示了其对应的 MD 5 指纹值、改进前后的相似哈希指纹值.

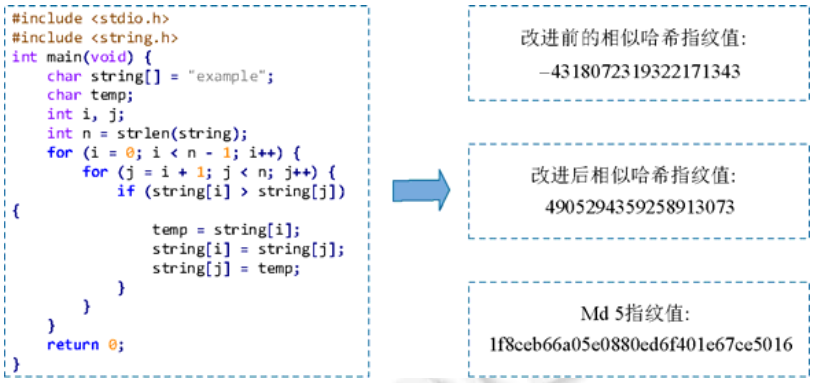


Fig.1 Examples of fingerprints

图 1 指纹生成示例

MD 5 消息摘要算法(MD 5 message-digest algorithm)即是一种常用的指纹生成方法,能够将输入内容视作一串长字符串,在进行一系列填充、不可逆变化等操作后生成一个独特的散列值.在代码漏洞检测领域,MD 5 可用于代码文件以及函数的指纹生成.漏洞代码同源检测工具 VUDDY[13]通过 MD 5 对代码进行降维映射,因此能够在大规模代码上具有良好的效率.VUDDY 将程序中的函数提取出来,进行抽象化和正则化的处理(包括对变量名、参数名、数据类型的重命名等),然后通过 MD 5 算法生成每一个函数对应的指纹,在漏洞代码指纹库中查找是否有相同的指纹,并以此为索引找到工程包含的相关漏洞信息.

传统哈希算法,如 MD 5 在本质上是将原始内容尽量均匀而随机地映射为一个指纹值,只能判断输入双方是否相同,而无法衡量两者相似的程度,因此在应用场景上具有局限性.即便只对输入内容修改一个空格,传统哈希算法也将产生截然不同的指纹值,而局部敏感哈希 LSH(locality sensitive hashing)[14]就能够很好地解决这一问题.

Charikar 等人[15]提出的相似哈希(simhash)算法是一种局部敏感哈希算法,它能够在将文件降维成一个指纹值的同时保留文档的特性.只需比较两个文件指纹的海明距离(Hamming distance)[16],就能高速、便捷地衡量文件之间的相似程度.谷歌公司的 Manku 等人[17]将相似哈希用于海量相似网页去重,将每个网页转化为一个 64 位的指纹值,构建一个数十亿量级的指纹库并取得了良好的实验结果.

由于相似哈希相似检索的快速性,它在很多领域均有应用.Xu 等人[18]将其用于分布式环境下的推荐系统中,将用户的服务信息转换为相似哈希指纹,既保护了不同云平台的用户信息隐私,又提高了推荐系统的效率和可扩展性.Rezaeian 等人[19]利用相似哈希进行俄语文件中剽窃内容的检索,使用 $N$ grams 方法将文件转换为一组连续的字符串,并将它们作为相似哈希计算的特征.

在代码分析领域,同样也有相似哈希的相关应用.王勇等人[20]将相似哈希用于安卓恶意应用程序的检测,运用动态分析和静态分析相结合的方法获取了多维度特征,并通过信息增益的方法进行筛选.另外,他们还提出了基于 simhash 算法的特征融合方法,将特征进行降维,通过对权重进行调节的方式解决了特征维度不平衡的问题.Uddin 等人[21,22]曾将相似哈希和经典克隆检测工具 Nicad[6]两种方法相结合提出了一种新的 SimCad 代码克隆簇聚类工具,主要用于对一个大型软件系统内部源代码进行检测,获得多组相似代码以及克隆簇.SimCad 将每个代码文件切分为代码段,对其进行格式化和标准化处理后转换为 simhash 指纹,同时建立索引.最后使用 DBSCAN 聚类算法对结果进行聚类,获得代码克隆簇.他们还提出了 Simcad 的双层索引策略,第 1 层中根据代

码段的行数进行索引,第 2 层中根据二进制哈希值中 1 的位数进行索引.通过这样的索引方式,加速聚类过程中获取相邻节点的速度.Qiao 等人[23]也将相似哈希用于二进制同源检测中,他们将二进制文件进行反编译,以汇编代码中的函数为单位,计算出每个函数标准化后的 simhash 指纹值,并建立两类优化索引.郭颖等人[24]认为,以词的频率为特征在代码克隆检测领域的效果不尽人意,于是结合数据消重中的基于内容可变长度分块(content-defined chunking,简称 CDC)思想和 simhash 算法思想,把源代码转换为词法单元序列,并采用 CDC 分块将词法单元序列分块并计算出指纹值.在这一方法中,并未直接采用 simhash 算法计算指纹,而是借鉴了 simhash 算法的思想,将所有代码文件的哈希值序列从大到小进行排序,并依次检查两个序列中每一个哈希值是否相同,如果两个文件的哈希序列有超过某个百分比以上的哈希值是相等的,则聚为一类.Zhu 等人[2]指出,由于代码中经常会出现重复词,以词粒度来获取代码相似哈希特征容易出现高频的特征把低频的特征覆盖掉的现象,于是提出以代码行为粒度进行特征累加,并且在预处理中筛除了纯符号行和空行,对相似哈希在代码检测方面的应用作了进一步改进.

## 2 一种改进的相似哈希代码克隆检测方法

### 2.1 传统相似哈希方法与行粒度相似哈希方法的提出

相似哈希算法能够将一个文档转换成一个指定长度的二进制数,即当前文档的指纹.指纹的长度通常取决于计算过程中采用的哈希算法输出值的长度.

相似哈希算法主要分为 5 个基本步骤.

(1) 特征提取:根据数据的特点进行特征提取,对于文本类数据常采用分词结果、文档关键字、文档标签等作为特征,对于视频类数据可以提取时空特征,对于网页可以提取超链接作为特征等.在提取特征的同时给每个特征赋予一个权重,在文本数据集中常用词频或 TF-IDF[3]方法来获取权重,亦可结合不同数据集的特征设计不同类型的权重值.

(2) 哈希:这一过程通常采用各种传统哈希算法将上一过程中的各个特征转化为哈希值.

(3) 加权:结合步骤 2 生成的各个特征的哈希结果与其权重生成加权字符串,哈希值每一位的 0 或者 1 决定了其与权重值是正相乘还是负相乘.

(4) 累加:将上一步骤中所有特征的加权结果按位累加,得到结果序列串.

(5) 降维:对加权求和获得的结果序列串进行变换.序列串中,每一位的值若为正数,则变换为 1,否则,变换为 0,得到了最终的相似哈希指纹值.

我们主要通过海明距离(Hamming distance)[16]对指纹的相似性进行比较.两个指纹值对应的二进制数(01 串)取值不同的位的数量称为这两个指纹的海明距离.对于二进制字符串 $a$ 和 $b$,海明距离等于 $a$ XOR $b$ 运算结果中 1 的个数.

在 Uddin 等人[21,22]提出的 SimCad 工具中,大规模软件系统的源代码被拆分为多个小代码块,每个代码块采用空格分割的方法计算出对应的相似哈希指纹值,通过聚类算法获取到大规模软件系统中相似的代码块簇.虽然该工具仅用于单个软件系统内部的相似代码发现,无法应用于大规模代码相似性检测,但其通过实验验证了相似哈希在软件代码相似性检测中应用的可行性.

