计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(3): 790−807, 2025

CBFuzzer: 基于执行上下文导向及保护突破的程序缺陷模糊检测

唐成华[1] 蔡维嘉[1] 杨萌萌[2] 强保华[3]

[1](广西可信软件重点实验室(桂林电子科技大学) 广西桂林 541004)

[2](联邦科学与工业研究组织 Data61 澳大利亚墨尔本 3168)

[3](广西云计算与大数据协同创新中心(桂林电子科技大学) 广西桂林 541004)

(tch@guet.edu.cn)

CBFuzzer: Fuzzy Detection of Program Defects Based on Execution Context Orientation and Protection Breakthrough

Tang Chenghua[1], Cai Weijia[1], Yang Mengmeng[2], and Qiang Baohua[3]

[1] (*Guangxi Key Laboratory of Trusted Software* (*Guilin University of Electronic Technology*), *Guilin*, *Guangxi* 541004)

[2] (*Data61*, *Commonwealth Scientific and Industrial Research Organization*, *Melbourne*, *Australia* 3168)

[3] (*Guangxi Cloud Computing and Big Data Collaborative Innovation Center* (*Guilin University of Electronic Technology*), *Guilin*, *Guangxi* 541004)

Abstract A large number of application practices have proven the effectiveness of fuzzy testing to detect program vulnerabilities. The existing fuzzy testing methods lack the analysis of differences in performance specific to the testing tasks and adjust testing policies appropriately. Instead, they mostly adopt a unified process, resulting in unsatisfactory testing results. It is necessary to modify the policy based on specific information during the testing process to achieve better testing performance, and a new program defect fuzzy testing method based on execution context orientation is proposed, which can break through the protection mechanism. By capturing and analyzing specific contextual information during the actual processing of input test cases by the tested program, and achieving rapid exploration of program structural features, the sample mutation policy can be optimized. Meanwhile, a prototype tool CBFuzzer for program defect fuzzy detection based on execution context orientation is implemented. The experimental results indicate that CBFuzzer can effectively explore the internal structure of programs (including breakthroughs in protection mechanisms), simulate unconventional program state transitions, and more efficiently expose vulnerability points. By comparison, CBFuzzer shows improvements ranging from 6.8% to 36.76% in terms of vulnerability exposure, with the highest increase in the number of actual vulnerabilities detected reaching up to 66.67%. With the investment of a small amount of additional testing resources within an acceptable range, CBFuzzer not only achieves improved detection performance for regular types of vulnerabilities but also exhibits higher detection capabilities for vulnerabilities with strong concealment. As of August 10, 2023, a total of 126 new vulnerabilities have been identified through the utilization of CBFuzzer in 13 testing tasks (reported to related software developers and submitted to CVE® organization).

Key words program defect; execution context; protection mechanism; fuzzy testing; vulnerability detection

收稿日期: 2023-09-20; **修回日期**: 2024-04-29

基金项目: 国家自然科学基金项目(62062028, 62262006); 广西自然科学基金项目(2018GXNSFAA294058); 广西可信软件重点实验室项目(KX202320); 广西研究生教育创新计划项目(YCSW2023295)

This work was supported by the National Natural Science Foundation of China (62062028, 62262006), the Guangxi Natural Science Foundation of China (2018GXNSFAA294058), the Project of Guangxi Key Laboratory of Trusted Software (KX202320), and the Innovation Project of Guangxi Graduate Education (YCSW2023295).

通信作者: 杨萌萌(mengmeng.yang@data61.csiro.au)

摘　要　大量的应用实践证明了模糊测试用于检测程序脆弱性的有效性. 现有的模糊测试方法缺少针对具体测试任务表现的差异性进行分析并适当地调整测试策略，更多的是采取统一流程导致测试结果差强人意. 有必要根据测试过程中的具体信息对策略进行修正以达到更好的测试表现，提出了一种新的基于执行上下文导向的程序缺陷模糊测试方法，并能突破保护机制，通过捕获并分析受检程序对输入测试用例实际处理过程中的具体上下文信息，快速探索程序结构特征，对样本突变策略进行优化. 同时，在相关算法的基础上实现了基于执行上下文导向的程序缺陷模糊检测原型工具CBFuzzer. 实验结果表明，CBFuzzer能有效地实现对程序内部结构的快速探索（包括对保护机制的突破）、非常规程序状态转换的模拟以及更高效的脆弱点暴露. 与对照方法相比，CBFuzzer在脆弱点暴露能力方面有6.8%~36.76%不同程度的提升，实际脆弱点的检出数量提升率最高达到66.67%. 在可接受范围内的少量额外测试资源的投入下，CBFuzzer不仅在常规漏洞类型的检出效果上得到提高，并且对于隐匿性强的漏洞类型有更高的检测能力. 截至2023年8月10日，通过CBFuzzer在13个测试任务中共发现了126个新的漏洞（已报告给软件开发者，并提交给CVE®组织）.

关键词　程序缺陷；执行上下文；保护机制；模糊测试；漏洞检测

中图法分类号　TP309

DOI: 10.7544/issn1000-1239.202330755　　**CSTR:** 32373.14.issn1000-1239.202330755

软件的安全性是关键的非功能需求. 在软件开发中有必要采取全面和有效的测试方法，识别程序中的污点、漏洞等缺陷，避免带病部署[1]. 传统测试方法往往只能针对特定的输入和输出，难以覆盖所有可能的情况，而模糊测试提供了解决该问题的有效途径[2]，通过生成随机、不合法或异常的输入，监测受检程序面向输入的处理过程是否出现程序崩溃、程序异常、挂起等安全违例事件，以此检验目标程序的容错能力和安全性. 模糊测试已广泛应用于软件安全测试[3]、网络安全测试[4]、Web应用程序测试[5]、移动应用程序测试[6]、智能设备安全测试[7]等诸多场景. 当前模糊测试方法可以从2个维度进行分类.

其一是根据对受检程序的信息掌握程度递增，分为黑盒类、白盒类和灰盒类的模糊器. 黑盒类只需了解输入与输出关系，无需掌握受检程序的内部结构，通常使用随机生成器生成不同的输入，根据程序反应确定是否存在漏洞，可用性和可重现性较高[8-9]. 白盒类对受检程序的内部结构、源代码和调试状态有深入了解，有较好的代码覆盖率，通常使用符号执行和路径分析等生成测试用例[10-11]. 灰盒类则是对输入输出关系与内部结构掌握的均衡方案，通常使用程序静态和动态分析生成有效的测试用例[12-14].

其二是按照提供给受检程序的测试样本生成方法的不同，分为2种模糊器：一种是基于样本突变的模糊器[15-16]，主要通过对预置的测试用例种子进行突变（如算数运算、字节反转等）来创建新的测试用例，其关键在于使得所选择的用于突变的用例尽可能不同，以提高代码覆盖率；另一种是基于生成的模糊器[17]，其在测试开始阶段并不提供相应的测试用例，而是根据预先掌握的被测试系统特定的协议、语法或者输入用例结构建模，据此生成输入并创建测试用例.

