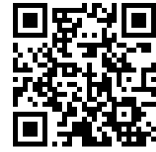
软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(7):3141−3161 [doi: 10.13328/j.cnki.jos.007105]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种聚类分析驱动种子调度的模糊测试方法*



张 文[1,2], 陈锦富[1,2], 蔡赛华[1,2], 张 翅[1,2], 刘一松[1]

[1](江苏大学 计算机科学与通信工程学院, 江苏 镇江 212013)

[2](江苏省工业网络安全技术重点实验室(江苏大学), 江苏 镇江 212013)

通信作者: 陈锦富, E-mail: jinfuchen@ujs.edu.cn; 刘一松, E-mail: liuyisong@ujs.edu.cn

摘 要: 作为当前被广泛应用的自动化软件测试技术, 模糊测试的首要目标是尽可能多地探索被测程序的代码区域以达到更高的覆盖率, 从而检测出更多的漏洞或者错误. 现有的模糊测试方法大多是根据种子的历史突变数据来调度种子, 实现起来比较简单, 但忽略了种子所探索程序空间的分布情况, 导致测试工作可能会陷入只对程序的某单一区域进行探测, 造成测试资源的浪费. 提出一种基于聚类分析驱动种子调度的模糊测试方法 Cluzz. 首先, Cluzz 结合种子执行路径覆盖的分布来分析种子在特征空间上的区别, 使用聚类分析对种子在程序空间中的执行分布情况进行划分. 然后, 根据不同种子簇群的路径覆盖模式与聚类分析结果对种子进行优先级评估, 探索稀有代码区域并优先调度评估得分较高的种子. 其次, 通过种子评估得分为种子分配能量, 将突变得到的有趣输入保留并进行归类以更新种子簇群信息. Cluzz 根据更新后的种子簇群重新评估种子, 以确保测试过程中种子的有效性, 从而在有限时间内探索更多的未知代码区域, 提高被测程序的覆盖率. 最后, 将 Cluzz 实现在 3 个当前主流的模糊器上, 并在 8 个流行的真实程序上进行大量测试工作. 结果表明: Cluzz 检测独特崩溃的平均数量是普通模糊器的 1.7 倍, 在发现新边缘数量方面, 平均优于基准模糊器 22.15%. 此外, 通过与现有种子调度方法进行对比, Cluzz 的综合表现要优于其他基准模糊器.

关键词: 模糊测试; 软件安全; 聚类分析; 种子调度; 能量分配
中图法分类号: TP311

中文引用格式: 张文, 陈锦富, 蔡赛华, 张翅, 刘一松. 一种聚类分析驱动种子调度的模糊测试方法. 软件学报, 2024, 35(7): 3141–3161. http://www.jos.org.cn/1000-9825/7105.htm

英文引用格式: Zhang W, Chen JF, Cai SH, Zhang C, Liu YS. Fuzzing Approach of Clustering Analysis-driven in Seed Scheduling. Ruan Jian Xue Bao/Journal of Software, 2024, 35(7): 3141–3161 (in Chinese). http://www.jos.org.cn/1000-9825/7105. htm

Fuzzing Approach of Clustering Analysis-driven in Seed Scheduling

ZHANG Wen[1,2], CHEN Jin-Fu[1,2], CAI Sai-Hua[1,2], ZHANG Chi[1,2], LIU Yi-Song[1]

[1](School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang 212013, China)

[2](Jiangsu Key Laboratory of Security Technology for Industrial Cyberspace (Jiangsu University), Zhenjiang, 212013, China)

Abstract: As a widely used automated software testing technique, the primary goal of fuzzy testing is to explore as many code areas of the program under test as possible, thereby achieving higher coverage as well as detecting more bugs or errors. Most of existing fuzzy testing methods schedule the seed based on the historical mutation data of the seed, which is simpler to implement but ignores the distribution of program space explored by the seed, resulting in that the testing may fall into only a single region of the program to be

* 基金项目: 国家自然科学基金(62172194, 62202206, U1836116); 江苏省自然科学基金(BK20220515, BK20202001); 中国博士后科学基金(2023T160275); 江苏省研究生科研与实践创新计划(KYCX21_3375, SJCX23_2092); 江苏省青蓝工程项目(2022JSDX001)
本文由“面向复杂软件的缺陷检测与修复技术”专题特约编辑张路教授、刘辉教授、姜佳君副研究员、王博博士推荐.
收稿时间: 2023-09-09; 修改时间: 2023-10-30; 采用时间: 2023-12-14; jos 在线出版时间: 2024-01-05
CNKI 网络首发时间: 2024-03-09

probed, and causing the waste of testing resources. This study proposes the Cluzz, a fuzzing approach of clustering analysis-driven in seed scheduling. Firstly, Cluzz analyzes the difference between seeds in the feature space by combining the distribution of seed execution path coverage, and uses cluster analysis to classify the distribution of seeds execution in the program space. And then, Cluzz prioritizes the seeds according to the path coverage patterns of different seed clusters and the results of cluster analysis, explores the rare code regions and prioritizes the seeds with higher evaluation scores. Secondly, energy is allocated to the seeds by their evaluation scores, and the interesting inputs obtained from mutations are retained and categorized to update the seed cluster information. Cluzz reevaluates the seeds based on the updated seed clusters to ensure the validity of seeds during testing process, thereby exploring more unknown code regions in a limited time and improving the coverage of the program under test. Finally, the Cluzz is implemented on three current mainstream fuzzers and extensive testing work is conducted on eight popular real-world programs. The results show that Cluzz can detect an average of 1.7 times more unique crashes than a regular fuzzer, and it also outperforms a benchmark fuzzer by an average of 22.15% in terms of the number of new edges found. In addition, compared with the existing seed scheduling methods, the comprehensive performance of Cluzz is better than that of other benchmark fuzzers.

Key words: fuzzing; software security; cluster analysis; seed scheduling; energy allocation

在软件安全测试领域中, 模糊测试(fuzzing)是其中一种被广泛应用的自动化软件测试技术, 用于暴露软件中的安全问题, 它已经应用到现实世界的程序测试中并识别出上千个错误和漏洞[1−4]. 其关键思想是: 通过自动化的方法对输入进行随机变异, 以不断生成新的测试输入, 从而得到涵盖程序潜在错误和异常情况的新的测试输入. 然后, 监控被测程序的运行状态, 以检测其是否存在漏洞、错误、异常行为或者崩溃等问题. 现如今, 大多数流行的模糊器通过进化搜索算法来生成新的测试输入, 这些进化搜索算法方法能够有效地引导模糊测试, 以探索被测程序中更多的未知代码区域. 在模糊过程中, 对一组种子输入进行突变的过程仅保留最有趣的新输入(即触发了新的程序状态的输入), 并在这些有趣输入基础上进行下一步的突变[5−7]. 最后, 利用突变得来的有趣测试输入, 在程序的输入空间中搜索潜在可利用的错误行为.

然而在现实中, 实际的程序输入空间往往很大, 通过随机变异方式无法穷尽地对其进行探索; 此外, 无引导的测试输入生成方式在探索被测程序错误方面也并不有效. 因此, 现有的模糊器大多采用边覆盖引导的进化方法来指导输入生成过程, 确保生成得到的新输入能够探索目标程序不同的控制流边[8,9], 以尽可能地探索程序的输入空间. 这种覆盖引导的灰盒模糊测试(coverage-guide greybox fuzzing, CGF)从一个种子输入语料库开始, 不断重复地从语料库中挑选一个种子对其进行突变, 只将那些产生了新的边缘覆盖的突变输入添加到种子语料库中, 以用于后续的模糊测试工作. CGF 的主要工作流程如图 1 所示. 这种模糊器的性能在很大程度上依赖于种子的前后调度顺序, 即选择要突变种子的先后顺序[10].

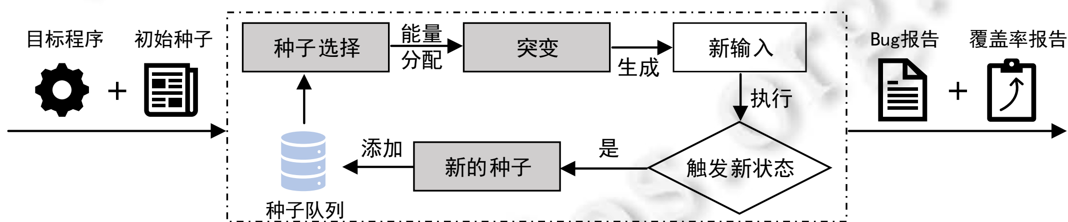


图 1　覆盖引导的灰盒模糊测试工作流程

随着人工智能的发展, 基于机器学习的技术也已经应用于模糊测试当中[11]. 基于机器学习的模糊测试方法通常使用现有的测试输入来训练一个模型, 通过训练得到的模型来指导模糊测试. 在定向灰盒模糊测试中, Fuzzguard[12]提出在执行测试输入之前通过神经网络模型预测测试输入的可达性, 过滤掉不可达的输入. 这种方法能够有效减少由于执行不必要测试输入所带来的资源浪费, 但模型训练所需要的时间成本过高. Neuzz[13]与 MTFuzz[14]利用神经网络模型来近似现实世界程序的分支行为, 通过神经网络的梯度信息去识别输入序列的关键字节来指导测试的突变, 并针对关键字节区域中字节的突变来提高测试覆盖率. SmartFuSE[15]提出了一种基于深度学习的符号执行与模糊测试的混合测试方法. 然而, 与其他所有的有监督学习任务相同, 这些方法在模型的训练上很大程度依赖于训练样本的数量与多样性. 此外, 在模糊测试中, 用于模型训练的数据

样本为测试输入, 而程序的输入空间往往很大, 测试输入的字节分布往往是稀疏的, 且存在输入样本不平衡现象, 导致模型的训练容易出现过拟合问题. 而且, 这些方法并没有考虑对突变种子选择的先后调度顺序是否会影响模糊器的工作效率.

通过优化 CGF 过程中的种子调度与变异算子选择是当前对模糊测试研究的两个关键问题. 合适的突变策略对模糊测试发现新的边缘覆盖和暴露崩溃方面有着很大的改善, 现有的变异调度方法根据历史突变数据来优化变异算子概率分布[16,17], 通过根据概率分布选择最优的突变策略. 对于种子优先级调度, 其主要难题是如何确定种子语料库中哪些种子突变所产生的新输入更有希望探索被测程序中尽可能多的新边, 从而对这些更有希望的种子进行更多的突变以获得更高的边缘覆盖. 一种简单的策略是根据种子执行路径在程序控制流程图(control flow graph, CFG)中所有可达边的数量来调度种子, 但这种理想的方法假设 CFG 中所有的边都是可以通过突变可达的. 然而, 现实世界的程序会存在复杂的约束条件, 难以通过随机突变的方式绕过约束, 因而这种理想化的策略不太可能有效. 先前的研究证据也表明: 通过对输入随机突变的方法, 到达子节点的难度通常比到达父节点更大[18], 即, 通过对输入字节随机突变的方式突破约束条件以到达新的边缘是很有难度的. 对于一些嵌入在较为简单的约束条件中的浅边, 可能可以通过大量的突变输入到达; 而对于嵌入在很多复杂约束条件中的深边, 在模糊过程中只能被少数的几个突变得到的新输入(如果有发生的情况)到达(因为许多分支可能是难以通过随机突变到达的). 另外一个理想的策略是, 根据一个种子可以通过突变到达的所有潜在可行边的数量来进行种子调度. 计算得到的潜在可行边的数量越高的种子, 应该被优先调度并且分配更多的能量. 然而, 计算所有种子到所有边的可达性是不切实际的, 一方面是因为计算将产生巨大计算成本; 另一方面是在程序控制流程图中, 某条边是否可达受很多因素的影响(如程序上下文因素、变量因素等). 所以, 边的可达性难以通过形式化的方式去衡量.

