计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2021,30(11):145−154 [doi: 10.15888/j.cnki.csa.008127]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于函数重要度的模糊测试方法①

王文硕[1,2], 程　亮[1,2], 张　阳[1,2], 李　振[3]

[1](中国科学院大学, 北京 100049)
[2](中国科学院 软件研究所 可信计算与信息保障实验室, 北京 100190)
[3](深圳市商用密码行业协会, 深圳 518118)
通讯作者: 张　阳, E-mail: zhangyang@iscas.ac.cn

摘　要: 针对现有的模糊测试方法缺乏对程序内部信息细粒度的认知, 使用孤立的因素进行种子筛选, 导致模糊测试时间消耗和增益不对等的问题, 提出了一种基于函数重要度的模糊测试方法, 首先, 本文使用属性标记的过程间控制流图 (Attributed Interprocedural Control Flow Graph, AICFG) 对函数信息和函数关系进行综合表征, 然后, 在该表征基础上对种子进行评分和评价, 根据评分和评价本文提出了更有效的种子变异策略, 同时, 本文在测试过程中根据函数命中次数对过程间控制流图的属性范围进行调整, 使用图传播算法传播属性的变化. 实验结果表明, 我们的两个优化策略对软件 flvemeta 测试中在路径数目发现方面与基线模糊测试工具 Azmerican Fuzzy Lop (AFL) 相比分别提升了 11.6% 和 13.7% 左右, 我们实现的工具 FunAFL 在对 jhead、flvmeta 和 libelfin 等软件测试中也获得了比 MOPT 和 FairFuzz 更高的覆盖率, 在实际应用中在 binutils、ffjpeg、xpdf、jhead、libtiff 和 libelfin 等软件上发现了 7 个 bug, 获得了 1 个 CVE 编号.

关键词: 模糊测试; 属性标记的过程间控制流图; 图传播; 种子筛选; 程序表征

引用格式: 王文硕,程亮,张阳,李振.基于函数重要度的模糊测试方法.计算机系统应用,2021,30(11):145–154. http://www.c-s-a.org.cn/1003-3254/8127.html

Fuzzing Method Based on Function Importance

WANG Wen-Shuo[1,2], CHENG Liang[1,2], ZHANG Yang[1,2], LI Zhen[3]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)
[2](Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[3](The Shenzhen Commercial Cipher Industry Association, Shenzhen 518118, China)

Abstract: We propose a fuzzing method based on function importance, because the existing fuzzing methods lack fine-grained knowledge of the program's internal information, use isolated factors for seed filtering, and result in the unfairness of time consumption and gain. First, the Attributed Interprocedural Control Flow Graph (AICFG) is used to comprehensively characterize function information and functional relationships. Then, the seed is scored and evaluated in light of the characterization and then a more effective seed filtering strategy is proposed. At the same time, the attribute range of the interprocedural control flow graph is adjusted according to the number of function hits, and the graph propagation algorithm is employed to propagate attribute changes. The experimental results show that the two optimization strategies have improved the number of paths by 11.6% and 13.7% respectively compared with the baseline fuzzing tool, Azmerican Fuzzy Lop (AFL), during the testing of flvmeta. The tool FunAFL implemented also achieves higher coverage during the testing of common software such as jhead, flvmate, and libtiffin than mainstream fuzzing

① 基金项目: 国家自然科学基金 (61772506, 62072448); 国家重点研发计划 (2017YFB0802902)
Foundation item: National Natural Science Foundation of China (61772506, 62072448); National Key Research and Development Program of China (2017YFB0802902)
收稿时间: 2021-01-12; 修改时间: 2021-02-07; 采用时间: 2021-02-23; csa 在线出版时间: 2021-10-22

tools, MOPT, and FairFuzz. FunAFL finds 7 bugs and gets 1 CVE number during the test of binutils, ffjpeg, xpdf, jhead, libtiff, and libelfin.

Key words: fuzzing; Attributed Interprocedural Control Flow Graph (AICFG); graph propagation; seed filtering; program representation

1 绪论

网络安全和软件安全问题不容忽视, 如 2017 年爆发的 WannaCry 勒索软件[1] 利用微软 Windows 操作系统的 MS17-010 漏洞进行自动传播, 造成了数百个国家和地区教育机构、政府机构的数据损失, 因此程序安全分析技术和漏洞挖掘技术已经成为保障信息技术可靠性、稳定性必不可少的部分.

在安全人员对抗恶意软件过程中, 提出了一系列行之有效的程序安全分析方法, 比如代码审计、模型检测、污点分析和符号执行等[2–6]. 这些方法在许多安全分析场景下都发挥了重要作用, 但是常常受限于资源消耗比较大、求解复杂等问题, 不容易进行自动化部署和扩展. 模糊测试[7] 恰好弥补了以上方法的缺点, 经典的模糊测试方法通过随机变异种子文件得到输入数据或者基于规则生成输入数据, 使用生成数据快速地对程序进行测试, 与此同时监控程序执行过程中的反馈信息, 根据反馈信息保留下对路径发现产生增益的种子文件, 收集造成程序错误的输入数据留待进一步分析, 模糊测试使用了遗传算法等启发式算法能够快速的探索程序路径, 因此模糊测试在软件规模和复杂度增加过程中能够表现出优秀的扩展性和适应性, 同时模糊测试使用轻量级的插桩获取反馈信息, 不会带来爆炸式的资源消耗.

虽然模糊测试相对于符号执行等方法具有以上的优势, 但是由于模糊测试使用随机性算法, 在很多时候进行随机性变异的过程是无意义的, 会造成时间和资源的浪费, 这个时候对于模糊测试的种子选择、能量调度、变异位置选择和变异方法等方面进行启发式的指导和改进, 可以提高模糊测试发现错误的效率和准确率. 因此, 本文基于国内外研究现状提出基于函数重要度的模糊测试改进方案, 用以提高模糊测试的效率和速度.

