软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(4):1594−1612 [doi: 10.13328/j.cnki.jos.006691]

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

一种结合动态链接库信息的崩溃输入分类方法*

王文祥[1,2,3], 高 庆[1,2], 许 可[4], 张世琨[1,2]

[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](对外经济贸易大学 统计学院, 北京 100029)

通信作者: 许可, E-mail: xk@uibe.edu.cn

摘 要: 软件崩溃是一种严重的软件缺陷, 可导致软件终止运行. 因此, 对软件崩溃的测试在软件迭代的过程中极为重要. 近年来, 模糊测试技术(如 AFL)由于可以自动化生成大量的测试输入来触发软件崩溃, 被广泛用于软件测试中. 然而, 通过该技术产生的导致软件崩溃的测试输入中, 大部分崩溃的触发原因都是重复的, 因此软件开发人员需要对测试输入进行分类, 带来了许多冗余工作. 目前, 测试输入分类已经有很多自动化方法, 主要包括基于程序修复的分类算法和基于软件崩溃信息的分类算法. 前者通过对程序在语义上进行分析, 在运行时通过在程序中替换修复模板后重新运行测试输入, 进而对输入分类. 因为此方法需要人为地对于软件崩溃编写修复模板, 所以其分类的效率与修复模板的质量存在很大联系; 且由于需要先修复崩溃、再对崩溃做分类, 影响了软件崩溃的修复效率. 采用后者的思想, 提出了一种轻量而高效的利用软件崩溃信息的测试输入分类算法 CICELY. 其在软件崩溃点堆栈信息分类的算法基础上, 在分析软件崩溃点堆栈时引入了动态链接库信息, 通过区分系统动态链接库与用户动态链接库, 结合用户代码位置信息, 得到用户关注的函数集合, 以在分类时以用户函数为基准对崩溃进行界定. 最后, 分别将 CICELY 与几种基于程序修复的分类算法和基于软件崩溃信息的流行分类工具进行了比较, 实验测试的数据集共计 19 个项目、42 组测试集. 在与基于软件崩溃信息的分类工具 Honggfuzz, CERT BFF 在相同数据集上比较时, CICELY 在分类结果的组数上比上述二者减少了 2112.89%和 135.05%, 说明 CICELY 在同类算法上的实验效果有较大提升, 具有更高的精确性. 在与基于程序修复的分类算法“语义崩溃分类”用其论文中提供的测试数据集进行比较时, CICELY 比“语义崩溃分类”的分组结果差 4.42%; 在由对应了多个崩溃的测试输入所组成的测试集上实验时, CICELY 比“语义崩溃分类”分组的重复度高了 3%. 但是语义崩溃分类只能对于空指针解引用和缓冲区溢出这两种崩溃输入导致的崩溃进行分类, CICELY 不受这样的限制.

关键词: 软件崩溃; 测试输入分类; 动态链接库; 程序修复; 模糊测试

中图法分类号: TP311

中文引用格式: 王文祥, 高庆, 许可, 张世琨. 一种结合动态链接库信息的崩溃输入分类方法. 软件学报, 2023, 34(4): 1594−1612. http://www.jos.org.cn/1000-9825/6691.htm

英文引用格式: Wang WX, Gao Q, Xu K, Zhang SK. Crash Input Classification Method Combined with Dynamic Link Library Information. Ruan Jian Xue Bao/Journal of Software, 2023, 34(4): 1594−1612 (in Chinese). http://www.jos.org.cn/1000-9825/6691.htm

Crash Input Classification Method Combined with Dynamic Link Library Information

WANG Wen-Xiang[1,2,3], GAO Qing[1,2], XU Ke[4], ZHANG Shi-Kun[1,2]

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

* 基金项目: 国家自然科学基金(12001102); 中央高校基本科研业务费专项资金(19QD22)
收稿时间: 2021-08-17; 修改时间: 2021-12-07; 采用时间: 2022-04-15

[2](National Engineering Research Center for Software Engineering, Peking University, Beijing 100871, China)
[3](School of Software and Microelectronics, Peking University, Beijing 102600, China)
[4](School of Statistics, University of International Business and Economics, Beijing 100029, China)

Abstract: Software crash is a kind of serious software flaw, which can lead to software crashes. Therefore, testing for software crashes is extremely important in the process of software iteration. In recent years, since a large number of test inputs can be automatically generated to trigger software crashes, fuzzing techniques (such as AFL) are widely used in software testing. Nevertheless, most of root causes of crashes that are generated by this technique are same. In this case, software developers have to classify the test inputs one by one, which brings a lot of redundant work. At present, there are many automated methods for testing input classification, mainly including classification algorithms based on program repair and classification algorithms based on software crash information. The former analyzes the program semantics, and re-runs the test input after replacing the repair templates in the program at runtime, and then classifies the inputs. Since this method requires the preparation of repair templates to be completed artificially, the efficiency of its classification is closely related to the quality of the repair templates. At the same time, the repair efficiency of the software has been greatly affected due to the need to repair the crash and classify the crash. Since certain advantages of the latter, this study proposes a lightweight and efficient test inputs classification algorithm, which uses software crash information. Based on the algorithm of software crash point stack information classification, this study introduces dynamic link library information in analyzing CICELY. By distinguishing system dynamic link library from user dynamic link library and combining with location information of user codes, this study gets the set of functions that are focused by programmers to define the crash based on the user function in the classification. In the end, this study also compares CICELY with some existing classification tools based on program repair and software crash information. The experimental test data sets total 19 projects, and 42 test sets. When comparing with other classification tools, Honggfuzz and CERT BFF, whose main classification algorithms are based on software crash information on the same data set, the numbers of classification results of the two are 2112.89% and 135.05% worse than that of CICELY, proving that the experimental effect of CICELY is greatly improved and has higher accuracy compared with similar algorithms. Compared with the classification algorithm "Semantic Crash Bucketing" based on program repair using the test data set provided in their article, CICELY is worse than it by 4.42%. When using the test set consisting of test inputs corresponding to multiple crashes, CICELY got 3% higher repeatability than it. However, Semantic Crash Bucketing can only classify crashes caused by two kinds of crash inputs, null pointer dereference and buffer overflow, while CICELY is not subject to such restrictions.

Key words: software crash; test inputs classification; dynamic link library; program repair; fuzzing

软件崩溃可能带来严重危害, 例如: 1990 年, 美国 AT&T 公司的通话网络发生故障[1], 使得相邻交换机之间陷入死循环, 进而导致整个美国的电话系统在 9 小时内便全部崩溃, 在当时造成了 6 000 万美元以上的损失; 2006 年, 美国 NASA 发射的火星探测器[1]突发崩溃, 导致其失去了与地面人员之间的联系, 浪费了 2.4 亿美元的成本, 是人类太空探索事业上的一次惨痛教训. 因此, 对软件崩溃的修复, 是所有软件开发与测试人员都密切关注的问题. 在修复软件崩溃时, 开发人员需要分析触发崩溃的测试输入, 从而进一步确定软件发生崩溃的原因. 然而, Klees 等人[2]研究表明, 大量的导致软件崩溃的测试输入通常只能对应少量的软件崩溃原因. 所以, 当开发者收集到触发崩溃时的测试输入信息后, 如果对每一个崩溃输入都进行测试与分析, 则将会浪费大量时间在相同原因所导致的崩溃上.

为了减少开发人员与测试人员分析冗余用例的时间, 提高崩溃修复的效率, 如今已经有多种技术用于为测试输入进行分类. 目前的主流方法是基于软件崩溃信息的分类算法, 代表性工具包括知名开源模糊测试工具 CERT BFF[3]和 Honggfuzz[4]. CERT BFF 在利用堆崩溃点处的函数调用堆栈进行哈希时, 为了降低运行中的细微变化所带来的执行命令不同所造成的影响, 设计了模糊堆栈哈希(fuzzy stack hash), 在计算哈希时, 对行号进行了模糊化处理, 即位置距离极近的两处崩溃可以映射到相同的哈希值上. Honggfuzz 默认会使用崩溃点处向上 7 层的函数调用堆栈计算哈希, 使用崩溃时的地址与指令信息确定崩溃的唯一性. 在判定崩溃的唯一性时, Honggfuzz 还会考虑溢出的堆栈中的数据. 这些工具在使用上较为简单, 且不受崩溃类型的限制, 可以对所有崩溃类型进行分类. 然而, 部分研究者的实验结果表明, 上述两工具在分类的效果上都有所不足[5].

近年来, 有研究者提出了基于程序修复的语义崩溃分类(semantic crash bucketing, SCB)[5]算法, 在对于触发了空指针解引用和缓冲区溢出这两种崩溃的输入进行分类时, 其分类结果对于 CERT BFF 和 Honggfuzz 有

明显提升. 在 SCB 论文所述实验中, SCB 可以对于上述两工具的分类结果进行进一步分类, 在很大程度上减少了重复分类的情况. 但是 SCB 的问题在于: 目前其只能在上述两种类型的测试集上进行分类, 且需要程序修复模板, 扩展性不足.

本文针对基于软件崩溃信息的分类算法 Honggfuzz 和 CERT BFF 进行了改进. 具体方法为: 本文在传统的基于软件崩溃信息分类算法的思想的基础上, 通过识别程序运行时的动态链接库信息, 进而确定用户函数范围的方法对传统方法进行了优化. 整体而言, 本文算法相较其他基于软件崩溃信息的算法而言, 拥有更高的准确性.

本文的主要贡献总结如下:

(1) 本文以传统的基于软件崩溃信息的分类工具的思想为基础, 提出了在分类时结合动态链接库信息的方法, 对现有的基于软件崩溃信息的分类算法进行了优化, 并命名为 CICELY(crash inputs classification employing library). CICELY 通过将动态链接库分为系统动态链接库和用户动态链接库,并利用动态链接库的地址范围信息判断用户函数, 大大提高了同类分类算法的准确性;

(2) 将 CICELY 与其他测试输入分类工具在共计 19 个项目、42 组测试集上进行了实验, 测试不同分类工具在测试集上进行分类时分类结果的组数, 并进行了比较和分析. 在与工业界和学术界常用的两种基于软件崩溃信息的分类工具 Honggfuzz, CERT BFF 在相同数据集上比较时, CICELY 在分类结果的组数上比上述二者减少了 2112.89%和 135.05%. 本文在与基于程序修复的分类算法 SCB 用其论文中提供的测试数据集进行比较时, CICELY 比 SCB 的分组结果差 4.42%; 在由对应了多个崩溃的测试输入所组成的测试集上实验时, CICELY 比 SCB 分组的重复度高了 3%. 实验结果说明: CICELY 在同类算法上的实验效果有较大提升, 具有更高的精确性; 与基于程序修复的分类算法 SCB 对比时, CICELY 依然可以达到相近的水平. 此外, CICELY 比 SCB 在其他方面更有优势, 如适用的崩溃类型多于 SCB 等.