为了提高软件缺陷的检测准确率和分析效率，当前大多数模糊测试方法要么考虑提升测试的代码覆盖率[18-19]，从而发现程序中更多的缺陷；要么动态修改和监控程序行为[20-21]，用来了解程序内部的实现过程；要么探索程序执行路径的可达性[22]，并提取关键部分的数据进行计算模拟来发现程序缺陷. 出于对恶意者的戒备或知识产权的保护，软件产品的内部处理逻辑往往实施校验或保护机制，这既无助于提高代码分析的覆盖率和准确率，又增加了测试资源开销. 因此，关注对程序中普遍存在的保护机制的绕过能力，深入程序内部处理逻辑，并使得测试资源的投入向有利于测试的方向倾斜，将有利于降低模糊测试的分析开销，提高程序缺陷检测的准确率和效率. 检测效率与模糊测试工具对受检程序内部信息的掌握程度密切相关，测试工具发展方向也由最初的完全随机向逐渐具备有效策略指导转变[23]，而指导策略的有效性依赖于对受检程序运行所获得的相关信息量. 同时，受检程序的代码体量与运行环境的复杂性等影响着内部信息的驳杂程度，将导致部分实验结果的准确率下降. 在测试阶段对捕获到的信息进行清洗，有助于降低测试复杂度并缓解准确率下降.

本文考虑多因素影响，要求能有效提升样本代

码覆盖率,并结合检出率引导测试方向的策略,提出了一种新的基于执行上下文信息导向的程序缺陷模糊检测方法,其本质属于基于样本突变的灰盒模糊测试.相比于传统随机发散的模糊测试,对程序脆弱点的暴露能力、受检程序代码的探索效率、突变生成测试用例的质量等方面均有很好的提升.主要贡献有3个方面:

1)引入共性逻辑概念,将程序保护机制的识别归为共性逻辑提取的范畴,基于程序面向不同输入的上下文信息存在差异的特点,提出了共性逻辑提取算法,并在干涉规则的约束下,提出了对程序内部结构的探索提升算法,有很好的保护机制突破能力.

2)提出了突变操作相关区域潜力值*pValue*的更新算法,结合局部性原理为测试用例的突变操作进行的位置进行潜力值评估,有利于优化模糊测试的样本生成策略,指导新测试用例生成.既提供了有效测试用例,也保留了模糊测试本身的随机性.

3)基于以上方法,设计并实现了基于执行上下文导向及保护突破的程序缺陷模糊检测原型工具CBFuzzer,在程序缺陷的模糊检测应用上有很好的表现.利用CBFuzzer在13个测试任务中发现了126个新的漏洞并取得了相应的CVE ID(见附录A表A1,截至2023年8月10日,125个编号已通过审核).

1 相关工作

1.1 模糊测试对输入用例结构的探索

高效的模糊测试工具往往依赖于对受检程序的内部逻辑及结构,甚至开发者的习惯等信息的掌握.通常,利用符号执行、路径分析、程序插桩等技术来获得输入用例的逻辑,并建立相应的模糊测试用例.研究表明将符号执行与模糊测试结合可以获得比各自单独分析更好的效果[24].大部分符号执行工具基于底层编译器中间表示进行分析,包含了详细的源代码信息,具有较高的测试精度,依赖于约束求解产生覆盖复杂条件分支的测试用例[25],其主要挑战在于符号执行中的路径爆炸和运算符的完整解析能力.

符号执行与路径分析密切相关.Noller等人[26]基于模糊测试生成路径相关的输入,并以符号执行生成满足条件的输入作为补充,实现了基于模糊化工具(Kelinci)和符号化路径分析的程序评估.Ognawala等人[27]通过模糊化孤立函数来发现漏洞,并采用符号执行确定漏洞被利用的可行性,在代码覆盖率、漏洞发现量和时间消耗等方面具有优势,不足在于若有影响孤立函数内部状态的全局变量,易引起路径搜索空间爆炸.Sargsyan等人[28]针对大型代码库的定期模糊测试的时间问题,通过提取用于分支条件的常数值,作为对输入缓冲区进行突变的依据,能更快地覆盖程序路径.Zheng等人[15]提出了一种程序代码的静态模糊变异方法,在漏洞进化规律的指导下,检测威胁程序安全的潜在进化路径,以提高在语法树级别生成包含漏洞的突变样本的效率.Huang等人[29]针对不可达路径导致计算资源浪费问题,提出了一种虚假执行路径的修剪方法,在路径中引导灰盒模糊器,以避免不必要的程序执行.以上方法在减少模糊测试中的路径冗余、建立有效的测试用例,以及降低人工成本等方面具有意义,给本文的测试样本在漏洞产出率引导突变方向提供了启发,但在深层次的路径关联、代码保护机制影响等方面仍有许多工作要做.

程序插桩用于模糊测试中有利于获得输入用例结构的状态分布,进而完成种子选择并实现均匀的程序状态覆盖[30],成为覆盖率导向的模糊测试常见的技术.极具代表性的是2013年Zalewski[31]发布的模糊测试工具AFL,通过对受检程序进行轻量级插桩处理和共享内存区域维护捕获程序运行的内部信息,并开创性地用于模糊测试的种子文件选择、测试用例评估等过程,在程序的代码覆盖率和漏洞测试效率上有很好的表现,对模糊测试领域产生了深远的影响.如UltraFuzz[32],Fuzz4B[33]等都是基于AFL衍生而来.此外,由谷歌公司维护的OSS-Fuzz[34],LibFuzzer[35],Honggfuzz[36]等,以及微软公司维护的SAGE[37],Project OneFuzz[38]等工具均使用了插桩技术,以帮助捕获目标信息,制定相应的模糊策略.在模糊测试实践中,一方面,使用基于编译器的静态插桩作为辅助已成为事实上的标准,主要用于安全性强化和导向信息收集,有利于提升程序缺陷的发现能力[39].另一方面,动态二进制插桩允许在不修改源代码的情况下,插入自定义代码,监控程序的执行过程,有助于掌握模糊测试输入用例的结构情况,并有利于测试用例的完善和优化[40].尽管如此,插桩过程存在的插桩代码的准确性、代码注入带来的性能影响等问题仍是插桩技术面临的挑战.

1.2 模糊测试中输入输出关系的分析

模糊测试通过构造各种异常和随机数据作为输入数据,并提交模糊器输出的测试用例给受检程序,在这个过程中检测目标系统对不合理或意外情况的反应.对输入输出关系的分析,有助于理解模糊器生

成模糊数据的过程及其性质，最终为改进模糊器的生成算法以及评估模糊测试用例的质量提供保障. 黑盒模糊测试需要在不访问源代码的情况下通过构建攻击来发现应用程序中的漏洞[9]. 申莹珠等人[41]提出协议状态模糊测试技术，利用协议中的客户端消息集和服务器端响应消息集构造输入输出关系，对网络安全协议的实现逻辑进行黑盒测试分析，发现了多条期望路径外的特别路径及可能的安全隐患. 灰盒测试则可以利用并分析部分目标运行信息，制定有效的测试策略，保障生成的输入数据的测试有效性. Doupé等人[42]提出了一种应用程序内部状态机推断的方法，即通过遍历 Web 应用程序的状态机，来查找和模糊用户的输入向量，并观察与输出的差异关系，以此来发现程序安全缺陷. Alexi 等人[43]为已有代码建立相关记录值数据库，并据此驱动模糊器来生成输入，在 R 语言中进行了模糊验证，具有较好地揭示函数中的签名输出和分析程序行为的能力.

模糊测试要求生成的输入与程序状态保持适宜性并能完整执行，才能保障测试的有效进行，而黑盒测试情况缺少对目标信息的掌握，使得黑盒相比灰盒在实际应用上有更大的局限性. Cheng 等人[44]研究了一种策略驱动的黑盒模糊器框架，为每个接口设置唯一的模糊器，并精确定义目标接口的消息格式、状态维护方法、需要变异的字段位置和特定的输入变异方法. 这些信息有利于生成模糊输入并发现程序中的深层执行缺陷，但大量原始种子信息的生成计算、多请求消息的处理等对模糊器工作影响较大. Redini 等人[45]建立用于黑盒测试中生成最佳模糊输入的函数(称为模糊触发器)，在任何数据转换函数之前执行，并在输入验证代码之后再次执行. 该模糊触发器生成的输入不受应用端净化代码的约束，也不会因其无效格式而被丢弃. 这为对软件保护机制的识别与突破研究提供了启示. 对模糊测试过程中输入输出关系的分析，一方面要考虑有效的输入与对应的输出异常之间的关系，以确定程序缺陷的存在. 另一方面要注意模糊输入的有效性，避免因保护机制的验证约束导致模糊输入无效和较低的代码覆盖率.