在将相似哈希应用到基于大规模代码库的同源分析过程中时,由于数据规模的庞大有时采用单个代码文件作为生成指纹的单位.Zhu 等人[2]发现,若以词为特征粒度来获取源代码文件的相似哈希指纹值,则会出现词频较高的词将词频较低的词覆盖的现象.由于代码数据集的特殊性,某一个词在代码文件中可能会频繁出现,若采用传统的分词方法直接对源文件进行分词,高词频词对结果的影响将会非常显著,甚至出现指纹结果完全等同于某一个词的哈希值的情况,显然这样的问题会带来很高的误报率.尽管源代码文件中可能会出现大量相同的词,但行与行之间的语义是较为独立的,同一个文件内不容易出现大量相同内容的行.因此,他们开始尝试以行作为特征提取的粒度.为了避免一些大量而无意义的行对算法结果的影响,该方法使用正则表达式对每行的字符串进行筛选,筛除不包含任何字母或数字的纯符号行或空行.最后得到改进后的相似算法,如图 2 所示.算法

将源代码文件以行为粒度进行切分,并对得到的行序列进行筛选和预处理.然后,对每一行进行传统哈希值计算并映射为一串由 1、–1 组成的序列.接着,对每一行的映射结果进行逐位累加,得到一个有符号数的序列.最后,对序列进行降维,得到改进后的相似哈希指纹结果.

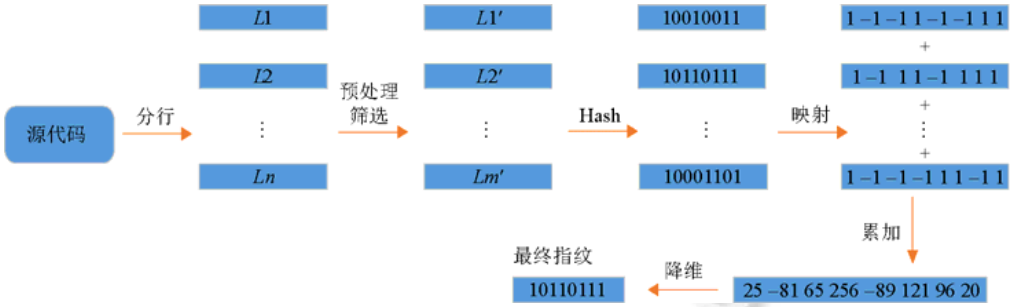


Fig.2 Simhash algorithm with line granularity

图 2 行粒度相似哈希方法

### 2.2 大规模数据集下行覆盖现象及分析

本文对上述提到的行粒度相似哈希检测方法[2]进行了复现,并在实际工程代码上进行了一系列的实验.尽管行粒度的代码特征提取算法在理论上一定程度地解决了词粒度特征提取算法中的词覆盖问题,但在实际的检测与实验中发现,行粒度提取方法仍会出现一些特殊的高频次特征对低频次特征的覆盖情况.经过实验分析和统计,行覆盖现象主要出现在以下两种情况中.

- 在代码中频繁出现的一些包含关键字的代码行,例如“break”“return;”“try{”.在某些情境下,这些行出现的频率非常高,以至于它对结果的影响覆盖了所有其他行的影响.
- 由于代码功能的特殊性(如一些表示信息提示的工具类或测试类),在代码中会频繁出现一些功能性行内容,例如对某一种方法的频繁调用、在多个方法里实现传递同一信息的功能等.

在以上两种情况中,后者的覆盖现象可以视作是文件代码功能单一性的体现,所以本实验中具有大量重复相同方法的两个相似代码文件对不算作在代码相似度量过程中的误报.第 1 种情况中这些代码行的大量出现不能特征性地代表代码文件的功能,因此会对结果的准确性带来较大影响.

针对行覆盖现象,如果单纯地将文件里频繁出现的行的权重降低,可能会将第 2 种情况中重复调用方法的行对指纹的影响减弱,无法体现文件的特殊功能,造成此类文件的漏报,因此本文选择从不同语言常见行的消除这一角度出发来降低行覆盖问题对结果的影响.

### 2.3 针对语言的行筛选优化方法

TF-IDF[3]是一种广泛应用于数据挖掘和信息检索的加权算法,其中,TF(term frequency)是指词频即单词在文档中出现的频次,IDF(inverse document frequency)是指逆向文档频率.其主要思想为一个词在一篇文章中出现的频率越高,且其在其他文章中出现的次数越低,说明这个词具有较好的区分能力,应该赋予它更高的权重.第 2.2 节中提到的第 1 种行覆盖现象通常出现在一些由于代码的特殊语法结构而频繁出现的行上,这些行在大量的代码文件中广泛存在,但包含的语义信息有限,且对代码文件之间的区分起到的作用极小.在现有的相似哈希方法中,高频行在计算特征时以出现的频次为权重,即体现了 TF 的信息,但是对于行的 IDF 信息并没有相应的体现.对此,我们对代码中出现的常见重复行进行了统计与分析,并结合统计结果创新性地提出了分语言行筛选优化方法.

为了排除由于代码的特殊语法结构而频繁出现的一些代码行对结果的影响,本文收集了 Github 上面按星数排行 Top 300 的工程项目作为统计来源,这些排名靠前的项目具有规模较大、影响范围较广、结构较规范等特点.根据后缀名筛选出其中的代码文件,并进行去空行、去行内空格、去注释等预处理步骤,对得到的结果中的代码行进行了统计,获取到常见重复行列表,部分结果见表 1.

在共计达 519 万行的结果中,仅 37 735 行的统计次数达到 20 次及以上,仅 10 473 行统计次数达 50 次及以上,而 100 次以上的行数仅有 4 679 行,可见在实际工程项目中频繁出现的行集中于极少一部分行内容上,这些行往往是与代码中一些特殊语义结构相关的.因此,只要消除了这些常见行所带来的负面影响,即筛除这些常见行而保留那些罕见且更能体现源代码功能特性的行,就能够解决大部分现存的误报,大幅度地提高算法可用性,本文第 4 节的实验结果也证实了这一点.

**Table 1** Common lines in code files

**表 1** 常见重复行

| 排序 | 行内容 | 出现次数 |
|---|---|---|
| 1 | }else{ | 74 960 |
| 2 | break; | 48 733 |
| 3 | @Override | 42 073 |
| 4 | break | 29 730 |
| 5 | return; | 24 601 |
| 6 | try{ | 22 353 |
| 7 | returnfalse; | 21 238 |
| 8 | return | 15 586 |

同时,可以注意到,如果将所有语言的代码文件一起进行常见行统计,其结果是不够合理的.这些影响结果的常见行往往跟语言相关联,不同语言的常见行差异非常显著,因此不能采用统一的筛选器对所有语言的源代码进行行筛选.对此,本文选取了 10 种常用语言(c#、c/c++、go、java、js、php、python、ruby、sql、swift),并对每种语言分别进行常用行统计.数据源选自 Github 星数排名前 50 000 的开源项目,将其中每种语言的源代码文件进行预处理后按出现频数降序排列,其中部分语言的常见行统计到前 15 名,结果见表 2.从表 2 可见,在代码文件中频繁出现的行不仅包含纯符号行,也包含一些特殊的与语义相关的行,这些行在代码中的频繁出现并不能直接地体现代码功能,包含的语义信息也较少,因此并不属于第 2.2 节中提到的频繁出现一些功能性行内容的行覆盖情况.若仅通过纯符号行对结果进行筛选,其余频繁出现且特性体现较差的行就会影响指纹特征的提取,比重较大时将出现行覆盖的情况,严重影响结果的精确度.同时,尽管不同语言之间有一些通用的常见行,如“returnfalse;”“}else{”“return”等,但是由于不同语言的语法特点不同,其常见行列表之间差异性也较大,存在一些语言特有的常见行,如 python 中的“pass”、ruby 中的“ensure”等.

