中国科学 信息科学
SCIENTIA SINICA Informationis

Static duplicate bug-report identification for compilers

陈俊洁, 胡文翔, 郝丹, 熊英飞, 张洪宇 and 张路

Citation: 中国科学: 信息科学 **49**, 1283 (2019); doi: 10.1360/N112019-00001

View online: http://engine.scichina.com/doi/10.1360/N112019-00001

View Table of Contents: http://engine.scichina.com/publisher/scp/journal/SSI/49/10

Published by the 《中国科学》杂志社

中国科学: 信息科学 2019 年 第 49 卷 第 10 期: 1283–1298
SCIENTIA SINICA Informationis
软件自动化前沿进展专题 • 论文

一种静态的编译器重复缺陷报告识别方法

陈俊洁[1†], 胡文翔[2], 郝丹[2*], 熊英飞[2], 张洪宇[3], 张路[2]

1. 天津大学智能与计算学部, 天津 300350, 中国
2. 北京大学信息科学技术学院, 北京 100871, 中国
3. School of Electrical Engineering and Computing, The University of Newcastle, Callaghan 2308, Australia
* 通信作者. E-mail: haodan@pku.edu.cn
† 该论文主要工作为该作者在北京大学期间完成

收稿日期: 2019–03–03; 修回日期: 2019–05–31; 接受日期: 2019–09–03; 网络出版日期: 2019–10–16

国家重点技术研发计划 (批准号: 2017YFB1001803) 和国家自然科学基金 (批准号: 61672047, 61872008, 61861130363, 61922003, 61828201) 资助项目

摘要 编译器缺陷报告在编译器质量保证中具有重要作用, 而重复缺陷报告往往带来不必要的人力、时间等资源浪费. 为了识别编译器重复缺陷报告, 本文提出了一种静态的重复缺陷报告识别方法 IdenDup. 该方法可以有效解决两个场景下的重复缺陷报告问题, 即模糊测试 (fuzz testing) 所产生的缺陷报告和缺陷管理系统中不同来源的缺陷报告. 具体来说, IdenDup 利用缺陷报告中静态文本和程序特征来识别重复缺陷报告, 其中程序特征包括程序词法、语法, 以及本文首次提出的数据流特征. 特别地, 程序数据流特征指的是程序中变量使用路径 (变量使用方式及使用方式的顺序) 特征. 之后, 我们使用 C 语言的两个主流编译器 GCC 和 LLVM 作为实验对象, 对 IdenDup 的效果进行了实验探究. 实验结果表明, IdenDup 可以有效地识别上述两个场景下的重复缺陷报告, 并且超过已有方法.

关键词 编译器调试, 编译器缺陷报告, 重复缺陷报告, 数据流分析, 静态方法

1 引言

编译器作为编译其他软件系统的核心工具, 一直被认为是十分重要的基础软件设施. 每一个运行在计算机上的程序, 比如复杂的操作系统以及终端用户自己编写的简单的脚本程序, 几乎都需要经过编译器的处理. 由于其重要性和基础性, 编译器一直被广泛使用. 一些流行程序语言所对应的编译器, 如 C/C++ 语言的编译器 GCC, 往往被数以百万计的用户所直接使用, 其中主要是程序开发人员. 除了编译器的直接用户以外, 还有更多用户间接地依赖编译器, 即他们使用着各种各样的经过编译器编译后的应用程序.

引用格式: 陈俊洁, 胡文翔, 郝丹, 等. 一种静态的编译器重复缺陷报告识别方法. 中国科学: 信息科学, 2019, 49: 1283–1298, doi: 10.1360/N112019-00001
Chen J J, Hu W X, Hao D, et al. Static duplicate bug-report identification for compilers (in Chinese). Sci Sin Inform, 2019, 49: 1283–1298, doi: 10.1360/N112019-00001

尽管编译器质量已经被广泛关注和研究, 但是与其他软件相似, 编译器中仍旧存在缺陷. 编译器的缺陷会导致正确的源代码被翻译成不正确的二进制代码, 从而导致基于该编译器编译的应用程序出现不符合预期甚至十分危险的行为. 比如, Java7 编译器中的一个缺陷引起了多个流行的 Apache 项目崩溃[1)]. 除此之外, 编译器的缺陷也会使软件调试过程变得十分困难. 其原因是开发人员很难确定一个软件的失败是由于他们所开发的软件存在缺陷, 还是他们所使用的编译器中存在缺陷. 比如, 当一个含有缺陷的编译器将一个正确的程序转换为一个不正确的可执行程序, 对于程序开发者来说, 他们通常会假设这些错误行为是由于自己所开发的软件含有缺陷导致的. 因此, 他们会花费很长时间来寻找自己软件中的问题, 但最终却发现编译器的缺陷才是导致该软件失败的根本原因.

由于编译器的重要性以及编译器缺陷的严重性, 编译器质量保证是十分必要的工作. 而在编译器质量保证过程中, 无论测试还是调试, 都会涉及到编译器缺陷报告, 即描述某一缺陷的报告. 比如, 编译器测试人员会对检测到的缺陷创建并提交缺陷报告, 同时开发人员会根据所收到的缺陷报告进行缺陷诊断及修复. 也就是说, 编译器缺陷报告在编译器质量保证过程中扮演着重要角色. 但是, 重复缺陷报告问题, 即不同的缺陷报告对应着同一个编译器缺陷, 对该过程有着严重的影响. 具体地, 该影响可以分为两个主要方面: 一方面, 编译器的测试往往是模糊测试 (fuzz testing), 即运行大量的测试程序来检测编译器缺陷. 这种方式通常会产生很多失败的测试程序. 测试人员需要对每一个失败的测试程序创建并提交相应的缺陷报告, 以便开发人员根据缺陷报告进行缺陷诊断及修复. 然而, 这些失败的测试程序中有很多是由相同的编译器缺陷所导致的. 如果测试人员对每一个缺陷都创建并提交缺陷报告, 则会产生大量的重复缺陷报告. 这不仅浪费了测试人员创建缺陷报告的努力, 也会造成后续开发人员诊断缺陷时的资源浪费. 另一方面, 缺陷报告不仅可以由测试人员提交, 也可以由任何用户提交. 这些不同来源的缺陷报告汇聚到缺陷管理系统中, 也会导致重复缺陷报告的问题, 从而造成开发人员诊断缺陷时的资源浪费. 这两个方面分别对应着缺陷报告在不同场景下的重复问题, 即模糊测试场景和缺陷管理系统场景. 值得注意的是, 识别模糊测试场景下的重复缺陷报告其实是发生在创建缺陷报告之前. 也就是说, 此时还不存在完整创建好的缺陷报告, 但是组成缺陷报告的元素 (比如触发缺陷的测试程序、触发缺陷的编译器配置等) 已经存在. 为了方便起见, 我们仍将其称为缺陷报告. 如果我们能够处理好这两个场景下的重复缺陷报告问题, 那么将大大地节约编译器缺陷调试过程中的人力、时间等资源. 因此, 编译器重复缺陷报告识别是一项十分重要的研究.

目前, 许多识别软件重复缺陷报告的方法已经被提出. 这些方法主要对缺陷报告中的文本信息进行分析, 利用信息检索技术来识别软件重复缺陷报告 [1~3]. 然而, 这些方法无法应用于编译器的重复缺陷报告识别上. 其原因是, 编译器缺陷报告中最主要的一类信息是触发缺陷的测试程序, 而一般软件缺陷报告不包含此类信息, 也就是说, 已有方法无法有效地分析该类程序信息. 针对编译器缺陷报告, Chen 等 [4] 提出一种识别模糊测试场景下的编译器重复缺陷报告方法. 该方法首先提取与缺陷相关的信息, 如测试程序动态执行时对编译器的覆盖信息、指令执行信息等, 然后基于这些信息计算不同缺陷报告之间的距离, 从而对缺陷报告进行排序, 使得排在前边的缺陷报告更有可能对应着不同的编译器缺陷. 尽管该方法可以有效识别编译器崩溃缺陷的重复报告, 但是对于编译器结果错误缺陷 (即编译之后得到不正确的结果), 仍无法有效处理. 与此同时, 该方法无法应用于缺陷管理系统场景下的重复缺陷报告问题. 其原因是该场景下的缺陷报告不包含该方法所需的测试过程中的动态信息 (如覆盖信息以及指令执行信息等). 因此, 也就是说, 目前没有方法可以有效地识别上述两个场景下的编译器重复缺陷报告.