目前已有的关于种子调度的模糊测试方法主要基于种子历史突变产生的边缘覆盖数量来选择种子, 这些种子在历史突变过程中产生了更高的路径覆盖[19]; 或根据测试过程反馈信息对种子进行优先级排序, 以确定哪个种子是更有希望的种子, 例如种子执行触发了更稀有的边[20]. 通过使用图分析的中心性度量方法, 来近似种子到达未访问边的可能性来选择种子[21]. 然而, 种子的数量会随着模糊工作的进行而爆发性地增长, 这会造成大量的计算过程, 时间成本是不可预估的. 并且, 这些现有方法在测试过程中并没有考虑不同种子的执行路径在程序空间中的整体状态分布. 由于种子输入之间的差异性, 不同种子所探索的程序空间也有所不同. 相似的种子有着相似的输入模式以及执行行为, 种子之间的差异越大, 则所探索的程序空间越不相同. 由于忽略了不同类别种子探索区域的信息, 会使得探测容易陷入对程序空间中某单一区域的探索, 从而限制了对程序空间的全面探索能力. 其次, 在现有的模糊测试方法中, 大多是通过历史运行数据推断下一时刻的最优选择, 这种方式涉及大量的计算, 会导致高额的时间开销.

为了解决现有方法存在的问题, 本文提出了一种基于种子聚类分析的模糊调度方法, 并在此想法上实现了轻量级模糊测试框架 Cluzz. 我们根据种子的执行状态序列特征, 分析种子在程序空间中的分布情况, 通过聚类分析, 将种子分为若干个种子簇群, 其中相似的种子在特征空间上更接近, 而不同类别之间的种子在特征空间上有明显的区分. 由于不同种子簇群之间的路径探索区域有着明显差异, 因此通过执行程序空间某区域的种子分布权重占比来衡量代码区域的稀有度以及相较于全局的种子稀有度得分. 我们优先考虑能够探索程序稀有代码区域的测试输入和稀有种子. 在完成对种子的选取后, Cluzz 根据种子的稀有度得分为种子分配能量, 对稀有度得分越高的种子分配更多的能量, 确保能够全面探测未知代码区域. 在模糊测试过程中, 通过分析上下文中的种子选择情况来平衡调度各个种子簇群中的种子, 保证后代种子的多样性, 实现对程序路径探索的多样性和测试输入的多样化. 随着语料库中的种子数量的增加, 阶段性地重新评估队列中种子的优先级, 避免对无用种子模糊导致的测试时间和资源的浪费, 从而提高测试的效率与覆盖率.

本文的主要贡献包括:

(1) 提出了一种基于种子聚类分析的模糊调度方法, 通过分析不同类别之间的种子在特征空间上的区别, 并结合种子执行状态在程序空间中的分布情况, 来划分种子簇群, 用于指导模糊测试中的种子

© 中国科学院软件研究所 http://www.jos.org.cn

调度.

(2) 基于划分得到的种子簇群信息与种子执行信息, 提出了种子簇群权重与种子稀有度得分评估策略, 并将两种分数用于种子的能量分配工作.

(3) 基于以上方法实现了轻量级模糊测试框架 Cluzz, 并在当前 3 个主流模糊器 AFL、MOPT 和 NEUZZ 上实现并设计相关实验, 以证明我们方法的通用性与实用性.

本文第 1 节介绍种子调度、能量分配以及聚类分析等基础知识. 第 2 节介绍本文提出的 Cluzz 方法, 包括多重随机初始化聚类分析、种子优先级调度和能量分配. 第 3 节将本文提出的 Cluzz 方法集成到现有模糊器中, 并和基准模糊器进行实验对比. 第 4 节对有效性威胁进行分析. 第 5 节对相关工作进行总结. 第 6 节对本文的研究进行总结, 并给出未来展望.

1 基础知识

1.1 无监督学习-聚类分析

无监督学习中的聚类方法对于挖掘数据的结构和模式以及不同数据点之间的相似性和差异性至关重要. 这种将数据划分为不同类别的过程可以帮助我们发现数据的内在结构和模式, 从而更好地理解和分析数据. 目前, 已有多种聚类方法用于数据挖掘与数据聚类中[22−25]. 例如, K-means[23]是一种基于质心的聚类算法, 其目标是使得每个数据点都属于离其最近的簇中心点所对应的簇, k-means++[22]是它的一种改进算法. 此外, DBSCAN[24]是一种基于密度的聚类算法, 它在聚类前不需要事先指定聚类的数量. 对于我们的应用场景, 初始数据集相对较小并且需要通过对程序和初始种子的静态分析并通过迭代来确定聚类的数量, 因此不需要 DBSCAN 聚类的自适应性质. 并且 DBSCAN 算法的时间复杂度也较高, 所以使用不会产生过多时间开销(在第 3 节中通过实验对其证明)的 K-means 聚类算法会更加高效和可靠. 并且, 由于我们数据集中数据本身是通过执行输入而得到路径覆盖的输出结果, 所以数据本身是准确的, 无异常值, 噪声极小, 有利于 K-means 算法明确地划定簇群的边界, 在较短的时间内收敛到稳定的结果.

由于种子输入与执行状态序列是反映种子特征与状态分布情况的数据, 具有输入空间的高维与稀疏的性质, 而高维稀疏的数据会导致聚类算法难以识别数据间的实际距离. 同时, 高维稀疏数据也会增加算法的运行时间, 导致聚类算法难以选择出数据中最重要的特征, 从而影响聚类效果. 这显然不符合我们的期待, 因此需要在聚类之前对数据进行降维操作. 对于每个数据样本, 通过计算它与 K 个聚类中心之间的距离, 并将该样本分配给距离最近的聚类中心所在的簇. 对于每个样本 X_i, 计算它与每个聚类中心 C_j 的距离, 然后将该样本分配到距离最近的聚类中心所在的簇 S_j 中, 其计算过程如公式(1)所示.

$$S_j=\operatorname{argmin}_{j\in\{1,2,\ldots,K\}} dist(x_i,C_j) \tag{1}$$

每完成一次样本所属的簇(即 S_j)的计算后, 通过公式(2)重新计算每个簇的聚类中心 C_j.

$$C_j=\sum\nolimits_{x_i\in S_j} x_i \Big/ |S_j| \tag{2}$$

通过上面描述的步骤不断迭代, 直到所有数据所分配的簇编号不再发生变化, 则完成一次聚类操作. 由于不同的初始聚类中心可能导致不同的聚类结果, 而初始聚类中心的选取充满了随机性, 因此, 我们通过多随机初始化的方式来减小随机因素的影响, 避免算法陷入局部最优, 提高聚类的准确性和可靠性.

1.2 模糊测试中的种子调度优化

提高模糊测试效率的一个主要方式是种子调度, 即种子的选择顺序. 对模糊测试中的种子调度进行优化, 可以选择正确的种子以极大地提高模糊测试的能效[10]. 与之相比, 没有种子调度优化的模糊器(例如 MOPT[16])在读入初始测试输入时仅仅按照文件名称顺序将其加入种子队列, 且后续突变得到的新输入也没有通过某种方法对其进行调度选择. 模糊测试的进行会产生大量的新输入, 不同的输入可能在发现新的覆盖方面具有不同的潜力, 并且不同类别的种子有着截然不同的覆盖模式. 同样, 通过突变得到的大量新输入中也会存在对测试没有贡献的输入, 通过种子调度避免执行这些无用的输入, 可以提高测试效率. 因此, 选择适当

© 中国科学院软件研究所 http://www.jos.org.cn

的策略优先调度有效种子, 可以提高测试覆盖率并尽快发现更多的程序错误.

1.3 模糊测试中的能量分配优化

另一种提高模糊测试性能的方法是通过优化模糊过程中的能量分配模块[26–28]. 对于一般的模糊器(例如AFL[5]), 它们为每个种子分配几乎相同的能量. 然而, 不同的种子在发现新的覆盖方面有着不同的潜力. 因此, 在基于模糊测试规则的基础上, 覆盖引导的模糊测试方法[29]通过给那些更有可能发现新覆盖的种子分配更多的能量. 如何选取为种子计算所分配能量的指标以及如何计算, 是关键问题所在. 不合理的计算方式会导致能量的浪费, 或者导致种子突变不充分, 造成模糊器探测新边缘的能力出现下降. 在 AFL 中, 种子的执行速度与路径覆盖大小都是计算种子能量大小的指标. Truzz[30]提出了一种基于路径转换的能量调度优化方法, 它通过观察每个突变的结果来构建路径转换的概率分布, 识别并且保护与验证检查有关的字节不受突变, 将更多的能量将分配给那些非验证检查的字节, 从而更加充分地探索程序的功能代码.

2 一种聚类分析驱动种子调度的模糊测试方法

2.1 方法动机

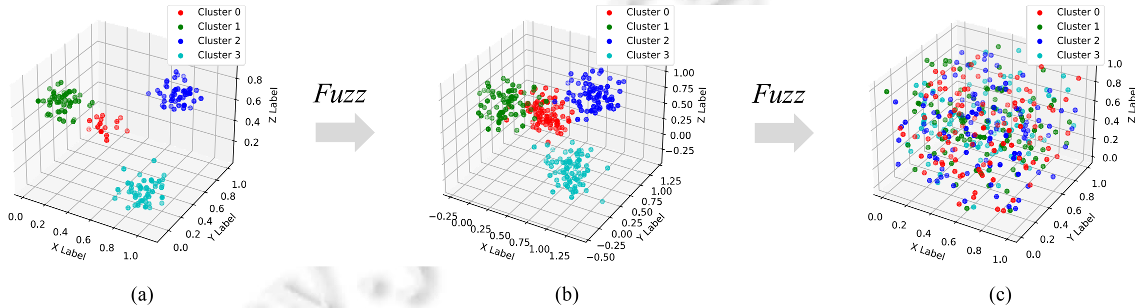
在现有的模糊测试方法中, FairFuzz[20]通过统计识别测试输入很少执行到的程序分支(该方法将 AFL[5]中的边的表示方式称为分支), 并将这些分支称为稀有分支. FairFuzz 通过在这些罕见的分支上生成更多的随机输入, 极大地增加了由它们保护的代码部分的覆盖率. 同样, AFL++[4]会优先并且额外地关注那些执行边缘很少被其他种子覆盖的种子, 这是一个有效的度量方式. 然而在真实的测试环境中, 被测程序的边与分支的数量非常庞大, 且模糊测试所生成的测试输入数量会急剧地增加最后趋于平缓. 种子数量的增长与这种精细化的统计反馈来指导种子调度, 所带来的计算开销是不可预估的. 此外, 这种根据种子覆盖到的稀有边缘(或分支)数量指导的种子调度, 其方法中对于稀有边缘(或分支)所定义的边界比较模糊(即: 选取统计结果中覆盖次数最少的 k 条边作为稀有边, 或是选取 $2k$ 条边作为稀有边).

在测试过程中, 差异性较大的测试输入之间有着截然不同的执行覆盖路径, 不同的测试输入执行了被测程序不同的代码区域. 如图 2 所示: 对于一个被测程序, 我们希望能够探测到的代码区域最大化(图 2(a)展示了初始种子语料库对被测程序空间探索的状态分布簇, 每个种子的执行状态用一个坐标表示, 同一簇中的种子探索程序空间的某一相同区域; 图 2(b)展示了对程序空间的探索过程; 图 2(c)展示了在理想情况下最终要达到均匀地探索整个目标程序空间的目标). 因此, 对于表示被测程序的高维空间(图 2 中以三维为例), 我们希望通过执行多样化的测试输入, 使得测试能够更全面地覆盖整个程序状态空间. 因此, 我们提出一种宏观直接的方法, 根据种子执行覆盖的路径对种子进行聚类分析, 有着相同路径覆盖模式以及执行行为的种子被归为一类. 不同类之间的种子所探索的区域可能大不相同, 而同属于某一类别的种子可能只会探索某处单一的代码区域, 如图 2(a)中所表示的情况. 对于种子数量较少的类, 代表对应代码区域被探索度较低, 本质即所处代码区域中的边缘与分支被覆盖的次数较少(即为稀有边、稀有分支). 通过对这些覆盖罕见边以及分支的种子突变生成更多的随机输入, 极大地增加了突破对应分支约束的概率, 使得探测由其保护的代码部分. 因此, 探索度越低的代码区域越稀有, 对该类区域的探索更有可能发现新的边缘覆盖. 通过种子的分类信息对种子优先级评估, 优先调度那些探索了稀有区域的种子, 并为其分配更多的能量以提高模糊测试的覆盖率. 如果在种子调度过程中按照种子簇群为单位去调用种子, 某类种子总优先于另外一类种子被调度, 这会导致后代种子的多样性下降, 从而引起对程序的探索不均匀. 我们根据分析的结果来平衡不同类别种子的调度数量, 确保后代种子的多样性. 最终, 通过不断的突变和探测, 使得测试能够执行程序的多样性路径, 让突变得到的输入更全面地覆盖整个程序空间, 实现程序空间的均匀的探索(如图 2(c)所表示的情况), 有助于检测出程序中更多潜在的错误行为.