1 相关工作

为了检测软件中的崩溃, 研究人员已经设计了许多方法并进行改进. 如模糊测试工具 AFL[6], 其旨在通过对初始的测试输入进行不断的变异, 再将其输入到被测程序中, 以尽可能地达到提高覆盖率的目的. 而许多研究人员也在 AFL 的基础上不断优化, 并设计了 AFLGo[7], AFLSmart[8]等工具. 模糊测试技术的核心思想为: 当测试输入在软件中执行时, 记录其执行路径: 如果执行路径不同, 则认为找到了一个全新的测试输入, 而最终在输出时, 每一条路径所对应的测试输入也只会保留一个. 然而, 这种只对测试输入路径进行唯一性判断的方法容易仅根据崩溃的触发路径进行初步的过滤, 其生成的结果中仍有大部分测试输入的触发原因是相同的. AFL 除了具有优秀的崩溃输入的探索能力外, 还提供了崩溃模式(crash mode), 通过该模式, 可以针对某一特定崩溃输入进行测试输入库(corpus)扩充, 生成一组大部分测试输入都与此指定崩溃的触发原因相同的测试输入集. 此方法曾在 SCB 中用于生成测试输入库, 作为测试集进行对其效果的测试, 且本文的测试集也是来源于此.

1.1 基于软件崩溃信息崩溃分类

许多研究者在设计算法对测试输入集分类时, 都利用了软件崩溃信息进行分类. Tejinder 等人[9]通过堆栈路径的相似性来对 Mozilla 收集到的崩溃报告进行分类, 从而降低崩溃的修复时间. Kim 等人[10]在研究微软公司收集到的崩溃报告时, 提出了一种为每一组崩溃构建崩溃图(crash graph)来构建树状图的方法, 来解决崩溃报告的重复度过高问题. 通过将堆栈中的函数作为图中的点, 将函数调用关系作为图中的边, 为每组崩溃构建出崩溃图后, 通过比较图的相似度来判断崩溃是否重复. Dang 等人[11]对测量测试输入间相似性的方法进行了优化, 他们设计了位置相关模型(position dependent model)来对崩溃进行度量, 并基于此提出了一种测试输入分类方法 ReBucket, 根据软件崩溃时堆栈的相似度对实现对崩溃的分类. Golagha 等人[12]提出了一种使用一些非代码性的特征对导致程序执行崩溃的测试输入进行分类的方法, 他们使用的测试特征包含标识符(identifiers)、组件成员变量(component membership)、测试执行的历史数据(history data of test execution)、损坏

/修复的特性(broken/repaired features)和从一些问题跟踪管理器(如 Jira)中收集的数据, 并利用这些信息对于执行失败的测试输入进行分类.

除此之外, 许多知名开源工具, 如 CERT BFF 和 Honggfuzz, 它们在对测试输入集进行分类时, 都是基于软件崩溃信息对测试输入进行的分类, 并将其工具迭代至今. CERT BFF 在利用堆崩溃点处的函数调用堆栈进行哈希时, 可以配置计算哈希的堆栈层数. Honggfuzz 默认使用崩溃点处 7 层的函数调用堆栈计算哈希, 进而通过查看测试输入触发的不同崩溃对其进行分类.

1.2 基于程序修复的崩溃分类

还有一些研究者基于程序修复的方法对测试输入进行分类. Chen 等人[13]在解决软件崩溃重复度高的问题时, 提出了一种将修复补丁与机器学习相结合的方法. 首先, 对模糊测试工具输出的测试输入集进行排序, 并以修复补丁作为标准答案, 最后针对软件运行时的信息, 如覆盖信息、指令信息等来计算崩溃之间的相似度. Tonder 等人[5]提出了一种基于程序修复的崩溃输入分类算法——语义崩溃分类, 对空指针解引用和缓冲区溢出引发的崩溃进行分类. 他们首先利用修复模板自动化地修复程序, 以消除程序中存在的此类漏洞; 然后重新生成工程, 观察哪些测试输入在程序修复后在运行时不再崩溃, 则将这些测试输入分为一组. Pham 等人[14]的工作也利用了语义信息, 他们将崩溃分类与符号执行相结合, 提出了一种关注于测试输入的执行路径、通过将输入的语义特征作为路径约束的聚类算法.

1.3 崩溃分类后在软件工程中应用

当崩溃被分类后, 分类后的集群除了可以简化开发人员的工作之外, 许多研究人员更深一步地研究了其更多的应用场景. Castelluccio 等人[15]提出, 可以通过对分类后的集群中的特征进行识别, 来帮助开发者理解崩溃信息. Qian[16]利用分类后的崩溃集群信息为 Facebook 的崩溃修复进行指导与协助. Khomh 等人[17]将崩溃分类应用在实际工程中, 对于 Firefox 收集到的崩溃进行分类后, 又对其进行优先级排序, 帮助开发团队决定哪些崩溃需要优先修复. Kim 等人[18]对于 Firefox 软件和 Thunderbird 软件收集到的崩溃进行了深入分析, 发现大部分崩溃报告只对应少量顶级崩溃(top crash), 优先修复这些崩溃有利于大幅度地减少崩溃报告的数量, 并基于此提出了一种通过旧的顶级崩溃特征预测新版本中顶级崩溃的机器学习方法. Wu 等人[19]提出了 ChangeLocator, 通过控制流分析和程序切片研究软件崩溃报告, 并结合代码特征, 从而精确地预测哪些提交可能会引入崩溃, 并辅助开发人员找到解决方案. Guo 等人[20]在 ChangeLocator 的基础上进行改进, 并提出了 ChangeRanker, 其对在原工作的基础上崩溃特征进行了过滤, 并实现了更优秀的性能.

2 一种结合动态链接库信息的崩溃输入分类算法

2.1 问题背景

本节将以一个工程为例, 描述 CICELY 的提出背景与流程.

AFL 对开源项目 sqlite[21]中的可执行文件 lt-sqlite3 进行模糊测试时, 2 小时产生了 144 条导致崩溃的测试输入. 由于 AFL 对冗余测试输入的判断是基于路径的, 如果某个测试输入在执行的过程中多进入了一次循环或进入了分支条件中的新的分支, 即便它可能并没有对软件崩溃点处的变量内容、调用堆栈造成任何影响, 它的分类策略依然会将其视为一个新的测试输入. 这是因为 AFL 的运行目的主要在于提高测试输入的覆盖率, 尽可能多地找到之前没有测试输入执行过的路径.

然而, 通过这样的分类方法而分类出的测试输入在被手动执行和验证时, 依然会有大量的冗余出现. 因为在运行导致崩溃的测试输入时, 最关键的信息仍是软件崩溃的点和导致软件崩溃的点. 如果两个测试输入只是在导致软件崩溃的点之前有某些执行路径上的不同, 但是从导致软件崩溃的点到软件崩溃时的执行过程完全一样, 那么只需要分析和修正其中一个测试输入, 便可将这两个测试输入全修正. 我们在对程序 lt-sqlite3 做测试输入的执行时便遇到了这个问题, 即不同测试输入所对应的崩溃的重复度太高, 分析时遇到了大量冗余, 大大影响了研究效率. 因此, 在此问题的背景下, 我们提出了 CICELY.

2.2 基于软件崩溃信息的测试输入分类方法

本文针对基于软件崩溃信息的分类算法 Honggfuzz 和 CERT BFF 进行了改进. 传统的基于软件崩溃信息的分类方法对导致程序崩溃的测试输入进行分类的思路是: 首先确定崩溃的唯一性, 再通过观察测试输入执行时触发了哪个崩溃, 从而对测试输入进行分类. 由于程序在崩溃时会抛出异常并立即停止运行, 因此我们可以通过观察程序在崩溃时的环境来对崩溃进行分析. 对于同一个崩溃, 不同的测试输入在触发时所遇到的异常必然是相同的, 而触发崩溃的行为也是相同的; 反之, 如果崩溃的行为不同, 那么崩溃的原因势必也是不同的. 例如: 当程序对 NULL 进行指针的解引用时, 会导致异常和程序崩溃; 进行算术运算时, 除 0 也会导致程序异常和崩溃. 但是这二者一定不会在同一条指令处出现, 也不会对应同一个崩溃, 所以不同崩溃的触发行为是不同的. 因此, 可以通过观察程序在崩溃点处的执行状态、变量内容、指令信息对崩溃的唯一性进行指认.

现有的基于软件崩溃信息的测试输入分类算法主要分为 4 个步骤.

(1) 将所有会导致程序崩溃的测试输入逐个执行到待测程序中, 等待程序发生崩溃;

(2) 对于每个测试输入, 通过专门的工具查看程序在崩溃时的状态, 如执行结束的指令地址、指令内容、变量信息等;

(3) 通过哈希算法, 对查看到的程序崩溃时信息进行哈希, 以唯一性地标识此处的崩溃;

(4) 当所有测试输入全部执行并触发崩溃后, 通过崩溃处的唯一性标识对测试输入进行分类.

然而, 即便是在某个固定程序位置触发的程序崩溃, 其函数访问的路径也有可能是不同的. 所以一些分类工具, 如 CERT BFF, 在对导致程序崩溃的测试输入分类时提供了可以手动设置的接口, 以控制工具在进行分类时计算的函数堆栈层数.

2.3 通过程序崩溃时的动态链接库地址确定用户函数的地址范围

通常, 开发人员在修复程序时更多地会关注其工程内部的代码逻辑, 而对于系统层面的运行情况的关注度却不高. 因此当软件在运行过程中遇到了崩溃问题时, 尽管有时其实际崩溃点在于系统库和系统函数上, 开发人员依然倾向于在崩溃点向上寻找堆栈, 查看在运行其项目相关的语句时哪里导致了崩溃. 也就是说, 系统库函数往往不是开发人员“关注”的代码. 本文受此启发, 在程序崩溃时查看其所在的地址: 如果处于系统函数范围, 则工具会在崩溃点处持续向上寻找堆栈, 直到其地址落在用户关注的函数范围内为止.

CICELY 在确定待测程序的用户函数的地址范围时, 我们首先查看二进制文件的静态地址, 确定用户代码在二进制文件中加载的相对地址的范围. 此外, 有些工程会在其工程的目录内存放一些与其项目有关的动态链接库, 这些库往往不像是存放在系统文件夹(如/lib 或/usr 目录)下的那些由其他开发者提供的库, 而是该项目组的相同成员所提供的工具库. 基于该观察, 本文将用户关注的代码划分为用户代码以及用户自身的动态链接库代码, 这些代码都可能是开发者对崩溃进行修复时所关注的内容. 所以本文算法在判断用户函数时, 同时考虑了项目用户自研代码的地址以及项目内提供的用户库的地址. 我们会在程序的运行过程中获取其动态链接库加载的地址范围, 再通过这些动态链接库对应的物理路径, 便可以确定此库是否属于用户关注的函数的范围.