为了解决该问题, 本文提出一种静态编译器重复缺陷报告识别方法 IdenDup, 即利用缺陷报告中

1) http://blog.thetaphi.de/2011/07/real-story-behind-java-7-ga-bugs.html.

静态的文本和程序信息来识别重复缺陷报告. 由于 IdenDup 仅需要缺陷报告中的静态信息, 因此可以应用于上述两个场景. 具体来说, IdenDup 所使用的静态文本信息包括缺陷报告中的触发缺陷的编译配置信息、编译器输出信息等; 所使用的程序静态信息包括 3 个层次的特征: 程序词法特征、语法特征, 以及本文首次提出的数据流特征 (即通过数据流分析来挖掘变量的使用路径特征). 针对数据流特征, 本文提出一种基于数据流分析的变量使用路径提取算法, 用来构造变量在过程内的使用方式以及各种使用方式之间的顺序关系的有向无环图. 本文将此图称为变量使用路径图. 类似于自然语言处理中的 N-gram 模型, IdenDup 将该图中所有长度等于 N 的路径作为该类特征. 在获取上述静态特征之后, IdenDup 将每一个缺陷报告转换为一个空间向量, 然后使用权重计算策略 (包括 TF-IDF[5] 和 BM25[6]) 对向量进行处理. 最后, IdenDup 利用基于向量空间距离 (包括余弦距离、欧几里得 (Euclidean) 距离、曼哈顿 (Manhattan) 距离) 的相似度度量策略, 计算向量之间的相似度, 从而识别重复缺陷报告.

为了验证 IdenDup 效果, 本文使用两个主流的 C 语言编译器 GCC 和 LLVM 作为实验对象进行探究. 同时, 本文对两个场景下的编译器重复缺陷报告识别均进行了实验. 针对模糊测试场景的重复缺陷报告识别, 我们使用 Chen 等[4] 研究中所使用的数据集, 即 GCC-4.3.0 的模糊测试数据, 包括 1275 个触发编译器结果错误缺陷的测试程序, 其共触发 35 个不同的编译器结果错误缺陷. 在该数据集上, IdenDup 比 Chen 等[4] 所提出的方法准确性提升了 13%. 针对缺陷管理系统场景, 我们在 GCC 和 LLVM 的缺陷管理系统中分别收集了 330 个 GCC 缺陷报告和 257 个 LLVM 缺陷报告作为数据集. 由于没有已有工作解决该场景下的编译器重复缺陷报告识别问题, 我们选择随机方法作为基准. 实验结果表明, IdenDup 的效果显著优于基准方法的效果.

本文的贡献主要包括以下 3 个方面:

- 提出一种新的针对编译器测试程序的数据流特征, 即变量使用路径特征, 其考虑变量在过程内的使用方式以及使用方式之间的顺序关系;
- 提出一种静态的编译器重复缺陷报告识别方法 IdenDup, 其可以用于解决两个场景下的重复缺陷报告问题;
- 在两个主流的 C 语言编译器 GCC 和 LLVM 上进行实验来探究 IdenDup 的效果, 且实验结果表明 IdenDup 在两个场景下均能够有效识别编译器重复缺陷报告, 并超过已有方法.

2 相关工作

2.1 重复缺陷报告识别

Chen 等[4] 首次尝试解决编译器重复缺陷报告问题, 并提出针对模糊测试场景的识别方法. 该方法基于测试中的各种信息, 如动态覆盖信息及指令执行信息等, 计算缺陷报告之间的相似度, 并使用最远点优先的方法对缺陷报告进行排序, 使得更有可能对应着不同缺陷的缺陷报告排在前面. 虽然该方法可以有效地识别编译器崩溃缺陷的重复报告, 但是针对编译结果错误缺陷的重复报告, 该方法识别效果不佳. 与此同时, 该方法无法应用到缺陷管理系统场景下的重复缺陷报告识别. 其原因是缺陷管理系统场景下的缺陷报告不包含测试中的动态信息.

除了针对编译器的重复缺陷报告检测, 许多工作提出了各种各样的方法来识别一般软件中的重复缺陷报告. 其中很多已有工作基于缺陷报告文本信息, 利用信息检索领域的相似文档识别方法来解决该问题[2,3,7]. 比如, Sun 等[1] 提出引入判别模型对缺陷报告中的每一个单词赋予不同的权重, 进而计算缺陷报告的文本相似度. 他们通过实验证明他们的方法超过了之前的简单使用文本信息的方法. 之

后, Sun 等 [8] 引入 BM25F [9] 来计算文本相似度, 从而进一步改善效果. Nguyen 等 [2] 提出应用主题模型来度量单词之间的语义相似性来帮助识别重复缺陷报告. 此外, 除了文本信息之外, Wang 等 [10] 首次引入了执行信息来识别重复缺陷报告. Lerch 等 [11] 提取缺陷报告中的堆栈轨迹, 将其转化为一组函数, 进而计算缺陷报告之间的相似度. 然而, 这些已有的方法无法有效识别编译器的重复缺陷报告. 其原因是, 编译器缺陷报告中最核心的部分是触发缺陷的测试程序, 而一般软件的缺陷报告不包含此类信息, 也就是说, 这些已有方法无法对测试程序信息进行有效分析.

2.2 编译器调试与测试

编译器缺陷报告识别是编译器调试过程中的重要一步. 除此之外, 还有许多工作解决编译器调试领域相关问题. 比如, Regehr 等 [12] 提出 CReduce 工具, 将触发 C 语言编译器缺陷的测试程序进行化简, 即将该程序化简为最小的程序但仍能触发该编译器缺陷. Pflanzer 等 [13] 进一步将 CReduce 拓展到化简 OpenCL 编译器的测试程序. 上述两个方法都是特定于某种语言的编译器的程序化简工具. 之后, Herfert 等 [14] 提出了一种通用的程序化简方法. 该方法能够化简所有树形结构的测试输入, 例如 Python 和 JavaScript. 此外, 还有一些编译器调试的工作是关于提供调试时所需要的信息或者调试可视化. 比如, Sloane [15] 提出针对 Eli 生成的编译器的图形化调试器, 即对该编译器处理的测试输入的执行提供了图形化界面.

编译器缺陷报告作为编译器质量保证过程中的重要方面, 也与编译器测试相关. 当前, 很多工作也是关于编译器测试的, 主要可以分为编译器测试程序生成与测试预言构造. 比如, Yang 等 [16] 提出了针对 C 语言编译器的测试程序自动生成工具 Csmith, 这也是当前最广泛使用的 C 程序自动生成工具. Lidbury 等 [17] 基于 Csmith 进一步提出了针对 OpenCL 编译器的自动程序生成工具 CLsmith. 编译器测试中的测试预言主要包括两类, 随机差异测试 [18] 和等价取模测试 [19]. 前者是对比一个测试程序在多个基于相同规约实现的编译器下的输出结果, 来判断是否触发编译器缺陷; 后者是将一个测试程序变异成一个在一组输入下等价的程序, 然后对比这一对等价程序在同一个编译器下的输出结果, 来判断是否触发编译器缺陷. Chen 等 [20] 进行了实证性研究来比较各种编译器测试技术, 并分析了影响编译器测试效果的因素. 除此之外, Chen 等 [21~23] 提出预测测试程序的揭错概率的方法, 并基于此来加速编译器测试. 之后, Chen 等 [23,24] 通过提出静态预测测试覆盖信息的方法来进一步加速编译器测试.

3 方法介绍

本文提出了一种静态编译器重复缺陷报告识别方法 IdenDup, 来解决两个场景下的编译器重复缺陷报告问题, 即识别模糊测试场景和缺陷管理系统场景下的重复缺陷报告. 该方法是第 1 种能够有效解决两个场景下的编译器重复缺陷报告问题的方法. 具体来说, IdenDup 分为 3 个步骤: 特征提取、特征处理、缺陷报告相似度度量, 其流程图如图 1 所示. 下面将详细介绍这 3 个步骤.

