计算机科学
COMPUTER SCIENCE

针对缺陷根源定位的测试用例生成技术

杜昊, 王允超, 燕宸毓, 李星玮

引用本文

杜昊, 王允超, 燕宸毓, 李星玮.针对缺陷根源定位的测试用例生成技术[J]. 计算机科学, 2023, 50(7): 10-17.

DU Hao, WANG Yunchao, YAN Chenyu, LI Xingwei. Test Cases Generation Techniques for Root Cause Location of Fault [J]. Computer Science, 2023, 50(7): 10-17.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.220700128

针对缺陷根源定位的测试用例生成技术

杜 昊 王允超 燕宸毓 李星玮
信息工程大学数学工程与先进计算国家重点实验室 郑州 450001
(504224090@qq.com)

摘 要 缺陷根源定位是软件调试的重要阶段,基于频谱的缺陷根源定位方法是软件自动化调试研究中的热点问题,但其定位效果很大程度上取决于测试用例的质量。不同类型软件的测试输入通用性差,随机生成的测试输入则存在过拟合或混杂项过多的问题,导致分析结果误差较大,致使目前该类技术的应用场景有限。针对测试用例生成问题,提出了基于崩溃路径的分阶段探索方法 Dgenerate,并实现原型工具 Dloc。首先利用二进制插桩手段在程序执行输入阶段于基本块中插桩路径信息,根据此信息将原始测试输入划分为普通型和引导型;然后利用动态能量调度算法探索崩溃相关路径生成高质量的测试用例;最后在原始程序中执行测试用例并追踪执行时信息,通过统计分析的方法有效地定位到程序缺陷根源的位置。文中选取了 6 个不同类型软件中的 15 个真实 CVE 漏洞进行实验,结果显示 Dloc 生成的测试用例与已有技术相比可以将定位效率平均提升 75%,并且 Dloc 能够以 87%的准确性在评分前五的位置中输出缺陷根源相关代码片段,验证了所提方法系统的可行性和实用性。

关键词:根源定位;测试用例;程序频谱;统计分析;定向模糊测试
中图法分类号 TP311

Test Cases Generation Techniques for Root Cause Location of Fault

DU Hao,WANG Yunchao,YAN Chenyu and LI Xingwei
State Key Laboratory of Mathematical Engineering and Advanced Computing,Information Engineering University,Zhengzhou 450001,China

Abstract Vulnerability root cause localization is an important stage of software debugging,and spectrum-based fault root cause localization method is a hot issue in software automation debugging research,but the effectiveness of the positioning depends to a large extent on the quality of the test cases. Test inputs of different types of software are poorly generalized,and randomly generated test inputs lead to large errors in analysis results due to overfitting or too many confounding items,resulting in limited application scenarios of such techniques at present. In this paper,we propose a phased exploration method Dgenerate based on crash paths to address the test case generation problem and implement the prototype tool Dloc. First,we use binary staking to insert staking path information in the basic block during the input stage of program execution,and then classify the original test inputs into common and guided types based on this information. Then,we use the dynamic energy scheduling algorithm to explore crash-related paths to generate high-quality test cases. Finally,the test cases are executed in the original program and execution information is traced to effectively locate the root cause of program fault through statistical analysis. In this paper,15 real CVE vulnerabilities in six different types of software are selected for experiments,and the results show that test cases generated by Dloc can improve location efficiency by 75% on average compared to previous techniques,and Dloc can output the code fragments related to the root causes of defect in the top five positions with an accuracy of 87%,which verifies the feasibility and practicality of the method system in this paper.

Keywords Root cause location,Test case,Program spectrum,Statistical analysis,Directed greybox fuzzing

1 引言

当一个软件崩溃时,软件维护人员需要在修复问题之前找到源代码中程序崩溃的根源所在。而软件规模和复杂性的日益增加、崩溃类型的复杂多变都使人工定位修复缺陷变成了一项代价高昂的任务。研究表明,软件维护人员仅仅在寻找定位缺陷的根源上就将花费一半工时或更多的时间[1]。

近年来人们提出了一些自动化技术来帮助程序员完成这项任务。基于频谱的缺陷根源定位技术(Spectrum based Fault Localization,SFL)由于具有效率和有效性,被认为是这方面研究中最突出的技术之一。据统计,1977 年至今发表的有关论文中,与频谱技术相关的缺陷根源定位研究占比达 42%[2]。

程序频谱[3],主要是指程序执行过程中产生的关于程序语句的覆盖信息,以及执行是否通过的信息。基于程序频谱的根源定位技术的工作原理是执行给定的测试用例,并根据

到稿日期:2022-07-12 返修日期:2022-11-04
基金项目:国家重点研发计划(2019QY0501)
This work was supported by the National Key Research and Development Program of China(2019QY0501).
通信作者:王允超(w_yunchao@sina.com)

执行测试用例的结果为程序的每个元素（如语句、块、函数）分配一个概率估计或分数，评估此程序元素被执行时缺陷触发的可能性。高得分表明其与缺陷根源相关性强，即为了触发缺陷，执行这条语句是充分且必要的，软件维护人员在此处打补丁也最有可能消除缺陷。

针对此问题的研究重点都在于提升语句的评分算法或单一谓词的复杂组合上[4]，但仅仅是统计分析的方法并不足以产生高保真性结果，因为不知道应该在哪一次的测试输入中统计概率分数。几乎所有频谱类缺陷定位的研究都假定有高质量的测试输入，而事实并非如此。高质量的测试用例是此项技术不可或缺的一个组成部分，实验证明随机生成的测试用例往往会导致统计结果过拟合于原始崩溃路径，即原始路径中与根源无关但执行率高的语句被误报，或由于测试输入的混杂性，无法分析出正确的结果。在单个执行中可以观测到，在超过 1000 个程序语句的大型程序里，此问题十分严重。