1.3 存在的问题

当前模糊测试的诸多方案是就如何提升模糊测试的结果表现而展开，仍然面临两大技术问题：一是如何突破程序中存在的软件保护机制，使得测试能覆盖到程序深层次的代码逻辑；二是如何有效生成新的测试用例，完成程序内部执行路径探索的优化以及污点、漏洞等缺陷产出效率的提升.

针对软件保护机制的识别与突破的问题，通常在校验位置处进行相关代码的转译、重写或修补[46]，依赖于对软件代码的静态分析，结合软件动态运行信息的捕获、分析与策略制定等，往往存在测试前期开销大、样本修复不易和复现困难等情况[3]. 例如，Li 等人[18]注意到传统模糊测试由于变异的粒度固定，因此重写源码中的相关校验代码，使得校验字段首字节的枚举变异能正确反映用于指导模糊器突变的代码覆盖率，但相比其他方案模糊测试的速度严重下滑.

针对如何高效地生成测试用例以完成对受检程序的漏洞发现和程序内部探索的问题，主要考虑样本突变的内容和方向性选择[20]. 比如，Lemieux 等人[19]注意到常规模糊测试策略很难覆盖到的某些分支，将其定义为稀有分支，利用分支标识和分支命中次数的结构来引导样本突变向能够触及稀有分支的方向倾斜，实现了一个 AFL 优化的工具 FairFuzz，一定程度上规避了传统的依靠代码覆盖率导致部分代码无法得到有效测试的问题.

2 提出的方法

2.1 研究动机

传统模糊测试方法无论是用例的结构分析、受检程序内部探索的策略，还是模糊器的输入生成，都具有较大随机性和发散性，导致模糊测试的效率不高，这是实现对受检程序的保护机制识别与突破、内部结构快速探索的难点所在. 本文根据保护机制、程序内部代码分支等的实施特征，提出相关内容的上下文的捕获、分析并利用，以达到对程序内部的快速探索，包括对保护机制的突破. 研究发现 AFL, LibFuzzer, Honggfuzz 等模糊测试工具都是通过预定义一些突变运算符，决定对种子在哪里(如哪些字节)进行突变以及如何突变(如添加、删除或替换字节). 同时，不同的变异操作在同一测试程序上挖掘有价值测试用例的效果不同，以及同一操作在不同程序上的效果也不同，特别是如 AFL 在低效的变异操作上浪费了较多的时间[47]. 因此，对模糊测试进行突变和适当干预，并观察突变影响用来对测试样本进行调整修复，以有利于获得高质量的模糊测试用例.

基于上述思想，本文研究一种新的程序缺陷模糊检测方法，提出实现受检程序内部快速探索(包括对保护机制的突破)，以及高质量测试用例生成的解

决方案. 该方法对目标程序执行中涉及程序状态转移的部分指令具体的上下文信息进行捕获与分析，设计并实现相关的探索提升算法完成对受检程序内部的快速探索. 同时，通过历史突变操作进行的位置记录，根据突变后对应测试用例的结果表现，综合评估该突变位置所在区域的潜力，优化样本突变策略，以指导有效测试用例的生成.

2.2 总体方法

高效的模糊测试应该将有限的算力投入到有价值和有意义的测试中，因此，有必要运用收敛模糊测试过程中随机性的策略，以提升整体测试表现. 方法的核心在于模糊测试的随机性收敛，依赖于对程序运行过程中的执行上下文信息的分析，构建相应的测试指导规则，引导模糊测试的有效进行.

定义 1. 执行上下文信息. 是指正在执行的程序或线程的上下文环境，包括但不限于程序计数器、堆栈信息、寄存器状态、标志寄存器等.

执行上下文是程序执行到某程序位置时的一种程序状态描述. 这种状态随着程序逻辑的执行而时刻变化，相比于静态分析方法，动态地对执行上下文进行分析更能准确地把握程序的执行状况. 本文关注的执行上下文信息是标志寄存器，特别是将条件转移指令的执行上下文信息作为采集对象，原因在于这类指令关乎执行流程和执行状态的转移，进而表现出程序的不同逻辑走向，而保护机制的实施正符合该特点，因此将保护机制归类于条件转移指令的一部分.

基于执行上下文导向及保护突破的程序缺陷模糊检测方法，主要包含 5 个环节：

1)执行上下文信息采集. 主要包括功能性插桩和初检 2 部分. 前者根据需要，有针对性地对目标程序进行插桩；后者根据种子用例进行分类，分别将相应类标签下的子样本输入给程序，并在程序执行过程中，根据插桩位置收集相应的执行上下文信息.

2)共性逻辑提取. 根据分类后测试用例执行过程中收集到的上下文信息，提取程序的共性逻辑，其过程包含对保护机制的识别.

3)内部快速探索. 对程序内部结构进行快速探索，包含对保护机制的突破，以及通过对非常规程序执行状态的模拟，达到对隐匿性强的脆弱点进行尝试性发现.

4)用例修复. 对内部快速探索环节中发现有价值内容的测试用例进行修复，以保证其可复现性.

5)样本突变指导. 用于指导生成高质量的模糊测试样本，同时保留模糊测试本身具备的随机性.

3 执行上下文信息和突破保护的模糊检测

3.1 执行上下文信息采集

3.1.1 功能性插桩

出于对程序运行时的上下文信息收集与分析，以及必要时对上下文实施干涉的需要，基于执行上下文信息导向的程序缺陷模糊测试方法首先完成对目标程序进行传统插桩之外的功能性附加(主要是上下文捕获的动作)，即是在传统插桩代码上额外添加上下文收集的插桩代码.

一段 AFL 的原生插桩代码如图 1 所示. 其中，根据 *use_64bit* 标志来选择使用 *trampoline_fmt_64* 版本或 *trampoline_fmt_32* 版本的插桩代码. 图 2 是在 AFL 插桩 *trampoline_fmt_32* 版本基础上，通过功能性插桩代码部署 *__log_context_action* 方法来采集执行上下文信息. 如果插桩场景为 *use_64bit* 标志位无效，同时识别到当前插桩位置是条件转移指令时，则在当前插

```
1 //来源: afl-as.h,以下是匹配非jmp的条件跳转指令
2 if (line[1] == 'j' && line[2] != 'm') {
3    fprintf(outf, use_64bit ? trampoline_fmt_64 :
           trampoline_fmt_32, R(MAP_SIZE));
4    ins_lines++;  //插桩计数
5 }
6 //...
7 if (ins_lines) {
8 fputs(use_64bit ? main_payload_64 :
        main_payload_32, outf);  //插入插桩代码
9 ...}
```

Fig. 1 A section of code related to AFL instrumentation

图 1 一段 AFL 插桩代码

```
static const u8* trampoline_fmt_32 =
  "\n"
  "/* --- AFL TRAMPOLINE (32-BIT) --- */\n"
  "\n"
  ".align 4\n"
  "\n"
  "call __log_context_action \n"        <-- 功能性插桩部署
  "leal -16(%%esp), %%esp\n"
  "movl %%edi,  0(%%esp)\n"
  "movl %%edx,  4(%%esp)\n"
  "movl %%ecx,  8(%%esp)\n"
  "movl %%eax, 12(%%esp)\n"
  "movl $0x%08x, %%ecx\n"
  "call __afl_maybe_log\n"
  "movl 12(%%esp), %%eax\n"
  "movl  8(%%esp), %%ecx\n"
  "movl  4(%%esp), %%edx\n"
  "movl  0(%%esp), %%edi\n"
  "leal 16(%%esp), %%esp\n"
  "\n"
  "/* --- END --- */\n"
  "\n";
```

Fig. 2 Extra instrumentation based on AFL

图 2 在 AFL 基础上的附加插桩

桩位置完成附加有*__log_context_action*功能的桩代码(即*trampoline_fmt_32*)的添加. 在运行阶段, 相关桩代码被执行时, 由*__log_context_action*完成此程序执行时刻的上下文收集.