**Table 2** Common code lines in different languages

**表 2** 部分语言常见行统计结果

| 语言 | 行内容 | 语言 | 行内容 | 语言 | 行内容 | 语言 | 行内容 |
|---|---|---|---|---|---|---|---|
| JAVA | } | PHP | } | python | else: | ruby | end |
| | @Override | | { | | ) | | else |
| | { | | <?php | | try: | | } |
| | @test | | ); | | } | | ) |
| | try{ | | ), | | }, | | beforedo |
| | }else{ | | ], | | pass | | begin |
| | break; | | }else{ | | ] | | { |
| | returnfalse; | | array( | | ), | | }, |
| | returnthis; | | ]; | | return | | private |
| | returnnull; | | break; | | @property | | ] |
| | }); | | [ | | continue | | includeaws::structure |
| | returntrue; | | returnfalse; | | }), | | super |
| | return; | | return$this; | | returnfalse | | [ |
| | importjava.util.list; | | ] | | }], | | ensure |
| | }; | | ) | | break | | require'spec_helper' |

如图 3 所示,左右两边的代码分别节选自不同工程的两个文件,从语义和功能的角度来看,两者截然不同.根据现有的相似哈希计算方法,两个文件的每一行都将作为一个特征来共同计算指纹值,由于两边均多次出现“#endif”行,导致最终计算出来的相似哈希指纹值的海明距离为 7.可以看到,两边多次出现的“#endif”行是对区分完全没有帮助的,真正能够体现程序具体特性的是剩下的那些行.因此,与 TF-IDF 方法不同,本文提出的算法

采用将这些常见行直接筛除的策略,仅保留那些有文件特色的行参与指纹的计算.

由此,本文在行粒度相似哈希算法的基础上进行了改进,结合代码文件的结构特点引入了常见行筛选列表以消除一些频繁出现且包含语义较少的行,如图 4 所示.从上述 10 种常见语言的行统计结果中对每种语言各取前 20 000 行作为该语言行筛选器的筛选内容,不同语言的行筛选器一起组成常见行列表.对于一个待测源代码文件,首先进行去空行、去注释、去空格、全部大写转换为小写的预处理过程,并将代码文件分行;其次根据文件的后缀名判断所属语言,从常见行列表中选取对应语言的常见行筛选器,并将文件中包含在筛选器中的对应行全部筛除,剩下的每一行作为该代码文件的特征;然后通过 MurMur Hash[25]方法对这些特征依次进行哈希计算,并将得到的结果逐位映射为一串由 1、–1 组成的序列;最后将这些序列经过累加和降维得到最终的指纹结果.使用本文提出的方法对图 3 中的例子进行指纹计算,两者的海明距离由 7 变为了 27,可见改进后的算法很好地体现了两段代码的差异.

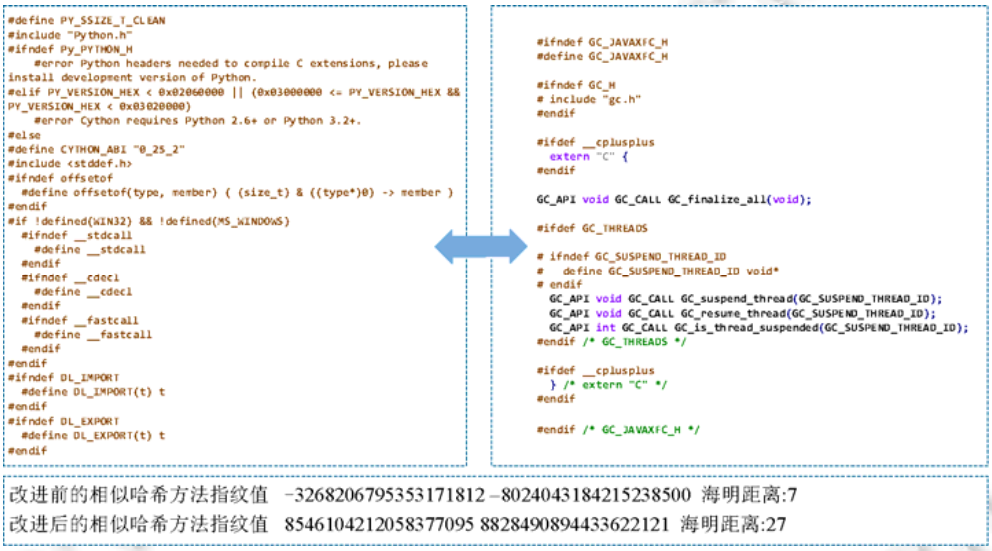


Fig.3 Examples of meaningless repeating lines

图 3 无意义重复行示例

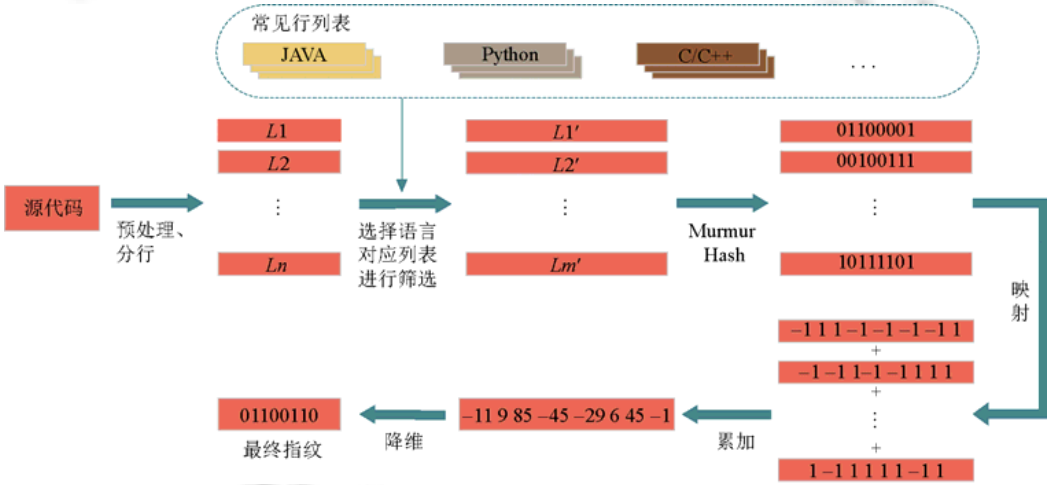


Fig.4 Enhanced simhash algorithm with line granularity

图 4 改进后的行粒度相似哈希方法

综上所述,本文针对行粒度相似哈希算法在实际工程上暴露的问题对相似哈希计算过程中的特征提取功能进行优化,通过对 Github 上著名开源项目的统计,构建不同语言的相应常见行列表并组成常见行列表库.在获取相似哈希特征时使用对应语言的常见行列表进行特征筛选,以消除常见重复行对其他行的覆盖.

## 3 实验设计

为了全面评价改进后的相似哈希方法是否在真实数据集上有较好的检测效果,本文设置了以下 3 个研究问题,对改进前的相似哈希方法、改进后的相似哈希方法、MD 5 方法以及文件名匹配的方法进行了对比和分析.其中,改进前的相似哈希方法是指 Zhu 等人[2]提出的基于行粒度提取特征且仅筛选纯符号行的相似哈希方法,改进后的相似哈希方法是指本文提出的基于行粒度提取特征并根据语言筛选常见行的相似哈希方法.

- RQ1:在真实工程构成的数据集上,改进后的相似哈希检测方法与改进前相比能否展现出更佳的效果?
- RQ2:简单匹配同名文件能否达到与本文方法相近的效果?
- RQ3:改进后的相似哈希方法在大规模数据集上的表现如何?

在本文的实验设计中,未与现有其他大规模代码相似性检测工具,如 SourcererCC[9]进行比较.其主要原因如下.

(1) 目前的相似性检测工具多以方法、函数为检测粒度,本文算法的检测粒度为文件级别.在 SourcererCC[9]、CCAligner[8]等工具上广泛采用的数据集 BigCloneBench[26]同样是基于函数粒度的,因此无法将本文的算法直接运用于该数据集上.