本文的主要贡献如下：

（1）借鉴定向模糊测试的思想提出了一种基于崩溃路径分阶段探索的测试用例生成技术 Dgenerate，可以有效生成高质量测试用例，打破了频谱类根源定位技术无法自动化生成高质量测试用例的局限。

（2）在 Dgenerate 技术的基础上实现了一个名为 Dloc 的原型工具，动态生成高质量测试用例，并根据测试用例执行结果进行缺陷根源定位工作。

（3）针对 6 款真实软件中的 15 个真实 CVE 漏洞，实验结果表明 Dloc 所生成的测试用例与已有技术相比可以将根源定位效率平均提升 75%，并且可以以 87% 的准确性在评分前五的位置中识别出缺陷根源，充分验证了本文方法的有效性。

本文第 2 节介绍了频谱类根源定位技术的发展和测试用例生成技术的相关工作及本文方法的整体思路；第 3 节介绍了 Dloc 的具体实现细节；第 4 节通过真实软件中 CVE 漏洞的定位实验验证了工具的效果；最后总结全文并展望未来。

2　背景

2.1　相关工作

在之前的工作中，研究人员根据是否需要执行测试用例，将根源定位方法分为静态根源定位技术[5]和动态根源定位技术[6]。其中动态根源定位技术需要执行测试用例，搜集程序执行过程的行为和结果来定位缺陷根源。在动态定位技术中，基于程序频谱的动态定位方法具有很好的定位效果，是目前研究的热点。Jones 等[7]认为，被崩溃测试用例执行覆盖次数越多的语句，与缺陷根源相关的可能性就越大，据此他们提出了 Tarantula 公式来进行根源定位工作。随后，Abreu 等[8]受到分子生物学基因相似度公式和聚类分析思想的启发，提出了 Ochia 方法。实证研究表明，Ochiai 的缺陷定位效果优于 Tarantula。Wong 等[9]进一步分析了测试用例对缺陷定位的影响，他们认为成功测试用例数一般比崩溃测试用例数多，而这会影响根源定位效果。为了突出失败用例的影响，他们提出了 D^* 方法来进行缺陷根源定位工作。

上述研究都假定有合适的测试输入，实验也仅在小规模测试套件集如西门子测试套件中进行测试，无法在真实大规模软件中应用，局限性较大。

针对测试用例生成问题，Artzi 等[10]提出了一种名为 Apollo 的工具，基于符号执行自动生成测试用例。Apollo 首先记录一个反映程序执行的控制流谓词的路径约束，随后通过更改路径约束中的谓词并求解所得到的约束来生成新的测试输入。但该方法依赖于二进制分析技术符号执行[11]来实施，产生了巨大的性能开销，成为了其进一步在大型程序中应用的阻碍。

Aurora[12]运用了 AFL 的 crash exploration mode[13]来生成测试用例，虽然这种方法效率与通用性都较高，但它的探索过程仅以覆盖率作为引导依据，会产生许多与根源定位无关的混杂输入。实验证明这种方法所生成的测试用例会严重干扰根源定位结果的准确性。

VulnLoc[14]提出了一种集中模糊的方法来进行测试用例的生成，其通过生成敏感性图来记录不同输入字节对不同代码行的敏感性，从而达到强制执行到目标代码行的目的。但此方法生成的测试用例缺少路径探索多样性，存在着与崩溃路径过拟合的问题。

2.2　需求分析

给定一个程序和一个崩溃输入，我们的目标是定位出程序崩溃的根源。首先需要创造两组高质量测试用例，分别是崩溃测试用例和非崩溃测试用例。基于崩溃输入和非崩溃输入执行的程序在路径语义上一定有差别这样的认知，我们通过统计分析的方法找到第一个导致程序状态偏差的指令，直观地说，这条指令就是程序崩溃的根本原因。

程序行为是一个二元分类——崩溃或者非崩溃。由于大型程序中代码量极大，无法分析所有程序语句，因此我们着重关注那些与缺陷根源相关性高的代码片段，即可以准确预测程序行为的代码。我们需要一组合适的测试用例作为输入来找到这些代码片段。

高质量的测试用例需要满足以下几点要求：

（1）为触发缺陷的崩溃输入创造一组执行路径相似的崩溃测试输入，尽可能保证其崩溃根源的相似性。

（2）创造一组与崩溃输入相对的非崩溃输入，用作对比分析。

（3）测试用例执行路径既不能过拟合于原始崩溃路径，否则会导致原始路径中与根本原因无关但执行率高的语句被误报；也不能过于发散，因为引入与触发原始缺陷无关的语句会导致统计结果出现偏差。

针对以上需求，本文提出基于崩溃路径分阶段探索的测试用例生成技术 Dgenerate，并在此基础上实现原型系统 Dloc。该系统可以自动化生成测试用例并通过执行测试用例、收集执行时信息对二进制程序进行精确的缺陷根源定位。系统流程如图 1 所示。具体来说，Dloc 对一个给定的目标二进制程序进行插桩，得到一个带有基本块权重分数信息的二进制文件。Dgenerate 通过给定的原始种子输入、插桩的二进制文件划分种子类别以生成不同的种子队列，接着经过动态能量分配、变异等步骤生成测试用例。Dloc 把生成的测试用例当作输入重新在原始二进制程序中执行，并追踪执行时信息，根据对执行时信息的统计分析得到缺陷根源所在位置。

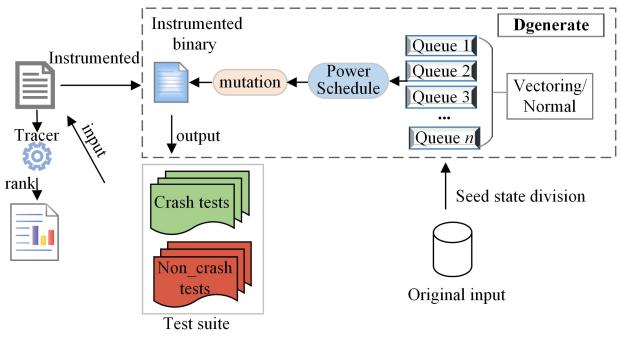

图 1 系统工作流程图

Fig. 1 System workflow diagram

3 Dloc 实现细节

3.1 基于崩溃路径引导的信息插桩

为了保持探索路径的相似性，我们需要在原始程序中标记出与崩溃执行相关联的路径，然后根据此路径来发散探索其余相似路径。我们选择的插桩粒度是基本块级别的，由于基本块只有一个出口，块内的所有指令都将被执行，因此将基本块作为最小粒度选择是合理的。插桩过程如算法 1 所示。