3.1 特征提取

IdenDup 利用缺陷报告中的静态信息来识别编译器重复缺陷报告, 具体包括文本特征和程序特征. 对于前述两个场景, 这两类静态信息都是存在的, 因此该方法可以用于解决该两个场景下的重复缺陷报告问题.

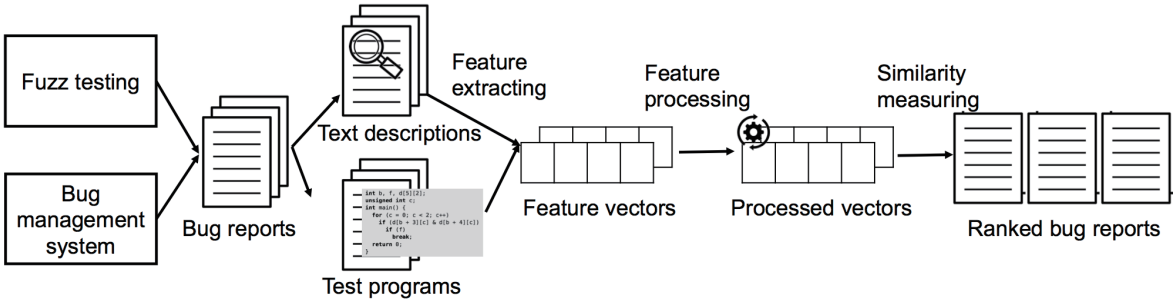


图 1 IdenDup 流程图

Figure 1 Overview of IdenDup

3.1.1 文本特征提取

IdenDup 使用了如下文本特征: 触发缺陷时的编译器配置、触发缺陷时的编译器输出、缺陷报告的标题及关键字. 前两类文本信息在两个场景下的缺陷报告中均存在, 最后一类文本信息仅存在于缺陷管理系统场景下的缺陷报告. 对于这些文本信息, IdenDup 采用自然语言处理中的常用方式进行处理, 具体分为如下 3 步:

- **分词:** 由于缺陷报告中的文本描述不仅包含单词, 还涉及到标点符号、空格等各种字符, 因此, IdenDup 首先移除缺陷报告文本中非单词的部分, 从而生成仅仅包含纯单词的文本信息.
- **抽取词干:** 由于单词会有不同的时态、单复数等形式, 因此, 为了更加精确地识别相同的单词, IdenDup 将单词转换为词干的形式. 例如: 将 “was” 和 “is” 均转换为 “be”.
- **去除停用词:** 由于英文文本中经常含有很多停用词, 如 “the” 和 “a” 等, 且这些停用词无法提供有用信息, 因此, IdenDup 根据词表去除停用词.

IdenDup 根据处理之后的文本信息, 将缺陷报告中的文本特征转化为向量表示形式. 具体来说, IdenDup 根据文本信息中的单词以及单词的数量对缺陷报告文本特征进行向量化. 在这里, 向量的每一维代表着一个单词所出现的次数, 即词频.

3.1.2 程序特征提取

与一般软件缺陷报告不同, 编译器缺陷报告中最主要的信息是触发缺陷的测试程序, 且该信息也是被许多编译器 (如 LLVM[2)]) 的缺陷报告提交标准明确规定要包含的. 编译器的测试程序在一定程度上可以有效反映编译器缺陷 [21, 22]. 其原因是测试程序中的某些元素、结构, 以及它们之间的各种组合及其使用方式是导致编译器缺陷被触发的关键. 因此, 程序特征的提取对编译器重复缺陷报告的识别十分重要. 与自然语言文本特征不同, 程序具有结构化、程序语义等特性. 在这里, IdenDup 根据程序的特性, 提取 3 个层次的程序特征, 包括程序词法特征、语法特征, 以及本文首次提出的数据流特征.

程序词法特征提取. 类比于自然语言, 程序语言也有其词法单元, 如关键词、运算符、常量等. 程序的词法特征在一定程度上是与编译器缺陷相关联的, 比如编译器 for 循环优化相关的缺陷, 只有当测试程序含有 for 关键词时, 才有可能触发 for 循环优化缺陷. 因此, IdenDup 提取程序的词法特征来帮助识别编译器重复缺陷报告.

为了提取程序词法特征, IdenDup 首先定义了程序词法特征的全集, 包括关键词、用户自定义标识符、常量值、类型、运算符等. 特别地, 本文实现 IdenDup 是以 *C* 语言的编译器为实验对象, 所分

2) https://llvm.org/docs/HowToSubmitABug.html.

$\mathrm{DATA}_{\mathrm{entry}}=I$
$\forall n \in (N-\mathrm{entry}): \mathrm{DATA}_n \leftarrow \top$
$\mathrm{ToVisit} \leftarrow N-\mathrm{entry}$
While (ToVisit.size > 0){
  $n \leftarrow$ getNode(ToVisit)
  $\mathrm{ToVisit} -= n$
  $\mathrm{MEET}_n \leftarrow \sqcap_{w \in \mathrm{pred}(n)} \mathrm{DATA}_w$
  If ($\mathrm{DATA}_n \neq f_n(\mathrm{MEET}_n)$) $\mathrm{ToVisit} \cup= \mathrm{succ}(n)$
  $\mathrm{DATA}_n \leftarrow f_n(\mathrm{MEET}_n)$
}

图 2　数据流分析单调框架实现算法

Figure 2　Dataflow analysis monotonous framework

析的程序为 C 语言程序. 因此, IdenDup 使用了 Clang Preprocessor 内部的所有 C 语言词法单元类型作为词法特征, 共 330 种. 根据所定义的词法特征全集, IdenDup 进而对每一个词法特征进行数量统计, 从而形成相应的词法特征向量.

程序语法特征提取. 词法特征可以通过程序元素的存在性在一定程度上反映与编译器缺陷的关联, 但是, 程序并不是简单地由词法特征组成的文本, 而是具有其特殊的语法特性. 因此, IdenDup 提取程序的语法特征来进一步帮助识别编译器重复缺陷报告.

相似地, 为了提取程序语法特征, IdenDup 首先定义程序语法特征的全集. 抽象语法树是一种常用的表示程序语法特征的方式, 其为程序的一种树形表示方式. 程序语法树上的节点表示程序语法特征. 在本文的实现中, IdenDup 使用了基于 Clang 所生成的抽象语法树上的所有节点类型作为语法特征, 共 125 种. 根据所定义的语法特征全集, IdenDup 统计每一个语法特征在抽象语法树上出现的次数, 从而形成相应的语法特征向量.

程序数据流特征提取. 除了程序词法特征与语法特征外, 通过对触发编译器缺陷的测试程序的分析, 我们发现变量的使用路径也可以较为准确地反映一个程序触发编译器缺陷的原因. 因此, IdenDup 通过数据流分析的方法提取变量的使用路径信息, 来帮助识别编译器重复缺陷报告. 本文将这一类特征叫做程序数据流特征, 且该类特征及其提取方法是由本文首次提出.

针对该类程序特征的提取, 本文提出了一种基于数据流分析的变量使用路径提取算法, 其本质上是构造变量在过程内的使用方式以及使用方式之间的顺序关系的有向无环图. 本文将此图称作变量使用路径图. 在这里, 本文提出的算法是基于数据流分析单调框架进行设计的. 具体来说, 数据流分析单调框架所需的元素包括控制流图、一个有限高度的半格、初值 I, 以及一组节点转换函数 f [25]. 其中, 半格是一个二元组 (S, MEET), S 是一个集合, MEET 是一个交汇操作. 图 2 描述了数据流分析单调框架实现算法, 其中 $n \in S$.