2 研究背景

这一节首先介绍基于变异的灰盒模糊测试算法流程, 然后讨论模糊测试的国内外研究现状, 其次讨论当前研究思路的一些限制, 最后基于以上讨论提出本文的研究思路.

2.1 基于变异的灰盒模糊测试研究现状

从对源代码的认知程度看, 模糊测试可以分为黑盒模糊测试、白盒模糊测试和灰盒模糊测试 3 种[8], 其中, 黑盒模糊测试指的是在不知道被测试程序内部结构情况下的测试, 与之相反, 白盒模糊测试指的是完全拥有程序内部结构信息的情况下构造可以到达指定代码区域的输入进行的测试, 灰盒模糊测试则是介于黑盒模糊测试和白盒模糊测试之间的测试方法, 灰盒模糊测试不能够完全获得程序的内部信息, 但是可以通过轻量级的插桩来获得程序执行时以覆盖率为代表的反馈信息, 利用信息反馈指导变异的方向, 从而提高模糊测试的效率. 使用遗传算法的灰盒模糊测试以比较低的消耗和较高的准确性在实践中发现了许多程序中的漏洞, 取得了极大成功. 因此本文研究的测试方法为灰盒模糊测试方法.

从输入数据的生成方式来划分, 模糊测试可以分为基于生成的模糊测试和基于变异的模糊测试. 基于生成的模糊测试指的是使用数据格式或者语法规范不断的生成输入数据[9], 更多时候只能局限于生成高度结构化的数据[10], 基于变异的模糊测试指的是对已有的种子输入文件进行随机变异来产生新的输入, 由于前者需要复杂的语法或者格式知识, 因此基于变异的模糊测试方法更具扩展性, 结合上述论述, 本文研究的是针对基于变异的灰盒模糊测试的改进策略. AFL (Azmerican Fuzzy Lop)[11] 是最经典的基于变异的灰盒模糊测试工具之一, 以 AFL 为代表的模糊测试的整体流程如图 1 所示, 其工作流程可以分为以下几步:

(1) 编译插桩被测试程序得到二进制文件.

(2) 在语料库中放入初始种子文件.

(3) 筛选掉无意义的种子文件, 选择下一个要使用的种子文件.

(4) 对选中的种子进行确定性变异, 同时监控执行

程序执行状态.

(5) 对选中的种子进行不确定变异, 同时监控程序执行状态.

(6) 如果程序发生 crash, 将种子放入 crash 队列, 留待进一步分析.

(7) 如果发现了此次执行产生了新的覆盖, 则将该种子放入语料库.

(8) 重复步骤 (3)–步骤 (8).

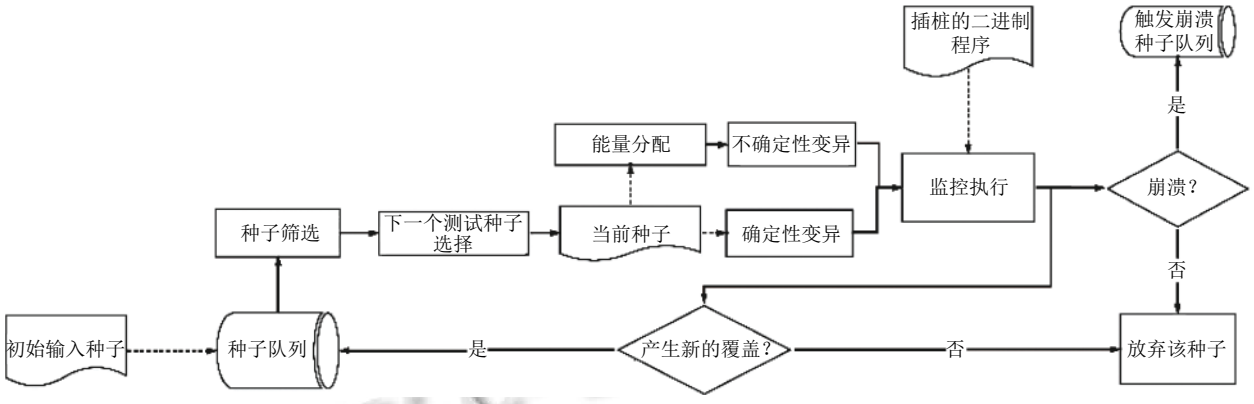

图 1 基于变异的灰盒模糊测试流程

根据模糊测试的工作流程, 目前有 4 种优化思路. 第 1 种优化思路是研究是针对模糊测试的种子筛选和选择策略进行优化. VUzzer[12] 优先考虑到达更深路径的种子, 目的是提高对于程序深层次部分的探索程度. SlowFuzz[13] 的目的是挖掘算法复杂性漏洞, 因此会考虑给在程序执行时消耗更多资源的种子更高的优先级. FuzzGuard[14] 是使用深度学习的方法将不能到达目的代码区域的输入过滤掉, 这样就可以使得留下来的种子更快速的到达指定的代码区域进行模糊测试. 第 2 种优化思路是优先选择有更高概率获得更大增益的变异方式, 比如 MOPT[15] 发现不同变异操作在一个目标程序上的效果不同, 而相同的变异操作在不同的目标程序上的效果也不同, 因此 MOPT 提出使用粒子群算法动态评估变异操作的有效性, 调整变异操作选择的概率, 使用适应性算法提高了模糊测试的效率. 第 3 种优化思路是将更多的资源消耗在能够发现更多路径的位置, 比如 Rajpal 等人研究发现输入文件中的字节变异获得的增益并不是完全相等的[16], 对文件头或者其他关键位置的变异更可能产生新的覆盖, 而对文件数据部分的变异却不太容易产生新的覆盖, 因此 Rajpal 等人使用神经网络来预测输入文件的每一个字节的重要程度, 提高输入文件重要字节所在位置的变异概率, 进而对模糊测试进行更有针对性的指导. FairFuzz[17] 通过动态计算“掩码”使得变异的位置倾向于命中稀有路径, 增强模糊测试对于稀有路径的测试进而提高模糊测试的覆盖率. 最后一种思路则是对模糊测试能量分配策略进行调整, 即调整不确定性变异的次数, 希望能够在有限的时间内发现更多的路径和漏洞, 比如 AFLFast[18] 会给被选次数更多的和被执行次数较少的种子分配更高的能量, 加大对低频路径的探索. AFLGo[19] 作为一种定向的模糊测试工具, 为了测试指定区域的代码, 会给距离目标代码位置更近的种子分配更多的能量, 以此来使得模糊测试向目标代码区域的方向进行变异. 同样作为定向模糊测试的工具 Hawkeye[20] 权衡了路径长度和函数的相似性来进行能量的分配, 目的是使得模糊测试变异更具方向性.