算法 1 基于敏感函数的信息插桩算法

输入：Program S / * S 是初始的目标二进制程序 * /

输出：Program S′ / * S′是被插桩后的目标二进制程序 * /

1. Score←0
2. / * 初始化全局共享变量 score 并赋初值 0 * /
3. Order←0
4. / * 初始化全局共享变量 order 并赋初值 0 * /
5. List∈ExploitTraceFunc
6. for each BB∈S do
7. / * BB 代表基本块 * /
8. Score=IfHasFunc(List)
9. Order=CheckOrder(List)
10. Inject Score into BB
11. Inject Order into BB
12. end for

本文引入了两个新的全局变量：*Score* 和 *Order*。被插桩的程序执行输入后，Dgenerate 可以读取这些值作为后续种子类型的判断依据及变异策略的选择依据。具体来说，我们在插桩的过程中为不同的基本块插入相应的代码以标识基本块的权重，并且与传统模糊测试工具 AFL[15] 的覆盖率信息一样，以共享内存的方式实现全局共享。

List 指崩溃时的堆栈回溯函数列表，它记录了程序崩溃时所调用的函数名称以及函数调用序列。算法 1 第 8 行的 *IfHasFunc* 函数用来判定当前基本块中是否存在 *Call Funcname* 这样的指令，其中 *Funcname* 是 *List* 列表中的函数。接着根据当前基本块中相关指令的数量为基本块赋予分数(Score)，不存在则统一赋初始值。

算法 1 第 9 行的 *CheckOrder* 函数用来判定此基本块中的 *Funcname* 在 *List* 序列中的位置并作为 *Order* 记录下来。若存在多个 *Funcname*，首先选取出现次数最多的函数序列作为基本块 *Order*，若出现次数一样则选取第一个出现的 *Funcname* 序列作为 *Order*，如果基本块中没有 *List* 列表中的函数，则统一赋初始值。我们需要与崩溃执行相似的执行路径，而相同的函数调用序列体现了执行轨迹的相似性。

3.2 基于动态能量分配的种子调度策略

我们把一个种子从种群中被选中的可能性称为种子的能量，它决定了一个种子在路径探索过程中被选择的次数。为了更好地进行测试用例生成，我们提出了基于动态能量分配的种子调度策略。

首先根据种子的特点对其进行状态划分，对插桩后的程序初始运行一遍输入；然后根据种子经过的基本块分数 *Score* 给予种子一个相应的分值 *s_score*。并把种子经过基本块中 *Order* 标签出现最多的值赋予种子序列标签的 *s_order*，没有则赋初始值。

如果仅仅探索崩溃函数相关的路径，测试用例会与原始崩溃路径过拟合，如果只关注路径多样性测试用例又很难保不引入与崩溃路径无关的干扰项。为解决此问题，本文借鉴了 AFLGO[16] 的思想，运用模拟退火算法模型进行能量分配。算法 2 给出了我们的能量分配策略，首先运行一遍初始种子获取每个种子的评分 *s_score* 与序列 *s_order*，接着依据式(1)将 *s_score* 评分进行标准化，得一个到 0～1 之间的值。

$$s\_score_{standard}=\frac{s\_score(x)-\min_{t\in P}(s\_score(t))}{\max_{t\in P}(s\_score(t))-\min_{t\in P}(s\_score(t))} \tag{1}$$

变量 H 决定了每一阶段探索模式的切换，其初始值为 0.5。$T=0.05^{x.times}$ 是模拟退火算法中的温度，$x.times$ 是种子被选择的次数，随着被选择次数的增加，T 的值会逐渐减小。初始种子的能量 K_{ori} 遵从原始 AFL 的能量赋予规则，即根据覆盖率信息进行分配，此时属于发散探索模式。随着种子被选择次数的增加，H 越来越接近于种子评分 *s_score*，种子能量 K 也随之变化，从基于覆盖率引导的调度策略切换至种子分值引导的调度策略，此时属于收敛探索模式。

算法 2 动态能量分配算法

输入：Program S′，SeedPool P，testcase x / * S′是带有插桩信息的二进制文件，P 是原始种子池，x 是要被赋予能量的种子 * /

输出：Power K of x / * K 是种子的能量，能量越高被调度器选择的可能性就越大 * /

1. (s_score, s_order)=Run(S′, x)
2. s_score(x)=Standard(s_score, P)
3. H=(1−T)s_score+0.5T
4. $K=K_{ori}\cdot 2^{\partial(2H-1)}$

3.3 基于种子类别的分阶段路径探索策略

由 3.2 节可知种子被赋予了种子分值 *s_score* 与种子序列标签 *s_order*，基于此提出了基于种子类别的分阶段路径探索策略。首先把综合得分超过设定阈值的种子定义为引导型种子(vectoring)，在路径探索阶段引导型种子主要负责使路径探索过程快速趋近预定目标。为了防止变异策略破坏其基本结构，引导型种子有不同于普通种子的变异策略(见 3.4 节)。把 *s_order* 相同的引导型种子放入同一个 *Order_seed_queue* 内，在相应的 *Order* 探索阶段再启用对应的 *Order_seed_queue* 种子队列，评分低于设定阈值的种子都将放入 *Order*1 队列中。具体来说，根据划分的种子类别，针对不同的崩溃栈函数分阶段进行路径探索。每一阶段分为发散探索模式和收敛探索模式。

