软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(5):2286−2299 [doi: 10.13328/j.cnki.jos.006491]

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

面向递增累积型缺陷的灰盒模糊测试变异优化*

杨 克[1,2], 贺也平[1,2,3], 马恒太[1,2], 董 柯[1,2], 谢 异[1,2], 蔡春芳[1,2]

[1](中国科学院 软件研究所 基础软件国家工程研究中心, 北京 100190)

[2](中国科学院大学, 北京 100049)

[3](计算机科学国家重点实验室 (中国科学院 软件研究所), 北京 100190)

通信作者: 贺也平, E-mail: yeping@iscas.ac.cn; 马恒太, E-mail: hengtai@iscas.ac.cn

摘 要: 大量访问越界、内存耗尽、性能故障等缺陷是输入中有效数据的规模过大, 超过临界值引起的. 而现有灰盒模糊测试技术中的数据依赖识别和变异优化技术大都针对固定规模输入数据格式, 对规模递增输入数据的构造效率不高. 为此, 针对这类累积型缺陷模糊测试对应的状态特征值最优化问题, 提出一种对特征值依赖的输入数据的格式判别和差分变异方法. 根据引发特征值最值更新的有效变异的位置分布和发现频次特征, 判别待发现缺陷状态优化是否依赖于输入中相关数据规模的增长, 将引发最值更新的有效变异内容应用于规模递增输入数据生成, 提升该类累积型缺陷的复现和定向测试效率. 依据该思想, 实现了模糊测试工具 Jigsaw, 在测评实验数据集上的实验结果表明提出的判别方法能够高效地区分特征值依赖的输入数据组织形式, 且提出的差分变异方法显著提升了需要大量输入才能触发累积型缺陷的复现效率.

关键词: 定向测试; 最优化; 累积型缺陷; 规模递增; 格式判别; 差分变异

中图法分类号: TP311

中文引用格式: 杨克, 贺也平, 马恒太, 董柯, 谢异, 蔡春芳. 面向递增累积型缺陷的灰盒模糊测试变异优化. 软件学报, 2023, 34(5): 2286–2299. http://www.jos.org.cn/1000-9825/6491.htm

英文引用格式: Yang K, He YP, Ma HT, Dong K, Xie Y, Cai CF. Mutation Optimization of Directional Fuzzing for Cumulative Defects. Ruan Jian Xue Bao/Journal of Software, 2023, 34(5): 2286–2299 (in Chinese). http://www.jos.org.cn/1000-9825/6491.htm

Mutation Optimization of Directional Fuzzing for Cumulative Defects

YANG Ke[1,2], HE Ye-Ping[1,2,3], MA Heng-Tai[1,2], DONG Ke[1,2], XIE Yi[1,2], CAI Chun-Fang[1,2]

[1](National Engineering Research Center of Fundamental Software, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

[3](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

Abstract: Many quantifiable state-out-of-bound software defects, such as access violations, memory exhaustion, and performance failures, are caused by a large quantity of input data. However, existing dependent data identification and mutation optimization technologies for grey-box fuzzing mainly focus on fixed-length data formats. They are not efficient in increasing the amount of cumulated data required by the accumulated buggy states. This study proposes a differential mutation method to accelerate feature state optimization during the directed fuzzing. By monitoring the seed that updates the maximum or minimum state value of the cumulative defects, the effective mutate offset and content are determined. The frequency is leveraged and the distribution of the effective mutation is offset to distinguish whether the feature value of the defect depends on a fixed field or cumulative data in the input. The effective mutation content is reused as a material in the cumulative input mutation to accelerate the bug reproduction or directed testing. Based on this idea, this study implements the fuzzing tool Jigsaw. The evaluation results on the experimental data set show that the proposed dependency detection method can

* 基金项目: 中国科学院战略性先导科技专项 (XDA-Y01-01, XDC02010600)
收稿时间: 2021-03-14; 修改时间: 2021-06-17; 采用时间: 2021-08-30; jos 在线出版时间: 2022-10-14
CNKI 网络首发时间: 2022-11-15

efficiently detect the input data type that drives the feature value of cumulative defects and the mutation method significantly shorten the reproduction time of the cumulative defect that requires a large amount of special input data.

Key words: directed fuzzing; optimization; cumulative defects; cumulative input; format discrimination; differential mutation

1 引 言

近几年, 基于变异的模糊测试技术 AFL、HongFuzz 和 LibFuzzer[1]等在各种类型软件的漏洞挖掘中取得了很好的效果. 但现代软件非常庞大, 进行完整程序测试非常耗时. 随着研究的深入, 越来越多的学者开始关注面向错误复现、变更代码测试[2,3]或特定类型缺陷检测[4−7]等特殊场景下的定制化模糊测试问题. 使用待发现缺陷状态的位置、代码、状态变化等特点优化模糊测试的设计, 提高测试有针对性, 已经成为当下灰盒模糊测试研究的新热点.

相比传统模糊测试, 该类应用仅关注于目标缺陷或代码相关的部分程序逻辑切面, 因此利用目标相关特征缩小分析和测试范围, 提高测试效率, 是该类应用问题特有的研究内容. 缺陷特征导向的灰盒模糊测试将目标缺陷检测问题转化为种子的最优化问题. 其通过刻画目标缺陷相关特征, 构造关于种子的最优化函数, 并改进种子度量, 优化种子生成, 从而提升目标缺陷的检测效率.

许多研究[8−12]指出集成在 AFL 等流行模糊测试工具中的随机种子变异方法很难求解某些特殊值约束. 如果缺陷状态依赖大量特殊值输入, 传统的随机变异方法就很难发现. 访问越界[13] (CWE-787/CWE-125/CWE-788/CWE-124)、内存耗尽[4]、性能故障[14,15]等缺陷是因某些状态累积效应超过阈值时产生的 (累积型缺陷). 如果缺陷状态依赖于大量特殊的输入, 就会因搜索空间过大, 有效数据占比较少而很难被随机变异策略发现. 我们对 Wen 等人[4]在内存耗尽型缺陷检测工作实验评估中的数据进行了分析. 发现, 在相同的工具 (随机变异策略也相同) 下, 少部分缺陷的复现时间为几分钟, 而大部分缺陷的复现时间为数小时甚至超过一天. 在性能故障缺陷检测案例[14]以及 CVE 官网中访问越界相关漏洞的检测报告中也发现了类似的现象.

进一步分析发现, 复现时间的长短与缺陷状态所依赖的输入数据形式密切相关. 一些缺陷状态由输入中的个别字节直接决定, 表征缺陷状态逼近程度的目标函数值会因这些字节内容的改变而大幅变化. 传统的随机变异生成每个种子时通常会进行大量的变异操作, 缺陷依赖的字节很容易在该过程中发生改变, 从而较快地实现对目标函数值的搜索优化. 而另一些缺陷在输入中有效数据超过一定规模时才触发, 每个有效数据元素对累积效应 (目标函数值) 的增幅有限. 此时, 需要在更大的输入空间内搜索缺陷依赖的有效输入内容, 而且基于大量随机变异生成新种子的传统策略很容易破坏既有的有效数据, 由于搜索空间非常大, 对输入内容要求更苛刻, 传统随机变异方法很难触发这类依赖有效数据规模增长的累积型缺陷.