根据种子输入覆盖的特征与执行状态序列对种子采用聚类分析, 可以帮助发现种子探测空间的结构和特征. 通过分析聚类结果, 我们可以了解种子覆盖的分布情况, 从而发现特定聚类中的种子群体可能表示某些

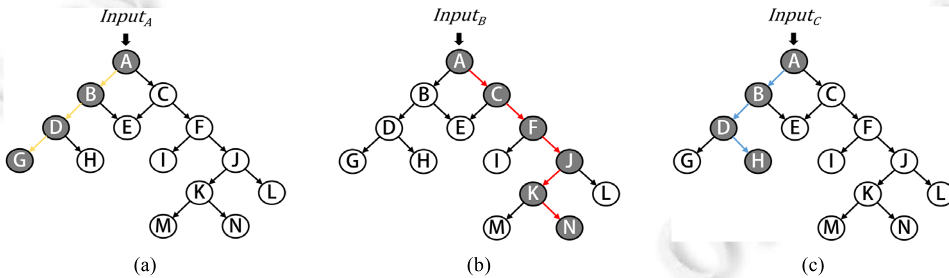
© 中国科学院软件研究所 http://www.jos.org.cn

特定的测试场景或输入域. 这些信息可以指导我们对模糊测试的改进和优化. 得到的种子聚类结果中每个类别种子之间彼此相似且具有共性, 但与其他类别中的种子相比具有明显的区别. 同一类别中的种子具有相似的输入模式、路径覆盖模式以及执行行为, 可以被看作是在特征空间中相互接近的种子集合, 在某种程度上代表了程序空间的特定区域与行为模式. 如图 3 所示, 给出一个程序的控制流图, 分别使用输入 A、输入 B 以及输入 C 作为该程序的测试输入, 执行路径的覆盖情况用不同颜色表示(图 3(a)–图 3(c)是同一个程序的控制流程图, 由不同输入执行的路径覆盖结果. 不同类别的种子输入的执行路径覆盖有着明显的差异). 输入 A 的执行路径集合通过代码执行的基本块可以表示为 $Path(A)=\{A,B,D,G\}$, 输入 B 的执行路径集合为 $Path(B)=\{A,C,F,J,K,N\}$, 输入 C 的执行路径集合为 $Path(C)=\{A,B,D,H\}$. 结合图 3 给出控制流程图可知: 输入 A 与输入 C 的路径覆盖模式明显区别于输入 B, 输入 A, C 与输入 B 探索了完全不同的程序区域, 按照聚类的思想, 将输入 A 与 C 归为一类. 在所有被覆盖到的边中, 被命中过两次的边为 $\{AB,BD\}$, 仅命中 1 次的边为 $\{DG,DH,AC,CF,FJ,JK,KN\}$. 可以发现: 所属类中种子输入数量较少的输入 B, 其覆盖了更多被命中次数较少的边, 且输入 B 执行覆盖所临近的未执行边的数量更多. 在模糊测试中, 优先调度这类种子并为其分配更多的能量产生更多的随机输入, 更有可能得到产生新覆盖的输入.



(a) (b) (c)

图 2 Cluzz 的动机和目标



(a) (b) (c)

图 3 一个体现种子类别差异的例子

基于这个问题, 我们提出了一种基于聚类分析驱动的种子调度的模糊测试方法 Cluzz. 我们将 Cluzz 设计为一个轻量级的工具, 可以集成到基准模糊器中, 以提高基准模糊器的测试效率与代码覆盖率. 与其他模糊测试方法相比, 我们通过将测试输入的执行路径覆盖在程序空间中的分布情况进行聚类分析, 以便根据种子状态分布对种子调度, 以更好地探索整个程序空间的各个区域, 进而实现测试探索路径的多样性. 通过种子稀有度为种子分配能量, 并随着模糊工作的进行对种子动态评估, 以更新种子调度的优先级, 让模糊器优先调度具有高优先级的种子, 从而提高测试效果.

2.2 Cluzz的工作流程

图 4 描述了集成 Cluzz 方法的模糊器的整体工作流程: 给定一个种子语料库和目标被测程序, 我们首先执行预演, 并通过 AFL[5]中的 AFL-showmap 工具来统计被测程序的路径执行情况, 获取种子语料库中所有种子

© 中国科学院软件研究所 http://www.jos.org.cn

的执行路径覆盖; 然后, 通过多重随机初始化的聚类分析, 将种子划分为不同的执行状态分布簇. 这些种子分布簇代表种子在程序执行状态空间中的不同区域.

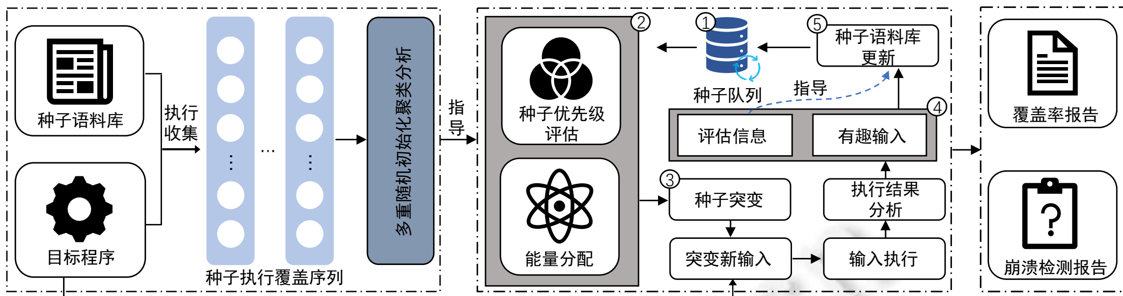


图 4 集成 Cluzz 的模糊器工作流程

接着, 利用执行状态分布簇的权重占比因子来指导种子的评估过程. 具体的, 通过计算种子的稀有度得分(①), 可以确定种子在探索程序空间中稀有代码区域的能力. 这个稀有度得分将指导后续种子的调度与能量分配(②). 模糊器将优先考虑那些能够探索稀有代码区域且稀有度得分较高的种子, 并分配更多的能量给它们, 从而让模糊测试可以更好地探索程序的稀有区域. 在种子的调度过程中, 我们选择具有高稀有度得分的种子, 并对它们进行突变(③). 突变操作会生成新的输入, 而我们只保留那些能够触发新的程序状态的突变输入, 也就是具有趣味性的新输入. 为了评估这些新输入的质量, 使用新输入与各个种子簇群之间的语义相似度来评估新输入的所属集群(④). 随着模糊测试的进行, 新输入的加入使得种子簇群之间的权重发生变化, 因而重新评估种子簇群的优先级以动态调整测试策略, 确保模糊测试的效果和效率, 并保证种子的样本多样性. 同时, 我们将这些新输入保存到种子语料库中, 并随着模糊工作的进行定期对种子语料库进行重新评估(⑤). 这个阶段性的重新评估将更新种子队列中种子的优先级, 确保始终选择最有潜力的种子进行后续的突变和探索. 通过上述优化过程, 我们能够在模糊测试中更好地利用种子的执行状态和特征信息, 从而高效地探索程序的不同区域并提高测试覆盖率.

通过种子执行序列进行聚类分析的详细过程如算法 1 所示.

算法 1. 种子的多重随机初始化聚类分析算法.

输入: 种子语料库: *S*, 目标程序: *P*.

变量: 聚类分析的目标数量: *K*, 初始化的聚类中心集合: *C*, 种子执行状态序列集合: *M*.

输出: 最优聚类结果: *BestResult*, 种子簇群权重占比: *FOP*.

```
1.  InitWithEmptyNumpy(M);
2.  BestResult←∅, Best_Score←+∞, FOP←∅;          //初始化参数
3.  for s∈S do                                     //种子执行信息采集
4.      executionSeq←GetCoverage(P,s);
5.      M←M∪executionSeq;
6.  end for
7.  M←DataProcess(M);                              //数据预处理
8.  for c∈C do                                     //执行聚类分析
9.      cluster_result←AssignCluster(M,c,K);
10.     eval_score←EvalClusterResult(cluster_result);  //评估聚类结果
11.     if eval_score<Best_Score then
12.         Best_Score←eval_score;
13.         BestResult←cluster_result;
```

© 中国科学院软件研究所 http://www.jos.org.cn

14. **end if**
15. **end for**
16. *FOP*←***CalcWeightProportions***(*BestResult*); //计算种子簇群权重信息
17. **return** *BestResult*, *FOP*

2.3 多重随机初始化聚类分析

在本节中, 我们通过种子的执行状态序列对种子进行聚类分析. 在聚类分析过程中, 过高的数据维度往往稀疏且难以处理. 而种子执行状态序列中的许多状态可能是重复或者无关紧要的. 这些无用信息会占据序列的大部分空间, 导致序列的稀疏性, 而直接进行聚类分析可能会带来过高的计算成本且影响聚类的效果. 因此, 我们只关注在训练数据中至少执行过 1 次的边. 对于单个输入的执行状态序列, 通过公式(3)的形式进行表示.

$$execSeq(seed)=\langle(edge_1,0),(edge_2,1),\ldots,(edge_k,1),\ldots\rangle \tag{3}$$

其中, 通过 0 或 1 来标识某条边是否被执行: 0 表示未执行, 1 表示被执行. 不同种子输入的边缘覆盖会存在偏差, 即, 某输入的边缘覆盖只包含程序所有边的少部分标签. 这一现象会导致训练数据中的一组标签之间存在多重共线性, 使得模型的损失难以收敛. 为了解决这个问题, 我们通过将所有输入都未覆盖的边信息进行消除, 只关注所有测试输入中至少执行过一次的边, 从而可以减少数据中的冗余信息和噪声. 例如, 种子 a 和种子 b 在某被测程序上的执行状态序列如公式(4)上半部所示, 通过上述的预处理方式得到的目标执行序列如公式(4)的下半部所示.

$$\begin{cases} edge_{\mathrm{id}}: \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \\ execSeq(a)=\langle 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1\rangle \\ execSeq(b)=\langle 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0\rangle \\ \Downarrow \\ execSeq(a)=\langle(edge_2,1),(edge_3,0),(edge_4,1),(edge_7,0),(edge_{10},1)\rangle \\ execSeq(b)=\langle(edge_2,1),(edge_3,1),(edge_4,0),(edge_7,1),(edge_{10},0)\rangle \end{cases} \tag{4}$$

其中, 种子 a 与种子 b 至少执行过 1 次的边的集合为$\{edge_2,edge_3,edge_4,edge_7,edge_{10}\}$. 在减少数据中的冗余信息和噪声后, 数据更易于可视化和理解, 同时保留有意义的特征. 通过降维, 我们可以更好地理解数据之间的关系, 发现数据的内在规律和模式, 从而提升聚类分析的效果.