如图 2 所示，这是针对漏洞 CVE-2021-36584 的分阶段路径探索示意图。根据算法 3，对其进行分阶段路径探索。每一个 *Order* 种子序列代表了一个探索阶段，以 *Order*2 为例，在此阶段我们只注重探索函数 *HintFile* 至函数 *gf_hinter_track_process* 之间的程序路径。首先启用 *Order*2 序列中的引导型种子，将其与之前的种子队列合并。探索路径时先进入发散探索模式，此时的目标是尽可能多地探索程序路径，覆盖基本块数多的种子被赋予更多的能量。随着种子被选择次数的增加，T 值会逐渐减少，对应的种子能量 K 则越来越接近于原本的 *s_score* 分值，此时进入收敛探索模式，目的是使发散的路径向目标函数靠拢。

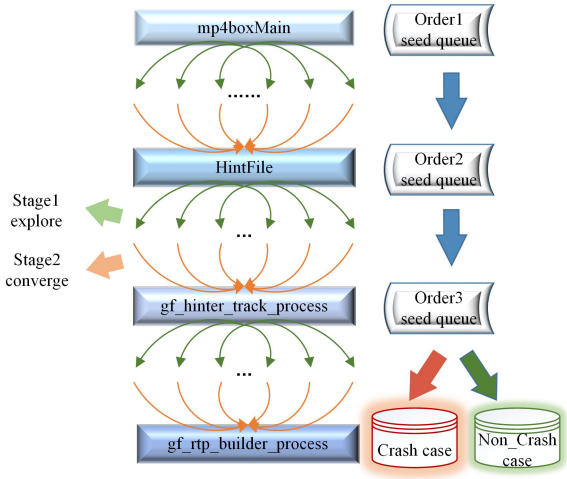

图 2　分阶段路径探索图

Fig. 2　Phased path exploration diagram

算法 3　分阶段路径探索算法

输入：Program S′，SeedPool P /* S′是带有插桩信息的二进制文件，P 是原始种子池 */

输出：Test-suit L /* L 是包含崩溃测试输入和非崩溃测试输入的测试用例 */

```
1. Num←seed_queue_num
2. /* Num 代表 order 队列的数量 */
3. U←seed_queue
4. /* U 是所有测试用例的集合 */
5. Order←Ø
6.    /* Order 代表当前探索的阶段 */
7. While Order ≤Num
8.    for each tests∈U do
9.       Power_Schedule(seed_queue)
10.      /* 依据算法 2 的动态能量分配策略
11.      为种子赋予能量 */
12.      Explore()
13.      /* 首先进入发散探索模式 */
14.      Converge()
15.      /* 随着种子能量的变化逐渐转入收敛
16.      探索模式 */
17.      if Order is Num then
18.         Output(Crash_tests,Noncrash_tests)
19.         /* 若处于最后一个 Order 序列的探索中，则输出 test-suit
L */
20.      if Pass>P_out or time>t_max then
21.         /* 若 Pass 或 time 超过设定阈值则结束当前 Order 阶段探索 */
22.      Order←Order+1
23.      U←U∪Order_seed_queue
24.      /* 将当前的种子队列与之前的合并 */
25.      end for
26. end while
```

为了兼顾探索时的效率，我们引入了一个执行程度衡量标准，定义为函数 *Accessibility*。根据 *Accessibility* 函数来判断当前探索阶段是否应该结束，函数的输入为 3 个值 P，Q，M。定义分别如下：

$$Pass=Accessibility(P,Q,M)\begin{cases}P=\dfrac{\sum_{\alpha=1}^{n} seed\_selected_{\alpha}}{seed\_num}\\ Q=\dfrac{\sum_{\beta=1}^{n} seed\_reached_{\beta}}{seed\_num}\\ M=\dfrac{\sum_{\chi=1}^{n} seed\_mutated_{\chi}}{seed\_num}\end{cases}$$

其中，P 代表此时种子平均被选择的次数，Q 代表此时到达目标块的种子占种子总数的比例，M 代表种子的平均突变次数。

通过 *Accessibility* 函数可以综合反映当前阶段的执行状态。具体计算方法如下：当 P 或 Q 或 M 任意一个指标达到设定阈值时，结束当前状态的探索。在本文的实验中，相应阈值设定为 $P_{max}=500$，$Q_{max}=50\%$，$M_{max}=2\,000$。若单个值未达到阈值，但 $Pass=k*Q+(1-k)*(P+M)$ 的计算结果大于设定的阈值 P_{out} 或当前阶段探索时间超过设定时间阈值 t_{max} 时，同样结束当前阶段的路径探索，比例系数 k 设定为 0.6。Q 的比例系数高是因为到达目标块的种子比例是判断当前阶段是否结束的最佳评判指标。但在某些情况下，路径探索较为复杂，Q 指标上升较慢，因此在兼顾效率的前提下还要结合 P 指标与 M 指标来综合判断是否结束当前探索阶段。

当前阶段探索结束后，*Order*2 种子序列会合并到 *Order*3 种子序列中。最后一阶段的路径探索中会把生成的崩溃测试用例与非崩溃测试用例保存下来构成最终的测试用例集。

3.4　基于种子类别的变异策略分配

传统的模糊测试技术预定义了许多变异操作，但盲目地选择不合适的变异策略会导致路径探索效率低下的问题。

如表 1 所列，传统模糊测试工具 AFL 预定义了 11 种突变操作。其变异调度策略分为 3 个不同的阶段，分别是确定性阶段（Deterministic stage）、破坏阶段（Havoc stage）和拼接阶段（Splicing stage）。

AFL 在确定性变异阶段花费了大量的时间，而研究表明[17]破坏阶段的突变操作在生成有趣的（覆盖率高）测试用例方面更有效。因此，针对引导型以外的普通型种子，我们仅保留了破坏阶段和拼接阶段两种变异算子，以便在短时间内探索大量不同的程序路径。