3.1.2 初 检

插桩环节完成上下文捕获的陷阱设置, 而上下文捕获的动作需要事件来触发. 因此, 对受检程序执行完功能性插桩后, 由初检环节提供程序输入并驱动程序运行, 通过事件对插桩代码进行触发并捕获执行上下文信息. 初检是以种子用例为主要操作对象, 针对输入的用例在目标程序处理过程中获取执行上下文信息. 对于给定用例所采集到的内容, 可以被视作目标程序对该测试用例处理流程的一种整体性描述, 能够为后续相关环节的执行上下文分析提供数据源.

定义 2. *posDes*. 是当前测试用例在程序中任意的指令位置*pos*处的运行状态, 或者说, *posDes*是程序运行过程中在任意的指令位置*pos*处采集到的、用于描述该位置执行状态的上下文信息, 可描述为⟨*insID*, *context*, *curState*, *historyState*, *isValue*⟩五元组. 其中*insID*是在插桩过程中随机分配给当前程序位置*pos*的唯一标识符, 用于模糊测试工具标识当前位置使用; *context*字段包含采集到的上下文内容; *curState*, *historyState*, *isValue*字段用于方法的后续环节, 如*curState*用于内部快速探索来指示是否对当前程序位置的上下文实施干涉, *historyState*用于指示之前测试过程中是否对该位置实施过干涉, *isValue*指明对*pos*位置实施干涉后测试效果是否提升.

一个关于*posDes*的执行上下文信息采集场景的例子是, 程序位置*pos*存在桩代码, 当*pos*位置被执行前, 桩代码会先行执行, 由*__log_context_action*完成对当前程序位置*pos*的上下文收集, 形成此程序执行时刻的状态描述.

定义 3. *caseDes*. 是测试队列T中每个测试用例*testcase*在目标程序P_{EI}内部执行过程中, 由上下文采集模块*Collector*采集的结果, 它是由*posDes*五元组信息组成的集合, 可描述为

$$caseDes::=Collector(P_{\mathrm{EI}}, testcase),\ \forall testcase \in T.$$

本文依赖于程序面向不同测试用例的处理过程中的执行上下文差异. 在微观的指令层面, 受检程序根据测试用例内容的不同, 实际处理过程中存在上下文差异; 在宏观的测试用例合法性层面, 受检程序内部上下文差异的程度会更加显著, 甚至表现为处理流程的不同. 在初检时, 首先将初始的种子用例*Seeds*按照合法性不同分为2类测试用例: 满足合法性要求的测试用例*wellformedCase*和不满足合法性要求的测试用例*malformedCase*. 随后驱动经过功能性插桩后的程序P_{EI}, 将*wellformedCase*和*malformedCase*中每一个测试用例作为输入提交给目标程序进行处理, 并在过程中由上下文采集模块*Collector*完成相应的上下文采集, 形成*caseDes*.

3.2 共性逻辑提取

定义 4. *wellformedCaseDes*. 是*caseDes*的集合, 其中*caseDes*是在目标程序P_{EI}接收并处理T_1集合中一个测试用例*testcase*的过程中, 由上下文采集模块*Collector*采集的内容, 此处T_1是满足合法性要求的测试用例集合, 可描述为

$$wellformedCaseDes::=\bigcup_{testcase\in T_1} Collector(P_{\mathrm{EI}}, testcase).$$

定义 5. *malformedCaseDes*. 是*caseDes*的集合, 其中*caseDes*是在目标程序P_{EI}接收并处理T_2集合中一个测试用例*testcase*的过程中, 由上下文采集模块*Collector*采集的内容, 这里T_2是不满足合法性要求的测试用例集合, 可描述为

$$malformedCaseDes::=\bigcup_{testcase\in T_2} Collector(P_{\mathrm{EI}}, testcase).$$

共性逻辑提取的出发点是基于目标程序接收不同测试用例的处理流程的差异(如执行的处理流程不同、同一处理逻辑的执行上下文不同等)寻找相同的部分, 该特征也存在于程序的保护机制之中. 基于差异表现, 共性逻辑用来表达程序对某一类测试用例处理逻辑中的核心部分, 包含保护机制的逻辑、相应分类下的处理逻辑. 共性逻辑提取模块根据测试用例的合法性对通过初检采集到的*caseDes*进行分类, 形成*wellformedCaseDes*和*malformedCaseDes*, 三者存在以下关系:

$$\forall caseDes \in (wellformedCaseDes \vee malformedCaseDes).$$

实际上, 对于满足目标程序对测试用例的合法性所采集到的上下文信息*caseDes*1和不满足目标程序合法性要求的测试用例所采集到的上下文信息*caseDes*2是存在差异的, 这种差异可以表现在上下文采集位置*pos*的不同, 也可以表现在相同程序位置*pos*采集到的上下文内容不同(如在程序保护机制所在位置上的采集). 正是这种差异性存在, 给了提取共性逻辑的可操作空间.

插桩后的目标程序接收不同输入测试用例, 其内部的处理逻辑随输入测试用例的不同而存在差异, 这在插桩代码的执行情况上已得到体现, 同时也会

反映插桩代码的命中情况. 例如, 对于某代码桩, 程序在接收某些测试用例后的处理流程中该代码桩被执行, 称为代码桩命中; 反之则称为代码桩不命中. 共性逻辑提取对这部分插桩代码的命中情况进行统计与分析. 兼顾到程序中存在的循环结构可能带来的统计偏差, 本文仅将代码桩的首次命中视为有效计数. 因此, 衡量某处逻辑是否属于该类测试用例共性逻辑的方法是在该类别下的所有测试用例的执行过程中, 对于某个逻辑是否属于该类测试用例共性逻辑的衡量标准(即有效计数值), 按照是否超过衡量的阈值进行判断. 对于保护机制而言, 其在所有测试用例均会被执行到, 所以保护逻辑部分在判定上是属于共性逻辑.

定义 6. *localBpHit*. 是以程序位置 *pos* 对应的 *insID* 为键和 *flag* 为值的哈希表, 可表示为 *hash*[*insID*]= *flag*.

localBpHit 用来描述当前分析的测试用例, 其中 *flag* 字段表明了根据 *insID* 对应逻辑在当前测试用例处理过程中是否为有效计数(即是否为首次命中).

定义 7. *globalBpHit*. 是以程序位置 *pos* 对应的 *insID* 为键和 *hitCount* 为值的哈希表, 可表示为 *hash*[*insID*]= *hitCount*.

globalBpHit 用来对当前类的测试用例进行描述, 其中 *hitCount* 字段记录了针对当前统计到的程序位置 *pos* 以及其对应的 *insID* 在某一类测试用例执行过程中的有效计数的数量值(即首次命中的次数).

算法 1. 共性逻辑提取.