(2) 本文提出的算法与其他相关工具的目标有所不同.本文的算法目标是在更大规模的数据量场景下实现粗粒度、高效率、高精度的文件相似检索,现有方法大多在小规模数据上获取更加细粒度、深层次的相似检测.现有的能够适应大规模数据的工具大多采用基于 token 的方法,采用的数据集量级最大为 BigCloneBench 的 IJaDataset,包含 25 000 个开源工程,共计 300 万文件.而本文的数据集包含 130 万个开源工程,共计约 3.8 亿代码文件,远大于现有数据集量级.

### 3.1 实验数据集

为了验证改进后的相似哈希方法在实际工程同源检测中的有效性,本文按照 Github 星数排名的顺序选取了较前位置的若干开源项目作为本文实验的数据集.这些评测对象是 Github 上比较著名的开源项目,代码规模较大,项目活跃时间较长,影响范围较为广泛,且覆盖了不同类型的应用领域,如比特币 bitcoin、文本编辑器 vim、图片处理工具 ImageMagick 等,因此具有一定的代表性.

在 RQ1 和 RQ2 的实验中随机选取了 Github 星数排名前 7 万个项目中的 200 个,并获取它们最新版本的源代码作为数据集,共计约 100 万个文件.将这些文件按照后缀名进行筛选,仅保留第 2.3 节中统计过的相关常见行的 10 种语言的源代码文件作为本次实验的研究对象,筛选后共计 297 210 个.

在 RQ3 的实验中构建了大规模的代码知识指纹库,按照星数排名的降序将 Github 上的项目进行入库,目前共入库 1 320 663 个项目的共计 6 244 310 个版本,预处理的方法与上述一致.由于空文件和极小的代码文件包含的语义信息较少,在同源分析中的重要性较低,本次实验对可执行行数小于 15 行和相似哈希指纹值等于 0 的文件进行了筛除.

### 3.2 评测指标

为了论证相似哈希改进前后的检测精确程度,本次实验设计了与代码文件行成分相关的精确度度量指标,以用于实验中对几种方法的精确度度量.首先对匹配文件对进行去空行、去空格、去注释的预处理,然后分别获取两个文件的行序列,并根据两文件行序列匹配的结果判定文件是否相似,判定标准如下.

任意满足以下两个条件之一即可判断两个文件相似.

(1) 两文件行序列共同行数/文件 1 文件行数≥50%且两文件行序列共同行数/文件 2 文件行数≥50%.

(2) 两文件行序列共同行数/文件 1 文件行数≥70%或两文件行序列共同行数/文件 2 文件行数≥70%.

其他情况均视作不相似.

上述两个条件均用于判断两个文件是否相似,其主要针对实际文件引用中的两个场景.其中,条件 1 针对的引用场景为开发者对引用文件进行了多次修改,但该引用文件的整体结构不变,因此两个文件的共同行数均占两个文件的一半以上.条件 2 针对的场景为开发者在原有引用文件的基础上额外增加或删除了一些方法和功能,因此两个文件的行数有一定差距,对其中的大文件来说增加的内容可能超过了 50%,因此无法满足条件 1,但是两个文件在实际功能上仍然是相似的.因此,条件 2 保证,当大文件与小文件的共同行数占小文件 70%以上(即大文件保留了小文件 70%以上的行)时将两个文件视作相似文件.

上述指标中,50%和 70%是我们根据实际开源项目中引用文件的一般修改情况选取的经验性取值,根据实际应用场景的不同,可以对指标进行调整和修改.为了验证该评测指标的合理性与正确性,在 RQ1 和 RQ3 的检测结果中,随机抽取各个海明距离在该标准下判断为正确的相似文件对各 10 对,共计 180 对结果并进行人工核验.主要针对的问题是判断为相似的文件对是否真正结构相似,是否会出现大量行相同但相同行所处的位置不同导致整体代码功能结构上并不相似的情况.核验结果表明,所有参与人工核验的文件对抽样都是结构相似、功能相关的文件对,在同源检测上有显著的相关性.具体人工核验结果统计情况见表 3.

**Table 3** Manual validation of the evaluation metric

**表 3** 评测标准人工核验结果

| 验证结果(文件对之间的关系) | 数量 | 百分比(%) |
|---|---|---|
| 预处理后完全一致 | 17 | 9.44 |
| 修改个别行,其他均一致,整体语义几乎不影响 | 46 | 25.56 |
| 修改或增删部分行,其他部分一致 | 79 | 43.89 |
| 多处修改或增删,整体结构一致 | 26 | 14.44 |
| 一个比另一个增加了一些部分,即一个是另一个的子集 | 12 | 6.67 |

在本次实验的相似性匹配过程中,涉及到对以下两个名词的定义.

(1) 相似文件对:这是指两个不同工程内部发现的一对相似的代码文件,可能产生于一个工程对另一个工程内文件的复制和复用.引用的文件可能经过了后期的增删和修改,因此与源文件有所差别.在实验过程中,主要通过 MD 5 指纹和相似哈希指纹值的匹配来发现相似文件对.

(2) 相似工程对:对于一个工程,只要其包含的所有代码文件中有任意一个文件与另一个工程中的文件被匹配为相似文件对,就将这两个工程认定为相似工程对.

在实际应用过程中,两个相似工程对的确认有助于我们发现随着代码复用而进行传播的漏洞和缺陷,发现抄袭或许可证违反现象.我们也可以使用文件相似性检测来进行粗粒度的代码搜索.

在验证过程中,相似文件对的验证方法为:通过本节提出的评测指标来评判两个文件是否的确相似.对相似工程的验证方法为:使用本节提出的评测指标对两个工程之间的所有文件对进行两两评判,只要存在一对真实相似的文件对,即判定两个工程对的确相似.

在实验分析过程中,本文选用了精确度、漏报、匹配对数、检测时间等指标来分析算法的有效性和效率.TP 表示真正例(即判断为相似且真实相似的文件对),TN 表示真负例(未检测出相似且确实不相似的文件对),FP 表示假正例(检测出的相似文件对中实际上不相似的),FN 表示假负例(未检测出相似但实际为相似的工程对).

第 4 节中的漏报对应 FN,即假负例.精确度(precision)的计算方法为

$$\text{精确度(precision)}=\frac{TP}{TP+FP},$$

$$\text{误报率(FNR)}=\frac{FP}{FP+TN},$$

$$\text{漏报率(FNR)}=\frac{FN}{FN+TP}.$$

由于该实验数据规模较大,对海量数据文件进行依次两两对比将消耗大量的时间和计算资源,因此我们没有从误报率和漏报率的角度对算法进行分析(因为难以获得 TN 与 FN),关于本文算法的详细漏报分析见第 4.1

节和第 4.2 节相关部分.

### 3.3 实验流程

如第 3.1 节所述,在本文的实验中将分别在一大一小两个数据集上对所提出的算法进行验证.

在 RQ1 和 RQ2 的实验中,我们随机选取了 Github 星数排名前 7 万个项目中的 200 个工程,对这些工程进行筛选,仅仅保留代码文件.对于每一个工程,将其所有代码文件进行去空行、去注释、去空白符、全部转换为小写处理后,分别计算 MD 5 值(将所有文件压缩到一行视作一个字符串进行计算)和改进前、改进后相似哈希的值并连同代码文件的文件名、相对路径、所属工程名称和版本号以及文件的行数存储在一个文件中作为该项目的信息文件,信息文件的每一行代表该项目中的一个代码文件.在相似性检测过程中,我们对前述 200 个工程中的近 30 万个代码文件的 MD 5 值、改进前后的相似哈希值进行两两比对.对于 MD 5 的比对标准为,若两个文件的 MD 5 相等,则判断两个文件相似;对改进前后两种相似哈希方法的比对标准为,若两个文件的海明距离在阈值以内,则判断为相似.我们对相似哈希海明距离阈值为 0~8 的情况均进行了实验,结果如第 4.1 节所示.

在 RQ3 的实验中,为了验证改进后的相似哈希算法在实际大规模代码库上的检测效果,本文构建了大规模开源代码指纹库.对 Github 上按星数降序排列的前 130 万个开源项目的不同版本进行了指纹计算和入库,根据文件的 MD 5 值筛除同一项目不同版本之间的内容完全一致的冗余文件.构建的大规模库包含共计 6 244 310 个版本的开源项目,386 486 112 个项目文件,在数据存储上选用面向大数据的 Mongodb 分布式数据库.