此外, 为了获得更具有鲁棒性的聚类结果, 我们采用多重随机初始化的迭代方法. 每次迭代后计算聚类结果的评价得分, 并保存具有最优得分的聚类中心和标签以将其作为最终结果. 这样做可以减少聚类结果受随机因素影响的程度, 确保得到更可靠的聚类结果. 最后, 基于聚类的结果计算每个种子的权重占比, 这个指标将在后续步骤中指导种子的评估、调度和能量分配. 算法 1 提供了通过种子执行状态序列进行多重随机初始化聚类分析的过程, 接下来的讨论中将详细介绍每个步骤.

- 多随机初始化聚类分析

种子语料库中每个种子的执行序列都是由 0 和 1 组成的二进制序列, 计算每个序列与 K 个聚类中心之间的距离, 并将该数据样本分配给距离最近的聚类中心所在的簇. 算法 1 中的第 1 行–第 8 行对应数据预处理过程, 第 10–17 行详细叙述了对数据进行聚类的过程. 在对种子执行聚类分析的 C 次迭代过程中, 每次迭代都随机初始化聚类中心 c, 并且保存每次聚类迭代的结果 *cluster_result*. 在多次随机初始化迭代过程中,对每次迭代得到的聚类结果进行评估, 以获得最优聚类结果. 通过聚类损失作为衡量此次聚类质量的指标, 使用如公式(5)所示的误差平方和(sum of squared errors, SSE)来计算.

$$SSE=\sum\nolimits_{k=1}^{K}\sum\nolimits_{i=1}^{n}1_{y_i=k}\,\|x_i-\mu_k\|^2 \tag{5}$$

其中, K 表示簇的数量; n 表示样本的总数量; x_i 代表样本点 I; y_i 表示样本点 i 所属的簇; μ_k 表示簇 k 的质心; $1_{y_i=k}$ 是一个指示函数, 如果样本点 i 属于簇 k 则取值为 1, 否则取值为 0; 公式的最后一部分表示样本点 i 到簇 k 中

心点的欧几里得距离的平方. 通过多次聚类迭代优化该损失函数, 使其达到最小, 从而获取最优聚类结果.

- 计算种子簇群权重占比

对于聚类结果中的每一个种子簇, 都为其计算一个权重. 每个簇的权重可以帮助确定簇中的种子所执行代码区域的稀有程度. 例如图5所示, 对一个程序输入域空间的探索. 对于所有测试输入在输入域中的分布情况(以二维形式表示), 红色(即类别 0 的输入)所在的区域中测试输入数量最少, 也是被探索最不充分的区域. 如果将更多的机会放到对类别 0 所属区域的探索, 能够更加充分地穷尽所有输入组合, 进而更有机会触发程序的潜在崩溃. 更准确地说, 根据种子的执行状态序列对种子进行聚类分析, 那些包含相对较少种子的簇所对应的代码区域更为稀有, 即探索度较低的区域. 对从相关稀有区域采样的种子分配更高的执行优先级与突变次数, 产生执行了相关区域新的代码覆盖的有效测试用例的概率要大于在其余代码区域采样的种子.

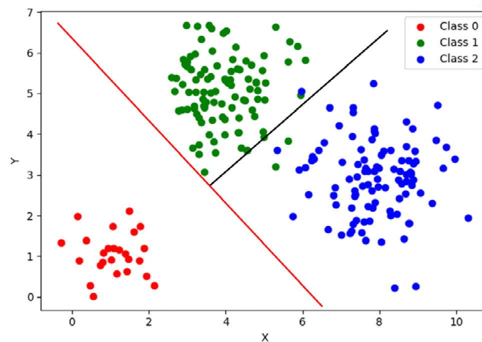


图 5 随机生成的测试用例在被测程序输入域中的分布情况

在模糊测试中, 通过将更多的能量分配给这些簇中的种子, 并优先考虑对稀有区域的探索, 从而提高模糊测试的效率和效果. 因此, 根据某个簇中种子数量与所有种子的数量比值的倒数来计算该种子簇群的权重, 具体如公式(6)所示.

$$f_i=N/n_i \tag{6}$$

其中, f_i 表示的是第 i 个簇的权重占比, N 表示的是所有种子的数量, n_i 表示被划分到第 i 个簇的种子数量. 簇中种子数量越少, 该簇对应的权重就越大. 占比因子较大的簇, 表明该簇所对应的代码区域更为稀有, 且簇中的种子更为稀有. 我们是通过种子的执行状态序列得到聚类结果, 所以簇中种子数量很少, 代表该类种子所探索的程序空间被探索的很浅, 因此需要给予这些种子更多的能量, 以让其能够更加充分地探索该代码区域. 这样的计算结果可以更直观地反映出簇内种子的相对稀缺程度, 在后续种子的能量分配过程中, 能够更直接地通过权重占比因子去衡量种子的稀有度, 并进一步根据种子的稀有度得分进行能量分配.

2.4 种子调度与能量分配

通过种子聚类分析结果, 对种子进行稀有度得分的计算与种子优先级的评估. 我们的目标是: 通过探索稀有代码区域以获得更高的测试覆盖率, 并通过平衡对不同类别种子的调度来保持后续生成新的测试输入的多样性与均衡性.

通过种子簇群信息对种子优先级评估的详细过程如算法 2 所示.

算法 2. 种子的优先级评估算法.

输入: 种子语料库: *S*, 种子聚类结果: *R*, 种子簇群权重信息: *FOP*.

变量: 记录种子评估信息: *V*, 已加入种子队列的种子集合: *was_selected*, 种子评估循环检查变量: *check_done*.

输出: 种子优先级队列: *Q*.

1. ***InitTheSeedQueue***(*Q*);
2. *V*←∅, *check_done*=0, *was_selected*←∅; //初始化参数和种子队列

3. R←***GetSeedGroupPriority***(R,FOP); //评估种子簇群的优先级
4. **for** s∈S **do** //评估种子稀有度得分
5. V[s]=***GetRarityScore***(FOP,s,R);
6. **end for**
7. **while** *check_done*==0 **do** //种子优先级评估
8. **for** r∈R **do**
9. *seed*←***ChoseBestSeed***(r,V,*was_selected*);
10. **if** *seed* **equals** *NULL* **then**
11. **continue**;
12. **end if**
13. Q←Q∪*seed*;
14. *was_selected*←*was_selected*∪*seed*;
15. **end for**
16. **if** *was_selected* **equals** S **then**
17. *check_done*=1;
18. **end if**
19. **end while**
20. **return** Q

通过算法 2, 我们首先对当前的种子语料库中的所有种子进行优先级评估, 并生成一个种子优先级队列. 在模糊测试器的工作流程中, 我们只需要按照评估的优先级顺序来调度种子. 首先, 在算法 2 中, 根据聚类分析的结果和各个种子簇群的权重, 结合种子执行信息计算种子的稀有度得分(如算法 2 中的第 4–6 行), 后续根据种子的稀有度对种子进行优先级评估. 种子簇群中, 越稀有的种子被调度的优先级越高. 权重越大的种子簇群代表执行的代码区域探索度越低, 这类种子簇群中的种子越应该被优先调度并分配更多的能量. 此外, 为了确保对程序空间进行均匀探索并获得路径探索的多样性, 我们进行集群一致性检查. 这意味着在每次调度种子时, 确保所选的种子来自不同的种子簇群, 并且所挑选的种子在之前的工作中没有被模糊过(如算法 2 的第 7–19 行). 重复这个过程, 直到种子语料库中所有种子都被评估并且加入队列. 这样可以避免测试过程陷入程序空间的某个单一区域, 保证新生成的输入具有多样性. 通过这种方式, 能够同时实现对程序多样性路径的探测并保持生成输入的多样性.

- 种子稀有度得分的计算

种子稀有度得分通过聚类分析结果中各种子簇群的权重占比来计算. 某组中的种子数量越少, 其权重占比越大, 代表相应代码区域被探索得越浅(聚类分析的结果是根据种子执行状态序列得到的). 权重越大的种子簇群所探索的相应区域越稀有, 被探索度越低, 相应组中的种子应该被分配更多的能量. 对于所属同一组的种子, 它们之间的表现也有差异. 通常情况下, 执行路径较长的种子有着更好的潜力. 如图 3 中所示的情况, 使用程序控制流程图中基本块编号来表示一条路径, 种子输入 A 和种子输入 B 的执行路径分别可以表示为 $Path(A)=\{AB,BD,DG\}$ 与 $Path(B)=\{AC,CF,FJ,JK,KN\}$. 种子输入 B 的执行路径更长, 且与种子输入 B 执行路径临近的未执行边的集合为$\{AB,CE,EI,JL,KM\}$, 而与种子输入 A 执行路径临近的未执行边的集合为$\{BE\}$. 我们发现, 种子 B 执行路径上的未执行临近边数量更多. 为种子 B 分配更多的能量, 更有希望通过突变得到覆盖这些未执行边的新输入, 即发现程序新边缘的可能性更大.

因此, 在计算的时候也将种子的执行路径长度作为衡量种子得分的指标. 与执行路径临近的边, 比较容易通过突变的方式到达. 通过对一个执行了更多边的输入进行突变(未执行的临近边数量更多), 更有可能探索程序的未执行代码区域. 稀有度得分计算如公式(7)所示.

$$Score_{\text{rarity}}(s)=\frac{fop_s}{fop_{\text{avg}}}\times[1+\beta\times(len_{\text{path}}(s)/Avg_{\text{len}})] \tag{7}$$

其中: fop_s 表示种子 s 所在种子簇群的权重占比; fop_{avg} 表示所有种子簇群权重的平均值; β是路径长度的权重系数, 反映执行路径长度参数对种子得分的影响程度(在这里, 我们设置为β=0.75, 以避免能量溢出造成资源的浪费); $len_{\text{path}}(s)$表示种子 s 的执行路径长度. 在公式(7)中, 种子执行路径的长度与平均执行路径长度的比值越大则得分越高, 分数越高表示种子越稀有. 通过为越稀有的种子分配越多的能量, 能够鼓励模糊器探索那些稀有的、不常见的路径, 从而提高覆盖率.

- 种子能量的计算

种子的能量大小, 直接决定对一个种子的突变次数. 为更有潜力的种子分配更多能量, 更有可能通过突变发现程序的潜在崩溃, 从而提高测试覆盖率. 我们通过上述步骤得到了种子的稀有度得分, 并根据种子优先级队列挑选优先级最高的种子进行模糊工作. 在进行模糊之前, 为当前选择的种子分配能量, 以确定后续种子突变的次数. 我们为稀有度得分越高的种子分配更多的能量, 根据 AFL[6]中对能量的计算方式, 我们对于所选种子 q 的能量计算公式(8)如下.

$$E(q)=perf_score\times[1+\omega\times((S_r(q)-Avg_{S_r})/Avg_{S_r})] \tag{8}$$

其中, $perf_score$ 代表种子的表现得分. 在 AFL 中, 种子的表现得分由执行速度与路径覆盖大小等因素决定, 最终通过表现得分计算得到种子的能量. 在我们的方法中, 我们并没有舍弃基准模糊器中为种子计算能量大小的工作, 而是将我们的工作作为一项新的指标, 用于种子能量的计算. 公式中的权重系数ω控制了对于当前种子稀有度得分和平均得分差异的关注程度(我们根据基准模糊器中计算能量得分的数量级, 取值ω=1). 当稀有度得分高于平均值的种子时, 权重系数ω所乘的项为正值, 这使得高于平均稀有度得分的种子(稀有度更高)获得了更多的能量; 反之, 使得低于平均稀有度得分的种子获得较少的能量, 相当于受到了一定的惩罚.