输入: 集合 *caseDesSet*; /*即 *wellformedCaseDes* 或 *malformedCaseDes**/

输出: 提取的共性逻辑 *commonLogic*.

① for each *caseDes* in *caseDesSet* do
② for each *posDes* in *caseDes* do /*遍历当前测试用例的上下文采集情况*/
③ *flag* = *getFlag*(*localBpHit*, *posDes*.*insID*);
④ if *flag* = 0 then
⑤ *setFlag*(*localBpHit*, *posDes*.*insID*);
⑥ *incHitCount*(*globalBpHit*, *posDes*.*insID*);
⑦ end if
⑧ end for
⑨ end for
⑩ *insIDSet* = *globalBpHit*.*keys*();
⑪ for *insID* in *insIDSet* do
⑫ *hitCount* = *globalBpHit*[*insID*];
⑬ if *hitCount* > *threshold* then
⑭ *commonLogic*.*append*(*insID*); /*判定为共性逻辑, 对逻辑位置进行记录*/
⑮ end if
⑯ end for
⑰ return *commonLogic*.

算法 1 中, 行④利用 *localBpHit* 的 *flag* 字段对当前采集到的 *posDes* 是否为首次在 *localBpHit* 命中进行判断. 若为首次, 则视为有效计数, 并使用 *incHitCount* 方法对 *globalBpHit* 中相应内容进行更新. 随后遍历 *globalBpHit*, 检测某类测试用例中逻辑位置 *insID* 在该类测试用例测试过程中有效计数值的情况, 判断该值是否超过阈值 *threshold*(根据当前提取的类中测试用例数量选取阈值), 从而完成对应程序逻辑是否属于该类测试用例的共性逻辑的判定. 设 N 为算法输入的 *caseDesSet* 所包含面向对应类别测试用例上下文提取结果 *caseDes* 的数量, M 为在处理单个 *caseDes* 时所对应的上下文采集点的数量, 那么, 在保证实现目标功能下, 算法 1 的整体时间复杂度为 $O(NM)$, 即与 N 和 M 呈线性关系.

根据初检阶段存在 2 类测试用例的特征, 算法 1 至少启动 2 次, 分别针对 2 类测试用例的执行上下文采集结果形成的集合 *wellformedCaseDes* 和 *malformedCaseDes* 进行测试用例的共性逻辑提取.

3.3 内部快速探索

内部快速探索的工作模式是作为弥补传统测试方法的功能插件而存在, 其操作对象是已提取的共性逻辑, 目的是实现对程序内部关键逻辑的快速探索.

内部快速探索主要解决 2 个问题: 一是程序内部探索效率慢; 二是漏洞或污点本身可能并未表现出破坏性, 但当恶意程序的执行链路对漏洞或污点位置代码进行触发并恶意利用时才显露其破坏性. 对于第 2 个问题, 本文注意到这些恶意构造出来的执行链路往往与不正确、非常规的程序状态转移有关. 传统的模糊测试方法无法对这种状态转移进行模拟, 导致对于这类隐匿性强的脆弱类型的发现能力有限.

内部快速探索依赖探索提升算法, 实现对程序执行流程尝试性地主动干预, 即通过干涉测试用例的处理过程中的执行上下文信息, 以达成突破保护机制, 实现对程序内部的快速探索, 以及模拟非常规程序状态转换的功能目标.

3.3.1 干涉规则

内部快速探索的需求场景是对保护机制的突破、程序内部的快速探索, 以及不正确或非常规的程序状态转换模拟, 其行为特征均与程序中条件转移指令密切相关. 条件转移指令的实施离不开标志寄存

器的内容. 可将 x86 架构的 CPU 中标志寄存器 EFLAGS 的各标志位比特信息作为执行上下文信息的主要内容.

干涉规则是内部快速探索的核心, 主要完成对执行上下文信息的干涉, 实现对程序执行流程的干预. 本文方法对于执行上下文是一种指令级别的采集, 这使得源数据集十分庞杂, 增加了数据处理的负担. 为了减少非必要分析工作的比重, 干涉规则的作用对象限于共性逻辑位置的执行上下文信息. 干涉规则是内部快速探索对执行上下文内容 *context* 进行干涉所遵循的操作规则, 可描述为

$$context\ op\ flipMask \rightarrow flipedContext.$$

其中 *op* 是干涉对应的运算操作, 表示该程序执行时刻的上下文 *context* 与干涉因子 *flipMask* 之间的运算方式; *flipedContext* 是干涉后的上下文.

对上下文实施干涉的本质是对条件转移指令的执行逻辑所涉及的上下文进行干涉, 从而左右指令的转移结果. 根据实施逻辑的复杂程度可分为简单条件转移指令与复杂条件转移指令.

简单条件转移指令对应的 *flipMask* 和 *op* 操作对应关系如表 1 所示. 这些简单条件转移指令, 只需要通过翻转修改 *context* 中对应条件转移指令检查的内容即可完成干涉. 以 je 转移指令为例, 若实施干涉, 则需要对 *context* 中 *ZF* 值进行翻转.

Table 1 Relationship Between *flipMask* and *op* in Simple Conditional Transfer Instructions

表 1 简单条件转移指令的 *flipMask* 与 *op* 关系

条件转移指令	*flipMask* 内容	指令检查的标志	*op*
je/jne/jz/jnz	0x40	*ZF*	XOR
js/jns/jl/jnl	0x80	*SF*	XOR
jp/jnp	0x4	*PF*	XOR
jo/jno/jge/jnge	0x800	*OF*	XOR
jb/jnb/jc/jnc	0x1	*CF*	XOR

由于 *ZF* 标志在 *context*[6], 因此选择 *op* 操作为 XOR, *flipMask* = 0x40, 使用干涉规则 "*context* XOR 0x40 → *flipedContext*" 即可快速完成对 *ZF* 的翻转修改, 从而左右 je 指令的转移结果.

复杂条件转移指令是多个标志位影响转移指令的执行结果, 如图 3 所示. 以 jbe 指令为例, 如图 3(a) 所示, jbe 指令检查 *CF* 和 *ZF* 标志, 当 *CF* 或 *ZF* 至少 1 个有效, 则转移实现. *CF*, *ZF* 标志分别位于 EFLAGS 的第 0 位和第 6 位. 当 "*context*[0] | *context*[6]" 有效, 即 *CF*, *ZF* 至少 1 个有效, 代表未干涉的 jbe 指令结果是转移实现. 内部快速探索若需要对 jbe 指令结果进行干涉, 将此时 *context* 经过干涉规则后的 *flipContext* 在 *CF*, *ZF* 标志均设置为 0 即可.

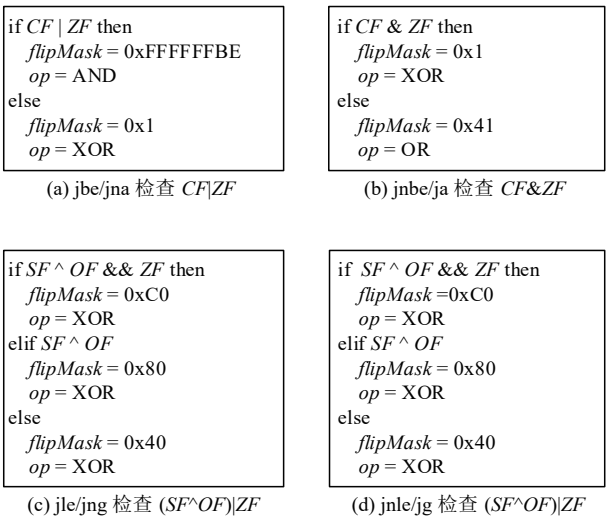


Fig. 3 Relationship between *flipMask* and *op* in complex conditional transfer instructions

图 3 复杂条件转移指令的 *flipMask* 与 *op* 关系

选择 *op* 操作为 AND, *flipMask*=0xFFFFFFBE.当 *context* 中 *CF*, *ZF* 均为 0, 意味着 jbe 指令转移未实现, 内部快速探索若需要对 jbe 指令结果进行干涉, 要求实施干涉后的 *flipContext* 在 *CF*, *ZF* 标志至少 1 个有效即可. 设置 *op*=XOR, *flipMask*=0x1, 对 *context* 的 *CF* 实施翻转修改即可实现干涉. 总之, 图 3 所示的逻辑在本质上是对于完成干涉后的上下文 *flipContext*, 在对应条件转移指令的分支转移表现与干涉前 *context* 上的分支转移表现是相反的.