现有的面向访问越界、内存耗尽、性能故障等缺陷的模糊测试工作, 主要利用反映缺陷状态逼近度的数值特征作为待优化的目标函数, 通过改进遗传算法中的种子生成策略, 促进缺陷状态向临界值逼近从而触发缺陷. 这些数值状态特征包含内存占用[4]、执行路径长度[5]、控制流频次[14]、CPU 占用率[15]或危险的 API 或语句的覆盖比率[16−18]等. IJON[19]提供了定制目标函数的原语和爬山优化机制, 允许测试人员利用程序中的变量或表达式定制指派待优化的变量或函数表达式. 然而, 这些研究工作并未对数值状态所依赖的输入形式做区分, 大都采用一种随机变异策略. 如果缺陷状态依赖于大量包含特殊内容的输入, 随机变异产生有效输入概率就很低.

现有对变异策略的优化研究主要关注筛选有效变异位置[20]或变异算子[21,22], 对有效变异内容的研究较少. 对固定字段导致的缺陷, 有效的变异位置存在复用价值, 但累积输入数据引发的缺陷需要增长数据规模以产生累积效应, 反复变异固定位置反而不利于该类缺陷的检测. 例如文献 [4] 中递归深度过深引发的内存耗尽缺陷 CVE-2018-17985. 其依赖于重复的关键词“P”的个数, 而不是个别固定位置字符“P”.

现有关注输入内容构造的模糊测试工作的研究问题主要是分支比较条件中苛刻字段内容识别, 其目的是增加覆盖率. 由于研究问题不同, 这些工作所提出的方法应用到对规模增长驱动的累积型缺陷的检测上还存在一定困难. 例如, 关注苛刻分支测试的工作[8−12]通过锚定特定的分支比较条件来定位其用到的数据字段, 并根据比较常量

确定满足苛刻条件的字段内容, 利用污点分析[8,12,23,24]或者输入变化与程序状态的关联分析[10,11]定位比较条件依赖的数据字节. 但是, 累积型缺陷常常依赖于大量的循环和递归计算, 逐个对每一次比较运算进行分析的时间和空间开销都是不可接受的. 污点分析等方法则在遇到循环依赖时会产生过污染和欠污染等问题, 因此这些方法并不适合用在依赖规模增长的累积型缺陷检测上. ProFuzzer[25]根据对不同字节值导致的执行路径长度的统计分析 (Probing) 来定位输入中的循环计数字段 (loop count) 和数据大小字段 (size). 该方法可以用来检测这两种固定字段引发的累积型缺陷. 但是, 可变规模输入数据的有效变异位置往往并不固定, 而 Probing 方法也无法有效检测出可变规模输入数据的关键内容, 因此该方法仍然难以直接用于可变输入格式依赖的检测.

由于有效输入数据规模决定了搜索空间的大小, 也决定了随机搜索难易, 本文将累积型缺陷按照其状态的累积效应是否依赖于实际有效数据规模增长进行分类, 将累积型缺陷分为输入规模递增累积型 (递增累积型) 和输入规模固定累积型 (固定累积型) 两类. 由于二者的数据组织形式和搜索空间差异巨大, 因此需要分别设计变异策略.

本文将反映种子与累积型缺陷状态逼近程度的度量值称为特征值, 累积型缺陷的检测问题可以视为是种子特征值的最大化问题. 本文分析了访问越界、内存耗尽、性能缺陷中固定累积型缺陷和递增累积型缺陷的差异, 发现固定累积型缺陷的特征值依赖于个别字节, 同一个字节取值变更可能导致特征值的多次最值更新, 反映递增累积型缺陷的特征值依赖于内容出现的频次, 同一个字节的内容变化通常仅带来一次最值更新. 因此有效变异位置被检测到的频次越少、分布越分散, 越表明该最优化问题是由规模递增输入驱动的. 在此基础上利用有效变更的位置分布和频次特征, 提出了一种区分待检测累积型缺陷依赖于固定的输入规模 (固定累积型特征值) 还是依赖于有效输入规模的增长 (递增累积型特征值) 的判定方法. 针对递增累积型缺陷, 提出了一种基于种子差分的有效变异策略.

首先, 根据待复现或待检测累积型缺陷的代码特征确定其特征值和其对应的变量或表达式, 以该表达式为参数在相应位置插入 IJON 最优化原语. 其次, 在测试时根据 IJON 原语捕获的特征值更新反馈, 收集随机交叉变异的过程有效种子的修改规则, 包含修改位置以及修改后的内容. 并对修改规则进行清洗, 去掉无关修改. 结合同一字节引发的目标函数最值更新次数以及引发最值更新的字节分布情况来区分待检测缺陷的特征值类型 (固定累积型特征值还是递增累积型特征值). 最后, 根据最优化问题依赖的输入形式, 应用不同的变异策略. 特别地, 对递增累积型特征值的最优化, 本文根据引起最值更新种子的差分内容获得递增累积型特征值依赖的输入素材. 借助规则清洗, 将有效变化内容作为字典值以较大的概率应用到后续的交叉变异中, 并贪心地应用成功率高的变异内容, 从而加速递增累积型缺陷依赖的输入数据的生成. 为了避免种子生成时变异次数过多破坏既有的累积输入内容, 对递增累积型缺陷采用较低的变异次数来生成后代. 最后, 通过实验表明本文提出的变异优化策略提高了递增累积型缺陷检测效率. 主要贡献如下.

(1) 提出了规模递增输入数据导致的缺陷问题. 根据有效变异位置和频次特征, 给出了一种对该类输入依赖形式判别的方法, 能够快速有效地区分其依赖于规模递增还是规模固定的输入数据.

(2) 提出了一种基于差分进化的变异策略, 比随机变异更加高效地生成特征值最优化所需的规模递增输入数据, 从而加速递增累积型缺陷的复现和检测.

(3) 实现了基于上述识别方法和变异策略的原型系统 Jigsaw. 实验显示本文提出方法的识别开销小于 5%, 且够准确地区分出待优化特征值依赖的输入数据格式. Jigsaw 在复现临界值越界类缺陷中的递增累积型缺陷时能够比传统随机变异策略有 3 倍以上的效率提升.

2 问题分析

本节通过一个案例来说明引发最优值更新的差异数据有助于提升递增累积型缺陷的发现效率.

例 1: Liblouis UTF 解码时边界检查与操作顺序不当引发的越界访问缺陷 CVE-2018-11683.