引导型种子起着定向路径探索的作用，Havoc 和 Splice 阶段的变异算子很可能破坏其原有结构从而影响定向引导的效果。为了保证其原始结构不被过度破坏，我们针对这类种子只保留了 Deterministic 阶段的变异算子。

表 1 由 AFL 定义的突变算子

Table 1 Mutation operators defifined by AFL

Name	Operators	Type
Bitflip	bitflip 1/1, bitflip 2/1, bitflip 4/1	Deterministic
Byteflip	bitflip 8/8, bitflip 16/8, bitflip 32/8	Deterministic
Arithmetic	arith 8/8, arith 16/8, arith 32/8	Deterministic
Interesting values	interest 8/8, interest 16/8, interest 32/8	Deterministic
User extras	user(over), user(insert)	Deterministic
Auto extras	auto extras(over)	Deterministic
Random bytes	random byte	Havoc
Delete bytes	delete bytes	Havoc
Insert bytes	insert bytes	Havoc
Overwrite bytes	overwrite bytes	Havoc
Cross over	cross over	Splice

由于 AFL 随机选择字节进行变异的策略效率并不高，因此本文针对引导型种子定义了一种基于字节敏感的变异策略。具体来说，在初始阶段我们为种子的每个字节都赋予初始分值 0。当一个新的种子 x' 被原始种子 x 生成后，Dgenerate将根据 3.1 节所述的插桩信息计算 x' 的 s_score，若 x' 的 s_score 比原始种子 x 高，Dgenerate 则把这次突变中被改变的字节定义为具有导向性的敏感字节并赋予其较高的分值。当再进行突变时，模糊器会以一定的概率选择分值高的字节进行突变。当一个输入包含大量字节时，找出输入中分值最大的字节会消耗过多时间。为此我们定义了一个字节概率数组 Pr[i]，i 代表输入 x 中的第 i 个字节，Pr[i]的值代表该字节被选择的概率，概率计算式为 $score(i)/\sum_{j=1}^{N} score(j)$，其中 $score(i)$ 为第 i 个字节的分数，N 为当前输入 x 中的字节总数。每次循环迭代中随机从区间[0,1]中选取一个比较值 n，再随机从字节概率数组中选取一个元素 Pr[i]与 n 比较，如果 $Pr[i]>n$，则选取输入中的第 i 个字节进行变异，否则选择第 $N-i$ 个字节进行变异。通过这种方法，可以在 O(1)的时间复杂度内解决突变字节的选择问题。

3.5 基于动态执行的排序定位策略

在排序定位阶段，把生成的高质量测试用例作为输入，重新在原程序上运行，并利用 Intel Pin[18] 跟踪程序执行，记录程序语句运行时的信息，此时的插桩粒度为代码行级别。值得一提的是，我们并不跟踪主程序之外的代码块，如共享库中的运行信息，这极大地减少了追踪时的开销。

接下来利用跟踪得到的信息，计算体现每条指令与缺陷根源相关程度的可疑度评分。程序语句与缺陷根源的相关性主要与两个量有关，分别是该语句在崩溃测试用例中出现的频次和该语句没有在非崩溃测试用例中出现的频次。

具体需要统计以下几个值：1)致使程序崩溃的测试用例中观察到语句 vi 出现的测试用例数量($Crash_{vi}$)；2)所有致使程序崩溃的测试用例数量($Crash_{all}$)；3)致使程序非崩溃的测试用例中没有观察到语句 vi 出现的测试用例数量($Non\_Crash_{\_vi}$)；4)所有致使程序非崩溃的测试用例数量(Non_Crash_{all})。

利用以上 4 个值来计算语句 vi 的重要性评分：

$$Importance\_score=\left(\frac{Crash_{vi}}{Crash_{all}}+\frac{Non\_Crash_{\_vi}}{Non\_Crash_{all}}\right)*0.5$$

图 3 中的示例是一个 MP4 解析软件中的真实漏洞，其根源在第 8 行 data[2]。正常用户的合法输入中，$data\_size$ 如果存在，则一定是大于 2 的，但恶意构造的输入 $data\_size$ 却可能小于 2。软件开发者并没有考虑到这种情况，因此当 $data\_size<2$ 时第 8 行的 $data$[2]便会出现越界访问，导致堆溢出漏洞。

```
1. for(i=1;i<arge;i++)
2. {
3.   …
4.     if(txt_size>2)
5.     {
6.       …
7.       if(data_size<4) return GF_NON;
8.       if(u8) data[2]==(u8)0xFE)
9.       {
10.          is_utf_16=GF_TRUE;
11.    }
12. }
13.}
```

图 3 MP4 解析器漏洞代码图

Fig. 3 MP4 parser vulnerability code diagram

为了具体说明计算过程，针对图 3 我们根据 Dloc 生成的数据给出计算第 8 行代码的重要性评分的过程。我们生成的测试用例中 $Crash_{all}$ 的数量为 1930，$Crash_{v8}$ 的数量为 1918(此处 v8 即指代第 8 行代码)，Non_Crash_{all} 的数量为 2503，$Non\_Crash_{\_v8}$ 的数量为 2440。由此可得：

$$Importance\_score=\left(\frac{1918}{1930}+\frac{2440}{2503}\right)*0.5\approx0.984$$

4 评估

本节采用真实软件中的 CVE 漏洞进行测试，把软件维护人员打补丁的位置认定为漏洞根源，并与 Dloc 分析得出的位置做比较来评估测试用例生成方法对缺陷根源定位的有效性。同时将实验结果与同类研究进行对比分析。实验设计旨在回答以下几个问题：