2.4 算法流程(以项目sqlite为例)

由于模糊测试工具 AFL 可以在测试的过程中生成和输出大量导致软件崩溃的测试输入, 因此, 我们选择了项目 sqlite 中的二进制文件 lt-sqlite3 作为待测程序, 并针对此程序, 用 AFL 生成了测试输入集来描述 CICELY 的流程, 算法期望的输出为测试输入集的分类结果. 对于每一个测试输入, CICELY 执行的流程概况如图 1 所示.

图 1 中, 用户代码指待测程序中的代码, 动态链接库分为系统动态链接库和用户动态链接库. 系统动态链接库指保存在系统文件夹中, 通常, 其存放地址以/lib, /usr 等开头; 而用户动态链接库是开发人员自己提供的动态链接库, 一般会存放在本工程的某些目录下. 我们将用户代码和用户动态链接库统称为用户关注的代码.

如图 1 所示, CICELY 会对于每一个测试输入逐个地进行分析: 首先, 静态地识别出程序中用户代码加载的地址范围; 然后运行程序, 执行将导致程序崩溃的测试输入, 期间, 程序在执行时将加载其运行时所需的动态链接库. 当程序发生崩溃时, 获取其崩溃点的地址, 判断此地址是否在用户关注的代码的地址范围内: 如果在, 则直接记录该行的地址与指令信息; 如果不在, 则说明目前崩溃点的位置在系统动态链接库的地址范围内, 则跳转到上一层堆栈, 以此作为新的崩溃点继续判断, 直到崩溃点的位置落在用户关注的代码的地址范围内为止. 算法将为每个崩溃输入都进行这样的执行与判断, 最后根据记录的信息对测试输入进行聚类.

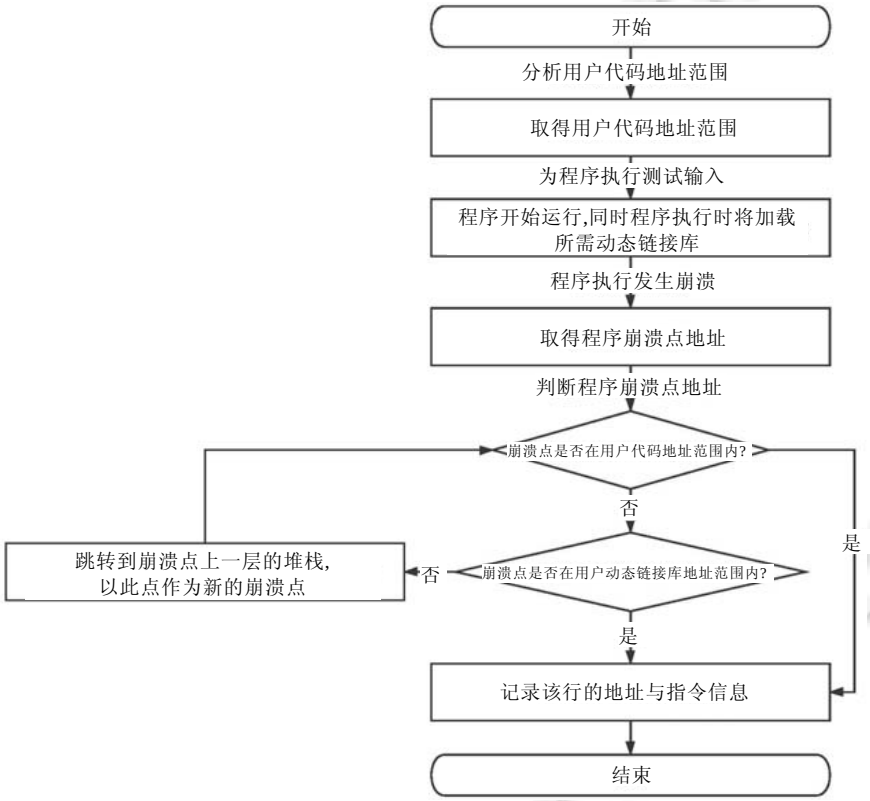

图 1 CICELY 的流程图

具体而言, CICELY 主要分为以下 6 个步骤.

(1) 分析用户代码地址.

在本例中, 对于每一个测试输入, CICELY 在待测程序 aac2mp4 开始执行前先静态地对程序进行执行前的用户代码地址范围分析, 找出待测程序中用户代码的起始地址与结束地址, 用于在后续的分析过程中区分用户函数范围. 我们挑选了一个测试输入(本例中为 id:000122, src:000038+000093, op:splice, rep:2)进行执行, 其返回结果如图 2 所示.

根据图 2 中框所示内容, 我们可以得知: 在执行该测试输入时, 用户函数的起始地址是 start_code 处的 0x400000, 终止地址是 end_code 处的 0x4106ac.

(2) 基于调试工具触发崩溃, 记录崩溃点位置信息.

本例中, 基于 pwndbg 工具, 在 pwndbg 中, 通过 file 指令指定将要运行的待测程序 libming, 通过 run 指令以及合适的参数以执行测试输入. 因为测试输入是由 aflgo 生成的一定会导致程序崩溃的程序输入, 所以在执行文件时会触发程序崩溃. 由于程序在 pwndbg 中执行, 所以会在程序发生崩溃时中断. 程序在中断时输出的内容如图 3 所示.

```
host mmap_min_addr=0x10000
Reserved 0x213000 bytes of guest address space
Relocating guest address space from 0x0000000000400000 to 0x400000
guest_base  0x0
start            end              size             prot
0000000000400000-0000000000411000 0000000000011000 r-x
0000000000610000-0000000000613000 0000000000003000 rw-
0000004000000000-0000004000001000 0000000000001000 ---
0000004000001000-0000004000801000 0000000000800000 rw-
0000004000801000-0000004000824000 0000000000023000 r-x
0000004000824000-0000004000a23000 00000000001ff000 ---
0000004000a23000-0000004000a26000 0000000000003000 rw-
start_brk   0x0000000000000000
end_code    0x00000000004106ac
start_code  0x0000000000400000
start_data  0x0000000000610dd0
end_data    0x000000000061202c
start_stack 0x00000040007fff40
brk         0x0000000000612198
entry       0x0000004000802260
argv_start  0x00000040007fff48
env_start   0x00000040007fff58
auxv_start  0x0000004000800148
```

图 2　待测程序中的 page 信息

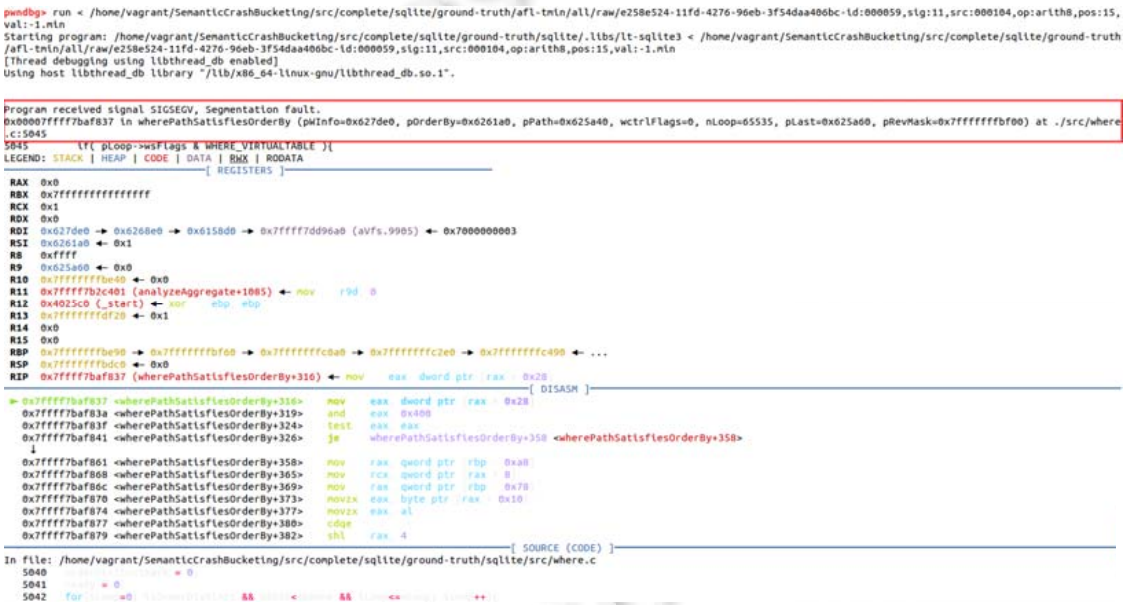

图 3　程序在执行中断后在 pwndbg 中展示的信息

在图 3 中，框中的信息表示程序在执行到地址为 0x00007ffff7baf837 的地方发生了崩溃，其他则为 pwndbg 工具输出的用于辅助分析崩溃情况的信息. 此类信息可以在人工分析时，协助开发人员了解程序崩溃的具体情况.

(3) 区分用户动态链接库函数和系统动态链接库函数.

通过前面获得的用户代码的地址范围，结合用户关注的函数的动态链接库地址，以继续确定哪些动态链接库属于用户函数的范围.

本步骤需要在步骤(2)之后运行，原因是动态链接库的加载是在程序运行启动后，在加载动态链接库之后才能进行动态链接库的地址识别. 本例中，在确定动态链接库的地址范围时，我们用到了 pwndbg 的 libs 指令. 因为当程序在通过 run 输入文件并正式启动后，才表示程序真正地加载到了内存中，此时，其运行过程中的动态链接库也才有了地址，所以只有程序开始运行后，pwndbg 才可以通过 libs 指令查找程序执行过程中的动态链接库地址. pwndbg 的 libs 指令会将待测程序 libming 的所有动态链接库全部输出，其中会有系统函数的动态

链接库和用户函数的动态链接库. 本例中, libs 指令的执行结果如图 4 所示.