图 1 显示了 CVE-2018-11683 的缺陷补丁. 该缺陷因在解析 UTF-8 字符进行特殊的 UTF-32 转换处理时边界检查滞后导致的循环写越界. 图 1 中缺陷代码 1158 行对数组索引 out 的边界检查应该放在 1156 行的内存访问之

前才能起到有效的边界保护. 图 1 代码中 token->chars[in] 来自输入内容, 而 token->chars[out] 则用于存储解析后的结果. 输出缓冲区的最大长度是 MAX_STRING=2048. 在处理正常的 UTF-8 内容时的边界检查已经充分. 但是当 token 足够长且包含足够多的非 UTF-8 内容就会触发访问越界缺陷. 这些非 UTF-8 内容主要由取值为 FF , 或 FE 的字节构成. 这是一个典型的递增累积型缺陷.

```
                @@ −1153,12 +1153,12 @@ parseChars(FileInfo *nested, CharsString *result, CharsString *token) {
1153    1153                    }
1154    1154                    utf32 = (utf32 << 6) + (token->chars[in++] & 0x3f);
1155    1155                }
1156        −               if (CHARSIZE == 2 && utf32 > 0xffff) utf32 = 0xffff;
1157        −               result->chars[out++] = (widechar)utf32;// IJON_MAX(out);
1158    1156                if (out >= MAXSTRING) {
1159    1157                    result->length = lastOutSize;
1160    1158                    return 1;
1161    1159                }
        1160    +           if (CHARSIZE == 2 && utf32 > 0xffff) utf32 = 0xffff;
        1161    +           result->chars[out++] = (widechar)utf32;
1162    1162                }
1163    1163                                        result->length = out;
1164    1164        return 1;
```

图 1　CVE-2018-11683 的补丁

为了验证导致最值函数更新的种子增量的内容是否可复用, 将输出目录中 ijon_max 子目录中引起最值更新的相邻种子差异 (radiff2 -O 的输出) 加入 AFL 的字典中 (对应图 2 中的 IJON′), 并与不使用字典数据的 IJON 进行对比试验. 为了研究随机变异次数对递增规模数据构造效率的影响, 我们构造了一个 Havoc 阶段的循环变异次数仅为 4 次的版本 IJON″(对应图 2 中的 IJON″).

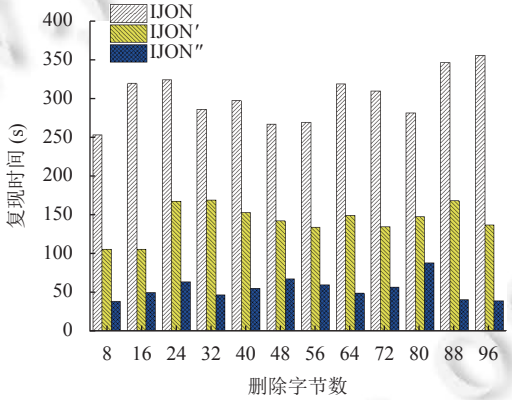

图 2　CVE-2018-11683 初始种子构造时删除 POC 的字节数以及复现时间

初始种子通过删除 POC 上与缺陷有关的字节来构造, 删去 POC 上与缺陷相关的那个 token (总长 1328) 偏移 0x55a 处的连续 $n \in \{8, 16, \ldots, 96\}$个字节, 并以不同的初始种子为起始分别运行试验. 将 IJON 最大化反馈原语, “IJON_MAX(out); ”插入到缺陷版本代码文件 CompileTranslationTable.c 的 1157 行. 使用 AFL 的-S 选项 (跳过确定性测试阶段), 并对每个种子实验 10 次取复现时间的平均值.

得到的平均复现时间如图 2, 其中左斜纹、右斜纹和深色柱分别对应 IJON、IJON′和 IJON″在不同初始种子下对 CVE-2018-11683 的复现时间. 可以看出, 使用引起最值更新的种子差异作为字典素材缩短了复现时间, IJON′比 IJON 平均快 2 倍. 而缩短 Havoc 阶段变异频次的 IJON″则比 IJON 快 6 倍以上.

该案例表明, 检测或复现依赖输入数据规模递增的累积型缺陷时, 降低灰盒模糊测试变异频率, 复用引发缺陷特征值更新的增量数据片段有助于提高缺陷检测效率. 由于 IJON 不能有效地感知到待优化目标函数是否依赖于有效输入数据的规模, 导致其对递增累积型缺陷构造输入内容时较为盲目, 触发效率较低.

与临界值有关的缺陷, 如访问越界、内存耗尽和性能缺陷等, 可以转化为数值状态的最值问题. 如果该度量的粒度够细, 就可以监控最值更新确定有效的累积素材. 这些有价值的输入素材不仅可以应用于构造连续形式的累积输入, 对递归等复杂的累积输入格式内容的构造同样有效.

3 面向递增累积型缺陷的复现方法

3.1 方法原理

累积型缺陷时因为某个内部状态超过阈值引起的. 因此此该类问题通常可以转化为该数值特征的最优化问题. 该问题较适合在灰盒模糊测试框架下借助一定的反馈信息来求解. IJON[19]提供了供测试人员定制其需要进行最值优化的表达式的最值反馈原语 (IJON_MAX/IJON_MIN). 当程序触发插入的 IJON 语句时, 就会触发 IJON 的种子调度策略, 进行爬山优化. IJON 通过单独存储使得数值特征增长的种子, 并挑选数值最优的种子进行优先变异. 在待发现累积型缺陷数值状态变量已知 (或已知如何构造) 的前提下可以用 IJON 提供的最优化机制来解决.

在这种最优化模型下, 依赖输入规模固定的目标函数 (例如, 依赖于个别整数字段), 输入中影响目标函数变化的字节比较集中, 同一个字节取值变更通常会引起目标函数值的多次更新; 而依赖输入规模递增的目标函数, 输入中目标函数变化的字节比较分散, 且大量字节的取值变化仅能影响到目标函数值的 1 次更新. 通过该特点可以区分目标函数对输入的依赖, 从而对不同类型特征值的优化问题分别采用不同的变异策略.

针对递增累积型特征值的优化问题, 本文根据其依赖的增量数据大都存在一定重复性的特点, 提取引起特征值更新的种子与更新前最值种子的差异内容, 并进行复用. 在后续种子的变异中, 提升插入以及替换这些复用内容的变异算子应用概率, 从而加速种子的进化, 提高优化效率.

3.2 目标函数的构造

本文假设可以提前获得累积型缺陷对应的特征值最优化函数表达式 E (例如, 通过静态分析报告或者缺陷报告和补丁信息获得). 目标函数 IJON_MAX(E) 或 IJON_MIN(E) 两个最值原语公式来指派. 具体地, 最值原语的选取以及表达式 E 构造主要在待检测累积型缺陷相关的累积计算中, 主要根据累积计算中的状态累积方向确定最值函数.

例如, 对于访问越界缺陷, 最值表达式通常是潜在或已经在错误报告中确定的引发访问越界的指针或整数索引. 不断变大的指针或索引用 IJON_MAX 指派优化方向, 反之用 IJON_MIN 指派优化方向. 递归过深导致的栈内存耗尽型缺陷则通过为递归函数加入递归层次变量来构造 E. 即, 构造静态整数变量, 在进入函数时自增, 退出函数时自减. 使用 IJON_MAX 指派其向层次更深的方向优化. 对于与循环或迭代次数有关的性能缺陷, 则引入多个计数变量进行衡量重复计算次数, 然后分别借助 IJON_MAX 指派其向更大值优化.

3.3 目标函数的输入驱动形式识别

为了识别出目标函数依赖的输入驱动形式, 本文监控引发目标函数最值更新的有效变化的输入和频次. 每当目标函数最值更新时就通过文件比较识别出从之前的最值种子到新的最值种子的修改规则: (O, C1, C2). 其中 O 为修改位置, C1 为被替换的内容, C2 为替换后的内容. 通常 C1 和 C2 的字节长度相同. 在实现时本文借助二进制分析工具 radare2 提供的“radiff2 -O <file1> <file2>”命令来获取修改规则. 图 3 中的命令输出形式为“O C1”=>“C2 O”. radare2 所采用的比较方法主要是经典的 Bindiff 算法[26].