基于上述数据流分析单调框架, 我们设计了针对变量使用路径提取的数据流分析算法. 该算法在程序的中间代码 (static single assignment, SSA) 级别进行, 并将其对应的控制流图作为输入. 由于程序可能存在循环结构, 所以控制流图上可能存在回边. 该算法在处理时会将回边移除, 目的是为了避免迭代过多, 从而提升算法效率. 与此同时, 该算法将所有出度为 0 的节点合并, 即创建一个内容为空的基本块并将所有的出度为 0 的节点用边连接到该点. 该基本块记为 final(CFG). 在该算法中, 所使用的控制流图为 SSA 形式的控制流图. 根据数据流分析单调框架所需元素, 对于任意基本块 bb 以及

变量 v (bb, v 属于集合 S), 该算法中的元素定义如下:

- **初始值:** $I = \mathrm{Graph}_{v,\mathrm{init}}$ 仅仅包含变量 v 的定义节点, 其为算法进入代码的入口基本块 init 时的变量使用路径图. 在算法执行过程中, 变量使用路径图不断更新, 其中节点转换操作以及交汇操作分别在该图上创建或者修改点与边.
- **节点转换函数:** $f_{v,\mathrm{bb}}(L) = \mathrm{Connect}(L, \mathrm{Gen}_{v,\mathrm{bb}})$, 其中 $\mathrm{Gen}_{v,\mathrm{bb}}$ 为变量 v 在基本块 bb 中所有被使用的语句的链表, $\mathrm{Connect}(A, B)$ 是将 A 的末尾节点跟 B 的首节点连接. 在该算法中, 根据交汇操作可知, 尽管 A 为有向图, 但是 A 的出度为 0 的节点是唯一的, 即末尾节点; B 为链表, 其首节点容易获取. 当 B 为空链表时, 此时相当于将图 A 作为返回值; 而当 A 为空时, 则将 B 作为返回值.
- **交汇操作:** $\mathrm{MEET}_{v,\mathrm{bb}} = \mathrm{Merge}_{\mathrm{all}}(\mathrm{Graph}_{v,p|p\in\mathrm{pre(bb)}})$, 其中 Merge_A 是用于创建一个不含数据的节点 u, 并为 A 集合中所有图的末尾节点添加指向该节点 u 的边, pre(bb) 表示基本块 bb 的前驱. 当基本块 bb 的前驱中不存在变量 v 被使用的情形时, 则不需要添加边.

由于程序控制流图已经进行了去除回边的处理, 因此, 每个节点仅仅更新一次. 节点更新操作为 $\mathrm{Graph}_{v,\mathrm{bb}} = f_{v,\mathrm{bb}}(\mathrm{MEET}_{v,\mathrm{bb}})$. 根据上述算法定义, 该算法的主要流程为: 当过程内存在未被访问的基本块, 则更新该基本块对应的节点. 在更新前, 该算法首先判断该节点的所有前驱是否已被更新. 如果所有前驱已被更新, 则更新该节点, 否则先更新该节点的未被更新的前驱. 该算法的最终输出为 $\mathrm{Graph}_{v,\mathrm{final(CFG)}}$, 即为变量 v 在程序中所有的使用方式及使用方式之间的顺序关系的有向图. 也就是说, 该方法可以获得所有变量以及变量所对应的使用路径图, 且该图具有如下性质: 有向无环图, 且包含无数据节点及一个出度为 0 的节点 (尾部节点), 并且包含一个入度为 0 的节点 (根节点, 即变量的定义节点).

为了有效地利用变量使用路径图的特征, 类比于自然语言处理中的 N-gram 模型, IdenDup 截取该图中所有长度等于 N 的路径作为程序数据流特征进行提取. 值得注意的是, 变量之间并不是独立的, 而是存在着变量传播关系. 比如, 对于 "$a = b + c$", b 和 c 的后继节点同样包含之后所有的 a 的使用. 因此, 实际上, IdenDup 会将 b 和 c 在其变量使用路径图中的节点添加出边, 用来指向 a 的使用路径图的根节点. 也就是说, IdenDup 首先根据上述算法获得所有变量的使用路径图集合 G_{set}. 然后, 根据实际的变量之间的关系, 对 G_{set} 进行更新得到 $G_{\mathrm{set}'}$. 最后, IdenDup 提取出集合 $G_{\mathrm{set}'}$ 中所有图的长度为 N 的路径作为特征. 为了方便表述, 本文后续描述中使用 1gram 代表 $N = 1$ 时该方法所提取的特征, 2gram, 3gram 等以此类推. 在本文实现中, IdenDup 使用了 LLVM 的中间语言所包含的所有变量使用类型, 共 90 种. 由于 IdenDup 所选择的路径长度大于等于 1, 所以特征全集为该 90 种类型的变量使用的路径长度大于等于 1 的序列集合. 根据在给定 N 下的特征全集, IdenDup 统计每一个数据流特征出现的数量, 从而形成相应的数据流特征向量.

此外, 拓扑序遍历也是一种可能的方式来获取上述变量使用路径信息 [26]. 具体来说, 拓扑序是指将有向无环图中的节点进行线性排列, 对于任意两个节点 u 和 v, 如果图中存在从 u 到 v 的有向边 (u, v), 则节点 u 应排在节点 v 之前. 但是, 拓扑序遍历认为程序的分支是存在先后顺序的, 而在提取变量使用路径信息时, 不同的分支应该是等同对待, 不存在先后顺序关系. 因此, 本文未采用拓扑序遍历的方式, 而是提出上述数据流分析方法. 对于这两种方式在实际中所带来的具体效果差异, 我们计划在将来进行较为系统地实验探究.

3.2 特征处理

在获取特征向量之后, IdenDup 进一步对这些特征向量进行处理. 由于不同缺陷报告的文本及程序长度并不相同, 甚至差异很大, 为了降低长度对于特征频度的影响, IdenDup 进一步计算向量各维度

的权值. 本文介绍两种不同的权重计算策略, 分别为 TF-IDF [5] 和 BM25 [6]. 在后续的实验中, 我们也会比较不同权重计算策略对 IdenDup 的影响. 下面分别介绍这两种权重计算策略.

3.2.1 TF-IDF 权重计算策略

TF-IDF 是信息检索与自然语言处理领域常用的特征权重计算方法. 具体地说, TF (term frequency), 指的是词频, 其计算方法如式 (1) 所示, 其中 $N_{i,w}$ 指的是词 w 在文档 i 中出现的次数, T_i 指的是文档 i 中所有词的总数. 从 TF 可知, 某个词项在文档中出现的次数越多, 它和文档的主题越相关. IDF (inverse document frequency), 指的是逆文本频率, 其计算方法如式 (2) 所示, 其中 D 指的是文档的总数, M_w 指的是包含词项 w 的文档总数. 从 IDF 可知, 某个词项在文档集合的文档中出现次数越多, 该词项的区分能力越差.

$$\mathrm{TF}_{i,w} = \frac{N_{i,w}}{T_i}, \tag{1}$$

$$\mathrm{IDF}_w = \log \frac{D}{M_w + 1}. \tag{2}$$

在本文中, TF-IDF 中所说的词项指的是特征, 文档指的是缺陷报告. 根据 TF 和 IDF 的值, 缺陷报告 i 中的特征项 w 的 TF-IDF 值由 $\mathrm{TF}_{i,w} \cdot \mathrm{IDF}_w$ 计算获得. 根据 TF-IDF, 表征缺陷报告的特征向量转换成了一组新的处理后的特征向量, 其每一个值为其所对应的 TF-IDF 值.