由于文件数目巨大,为了加速检测速率,对所有文件进行索引和分表.

(1) 借鉴谷歌公司 Manku 等人[17]基于抽屉原理提出的索引优化方法,将每个文件的指纹值分为 5 段,长度分别为 13、13、13、13、12,任意每两段进行组合作为一个字段存储到文件文档中,共有 10 个相应字段(1&2,1&3,1&4,1&5,2&3,2&4,2&5,3&4,3&5,4&5),对每个字段建立索引以便快速检测到与待测指纹值距离 3 位以内的指纹结果.

(2) 不同语言之间几乎不存在相似文件,为了进一步减少每次需要检测的候选匹配对,将获取到的文件信息按文件语言进行分表,每次查询时按照待测文件的语言查询对应表.

在大规模库上的文件匹配同样分为通过 MD 5 匹配和通过相似哈希进行匹配.通过 MD 5 匹配时利用 MD 5 字段建立的索引搜索 MD 5 完全相同的文件,在通过相似哈希进行匹配时,先根据文件语言获取对应的表,再通过分段索引的方法找到潜在的候选文件对,对候选文件对进行两两海明距离计算,距离在阈值内的视作相似.实验结果如第 4.2 节所示.

## 4 实验结果与分析

### 4.1 对研究问题1的分析

随机选取 Github 星数排名前 7 万个项目中的 200 个工程并两两之间进行比较,相似哈希海明距离[16]阈值取 8(即海明距离小于等于 8 的文件对视作相似文件对),实验数据以及各个指纹计算方法获取到的相似文件对结果见表 4,其中,改进前的相似哈希方法是指仅筛选纯符号行且以行为特征粒度的相似哈希方法,改进后的相似哈希方法是指根据文件所属语言筛除指定常见行且以行为特征粒度的相似哈希方法,错误匹配对数是指所有检测出的文件对中根据第 3.2 节给出的判别方法判别实际为不相似的文件对数,精确度是指通过该判别方法判别为相似的文件对数占检测出文件对总数的百分比.

**Table 4** Experimental results of RQ1

**表 4** RQ1 的实验结果

| | 相似文件对 | 与 MD 5 的交集文件对数 | 错误匹配对数 | 精确度(%) |
|---|---|---|---|---|
| MD 5 | 21 879 | – | 0 | 100 |
| 相似哈希-改进前 | 8 923 644 | 21 858 | 8 816 731 | 1.20 |
| 相似哈希-改进后 | 40 082 | 21 862 | 67 | 99.83 |

从实验结果可以看出,无论是改进前还是改进后的相似哈希算法,其匹配结果都能够基本上完全覆盖 MD 5

的匹配结果.改进后的相似哈希方法获取到的相似文件对数是MD 5方法的2倍,而改进前的相似哈希方法则获取到远大于另外两种方法的文件对数.在改进前的相似哈希检测结果中出现了大量误报的文件对,主要原因是改进前的相似哈希算法在实际工程上出现了大量的行覆盖现象,包括“returnfalse;”“}else{”“break;”等.在改进后的相似哈希算法中,通过不同筛选器对不同语言的常见行进行了筛选,排除了此类行对结果的影响,使得最终匹配的精确度得到了大幅度提升.为了进一步查看不同海明距离下的检测数量以及精确度的变化情况,对改进前后两种相似哈希方法在海明距离阈值为0~8的情况下分别进行检测,结果如图5所示.

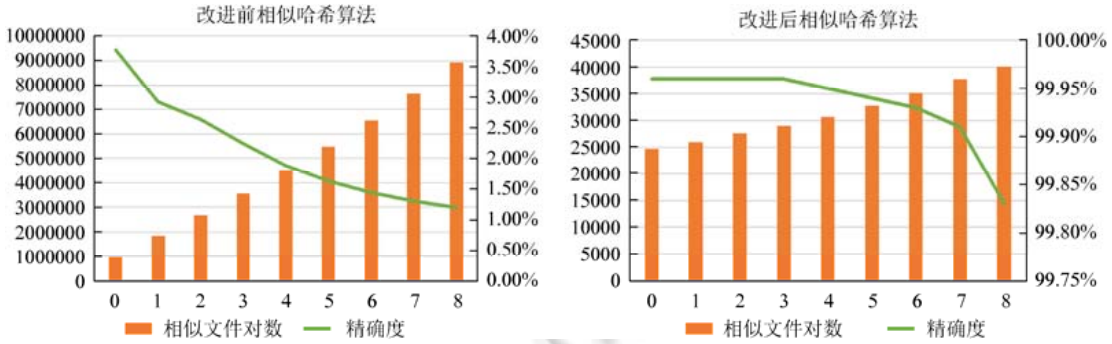


Fig.5 Experimental results of two methods under different Hamming distance thresholds

图5 相似哈希改进前后算法在不同海明距离阈值上的检测情况

可以看到,在整体检测效果上使用不同语言筛选器对特征进行行筛选的相似哈希算法都显著优于改进前算法,两种算法的检测文件数量均与海明距离阈值呈正相关,精确度与海明距离阈值呈负相关.改进后的相似哈希方法在所测的各个海明距离阈值上都能保持 99%以上的精确度,而改进前的方法在最优情况下也不足 4%,差距悬殊.

上述匹配结果均为文件级别的匹配结果,即匹配到的相似文件对中有多少是正确的.在工程级别的评测主要衡量相似性匹配算法能否寻找到更多潜在相似的工程对,为后续的工程两两对比作铺垫.获取到的相似工程对结果见表5.改进后的相似哈希方法可以覆盖全部MD 5方法匹配到的工程对,并且能够额外发现新的相似工程对,发现的新工程对的精确度也较高.改进前的方法比改进后的相似哈希方法能够匹配到更多的相似工程对,主要是因为该方法在两两比对中匹配出大量相似文件对,因此关联到的工程对数目更多.

**Table 5** Project pairs detected by simhash before enhancement, simhash after enhancement, and MD 5

**表5** 改进前后的相似哈希方法与MD 5方法的匹配工程

| 指纹计算方法 | 检测出的工程对 | 筛选后的工程对 |
|---|---|---|
| 相似哈希-改进前 | 9 697 | 813 |
| 相似哈希-改进后 | 628 | 607 |

对改进后方法无法检测而改进前能够检测到的文件对作进一步漏报分析,发现主要由两种情况所致.

(1) 改进后的相似哈希方法筛除了一些常见行,因此在一定程度上放大了剩下行的差异对结果的影响,导致本来海明距离小于等于8的结果在筛选之后大于8.

(2) 对于某些行数较小的文件,由于其整个文件均包含在其对应语言常见行筛选的范围内,导致其所有内容均被筛除,最后获取到的特征为空,即相似哈希指纹值为0,不参与后续的指纹匹配过程.

其中,情况1为出现漏报的主要原因.关于改进后方法漏报情况的进一步详细分析见第4.2节.

在指纹计算效率方面,同样对3种方法进行了统计和比较.在200个工程中的近30万个代码文件上进行预处理和3种不同指纹的计算,耗费的总时间见表6.改进后的相似哈希方法指纹生成速度与MD 5算法的处理速度相近,比改进前的相似哈希方法在速度上有所提升.其主要原因是,在对各种语言的常见行进行筛选后,一些无意义行的特征被过滤,不需要再计算其哈希值并参与累加,因此节省了时间.