3 方法

3.1 研究思路

结合第 2 节中对于国内外研究背景的论述, 我们认为现有的模糊测试优化方法存在一些限制. 现有模糊测试优化方法没有充分利用程序代码及模糊测试的历史信息. 首先, 当前的种子选择策略依据的是种子对应执行路径上特定属性节点的孤立特征, 有的根据种子的执行深度, 比如 Vuzzer[12]; 有的根据是否接近未知区域, 比如 CollAFL[21]; 有的根据覆盖的罕见边个数, 比如 SAFL[22]、FairFuzz[17] 和 MuFuzzer[23] 等; 有的根据和特定指令的接近程度, 比如 NeuFuzz[24]、FuzzGuard[14]、UAFL[25]、MemLock[26]、Ankou[27] 等. 我们认为, 执行

路径上各个节点由多个属性共同决定其对于模糊测试覆盖率的价值, 并且节点之间存在相关性, 而这种相关性会影响种子对测试覆盖率的增益效果. 对于一个节点 C, 如果 C 的综合属性体现出更高的重要度, 比如比较指令数量比较多、内存操作指令数量比较多, 那么 C 就有更高概率对覆盖率产生更多的增益, 这个时候应该提高 C 被测试的次数和概率. 假设节点 C 存在两个父节点 L 和 R, 如果 L 和 R 在测试一段时间后均被频繁测试, 那么 C 有大概率也被频繁测试过, 无论 C 深度如何、是否接近未知区域或是否接近特定漏洞特征, 通过 C 的种子不需要再提高其优先级; 反之亦成立. 此外, 这种相关性导致各节点对覆盖率增益的影响会随着测试过程而不断变化, 现有方法只能在运行时更新执行路径上单个节点的属性赋值, 缺少将这种变化传播到程序不同区域的机制. 仍以上例为例, 如果 L、R 在测试一段时间后由低频测试变为高频测试, 这意味着这段时间内 C 也被频繁测试过, 那么后续可适当降低经过 C 的种子的优先级, 将计算资源调配给别的未经充分测试的区域.

综上所述, 我们分析认为目前的模糊测试工作缺乏程序更丰富信息的指导, 现有的种子选择和筛选策略的决定因素过于单一, 不能充分利用程序的动静态分析对模糊测试的方向进行指导, 没有综合考虑模糊测试对不同指导方式的敏感性, 因此本文提出将模糊测试对不同指导方式的敏感性使用一个统一的数据结构进行综合表征, 然后来指导模糊测试的种子筛选, 进而获得更高的覆盖率、发现更多的漏洞.

本节主要介绍我们对模糊测试进行改进的关键工作: 首先, 我们通过构建属性标记的过程间控制流图 (Attributed Interprocedural Control Flow Graph, AICFG) 来表征程序信息, 同时使用改进的 PageRank 算法强化对函数内跳转关系、函数间调用关系的表征, 此外, 在模糊测试过程中记录函数命中次数并基于此使用图传播算法动态地调整特征向量属性范围, 然后, 我们使用命中函数中激活基本块的特征向量对种子进行评分, 使用命中的函数调用序列对种子进行评价, 最后基于种子评分和评价改进模糊测试的种子筛选策略, 最终实现这样一个动静态结合、高效率的模糊测试系统 FunAFL. FunAFL 的整体流程如图 2 所示, 虚线代表数据流, 实线代表控制流, 无边框的模块是 AFL 原有工作流程, 虚线边框的模块是我们对模糊测试原有流程的改进, 实线边框的模块则是我们为了对模糊测试的种子筛选策略进行指导的而做的分析、处理流程. 接下来详细介绍每一部分的实现过程.