3.2.2 BM25 权重计算策略

BM25 是在信息检索中根据查询对被查询的文档进行评分排序的常用算法, 其主要包含两个部分: 权值算法 (BM25 weighting scheme) 和检索状态值计算方法 (retrieval status value). 该算法已经在诸多全文检索引擎 (如 Apache Lucene, Xapian) 中被应用. 在这里, 我们的目的是进行权值计算, 因此本文仅使用 BM25 算法中的权值算法部分. 具体来说, BM25 的权值算法部分对于文档 i 中的词项 w 的权值计算如式 (3) 所示, 其中 L_i 为文档 i 的长度, L_{avg} 为所有文档的平均长度, k 和 b 为可变参数, k 调整词频在算法中的重要性, b 调整长度在算法中的重要性. 在这里, 根据已有工作 [27], IdenDup 设置 k 为 2, b 为 0.75.

$$\mathrm{Weight}_{i,w} = \mathrm{IDF}_w \cdot \frac{(k+1) \cdot \mathrm{TF}_{i,w}}{\mathrm{TF}_{i,w} + k \cdot (1 - b + \frac{b \cdot L_i}{L_{\mathrm{avg}}})}. \tag{3}$$

3.3 缺陷报告相似度度量

将特征进行处理之后, IdenDup 据此进一步度量不同缺陷报告是否属于重复报告. 为了完成该目标, 本文提出基于向量空间距离的相似度度量策略. 每一个缺陷报告已经被表示成特征向量形式, 因此, 直观上说, 两个向量之间的距离越大, 则相应的缺陷报告越不相似, 即更不可能是重复缺陷报告; 反过来, 两个向量之间的距离越小, 则相应的缺陷报告越有可能是重复缺陷报告. 在这里, 本文使用了 3 种度量向量空间距离的公式, 并对其进行了实验对比. 具体来说, IdenDup 所使用的 3 种距离公式分别为余弦距离、欧几里得距离, 以及曼哈顿距离, 其计算公式分别如式 (4)~(6) 所示, 其中, x 和 y 为两个 n 维的向量, x_i 和 y_i 分别代表向量 x 和 y 中的第 i 个元素的值.

$$\mathrm{Cos}(x,y) = \frac{\sum_{i=1}^{n}(x_i \cdot y_i)}{\sqrt{\sum_{i=1}^{n}(x_i)^2} \cdot \sqrt{\sum_{i=1}^{n}(y_i)^2}}, \tag{4}$$

$$\mathrm{Euc}(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}, \tag{5}$$

$$\mathrm{Man}(x,y)=\sum_{i=1}^{n}|x_i-y_i|. \tag{6}$$

当使用上述基于向量空间距离的相似度度量策略计算缺陷报告之间的相似度之后, 对于模糊测试场景的重复缺陷报告检测, IdenDup 将按照相似度对缺陷报告进行排序, 使得排在前边的缺陷报告更有可能是非重复的缺陷报告. 具体来说, IdenDup 采用最远点优先的排序策略 [4], 对缺陷报告进行排序. 该排序策略维护两个集合: 已排序的缺陷报告集合和未排序的缺陷报告集合, 其具体步骤如下:

- 随机选择一个缺陷报告放入已排序的缺陷报告集合中, 将其放在排序列表的第 1 位;
- 依次从未排序的缺陷报告集合中选择与当前已排序集合中所有缺陷报告最近距离最大的缺陷报告, 作为排序列表中下一位的缺陷报告并放入已排序集合中, 直到所有缺陷报告完成排序;
- 输出最终排序列表.

对于缺陷管理系统场景下的重复缺陷报告识别, IdenDup 对于给定的缺陷报告, 根据计算的相似度降序对缺陷报告进行排序, 即排在前边的缺陷报告与给定的缺陷报告更为相似.

4 实验设置

为了验证 IdenDup 的效果, 本文进行了实验探究. 具体地说, 本文的实验主要回答以下研究问题:

RQ1: IdenDup 在模糊测试场景下识别重复缺陷报告的效果如何?

RQ2: IdenDup 在缺陷管理系统场景下识别重复缺陷报告的效果如何?

RQ3: 不同权重计算策略对 IdenDup 的结果影响如何?

RQ4: 不同相似度计算公式对 IdenDup 的结果影响如何?

4.1 实验对象及数据

本文的实验使用了两个被广泛使用的 C 语言开源编译器作为实验对象, 即 GCC 和 LLVM [28, 29]. 本文所提出的方法 IdenDup 可应用于两个场景下的编译器重复缺陷报告识别. 因此, 我们分别在两个场景进行了实验来验证方法的有效性. 对于模糊测试场景下的重复缺陷识别, 我们使用了已有工作的数据集 [4]. 由于 Chen 等 [4] 提出的方法能够很好地解决模糊测试场景下的编译器崩溃缺陷的重复报告识别问题, 而无法有效解决编译器错误结果缺陷的重复报告识别问题. 因此, 在该场景下, 我们集中在编译器结果错误缺陷的重复报告识别上, 即我们使用了他们的编译器结果错误缺陷的数据集作为该场景的数据集. 具体来说, 该数据集是对 GCC-4.3.0 进行模糊测试的数据集, 共包含 1275 个触发编译器结果错误缺陷的测试程序, 共触发 35 个非重复的编译器结果错误缺陷. 对于缺陷管理系统场景下的重复缺陷报告识别, 由于没有已有工作解决过该问题, 因此没有相关数据集. 为此, 我们分别从 GCC 和 LLVM 缺陷管理系统中收集了 330 个 GCC 缺陷报告和 257 个 LLVM 缺陷报告作为实验数据集.

4.2 方法实现

在实现 IdenDup 时, 针对特征提取部分, 我们的实现是基于 LLVM Pass 插件、LibClang 和 Python 库 Scikit-learn 完成的. 其中 LibClang 负责提取程序的词法特征和语法特征, LLVM Pass 插件提供了对中间语言代码的数据流分析框架, 其余特征信息是基于 Scikit-learn 进行提取. 针对特征处理和距离计算, 我们的实现是基于 Python 库 Numpy 和 Scikit 完成的. 其中前者用于数据存取, 后者进行权值的计算以及距离的计算.

4.3 实验自变量

IdenDup 由 3 个步骤组成, 其中每一个步骤都涉及到多种选择. 因此, 每一步骤下的各种选择都将作为实验中的自变量, 具体如下:

- **程序特征:** 文本特征和程序特征, 其中程序特征包含程序词法特征、语法特征、数据流特征. 针对程序数据流特征, 根据所设置的 N 值, IdenDup 将有不同路径长度的数据流特征. 本文选取 N 分别为 1, 2, 3. 因此, IdenDup 可使用的特征包括文本特征、程序词法特征、语法特征、数据流 1gram 特征、数据流 2gram 特征, 以及数据流 3gram 特征, 共 6 种.
- **权重计算策略:** TF-IDF 和 BM25, 其中 IdenDup 使用 TF-IDF 作为默认配置.
- **向量空间距离公式:** 余弦距离、欧几里得距离, 以及曼哈顿距离, 其中 IdenDup 使用余弦距离作为相似度计算公式的默认配置.

另一个实验自变量为对比方法. 针对模糊测试场景的重复缺陷报告识别, 我们所使用的对比方法为 Chen 等 [4] 所提出的方法以及随机方法. Chen 等所提出的方法是解决该问题的最新方法, 也是唯一方法. 他们的方法也尝试了各种特征的组合, 在这里, 我们选择其效果最好的版本作为对比方法. 该效果最好的版本是使用程序动态执行下的函数覆盖信息作为特征进行欧几里得距离计算, 然后利用最远点优先策略进行排序. 随机方法在实验中作为基准方法, 该方法随机对缺陷报告进行排序. 针对缺陷管理系统场景下的编译器重复缺陷报告识别, 由于目前没有工作尝试解决过该问题, 我们也选择随机方法作为基准方法.

4.4 效果度量方式

针对模糊测试场景的编译器重复缺陷报告识别, 我们采用已有工作的度量方式, 即缺陷发现曲线 [4]. 对缺陷报告的排序列表, 查看前 k 个时所能发现的非重复的缺陷报告的数量. 针对缺陷管理系统场景的重复缺陷报告识别, 我们采用一般软件的缺陷管理系统中重复缺陷报告识别的度量方式, 即召回率曲线 [1,8]. 具体来说, 对于给定的缺陷报告, 方法能够找到的 k 个与它最可能重复的缺陷报告中, 真正是重复缺陷报告的数量与跟它重复的全部缺陷报告的数量的比例.

4.5 实验过程

为了回答上述研究问题, 我们使用默认的 TF-IDF 和余弦距离, 对各种特征的不同组合进行实验, 来探索 IdenDup 的效果. 通过组合不同特征, 我们的方法有各种版本, 每一个版本都会输出一个缺陷报告的排序列表. 与此同时, 每一个对比方法也都输出一个缺陷报告的排序列表. 对于随机方法, 为了降低随机因素带来的影响, 我们对其重复了 1000 次, 然后计算其平均结果. 在模糊测试场景下, IdenDup 也包含随机因素, 即随机选择排在第 1 位的缺陷报告. 因此, 我们在该场景下对 IdenDup 重复 20 次, 并计算其平均效果. 之后, 我们分别对不同的权重计算策略和相似度度量公式的效果进行了探索. 每探索一个自变量的影响时, 我们会固定其他自变量的选择, 以达到控制变量的效果. 具体的控制策略在后续的实验结果中进行较为详细地描述.