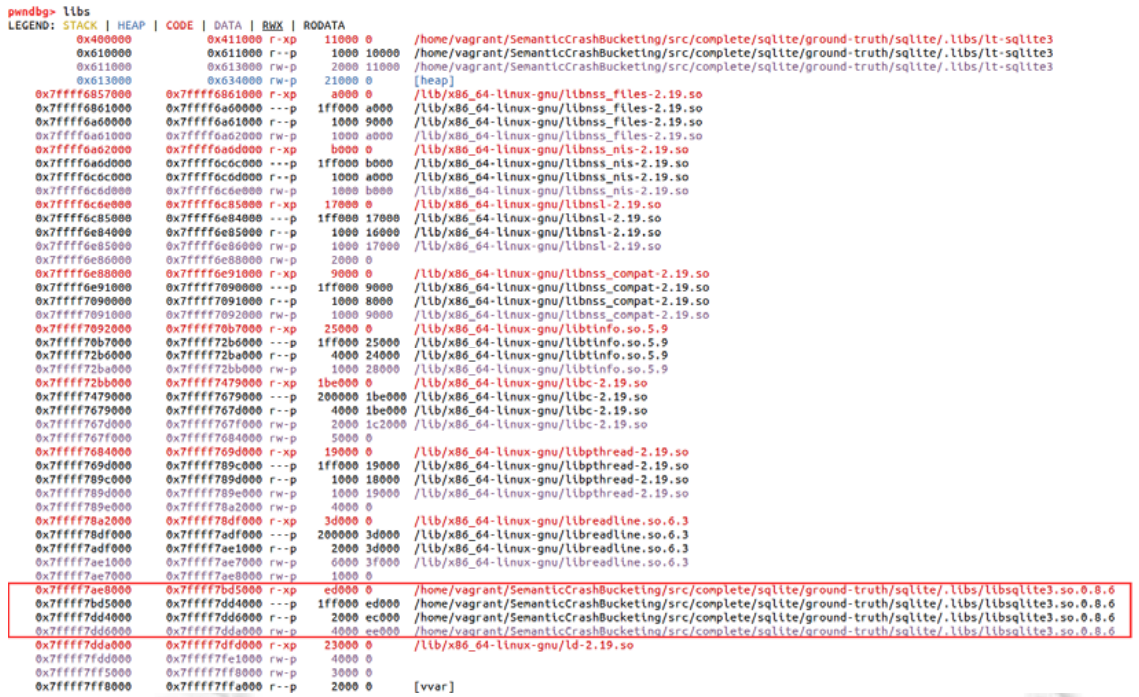

图 4 libs 指令的执行结果

为区分用户动态链接库函数和系统动态链接库函数, CICELY 使用了启发式的判定方法. 一般而言, 系统函数的动态链接库会保存在系统文件夹中, 通常以/lib, /usr 等开头; 而用户函数的动态链接库会保存在待测工程所在的文件夹内. 因为 libs 指令会输出程序所加载的动态链接库名称和地址范围, 所以只需要识别动态链接库的文件位置便可以确定哪些动态链接库属于用户函数, 哪些属于系统函数.

如图 4 中框处所示, 本例中, 程序中的用户动态链接库为

/home/vagrant/SemanticCrashBucketing/src/complete/sqlite/ground-truth/sqlite/.libs/libsqlite3.so.0.8.6.

因此, 将记录其地址, 划入用户关注的代码地址范围内.

(4) 识别用户函数.

步骤(1)获得了用户代码的地址范围, 步骤(3)识别了用户动态链接库, 将这两部分的代码一起划作为用户函数的地址范围.

(5) 程序崩溃位置判定.

当确定待测程序的用户函数的地址范围后, 判断程序崩溃位置是否在用户函数的范围内. 具体步骤为: 如果发生崩溃的地址没有在用户函数的范围内, CICELY 将通过 pwndbg 中的 up 1 指令向上寻找函数的调用堆栈, 然后以上层函数的地址作为新的判断地址; 如果依然没有落入用户函数的地址范围内, 则继续向上寻找调用堆栈, 直到其地址落入用户函数的范围内为止. 本工具将以此处的指令地址和指令内容拼接起来, 作为唯一性的判断标准.

对于本测试而言, 0x00007ffff7baf837 的地址在用户关注的代码的范围内, 所以直接记录本行所对应的地址和指令. 当前位置对应的指令可以通过 x/i $pc 指令得出:

```
pwndbg> x/i $pc
=> 0x7ffff7baf837 <wherePathSatisfiesOrderBy+316>:      mov    eax,DWORD PTR [rax+0x28]
```

图 5 待测程序在 0x00007ffff7baf837 地址处对应的指令

图 5 表示当前的指令是 0xf7fdb430 〈__kernel_vsyscall+16〉: pop ebp, 因此将此信息记录下来, 作为此崩溃的唯一性标识, 即, 此测试输入对应的就是这个软件崩溃.

(6) 对所有测试输入分类.

对于每一个测试输入, 重复执行上述步骤(1)~步骤(5), 对每一个测试输入都记录其执行触发崩溃时所对应的唯一性标识. 当所有测试输入全部执行结束后, 将唯一性标识相同的测试输入划为同一组, 表示他们所对应的软件崩溃是同一个崩溃. 开发人员分析其中的任意一个, CICELY 认为相当于将同组的测试输入全部分析.

至此, CICELY 将待测程序的导致崩溃的测试输入集根据其触发崩溃的触发原因分为了不同组, CICELY 的执行步骤结束.

而对于传统的基于软件崩溃信息的测试输入分类方法而言, 例如 CERT BFF, 它与 CICELY 相比, 不同之处在于 CERT BFF 没有使用程序的动态链接库信息, 因为也没有识别用户函数, 而是直接使用了崩溃点的信息. 当测试输入运行到发生程序崩溃时, CERT BFF 会直接观察和记录崩溃点的堆栈信息, 并结合一些其他信息, 如程序当前运行的行号等, 作为此崩溃的唯一性标识.

因此, CICELY 对于同类算法的改进主要在于: 在标识崩溃时结合了程序的动态链接库信息, 并利用动态链接库中的地址范围对用户函数进行识别, 以期使分类算法在分类时拥有更高的准确度.

2.5 基于程序修复的测试输入分类方法

基于程序修复的测试输入分类算法在对崩溃进行分类时, 其流程与基于软件崩溃信息的分类算法有较大差异. 以 SCB 在处理空指针解引用类型崩溃时的过程为例, 其主要步骤如下.

(1) 首先, 人为地定义一份针对空指针解引用的通用的修复模板. 例如, SCB 在解决空指针解引用崩溃时定义的修复模板为: 在出现解引用的代码的前一行插入一条判断, 如果判断此发现指针的内容是 NULL, 便直接退出程序, 以避免其执行后面的解引用操作;

(2) 在会导致程序运行出现空指针解引用崩溃的测试集中, 任选一个测试输入并执行, 当运行到发生崩溃时, 查看其在源码中的对应代码, 找出进行了解引用的指针, 对其应用步骤(1)中定义好的修复模板进行修复. 例如: 当发现某行代码的解引用会触发空指针解引用崩溃后, 在代码中的此行前插入步骤(1)中所述的修复模板, 这样便避免了程序再在此处运行时, 因空指针解引用而发生崩溃;

(3) 对于修复后的程序源码, 按照其原本的生成方式重新进行生成, 产生修复后的程序二进制文件;

(4) 在会导致原程序运行出现空指针解引用崩溃的测试集中, 令测试集中的所有测试输入依次在修复后的程序中执行. 在新程序中执行时, 没有再使程序因空指针解引用而发生崩溃的测试输入将被分为一组;

(5) 在未被分组的测试输入中任选一个测试输入, 重复上述步骤(2)~步骤(4), 直到所有测试输入全部都被分组为止.

通过以上步骤, 算法完成了对测试输入的分类.

然而, 因为本类算法需要对程序进行修复和再生成, 所以只能在程序源码的层级上对测试输入进行分类, 而无法直接基于二进制文件对测试输入进行分类; 而基于软件崩溃信息的分类方法则不受这方面的限制, 其分类的过程与程序源码无关, 可以直接对二进制文件的输入进行分类. 而且, 因为程序的修复模板是人为定义的, 当定义完修复模板后, 程序将自动化地对程序进行修改, 所以其分类质量与修复模板的质量、开发人员编码的习惯都有很大关联. 除此之外, SCB 在其论文中也指出, 其算法目前只能对于空指针解引用和缓冲区溢出这两种崩溃类型进行修复, 说明其覆盖的崩溃种类也较为有限.

3 工具实现

本文在实现算法时主要使用了 Python 语言(工具原型和实验结果可从 https://github.com/wehann/CICELY 下载), 并在 Ubuntu 14.04 环境下进行了测试. 算法在执行过程中使用了 subprocess 库, 以执行命令行的指令; 使用了 gdb 库, 以使得算法在 Python 运行过程中调用 gdb 并执行 gdb 的各种指令.

在对于程序中的用户代码的范围进行分析时, 使用了基于 qemu 实现的 trace 追踪工具找出待测程序中用户代码的起始地址与结束地址. 由于 Ubuntu 系统自带的 gdb 无法直接查看待测程序的动态链接库, 因此我们使用了 pwndbg[22], 通过其 libs 指令查看待测程序的动态链接库的加载情况.

4 实验设计

为衡量崩溃输入分类工具的能力, 需从准确性和有效性两个维度对分组结果进行评判. 准确性代表分组结果能否将每个测试输入正确地分到其对应的组内, 衡量该工具让开发和测试人员漏掉模糊测试报告的某种类型的崩溃的风险; 有效性代表分组的结果中每组代表的崩溃的重复程度, 衡量该工具让开发和测试人员查看不同崩溃所节省的时间成本. 为了全面评估改进后的对导致软件崩溃的测试输入进行分类的算法是否在真实数据集上有较好的分类效果, 本文设计了以下 3 个研究问题(RQ), 对 CICELY 与一些流行的分类算法和工具进行了对比和分析. 因为 Honggfuzz 和 CERT BFF 在工业和研究领域的使用较为广泛[23−25], 并均可以对测试输入集进行分类; 而 SCB 是最近提出的一种基于程序修复的测试输入分类工具, 并在其论文[5]中和 Honggfuzz 和 CERT BFF 进行了比较, 提到其效果优于 Honggfuzz 和 CERT BFF, 所以我们以这 3 种算法和工具作为对比算法设计了实验.

- RQ1: 在真实工程中, CICELY 与其他同类的基于软件崩溃信息的分类算法相比, 分类有效性如何?
- RQ2: 对于 SCB 论文中所实验的 21 例崩溃, SCB 在其提供的数据集下的分类结果与 CICELY 相比, 分类有效性如何?
- RQ3: 对于 SCB 论文中的实验数据集, 当同一项目中的不同崩溃对应的测试输入被合并到一起时, CICELY 和 SCB 能否将不同的崩溃区分开? 区分后的准确性和有效性如何?