**Table 6** Fingerprint generation efficiency
**表 6** 3 种指纹的计算效率

| 指纹计算方法 | 耗时(s) | 平均速度(工程/s) |
|---|---|---|
| MD 5 | 2 144.36 | 10.72 |
| 相似哈希-改进前 | 2 959.34 | 14.8 |
| 相似哈希-改进后 | 2 143.7 | 10.71 |

### 4.2 对研究问题2的分析

在 RQ1 的实际检测结果中,出现了大量相似文件对均为文件名相同的文件的情况.在 Mockus[27]关于开源软件中大规模代码复用的研究中,曾使用基于文件名的比较方法,以代码功能模块为检测的粒度进行实验,将文件名相同的文件均判定为相似,若两个文件夹下超过一定比例的文件名称相同,则将其视作一对克隆的功能模块对.为了验证能否简单地通过文件名匹配取得 RQ1 中相近的检测结果,仅使用文件名作为匹配标准对数据集进行检测,获取的结果见表 7.

**Table 7** Experimental results of RQ2
**表 7** RQ2 的实验结果

| 匹配工程总数 | 总工程对数 | 总匹配对数 | 精确度(%) |
|---|---|---|---|
| 200 | 19 900 | 849 757 | 12.65 |

事实证明,仅靠文件名匹配的相似文件对中会有大量的误报情况,因此在实际同源检测过程中,仅靠文件名匹配会报出大量不正确的结果,结果的精确度会随着库规模的扩大越来越低,因此在小规模库上或许可以通过文件名匹配来缩小检测范围,但在大规模数据集上,这种方法是不可行的.另外,对 RQ1 中的改进后相似哈希的文件匹配结果进行统计后可以发现,虽然大部分结果中文件名是相同的,但仍然有 17%的正确文件对文件名并不相同,这些相似文件对仅通过文件名匹配的方法是无法被发现的.

然而,通过对比文件名匹配的结果,改进后的相似哈希算法也暴露出一些漏报的问题.文件名匹配的方法精确度较低,仅为 12.65%,但实际匹配出的正确文件对数量达 107 512 对,而在相同数据集下改进后的相似哈希方法仅报出 40 082 对.为了分析本文方法产生漏报的主要原因,对漏报的文件随机抽取了其中 30 对进行人工核验统计与分析,分析结果如图 6 所示.由分析结果可见,漏报的出现主要分为以下几种情况.

(1) 行筛选后文件为空

由于在本文提出的方法中,对不同语言的文件所对应的常见行进行了筛选,因此会出现一些小文件中所有行都是常见行,筛选后文件的所有特征都被筛除的情况,约有 27%的漏报属于此类原因.此类文件中包含的语义信息通常较少,因此同源分析价值比较低,文件的漏报对结果的影响也较低.在未来的工作中,也可以对目前统计的不同语言的常见行进行筛选和取舍,以减少非必要的筛选对结果的影响.

(2) 两文件行数相差 1.5 倍及以上

此类情况约占漏报情况中的 17%,主要是由相似哈希的计算特点所致.相似哈希的原理是将所有特征转化为哈希值并累加起来得到最终的指纹结果,在本文的方法中,这些特征就是文件的每一行.当两个文件行数相差较大时,如一个文件的行数达到了另一个行数的 1.5 倍,那么即使两文件共同行数占据了小文件的绝大部分(若超过 70%,则按目前的标准将判定两文件为相似),大文件额外多出的行数仍会对大文件的指纹造成较大的额外影响,从而导致两者的指纹距离超过了阈值.因此,本方法不适用于行数差距过大的相似文件对的比较,对于这些文件可以考虑将指纹采集的对象从文件细化到方法和函数,以获得更加精准的匹配结果.

(3) 其他情况

最后,对不同海明距离的情况进行统计.可见,有将近一半的情况是距离大于 8 小于等于 15 的情况,这些情况下的文件可以通过对海明距离阈值的向上调整而被检测到,需要注意的是,上调阈值可能会伴随着精确度的降低.另一半的海明距离大于 15,无法通过少量提高阈值检测到这部分漏报文件.为了进一步分析漏报原因,对图 6 所示其他情况的文件对作进一步统计分析,结果如右侧饼状图所示.在其他情况中,44%的文件行数小于等于 50,由于小文件本身的特征较少,较小的修改就会导致文件整体指纹有较大波动,因此在小文件上大规模相似

哈希同源分析效果不佳,但小文件本身包含语义也较少,相似性分析的重要性较低.在剩下的文件对中,25%的文件通过改进前的方法可以检测,而改进后其距离超出阈值,其原因主要是常见行的筛选放大了剩余行差异对结果的影响和部分小文件经过行筛选后特征为 0,具体情况在第 4.1 节中已进行过详细阐述.

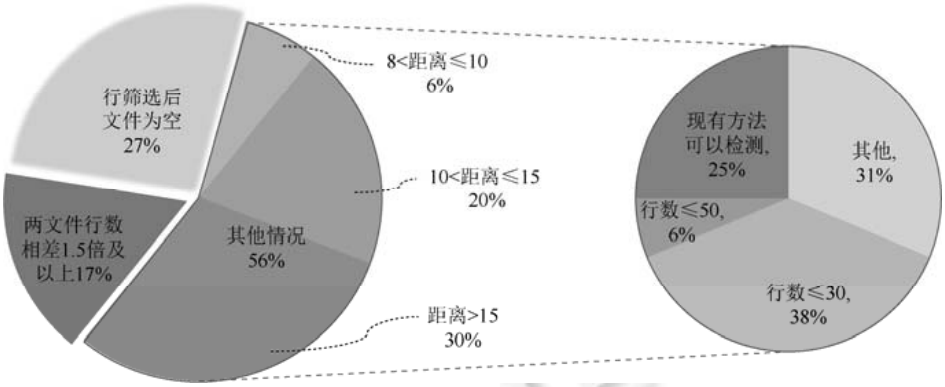


Fig.6 False negatives analysis of enhanced simhash method

图 6 改进后相似哈希算法漏报分析

综上所述,本文提出的相似哈希方法在精确度方面远远高于只简单地通过文件名进行匹配的方法,但是会出现一些漏报的文件对.漏报的情况可以在未来工作中通过调整海明距离阈值、优化各个语言行筛选范围等方法加以进一步改进.

### 4.3 对研究问题3的分析

在构建的 130 万项目的大规模库上进行实验,随机抽取来自 10 个覆盖本文支持的 10 种不同语言的开源工程,筛除非代码文件、行数小于 20 行或相似哈希特征为空的文件后剩余共 41 437 个代码文件作为输入的待测文件,海明距离阈值为 3.通过本文的方法对这些文件与库中文件进行相似性检测,获取的结果见表 8.表中第 1 列是所选开源项目的名称,第 2 列是所选项目版本的提交号,第 3 列是指项目包含的部分语言(排名不分先后).总文件数是指待测项目文件经过筛选后剩余的文件数量.有结果的文件数是指在本项目所有参与检测的文件中最终匹配到相似文件的文件数.总候选文件数是指通过本文建立的索引和分表策略筛选后,该项目所有文件获取到的候选文件序列的总文件数量.总实际匹配个数是指该项目所有候选文件与其对应项目文件进行海明距离计算后,海明距离在 3 以内的文件对个数,即相似匹配最终匹配到的结果个数.

可见,本文的索引方法大幅缩小了候选文件的规模,降低了最终需两两比较的文件对数量,候选文件数量与实际匹配数量差距不大.在大规模代码库上检索相似文件的平均速度为 0.43s/文件(仅考虑匹配到相似文件的文件检测速度),平均候选相似文件数为 124.78 个,平均匹配相似文件 94.38 个.

为了对比改进后的相似哈希算法与传统 MD 5 方法相比是否能够检测出更多的相似文件,选取通过 MD 5 能够匹配出相似文件的 13 886 对工程对,涉及 4 589 个工程、45 383 个版本.在这些工程对上采用改进后相似哈希方法进行匹配,结果见表 9.

可见,在大规模数据集下,改进后的相似哈希算法依旧能够比传统的 MD 5 指纹方法检测出两倍以上的相似文件对,基本上能够完全覆盖 MD 5 方法的检测结果.由于工程数量巨大,对全部检测结果进行验证需要耗费大量的时间和计算资源.为了验证本文所提方法在大规模检测时能否保持较高的精确度,从上述 13 886 对工程文件中随机抽取了 309 对工程对,涉及 124 个工程的 218 个版本进行精确度的检验.检验结果见表 10.