4.6 实验影响因素

实验中的内部影响因素主要在于我们的实现. 为了减少实验代码中的缺陷, 前两个作者仔细检查了代码的正确性, 同时尽可能使用了程序的第三方工具来辅助我们的实验, 比如 Libclang 和 LLVM Pass 插件等.

实验中的外部影响因素主要在于实验对象和实验数据. 在实验中, 尽管我们使用了 C 语言的两个主流开源编译器 GCC 和 LLVM, 但他们仍旧无法代表其他编译器. 之后, 我们将进一步使用更多语言的更多编译器来验证方法的效果. 对于所使用的实验数据, 针对模糊测试场景的实验, 我们使用了 Chen 等所使用的数据集; 针对缺陷管理系统场景, 我们自己从 GCC 和 LLVM 的缺陷管理系统中分别收集了 300 个 GCC 缺陷以及 257 个 LLVM 缺陷. 由于收集缺陷十分耗时, 因此我们的数据规模不是十分大. 为了减小数据带来的影响, 我们也将收集更多模糊测试场景的数据, 同时进一步收集缺陷管理系统场景的缺陷, 来扩大数据规模.

实验中的构造影响因素主要在于实验使用的度量方式、随机因素, 以及方法中的各种策略. 对于实验中的度量方式, 我们使用了已有工作常用的度量方式: 缺陷发现曲线和召回率曲线. 之后, 我们将引入更多的度量方式来更充分地衡量方法的效果. 对于随机因素, 我们的实验通过重复多次来减少其影响. 对于方法中的各种策略, 尽管我们无法保证所用的策略是最优的, 但是我们对多种策略 (包括多种权重计算策略和多种相似度度量公式) 进行了实验探究, 从而降低了该影响.

5 实验结果与分析

5.1 RQ1: 模糊测试场景下的重复缺陷报告识别效果

我们首先探索 IdenDup 在模糊测试场景下识别重复缺陷报告的效果. 由于 Chen 等 [4] 所提方法利用了多种特征, 并尝试了多种特征的组合, 因此, 我们使用其效果最好的一组特征的结果进行对比. 具体地, 该版本是利用测试程序动态执行时的编译器函数覆盖信息来度量缺陷报告的重复性. 由于本文所提方法 IdenDup 同样存在着多种特征的组合, 我们首先比较 IdenDup 在不同特征组合下的效果, 其结果如图 3 和 4 所示. 图 3 展示了 IdenDup 在不同二元特征组合下的效果, 图 4 展示了 IdenDup 在不同三元特征组合下的效果. 在图 3 和 4 中, 横坐标代表了按照缺陷报告的排序列表, 依次检查的缺陷报告的数量; 纵坐标代表了检查一定数量的缺陷报告时所发现的非重复缺陷的数量, FunCov 代表了 Chen 等 [4] 所提方法的效果最好的版本, Random 代表了随机方法. 从图 3 和 4 中, 我们可以发现, IdenDup 在不同特征组合下的效果较为接近 (尤其是缺陷报告的数量较少的时候), 其中词法特征和数据流 2gram 特征组合的效果在缺陷报告数量较多时 (从 35 到 50) 一直稳定地优于所有其他特征组合的效果. 综合来看, IdenDup 在词法特征和数据流 2gram 特征的组合下表现更优. 因此, 我们将词法特征和数据流 2gram 特征组合下的 IdenDup 作为代表进行后续的实验与分析, 记为 IdenDup(Best).

图 5 展示了 IdenDup(Best) 与对比方法的比较结果. 与此同时, 我们还计算了 IdenDup 在各种特征组合下的平均效果, 记为 IdenDup(Average). 从图 5 中可以看出, 已有方法的最优效果与 IdenDup 的不同特征组合的平均效果十分接近. 而与 IdenDup 的最优特征组合的效果相比, 当检查的缺陷报告的数量超过 25 时, IdenDup 一直稳定地优于已有方法的最优效果. 例如, 当检查的缺陷报告数量为 50 的时候, IdenDup 发现了 25 个非重复缺陷, 比 FunCov 提升了 13% 的准确性. 与此同时, 从图 5 中可以发现, IdenDup 的效果一直显著好于随机方法的效果.

5.2 RQ2: 缺陷管理系统场景的重复缺陷报告识别效果

我们进一步探索 IdenDup 在缺陷管理系统场景识别重复缺陷报告的效果, 其中 IdenDup 在 GCC 编译器上的效果如图 6 所示, 在 LLVM 编译器上的效果如图 7 所示. 在图 6 和 7 中, IdenDup 代表我们所提方法的效果最好的版本 (即使用词法特征和数据流 2gram 特征组合), Random 代表随机方法. 图 6 和 7 中的横坐标代表所推荐的相似缺陷报告列表, 纵坐标代表计算得到的召回率. 从图 6 和 7 中

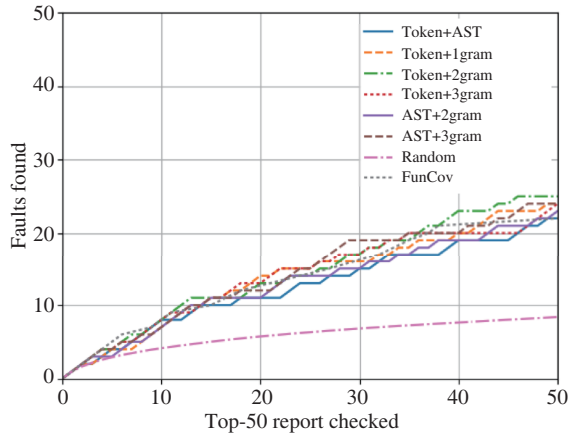


图 3 (网络版彩图) 模糊测试场景不同二元特征组合的 IdenDup 效果

Figure 3 (Color online) Effectiveness of IdenDup with different two-tuple feature combinations during fuzz testing scenario

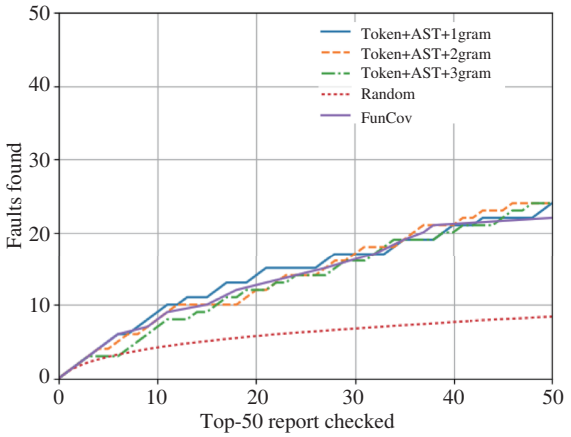


图 4 (网络版彩图) 模糊测试场景不同三元特征组合的 IdenDup 效果

Figure 4 (Color online) Effectiveness of IdenDup with different three-tuple feature combinations during fuzz testing scenario

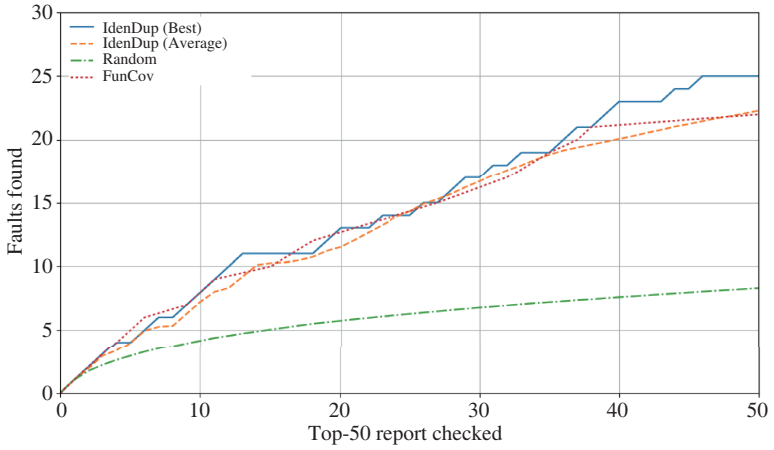


图 5 (网络版彩图) 模糊测试场景的重复缺陷报告识别效果

Figure 5 (Color online) Effectiveness of duplicate bug report identification during fuzz testing scenario

可以看出, IdenDup 可以在 GCC 和 LLVM 两个编译器上均表现出很好的效果. 比如, 当所推荐的相似缺陷报告列表为 25 时, IdenDup 在 GCC 编译器上的召回率接近 75%, 在 LLVM 编译器上的召回率也达到了 65%. 特别地, IdenDup 的效果一直显著地优于对比方法的效果. 作为第 1 个可应用于编译器缺陷管理系统场景下的重复缺陷检测方法, IdenDup 确实取得了不错的效果.