(1)在真实软件中应用时，本系统所生成的测试用例是否能帮助定位出缺陷根源？

(2)测试用例生成技术 Dgenerate 生成的测试用例质量如何评估？是否比前人的测试用例生成技术所生成的测试用例质量更高？

(3)与同类研究相对比，Dgenerate 所生成的测试用例能否更好地协助定位缺陷根源？

为探究以上问题，本文在 ubuntu 上选择了不同类型的 6 款常用软件进行实验，开源项目信息如表 2 所列。

表 2 实验所采用的开源项目

Table 2 Open-source projects for experiment

Program	Size/MB	Lines of code
gpac	797.6	792222
libzip	24.7	76854
tsmuxer	54.9	57141
libde265	40.3	53198
Bento4	46.6	96114
libxml2	156.3	609678

实验环境配置为：Ubuntu 20.04.3 64 位操作系统，Intel

(R) Xeon(R) Silver 4216 CPU @ 2.10 GHz 处理器，128 GB 内存。测试用例生成时间统一为 6h，其中基础种子库由触发漏洞的 POC 文件进行 1h 的 AFL crash exploration mode 生成。

4.1　定位效果实验

我们选择实验对象的条件有 3 个：1）二进制程序可以被 Dloc 所插桩分析；2）有触发漏洞相关的 POC 文件；3）软件维护者已为此漏洞打上补丁。

为了确保实验数据的多样性，我们选取 6 款不同类型的软件中的 15 个 CVE 漏洞作为实验对象，涵盖了堆溢出、栈溢出、双重释放、整数溢出、段错误、释放后重利用和可达断言共 7 种不同的漏洞类型。

如表 3 所列，实验以软件维护人员打补丁位置为基准来评估 Dloc 定位的准确性。

表 3　Dloc 缺陷定位实验结果

Table 3　Results of Dloc for fault localization

Program	CVE ID	Bug Type	TOP5?	Rank	BS
gpac	cve-2021-36854	heap-buffer-overflow	√√	1	1.000
	cve-2021-36417	heap-buffer-overflow	√√	1	1.000
	cve-2021-36412	heap-buffer-overflow	×	—	<0.900
	cve-2021-36414	heap-buffer-overflow	√√	2	0.989
libzip	cve-2017-12858	double-free	√	5	0.998
tsmuxer	cve-2021-35346	heap-buffer-overflow	√√	4	0.984
	cve-2021-45860	integer-flow	√√	1	1.000
libde265	cve-2021-35452	segmentation-fault	√	3	0.986
	cve-2021-36408	heap-use-after-free	√√	1	1.000
	cve-2021-36409	reachable-assertion	√√	5	0.970
	cve-2021-36411	segmentation-fault	√√	3	0.978
	cve-2021-36410	stack-buffer-overflow	√	4	0.973
Bento4	cve-2021-35306	segmentation-fault	√√	2	0.992
	cve-2021-35307	segmentation-fault	√√	1	0.995
libxml2	cve-2021-3516	heap-use-after-free	×	—	<0.900

“√√”表明评分前五的位置中有和补丁位置相同的代码片段；“√”表明评分前五的位置中有与补丁位置语义相同的代码片段，即在此处打上补丁可以达到与软件维修人员补丁相同的效果；“×”代表评分前五的位置中没有与补丁位置一样或语义相同的代码行；Rank 是缺陷根源语句在所有可疑语句中的排名；BS 是所有被评分语句中可疑度评分的最高得分。

从表 3 中的结果可以看出，Dloc 可以在排名前五的位置中有效定位出 13 个真实 CVE 漏洞的根源，定位成功率高达 87%。且结果涵盖了不同类型软件的不同的漏洞种类，这表明 Dloc 的定位效果与软件类型或漏洞种类无关。

在对 cve-2021-36412 定位过程手动分析时可以发现，其生成的崩溃测试用例包含不止一种根源的崩溃，这是由于此漏洞原始崩溃点附近还存在其他根源导致的崩溃，因此对测试用例生成及最后统计分析的结果造成了干扰。在对 libxml2 的定位过程手动分析时，我们发现 cve-2021-3516 的原始 POC 文件可以触发两个不同根源的漏洞，导致崩溃路径并不唯一，干扰了测试用例生成的质量，这是 Dloc 定位失败的主要原因。

4.2　测试用例质量实验

本节研究了不同工具生成的测试用例质量。为了评估测试用例质量，我们设计了两个实验，分别为：

实验 1　测试用例定位效率实验

实验 2　测试用例可区分度实验

测试用例定位效率实验：假设软件维护人员按照可疑度评分从高到低的顺序对语句进行排查，本实验统计了使用 Dloc，Aurora，VulnLoc 和 AFLGO 生成的测试用例进行根源定位的过程中发现根源前需要检查的代码行数量，需要检查的代码行数越少，代表测试用例定位效率越高。其中进行定位效率实验时的目标漏洞均是表 3 中所提及的漏洞，未提及的其他漏洞不在本实验评估范围之内。为了控制变量，此处运用测试用例进行根源定位时均采用 3.5 节中描述的排序定位策略。

4 种测试用例生成技术在不同目标软件上的实验结果如表 4 所列，$\%sl$ 表示在不同软件中排名前五的位置被成功定位根源的漏洞占此软件中所有进行定位实验的漏洞比例，我们把能在排名前五的位置中输出根源相关位置的定位实验视为成功的定位实验，此指标表明了同一软件中定位实验成功的比例，体现了本方法的有效性；$\#stmts$ 表示为了定位每个漏洞根源，需要检查的程序语句平均数量，需要人工检查的语句数量越少代表在根源定位过程中需要的人工干预越少，体现了本方法所带来的效率提升；$\%stmts$ 表示为定位每个漏洞需要检查的语句平均数量占软件总代码行数的比例，此指标表明了本方法帮助人工将需要检查的代码数量缩减到一个可以接受的比例内，从而印证了方法的实用性。