设发现了 n 个有效的变异字节位置, 每个位置触发最值变更的频次为 $\boldsymbol{H}(i)$, $0<i<n$;, 识别到的特征值类型分为递增累积型、固定累积型和未知 3 种判定条件如下.

(1) 固定累积型: $\exists i, 0<i<n$, $\boldsymbol{H}(i)\geqslant\alpha$.

(2) 递增累积型: $\forall i, 0<i\leqslant n, \boldsymbol{H}(i)<\alpha$ 且$|\{i|\boldsymbol{H}(i)<3, 0<i<n\}|>\beta$.

(3) 未知: 其他.

其中, α 是触发频次阈值, 用于判定高频有效变异字节, 并根据是否存在高频有效字节来确定当前特征值是否为固定累积型特征值, β 是有效变异位置数阈值, 其用于判定有效变异位置的分散程度. 当存在高频有效变异字节时该

特征值被判定为固定累积型特征值. 否则, 当频次低于 3 的有效变异位置数分布足够多超过 β 时, 该特征值被判定为固定累积型特征值. 其他情况下该特征值的类型被标记为未知.

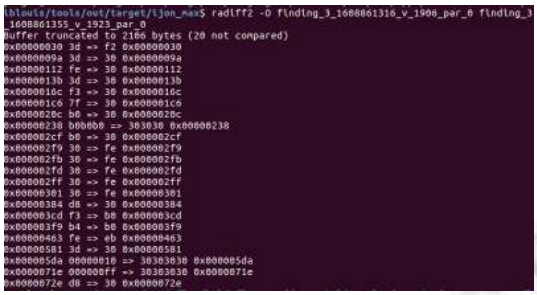

图 3 radiff2 的输出修改规则

在该模型下, 本文在训练集上根据对固定累积型和递增累积型特征值的识别准确率和识别时间对 α 和 β 的取值进行调优, 得到 α=5, β=8. 在测试缺陷集合上对该阈值组合进行验证后发现该组合能够在较短的时间内给出识别结果并保证准确率 (详见第 4 节).

3.4 面向递增累积型缺陷的差分变异策略

递增累积型缺陷的特征值依赖于有效输入数据规模的增长. 带来规模差异的数据内容往往是一些特殊取值的数据片段, 其内容的修改至多仅影响一次状态特征值最值更新, 因此有效的变异位置不具备复用性. 但是, 增量输入的数据内容为相同类型的数据, 因此可以复用. 本文利用了目标函数最值更新的相关种子变化特征, 对比带来目标函数最值更新的新种子与父亲种子的差异来获得有效的变异内容, 从而应用到其他后代的变异中.

由于识别到的修改规则集合中可能包含与目标函数无关的冗余规则, 需要进一步进行清洗. 本文采用逐个恢复并验证目标函数值是否退化的方式来识别与目标函数相关的修改. 即以引发最值更新的种子为基础, 进行规则还原, 将固定偏移 O 开始的 C2 替换为 C1. 如果修改后的种子所触发的目标函数值退化, 则说明该变异内容有效. 为了对变异内容进行精简, 本文在算法变异规则有效性判定的基础上进行了规则约简, 如算法 1 所示.

算法 1. 规则清洗算法.

输入: 规则 r=(O, C1, C2), 新种子 s, 全局最大特征值 $gmax$;

1. t=recover(s, r)
2. **if** $gmax$= =get_cur_max(t) **then**
3. **return** NULL;
4. **else**
5. r'=r
6. **while** len($r.c_2$)>1 **do**
7. r_1, r_2 =split (r', 2);
8. t_1=apply(t, r_1);
9. t_2=apply(t, r_2);
10. **if** get_cur_max(t_1)= = $gmax$ **then**
11. r'=r_1;
12. **else if** get_cur_max(t_2)= = $gmax$ **then**
13. r'=r_2;
14. **else**
15. r'=r; **break**;

16. **end if**
17. **end while**
18. **return** *r*;
19. **end if**

输出: 约简后的规则 *r*′或 NULL.

在具体实现时, 最小化问题也可以通过取负值转化为最大化问题, 因此算法 1 中统一以最大值变化来判定变异有效性. 算法 1 接受输入规则 $r=(o, c_1, c_2)$, 产生该规则的新种子 *s* 以及当前目标函数的最大值 *gmax* 作为输入. 在规则无效时输出 NULL, 否则输出精简后的规则 *r*. 该算法首先根据对规则 *r* 在 *s* 上进行还原构造临时种子 *t*(第 1 行), 然后运行构造的种子 *t* 并获得该种子触发的特征值 (第 2 行), 如果特征值不变, 说明该规则无效返回 NULL, 否则, 特征值变小了, 说明该规则有效. 将该规则二分, 并在 *t* 的基础上验证每一部分自身是否足够引发特征值边为最大值. 如果是, 则规则 *r* 可以进一步缩小, 当规则已经缩小至单字节内容的改动时停止约简, 最后返回约简后的规则 *r*′. 第 7 行函数 r_1, r_2 =split (*r*′, 2); 将规则 *r*′平均分割成两部分 r_1, r_2, r_1, r_2 各负责一半内容的修改; 第 8 行函数 t_1=apply(t, r_1) 是将种子 *t* 应用规则 r_1. 为了实现对有效变异内容的复用, 当待检测特征值被判定为递增累积型时, 将识别到的有效变异规则中的修改内容 C2 加入到 AFL 的字典中, 并在检测到目标函数依赖于输入规模递增时以 2/3 的概率使用字典相关的变异算子. 即随机替换和随机插入字典值两个变异操作, AFL Havoc 阶段的 15 和 16 号变异算子.

由于 AFL 默认 Havoc 阶段的变异策略在产生一个种子时会随机变异多次, 该次数平均值通常在 10 次以上. 对于递增累积型缺陷, 这很容易破坏已经形成的促进状态累积效应的大规模格式化数据. 因此本文在检测到递增累积型特征值后, 降低 Havoc 阶段的变异次数至 1–4 次, 从而避免破坏触发缺陷所需的已有格式化数据. 当检测到输入类型为递增累积型时禁用种子的约简操作 (trim), 以促进数据累积. 特别地, 为了加速依赖重复数据的特征值优化, 在 Havoc 阶段贪心地使用克隆变异算子 (13、14 号), 以扩大特征值的平均增幅.

4 实验评估

本文主要关注于如何识别累积型缺陷特征值的输入依赖类型, 以及构造变异方法提升累积型缺陷的检测效率. 为验证本文方法的有效性, 并保证实验时间可控, 选择错误复现应用场景对提出的方法进行评估. 本文的实验关注如下 3 个问题.

RQ1: 目标函数的输入依赖形式识别方法是否准确?

RQ2: 目标函数的依赖格式识别的性能以及对正常模糊测试的影响如何?

RQ3: 本文的变异策略对是否提升了累积型缺陷的检测效率?