5.3 RQ3: 权重计算策略的影响

在验证了 IdenDup 在上述两个场景下的整体效果之后, 我们进一步探索 IdenDup 中不同的选择所带来的影响. 这里我们使用模糊测试场景下的重复缺陷报告识别作为代表进行实验探究. 在实验中, 为了探索某一个因素对 IdenDup 的影响, 我们需要对其他影响因素进行控制. 本小节控制所使用的特征以及相似度度量公式相同, 来探索不同权重计算策略对于 IdenDup 的影响. 具体来说, 我们使用了

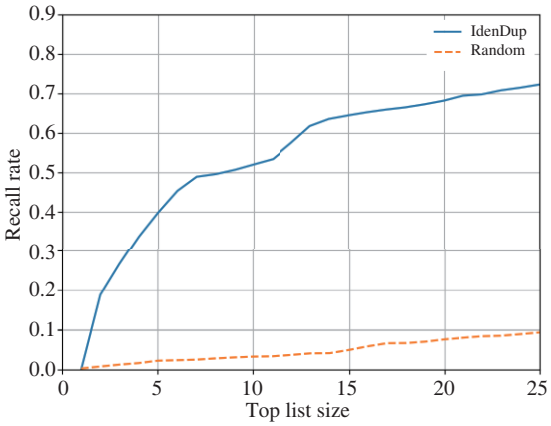


图 6 (网络版彩图) GCC 缺陷管理系统场景的重复报告识别效果

Figure 6 (Color online) Effectiveness of GCC duplicate bug report identification in the bug management system

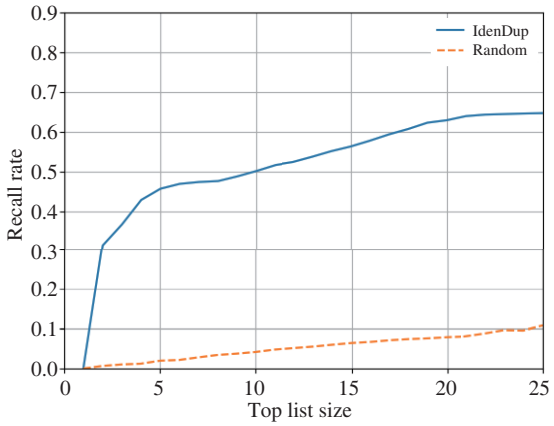


图 7 (网络版彩图) LLVM 缺陷管理系统场景的重复报告识别效果

Figure 7 (Color online) Effectiveness of LLVM duplicate bug report identification in the bug management system

数据流 3gram 特征作为特征代表, 并且使用余弦距离公式作为相似度度量公式的代表. 在这里, 所控制的因素具有多种选择, 不同的选择也可能会对实验产生一定的影响, 因此, 该部分也是实验中的可能影响因素之一. 但实际上, 这里所涉及的不同因素是正交的, 比如, 特征是为了刻画一个测试程序与触发缺陷之间的关系, 最终以向量的形式进行呈现, 而权重策略则不在意向量的生成过程, 而是对向量进行处理. 因此, 该影响因素实际上对实验的影响较小. 在将来, 我们也将进一步通过实验来探索所控制因素的选择不同对实验的具体影响情况.

图 8 展示了不同权重计算策略的对比效果, 包括 TF-IDF, BM25, 以及基准方法随机方法. 从图 8 可以看出, 无论哪一种权重计算策略 (即 TF-IDF 和 BM25), 都显著优于随机方法. 对比 TF-IDF 和 BM25 的效果可知, 该两种权重计算策略可以达到较为相似的效果, 其中 BM25 较优于 TF-IDF. 具体地, 当所检查的缺陷报告的数量为 50 的时候, 在该设置下, BM25 可以发现 22 个非重复的缺陷报告.

5.4 RQ4: 相似度度量公式的影响

本小节控制所使用的特征以及权重计算策略相同, 来探索不同相似度度量公式对于 IdenDup 的影响. 具体来说, 我们也使用了数据流 3gram 特征作为特征代表, 并且使用 TF-IDF 作为权重计算策略代表. 图 9 展示了不同相似度度量公式的对比效果, 包含余弦距离公式、欧几里得距离公式、曼哈顿距离公式, 以及基准方法随机方法. 从图 9 中, 我们可以发现, 余弦距离公式的效果要显著优于欧几里得距离公式和曼哈顿距离公式的效果, 特别地, 前者显著优于随机方法的效果, 而后两者则差于随机方法的效果. 具体地, 当所检查的缺陷报告的数量为 50 的时候, 基于余弦距离公式的 IdenDup 可以发现 20 个非重复缺陷, 然而, 基于欧几里得距离公式的 IdenDup 仅仅能发现 4 个非重复缺陷. 与此同时, 基于曼哈顿距离公式的 IdenDup 仅仅能发现 3 个非重复缺陷. 也就是说, 余弦距离公式能够更加有效地度量缺陷报告之间的相似度. 这也是与自然语言处理领域中的文本处理的经验一致, 即余弦距离公式往往具有更优的效果.

5.5 实验总结

通过上述 4 个实验对 IdenDup 效果的探究, 我们将主要结论总结如下:

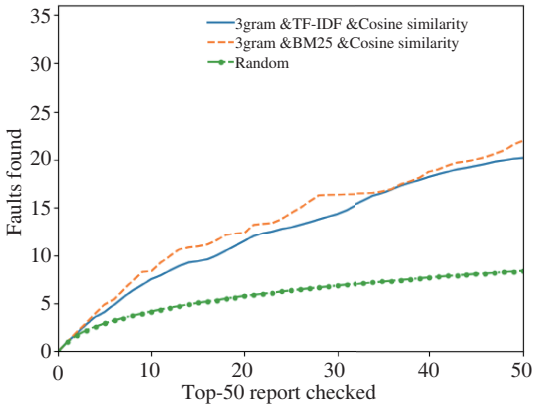


图 8 (网络版彩图) 不同权重计算方式对 IdenDup 的影响

Figure 8 (Color online) Impact of weight calculation strategies

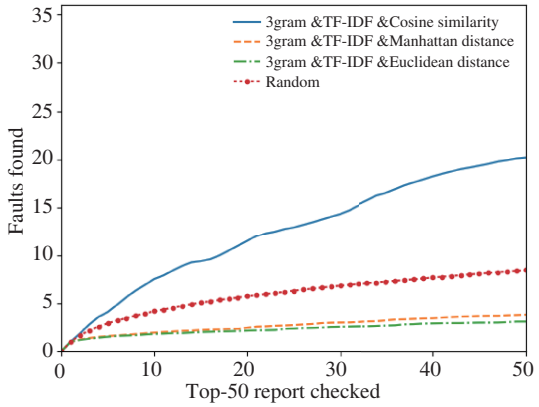


图 9 (网络版彩图) 不同相似度度量公式对 IdenDup 的影响

Figure 9 (Color online) Impact of similarity formulae

• IdenDup 可以在模糊测试场景与缺陷管理系统场景下均取得较优效果, 且优于对比方法. 值得注意地是, IdenDup 是第 1 个可用于编译器缺陷管理系统场景的重复缺陷检测方法.