我们在算法 3 中给出了集成了 Cluzz 的模糊器工作流程. 根据算法 2 得到种子的优先级队列后, 在算法 3 中, 根据评估得到的种子优先级队列 Q, 选择下一个要模糊的种子(算法 3 第 11 行). 根据种子的稀有度对其分配能量(算法 3 第 12 行), 以确保对稀有代码区域的深入探索. 通过对当前种子进行突变生成新的输入, 并对其进行评估, 筛选出有趣的输入并将其添加到种子语料库中(算法 3 第 14 行、第 15 行). 同时, 随着模糊工作进行, 原本稀有度高的种子被分配更多的能量, 产生更多的新输入. 我们通过计算新输入与各个种子簇群之间的语义相似度, 对突变得到的新输入进行归类. 原本权重值高的种子簇群会因为更多的新种子加入, 权重值会下降. 为了平衡模糊过程中新输入的多样性, 模糊过程会重新评估各个种子簇群的权重, 并且根据状态更新阈值来重新评估整个种子语料库的种子优先级, 以适应测试过程的变化. 这一过程的执行频率由变量 *execute_num* 和 *step_size* 控制.

- 新输入的评估归类

随着模糊工作的不断进行, 突变得到的新输入数量也会急剧增长, 我们需要平衡各类种子之间的数量平衡. 被评估为稀有的种子会随着突变产生更多的同类新输入, 从而变得不再稀有. 然而, 由于随机突变的不确定性, 突变得到的新输入可能从语义上面不同于父类种子所属的类别. 其中, 语义相似度描述了一个种子输入的执行路径和某个种子簇群所覆盖的执行路径的相似性. 该种子与某个种子簇群的执行路径越相同, 它们就越相似, 将突变得到的新输入归入相似度最高的种子簇群中. 相似度得分通过公式(9)来计算.

$$Scoer_{\text{sim}}(s,R)=|Path(S)\cap Path_{\text{all}}(R)|/|Path_{\text{all}}(R)| \tag{9}$$

其中, $Path_{\text{all}}(R)$表示种子簇群 R 中所有种子的执行路径的集合, $Path(s)$是得到的新输入 s 的执行路径集合. 例如: 有一个新输入 S 与一个种子簇群 R, 他们的执行路径集合分别为 $Path(S)=\{A,B,C,D\}$ 与 $Path_{\text{all}}(R)=\{A,B,C,E,F,G,H,I\}$. 我们发现: 种子 S 和种子簇群 R 有着相同的执行路径$\{A,B,C\}$, 种子簇群的执行路径集合的大小为 8. 所以, 可计算得到种子 S 与种子簇群 R 的语义相似度得分为 0.375.

为避免过多聚类分析带来的时间开销, 对于得到的新输入, 暂时根据公式(9)评估其所属的种子簇群, 用于该阶段中后续对新种子的评估. 在当前种子语料库中, 所有种子都被执行完一轮后进入下一测试阶段, 重

© 中国科学院软件研究所 http://www.jos.org.cn

新调用聚类分析评估所有种子的聚类结果, 并根据新的权重与种子类别信息, 重新对种子语料库中的所有种子进行优先级的评估, 从而指导后续测试工作.

配备 Cluzz 的模糊器的详细工作流程如算法 3 所示.

算法 3. 配备 Cluzz 的模糊器工作流程算法.

输入: 种子语料库: *S*, 目标程序: *P*, 种子聚类结果: *R*, 种子簇群权重信息: *FOP*.

变量: 已执行种子集合: *was_fuzzed*, 种子优先级队列: *Q*, 已执行种子数量: *exe_num*, 状态更新阈值: *step_size*.

输出: 测试覆盖率报告,程序缺陷检测报告.

```
1.  InitTheThreshold(step_size);
2.  InitTheSeedQueue(Q);
3.  Was_fuzzed←∅, exe_num=0;                  //初始化参数和种子队列
4.  Q←EvalSeedPriority(S,R,FOP);              //基于算法 2
5.  while fuzzer is running do                //模糊循环
6.      if exe_num equals step_size then
7.          Q←EvalSeedPriority(S,R,FOP);      //基于算法 2
8.          exe_num.reset(·);
9.          step_size.update(·);
10.     end if
11.     seed, rarity_score←GetTopSeed(Q,was_fuzzed);
12.     energy←CalcEnergyScore(seed,rarity_score);
13.     new_seeds←SeedMutation(seed,energy);
14.     eval_info←EvalOffSpring(new_seeds);
15.     S.update(new_seeds,eval_info);
16.     R.update(new_seeds,eval_info);
17.     FOP.update(S,R);
18.     was_fuzzed←was_fuzzed∪seed;
19.     exe_num=exe_num+1;
20. end while
21. return 覆盖率报告,崩溃检查报告
```

3 实验评估

在本节中, 我们通过设计实验来证明 Cluzz 的性能, 并通过实验结果来回答以下研究问题.

- RQ1: 相比于基准模糊器, Cluzz 的集成能够使得模糊器在覆盖率方面带来多大程度的提升?
- RQ2: Cluzz 是否提高了模糊器检测独特崩溃(unique crash)的能力?
- RQ3: Cluzz 中, 对种子进行的聚类分析会造成的额外时间开销是多少?
- RQ4: 相比其他有关种子调度及能量分配的模糊测试研究工作, Cluzz 的表现如何?

3.1 实验设计

为了评估我们的 Cluzz 方法的有效性, 我们在 3 个当前最流行的模糊器上面实现了 Cluzz, 并从发现新边缘的能力与崩溃等方面, 将集成了 Cluzz 的模糊器与基准模糊器进行比较. 具体来说, 我们将每个模糊器在主流的 8 个目标程序上运行 24 h, 每个模糊器与对应的基准模糊器都在相同实验环境下进行实验, 并在每个目标程序上重复 5 次实验, 并对 5 次实验结果求取平均值, 与该研究领域的通行做法一致. 此外, 我们选择了 4 个主流的有关种子调度与能量分配的模糊测试工具, 验证我们 Cluzz 中种子调度与能量分配的研究工作的有

效性. 我们的所有实验均在 Ubuntu 20.04 系统上进行, 系统硬件环境为 Intel(R) Core(TM) i5-10400F CPU 和 16 GB 内存.

- 基准模糊器

如前面所述, 我们选择了 3 个当前主流的模糊器来评估我们方法的通用性. 其中:

- NEUZZ[13]利用神经网络对输入字节和程序分支行为之间的关系进行建模, 通过模型推断输入字节与约束条件关系来解析约束. 与传统模糊器运行不同, NEUZZ 是作为一个独立的进程运行, 它与模糊器通信. NEUZZ 中对测试种子的选择是随机的, 将我们的聚类分析整合到模型训练之后, 根据分析的结果挑选 NEUZZ 模糊过程中使用的种子, 并随着 NEUZZ 的增量学习实现种子语料库的动态评估.
- AFL[5]是最为经典的覆盖引导的模糊测试工具, 目前有很多模糊测试方法是在 AFL 基础上进行改进的.
- 此外, 还选择了 AFL 的扩展方法 MOPT[16]. MOPT 在 AFL 基础上采用粒子群优化(PSO)算法实现变异算子调度, 以实现更高的代码覆盖率.

为了将我们的方法整合到 AFL 与 MOPT 中, 我们将直接根据种子簇群的信息、种子稀有度得分来对种子进行优先级排序, 并根据种子的稀有度得分来计算种子的能量.

在我们选择的 4 个有关种子调度与能量分配的模糊器中:

- Ecofuzz[19]提出了一种对抗性多臂老虎机(multi-armed bandit, MAB)的变体, 它对 AFL 的能量调度建模并阐述种子集的 3 种状态, 并开发自适应调度算法和基于概率的搜索策略.
- AFL++[4]是一种结合现今所有基于 AFL 框架的改进方案, 即取其精华去其糟粕形成的一款测试工具, 它在种子调度与能量分配中结合了 AFLFast[29]的调度算法, 并增加了最新种子用例的分数以深入研究新发现的路径; 在调度过程中, 忽略了种子的运行时间, 将重点放在边缘很少被其他种子用例执行后覆盖的种子上.
- FairFuzz[20]对测试中的稀有分支进行识别, 优先调度能够命中稀有分支路径的种子, 并使变异偏向于产生覆盖稀有分支的输入.
- AFLFast 提出了一种使模糊测试中种子调度偏向于执行低频路径的种子策略, 即给予低频路径更多机会, 以便在相同的模糊测试量下探索更多的路径.

这 4 个模糊器都是在 AFL 基础上进行的研究开发. 为了对比 Cluzz 中种子调度与能量分配的有效性, 我们将配备了 Cluzz 的 AFL 模糊器与之进行对比.

- 目标程序

我们在 8 个不同的真实程序(见表 1)上对 Cluzz 方法进行了评估. 选用的被测程序来源于开源的 GNU 编程语言工具程序以及其他开源工具程序. 我们的被测程序与NEUZZ实验中的被测程序相同, 并且所有对比实验都在相同环境与相同的初始种子下进行. 其中, readelf 用于显示有关 ELF 二进制文件的信息, objdump 用于查看目标文件或可执行文件的组成信息, nm 的功能为列出对象文件中的符号, size 的功能为列出每个二进制文件的节大小, strip 用于去除程序文件中的调试信息, djpeg 是一种广泛使用的处理 JPEG 图像文件的工具, harfbuzz 是一个开源的文本整形引擎, xmllint 是一个很方便的处理及验证 xml 的工具.

表 1 目标程序信息

目标程序	程序来源	测试输入格式	测试指令
readelf	binutils-2.30	ELF	readelf -a @@
objdump			objdump -D @@
nm			nm -C @@
size			size @@
strip			strip -o/dev/null @@
harfbuzz	harfbuzz-1.7.6	TTF	harfbuzz @@
djpeg	libjpeg-9c	JPEG	djpeg @@
xmllint	libxml2-2.9.7	XML	xmllint @@