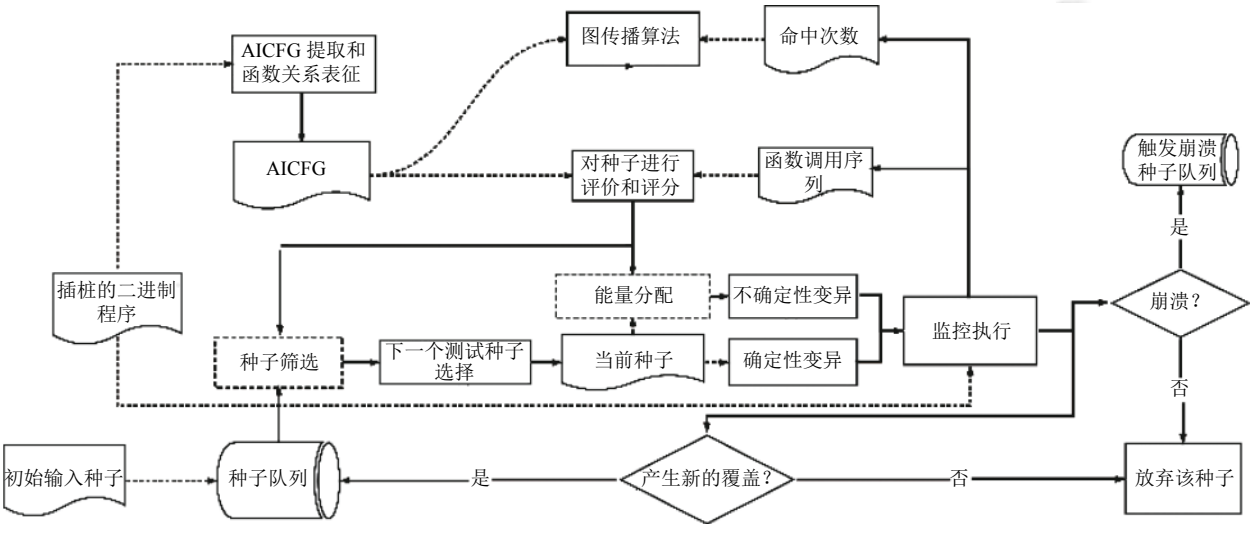

图 2 FunAFL 系统架构图

3.2 基于 AICFG 的程序表征

函数是进行程序分析、安全研究的一个重要粒度, 对一个函数内部结构进行表征的常用形式是控制流图 (control flow graph), 对一个程序中所有函数及其关系进行表征最常用的形式是调用图 (call graph), 控制流图和调用图虽然分别表征出了函数内的跳转关系和函数之间的调用关系, 但是包含函数内部的信息比较少, 不足以表征出函数在程序中的位置特征和语义信息.

Feng 等人[28] 在做二进制代码相似性检测研究时提出了一种带有属性的控制流图, 在控制流图的基础上加入了图中每个基本块的统计信息和结构信息, 即使用特征向量来表示图中的每一个节点, 赋予了控制流图更多的语义信息. 模糊测试针对的对象是一系列函数组成的程序, 每一个函数会因为其所处位置以及包含信息的不同而在模糊测试中表现出不同的重要度, 因此基于以上研究, 我们提出使用属性标记的控制流图对程序信息进行表征, AICFG 的定义如定义 1 所示, 即每一个基本块节点使用其统计特征和结构特征组成的特征向量来表示, 整个软件表示成为一个有向图的形式, 边的方向表明了函数内基本块的跳转关系和函数之间的调用关系, 特征向量由 5 个统计特征和 2 个结构特征组成, 选定的 AICFG 属性及解释如表 1 所示.

表 1 AICFG 属性及解释

特征	解释
子分支数量	jz, jnz, cmp, test等指令分支数量
内存及读写操指令数量	malloc, free等内存操作相关指令和read, write等读写相关指令
指令数量	总的指令数量
字符串操作指令	每个基本块中含有字符串操作的指令数量
立即数数量	每个基本块中含有立即数操作的指令数量
Offspring	节点的子孙节点数量
Betweeness	经过该节点的最短路径数目与经过该节点路径总数目的比值

定义 1. AICFG. 属性标记的过程间控制流图是一个有向图$G =< V,E,\Phi >$, 其中图的节点集V是基本块集合, 基本块是一段连续没有跳转的指令, 图中的边E则代表了基本块之间的跳转关系, 包括函数内的跳转关系和函数间的调用关系, 同时$\Phi : V \to \Sigma$是从基本块节点到基本块属性的一个映射.

定义 2. Betweeness. 在一个图$G' =< V',E' >$中, 其中V'是节点集合, E'是边集合, 对于节点$v \in V'$来说, 其介数为:

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

其中, $\sigma(s,t|v)$表示从s到t经过v的路径数目, 并且$v \neq s, v \neq t$, $\sigma(s,t)$表示从s到t的所有路径. 当$s = t$时, 有$\sigma(s,t) = 1$, 当$v \in s,t$时, 有$\sigma(s,t|v) = 0$.

选择这些统计属性和结构属性的原因在于: 分支数量的多少与路径数目存在着正相关关系, 内存操作和读写操作相关指令数量与内存错误发生的概率有着很强的关联性, 总的指令数量表明了函数的大小及复杂程度, 立即数操作指令和字符串操作指令数目也与特定漏洞发生的概率呈现出正相关关系. 对于选择的两个结构属性, Betweenness[29] 的定义如定义 2 所示, 节点 v 的 Offspring 属性指的是 v 的子孙节点数量, 两个结构属性表征了节点在图中位置的重要性. 综合来看, 这些属性表明了函数在模糊测试时发现新路径及漏洞方面值得探索的程度.

3.3 基于改进的 PageRank 算法的函数调用关系强化表征

上一节提出的 AICFG 只能表明函数内及函数间静态的属性信息, 结构属性虽然能表征出一定的位置特征, 但是相对于整个图来说, 不应该孤立地看待每个节点对模糊测试的增益而忽视了节点之间的联系, 一个节点的重要性会影响到所处位置附近节点的重要性, 尤其是函数内基本块的跳转关系和函数间的调应该更明显的在节点属性中体现出来.

PageRank[30] 是 Page 等人提出的一种用于提高搜索质量和速度的网页连接分析算法. 该算法将网页看作是互相连接的节点并基于数量假设和质量假设对网页的重要性进行排序. 所谓的数量假设, 指的是在 Web 图模型中, 如果一个页面节点接收到的其他网页指向的入链数量越多, 那么这个页面越重要. 而质量假设指的是, 指向页面 page 的入链质量不同, 质量高的页面会通过连接向其他页面传递更多的权重. 所以越是质量高的页面指向页面 page, 则页面 page 越重要. 如果存在这样一个有向图, 节点$B_1, B_2, \cdots, B_n$指向节点 A, 那么此时节点 A 的 PageRank 值$PR(A)$如下:

$$PR(A) = \sum_{i=1}^{|B|} \frac{PR(B_i)}{L(B_i)} \times q + (1-q)$$

其中, $L(B_i)$是节点B_i的出度, q为阻尼系数.