本文主要关心的是加速递增累积型缺陷复现或定向测试. 为了验证本文提出的目标函数输入驱动形式识别方法的有效性. 本文构造了 8 个递增累积型缺陷和 5 个固定累积型缺陷构成的 CVE-缺陷复现集 (如表 1), 用于评估目标函数输入数据依赖形式的识别, 以及本文的优化变异策略对累积型缺陷的检测效率. 这些缺陷是从 MemLock 实验集合中的内存耗尽型缺陷和现有 CVE 漏洞库中的访问越界缺陷中随机选取的. 针对性能缺陷, 本文使用 SlowFuzz 和 PerfFuzz 的实验中词频统计程序 wf-0.41 进行目标性能阈值优化效率评估.

在实验评估中, 本文关心特征值依赖的判别方法是否高效、准确, 以及加入有针对性的变异策略后是否比原始随机变异方法效率更高. 为了保证验证结果的一般性, 本文在 IJON 这个通用的优化框架上进行评估. 在错误复现实验集合上, 以对比提出的变异策略与 IJON 用的传统随机变异策略 (AFL 的变异策略) 对累积型缺陷的复现性能差异. 通过划分训练集和测试集, 验证基于有效变异位置数和频次的特征值依赖判别方法的准确性和效率. 本文在 IJON 的基础上实现了对累积型缺陷识别和变异策略, 下文用 Jigsaw 来指代本文实现的模糊测试器. 由于固定累积型缺陷的变异优化不是本文关心的主要问题, 用于实验评估的 Jigsaw 版本并未加入固定累积型的变异优化

策略.

(1) 测试程序、参数训练集和测试集

待复现的目标缺陷有 13 个, 其中 8 个递增累积型缺陷 5 个固定累积型缺陷. 详细信息如表 1 所示. 其中 CVE-2018-17985 和 CVE-2019-6293 和 CVE-2018-4868 为内存耗尽缺陷, Wf-hash-collison 为性能故障缺陷 (通过加入人工阈值 M=5, M=10, M=20, M=40, M=80 的判定语句以反馈缺陷触发信号), 其余为访问越界缺陷. 缺陷类型根据检测缺陷所需优化的变量是否依赖输入规模增长结合源码分析事先判定. 其中灰色底纹标注的 CVE 缺陷是在设置位置阈值 (β) 和频次阈值 (α) 参数时的参考缺陷 (训练集), 而其他的缺陷则是用于验证参数设置是否具有一般性 (测试集), 并评估本文提出的输入格式识别方法的准确性. 测试集合中的累积型缺陷用于评估和比较本文针对累积型缺陷的变异方法相比 IJON 所用的传统随机变异方法的错误复现性能. 固定累积型缺陷用于评估格式识别方法计算开销. 通过对比原始 IJON 以及加入了格式识别方法后的复现时间进行评估.

表 1 用于比较 IJON 使用和不使用 Jigsaw 的测试集

缺陷类型	缺陷标识	软件名	被测程序名	插入语句和位置
递增累积型	CVE-2018-11 683	Liblouis	lou_checktable	IJON_MAX(out); compileTranslationTable.c:1157
	CVE-2018-11440	Liblouis	lou_checktable	IJON_MAX(out); compileTranslationTable.c:1140
	CVE-2018-12085	Liblouis	lou_checktable	IJON_MAX(out); compileTranslationTable.c:1130
	CVE-2018-20460	Radare2	rasm	IJON_MAX(operand); armass64.c:739
	CVE-2017-9048	Libxml2	xmllint	IJON_MAX(strlen(buf)+xmlStrlen(content->name)); (BUG 781333) valid.c:1271
	CVE-2018-17985	Binutils	cxxfilt	IJON_MAX(*rc*);(*rc*是新增的用于统计递归深度的变量) cp-demangle.c:2565
	CVE-2019-6293	Flex	Flex	IJON_MAX(*rc*);(*rc*是新增的用于统计递归深度的变量) nfa.c:351
	Wf-hash-collison	wf-0.41	wf	**else**{IJON_MAX(rc++); **if**(cnt>M) abort();}(*rc*为触发次数累积变量, *M*为人工阈值) freq.c:103
固定累积型	CVE-2018-14550	Libpng	pnm2png	IJON_MAX(i); pnm2png.c:546
	CVE-2017-7864	Freetype2	ftfuzzer	IJON_MAX(new_max); f tgloadr.c:226
	CVE-2018-20552	Tcpreplay	tcpprep	IJON_MAX(vlan_hdr->vlan_len); get.c:183
	CVE-2018-4868	Exiv2	exiv2	IJON_MAX(subBox.length) jp2image.cpp:271
	Ngiflib-issue#3	Ngiflib	SDLaffgif	IJON_MAX(2000000+(int)act_code-(int)free); ngiflib.c:534

(2) 评估指标

为了进行性能比对, 采用多次实验的方式获得其某一性能指标 (这里主要是首次复现时间) 的数据. 对于每一个 CVE 缺陷根据 20 次重复实验中首次错误复现时间进行记录. 对同一个性能指标 M, 使用 Vargha-Delaney 统计量[27]: A_{12} 来衡量被对比的两个模糊测试器在错误复现任务中生成的两组数据的优劣, 并采用 Mann-Whitney U 检验[28](显著性阈值 p<0.05) 衡量其显著性 (在实验结果中相关的 A_{12} 数值会被加粗显示). μTTE 表示平均首次触发目标缺陷的时间. 加速倍数为原始 IJON 方法平均首次触发时间与本文方法的比值.

(3) 实验环境和实验参数

Ubuntu 16.04 3 GB 内存的虚拟机, 物理 CPU AMD Ryzen 7 Pro 3700U. 实验均采用触发了目标 IJON_MAX 语句的初始种子. 在对比 Jigsaw 和 AFL 性能时插入的 IJON_MAX 语句相同, 使用的初始种子也相同. 对递增累积型缺陷, 以这种缺失了部分或全部缺失相关重复数据的输入作为初始种子. 其他固定累积型缺陷的初始种子是将 POC 上的目标字段修改为非缺陷值构造的. 为了保证评估实验在有限的时间内完成, 递增累积型缺陷的初始种子是从 POC 的基础上删去一定量相关数据构造的. 对阈值 (宏) 可调的缺陷, 调节软件中阈值设定来缩短复现时间.

表 2 显示了对 CVE 测试集合中对目标依赖的输入类型的检测时间和检测结果对比. 根据灰色部分累积型缺陷设置的经验阈值为 α=5 和 β=8.可以看出在测试集合上本文的格式识别方法对这些临界值越界类缺陷的累积型特征值全部识别正确. 说明基于位置和频次阈值格式分类法对累积型特征值判别方法具有较好的一般性. 实验结果中存在对固定累积型缺陷 (CVE-2017-7864) 无法识别的情况. 该缺陷的复现过程中并没有检测到所关注变量的最值更新. 进一步分析发现, 该缓冲区溢出由输入中特殊数据元素的出现而引起. 这种突变型的访问越界缺陷, 由于无法捕获历史最值更新相关的信息提供参考, 本文的方法暂时无法有效地识别.