注: @@为指令中的占位符, 表示输入是文件

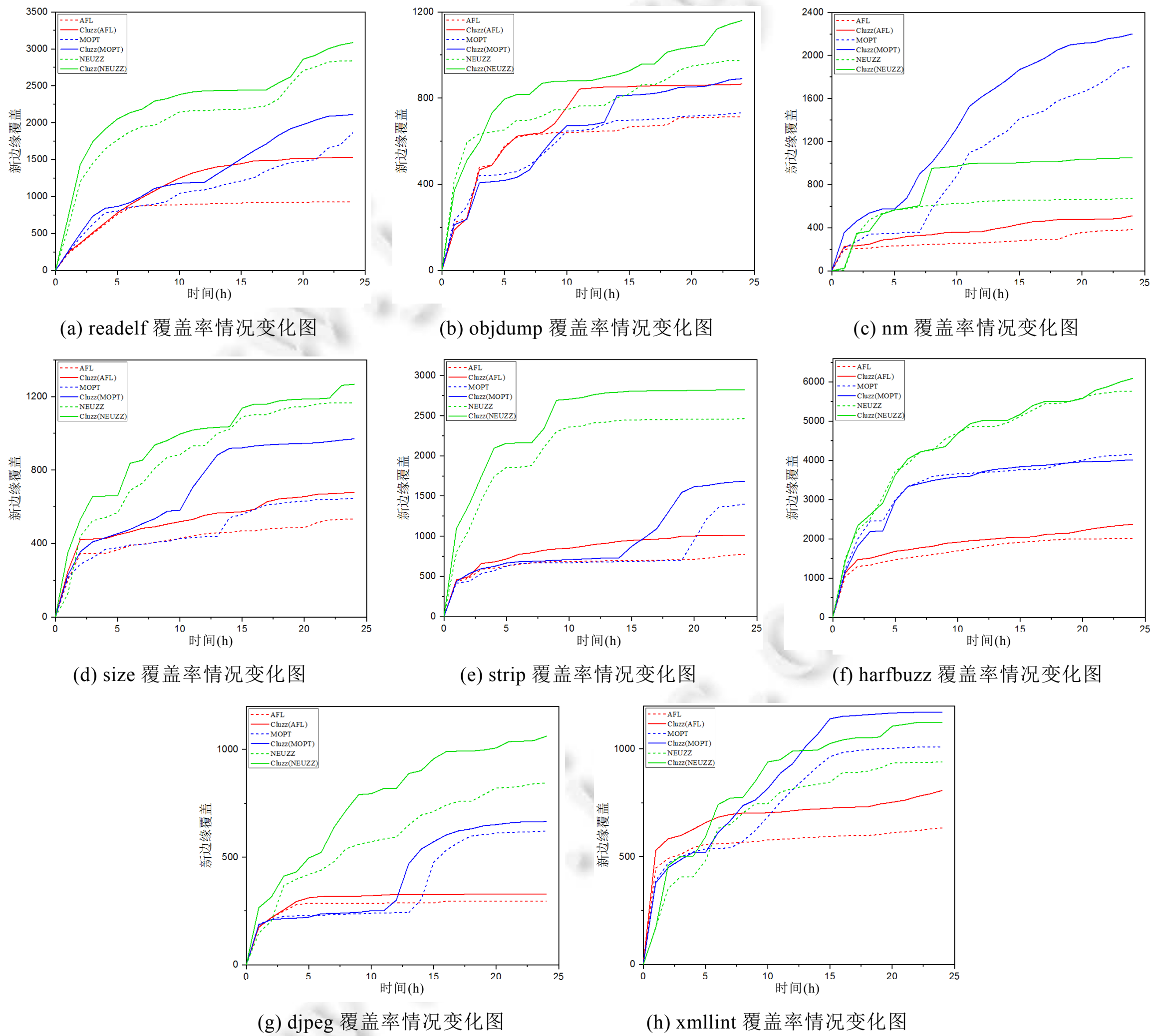
© 中国科学院软件研究所 http://www.jos.org.cn

- 初始种子

为了对比实验的公平, 所用的种子输入全部从 NEUZZ 和 MOPT 公布的种子集中收集得到, 并且集成 Cluzz 的模糊器与基准模糊器使用同样的初始种子去运行被测程序. NEUZZ 与其他的模糊器不同, 因为它需要初始数据集, 因此直接选择使用 NEUZZ 中的公开数据集作为实验的初始种子输入, 用于 Cluzz 在 NEUZZ 上面的对比实验.

3.2 RQ1: 新边缘覆盖的探测能力

被测程序的代码覆盖率是模糊测试的一个重要指标. 因为程序的错误是隐藏在代码中的, 只有通过对各个代码区域的探索才会触发潜在的程序崩溃. 我们在 8 个被测程序上同时运行两个版本的模糊器(集成 Cluzz 的版本和普通版本), 并且每组对照模糊器在同一个被测程序上进行 5 次实验(每次实验为 24 h). 通过 AFL[5] 中的 AFL-showmap 工具来统计被测程序的代码覆盖情况, 并通过绘制折线图反映实验中的各个模糊器在发现新路径方面的平均性能. 如图 6 所示, 除了 harfbuzz 的结果之外, 集成 Cluzz 的模糊器相比于基准模糊器, 在所有目标程序上都有着较为明显的增长趋势.



(a) readelf 覆盖率情况变化图 (b) objdump 覆盖率情况变化图 (c) nm 覆盖率情况变化图

(d) size 覆盖率情况变化图 (e) strip 覆盖率情况变化图 (f) harfbuzz 覆盖率情况变化图

(g) djpeg 覆盖率情况变化图 (h) xmllint 覆盖率情况变化图

图 6 不同模糊器运行 24 h 的覆盖率情况变化图

© 中国科学院软件研究所 http://www.jos.org.cn

通过观察, 我们发现: 对于 MOPT 实验中新边缘覆盖的增长趋势并不平滑, 相比于另外两组对比, MOPT 没有一个很稳定的增长趋势. 这是因为 MOPT 认为: 在对一个种子模糊的过程中, AFL 在确定性阶段(deterministic stage)花费的时间比在浩劫(havoc)和拼接(splice)阶段花费的时间要多得多(MOPT 证明, 在这两个阶段可以发现更多独特崩溃和路径). 因此, MOPT 基于 AFL 的模糊器提供了一种优化, 称为起搏器模糊模式(pacemaker fuzzing model), 可以选择性地避免耗时的确定性阶段. 具体来说, 如果 MOPT 在很长一段时间内没有发现任何新的唯一崩溃或路径, 它将有选择地禁用后续测试用例的确定性阶段. 所以可以发现: 在我们图 6 所有 MOPT 的实验中, 在发现新边缘覆盖的趋势激增之前, 都会有一段较长的时间新边缘覆盖趋势几乎没有波动. 这是因为在这段时间 MOPT 并没有发现新的边缘覆盖或崩溃, 使得后续 MOPT 进入了起搏器模糊模式.

如表 2 所示, 配备了 Cluzz 的模糊器相比于基准模糊器在大多数的被测程序上都实现了更好的新边缘覆盖.平均来说, Cluzz (AFL)在 8 个目标程序上发现的新边缘数量平均要比 AFL 多 29.25%. Cluzz (MOPT)在除 harfbuzz 之外的 7 个被测程序上都表现得比 MOPT 要好, 并且在 8 个被测程序上发现新边缘的数量平均比 MOPT 高 17.48%. 最后, Cluzz (NEUZZ)在 8 个被测程序上的表现都有提升, 发现新边缘的数量平均比 NEUZZ 多 19.72%.

表 2 原始模糊器与配备 Cluzz 的模糊器在 24 h 发现的新边缘数量的对比

目标程序	#新边缘数量					
	AFL	Cluzz (AFL)	MOPT	Cluzz (MOPT)	NEUZZ	Cluzz (NEUZZ)
readelf	926	1 531 (+65.33%↑)	1 862	2 110 (+13.32%↑)	2 839	3 085 (+8.67%↑)
objdump	713	865 (+21.32%↑)	732	890 (+21.58%↑)	975	1 160 (+18.97%↑)
nm	358	511 (+32.73%↑)	1 904	2 199 (+15.49%↑)	672	1 047 (+55.80%↑)
size	534	679 (+27.15%↑)	647	970 (+49.92%↑)	1 165	1 267 (+8.76%↑)
strip	773	1 014 (+31.18%↑)	1 402	1 684 (+20.11%↑)	2 465	2 824 (+14.56%↑)
harfbuzz	2 010	2 372 (+18.01%↑)	4 157	4 012 (−3.48%↓)	5 765	6 091 (+5.65%↑)
djpeg	296	328 (+10.81%↑)	622	665 (+6.91%↑)	843	1 061 (+25.86%↑)
xmllint	633	807 (+27.49%↑)	1 010	1 171 (+16.00↑)	940	1 123 (+19.47%↑)

此外, 我们通过对每组对比实验的结果进行计算统计形成表 3 中的内容.

表 3 配备 Cluzz 的模糊器相较普通模糊器 24 h 发现的新边缘数量的提升

模糊器	被测程序	Average (%)	Min (%)	Max (%)	Median (%)	CV
Cluzz (AFL)	readelf	+65.33	+57.22	+83.10	+62.16	0.14
	objdump	+21.32	+12.28	+36.70	+24.15	0.38
	nm	+32.73	+23.47	+37.06	+34.13	0.15
	size	+27.15	+6.96	+68.81	+15.96	0.82
	strip	+31.18	+24.75	+41.60	+28.81	0.19
	harfbuzz	+18.01	+9.14	+41.59	+13.44	0.66
	djpeg	+10.81	+7.57	+13.38	+11.27	0.19
	xmllint	+27.49	+9.19	+73.62	+20.94	0.85
Cluzz (MOPT)	readelf	+13.32	+7.59	+24.15	+11.79	0.46
	objdump	+21.58	+18.73	+28.75	+19.62	0.18
	nm	+15.49	+5.49	+37.18	+11.35	0.73
	size	+49.92	−1.21	+89.13	+77.67	0.79
	strip	+20.11	+6.39	+35.67	+15.91	0.54
	harfbuzz	−3.48	−12.79	+3.40	−1.48	1.61
	djpeg	+6.91	−5.53	+28.08	+4.53	1.65
	xmllint	+16.00	−3.48	+98.81	+4.58	1.12
Cluzz (NEUZZ)	readelf	+8.67	+5.54	+11.73	+8.69	0.24
	objdump	+18.97	+16.51	+24.48	+17.19	0.16
	nm	+55.80	+7.50	+101.61	+44.79	0.64
	size	+8.76	+0.13	+15.82	+8.19	0.60
	strip	+14.56	−3.62	+44.58	+7.19	1.13
	harfbuzz	+5.65	+1.19	+10.71	+5.57	0.56
	djpeg	+25.86	+4.49	+63.29	+20.53	0.76
	xmllint	+19.47	+15.69	+21.38	+20.67	0.11

具体来说, 我们将集成 Cluzz 的模糊器与基准模糊器的 8 组对比实验的结果进行统计, 统计实验结果提升

© 中国科学院软件研究所 http://www.jos.org.cn

率的最小值、最大值以及中位数, 并计算新边缘数量提升的平均大小. 此外, 还根据统计的实验结果计算离散系数(coefficient of variation, CV)来度量每组实验结果之间的离散程度. 离散系数越大, 说明每组实验结果之间的离散程度也大, 代表模糊器的运行结果越不稳定, 离散系数越接近 0, 则说明离散程度越小. 其中, 集成 Cluzz 的模糊器在大多数的被测程序上面, 每一次的实验结果都比基准模糊器有着提升; 且 Cluzz (AFL)在 8 个被测程序上的发现新边缘数量提升的平均离散系数为 0.42, 比平均离散系数分别为 0.89 和 0.53 的 Cluzz (MOPT)与 Cluzz (NEUZZ)稳定性要好. 表 3 中各项数据的统计以及离散系数表明 Cluzz 方法的有效性以及实验结果的稳定性(离散系数少有大于 1 的情况).

综上实验结果回答 RQ1. 配备 Cluzz 的模糊器相较于基准模糊器可以显著提高测试覆盖的发现(平均增加 22.15%的新边缘覆盖). 在 24 组对比实验中, 有 23 组配备了 Cluzz 的模糊器比普通的模糊器实现了更多的代码覆盖率. 并且, 在给出的新边覆盖数量随时间变化对比图中, 配备了 Cluzz 的模糊器普遍能够保持优势直到测试结束. 通过对所有实验进行统计, 发现新边缘数量的提升, 能够说明配备 Cluzz 的模糊器几乎在所有实验中都优于普通模糊器, Cluzz 在发现新边缘数量方面有着显著的提升.

3.3 RQ2: 崩溃检测能力

程序崩溃是指程序在执行过程中发生了无法处理的错误或异常, 导致程序无法正常继续执行并终止运行. 崩溃检测能力是衡量测试能效的重要指标. 根据测试过程中触发的崩溃数量, 我们将集成 Cluzz 的模糊器与普通模糊器进行比较. 如表 4 所示, 我们在每个程序上进行 5 次实验, 所有的数据都是根据实验结果计算得到的平均值.

表 4 配备 Cluzz 的模糊器相较普通模糊器 24 h 内检测独特崩溃数量的对比

目标程序	#独特崩溃数量					
	AFL	Cluzz (AFL)	MOPT	Cluzz (MOPT)	NEUZZ	Cluzz (NEUZZ)
readelf	20.6	44	27.2	42.4	32	41.6
objdump	0	0	0	0	0	2
nm	0	0.6	1	3.8	0	1
size	0.6	2.6	0.6	3.6	0.6	2
strip	0	0	0.4	1.4	0	0

其中, 在被测程序 readelf、size、nm 上面, 3 个集成 Cluzz 的模糊器都要比对应的基准模糊器检测到更多的崩溃. 对于被测程序 objdump, 只有 Cluzz (NEUZZ)发现了崩溃, 其余的模糊器都没有发现额外的崩溃. 对于被测程序 strip, 模糊器 AFL 与 NEUZZ 对应的实验都没有发现额外崩溃, Cluzz (MOPT)相比于 MOPT 在 24 h 内多发现额外崩溃. 此外, 对于实验中没有在表 4 中出现的 3 个被测程序 harfbuzz、jpeg 与 xmllint, 在我们实验中的所有模糊器上都没有检测出崩溃.