PageRank 算法是用来表明在网页点击过程中的每个网页的重要程度, 我们改进该方法进一步强化 AICFG 对节点关系的表征, 根据 PageRank 算法的质量假设和数量假设对 AICFG 数据结构提出了符合当前应用场景的质量假设和数量假设. 当前应用场景的数量假设是, 在模糊测试中, 在函数间如果一个函数被其他函数指向的次数比较少, 在函数中如果一个基本块被其他基本块指向次数比较少的话, 在静态分析结果上的表现为指向该节点的节点比较少, 那么这个函数或基本

块就比较重要, 因为这个函数在结构上表现为不容易到达. 质量假设则是, 在函数间指向一个函数的函数质量不同, 在函数内指向一个基本块的基本块质量不同, 质量高的函数或者基本块节点会通过链接向其他节点传递更多的权重. 所以越是质量高的节点指向一个节点, 则被指向节点越重要. 这个实际意义在于, 到达函数或者基本块的父节点的难易程度会传递给该函数. 此时我们的方法就是权衡了函数质量和数量两个因素下对特征向量的一个综合计算. 基于以上数量假设和质量假设, 仍以 PageRank 值$PR(A)$所描述的图为例, 这时给出如下的 PageRank 值$PR'(A)$计算公式:

$$PR'(A)=\sum_{i=1}^{|B|}\frac{PR(B_i)}{L(B_i)\times IN(A)\times IN(A)}\times p+(1-p)\times PR(A)$$

其中, $L(B_i)$指的是节点B_i的出度, $IN(A)$指的是A的入度, p为阻尼系数. 该计算公式就是我们工具中使用到的 PageRank 算法公式.

3.4 AICFG 属性动态调整

模糊测试是一个需要长时间不断使用输入数据进行测试的过程, 我们在模糊测试进行前的静态分析即 AICFG 和 PageRank 处理可以看作是时间节点为 0 时的对程序的静态分析, 我们同时要考虑模糊测试过程中其他时间节点的函数属性变化情况, 随着模糊测试的进行, 如果一个函数被命中的次数足够多, 那么我们就有理由认为该函数已经被充分测试足够多次了, 就不应该继续浪费时间在这一类函数上, 这个时候就需要根据函数的命中次数来调整函数的节点属性, 同时, 一个节点的变化也会对周围节点产生影响, 因此, 我们根据函数命中次数对函数属性进行调整, 在调整完节点属性之后, 使用图算法进行一次节点属性的传播. 总的来看, 我们这里要完成两个任务, 一是使用函数命中次数进行节点特征向量范围的缩放, 二是进行节点属性的传播.

对于第一个任务来说, 模糊测试执行一定时间后, 设函数f及其基本块b命中的次数为$p(f,b)$, 对每一个特征向量做如下调整: $\delta*[s_1,s_2,\cdots,s_n]$, 其中:

$$\delta=adjust\_time(p(f,b))=\left(lower^{\frac{1.0}{MaxCount}}\right)^{p(f,b)}$$

其中, $lower$ 是归一化后最小的值, $MaxCount$ 是最大的命中次数, 函数$adjust_time$满足: 频数$p(f,b)$越大, δ越小; 反之, $p(f,b)$越小, δ越大. 通过这种方式我们处理之后的数据具有如下特征: 对于命中次数, 值越大经过归一化后的值越小, 反之成立, 归一化后的值能够保证在合理数据范围内, 且有明显区分形势.

第一个任务完成以后, 为了体现变化节点对周围节点的影响, 我们使用图传播算法完成这一点. 图节点信息传播算法很多, 我们在这里受到了 Weisfeiler-Lehman 算法[31] 启发, 该算法多用于图相似或者同构等场景中, 其 relabel 的过程指导了后续许多传统以及深度学习的图算法, 我们这里也基于 Weisfeiler-Lehman 算法的 relabel 过程对使用函数命中次数调节之后的函数属性进行一次属性传播.

如图 3 所示是 Weisfeiler-Lehman 算法的一次标签压缩过程, 每一次标签压缩会将一个节点的信息传播到距离为 1 的节点处, 迭代的次数越多, 则传播的距离越远. 我们的应用场景中, 每一个节点是由一个特征向量构成, 使用函数命中次数对函数节点属性进行范围调节之后, 我们基于图算法进行属性传播, 同时根据图的大小我们动态的调整节点传播次数, 每进行一次迭代, 节点属性就会传播到距离为 1 的节点处, 迭代次数越多, 传播的距离就越远.

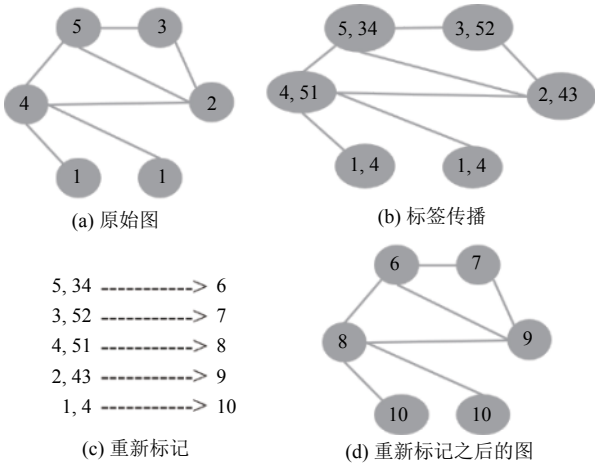

图 3 Weisfeiler-Lehman 算法进行一次标签传播的过程

我们的传播过程如下:

$$v_i=p\times v_i+(1-p)\times\frac{1}{n}\sum_{j\in M(i)}v_j$$

其中, $M(i)$是节点i的邻接节点, v_i是节点i的特征向量.

完成以上两个任务就可以做到, 基于每一个函数测试的充分程度动态的调整每一个函数及周围节点的特征向量值, 同时表征出节点及邻接节点的相互影响关系, 进而使得 AICFG 更准确地反映出程序不同时间节点的程序信息.