表 2　对缺陷特征值依赖类型的检测结果和时间

缺陷类型	缺陷标识	特征值类型检测结果	检测时间 (s)
递增累积型	CVE-2018-11683	递增累积型	10
	CVE-2018-11440	递增累积型	8
	CVE-2018-12085	递增累积型	9
	CVE-2018-20460	递增累积型	25
	CVE-2017-9048	递增累积型	36
	CVE-2018-17985	递增累积型	53
	CVE-2019-6293	递增累积型	27
	Wf-hash-collison	递增累积型	17
固定累积型	CVE-2018-14550	固定累积型	21
	CVE-2018-4868	固定累积型	52
	CVE-2017-7864	未知	124
	CVE-2018-20522	固定累积型	64
	Ngiflib-issue#3	固定累积型	1

表 3 显示了加入自动格式检测和针对累积型缺陷的差分变异策略的 IJON 即 Jigsaw 与原始 IJON 对累积型缺陷复现时间的对比. 可以看出 Jigsaw 明显加速了累积型缺陷的复现, 加速倍数在 3–6 倍之间. 并且 Mann-Whitney U 检验都显示在所有案例的上 Jigsaw 复现时间统计优势都是显著的. Jigsaw 对 CVE-2018-17985 缺陷加速非常明显. 该缺陷是 MemLock[4]中的递归深度过深导致的栈溢出案例. 该缺陷的触发需要在初始种子 trim (约简) 后的基础上, 添加将近 2 万个连续重复字符“P”, 而任何其他字符都会破坏递归深度. AFL 的变异方法产生每个后代所应用的变异次数是随机的, 大都在数十次以上, 很容易破坏已形成的连续序列. 而 Jigsaw 在识别到特征值为递增累积型后, 降低了生成后代时的变异次数, 每次仅做微小的修改.

表 4 显示了原始 IJON 和加入了格式识别的 IJON 对固定累积型缺陷复现时间的对比. Jigsaw 的种子差分计算稍微拖慢了平均复现时间, 但是与 IJON 原始方法的差别并不显著. 这说明本文使用的规模递增数据格式识别方法的开销对规模固定累积型缺陷检测的影响很有限.

本文为 wf-0.41 的 addword 函数 (freq.c) 增加了哈希冲突次数统计的静态变量, 并插入 IJON_MAX 原语通知 IJON 将该统计变量最大化 (图 4 中加粗的语句). IJON_MAX 原语的插入位置 (freq.c:105) 对应 PerfFuzz 文献[14] 中 Figure 1 中“// traverse linked list”注释行. 当发生哈希冲突时会造成词链表的遍历, 从降低查找性能. 在实验中分别使用传统随机变异策略 (IJON) 和本文的优化变异策略 (Jigsaw) 对记录执行次数的变量进行最优化, 并记录各自发现超过 5 次、10 次、20 次、40 次、80 次 (对应 M 的值) 哈希冲突的触发输入的平均时间 (运行 10 次取平均).

表 3 Jigsaw 和 AFL 的变异策略在固定累积型缺陷测试集上首次触发时间的统计对比

缺陷标识	模糊测试器	运行次数	μTTE (s)	加速倍数	A_{12}
CVE-2018-11683	Jigsaw	20	41	—	—
	IJON	20	182	6.12	**1.00**
CVE-2017-11440	Jigsaw	20	6	—	—
	IJON	20	27	4.50	**1.00**
CVE-2018-12085	Jigsaw	20	39	—	—
	IJON	20	179	4.59	**1.00**
CVE-2017-20460	Jigsaw	20	61	—	—
	IJON	20	282	4.62	**1.00**
CVE-2017-9048	Jigsaw	20	74	—	—
	IJON	20	365	4.93	**1.00**
CVE-2018-17985	Jigsaw	20	93	—	—
	IJON	20	383	3.39	**1.00**
CVE-2019-6293	Jigsaw	20	53	—	—
	IJON	20	603	11.38	**1.00**

表 4 Jigsaw 和 AFL 的变异策略在固定累积型缺陷测试集上首次触发时间的统计对比

缺陷标识	模糊测试器	运行次数	μTTE (s)	加速倍数	A_{12}
CVE-2018-14550	Jigsaw	20	62	—	—
	IJON	20	63	0.98	0.50
CVE-2018-11460	Jigsaw	20	103	—	—
	IJON	20	100	0.97	0.48
CVE-2016-7864	Jigsaw	20	124	—	—
	IJON	20	125	1.01	0.50
CVE-2016-20522	Jigsaw	20	331	—	—
	IJON	20	328	0.99	0.50
Ngiflib-issue#3	Jigsaw	20	1	—	—
	IJON	20	1	1.00	0.50

```
89      static int cnt=0;
90      int addword (char *orig, char *name)
91      {
92          int h;
93          char * text;
94          Wordlist *w;
96          h = hash(name);
97          for (w = wordhash[h]; w != NULL; w = w->next)
98          {
99              if (strcmp(name, w->text) == 0) {
100                 w->count++;
101                 w->source = opt.per_word ? addword_source(w->source,
102     orig): NULL;
103                 return 0;
104             }
105             else{IJON_MAX(cnt++);if(cnt>M) abort();}//插入的代码
...         ...
```

图 4 wf-0.41 程序 freq.c 中插入 IJON_MAX 原语的代码上下文

将超过不同阈值平均时间绘制如图 5 所示, 其中方块点表示 IJON 的原始结果, 圆点表示本文优化策略的结果横轴时间以 s 为单位. 此外, 对不同阈值下的 Jigsaw 和 IJON 首次触发时间数据的 A_{12} 也进行了统计 M=5 和 M=10 时 A_{12} 分别为 0.56, 0.48, 随机性较大, Mann-Whitney U 检验显示 Jigsaw 和 IJON 触发时间的差异并不显著, 而 M=20、M=40 和 M=80 时 A_{12} 分别为 0.97, 0.85, 0.97, Mann-Whitney U 检验显示 Jigsaw 的加速效果较显著. 可以看出, 随着阈值设定得更大, 本文的优化方法相比 IJON 原始方法的优势逐渐显现出来.

进一步分析发现, 对 wf 词频统计程序, 检测出不同单词的哈希冲突较为困难. 但是, 一旦已经检测到某个哈希冲突, 重复增加相关词汇可以线性增加词频统计中哈希冲突次数. Jigsaw 在该案例上平均在 17 s 检测出目标函数取值是递增累积型, 说明平均情况下在此前已经发现存在哈希冲突的单词, 后续 Jigsaw 通过重复添加这些词汇使得查找冲突次数快速增加, 因此能够促进存放特征值的变量 cnt 更快地突破设定的阈值. 该实验说明了本文的方法对重复内容导致的性能缺陷的触发输入构造有明显的加速作用.

综上, 对 RQ1, 本文提出的基于位置和频次的累积型缺陷依赖格式识别方法具有一般性. 对 RQ2, 本文提出对累积计算的依赖格式识别大都能在 1 分钟内完成, 且对依赖特殊字段的固定累积型缺陷模糊测试性能影响并不明显. 对 RQ3 本文的差分变异方法确实提升了访问越界和内存耗尽缺陷中的递增累积型缺陷的复现效率, 当特征值依赖的输入依赖因素单一时, 相比随机测试有 3 倍以上的性能提升. 但是, 本文的方法对高复杂度的执行路径的检测并无帮助, 甚至可能产生性能降低等副作用.