3.3.2 探索提升

探索提升依赖于 *fork* 函数方法. 在 Unix/Linux 操作系统中可以通过 *fork* 创建新进程的系统调用函数, 也可以创建与原进程一样的新进程. 新进程从原进程的当前位置开始运行, 并使用原进程的资源和状态, 包括虚拟地址空间、文件描述符、信号处理器等.

算法 2. 探索提升.

输入: 测试用例 *tesecase*, 测试用例上下文采集情况 *caseDes*, 受检的目标程序 P_{EI}, 测试用例所属类下的共性逻辑 *CL*(*commonLogic*);

输出: *repairQueue*.

① for run P_{EI} on *testcase* until *insID* match *insID* in *CL* do

② *curPosDes* = *getPosDes* (*caseDes*,*insID*); /*获取当前程序位置的上下文采集内容*/

③ if *curPosDes.historyState* = 0 then

④ *curPosDes.curState* = *R*(); /*随机函数*/

```
⑤  elseif curPosDes.historyState & isValuable then
     /*此前对该处逻辑的干涉并存在表现提升*/
⑥    curPosDes.curState = R();
⑦  else
⑧    curPosDes.curState = 0;
⑨  end if
⑩  if curPosDes.curState then
⑪    if fork() = 0 then    /*在子进程中*/
⑫      modifyContext(); /*对上下文进行干涉*/
⑬      (trace_bits, result) = subprocessWake();
⑭      if result ≠ boring then
⑮        isvaluable();
⑯        repairQueue.add(testcase,caseDes);/*需
            要进行样本修复的队列*/
⑰        DiscoverBoostPool.add(testcase,
            caseDes);
⑱      end if
⑲    end if
⑳  end if
㉑  Parent process test continue;
㉒ end for
```

程序接受测试用例 *tesecase* 并开始执行, 当程序运行位置 *insID* 属于共性逻辑时, 算法 2 的行②通过 *getPosDes* 方法对当前共性逻辑对应的 *insID*, 在测试用例 *testcase* 整体上下文收集结果 *caseDes* 中, 获取对应的上下文描述信息, 并以定义 2 的五元组形式存入 *curPosDes* 中. 行③~⑨的运算逻辑决定 *curState* 字段的内容, 该字段用于指示探索提升是否需要对该处上下文实施干涉, 其中方法 *R* 随机映射为 0 或 1, 用以保留模糊测试自身的随机性. 当 *curState*=1, 算法 2 需要对此位置实施干涉, 在当前测试位置 *fork* 出子进程. 父进程则按照行㉑以常规测试流程(不干涉程序流程)进行测试. 需要注意的是对子进程的处理, 即行⑫使用 *modifyContext* 方法对子进程内部的执行上下文实施干涉. 随后放行子进程, 并观测后续测试的结果. 行⑭, 若存在有利于测试结果的提升, 如新路径发现、程序异常等, 则启动 *isvaluable* 方法对指示该轮测试中实施干涉的程序位置(即 *posDes* 中 *curState* 字段为 1 的部分)进行标记, 或者说, 对能带来测试提升的干涉位置所对应的 *posDes* 中的 *isValue* 字段设置为 1, 用来表示干涉该位置对测试表现有提升. *isValue* 字段为后续是否继续对相同位置进行干涉尝试提供参考. 最后, 将测试用例 *testcase* 添加到对应的队列中.

算法 2 包含采用算法 1 实现共性逻辑提取的输出模块以及干涉操作模块. 干涉操作模块的复杂度为 $O(P)$, 其中 P 是目标程序对实施干涉的后续测试的开销, 2 个模块之间的数据传递开销为 $O(K)$, 其中 K 是共性逻辑模块的输出所包含的逻辑数量. 因此, 算法 2 的整体复杂度为 $O(NM)+O(P)+O(K)$.

图 4 是一段程序的某段处理逻辑映射到算法 2 上的核心操作流程, 其中, Vulnerable Point 是脆弱点, 模糊测试期望对该脆弱点进行暴露. 因此, 需要对 2 处关键的分支转移进行控制: 一是图 4 中实线方框部分, 此为共性逻辑中与保护机制有关的特殊部分; 二是条件转移指令 je LBB0_3. 算法 2 对二者的干涉手法是一致的, 以后者为例, 具体的干涉流程有 4 个步骤:

①对应算法 2 中行③~⑨的逻辑, 使用方法 *R* 在保留一定随机性的基础上对该位置的 *curState* 字段进行设置, 用于指示是否实施干涉. 如图 4 中的第 1 步, 方法 *R* 设置 *curState* = 1 表明对该位置实施干涉.

②通过 *forkserver* 创建出子进程. 处理逻辑根据父子进程而不同, 其中, 虚线箭头部分表示父进程按照传统测试流程进行(不实施干涉), 而子进程部分对应虚线框子进程内容部分.

③在子进程内部, 根据此时条件转移指令 je 完成对干涉规则中的 *flipMask* 及运算规则 *op* 的设置.

④对上下文进行干涉, 干涉的结果使得 je 指令的执行结果区别于干涉之前的执行结果.

图 4 展示了示例程序关键汇编指令映射到算法 2 上的操作流程, 完成了 je 指令执行结果变为步入脆弱点(即 Vulnerable Point)所在程序位置, 同时在父子进程中使得 je 指令的 2 个逻辑分支都被纳入测试面, 即为算法 2 的探索提升过程.

采取类似的实施流程进行干涉可以达到对保护机制的突破. 当出现有价值的发现, 才进行相应输入值的求解, 这样可以减少不必要的算力投入, 并且相比于完全依赖于模糊测试随机发散, 通过碰撞求解获得突破保护机制具体值的方式, 在测试过程中具有更高的可靠性.