4.1 实验数据集

AFL 作为一个经典的模糊测试工具, 其在官方网站上提供了大量由此工具检测出的崩溃, 并展示了崩溃对应的项目与链接. 其涉及了各种类型的软件, 如浏览器 Mozilla Firefox、Apple Safari、远程连接软件 PuTTY、网络抓包工具 wireshark 等, 因此, AFL 官网中提供的软件类型比较广泛, 其崩溃具有一定的代表性. 因此, 对于 RQ1 的实验, 为了对不同的测试输入分类算法在实际工程中进行分类的效率对比, 本文在 AFL 的网站上随机挑选了若干个 GitHub 开源项目进行测试. 具体做法为: 首先, 在 AFL 官网上随机挑选若干个崩溃, 对于每个崩溃对应的项目进行模糊测试; 然后, 对 AFL 输出的模糊测试结果进行人工分析. 为了降低实验误差, 在其中选取若干个崩溃的触发原因相同的测试输入进行测试输入库扩充(corpus generation). 本文在测试输入库扩充时, 使用的方法与 SCB 相同, 均是模糊测试工具 AFL 提供的崩溃模式. AFL 的崩溃模式只需要提供一个可以使待测程序产生崩溃的测试输入种子(seed), 便可以让其生成大量与种子的执行路径相同的测试输入. 因为我们在每个项目中挑选的崩溃对应的触发原因是相同的, 所以我们将同一个项目中每个崩溃生成的测试输入库合并到一起, 作为这个项目的测试输入集. 对于 RQ1 而言, 每个测试集中包含了大量会使程序运行发生崩溃的测试输入, 且这些输入对应的崩溃是相同的.

对于 RQ2 的实验, 为了直观地与 SCB 算法进行对比, 我们选用了 SCB 论文中的 6 个项目、21 个崩溃作为实验数据集. 这 21 个崩溃与项目的对应关系为: 项目 SQLite 中有 12 个崩溃, 崩溃类型为空指针解引用; 项目 w3m 中有 4 个崩溃, 崩溃类型为空指针解引用; 项目 PHP 中有 2 个崩溃, 其中一个是 PHP-5, 一个是 PHP-7, 崩溃类型均为空指针解引用; 项目 R 中有 1 个崩溃, 崩溃类型为缓冲区溢出; 项目 Conntrackd 中有 1 个崩溃, 崩溃类型为缓冲区溢出; 项目 libmad 中有 1 个崩溃, 崩溃类型为缓冲区溢出. SCB 生成测试集的具体步骤为: 首先, 针对每个崩溃, 利用 AFL 的崩溃模式生成测试输入库; 然后, 针对每个测试输入库, 分别使用 afl-tmin、在崩溃点处根据 5 层堆栈信息做哈希的 CERT BFF 算法(记作 BFF-5)、在崩溃点处根据 1 层堆栈信息做哈希的 CERT BFF 算法(记作 BFF-1)、默认配置下的 Honggfuzz(记作 HFuzz)和基于反馈驱动的 Honggfuzz(记作 HFuzz-Cov)对测试输入库逐个进行了分类; 然后, 以每个工具的分类结果作为 SCB 的测试输入集, 使用 SCB 在测试输入集上运行并记录 SCB 的分类结果. 作为与 SCB 的对比实验, 我们在 RQ2 中使用了与 SCB 相同的

数据集及策略进行测试, 即: 以这 5 个工具对测试输入库进行分类的结果作为测试输入集, 对其结果再次进行分类. 在 RQ2 中, 我们直接选择了 SCB 的测试集进行测试.

对于 RQ3 的实验, 我们选择了 SCB 的实验测试集中崩溃个数不为 1(为了避免混淆, 在本文中, 判断多个崩溃是否属于“同一个崩溃”的依据是崩溃发生的根本原因(root cause). 具有相同根本原因的多个崩溃输入应被划分为一组, 其对应的崩溃在本文中称为“一个崩溃”. 而判断多个崩溃是否属于“一种崩溃”, 将通过崩溃的崩溃类型来判断, 如空指针解引用等)的项目, 即 SQLite 和 w3m. 我们将其中所有崩溃对应的测试输入合并在一起, 作为一个新的测试集. 与 RQ2 不同的是: 我们在 RQ2 中使用的是与 SCB 相同的测试集, 即测试集中包含的是测试输入只对应了一个崩溃; 而 RQ3 中的测试集对应的是多个崩溃, 项目 SQLite 的测试集对应了 12 个崩溃, 项目 w3m 的测试集对应了 4 个崩溃.

4.2 评测指标

当分类工具对完成测试集的分组后, 在最理想的情况下, 每组结果中的测试输入对应的崩溃应是相同的, 即所有测试输入应当对应着同一个崩溃; 每组结果代表的崩溃应互不相同, 且数量之和恰等于测试集中测试输入对应的崩溃的总数. 然而, 在实际的分组结果中, 每组结果中的测试输入对应的崩溃可能不唯一, 而且每组结果代表的崩溃可能会有重叠, 每组结果代表的崩溃的并集也不一定能覆盖测试集中所有崩溃的种类.

SCB 在其论文中的评估标准是, 统计每个工具在分组结果中分组出现重复的测试输入个数. 因在其实验数据中, 每组测试的测试集对应着同一个崩溃, 所以在其实验条件下, 分类工具不需要判断分组结果中每一组结果对应哪个崩溃. 然而在本文的 RQ3 中, 我们在分析实验数据时, 需要先为分组结果中的每一组划定其所代表的崩溃, 而 SCB 在其论文中没有提到这一点. 此外, 在上一段的描述中, 我们对于“分组出错”的情况提到了多种场景, 仅凭 SCB 的评估标准很难对于各种情况尽数表达. 因此, SCB 的标准并不适用于本文提出的 RQ. 针对以上问题, 我们提出了以下指标以评估分组结果.

(1) 当分组结束后, 首先为每一组结果划定本组所代表的崩溃. 该崩溃的选择按以下优先顺序确定.

① 该崩溃所对应的测试输入的数量占本组的多数;

② 如果至少有 2 个崩溃对应的测试输入数量为同样多数, 则选择的崩溃应能使得本分类算法的覆盖度更高(覆盖度定义见下);

③ 如果仍未选取到相应的崩溃, 则随机选取一个测试输入数量为同样多数的崩溃, 作为本组所代表的崩溃.

当分组结果中的每一组都确定其所代表的崩溃后, 本组内的所有测试输入可以以其各自对应的崩溃为标准, 通过比较测试输入对应的崩溃与本组代表的崩溃是否相同, 来判断本测试输入是否被划分到了正确的组中.

(2) 分别从准确性和有效性这两个维度对分组结果进行评判: 准确性用准确度表示, 有效性用重复度表示. 其中, 我们在为准确度进行衡量时, 又将其分为了覆盖度和纯度两部分: 覆盖度用于反映分组的结果对于所有崩溃的覆盖程度, 纯度用于反映每个组中有多少不同类型的崩溃的测试输入.

① 评估分组的准确性.

- 覆盖度(coverage, 简记为 C): 计算去重后的组的崩溃类别个数/崩溃类别的总数.
- 纯度(purity, 简记为 P): 首先, 对于每一组, 计算(本组代表的崩溃在本组对应的测试输入个数/该组中的测试输入总数), 然后再对于每一组的计算结果取平均.

定义. 准确度(accuracy, 简记为 A)=覆盖度×纯度.

② 评估分组的有效性.

定义. 重复度(repeatability, 简记为 R)=不同组的崩溃类别重复的次数/总组数. 这里, 重复的次数指的是多余的组的个数. 例如在分组结果中, 有 3 个组对应了同一个崩溃, 则重复次数为 2.

因为 SCB 在其论文中统计的是每个工具在分组结果中分组出现重复的测试输入个数, 仅在有效性上对于不同的分类工具进行了衡量; 而本文评估标准还额外衡量了准确性, 覆盖了 SCB 论文中的评估标准.

在实际应用中, 分组算法首要保证的应当是准确性, 即分组结果中每一组的测试输入对应的崩溃是相同的, 否则可能导致重要的崩溃类型被分到错误的组中, 使开发和测试人员因使用崩溃输入分类工具而漏看该崩溃. 所以应满足保证准确性的基础上, 尽可能提高分组的有效性. 即保证准确度=100%(覆盖度=100%、纯度=100%)的基础上, 尽可能降低重复度. 理想情况下, 重复度=0%.

此外, 因为在 RQ1 和 RQ2 中, 我们每一组测试的测试集的组成与 SCB 实验中的是相同的, 即每一组测试集中的测试输入对应的崩溃个数为 1, 所以其覆盖度和纯度一定是 100%, 所以其覆盖度、纯度和准确度均为 100%, 所以无法从准确性这一个维度上进行检验. 同样因为其测试集的特殊性, RQ1 和 RQ2 在计算重复度时, 只需统计分组结果的组数 n, 便可以直接通过以下公式直接计算出这两个 RQ 中的重复度:

$$R=\frac{n-1}{n}.$$

这是因为重复度的计算方式为: 不同组的崩溃类别重复的次数/总组数. 而在 RQ1 和 RQ2 的每组测试中, 崩溃的个数仅为 1, 那么不同组的崩溃类别重复的次数一定就是总组数减 1. 也就是说, 通过统计分组结果的组数 n, 便可以直观地反映出 RQ1 和 RQ2 中分组的结果. 同样也因如此, 我们在统计 RQ1 和 RQ2 的实验结果时, 没有使用 SCB 的实验评估标准. 因为 SCB 的实验评估标准统计的是分组出现重复的测试输入个数, 而这一数据很难像组数这样直观地与重复度进行关联并换算. 因为上述原因, 同时为了减少表格篇幅, 所以我们在 RQ1 和 RQ2 中仅记录了分组结果中的组数作为实验数据. 而因为 RQ3 的测试集相比 RQ1 和 RQ2 更复杂, 所以我们将以实验数据为基础, 从准确性和有效性这两方面对不同的分类算法进行评估.

因为 RQ1 和 RQ2 记录的实验数据是分组结果中的组数, 所以在 RQ1 和 RQ2 中, 当两个算法的实验结果之间需要进行对比时, 假设实验结果中组数数量较大的记为 RB(单位: 组数), 实验结果中数量较小的记为 RS(单位: 组数), 我们定义分类结果中多分的比例 *Ratio* 为

$$Ratio=\frac{RB-RS}{RS}.$$

4.3 实验流程

在 RQ1 的实验中, 我们在 AFL 官网上随机挑选了共 11 个项目, 首先对这 11 个项目都使用 AFL 逐个进行了模糊测试; 然后, 在模糊测试的结果中, 我们为每个项目都选择了 4 个崩溃原因相同的输出文件作为进行测试输入库扩充时的种子(其中, binutils 项目因其在 AFL 中运行 2 小时后的输出结果只有 3 个文件, 所以只选择了 3 个输出文件), 每个种子通过 AFL 的崩溃模式运行 2 小时; 最后, 我们将每个项目的 3 组或 4 组测试输入集混合到一起作为一组大的测试输入集, 用于此项目的测试.

在 RQ2 的实验中, 我们直接使用了 SCB[5]的数据集. 因为 SCB 提供了 6 个项目, 共包含 21 个崩溃, 所以我们在 RQ2 中的测试集与之相同. 实验方法与 SCB 在论文中的实验方法相同, 均为分别在 5 种工具的分类结果的基础上运行分类算法.

在 RQ3 的实验中, 我们选择了 SCB 的实验测试集中崩溃个数不为 1 的 2 个项目, 将其中所有崩溃对应的测试输入合并在一起, 作为一个新的测试集. 实验方法为: 在测试集上分别使用 SCB 和 CICELY 进行分类, 记录分类结果的准确度和重复度, 以比较二者的准确性和有效性.