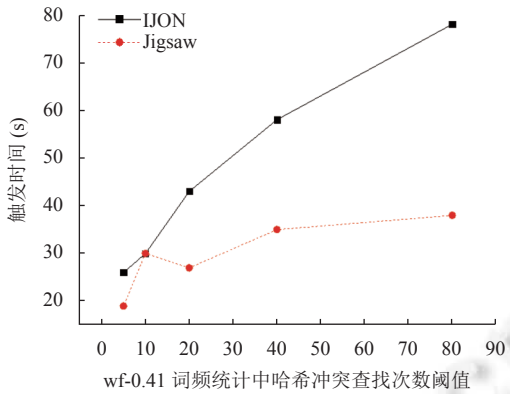

图 5 wf-0.41 词频统计程序中查找哈希冲突次数和平均触发时间

5 相关工作

缺陷特征导向的灰盒模糊测试通过刻画目标缺陷的相关特征, 改善种子度量和种子生成, 提高所关注缺陷的检测效率. 通过度量种子生成触发缺陷后代的潜力, 从而间接定位潜在缺陷, 捕获相关状态的优质种子; 通过调节种子生成 (种子调度和变异) 策略, 提高逼近潜在缺陷状态的优质后代的产生概率, 加速对潜在缺陷的验证. 种子度量影响定位潜在缺陷的全面性和准确性, 而种子生成策略决定潜在缺陷的验证效率. 二者都有助于提升目标缺陷检测效率.

现有研究主要利用缺陷相关的代码访问特征和状态演化特征度量种子优劣. 例如, 使用的内存读写功能越多, 越容易触发内存破坏相关的缺陷. TortoiseFuzz[17]事先识别 memcpy 等存在潜在危险内存操作, 按种子对这些潜在危险操作的触发数量多少度量种子优劣, 在种子调度时赋予优质种子更多繁衍机会 (更高的能量), 引导灰盒模糊测试对这些可能发生内存破坏代码进行集中测试. MemFuzz[16]则指出探索访问不同内存地址的执行有助于发现内存破坏型缺陷, 其根据内存访问地址的差异对种子进行区分, 引导搜索访问不同内存区域的执行路径从而更高效地检测内存破坏相关缺陷. Leopard[18]针对复杂代码中的缺陷进行检测, 其度量种子所触发的执行逻辑的复杂度来评价种子优劣, 在种子调度时给优质种子更多繁衍机会, 从而使模糊测试聚焦于复杂代码的测试.

针对内存耗尽型缺陷的检测的模糊测试研究 MemLock[4]通过监控函数调用深度和堆内存分配和回收的关键 API 来确定内存消耗信息, 将该类缺陷检测问题转换为内存消耗的最大化问题. MemLock 增加内存消耗较大的种子的后代生成机会 (能量), 进行内存耗尽型缺陷的检测.

针对性能缺陷检测的模糊测试研究 SlowFuzz[5]将对高复杂性的算法路径的检测问题转化为执行路径总长度的最大化问题. 以执行路径长度来表征性能特征, 结合灰盒模糊测试中的遗传变异框架进行最优化. 在种子生成方面, SlowFuzz 除了为执行路径长的种子赋予更多的能量, 也对变异策略进行了修改. 通过监控执行路径长度最值增长统计有效的变异算子和变异位置, 增加历史上多次引起最值增长的变异算子和变异位置被选择的概率. PerfFuzz[14]所有控制流执行次数进行最优化, 通过按不同控制流执行频次多维度地度量种子, 从而避免陷入总长度局部最优. HotFuzz[15]针对 Java 程序中的算法复杂度缺陷检测, 其借助虚拟执行技术实现了对单个函数的测试输入生成和性能测试 (micro-fuzzing).

上述研究并未对缺陷依赖的输入形式进行区分, 大都采用传统的随机变异策略, 导致对递增累积型缺陷检测较低. SlowFuzz 虽然使用历史上的有效变异加速缺陷检测, 但是其仅关注于变异算子和变异位置, 缺乏对有效输入内容的分析, 对递增累积型缺陷所依赖输入内容的构造效率仍然较低. 本文按照依赖的输入组织形式对累积型缺陷进行区分, 并提出了针对规模递增累积型缺陷的识别和变异优化策略, 有针对性地提升了该类缺陷的检测效率.

一些识别输入内容和格式的灰盒模糊测试研究关注于提升代码覆盖率, 主要围绕深层次执行路径搜索以及苛

刻的内容比较条件的求解开展. 由于研究问题不同, 它们无法直接应用于累积型缺陷的检测. 例如, VUzzer[8]借助污点分析定位比较字段的位置, 并根据被比较的常量确定魔数和格式标签字段的内容, 加速苛刻路径的搜索. REDQUEEN[10]和 GREYONE[11]通过监控比较分支, 通过分析输入与监控内容的关系定位特殊比较字段的位置和内容, 从而加速苛刻路径的搜索. FairFuzz[20]通过一些小的变异测试, 识别决定小概率分支重要的内容字段. 然后在后续变异中, 避免破坏它们, 从而引导苛刻路径次路径的测试. 这些方法虽然能够识别和保护某些特殊字段. 但是, 它们需要对每一次比较进行监控和分析, 对于递增累积型缺陷, 逐个分析每次内容的比较并确定它们的位置太过耗时, 因此这些方法不适用于解决递增累积型缺陷的输入构造问题.

ProFuzzer[25]对输入中不同的字段格式进行识别, 从而进行有针对性地输入构造, 根据各个字节在不同取值下造成的覆盖情况, 识别其从属的字段格式. ProFuzzer 能够识别断言 (assertion)、原始数据 (raw data)、枚举类型字段 (enumeration)、循环计数字段 (loop count), 偏移字段 (offset), 大小字段 (size). ProFuzzer 仅能解决这些位置和大小相对固定的字段格式识别问题. 其不适用于用于识别规模可变的输入格式. 本文利用了促进递增累积型特征值增长的有效变异位置和频次特征, 对累积型缺陷特征值数据依赖形式 (规模递增/规模固定) 进行识别. 并给出了高效识别和构造特征值所依赖的规模递增的输入内容的方法. 本文的工作是对非固定长度内容格式识别方面的有效补充.

6 结 论

现有的针对累积型缺陷如访问越界、内存耗尽和性能故障的定向灰盒模糊测试方法缺乏对最优化问题所依赖的输入格式的分类分析. 当缺陷依赖于有效输入数据规模时, 传统的随机变异策略的构造效率较低. 而现有的格式识别和有针对性的数据内容识别方法大都针对输入规模固定的数据格式, 缺乏对可变规模数据的识别和有效变异方法. 本文对依赖这两种形式的输入格式的累积计算进行了区分 (递增累积型特征值和固定累积型特征值), 提出了一种基于有效变异位置和频次区分递增和固定累积型缺陷特征值的方法. 针对递增累积型缺陷检测, 本文提出了一种基于种子差分内容的变异策略. 在测评实验数据集上的实验结果表明我们的方法检测能够准确区分目标函数依赖的输入形式, 并显著提升了递增累积型缺陷的复现效率, 且对缺陷类型的判别开销并不影响固定累积型缺陷的复现.