3.5 种子评分和评价

对于一个输入种子$seed_i$在执行之后都会产生一个函数命中序列, 这个函数命中序列上对应着多个函数, 我们要通过命中函数的重要程度来评估种子的重要程度. 如图 4 所示, 每个函数内部也有着复杂的结构, 因此我们将命中的基本块信息作为函数重要度的表征, 我们将命中的基本块称为“激活”状态, 即命中函数的重要程度由激活基本块表征. 我们设$seed_i$命中的函数序列为$[f_1, f_2, \cdots, f_n]$, 函数f_i对应的激活的基本块为$[b_1, b_2, \cdots, b_m]$, 每一个命中基本块b_j特征向量为$v_{(f_i,b_i)} = [s_{i1}, s_{i2}, \cdots, s_{iq}]$, 我们根据种子覆盖到的函数特征向量序列对种子进行一个综合评分, 最终评分如下:

$$score(seed_k) = average(score\_fun(f_1) + score\_fun(f_2) + \cdots + score\_fun(f_n)) = \frac{1}{sum(bbs)} \sum_{i=1}^{n} \sum_{j=1}^{m} \left(\left\| v_{(f_i,b_j)} \right\|_2 \right)$$

其中, $sum(bbs)$表示激活的基本块数目.

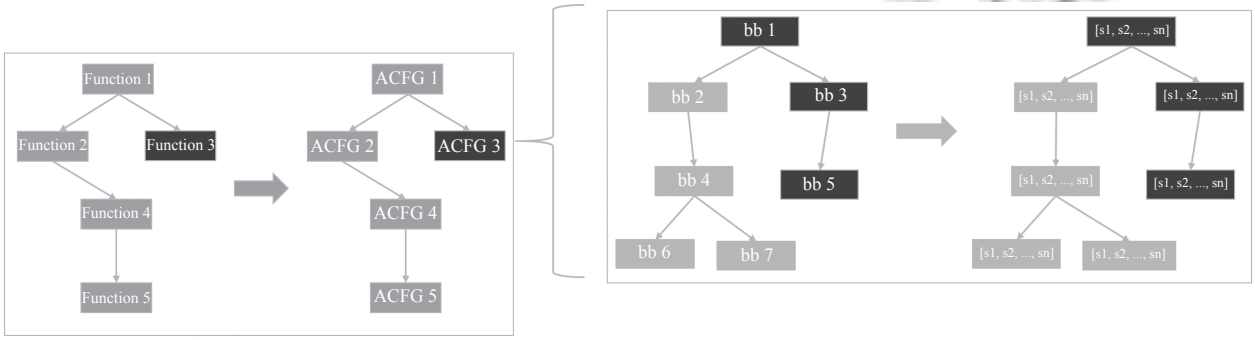

图 4 函数重要度的表征

我们在获取种子评分的同时, 也获得了一系列的函数调用序列, 如果 AFL 原有的 bitmap 的覆盖度量方式看作是基本块粒度的一个覆盖统计, 那么函数调用序列可以看作是函数粒度的一种覆盖度量方式, 当过多基本块命中出现在相同函数序列命中时, 我们适当的调整种子筛选策略, 算法 1 即函数调用序列进行 hash 的过程, 该 hash 算法不仅能够区分不同元素组成的序列, 如序列 (1, 2, 3, 4) 和序列 (1, 2, 4), 还可以区分相同元素不同顺序组成的序列, 如 (1, 2, 3, 4) 和 (1, 3, 2, 4), 因为往往漏洞的发生和函数调用序列是密切相关的, 所以区分相同函数不同顺序的调用序列是有意义的. 同时, 我们也认为函数调用序列的稀有度也是函数重要度的一种体现.

从函数评分的角度看, 函数重要度是由种子命中的函数中“激活”基本块的特征向量决定, 得分越高, 发现更多路径和漏洞的概率越大, 则该函数越重要; 从函数评价的角度看, 函数重要度是由种子命中的函数调用序列的稀有度决定, 函数调用序列通过哈希算法标记, 序列命中次数越少, 则序列对应的函数越值得探索, 其重要度越高, 函数组成的序列重要度也越高. 函数重要度最终成为种子重要度的组成单位, 指导模糊测试的种子筛选过程.

算法 1. 函数调用序列 hash 计算方法

输入: 函数调用序列对应的标识序列 *array*, 数组长度 *n*
输出: 函数调用序列的 hash

```
1. hashval=1
2. seed=31
3. for i=1 to n
4.     hashval = (hashval×seed+array[i])&65535
5. endfor
6. return hashval&65535
```

3.6 种子筛选策略优化

我们获得了种子评分和评价之后, 也就获得了每一个种子在不同时刻对于漏洞发现和新路径发现的潜力和能力, AFL 原来的种子筛选策略为:

(1) 保证当前覆盖率不会减少.

(2) 在保证条件 (1) 的情况下选择“执行时间×长度”更小的种子.

我们现在不再单一的考虑每一个种子基础的特征, 而是使用体现种子综合潜力的评分和评价作为筛选条件, 将 AFL 种子筛选策略改为:

(1) 保证当前覆盖率不会减少.

(2) 如果过多种子落在同一个函数命中序列上, 在保证条件 (1) 的情况下保留函数调用序列稀有度更高的种子.

(3) 保证条件 (1), 条件 (2) 的情况下保留得分更高的种子.

(4) 不满足条件 (1), 条件 (2) 的情况下选择“执行时间×长度”更小的种子.

这样的话一方面能够做到保证最大的覆盖率, 另一方面同时考虑了每一个种子的评分和评价情况, 使用更可靠的指标对种子进行了综合筛选.

4 评估

4.1 验证不同优化策略的有效性

我们在这里选取了程序 flvmeta 作为实验对象, 来研究不同的优化策略是否真的对模糊测试的结果产生了增益. 我们的策略包括了种子评分对应的种子筛选策略 (SEED) 和种子评价对应的种子筛选策略 (TRACE_SEED).

我们对 flvmeta 在相同条件下进行了 40 小时的实验, 如图 5 所示, 该图进行 40 小时实验后我们的两个改进策略发现的路径数目, 可以看到, 我们的两个优化策略的实验结果都优于原始的 AFL 的实验结果, 我们的两个策略比原始的 AFL 路径发现数目分别提高了 11.6% 和 13.7%, 也就是说, 我们的优化策略都对模糊测试结果产生了增益.