综上实验结果回答 RQ2. 集成了 Cluzz 的模糊器在暴露程序崩溃方面的能力要优于普通的模糊器, 并且在能够发现崩溃的 11 组实验中, 配备了 Cluzz 的模糊器全都比普通模糊器检测出更多的崩溃数量. 而且, 所有实验在发现独特崩溃的数量方面, 配备了 Cluzz 的模糊器检测数量平均是普通模糊器的 1.7 倍. 实验结果能够证明, 本文提出的 Cluzz 方法在改善模糊器检测独特崩溃的能力方面有着显著的提升.

3.4 RQ3: Cluzz中有关种子聚类的开销

在这个实验中, 我们统计将 Cluzz 集成到模糊器中所造成的额外时间开销. 额外的运行开销是模糊器运行期间对种子进行聚类分析开销. 为了测量这些开销, 我们针对集成了 Cluzz 的模糊器进行记录, 记录使用这些模糊器运行被测程序 24 h 所有有关种子聚类总的时间开销. 我们在表 5 中给出了配备了 Cluzz 的模糊器在各个被测程序上运行过程中, 由种子聚类分析所造成的额外时间开销, 以及这些时间开销相较于整个运行周期(24 h)的占比大小.

通过表 5 中的统计可以发现: Cluzz (AFL)在 8 个被测程序上的平均额外开销为 0.75%, Cluzz (MOPT)的平均额外开销为 1.25%, Cluzz (NEUZZ)的平均额外开销为 1.42%. 同一个模糊器在不同被测程序上的额外开销大不相同, 最直接的原因是被测程序的差异性所造成. 不同被测程序之间的功能差异以及代码复杂度各不相

© 中国科学院软件研究所 http://www.jos.org.cn

同,导致相同时间内的同一个模糊器在不同被测程序上发现新边缘的能力不同, 所生成的有趣测试用例数量也不同. 间接导致种子语料库中种子数量的差异, 使得模糊器在种子聚类分析上面的时间开销不同. 尽管种子聚类分析带来了额外的时间开销, 但是相较于整个测试周期(24 h), 额外开销的占比很小, 几乎对测试结果没有影响.

表 5 配备 Cluzz 的模糊器在 24 h 中的额外时间开销

目标程序	Cluzz (AFL)		Cluzz (MOPT)		Cluzz (NEUZZ)	
	时间(s)	开销占比(%)	时间(s)	开销占比(%)	时间(s)	开销占比(%)
readelf	1 056.22	1.22	1 399.21	1.62	1 651.55	1.91
objdump	530.64	0.61	549.26	0.64	622.43	0.72
nm	371.79	0.43	1 479.28	1.71	913.15	1.06
size	481.26	0.56	967.18	1.12	1 217.49	1.41
strip	852.59	0.99	1 106.28	1.28	1 617.29	1.87
harfbuzz	1 177.63	1.36	1 896.73	2.20	2 249.17	2.60
djpeg	220.64	0.26	417.21	0.48	726.31	0.84
xmllint	511.29	0.59	824.16	0.95	779.64	0.90
Average	650.26	0.75	1 079.91	1.25	1 222.13	1.42

综上实验结果回答 RQ3. 集成 Cluzz 的基准模糊器相较于对应的普通模糊器, 在实验中最多增加 2.6%的时间开销, 最小只有 0.26%, 所有的实验平均只增加了 1.14%的额外时间开销. 这一开销相较于我们整个的测试实验周期 24 h 是非常少的. 所以, 可以证明: 我们方法能够通过少量的时间开销换取更好的测试效果. 综合, 融入 Cluzz 并未对测试工作造成负面的影响.

3.5 RQ4: Cluzz中种子调度与能量分配策略的性能

在该实验中, 我们选取了 4 个当前先进的模糊测试工具, 分别为 EcoFuzz[19]、AFL++[4]、AFLFast[29]和 FairFuzz[20]. 这 4 个模糊测试工具均在 AFL 的基础上展开了与种子调度及能量分配有关的研究工作, 这与我们提出的研究工作方向相同. 为了验证本文提出的 Cluzz 在种子调度与能量分配工作的优劣, 我们将配备了 Cluzz 的 AFL 模糊器与这 4 个模糊器设计实验进行对比. 我们对 GNU 的 5 个编程语言工具进行测试, 并统计在每个被测程序上测试运行 24 h 后, 各个模糊器发现新边缘的数量和独特崩溃的数量. 在模糊测试中, 能量分配阶段所计算的能量大小代表了种子在浩劫(havoc)和拼接(splice)阶段的突变次数. 为了体现各个模糊器在能量分配阶段的优劣, 我们选择在对所有被测程序进行测试时, 使所有模糊器跳过种子突变的确定性阶段(deterministic stage). 我们在相同的实验环境下, 分别使用 Cluzz (AFL)以及所选取的 4 个模糊器对一个被测程序同时开展测试工作, 并对同一个被测程序开展 4 轮实验, 最后统计实验结果的平均值, 形成表 6 中的实验数据.

表 6 有关种子调度与能量分配的模糊器与 Cluzz (AFL)在 24 h 中发现新边缘数量与独特崩溃数量的对比

目标程序	EcoFuzz		AFL++		AFLFast		FairFuzz		Cluzz (AFL)	
	新边缘数量	独特崩溃数量	新边缘数量	独特崩溃数量	新边缘数量	独特崩溃数量	新边缘数量	独特崩溃数量	新边缘数量	独特崩溃数量
readelf	2 526	90	2 843	74.5	2 673	92	2 444	48	**2 909**	**107**
objdump	1 596	2.5	1 135	0	1 641	0	1 278	2	**1 658**	**5**
nm	1 282	5	1 226	0	1 513	5	762	0	**1 590**	**8**
size	**1 046**	3	569	0	981	5.5	1 037	3	1 034	**8**
strip	1 883	**4**	969	0	1 760	1	**1 884**	0	1 774	**4**
Total	8 333	104.5	6 742	74.5	8 568	103.5	7 405	53	**8 965**	**132**

通过表 6 中的统计可以发现: 对于被测程序 readelf、objdump 以及 nm、Cluzz (AFL)在发现新边缘数量以及独特崩溃数量方面的表现皆优于其他 4 个模糊器. 而对于被测程序 size 和 strip, Cluzz (AFL)在发现独特崩溃的数量方面优于其他 4 个模糊器, 在发现新边缘数量方面与表现最好的模糊器相差无几. 总的来说, 在 5 个被测程序上, Cluzz (AFL)在发现总的新边缘数量方面相比于 EcoFuzz、AFL++、AFLFast 和 FairFuzz 分别提升了 7.58%、32.97%、4.63%和 21.07%, 在发现总的独特崩溃数量方面优于其他 4 个模糊器.

综上实验结果回答 RQ4. 集成 Cluzz 的 AFL 模糊器 Cluzz (AFL)相较于其他 4 个基于 AFL 的模糊器, 其在 5 个被测程序上的综合表现最好. 在发现新边缘数量以及总的独特崩溃数量方面, 皆优于其他模糊器, 并且测试效果有着显著的提升, 这也可以说明 Cluzz 在种子调度与能量分配方面的有效性. 相较于现有的种子调度与能量分配研究工作, Cluzz 有着更优秀的表现.

4 有效性威胁

- 内部有效性

对内部效度的一个威胁在于我们评估阶段中所选取的对比模糊器的实施. 为了减少这种威胁, 我们在实验阶段中重用了它们论文公开的原始源代码. 此外, 还由各位作者手动仔细检查了所有代码, 以确保对比模糊器中代码实现的正确性, 以及与其对应的研究内容方法的功能一致性. 并且, 所有的模糊器在运行时采用相同的运行命令参数、相同的初始测试输入以及测试命令.

- 外部有效性

外部效度的威胁主要存在于测试对象和基准上. 为了减少外部威胁, 我们在验证 Cluzz 有效性的实验中选择了 3 个具有代表性的、最先进的模糊测试器, 包括基线模糊器 AFL[5]、基于 AFL 的拓展模糊器 MOPT[16]和基于神经程序平滑的模糊器 NEUZZ[13]. 在验证 Cluzz 相较其他种子调度研究工作有效性的实验中, 我们选择了 4 个先进且具有代表性的模糊测试器[4,19,20,29], 该 4 个模糊器均在 AFL 的基础上展开有关种子调度的研究工作, 所以在我们的对比实验中, 我们将这 4 个模糊器与配备了 Cluzz 的 AFL 模糊器进行对比. 此外, 根据所研究的模糊测试领域相关原始论文, 选择了在实验评估中使用频率最高的 8 个主流被测程序(具体信息见表 1). 另一个威胁外部有效性的因素是评估结果的随机性. 为了减少这种威胁, 我们在 RQ1–RQ3 的实验中, 对所有评估结果进行 5 次运行并取平均值. 在 RQ4 的实验中, 我们对所有实验运行 4 次并计算平均值,以减少随机性对实验评估的影响.

- 结构有效性

对构造有效性的威胁主要在于本文评估阶段使用的主要度量指标, 即反映代码覆盖率的边缘覆盖数量. 为了减少这种威胁, 虽然可以有各种方法来测量边缘覆盖, 但我们选择遵循许多现有的模糊器[6,13,14,20]所使用的方法, 通过 AFL 内置的名为 AFL-showmap 的工具来收集边缘覆盖. 此外, 我们还根据实验发现的独特崩溃数量评估了 Cluzz 模糊测试方法的有效性.

5 相关工作

5.1 聚类分析

聚类分析, 也被称为簇分析, 是数据挖掘和探索性数据分析中的基本技术之一. 通过将相似的对象归为一组来识别数据集中的内在模式和结构. 多年来, 研究人员已经进行了大量的研究, 提出了很多种聚类方法, 其中一种常见的聚类方法是 K-means 聚类算法[23]. 该算法将数据集分为 K 个簇, 每个簇由与之最相似的数据点组成, 通过迭代优化簇的中心点来最小化数据点与其所属簇中心点之间的距离, 从而得到聚类结果. 层次聚类算法[25]通过不断地将最相似的对象归并为一组来构建一个层次结构, 层次聚类可以根据簇之间的相似度进行凝聚聚类(自底向上)或分裂聚类(自顶向下). DBSCAN 聚类算法[24]是一种基于密度的聚类方法, 它将密度较高的数据点归为一组, 而较低密度的数据点被视为噪声. DBSCAN 算法可以有效地处理具有不规则形状的簇, 并对噪声数据具有一定的鲁棒性. 这些聚类分析方法已经被广泛应用于自然语言处理、图像处理等各个领域. 目前, 在模糊测试领域中有根据种子相似度对种子归类的方法, 通过自适应的方式将种子归类, 并为每个类的种子维护一组突变算子的概率分布来挑选种子最优的突变策略. 但是并没有通过聚类分析指导种子调度的方法.

© 中国科学院软件研究所 http://www.jos.org.cn

5.2 模糊测试

到目前为止, 已经提出了很多用于改进模糊测试的技术, 如符号执行[31,32]、动态污染分析[33,34]和机器学习[11,13,14]. 这些方法通过覆盖率来引导种子变异, 确定应该修改测试输入中的哪一部分, 能够提高代码覆盖率. 一些工作侧重于定位那些用于安全操作的字节. 如 TaintScope[35]和 BuzzFuzz[36]使用污染跟踪技术来识别输入中对于系统安全指令或库调用的输入字节, 然后专注于修改这些字节. Angora[6]侧重于引导模糊测试去解决程序分支约束来扩大测试的分支覆盖. 它利用字节级污染分析来定位流向分支的字节, 然后, 通过改变这些字节来绕过约束条件. 在本文中, 我们的主要目标是优化模糊测试中的种子调度阶段. 通过对种子聚类分析结果指导测试输入的优先级和对种子能量分配的大小. 现在已有的优化的模糊测试方法通常是基于历史边缘覆盖情况[19,20,27,29]或者其他反馈指标指导种子调度, 例如执行时间[37,38]. 也有通过内存访问情况[39]指导种子能力调度. 如 K-Scheduler[21]基于运行过程间的程序控制流程图, 构建包含种子节点的边缘水平图, 通过计算种子节点的 Katz 中心性对种子进行优先排序. 但是, 对于这些运行过程数据的获取以及指标的评价通常需要复杂的计算过程. 相反, 我们通过分析种子在程序空间中的执行状态分布, 指导种子调度与能量分配, 这一个过程是轻量级的, 时间和空间上的开销极小. 根据我们的初步实验表明, Cluzz 对种子调度的优化可以改善模糊测试效果.