综上所述, 每个 RQ 对应的测试输入组数见表 1.

表 1 每个 RQ 对应的测试输入组数

测试种类	测试项目数	测试总组数
RQ1	11	11
RQ2	6	21
RQ3	2	10
总计	19	42

在本文的实验中, SCB 的配置直接使用了其在论文[5]与开源项目[26]中提供的代码与修复模板进行测试, Honggfuzz, CERT BFF 和 CICELY 的运行方法均为其默认运行方法.

对于每组测试输入集, 本文将分别运行该测试对应的测试输入分类算法, 并对分类结果进行记录和对比, 实验结果见第 5 节.

5 实验结果

5.1 RQ1的实验结果及分析

RQ1 中, 每个项目的测试结果见表 2.

表 2 RQ1 的实验结果

项目名	项目中的被测文件名	测试编号	测试集大小	Honggfuzz	CERT BFF	CICELY	种子的崩溃原因
Libav-11.8	avconv	1	2 498	1327	6	7	越界读取
Libtiff-4.0.7	Tiffcp	2	104	3	2	1	违反断言
Mozjpeg-2.1	Cjpeg	3	417	144	6	6	堆缓冲区溢出
Bento4-1.5.0-617	aac2mp4	4	624	1	1	1	堆缓冲区溢出
Lrzip-0.631	Lrzip	5	289	32	24	14	内存释放后使用
Pcre-8.40	Pcretest	6	2 614	11	5	4	堆缓冲区溢出
Binutils-2.29.1	nm-new	7	3 966	831	339	87	内存双重释放
pax-utils-1.2	Dumpelf	8	985	1 922	7	6	堆缓冲区溢出
Potrace-1.3	Potrace	9	1 848	0	44	47	堆缓冲区溢出
Ytnef-1.9.2	Ytnefprint	10	24	4	7	2	堆缓冲区溢出
Bento4-1.5.0-617	mp42aac	11	3 274	18	15	19	堆缓冲区溢出
总计			16 643	4 293	456	194	堆缓冲区溢出

根据第 4.2 节中对于评估标准的描述, 因为在 RQ1 和 RQ2 的实验中, 分类结果中的组数是可以直接与重复度进行换算的, 且限于篇幅, 所以我们此处直接将组数作为实验数据进行记录. 组数越小, 代表分组结果中的重复度越低.

从表 2 我们可以得知: 在随机挑选的崩溃类型中, 缓冲区溢出所导致的崩溃占了大多数. 然而作为基于软件崩溃信息的测试输入分类算法, 不管是 CICELY 还是另外两种工具, 对于此类型的崩溃的分析能力都不强, 我们将在第 5.2 节中对此类型的崩溃进行详细分析.

对于 RQ1 中的每组测试, 其测试集中的所有测试输入对应的崩溃都是同一个, 所以经过分类算法处理后, 组数应越小越好. 整体而言, Honggfuzz 比 CICELY 在分类结果上多分的比例为 2112.89%, CERT BFF 比 CICELY 在分类结果上多分的比例为 135.05%, 说明 CICELY 仍然对另外两工具的结果得到了一定程度上的提升.

尽管算法的基本思想类似, 且同属于基于软件崩溃信息的分类算法, 但是几种工具在其原理上仍有些不同之处: CICELY 在对崩溃点信息进行分析的基础上增加了对于动态链接库的分析; CERT BFF 在对崩溃点信息进行哈希时会对行号进行模糊化, 即认为在崩溃点附近的崩溃都属于同一个崩溃; Honggfuzz 的默认配置在判定崩溃的唯一性时, 还会考虑溢出的堆栈中的数据.

在表 2 中, Honggfuzz 在对编号为 9 的项目 potrace 中的测试输入进行分类时, 其分类结果为 0. 经过手动检查, 我们确保了指令无误、测试输入可以导致软件崩溃、工具配置正常的条件, 且另外两种算法均能在 potrace 上产生结果, 但 Honggfuzz 的分类结果依然是 0. 因此我们认为, Honggfuzz 是因其自身原因无法在 potrace 上进行分类, 这反映了 Honggfuzz 可能在某些情况下无法正常地对测试输入集进行分类.

整体而言, CICELY 在对同一崩溃进行分类时, 比 Honggfuzz 和 CERT BFF 分出的结果更少, 说明 CICELY 的效果相比其他二者更好, 能更容易将同类的测试输入分在一起. 然而在个别崩溃的检测结果中, 仍然存在 CICELY 的结果比 CERT BFF 或 Honggfuzz 差的情况. 在我们检测的 11 组测试输入中, 有编号为 1, 9, 11 共 3 组测试集的分类结果差于 CERT BFF, 但是在这几组中, CICELY 的分类结果比 CERT BFF 的分类结果的数量只多于 5 以内, 说明尽管 CICELY 在某些测试集上的效果与 CERT BFF 相比有些许差距, 但是差距不大. 而与 Honggfuzz 相比, 对于大部分测试集而言, CICELY 的分类效果是优于 Honggfuzz 的, 只有在编号为 11 的测试集中, CICELY 的效果差于 Honggfuzz. 由于本节中进行对比的方法都属于基于软件崩溃信息的分类方法, 所

以限于本类方法的共同特性, 我们难以对随机选取的每一个项目的测试输入都归为同一类. 因此, 我们的关注点在于同类型算法之间分类结果的差异性.

在所有项目的结果中, 我们首先注意到了 3 种工具对于编号为 7 的项目 binutils 进行分类时的分类结果数量都远超过其他项目, 这样的测试结果在理论上是不合理的. 我们对其测试输入进行深入分析后发现: 测试输入在经过 AFL 的崩溃模式后, 测试输入库中出现了大量其他类型的崩溃, 即测试输入集受到了污染, 不同输入的崩溃原因不一致, 因此这里属于 AFL 崩溃模式的缺陷. 然而尽管如此, CICELY 在分类时的分类结果依然低于其他两种工具. 限于有限的资源, 尚未对每个测试输入进行崩溃的人工分类, 因此尚不清楚实际的分类个数, 但考虑到 Binutils 2.29.1 是被充分测试的程序, 在 AFL 围绕崩溃点按照崩溃模式产生的大量测试输入中, 实际崩溃分类个数很可能小于 87, 因此我们倾向于认为 CICELY 在处理特殊情况时同样具备较好的效果.

尽管 CICELY 在对测试输入进行分类时, 效果差于其他两种工具的情况不多, 且差的程度不大, 但我们仍将以一个例子分析 CICELY 的分类结果差于其他工具的原因. 在编号为 11、项目 bento4-mp42aac 的崩溃中, CICELY 的效果比 Honggfuzz 和 CERT BFF 都略差. 经过分析, 我们发现差异之处在于: 测试输入集在执行时, 会有多处不同的用户函数在调用某系统级的动态链接库后的崩溃点被聚集在了动态链接库的某相同位置的情况产生, 因此, Honggfuzz 和 CERT BFF 直接以此处的堆栈信息做了哈希. 由于 CICELY 只考虑用户函数, 所以在这种情况下便会向上寻找堆栈, 而找到用户函数层级时, 发现其对应了多个不同的调用点. 而因为另外两种工具直接以程序最底层的崩溃点的信息做哈希, 相当于将 CICELY 分类结果中的不同的组聚到了一起, 所以最终 CICELY 的分类结果的数量大于 Honggfuzz 和 CERT BFF.

5.2 RQ2的实验结果及分析

RQ2 的结果如表 3–表 5 所示, 其中, 表 3 和表 4 是 SCB 和 CICELY 在 SCB 所提供的每个测试集上逐个运行后的结果, 限于篇幅, 本文将测试结果分为“AFL-tmin, BFF-5, BFF-1”和“HFuzz, HFuzz-Cov”两部分, 分别对应表 3 和表 4; 表 5 是对于表 3 和表 4 的数据的总结, 直观地对于 SCB 与 CICELY 的性能进行对比.

如我们在第 4.3 节中所述, 我们在 RQ2 中实验的测试集是分别由测试工具 AFL-tmin, BFF-5, BFF-1, HFuzz 和 HFuzz-Cov 对测试输入库进行分类后的结果. 在表 3 和表 4 中, 以这 5 种工具命名的列下的 GT 代表的是实际测试集的大小, 其生成方法是令每个工具对测试输入库进行精简, 即代表在此工具的分类标准下, GT 中的每个测试输入都对应着一个崩溃, 且这些崩溃互不相同. 例如, AFL-tmin 下的 GT 就代表使用 AFL-tmin 对测试输入库进行精简后, 测试输入的个数. 此处 GT 的数字越小, 说明此工具的分类效果越好. GT+SCB 表示使用 SCB 在 GT 的结果中进行分类时, 分类结果中的组数; GT+CICELY 同理. 这里, 我们实际上沿用了与 SCB 相同的测试方法, 只是在评价标准上略有不同: SCB 在其论文中统计的是分组出现重复的测试输入个数; 而在 RQ2 中, 我们统计的是分组结果的组数.

表 3 RQ2 在 AFL-tmin, BFF-5, BFF-1 测试集上的实验结果

项目	测试编号	测试输入库大小 (个)	AFL-tmin			BFF-5			BFF-1			崩溃类型
			GT	GT+SCB	GT+CICELY	GT	GT+SCB	GT+CICELY	GT	GT+SCB	GT+CICELY	
SQLite	12	191	26	1	1	3	2	2	2	2	2	①
	13	482	86	1	1	3	1	1	2	1	1	
	14	153	39	1	1	7	1	1	1	1	1	
	15	326	49	1	1	1	1	1	1	1	1	
	16	139	35	1	1	1	1	1	1	1	1	
	17	66	22	1	1	1	1	1	1	1	1	
	18	97	21	1	1	1	1	1	1	1	1	
	19	235	83	1	1	2	1	1	1	1	1	
	20	389	30	1	1	2	1	1	1	1	1	
	21	270	66	1	1	1	1	1	1	1	1	
	22	167	46	2	2	1	1	1	1	1	1	
	23	108	37	1	1	1	1	1	1	1	1	
总计		2 623	540	13	13	24	13	13	14	13	13	

表 3 RQ2 在 AFL-tmin, BFF-5, BFF-1 测试集上的实验结果(续)

项目	测试编号	测试输入库大小 (个)	AFL-tmin			BFF-5			BFF-1			崩溃类型
			GT	GT+SCB	GT+CICELY	GT	GT+SCB	GT+CICELY	GT	GT+SCB	GT+CICELY	
w3m	24**	458	104	1	1	26	1	1	2	1	1	①
	25	545	24	1	1	1	1	1	1	1	1	
	26	507	37	1	1	1	1	1	2	1	1	
	27	525	12	1	1	1	1	1	2	1	1	
总计		2 035	177	4	4	29	4	4	7	4	4	
PHP	28	81	9	1	1	1	1	1	1	1	1	①
	29	272	33	1	1	1	1	1	1	1	1	
R	30	7	6	1	1	4	1	1	1	1	1	②
Conntrackd	31	25	1	1	1	1	1	1	1	1	1	②
libmad	32	138	9	1	1	2	1	1	1	1	1	②
总计		5 181	775	22	22	62	22	22	26	22	22	