与完全无行筛选的行粒度相似哈希算法相比,改进前的相似哈希方法确实能够通过对纯符号行的筛选减少错误匹配结果数,提高精度值,表 2 中的行统计结果也显示各种语言中频繁出现的行中包含部分纯符号行.改进后的相似哈希算法筛除了更多的常见行,在精确度上远远高于改进前的相似哈希算法,且比起 MD 5 算法能

够发现更多的相似文件对.为了查看改进后相似哈希方法在不同海明距离下的表现,在不同海明距离阈值上进行实验,实验结果见表 11.改进后的相似哈希算法在海明距离阈值为 8 时已经能够达到 97.51%的精确度,而在阈值为 6 及以下的情况下甚至可以达到 99%以上,其整体的精确度都维持在比较高的水平.

**Table 8** Efficiency on large-scale database

**表 8** 大规模库上检测效率

| 工程名 | 提交号 | 编程语言 | 总文件数 | 有结果的文件数 | 总候选文件数 | 平均候选文件数 | 总实际匹配个数 | 平均实际匹配个数 | 总时间(s) | 平均时间(文件/s) |
|---|---|---|---|---|---|---|---|---|---|---|
| electron | eaa1804 | C++,JS, python | 897 | 537 | 6 627 | 12.34 | 1 827 | 3.4 | 193.9 | 0.36 |
| node | 2daf883 | JS,C++, python | 23 065 | 7 088 | 900 612 | 127.06 | 689 636 | 97.3 | 3 003.6 | 0.42 |
| numpy | 63ef78b | python, C,JS | 661 | 162 | 7 731 | 47.72 | 4 456 | 27.51 | 63.85 | 0.39 |
| bitcoin | 8830cb5 | C++,C python | 1 100 | 545 | 134 845 | 247.42 | 106 025 | 194.54 | 245.61 | 0.45 |
| redis | 8ea4bdd | C,ruby, C++ | 533 | 392 | 70 795 | 180.6 | 58 929 | 150.33 | 211.81 | 0.54 |
| ruby | 287bfb6 | ruby,C, C++ | 8 294 | 342 | 15 049 | 44.00 | 8 115 | 23.73 | 116.68 | 0.34 |
| container-diff | aae5709 | python, C,go | 1 029 | 723 | 1 032 341 | 1 427.86 | 867 737 | 1 200.19 | 454.53 | 0.63 |
| zoneminder | 98f7fb6 | PHP,C++,JS | 1 365 | 558 | 54 041 | 96.85 | 38 432 | 68.87 | 196.74 | 0.35 |
| githawk | 18fef6c | Swift, JS,C | 1 586 | 653 | 26 661 | 40.83 | 23 389 | 35.82 | 214.04 | 0.33 |
| postgre | b55413d | C,SQL, C++ | 2 907 | 1 634 | 93 091 | 56.97 | 42 575 | 26.06 | 589.30 | 0.36 |
| **共计** | | | **41 437** | **12 634** | **2 341 793** | **185.36** | **1 841 121** | **145.73** | **5 290.06** | **0.42** |

**Table 9** Experimental results of MD 5 and enhanced simhash on large-scale database

**表 9** MD 5 和改进后相似哈希方法在大规模库上的检测结果

| | 相似文件对 | 交集占总匹配对的比例(%) |
|---|---|---|
| MD 5 | 818 120 | 99.50 |
| 相似哈希-改进后 | 1 929 097 | 42.20 |

**Table 10** Experimental results of samples

**表 10** 抽样实验结果

| | 相似文件对 | 与 MD 5 的交集文件对数 | 错误匹配数 | 精确度(%) |
|---|---|---|---|---|
| MD 5 | 11 184 | – | 0 | 100 |
| 相似哈希-无行筛选 | 107 736 742 | 11 177 | 27 598 022 | 0.075 |
| 相似哈希-改进前 | 8 923 644 | 11 177 | 49 469 | 45.3 |
| 相似哈希-改进后 | 20 571 | 11 179 | 512 | 97.5 |

**Table 11** Experimental results of enhanced simhash under different Hamming distance thresholds

**表 11** 不同海明距离阈值下改进后方法的实验结果

| 海明距离(≤) | 检测文件对 | 错误数 | 精确度(%) |
|---|---|---|---|
| 0 | 12 300 | 55 | 99.55 |
| 1 | 12 964 | 89 | 99.31 |
| 2 | 13 728 | 107 | 99.22 |
| 3 | 14 608 | 107 | 99.27 |
| 4 | 15 605 | 107 | 99.31 |
| 5 | 16 652 | 129 | 99.23 |
| 6 | 17 701 | 147 | 99.17 |
| 7 | 19 136 | 327 | 98.29 |
| 8 | 20 571 | 512 | 97.51 |

在本次实验中,改进前的相似哈希精确度达到了 45.3%,比 RQ1 中检测出的结果要高很多,为了解释这一现象,对两次实验的结果作了进一步分析.改进前的相似哈希方法主要产生误报的原因是大量的文件中出现了行覆盖现象,例如“returnfalse;”在一个文件中多次出现,导致最终文件的指纹值直接等于这一行的哈希结果,其他

行对结果的影响被覆盖.因此,假如有 $n$ 个文件内部多次出现“returnfalse;”,则这些文件两两之间均会被识别为相似文件对,总计 $n\times(n-1)/2$ 对假克隆对.误报文件对个数与出现行覆盖的文件个数的平方成正比,因此数据规模越大,越容易出现大规模的误报,精确度也随之越低.而本文通过常见行筛选改进后的相似哈希方法则能够很好地解决这一问题,在大规模数据集上也能够保持较优的效果.

在本文的实验中,测试了不同海明距离下相似哈希算法结果的精确度,整体趋势均为随着海明距离阈值的逐渐增加,精确度逐渐降低.在两个文件指纹值进行比较的过程中,海明距离越大,意味着两个指纹不同位数越多,两个文件也就越不相似.当阈值较小时,只有非常相近的指纹才被视作相似,因此精确度比较高,但是会将一些海明距离较大的相似文件对遗漏,导致漏报增多;当阈值逐渐增大、相似性判断的条件更加宽松时,更多的文件会被判断为相似,因此可能出现一些本来不相关的文件被判断成相似的情况,导致精确度降低.在不同的实际应用场景下,对误报、漏报的容忍度各不相同,需要对误报和漏报加以权衡,结合具体需求选择合适的海明距离阈值.

## 5 总结与展望

目前,日益庞大的软件系统规模和高速增长的开源代码数量对代码克隆检测工具的高效性和可扩展性带来了更高的挑战.对软件代码的相似性分析可以帮助开发者识别和管理软件系统中的开源代码组件,提高软件质量,降低开发和维护的成本.本文对基于行粒度相似哈希的代码相似性检测算法进行了实验验证和分析,通过分语言常见行筛选的方法对出现的行覆盖现象进行了优化.本文提出的改进后的相似性检测方法将源代码文件降维成一个 64 位二进制字符串,能够实现代码特征的快速生成以及匹配,不需要额外的基于词法、语法的代码分析工具辅助,且在不同语言上的迁移成本极低,可移植性强.

通过一系列的实验,对比了改进前后的相似哈希方法、普通 MD 5 方法以及仅靠文件名匹配的方法在相似代码文件检测上的效果.无论是改进前还是改进后的相似哈希算法,其匹配结果都能够基本上完全覆盖 MD 5 的匹配结果,改进后的相似哈希方法在各个海明距离[16]阈值上都能保持 99%以上的精确度,而改进前的方法在最优情况下也不足 4%,改进效果非常明显.仅靠文件名匹配的相似文件对的方法,其精确度仅为 12.65%,因此在实际同源检测过程中,仅靠文件名匹配会误报出大量不正确的结果.在组建的 130 万个开源工程、3.86 亿个项目文件的大规模代码指纹库上进行文件检测,平均单文件检测时间为 0.43s,通过索引和分表大幅度降低了候选文件数量.从库中选取通过 MD 5 能够匹配出相似文件的 13 886 对工程对进行实验分析,证明了改进后的相似哈希方法在大规模数据集上仍能够保持较高的精确度,较改进前方法在精确度上有大幅度提升,较 MD 5 方法能够获取将近两倍的相似文件对结果.