表 4　测试用例定位效率实验结果

Table 4　Results of test case location efficiency

Program	Aurora			VulnLoc			AFLGO			Dgenerate		
	$\%sl$	$\#stmts$	$\%stmts$	$\%sl$	$\#stmts$	$\%stmts$	$\%sl$	$\#stmts$	$\%stmts$	$\%sl$	$\#stmts$	$\%stmts$
gpac	50.0	35.5	0.0045	50	31.00	0.0039	50.0	49.0	0.0062	75	23.00	0.0029
libzip	0.0	—	—	0	—	—	0.0	—	—	100	5.00	0.0065
tsmuxer	50.0	17.5	0.0310	50	14.50	0.0250	50.0	37.0	0.0640	100	2.50	0.0043
libde265	40.0	42.5	0.0790	60	19.60	0.0370	40.0	17.8	0.0330	100	3.20	0.0060
Bento4	0.0	—	—	50	25.50	0.0270	50.0	22.5	0.0230	100	1.50	0.0016
libxml2	0.0	—	—	0	—	—	0.0	—	—	0	—	—
Average	23.3	29.6	0.0340	35	22.65	0.0230	31.6	31.2	0.0310	79	5.86	0.0043

结果表明使用 Dgenerate 策略生成的测试用例定位效率最高。虽然所有策略都能将需要检查的代码数量控制在程序

语句总量的1%以内，但在根源定位时Dgenerate平均只需要检查5.86个程序语句，相比其他3种策略平均需要检查27.8个程序语句，效率约提升了75%。

测试用例可区分度实验：低质量测试用例会引入许多与缺陷触发无关的代码路径，很难区分出缺陷根源相关的代码片段位置，这体现在许多程序代码存在着相同的排序分数上。高质量的测试用例则可以很好地区分缺陷相关代码片段的不同位置，使得在定位排序时可以在排名靠前的位置中把缺陷根源相关的代码片段统计出来。

我们把相同分数的代码片段称为一个集合，并在相同的实验环境中统计了由不同工具生成的测试用例的平均集合数量。图4给出了随着测试用例使用比例的增加，不同工具所生成的测试用例的平均集合数量变化。

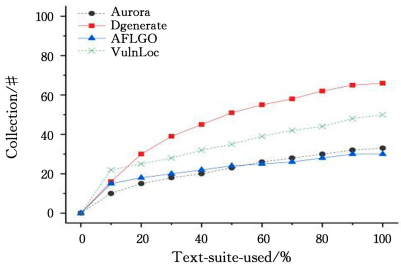

图4 测试用例集合数量变化图

Fig. 4 Change in the number of test case sets diagram

结果表明随着测试用例使用比例的升高，Dgenerate生成的测试用例所产生的集合数量最多，即应用此测试用例后代码的可区分度最强。

以上两个实验分别从定位效率与代码可区分度两个方面评估了4种不同技术所生成的测试用例质量，结果表明Dgenerate技术所生成的测试用例综合评估质量最高。

4.3 综合对比实验

为了表明高质量的测试用例可以更好地定位根源，我们对比评估了由Aurora，AFLGO，VulnLoc和Dgenerate产生的测试用例在同等条件下进行根源定位的效果。以下实验皆采用第4节中的排序定位算法。

实验结果如表5所列，√代表在评分前5的位置中能定位到漏洞根源或有与补丁位置语义相同的代码片段，×则代表不能在评分前五的位置中定位到漏洞根源。

(1)与AFLGO对比

将崩溃位置设置为目标点，在相同的实验环境下把所有由AFLGO生成的输入文件作为测试用例。由于AFLGO的运行过程依赖于内部控制流图的构建，运行过程代价较高，因此6h内其生成的测试用例数量有限。运用AFLGO生成的测试用例作为输入，以相同的排序定位算法进行排序时，只能在排名前五的位置中定位出gpac中的cve-2021-36854，cve-2021-36417，cve-2021-36414，tsmuxer中cve-2021-35346，libde265中的cve-2021-36409，cve-2021-36411和Bento4中的cve-2021-35306。通过人工分析发现，定向模糊工具生成的测试输入偏向于崩溃位置，并且可能引入与目标漏洞不同根源的其他漏洞。实验结果表明定向模糊技术并不能直接应用于测试用例的生成。

表5 不同测试用例的缺陷定位实验结果

Table 5 Results of different test cases for fault localization

CVEID	AFLGO	Aurora	VulnLoc	Dloc
cve-2021-36854	√	√	√	√
cve-2021-36417	√	×	√	√
cve-2021-36412	×	√	×	×
cve-2021-36414	√	×	×	√
cve-2017-12858	×	×	×	√
cve-2021-35346	√	×	√	√
cve-2021-45860	×	×	×	√
cve-2021-35452	×	×	√	√
cve-2021-36408	×	√	×	√
cve-2021-36409	√	×	√	√
cve-2021-36411	√	×	√	√
cve-2021-36410	×	√	√	√
cve-2021-35306	√	×	√	√
cve-2021-35307	×	×	×	√
cve-2021-3516	×	×	×	×

(2)与Aurora对比

在相同的实验环境下，我们运用Aurora生成的测试用例对6款软件进行测试实验，并且运用相同的排序定位方法，其只在排名前五的位置中成功定位出了gpac中的cve-2021-36854，cve-2021-36412和libde265中的cve-2021-36408，cve-2021-36410的漏洞根源。由于其测试用例是利用AFL crash exploration生成的，通过分析发现，Aurora生成的测试用例中包含许多与当前漏洞根源无关的执行路径。

比如在对cve-2021-36417的实验中，其生成的测试用例包含了可以触发4种与目标漏洞不同根源的崩溃测试用例和大量与原始崩溃路径完全不相干的非崩溃测试用例。造成这种情况的原因是其测试用例生成模式仅仅通过变异原始POC文件进行，变异策略随机，且没有对探索路径进行约束，而仅由覆盖率进行引导。若其变异POC文件使路径探索进入崩溃栈外层函数的其他分支，在覆盖率的引导下探索会偏离原始崩溃路径从而进入完全不相干的程序路径。用这种测试用例作为输入时，根源定位结果的准确性会受到严重干扰，无法区分出与漏洞根源相关的代码片段。