**本文与 SCB 的论文在编号 24 处软件崩溃的认定存在不同:

- 崩溃类型①: 空指针解引用;
- 崩溃类型②: 缓冲区溢出

表 4 RQ2 在 HFuzz, HFuzz-Cov 测试集上的实验结果

项目	测试编号	测试输入库大小 (个)	HFuzz			HFuzz-Cov			崩溃原因
			GT	GT+SCB	GT+CICELY	GT	GT+SCB	GT+CICELY	
SQLite	12	191	11	2	1	10	2	2	①
	13	482	5	1	1	3	1	1	
	14	153	17	1	1	15	1	1	
	15	326	2	1	1	1	1	1	
	16	139	1	1	1	1	1	1	
	17	66	1	1	1	1	1	1	
	18	97	1	1	1	1	1	1	
	19	235	4	1	1	4	1	1	
	20	389	2	1	1	2	1	1	
	21	270	2	1	1	2	1	1	
	22	167	5	3	2	2	2	1	
	23	108	1	1	1	1	1	1	
总计		2 623	52	15	14	43	14	13	
w3m	24**	458	76	1	2	78	1	2	①
	25	545	1	1	1	1	1	1	
	26	507	7	1	1	5	1	1	
	27	525	1	1	1	1	1	1	
总计		2 035	85	4	5	85	4	5	
PHP	28	81	1	1	1	1	1	1	①
	29	272	1	1	1	1	1	1	
R	30	7	146	1	3	199	1	5	②
Conntrackd	31	25	771	1	1	428	1	1	②
libmad	32	138	2	1	1	1	1	1	②
总计		5 181	1 058	24	25	758	23	27	

**本文与 SCB 的论文在编号 24 处软件崩溃的认定存在不同:

- 崩溃类型①: 空指针解引用;
- 崩溃类型②: 缓冲区溢出

表 5 SCB 与 CICELY 结果对比

测试工具	SCB 分类结果	CICELY 分类结果
AFL-tmin	22	22
BFF-5	22	22
BFF-1	22	22
HFuzz	24	25
HFuzz-Cov	23	27
总计	113	118

根据表 5 我们可以得知: 与基于程序修复的分类工具 SCB 相比, CICELY 在 SCB 所提供的数据集上分类结果仅比其结果多 5 组, 比 SCB 差 4.42%. 然而, 因为 SCB 在其论文中指明其只能对空指针解引用和缓冲区

溢出这两种漏洞进行分类; 而在第 5.1 节中的实验结果中, CICELY 已覆盖了比以上两种漏洞更多的漏洞类型. 这表示 CICELY 可以在达到与 SCB 的实验效果相当的基础上, 支持更多的漏洞类型. 除此之外, 理论上, CICELY 可以适用于所有类型的漏洞.

在项目 SQLite 中, CICELY 的效果优于 SCB. 两个工具在分类时出现了分类过多的崩溃只有编号 12 和 22 处的崩溃. 经过分析, 我们发现这两处的崩溃在执行过程中会导致不同的崩溃行为, 因此其崩溃点也是不同的, 所以两种方法在分类时未能完全地将这些崩溃的测试输入正确分类.

在项目 w3m 中, CICELY 在编号 24 处的崩溃处会分成 2 组. 经分析, 我们发现编号 24 处对应的崩溃是一个带有 3 个指针变量的结构体, 其中的指针发生了空指针解引用. 在运行测试输入时, 我们发现测试输入集中对应着两种路径的测试输入: 在第 1 种测试输入下, 结构体中的 3 个指针变量都是空指针, 在解引用第 3 个指针变量时发生了空指针解引用, 从而导致软件崩溃; 在第 2 种测试输入下, 结构体中的第 3 个指针变量有内容, 另外两个指针变量是空指针, 在解引用第 2 个指针变量时发生了空指针解引用, 从而导致软件崩溃. 我们认为: 这两种情况下虽然都是空指针解引用, 且引发软件崩溃的原因也极为相似, 但是仍应属于两个不同的软件崩溃, 因为二者执行路径和产生崩溃的位置与变量都是截然不同的, 因此理应将编号 24 对应的软件崩溃的测试输入集分为两组.

在编号为 30 的测试, 即项目 R 中, CICELY 在分类时有分类过多的现象发生. 这是因为项目 R 中的崩溃是由缓冲区溢出引起的, 而 CICELY 作为基于软件崩溃信息的分类算法, 本类在分析缓冲区溢出时的分析效果均有所不足. 这是因为缓冲区溢出的特点是: 其崩溃点与崩溃的实际导致点往往可以相隔很远, 导致其分析起来比较困难. 基于软件崩溃信息的算法会着重分析调用栈顶的信息, 而出现因缓冲区溢出而导致的崩溃时, 其函数调用堆栈很可能与其溢出点无关, 因为这类崩溃在缓冲区溢出发生的时候很可能不会出现崩溃, 而会在使用因缓冲区溢出而被覆盖的内存时出现程序崩溃.

在项目 PHP, Conntrackd, libmad 中, CICELY 的效果与 SCB 相同, 将每组测试的测试集都正确地分到了同一组中.

5.3 RQ3的实验结果及分析

RQ3 的实验结果见表 6.

表 6 RQ3 的实验结果

项目	测试编号	测试输入库大小 (个)	测试集生成方法	测试集大小 (个)	SCB				CICELY			
					C (%)	*P* (%)	*A* (%)	*R* (%)	*C* (%)	*P* (%)	*A* (%)	*R* (%)
SQLite	33	2 623	AFL-tmin	540	100	100	100	8.33	100	100	100	8.33
	34	2 623	BFF-5	24	100	100	100	8.33	100	100	100	8.33
	35	2 623	BFF-1	14	100	100	100	8.33	100	100	100	8.33
	36	2 623	HF	52	100	100	100	25	100	100	100	16.67
	37	2 623	HF-cov	43	100	100	100	16.67	100	100	100	8.33
w3m	38	2 035	AFL-tmin	177	100	100	100	0	100	100	100	0
	39	2 035	BFF-5	29	100	100	100	0	100	100	100	0
	40	2 035	BFF-1	7	100	100	100	0	100	100	100	0
	41	2 035	HF	85	100	100	100	0	100	100	100	25
	42	2 035	HF-cov	85	100	100	100	0	100	100	100	25
平均					100	100	100	7	100	100	100	10

注: *C*: 覆盖度 Coverage; *P*: 纯度 Purity; *A*: 准确度 Accuracy; *R*: 重复度 Repeatability

从第 5.2 节 RQ2 的实验数据和分析中我们可以得知: SCB 在进行每一组实验时, 首先生成了一组测试输入库; 然后, 分别用不同的分类工具对测试输入库进行分组; 最后, SCB 再在每个工具分组结果的基础上进行分组. 因此, 为了直观地与 SCB 进行对比, 在 RQ3 中, 我们也直接使用了 SCB 所提供的测试集, 且实验方法与 SCB 类似, 同样先分别用不同的分类工具对测试输入库进行分组, 最后再令 SCB 和 CICELY 分别在每个工具分组结果的基础上进行分组. 而 RQ3 与 RQ2 的不同之处在于: 我们将每个工具在同一项目、不同编号上分组的结果混合在一起组成了新的测试集; 并且在 RQ3 中, 我们以每个工具分组后又混合而成的测试输入的集

合作为每次测试的测试集. 由于项目 SQLite 中共包含了 12 个崩溃, 项目 w3m 中共包含了 4 个崩溃, 因此, 测试编号 33–39 中, 每个测试集中的测试输入都对应了 12 个崩溃; 测试编号 38–42 中, 每个测试集中的测试输入都对应了 4 个崩溃. 在测试结果中, 我们将根据分组后的组内和组间数据计算分组的准确度和重复度.

从表 6 的实验结果中我们可以得知, CICELY 与 SCB 在准确度上均为 100%, 说明在测试集的结果中, 两种分类算法都没有在分组结果中将不同崩溃的测试输入混在一起, 保证了每组结果对应崩溃的唯一性.

在重复度的结果上, 整体而言, 两种分类算法的实验效果较为接近, CICELY 的平均重复度仅比 SCB 差 3%. 在 SQLite 项目的结果中, CICELY 的重复度略低于 SCB; 在 w3m 项目的结果中, CICELY 的重复度略高于 SCB. 这样的实验数据与 RQ2 实验结果中的表 3、表 4 的实验数据完全吻合. 其中, 在编号为 33–35, 38–40 的测试中, CICELY 与 SCB 的实验数据相当. 在编号为 36, 37 的测试中, CICELY 的重复度低于 SCB 的重复度, 这分别对应了表 4 中崩溃编号为 22 的两组数据. 在编号为 41, 42 的测试中, CICELY 的重复度高于 SCB 的重复度, 这分别对应了表 4 中编号为 24 的两组数据. 因上述实验结果已在 RQ2 的实验结果分析中有过陈述, 故此处不再赘述.

RQ3 的实验结果表明: 在处理由多个崩溃的测试输入所组成的测试集时, CICELY 依然可以较好地对不同崩溃进行分组, 并且可以达到与基于程序修复的分类算法 SCB 相当的水平.

5.4 讨 论

本文使用的测试集较为有限, 所以实验结果有缺乏代表性的风险. 为了尽可能地消除这个风险, 我们使用 SCB 论文中的数据集, 并使用 AFL 的典型崩溃样例作为扩展, 且数据集规模大于之前所有的相关工作, 尽可能保证了实验结果的可靠性.

可以观察到: CICELY 在 RQ1 中出现分组过多问题时, 其多分的程度远超过 RQ2 中出现分组过多问题时多分的程度. 这主要与测试集本身的测试输入及其崩溃类型有关. 我们的实验结果表明: CICELY 作为一种基于软件崩溃信息的分类算法, 其与同类算法 Honggfuzz, CERT BFF 相比有很大改进; 而与基于程序修复的分类算法 SCB 对比时, CICELY 依然可以达到相近的水平, 并且在其他方面比 SCB 更有优势. 正如我们在第 2.5 节介绍基于程序修复的分类算法的流程时时所述, SCB 对测试输入进行分类的原理是: 首先, 对崩溃提供一个修复模板; 然后, 在程序的运行路径中寻找格式符合修复模板格式的点, 将其修改后再重新构建文件. 对于新生成的可执行文件, SCB 将分别执行测试输入: 如果测试输入没有再触发崩溃, 则说明此测试输入属于当前修复的这一崩溃; 反之, 则说明此测试输入没能被正确分类. 因其分类原理与其他 3 种分类工具均不同, 而作为基于程序修复的分类算法, SCB 在实际应用中有这样的局限性.