6 总 结

在本文中, 我们提出了一种基于聚类分析驱动的种子调度的模糊测试方法. 通过分析种子在特征空间上的区别, 结合种子执行覆盖状态在程序空间中的分布情况对种子进行分类. 根据不同类别种子所探索的程序空间不同以及不同种子簇群的路径覆盖模式与聚类分析结果, 对种子进行优先级评估, 优先调度探索稀有区域的种子. 通过种子簇群的信息与种子执行信息为种子计算所分配的能量大小. 并且, 通过种子聚类结果平衡种子调度过程中各个类别种子的数量, 从而确保后代种子的多样性与平衡性, 避免测试在某个代码区域消耗过多的测试资源, 实现对程序空间均衡性的探索.

在我们的方法中, 我们按照种子执行路径进行聚类分析并对种子进行划分. 为执行了对应探索度较低区域的种子分配更多的能量, 这些种子覆盖到了更多的稀有边以及稀有分支, 在其基础之上突变生成更多的随机输入, 极大地提高了突破对应代码区域中分支约束的概率, 从而实现对程序代码空间更深入的探索. 此外, 通过模糊过程中动态重新评估种子优先级, 确保有趣的种子总是能被优先调度, 从而提高模糊测试效果. 根据本文叙述的方法实现了 Cluzz, 为了验证本文提出的 Cluzz 方法的有效性, 将 Cluzz 集成到 3 个现有的模糊测试工具并设计实验对真实世界的 8 个开源程序进行实验评估. 评估表明, 配备了 Cluzz 的模糊器的在新边缘的发现方面与崩溃检测能力方面明显优于普通模糊器. 此外, 通过与现有的种子调度与能量分配研究工作进行对比, 进一步证明了 Cluzz 在该方面工作的有效性.

虽然本文提出的 Cluzz 方法取得了较好的实验效果, 但也存在一些不足, 需要在今后的工作中加以克服. 首先, 需要对种子聚类所依赖的指标加以完善, 充分依据程序信息与种子执行覆盖的情况. 这可以通过程序分析技术与静态分析方法来完成, 将种子聚类做得更加完善以提高指导种子调度的有效性. 其次, 将新的字节推断与符号执行以及学习算法等指导突变的方法与我们的工作相结合, 进一步提高测试覆盖与漏洞检测的能力.

References:

[1] Aschermann C, Schumilo S, Blazytko T, Gawlik R, Holz T. REDQUEEN: Fuzzing with input-to-state correspondence. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2019. [doi: 10.14722/ndss.2019.23xxx]

[2] Zheng Y, Davanian A, Yin H, Song C, Zhu H, Sun L. FIRM-AFL: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In: Proc. of the 28th USENIX Security Symp. (USENIX Security 2019). 2019. 1099–1114.

[3] Schumilo S, Aschermann C, Gawlik R, Schinzel S, Holz T. kAFL: Hardware-assisted Feedback Fuzzing for OS Kernels. In: Proc. of the 26th USENIX Security Symp. (USENIX Security 17). 2017. 167–182.

[4] Fioraldi A, Maier D, Eißfeldt H, Heuse M. AFL++: Combining incremental steps of fuzzing research. In: Proc. of the 14th USENIX Workshop on Offensive Technologies. (WOOT 2020). 2020. 1–12.
[5] Zalewski M. American Fuzzy Lop (AFL). 2021. http://lcamtuf.coredump.cx/afl/
[6] Chen P, Chen H. Angora: Efficient fuzzing by principled search. In: Proc. of the IEEE Symp. on Security and Privacy (SP). IEEE, 2018. 711–725.
[7] Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H. VUzzer: Application-aware evolutionary fuzzing. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2017. [doi: 10.14722/ndss.2017.23404]
[8] libFuzzer—A library for coverage-guided fuzz testing. 2021. https://llvm.org/docs/LibFuzzer.html
[9] Zhao L, Duan Y, Yin H, Xuan J. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2019. [doi: 10.14722/ndss.2019.23504]
[10] Herrera A, Gunadi H, Magrath S, Norrish M, Payer M, Hosking AL. Seed selection for successful fuzzing. In: Proc. of the 30th ACM SIGSOFT Int’l Symp. on Software Testing and Analysis (ISSTA). ACM, 2021. 230–243.
[11] Godefroid P, Peleg H, Singh R. Learn&fuzz: Machine learning for input fuzzing. In: Proc. of the 32nd IEEE/ACM Int’l Conf. on Automated Software Engineering (ASE). IEEE, 2017. 50–59.
[12] Zong P, Lv T, Wang D, Deng Z, Liang R, Chen K. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In: Proc. of the 29th USENIX Security Symp. (USENIX Security 2020). 2020. 2255–2269.
[13] She D, Pei K, Epstein D, Yang J, Ray B, Jana S. NEUZZ: Efficient fuzzing with neural program smoothing. In: Proc. of the IEEE Symp. on Security and Privacy (SP). IEEE, 2019. 803–817.
[14] She D, Krishna R, Yan L, Jana S, Ray B. MTFuzz: Fuzzing with a multi-task neural network. In: Proc. of the 28th ACM Joint Meeting. on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE). ACM, 2020. 737–749.
[15] Gao FJ, Wang Y, Situ LY, Wang LZ. Deep learning-based hybrid fuzz testing. Ruan Jian Xue Bao/Journal of Software, 2021, 32(4): 988–1005 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6225.htm [doi: 10.13328/j.cnki.jos.006225]
[16] Lyu C, Ji S, Zhang C, Li Y, Lee WH, Song Y, Beyah R. MOPT: Optimized mutation scheduling for fuzzers. In: Proc. of the 28th USENIX Security Symp. (USENIX Security 2019). 2019. 1949–1966.
[17] Jauernig P, Jakobovic D, Picek S, Stapf E, Sadeghi AR. DARWIN: Survival of the fittest fuzzing mutators. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2023. [doi: 10.14722/ndss.2023.23159]
[18] Peng H, Shoshitaishvili Y, Payer M. T-Fuzz: Fuzzing by program transformation. In: Proc. of the IEEE Symp. on Security and Privacy (SP). IEEE, 2018. 697–710.
[19] Yue T, Wang P, Tang Y, Wang E, Yu B, Lu K, Zhou X. EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In: Proc. of the 29th USENIX Security Symp. (USENIX Security 2020). 2020. 2307–2324.
[20] Lemieux C, Sen K. FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proc. of the 33rd ACM/IEEE Int’l Conf. on Automated Software Engineering (ASE). ACM, 2018. 475–485.
[21] She D, Shah A, Jana S. Effective seed scheduling for fuzzing with graph centrality analysis. In: Proc. of the IEEE Symp. on Security and Privacy (SP). IEEE, 2022. 2194–2211.
[22] Arthur D, Vassilvitskii S. k-means++: The advantages of careful seeding. In: Proc. of the 18th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA). 2007. 1027–1035.
[23] Hartigan JA, Wong MA. Algorithm AS 136: A K-means clustering algorithm. Journal of the Royal Statistical Society, Series C (Applied Statistics), 1979, 28(1): 100–108. [doi: 10.2307/2346830]
[24] Ester M, Kriegel HP, Sander J, *et al*. A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proc. of the 2nd Int’l Conf. on Knowledge Discovery and Data Mining (KDD’96). 1996. 226–231.
[25] Ward Jr JH. Hierarchical grouping to optimize an objective function. Journal of the American Statistical Association, 1963, 58(301): 236–244.
[26] Lyu C, Liang H, Ji S, Zhang X, Zhao B, Han M, Li Y, Wang Z, Wang W, Beyah R. SLIME: Program-sensitive energy allocation for fuzzing. In: Proc. of the 31st ACM SIGSOFT Int’l Symp. on Software Testing and Analysis (ISSTA). ACM, 2022. 365–377.
[27] Böhme M, Manès VJM, Cha SK. Boosting fuzzer efficiency: An information theoretic perspective. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE). ACM, 2020. 678–689.
[28] Chen J, Wang S, Cai S, Zhang C, Chen H, Chen J, Zhang J. A novel coverage-guided greybox fuzzing based on power schedule optimization with time complexity. In: Proc. of the 37th IEEE/ACM Int’l Conf. on Automated Software Engineering (ASE). ACM, 2022. 1–5.

[29] Böhme M, Pham VT, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security (CCS). ACM, 2016. 1032–1043.
[30] Zhang K, Xiao X, Zhu X, Sun R, Xue M, Wen S. Path transitions tell more: Optimizing fuzzing schedules via runtime program states. In: Proc. of the 44th Int'l Conf. on Software Engineering (ICSE). ACM, 2022. 1658–1668.
[31] Yun I, Lee S, Xu M, Jang Y, Kim T. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In: Proc. of the 27th USENIX Security Symp. (USENIX Security 2018). 2018. 745–761.
[32] Xie XF, Li XH, Chen X, Meng GZ, Liu Y. Hybrid testing based on symbolic execution and fuzzing. Ruan Jian Xue Bao/Journal of Software, 2019, 30(10): 3071–3089 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5789.htm [doi: 10.13328/j.cnki.jos.005789]
[33] Gan S, Zhang C, Chen P, Zhao B, Qin X, Wu D, Chen Z. GREYONE: Data flow sensitive fuzzing. In: Proc. of the 29th USENIX Security Symp. (USENIX Security 2020). 2020. 2577–2594.
[34] Liang J, Wang M, Zhou C, Wu Z, Jiang Y, Liu J, Liu Z, Sun J. PATA: Fuzzing with path aware taint analysis. In: Proc. of the IEEE Symp. on Security and Privacy (SP). IEEE, 2022. 1–17.
[35] Wang T, Wei T, Gu G, Zou W. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Proc. of the IEEE Symp. on Security and Privacy (SP). IEEE, 2010. 497–512.
[36] Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing. In: Proc. of the IEEE 31st Int'l Conf. on Software Engineering (ICSE). IEEE, 2009. 474–484.
[37] Lemieux C, Padhye R, Sen K, Song D. PerfFuzz: Automatically generating pathological inputs. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2018. 254–265.
[38] Petsios T, Zhao J, Keromytis AD, Jana S. SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security (CCS). ACM, 2017. 2155–2168.
[39] Coppik N, Schwahn O, Suri N. MemFuzz: Using memory accesses to guide fuzzing. In: Proc. of the 12th IEEE Conf. on Software Testing, Validation and Verification (ICST). IEEE, 2019. 48–58.

附中文参考文献:

[15] 高凤娟, 王豫, 司徒凌云, 王林章. 基于深度学习的混合模糊测试方法. 软件学报, 2021, 32(4): 988–1005. http://www.jos.org.cn/1000-9825/6225.htm [doi: 10.13328/j.cnki.jos.006225]
[32] 谢肖飞, 李晓红, 陈翔, 孟国柱, 刘杨. 基于符号执行与模糊测试的混合测试方法. 软件学报, 2019, 30(10): 3071–3089. http://www.jos.org.cn/1000-9825/5789.htm [doi: 10.13328/j.cnki.jos.005789]

张文(2001－), 男, 硕士生, CCF 学生会员, 主要研究领域为软件安全测试, 模糊测试.

张翊(1995－), 男, 博士生, 主要研究领域为软件安全测试, 模糊测试.

陈锦富(1978－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为软件测试, 软件安全, 可信软件.

刘一松(1966－), 男, 博士, 教授, 主要研究领域为人机交互技术, 软件工程, 可信软件.

蔡赛华(1990－), 男, 博士, 讲师, CCF 专业会员, 主要研究领域为恶意流量检测, 异常数据检测, 软件安全测试.