在实验过程中也暴露出目前方法的一些漏报现象,主要分为下几种情况:文件本身较小或包含的语义信息较少;两文件总行数差距较大,导致指纹的差异较大;某些小文件经过常见行筛选后文件特征为空;常见行的筛选一定程度上放大了其他行的差异对结果指纹值的影响,导致海明距离超出阈值.虽然本文仅在由 Github 开源项目组成的两个数据集上进行了实验(分别为 RQ1、RQ2 中 200 个项目的数据集和 RQ3 中 130 万个开源项目组成的大规模库),但是这些项目均来自于 Github 上星数排名的前列,具有应用范围广泛、影响项目众多的特点.本文中提及的许多算法适用的场景,如漏洞传播发现、抄袭检测等都以这些项目作为研究重点,因此本文的数据集具有一定的代表性.在数据规模上达到了 3.8 亿这一量级,能够覆盖到多种语言和不同结构的实际项目,得出的实验结果较为可靠.

在未来的工作中,我们将综合考虑实验中分析出的各种漏报情况对本文的相似哈希方法作进一步的改进.具体来说,包括:通过启发式分析对当前通过统计得到的各种行作进一步取舍,保证更新后的行筛选器只筛除对代码文件语义影响不大的行,而不过多筛除一些有用行以致剩余行差异被放大;对相似哈希方法在小文件上的检测能力作进一步的实验和优化;在海明距离阈值方面,在更大范围内对精确度和漏报率作评估和权衡.

**References**:


[1] Novak M, Joy M, Kermek D. Source-code similarity detection and detection tools used in academia: A systematic review. ACM Trans. on Computing Education (TOCE), 2019,19(3):1–37.

[2] Zhu C, Tang Y, Wang Q, Li M. Enhancing code similarity analysis for effective vulnerability detection. In: Proc. of the 2nd Int'l Conf. on Computer Science and Software Engineering. 2019. 153–158.

[3] Jones KS. A statistical interpretation of term specificity and its application in retrieval. Journal of Documentation, 1972, 11–21.

[4] Ottenstein KJ. An algorithmic approach to the detection and prevention of plagiarism. ACM SIGCSE Bulletin, 1976,8(4):30–41.

[5] Halstead MH. Elements of Software Science. New York: Elsevier, 1977.

[6] Roy CK, Cordy JR. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: Proc. of the 16th IEEE Int'l Conf. on Program Comprehension. 2008. 172–181.

[7] Kamiya T, Kusumoto S, Inoue K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. on Software Engineering, 2002,28(7):654–670.

[8] Wang P, Svajlenko J, Wu Y, Xu Y, Roy CK. CCAligner: A token based large-gap clone detector. In: Proc. of the 40th Int'l Conf. on Software Engineering. 2018. 1066–1077.

[9] Sajnani H, Saini V, Svajlenko J, Roy CK, Lopes CV. SourcererCC: Scaling code clone detection to big-code. In: Proc. of the 38th Int'l Conf. on Software Engineering. 2016. 1157–1168.

[10] Baxter ID, Yahin A, Moura L, Sant'Anna M, Bier L. Clone detection using abstract syntax trees. In: Proc. of the Int'l Conf. on Software Maintenance. 1998. 368–377.

[11] Krinke J. Identifying similar code with program dependence graphs. In: Proc. of the 8th Working Conf. on Reverse Engineering. 2001. 301–309.

[12] Chen QY, Li SP, Yan M, Xia X. Code clone detection: A literature review. Ruan Jian Xue Bao/Journal of Software, 2019,30(4): 962–980 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5711.htm [doi: 10.13328/j.cnki.jos.005711]

[13] Kim S, Woo S, Lee H, Oh H. Vuddy: A scalable approach for vulnerable code clone discovery. In: Proc. of the IEEE Symp. on Security and Privacy (SP). 2017. 595–614.

[14] Gionis A, Indyk P, Motwani R. Similarity search in high dimensions via hashing. VLDB, 1999,99(6):518–529.

[15] Charikar MS. Similarity estimation techniques from rounding algorithms. In: Proc. of the 34th Annual ACM Symp. on Theory of Computing. 2002. 380–388.

[16] Hamming RW. Error detecting and error correcting codes. The Bell System Technical Journal, 1950,29(2):147–160.

[17] Manku GS, Jain A, Das Sarma A. Detecting near-duplicates for Web crawling. In: Proc. of the 16th Int'l Conf. on World Wide Web. 2007. 141–150.

[18] Xu Y, Qi L, Dou W, Yu J. Privacy-preserving and scalable service recommendation based on simhash in a distributed cloud environment. In: Proc. of the Complexity 2017. 2017. 1–9.

[19] Rezaeian N, Novikova GM. Detecting near-duplicates in Russian documents through using fingerprint algorithm simhash. Procedia Computer Ence, 2017,103:421–425.

[20] Wang Y, Cai J, Meng C, Liu Z, Xue J. Android malware detection based on multi-feature fusion. Journal of Cyber Security, 2018, 3(4):54–62.

[21] Uddin MS, Roy CK, Schneider KA, Hindle A. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In: Proc. of the 18th Working Conf. on Reverse Engineering. 2011. 13–22.

[22] Uddin MS, Roy CK, Schneider KA. SimCad: An extensible and faster clone detection tool for large scale software systems. In: Proc. of the 21st Int'l Conf. on Program Comprehension (ICPC). 2013. 236–238.

[23] Qiao YC, Yun XC, Tuo YP, Zhang YZ. Fast reused code tracing method based on simhash and inverted index. Journal on Communications, 2016,37(11):104–113.

[24] Guo Y, Chen FH, Zhou MH. Code clone detection method for large-scale source code. Journal of Frontiers of Computer Science and Technology, 2014,8(4):417–426.

[25] Appleby A. Murmurhash 2.0. 2008. http://code.google.com/p/smhasher/wiki/MurmurHash2

[26] Svajlenko J, Roy CK. Evaluating clone detection tools with bigclonebench. In: Proc. of the 2015 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE, 2015. 131–140.

[27] Mockus A. Large-scale code reuse in open source software. In: Proc. of the 1st Int'l Workshop on Emerging Trends in FLOSS Research and Development (FLOSS 2007: ICSE Workshops 2007). IEEE, 2007. 7.

**附中文参考文献:**

[12] 陈秋远,李善平,鄢萌,夏鑫.代码克隆检测研究进展.软件学报,2019,30(4):962–980. http://www.jos.org.cn/1000-9825/5711.htm [doi: 10.13328/j.cnki.jos.005711]

[20] 王勇,蔡建宇,孟春,刘振岩,薛静锋.基于多特征融合的安卓恶意应用程序检测方法.信息安全学报,2018,3(4):54–62.

[23] 乔延臣,云晓春,庹宇鹏,张永铮.基于 simhash 与倒排索引的复用代码快速溯源方法.通信学报,2016,37(11):104–113.

[24] 郭颖,陈峰宏,周明辉.大规模代码克隆的检测方法.计算机科学与探索,2014,8(4):417–426.

**李玫**(1997－),女,硕士,主要研究领域为软件工程.

**张世琨**(1969－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为软件工程,网络安全,知识计算.

**高庆**(1989－),男,博士,助理研究员,主要研究领域为软件工程.

**胡文蕙**(1977－),女,博士,副研究员,主要研究领域为软件工程,大数据技术,知识图谱.

**马森**(1980－),男,博士,副研究员,主要研究领域为软件安全.

**张兴明**(1963－),男,教授,主要研究领域为网络安全.