(1) SCB 的运行需要提供修复模板, 并且只能在程序源码的层级上对测试输入进行分类. 因为程序的修复模板是人为定义的, 当定义完修复模板后, 程序将自动化地对程序进行修改, 所以其分类质量与修复模板的质量、开发人员编码的习惯都有很大关联. 一般而言, 一个项目通常是会由项目组内许多开发人员共同完成的, 而不同开发人员的编码风格不尽相同, 因此修复模板很难保证对于不同风格的代码都能生效. 除此之外, 因为 SCB 对测试输入分类的方法是依赖于修复模板的, 而对于任意一处崩溃, 即便是开发人员也很难保证此崩溃在修复后便永不会出错, 不会再需要二次修复. 因此,不仅修复模板的确定是一件困难的事, 而且修复模板的质量也与 SCB 的分类效果有着密切的联系.本文无法保证能为崩溃提供完美的修复模板, 所以对于 SCB 的实验及其效果也难以界定. 在对 RQ2 的研究中, 使用的是 SCB 工具自带的模板, 并复现了其实验;

(2) SCB 在实际场景下的可扩展性较有限. SCB 在论文中指出, 只能对空指针解引用和缓冲区溢出两种崩溃进行分类. 然而在实际工程场景下, 崩溃类型远不止这两种. 除此之外, 当开发人员收集到大量崩溃输入时, 没有预先设好的修复模板可提供的. 如果想要先找到几个测试输入, 先修复再分类,会影响崩溃的修复效率. 因为对测试输入分类的目的就是在于降低开发人员修复崩溃时的冗余工作, 而如果要“先修复、再分类”, 那冗余的分析工作同样是难以避免的.

除此之外, 在 RQ1 中, 与同类型的分类算法进行比较时, 因为每一组测试集都只对应着一个崩溃, 而由于 CERT BFF 在对测试输入进行分类时会对崩溃点信息进行哈希时会对行号进行模糊化, 所以这样的模糊化

一定会使分得的组数不变或减少. 但在实际应用中, 这样的模糊化却存在对分类结果造成分类过少的风险. 在分类结果少于真实结果的情况下, 没有被报告出来那些崩溃会因被开发人员忽略而没能被修复; 而在分类结果多于真实结果的情况下, 额外地分析一组崩溃只是会为开发人员带来一些多余的时间开销, 不会导致崩溃未修复的后果. 所以二者相比, 我们认为当开发人员对崩溃进行修复时, 分类结果的数量少于真实结果带来的影响是远大于分类结果的数量多于真实结果的. 因此, 我们没有使用这样的模糊化方法.

本文在区分系统和用户的动态链接库函数时, 使用了基于特定路径的启发式方法. 对于不同的操作系统、编译环境, 可以根据其特点形成常见的系统库路径集合. 同时, 本文对用户关注的代码的划分是经验式的. 在实际中, 可能存在一些系统库仍被用户关注的场景. 例如: 在对库进行模糊测试和漏洞挖掘时, 开发及测试人员可能会关注哪个系统库会引起崩溃, 并针对性地进行分类分析. 此外, 也可能有一些用户动态链接库并不被用户关注, 应作为系统库函数对待. 在实际应用中, 通过工具提供用户接口, 使开发人员在进行崩溃输入分类前可以指定用户或系统库路径, 将能进一步提升 CICELY 的准确度.

由于限于 CICELY 的核心方法在于对崩溃点的信息进行分析, 因此其效果在由缓冲区溢出引发的崩溃上会有较差的效果. 为了减轻其对 CICELY 在分类时的影响, 我们考虑在未来的工作中引入动态插桩工具(如 Valgrind), 在缓冲区溢出发生时的溢出点(即缺陷的触发点)自动触发崩溃, 以减少在分析缓冲区溢出时的分类过多现象. 但是由于 Valgrind 的处理会也会对算法执行带来额外的开销, 其会使分析速度降低 10 到 50 倍[27], 因此, 在引入 Valgrind 的同时保证 CICELY 的高效性, 也将是未来考虑的问题之一.

6 总结与展望

本文对从大量导致崩溃的输入中对输入进行分类这一问题设计了算法 CICELY, 并对其进行了实验验证和分析. CICELY 在现有的基于软件崩溃信息的基础上, 又通过动态链接库识别用户函数与系统函数, 当崩溃发生在系统函数中时, 因为开发者关心的和最能直观感受的是用户函数, 所以 CICELY 会在堆栈中向上寻找用户函数, 然后再应用堆栈哈希的思想, 大大提升了原有的分类效果. 实验结果显示: CICELY 在与同类的基于软件崩溃信息的分类算法 Honggfuzz 和 CERT BFF 相比时, 大大提升了同类算法的性能; 而与基于程序修复的算法 SCB 相比, 我们也能达到与之相近的表现, 并且在其他方面相比 SCB 有更多优势.

然而, 实验结果也暴露出了 CICELY 的一些问题. 尽管本算法整体而言比同类型的其他工具的效果更好, 但是在某些测试集中, CICELY 的分类结果依然不如 Honggfuzz, CERT BFF. 除此之外, 限于基于软件崩溃信息的分类算法的特性, CICELY 对于缓冲区溢出导致的崩溃的分析能力还不够强. 在未来的工作中, 我们将综合考虑实验中遇到的分类过多的情况, 对工具进行进一步改进, 以尽可能地提高分类的正确性. 我们还会考虑通过不同手段优化算法的执行策略, 如采用并发的方式执行待测程序, 以尽可能地提高工具的运行效率. 此外, 我们还会尝试在尽可能不对分类效果产生太大影响的情况下引入动态插桩工具, 以增强算法对于缓冲区溢出所引发的崩溃的分析能力.

References:

[1] Kim J, Wrote; Ye LL, Trans. Fatal Bugs: Disasters and Revelations from Software Defects. Beijing: Post & Telecom Press, 2014 (in Chinese).

[2] Klees G, Ruef A, Cooper B, *et al*. Evaluating fuzz testing. In: Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. 2018. 2123–2138.

[3] CERT bff. 2021. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=507974

[4] Google/Honggfuzz: Security Oriented Software Fuzzer. Supports evolutionary, feedback-driven fuzzing based on code coverage (sw and hw based). 2021. https://github.com/google/Honggfuzz

[5] Van Tonder R, Kotheimer J, Le Goues C. Semantic crash bucketing. In: Proc. of the 33rd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2018. 612–622.

[6] American fuzzy lop. 2021. https://lcamtuf.coredump.cx/afl/

[7] Böhme M, Pham VT, Nguyen MD, *et al*. Directed greybox fuzzing. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. 2017. 2329–2344. [doi: 10.1109/TSE.2019.2941681]

[8] Pham VT, Böhme M, Santosa AE, *et al*. Smart greybox fuzzing. IEEE Trans. on Software Engineering, 2021, 47(9): 1980−1997.
[9] Dhaliwal T, Khomh F, Zou Y. Classifying field crash reports for fixing bugs: A case study of mozilla firefox. In: Proc. of the 2011 27th IEEE Int'l Conf. on Software Maintenance (ICSM). IEEE, 2011. 333−342.
[10] Kim S, Zimmermann T, Nagappan N. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In: Proc. of the 2011 IEEE/IFIP 41st Int'l Conf. on Dependable Systems & Networks (DSN). IEEE, 2011. 486−493.
[11] Dang Y, Wu R, Zhang H, *et al*. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In: Proc. of the 2012 34th Int'l Conf. on Software Engineering (ICSE). IEEE, 2012. 1084−1093.
[12] Golagha M, Lehnhoff C, Pretschner A, *et al*. Failure clustering without coverage. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. 2019. 134−145.
[13] Chen Y, Groce A, Zhang C, *et al*. Taming compiler fuzzers. In: Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2013. 197−208.
[14] Pham VT, Khurana S, Roy S, *et al*. Bucketing failing tests via symbolic analysis. In: Proc. of the Int'l Conf. on Fundamental Approaches to Software Engineering. Springer, 2017. 43−59.
[15] Castelluccio M, Sansone C, Verdoliva L, *et al*. Automatically analyzing groups of crashes for finding correlations. In: Proc. of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017. 717−726.
[16] Qian R, Yu Y, Park W, *et al*. Debugging crashes using continuous contrast set mining. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering: Software Engineering in Practice. 2020. 61−70.
[17] Khomh F, Chan B, Zou Y, *et al*. An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox. In: Proc. of the 2011 18th Working Conf. on Reverse Engineering. IEEE, 2011. 261−270.
[18] Kim D, Wang X, Kim S, *et al*. Which crashes should i fix first? Predicting top crashes at an early stage to prioritize debugging efforts. IEEE Trans. on Software Engineering, 2011, 37(3): 430−447. [doi: 10.1109/TSE.2011.20]
[19] Wu RX, Wen M, Cheung SC, *et al*. ChangeLocator: Locate crash-inducing changes based on crash reports. Empir. Softw. Eng., 2018, 23(5): 2866−2900.
[20] Guo ZQ, Li YH, Ma WWY, *et al*. Boosting crash-inducing change localization with rank- performance-based feature subset selection. Empirical Software Engineering, 2020, 25(3): 1905−1950.
[21] SQLite home page. 2021. https://www.sqlite.org/index.html
[22] Pwndbg. 2021. https://github.com/pwndbg/pwndbg
[23] Public vulnerabilities discovered using bff-tools-vulwiki. 2021. https://vuls.cert.org/confluence/display/tools/Public+Vulnerabilities+Discovered+Using+BFF
[24] OSS-fuzz-google's continuous fuzzing service for open source software. 2021. https://google.github.io/oss-fuzz/
[25] Woo M, Cha SK, Gottlieb S, *et al*. Scheduling black-box mutational fuzzing. In: Proc. of the 2013 ACM SIGSAC Conf. on Computer & Communications Security. 2013. 511−522.
[26] SquaresLab/Semanticcrashbucketing. 2021. https://github.com/squaresLab/SemanticCrashBucketing
[27] Valgrind documentation. Valgrind Documentation, 2021. https://www.valgrind.org/docs/manual/valgrind_manual.pdf

附中文参考文献:

[1] 金钟河, 著, 叶蕾蕾, 译. 致命 Bug: 软件缺陷的灾难与启示. 北京: 人民邮电出版社, 2016.

王文祥(1998－), 男, 硕士生, 主要研究领域为软件工程, 代码静态分析.

许可(1990－), 女, 博士, 讲师, 主要研究领域为数据挖掘, 非结构化数据分析.

高庆(1989－), 男, 博士, 助理研究员, 主要研究领域为软件分析, 漏洞检测.

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.