4.2 验证整体策略的有效性

最终的系统 FunAFL 使用了第 4.1 节中的两个优化策略, 在 FLV 文件处理软件 flvmeta、图像处理软件库 jhead 和 EFL 二进制文件解析库 libelfin 等程序上进行了实验, 使用覆盖率作为统计指标, 同时我们选择了近期实验效果比较好的两款模糊测试工具 FairFuzz[17] 和 MOPT[15] 作为对比. 如图 6 所示, 是我们在这些程序上测试过程中“路径发现-时间”图, 其中实线是我们的 FunAFL 的实验效果, 点画线和虚线分别是 FairFuzz 和 MOPT 的实验效果, 分析图中数据我们可以得到如下的结论: 我们的工具 FunAFL 在 100 小时之后发现的路径优于 FairFuzz 和 MOPT. 其中, 在部分实验中, 如对 jhead 的测试中, 发现的数目要远远优于 MOPT 和 FairFuzz. 这说明我们的方法对于模糊测试产生了增益, 我们的工具 FunAFL 的 4 个优化策略在实际使用中是有效的.

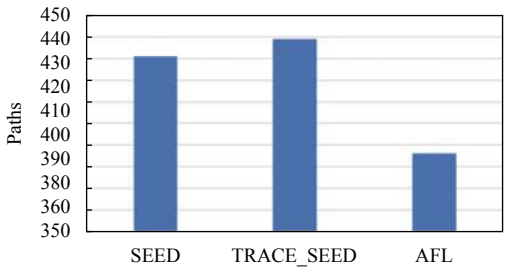

图 5 不同策略下测试程序 flvmeta 路径发现对比图

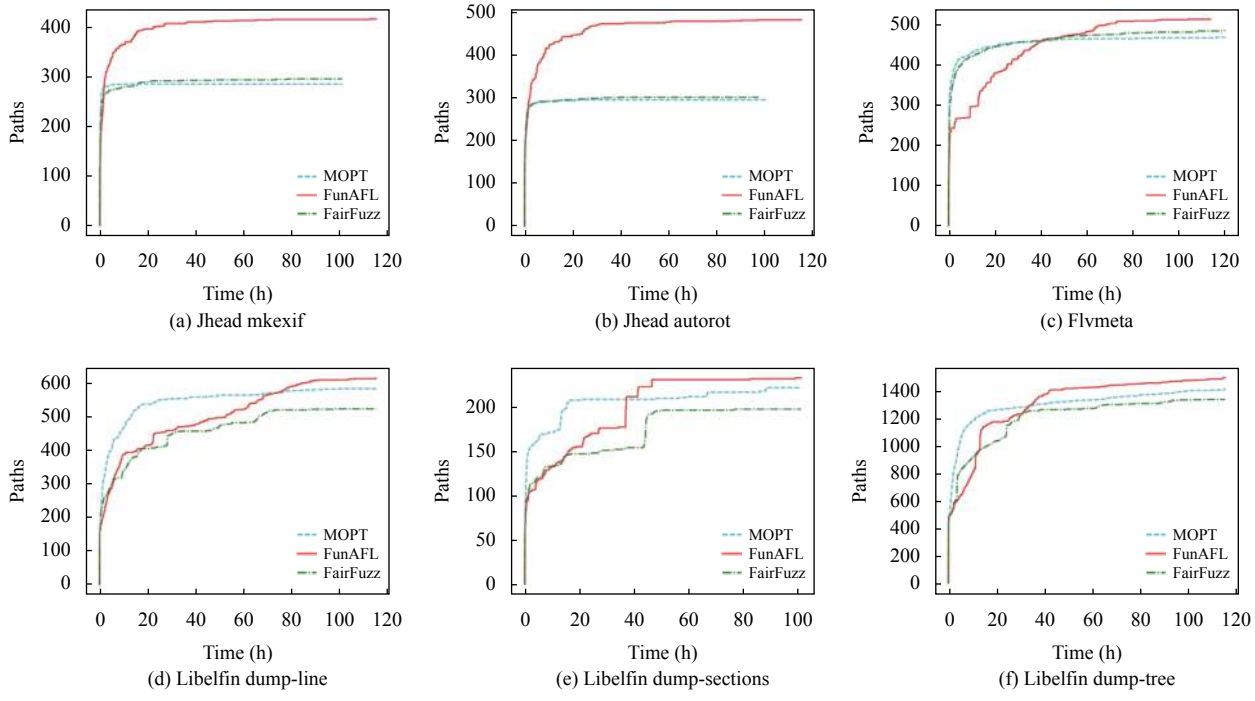

图 6 路径发现实验对比

4.3 发现了真实世界程序中的漏洞

在整个系统实现过程中, 我们先后对 jpeg 编码解码库 ffjpeg、二进制程序修改处理库 binutils、pdf 文件处理库 xpdf、tiff 图像处理库 libtiff、ELF 二进制程序解析库 libelfin 和图像处理库 jhead 等程序进行了测试, 在实际测试中我们发现了大量的 crash, 我们对其进行逐一的分析, 整理之后的漏洞信息如表 2 所示, 我们将漏洞信息通知给了软件作者, 部分作者对此做出回应并修复了软件存在的漏洞, 同时截至目前我们获得了一个 CVE 编号: CVE-2020-13438[32]. 该研究成果表明我们的系统 FunAFL 具备在实践中发现漏洞的能力.

表 2 真实世界程序中发现的漏洞

软件	版本	漏洞类型	状态
ffjpeg	2020.02.24	Invalid read	CVE-2020-13438
binutils (nm)	2.34	Bad free	已存在该Bug报告
xpdf (pdfimages)	xpdf-4.02	SEGV	已存在该Bug报告
libtiff (tiff2pdf)	4.1.0	Invalid write	作者修复
libelfin	2020.07.29	Invalid write	等待CVE审核回复
libelfin	2020.07.29	Invalid read	等待CVE审核回复
jhead	3.04	heap_buffer_overflow	等待CVE审核回复

5 结束语

本文基于 AICFG 的表征形式探究了基于函数重要度的模糊测试方法的有效性, 在未来的工作中, 我们将继续研究使用函数重要度的方法对模糊测试的能量调度等其他方面进行优化, 此外, 在进行动态调整和静态分析的过程中, 我们考虑对数据进行一些预处理以使得特征向量更具有代表意义, 最终, 我们期待着该模糊测试工具能够在发现更多的程序安全问题, 帮助软件开发者修复漏洞.