• IdenDup 在两种权重计算策略 TF-IDF 和 BM25 中, 具有相似的效果, 其中 BM25 要稍优于 TF-IDF.

• IdenDup 在不同相似度度量公式中, 具有较大的效果差异, 其中余弦距离公式显著优于欧几里得距离公式和曼哈顿距离公式.

6 总结

识别编译器的重复缺陷报告, 对编译器的质量保证具有重要作用. 本文提出了一种静态的编译器重复缺陷报告识别方法 IdenDup, 其可以用于解决两个场景下的编译器重复缺陷报告问题, 即模糊测试场景和缺陷管理系统场景. 具体来说, IdenDup 使用了缺陷报告中的静态文本与程序特征, 其中程序特征包括程序词法特征、语法特征, 以及本文首次提出的数据流特征. 其中, 程序数据流特征指的是程序中变量的使用路径特征 (包括变量使用方式以及使用方式之间的顺序). 基于这些特征, IdenDup 进一步对其进行了权重计算, 进而提出不同度量公式来计算缺陷报告之间的相似度. 之后, 本文使用两个主流的 *C* 语言编译器 GCC 和 LLVM, 对 IdenDup 的效果进行了实验验证. 实验结果表明, IdenDup 可以有效地识别上述两个场景下的编译器重复缺陷报告, 并且超过了已有方法和基准方法.

参考文献

1 Sun C N, Lo D, Wang X Y, et al. A discriminative model approach for accurate duplicate bug report retrieval. In: Proceedings of 2010 ACM/IEEE 32nd International Conference on Software Engineering, Cape Town, 2010. 45–54

2 Nguyen A T, Nguyen T T, Nguyen T N, et al. Duplicate bug report detection with a combination of information retrieval and topic modeling. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, 2012. 70–79

3 Alipour A, Hindle A, Stroulia E. A contextual approach towards more accurate duplicate bug report detection. In: Proceedings of the 10th Working Conference on Mining Software Repositories, San Francisco, 2013. 183–192

4 Chen Y, Groce A, Zhang C Q, et al. Taming compiler fuzzers. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, Seattle, 2013. 197–208
5 Ramos J. Using TF-IDF to determine word relevance in document queries. In: Proceedings of the 1st Instructional Conference on Machine Learning, 2003. 242: 133–142
6 Robertson S, Zaragoza H. The probabilistic relevance framework: BM25 and beyond. Found Trends Inf Ret, 2009, 3: 333–389
7 Tian Y, Sun C N, Lo D. Improved duplicate bug report identification. In: Proceedings of the 16th European Conference on Software Maintenance and Reengineering, 2012. 385–390
8 Sun C N, Lo D, Khoo S C, et al. Towards more accurate retrieval of duplicate bug reports. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, 2011. 253–262
9 Robertson S, Zaragoza H, Taylor M. Simple BM25 extension to multiple weighted fields. In: Proceedings of the 13th ACM International Conference on Information and Knowledge Management, Washington, 2004. 42–49
10 Wang X Y, Zhang L, Xie T, et al. An approach to detecting duplicate bug reports using natural language and execution information. In: Proceedings of the 30th International Conference on Software Engineering, Leipzig, 2008. 461–470
11 Lerch J, Mezini M. Finding duplicates of your yet unwritten bug report. In: Proceedings of the 17th European Conference on Software Maintenance and Reengineering, 2013. 69–78
12 Regehr J, Chen Y, Cuoq P, et al. Test-case reduction for C compiler bugs. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, Beijing, 2012. 335–346
13 Pflanzer M, Donaldson A F, Lascu A. Automatic test case reduction for OpenCL. In: Proceedings of the 4th International Workshop on OpenCL, Vienna, 2016
14 Herfert S, Patra J, Pradel M. Automatically reducing tree-structured test inputs. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, 2017. 861–871
15 Sloane A M. Debugging Eli-generated compilers with Noosa. In: Proceedings of International Conference on Compiler Construction. Berlin: Springer, 1999. 17–31
16 Yang X J, Chen Y, Eide E, et al. Finding and understanding bugs in C compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, San Jose, 2011. 283–294
17 Lidbury C, Lascu A, Chong N, et al. Many-core compiler fuzzing. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, 2015. 65–76
18 McKeeman W M. Differential testing for software. Digit Tech J, 1998, 10: 100–107
19 Le V, Afshari M, Su Z D. Compiler validation via equivalence modulo inputs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Edinburgh, 2014. 216–226
20 Chen J J, Hu W X, Hao D, et al. An empirical comparison of compiler testing techniques. In: Proceedings of 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016. 180–190
21 Chen J J, Bai Y W, Hao D, et al. Test case prioritization for compilers: a text-vector based approach. In: Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST), 2016. 266–277
22 Chen J J, Bai Y W, Hao D, et al. Learning to prioritize test programs for compiler testing. In: Proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017. 700–711
23 Chen J J. Learning to accelerate compiler testing. In: Proceedings of the 40th International Conference on Software Engineering, 2018. 472–475
24 Chen J J, Wang G C, Hao D, et al. Coverage prediction for accelerating compiler testing. IEEE Trans Softw Eng, 2019. doi: 10.1109/TSE.2018.2889771
25 Nielson F, Nielson H R, Hankin C. Principles of Program Analysis. Berlin: Springer, 2015
26 Kahn A B. Topological sorting of large networks. Commun ACM, 1962, 5: 558–562
27 Schutze H, Manning C D, Raghavan P. Introduction to Information Retrieval. Cambridge: Cambridge University Press, 2008
28 Chen J J, Han J Q, Sun P Y, et al. Compiler bug isolation via effective witness test program generation. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019. 223–234
29 Chen J J, Wang G C, Hao D, et al. History-guided configuration diversification for compiler test-program generation. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2019

Static duplicate bug-report identification for compilers

Junjie CHEN[1†], Wenxiang HU[2], Dan HAO[2*], Yingfei XIONG[2], Hongyu ZHANG[3] & Lu ZHANG[2]

1. *College of Intelligence and Computing, Tianjin University, Tianjin* 300350*, China*;
2. *School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871*, China*;
3. *School of Electrical Engineering and Computing, The University of Newcastle, Callaghan* 2308*, Australia*

* Corresponding author. E-mail: haodan@pku.edu.cn

† This work was done when the author was in Peking University

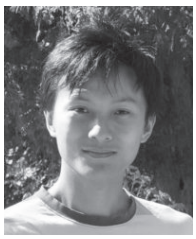
Abstract Compiler bug reports are important for guaranteeing compiler quality; however, duplicate bug reports tend to incur extra costs. To identify duplicate bug reports for compilers, we propose a static approach (IdenDup) to identifying duplicate bug reports for compilers. This method effectively identifies duplicate bug reports for compilers in two scenarios (fuzz testing and the bug-management system) by utilizing static text and program information, including lexical features, syntax features, and proposed dataflow features that describe variable-usage path features (i.e., how variables are used and their order). We conducted empirical evaluations of the effectiveness of IdenDup based on the use of GCC and LLVM, with our results demonstrating that IdenDup effectively identified duplicate bug reports in the two scenarios for compilers and outperformed existing approaches.

Keywords compiler debugging, compiler bug report, duplicate bug report, dataflow analysis, static approach

Junjie CHEN is an associate professor at the College of Intelligence and Computing, Tianjin University. He received his Ph.D. from Peking University in 2019. His research interests include software testing and debugging and mainly focus on compiler testing, regression testing, and automated debugging.

Wenxiang HU received his master of science degree from Peking University in 2017. His research interest is software testing primarily focused on compiler testing.

Dan HAO is an associate professor at the School of Electronics Engineering and Computer Science, Peking University, China. She received her Ph.D. degree in computer science from Peking University in 2008, and a B.S. degree in computer science from Harbin Institute of Technology in 2002. Her current research interests include software testing and debugging.



Yingfei XIONG is an associate professor at Peking University. He received his Ph.D. degree from the University of Tokyo in 2009 and worked as a postdoctoral fellow at the University of Waterloo from 2009 to 2011. His research interests include software engineering and programming languages.