3.4 用例修复

为了找出使程序表现异常的输入, 要求测试结果对于当前样本在目标程序执行过程中的上下文具有可复现性. 可以根据给定样本对该样本的执行上下文的复现, 进而分析出程序执行到触发脆弱点的完整调用链路, 实现对脆弱点的定位、分析与修复. 在反馈程序脆弱点给相应的开发者时, 需要漏洞披

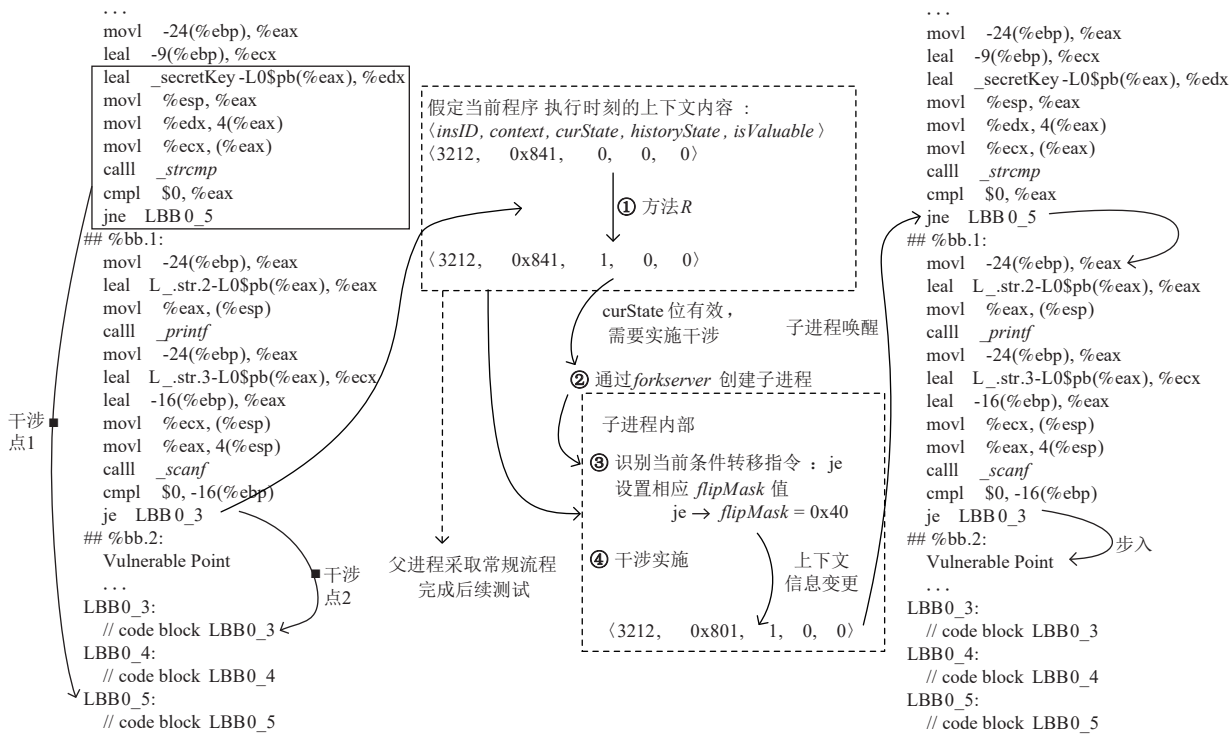


Fig. 4 The operation process of mapping the key assembly instructions of a sample program

图 4 程序关键汇编指令与探索提升算法的映射的操作过程

露结果的可复现性作为支撑.

内部快速探索的作用过程能够强行干涉程序的执行流程，这是出于探索提升算法核心功能的需要. 不幸的是，这种强行干涉会破坏当前样本执行结果的可复现性.

用例修复方法对测试过程中由于干涉程序流程而发现有价值内容（如程序发现新执行路径、出现安全违例、程序崩溃等）的测试用例进行修复. 对程序执行流程进行的干涉属于一次性的干涉，执行结果并非持久，即同样的测试样本，若在目标程序执行过程中不施加相同的干涉，则无法对执行结果进行复现. 用例修复的核心功能便是对存在探索价值发现的测试样本进行修复，达到对测试用例执行结果持久化的目的.

用例修复引入符号执行技术，通过程序分析得到让特定代码块被执行的输入，契合了用例修复的功能需求. 将输入的位置进行符号化，在程序运行过程中，根据程序的执行逻辑给该符号施加约束，当程序执行到目标代码块时形成该符号的一个约束条件集合，最后使用约束求解器完成对约束条件集合的求解. 对于用例修复的需求场景，只需要对内部快速探索环节施加干涉的位置进行轻量化的符号执行技术应用. 用例修复操作的对象是探索提升算法所维护的 *repairQueue*. 探索提升通过对上下文干涉实现劫持程序的执行流程，将对模糊测试结果有提升的样本添加到待修复的队列中. 添加的内容包含测试用例 *testcase* 和对应的描述信息 *caseDes*，其中 *caseDes* 描述了对应测试用例在目标程序执行过程中的干涉情况.

用例修复需要完成以下核心功能：首先，当 *repairQueue* 不为空，取出 *repairQueue* 中的〈*testcase*, *caseDes*〉对，遍历 *caseDes* 并筛选出该 *testcase* 实施干预的程序位置，即五元组信息中 *curState* 字段为 1 对应的插桩位置；接着利用逆向分析技术、hook 技术得出程序位置相关操作数位于 *testcase* 中的对应内容；最后将 *testcase* 中相关内容进行符号化，启动符号执行引擎，求解状态转换符号所对应的值.

3.5 样本突变指导

样本突变策略用于指导突变操作并生成新测试用例. 该策略的优劣对于模糊测试的结果表现至关重要. 在样本突变策略的指导下通过不断突变生成测试用例，期望在这过程中的某些测试用例得以触发程序异常或安全违例等事件发生，从而达到对程序内部运行异常位置进行检测的目的. 因此，突变环节的策略设置很大程度决定了突变生成高质量测试用例的效率. 高质量的测试用例能在代码覆盖率、程序异常发现数量等指标上表现优异. 与传统方法相比，样本突变指导维护着生成新测试用例环节中突

变操作对应突变位置的潜力值*pValue*,并通过该值为突变策略生成新测试用例提供参考.

AFL是基于代码覆盖率的遗传算法生成策略,通过保留测试结果表现优异的测试样本,并用于产生新测试用例的突变母本,但在具体的突变位置的选择上存在盲目性,导致生成高质量测试用例效率较低.为了解决AFL在生成高质量测试用例表现低效的问题,引入程序局部性原理,假定输入给程序的测试用例的某个字段内容在程序执行过程中存在样本质量指标上的提升,那么,对于该字段周围一定范围内的内容做相应的突变操作所生成的新测试用例,在模糊测试过程中可能带来相同的效益[20].

样本突变模块中的导向性突变并不关心具体输入测试用例每个存储单元里的内容,而主要考虑对突变操作区(MutateZone)进行潜力评估,并设置突变操作实施范围的约束.潜力值*pValue*表示对当前突变操作区的内容进行突变所生成的测试用例,在测试质量评估指标上可能带来提升的可能性.本文样本突变环节中以*pValue*为导向,期望将突变操作实施范围约束在指定的区域内(即*pValue*高的突变操作区)进行,在提升突变所产生的高质量测试用例概率的同时,也保留了模糊测试本身具备的随机性操作空间.

3.5.1 突变操作区构造方法

导向性突变指导的操作粒度,是突变操作区对应于测试用例进行突变操作区构造的大小.突变操作区构造包括突变操作区的划分以及相应描述表的构建.其中,描述表中的表项用于描述测试用例的某个突变操作区信息,包含各个突变操作区的唯一标识*MzId*、由当前突变操作区内容进行计算的校验值,以及突变操作区的*pValue*.需要注意的是,每次突变操作所生成的测试用例会继承母本的突变操作区描述表.区别于母本突变操作区描述表,新测试用例突变操作区描述表改变的是涉及突变导致内容发生改变的对应突变操作区的校验值.

根据指令级别程序对输入的存取单元来构造突变操作区,该操作区的大小决定了对*pValue*评估的粒度.以构造512 b单位的突变操作区为例,首先,接受输入流并将内容填充到突变操作区中,将输入流分成以512 b为单位的块.随后,对当前突变操作区描述表进行构建,根据当前内容计算校验值并设置初始化突变操作区的*pValue*.重复上述操作,循环处理完输入流的全部内容,构造出当前测试用例的突变操作区划分.后续则根据测试用例的执行表现对突变操作区描述表的信息进行更新.