(3)与VulnLoc对比

在相同的实验环境下，我们运用VulnLoc生成的测试用例对6款软件进行实验，在相同的评分算法中，其在排名前五的位置中成功定位出了gpac中的cve-2021-36854，cve-2021-36417，tsmuxer中的cve-2021-35346，lidbe265中的cve-2021-35452，cve-2021-36410，cve-2021-36409，cve-2021-36411，Bento4中的cve-2021-35306。

由于VulnLoc的测试用例生成方法过度关注与原始POC崩溃路径相似的路径而缺少探索路径的多样性，因此其生成的测试输入与原始崩溃路径存在着过拟合问题，这会导致路径上与漏洞根源并不相关但却一定会被执行的语句评分过高。如在定位libzip中的cve-2017-12858时，其评分最高的代码行是一段与根源不相关却在崩溃路径上的循环处理代码，而真正的漏洞根源代码排序在36名的位置。这表明过拟合的测试用例得到的统计结果会大幅增加软件维修人员需要检查的代码数量，降低整体效率。

Dgenerate的测试用例生成策略很好地平衡了路径探索中发散与收敛的问题，极大地降低了这种情况发生的可能性。

结束语　本文提出了一种测试用例生成技术 Dgenerate，并在 Dloc 中实现了该技术。其运用了基于崩溃路径的信息插桩策略、种子类别划分以及动态能量调度算法，可以有效地生成高质量测试用例，极大提升了缺陷根源定位的准确性。

实验结果表明，基于崩溃路径探索的分阶段测试用例生成技术所生成的测试用例与之前的测试用例生成技术所生成的测试用例相比质量更高，运用这种测试用例技术进行根源定位可以将效率平均提升 75%，并且 Dloc 能够以 87%的准确性在评分前五的位置中输出缺陷根源相关代码片段。

虽然有着严格的路径约束，但若原始 POC 崩溃地点附近有其他原因导致的崩溃，Dgenerate 技术也会将其捕捉到崩溃测试用例中，这会对下一步的统计分析工作造成干扰，导致准确性降低。在未来的工作中，我们将着重解决非同根源性的崩溃测试用例筛选问题，将无关崩溃项剔除，并持续优化调度算法，降低开销，进一步提升系统执行的效率。

参 考 文 献

[1] SASAKI Y，HIGO Y，MATSUMOTO S，et al. SBFL-Suitability：A Software Characteristic for Fault Localization[C]// IEEE International Conference on Software Maintenance and Evolution(ICSME). 2020：702-706.

[2] ERIC W，GAO R Z，LI Y H，et al. A Survey on Software Fault Localization[C]// IEEE Transactions on Software Engineering. 2016：707-740.

[3] XIE X，CHEN T Y，KUO F C，et al. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization[J]. ACM Transactions on Software Engineering and Methodology(TOSEM)，2013，22(4)：1-40.

[4] HIGOR A，MARCOS L，FABIO K. Spectrum-based software fault localization：A survey of techniques，advances，and challenges[J]. arXiv：1607. 04347，2016.

[5] WEN W Z，LI B X，SUN X B，et al. Technique of software fault localization based on hierarchical slicing spectrum[J]. Journal of Software，2013，24(5)：977-992.

[6] LIU C，YAN X，FEI L，et al. SOBER：statistical model-based bug localization [J]. ACM SIGSOFT Software Engineering Notes，2005，30(5)：286-295.

[7] JONES J A，HARROLD M J. Empirical evaluation of the tarantula automatic fault-localization technique[C]// Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. 2005：273-282.

[8] ABREU R，ZOETEWEIJ P，VAN GEMUND A J C. On the accuracy of spectrum-based fault localization[C]// Testing：Academic and Industrial Conference Practice and Research Techniques-Mttation(TAIC PART-Mutation 2007). IEEE，2007：89-98.

[9] WONG W E，DEBROY V，GAO R，et al. The DStar method for effective software fault localization[J]. IEEE Transactions on Reliability，2013，63(1)：290-308.

[10] ARTZI S，DOLBY J，TIP F，et al. Directed test generation for effective fault localization[C]// Proceedings of the 19th International Symposium on Software Testing and Analysis. 2010：49-60.

[11] SCHWARTZ E J，AVGERINOS T，BRUMLEY D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution(but might have been afraid to ask)[C]// 2010 IEEE Symposium on Security and Privacy. IEEE，2010：317-331.

[12] BLAZYTKO T，SCHLÖGEL M，ASCHERMANN C，et al. AURORA：Statistical Crash Analysis for Automated Root Cause Explanation[C]// 29th USENIX Security Symposium(USENIX Security 20). 2020：235-252.

[13] MICHAEL Z. afl-fuzz：crash exploration mode [EB/OL]. https：//lcamtuf. blogspot. com/2014/11/afl-fuzz-crash-exploration-mode. html.

[14] SHEN S Q，KOLLURI A，DONG Z，et al. Localizing Vulnerabilities Statistically From One Exploit[C]// Proceedings of the ACM Asia Conference on Computer and Communications Security. 2021：537-549.

[15] MICHAŁ Z. American fuzzy lop [EB/OL]. http：//lcamtuf. coredump. cx/afl/.

[16] BÖHME M，PHAM V T，NGUYEN M D，et al. Directed greybox fuzzing[C]// Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017：2329-2344.

[17] LYU C，JI S，ZHANG C，et al. MOPT：Optimized mutation scheduling for fuzzers[C]// 28th USENIX Security Symposium (USENIX Security 19). 2019：1949-1966.

[18] Intel Corporation. Pin－a dynamic binary instrumentation tool [EB/OL]. https：//software. intel. com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

DU Hao，born in 1997，postgraduate. His main research interests include reverse engineering and vulnerability mining.

WANG Yunchao，born in 1992，Ph. D. His main research interests include reverse engineering and vulnerability mining and exploitation.

（责任编辑：何杨）