References:

[1] Serebryany K. Continuous fuzzing with LibFuzzer and AddressSanitizer. In: Proc. of the 2016 IEEE Cybersecurity Development (SecDev). Boston: IEEE, 2016. 157. [doi: 10.1109/SecDev.2016.043]

[2] Böhme M, Pham VT, Nguyen MD, Roychoudhury A. Directed greybox fuzzing. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. Texas: ACM Press, 2017. 2329−2344. [doi: 10.1145/3133956.3134020]

[3] Chen HX, Xue YX, Li YK, Chen BH, Xie XF, Wu XH, Liu Y. Hawkeye: Towards a desired directed grey-box fuzzer. In: Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. Toronto: ACM Press, 2018. 2095–2108. [doi: 10.1145/3243734.3243849]

[4] Wen C, Wang HJ, Li YK, Qin SC, Liu Y, Xu ZW, Chen HX, Xie XF, Pu GG, Liu T. MemLock: Memory usage guided fuzzing. In: Proc. of the 42nd Int'l Conf. on Software Engineering. Seoul: ACM Press, 2020. 765–777. [doi: 10.1145/3377811.3380396]

[5] 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. Texas: ACM Press, 2017. 2155–2168. [doi: 10.1145/3133956.3134073]

[6] Jeong DR, Kim K, Shivakumar B, Lee B, Shin I. Razzer: Finding kernel race bugs through fuzzing. In: Proc. of the 2019 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2019. 754–768. [doi: 10.1109/SP.2019.00017]

[7] Xu M, Kashyap S, Zhao HQ, Kim T. Krace: Data race fuzzing for kernel file systems. In: Proc. of the 2020 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2020. 1643–1660. [doi: 10.1109/SP40000.2020.00078]

[8] Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H. VUzzer: Application-aware evolutionary fuzzing. In: Proc. of the 24th Annual Network and Distributed System Security Symp. San Diego: Internet Society, 2017. 1–14.

[9] Li YK, Chen BH, Chandramohan M, Lin SW, Liu Y, Tiu A. Steelix: Program-state based binary fuzzing. In: Proc. of the 11th Joint Meeting on Foundations of Software Engineering. Singapore: ACM Press, 2017. 627–637. [doi: 10.1145/3106237.3106295]

[10] Aschermann C, Schumilo S, Blazytko T, Gawlik R, Holz T. REDQUEEN: Fuzzing with input-to-state correspondence. In: Proc. of the 26th Annual Network and Distributed System Security Symp. San Diego: Internet Society, 2019. 1–15.

[11] Gan ST, Zhang C, Chen P, Zhao BD, Qin XJ, Wu D, Chen ZN. GREYONE: Data flow sensitive fuzzing. In: Proc. of the 29th USENIX Conf. on Security Symp. Berkeley: USENIX Association, 2020. 145.

[12] Mathis B, Gopinath R, Mera M, Kampmann A, Höschele M, Zeller A. Parser-directed fuzzing. In: Proc. of the 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation. San Diego: ACM Press, 2019. 548–560. [doi: 10.1145/3314221.3314651]

[13] Haller I, Slowinska A, Neugschwandtner M, Bos H. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In: Proc. of the 22nd USENIX Conf. on Security. Washington: USENIX Association, 2013. 49−64.

[14] 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. Amsterdam: ACM Press, 2018. 254–265. [doi: 10.1145/3213846.3213874]

[15] Blair W, Mambretti A, Arshad S, Weissbacher M, Robertson W, Kirda E, Egele M. HotFuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. In: Proc. of the 27th Annual Network and Distributed System Security Symp. San Diego: Internet Society, 2020. 1–19.

[16] 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. Xi'an: IEEE, 2019. 48–58. [doi: 10.1109/ICST.2019.00015]

[17] Wang YH, Jia XK, Liu YW, Zeng K, Bao T, Wu DH, Su PR. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In: Proc. of the 27th Annual Network and Distributed System Security Symp. San Diego: Internet Society, 2020. 1–17.

[18] Du XN, Chen BH, Li YK, Guo JM, Zhou YQ, Liu Y, Jiang Y. LEOPARD: Identifying vulnerable code for vulnerability assessment through program metrics. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 60–71. [doi: 10.1109/ICSE.2019.00024]

[19] Aschermann C, Schumilo S, Abbasi A, Holz T. IJON: Exploring deep state spaces via fuzzing. In: Proc. of the 2020 IEEE Symp. on Security and Privacy (SP). San Francisc: IEEE, 2020. 1597–1612. [doi: 10.1109/SP40000.2020.00117]

[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. TBA: ACM Press, 2018. 475–485. [doi: 10.1145/3238147.3238176]

[21] Lyu CY, Ji SL, Zhang C, Li YW, Lee WH, Song Y, Beyah R. MOPT: Optimized mutation scheduling for fuzzers. In: Proc. of the 28th USENIX Security Symp. Santa Clara: USENIX Association, 2019. 1949–1966.

[22] Zou YY, Zou W, Yin JW, Huo W, Yang MF, Sun DD, Shi J. Research on mutator strategy-aware parallel fuzzing. Journal of Cyber Security, 2020, 5(5): 1–16 (in Chinese with English abstract). [doi: 10.19363/J.cnki.cn10-1380/tn.2020.09.01]

[23] Chen P, Chen H. Angora: Efficient fuzzing by principled search. In: Proc. of the 2018 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2018. 711–725. [doi: 10.1109/SP.2018.00046]

[24] Xu P, Shu H, Yu YC. Program sensitive method for generating fuzzing samples. Computer Engineering and Design, 2020, 41(12): 3368–3375 (in Chinese with English abstract). [doi: 10.16208/j.issn1000-7024.2020.12.011]

[25] You W, Wang XQ, Ma SQ, Huang JJ, Zhang XY, Wang XF, Liang B. ProFuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In: Proc. of the 2019 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2019. 769–786. [doi: 10.1109/SP.2019.00057]

[26] Myers EW. An O(*ND*) difference algorithm and its variations. Algorithmica, 1986, 1(1–4): 251–266. [doi: 10.1007/BF01840446]

[27] Vargha A, Delaney HD. A critique and improvement of the *CL* common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics, 2000, 25(2): 101–132. [doi: 10.3102/10769986025002101]

[28] Mann HB, Whitney DR. On a test of whether one of two random variables is stochastically larger than the other. Annals of Mathematical Statistics, 1947, 18(1): 50–60. [doi: 10.1214/aoms/1177730491]

附中文参考文献:

[22] 邹燕燕, 邹维, 尹嘉伟, 霍玮, 杨梅芳, 孙丹丹, 史记. 变异策略感知的并行模糊测试研究. 信息安全学报, 2020, 5(5): 1–16. [doi: 10.19363/J.cnki.cn10-1380/tn.2020.09.01]

[24] 许朴, 舒辉, 于颖超. 程序敏感的模糊测试样本生成方法. 计算机工程与设计, 2020, 41(12): 3368–3375. [doi: 10.16208/j.issn1000-7024.2020.12.011]

杨克(1989－), 男, 博士, 主要研究领域为软件安全分析, 操作系统安全.

董柯(1996－), 男, 硕士, 主要研究领域为软件安全分析, 操作系统安全.

贺也平(1962－), 男, 博士, 研究员, 博士生导师, 主要研究领域为系统安全, 隐私保护.

谢异(1995－), 男, 硕士, 主要研究领域为软件安全分析、操作系统安全.

马恒太(1970－), 男, 博士, 副研究员, 主要研究领域为软件安全分析、操作系统安全.

蔡春芳(1996－) 女, 硕士, 主要研究领域为软件安全分析, 操作系统安全.