参考文献

1 Akbanov M, Vassilakis VG, Moscholios ID, *et al*. Static and dynamic analysis of WannaCry ransomware. IEICE Information and Communication Technology Forum (ICTF). Graz: ICTF, 2018.

2 朱雪阳, 张文辉, 李广元, 等. 模型检测研究进展. 中国科学技术协会, 中国计算机学会. 2011–2012 计算机科学与技术学科发展报告. 2012.

3 Ceara D, Potet ML, Ensimag GI, *et al*. Detecting software vulnerabilities-static taint analysis. Bucharest: Verimag Distributed and Complex System Group, Polytechnic University of Bucharest, 2009.

4 Newsome J, Song DX. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05). San Diego: NDSS, 2005. 3–4.

5 Schwartz EJ, 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). 2010 IEEE Symposium on Security and Privacy. Oakland: IEEE, 2010. 317–331.

6 Cadar C, Sen K. Symbolic execution for software testing. Communications of the ACM, 2013, 56(2): 82–90. [doi: 10.1145/2408776.2408795]

7 Miller BP, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. Communications of the ACM, 1990, 33(12): 32–44. [doi: 10.1145/96267.96279]

8 Li J, Zhao BD, Zhang C. Fuzzing: A survey. Cybersecurity, 2018, 1(1): 6. [doi: 10.1186/s42400-018-0002-y]

9 Godefroid P, Peleg H, Singh R. Learn & fuzz: Machine learning for input fuzzing. 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). Urbana: IEEE, 2017. 50–59.

10 Wang JJ, Chen BH, Wei L, *et al*. Skyfire: Data-driven seed generation for fuzzing. 2017 IEEE Symposium on Security and Privacy (SP). San Jose: IEEE, 2017. 579–594.

11 Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt. [2021-01-10].

12 Rawat S, Jain V, Kumar A, *et al*. VUzzer: Application-aware evolutionary fuzzing. Proceedings of the NDSS Symposium 2017. San Diego: NDSS, 2017. 1–14.

13 Petsios T, Zhao J, Keromytis AD, *et al*. SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas: ACM, 2017. 2155–2168.

14 Zong PY, Lyu T, Wang DW, *et al*. FuzzGuard: Filtering out

unreachable inputs in directed grey-box fuzzing through deep learning. Proceedings of the 29th USENIX Security Symposium. Boston: USENIX, 2020. 2255–2269.

15 Lyu CY, Ji SL, Zhang C, *et al*. MOPT: Optimized mutation scheduling for fuzzers. Proceedings of the 28th USENIX Security Symposium. Santa Clara: USENIX, 2019. 1949–1966.

16 Rajpal M, Blum W, Singh R. Not all bytes are equal: Neural byte sieve for fuzzing. https://arxiv.org/abs/1711.04596. (2017-11-10) [2021-01-10].

17 Lemieux C, Sen K. FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. Montpellier: ACM, 2018. 475–485.

18 Böhme M, Pham VT, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna: ACM, 2016. 1032–1043.

19 Böhme M, Pham VT, Nguyen MD, *et al*. Directed greybox fuzzing. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas: ACM, 2017. 2329–2344.

20 Chen HX, Xue YX, Li YK, *et al*. Hawkeye: Towards a desired directed grey-box fuzzer. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto: ACM, 2018. 2095–2108.

21 Gan ST, Zhang C, Qin XJ, *et al*. Collafl: Path sensitive fuzzing. 2018 IEEE Symposium on Security and Privacy (SP). San Francisco: IEEE, 2018. 679–696.

22 Wang MZ, Liang J, Chen YL, *et al*. SAFL: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing. Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. Gothenburg: ACM, 2018. 61–64.

23 傅玉, 石东辉, 张阳, 等. 基于覆盖频率的模糊测试改进方法. 计算机系统应用, 2019, 28(1): 17–24. [doi: 10.15888/j.cnki.csa.006714]

24 Wang YC, Wu ZH, Wei Q, *et al*. NeuFuzz: Efficient fuzzing with deep neural network. IEEE Access, 2019, 7: 36340–36352. [doi: 10.1109/ACCESS.2019.2903291]

25 Wang HJ, Xie XF, Li Y, *et al*. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. Seoul: ACM, 2020. 999–1010.

26 Wen C, Wang HJ, Li YK, *et al*. Memlock: Memory usage guided fuzzing. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. Seoul: ACM, 2020. 765–777.

27 Manès VJM, Kim S, Cha SK. Ankou: Guiding grey-box fuzzing towards combinatorial difference. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. Seoul: ACM, 2020. 1024–1036.

28 Feng Q, Zhou RD, Xu CC, *et al*. Scalable graph-based bug search for firmware images. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna: ACM, 2016. 480–491.

29 Brandes U. A faster algorithm for betweenness centrality. The Journal of Mathematical Sociology, 2001, 25(2): 163–177. [doi: 10.1080/0022250X.2001.9990249]

30 Page L, Brin S, Motwani R, *et al*. The PageRank citation ranking: Bringing order to the Web. Rondo: Stanford Digital Library Technologies Project, 1998.

31 Shervashidze N, Schweitzer P, Van Leeuwen EJ, *et al*. Weisfeiler-lehman graph kernels. The Journal of Machine Learning Research, 2011, 12: 2539–2561.

32 National Vulnerability Database. CVE-2020-13438 detail current description. https://nvd.nist.gov/vuln/detail/CVE-2020-13438. (2020-05-27) [2021-01-10].