3.5.2 *pValue*更新

*pValue*用于表示一个突变操作区的潜力值,初始值为0.*pValue*越大,意味着对该突变操作区进行突变操作所生成的新测试用例,对于提升模糊测试结果的可能性越大.*pValue*更新策略根据当前测试用例是否第1次被目标程序接收并执行存在差异.

算法3. *pValue*更新.

输入:受检的目标程序P_{EI},用于突变的母本测试用例*testcase*;

输出:更新后的测试用例突变操作区描述表.

① if *isDebut* then /*判断*testcase*是否首次出现*/
② *MutateZoneBuild*(*testcase*);
③ *methodFlag* = 0;
④ else
⑤ *methodFlag* = 1;
⑥ end if
⑦ if *methodFlag* then
⑧ *targetZone* = *pValuePeek*(); /*突变操作位置*/
⑨ *mutateFile* = *AFL_Mutate*(*targetZone*); /*针对*targetZone*进行突变,生成新的测试用例*/
⑩ else
⑪ *mutateFile* = *AFL_Mutate*(); /*AFL突变*/
⑫ *targetZone* = *diff*(*newZInfoSet*, *baseZInfoSet*);
⑬ end if
⑭ (*trace_bits*, *result*) = *Execute*(P_{EI}, *mutateFile*); /*由突变生成的用例表现评估*pValue*/
⑮ if *result* = Crash then /*程序出现异常Crash */
⑯ *pValueINC*(*targetZone*);
⑰ *Tcrash.add*(*mutateFile*); /*发现新的漏洞*/
⑱ end if
⑲ if *result* = NewEdgeFound then /*覆盖率提升*/
⑳ *pValueINC*(*targetZone*);
㉑ *T.add*(*mutateFile*); /*发现新的路径*/
㉒ end if
㉓ if *result* = boring then
㉔ *pValueDEC*(*targetZone*);
㉕ *Discard*(*mutateFile*); /*无新发现,舍弃当前干涉位置*/
㉖ end if

算法3中,*MutateZoneBuild*方法对当前测试用例进行突变操作区划分并初始化所有对应的描述信息形成基本描述表*baseZInfoSet*,并设置*methodFlag*标志.随后根据情况有2种突变策略选择:一种是当测

试用例并非首次出现在测试过程中，即此测试用例至少测试过 1 轮或者是由其他测试用例突变产生，此时，用于描述该测试用例所有突变操作区的 *baseZInfoSet*，满足样本突变指导模块介入策略调整的要求. 另一种是 AFL 原生的突变策略，该策略适用于测试用例首次出现在测试过程的情况，在突变操作区描述表 *baseZInfoSet* 刚完成初始化后，所包含的信息不满足样本突变指导模块介入策略调整的要求，因为此时该测试用例所有突变操作区的 *pValue* 均为初始值，不具备参考价值.

算法 3 是导向性突变组件和执行模块的协作. 导向性突变组件负责突变操作区的维护与测试用例的突变，复杂度为 $O(Q)$，其中 Q 为突变测试用例对应的突变操作区的数量. 执行模块基于导向性突变组件输出的突变测试用例，驱动目标程序对突变测试用例进行处理，其复杂度为 $O(P)$. 2 个模块之间的数据传递开销为 $O(Z)$，其中 Z 是导向性突变组件输出的突变测试用例所包含的突变操作区数量. 因此，算法 3 的整体复杂度为 $O(Q)+O(P)+O(Z)$.

算法 3 中的 *targetZone* 通过行⑧的 *pValuePeek* 方法选取 *pValue* 高的突变操作区作为突变操作的位置. 对于 AFL 原生的突变策略而言，*targetZone* 是由行⑫的 *diff* 方法计算得出，主要是基于突变操作必定对某突变操作区内容进行过更新，这将导致根据该突变操作区的内容重新计算得到的校验值与原值存在差异. *diff* 方法通过比对更新后的 *newZInfoSet* 与更新前的 *baseZInfoSet* 之间每个突变操作区内容的校验值差异，从而定位突变位置所在的突变操作区（即 *targetZone*）. 行⑭将突变生成的新测试用例 *mutateFile* 提交给目标程序 P_{EI} 执行. 行⑮~⑯根据 *mutateFile* 的测试表现，评估对于 *targetZone* 的突变是否带来收益，从而更新 *targetZone* 的 *pValue*.

pValue 的更新具备一定的可拓展性，可根据测试用例其他可能的运行情况，如 timeout, hangs 等进行拓展. 算法 3 展示的是依赖测试用例执行表现的 3 类情况：其一是程序执行出现崩溃，即出现程序异常 Crash.其二是代码覆盖率存在提升情况. 以上二者需要增加 *targetZone* 的 *pValue*. 其三是无提升的 boring 情况，表明当前突变生成的 *mutateFile* 对于测试结果表现不存在提升，则相应地需要降低 *targetZone* 的 *pValue*.

4 实验与分析

4.1 实验环境与数据

实验所用机器的配置是：6 核 3.0 GHz 的 Intel® i5-12500H CPU, 16 GB RAM, Ubuntu LTS 18.04 操作系统. 原型工具开发使用 Visual Studio Code 集成开发环境. 实验选择了 13 个受检项目对本文方法及其原型工具 CBFuzzer 的有效性等进行验证. 受检的 13 个受检项目包括 otfcc, swftools, xpdf, advancecomp, swfmill, pngdec, ffjpeg, jpegdec, jpeg-quansmoorth, jpgoptim, pdfalto, tifig, fdkaac（含 1.0.2 和 1.0.3 版本）. 项目初始测试用例来源于程序开发中公布的示例文件和历史缺陷 poc 文件、网络收集和模糊字典等. 对于各测试任务搜集初始样本，比如，对于 pngdec, ffjpeg, jpegdec 等处理程序，搜集一些合法图片样本和畸形图片样本作为初始测试用例.

4.2 程序缺陷的模糊检测工具 CBFuzzer

参考 AFL 和使用 C 语言设计并实现了执行上下文信息导向和保护机制突破的模糊检测原型工具 CBFuzzer. 在实验阶段对 13 个受检项目进行 CBFuzzer 的应用和测试，取得了较好的实验效果.

CBFuzzer 由上下文信息收集模块、共性逻辑提取模块、队列维护模块、驱动模块、样本突变模块、执行模块 6 个主要功能模块构成，如图 5 所示，其中虚线框部分涵盖本文方法相关的大部分算法、功能以及突变策略调整.

图 5 中，预处理部分主要完成本文方法的准备工作，包括对受检程序 P 进行功能性插桩处理生成 P_{EI}、初始种子用例的初检、共性逻辑的提取三大功能. 队列维护模块管理模糊测试队列，由驱动模块的调度器从测试队列 T 中取出本轮测试所需的测试用例 *testcase*. 样本突变模块是根据突变策略完成对测试用例 *testcase* 的突变生成突变样本文件，并提交给执行模块进行测试. 执行模块主要由探索提升和过程观测器 2 个组件构成，前者在一般测试流程的基础上按照探索提升算法完成模块功能；后者用于观测目标程序 P_{EI} 对输入的处理过程信息，观测结果提供给驱动模块的测试用例评估器进行样本质量评估. 若存在测试结果的提升且无需进行用例修复，则直接将对应测试用例添加到测试队列 T 中；否则说明需要进行用例修复，如果成功修复，则添加到测试队列 T 中，反之则丢弃该测试用例.

4.3 实验结果及分析

模糊测试理论上是可以不断进行的，何时中止测试任务需要测试人员根据测试的过程状态进行判断. 在实验阶段，中止测试任务的判断基准是测试的过程对于测试结果表现的提升很微弱，比如代码覆盖率指标不再上升、不再产生新的异常等.

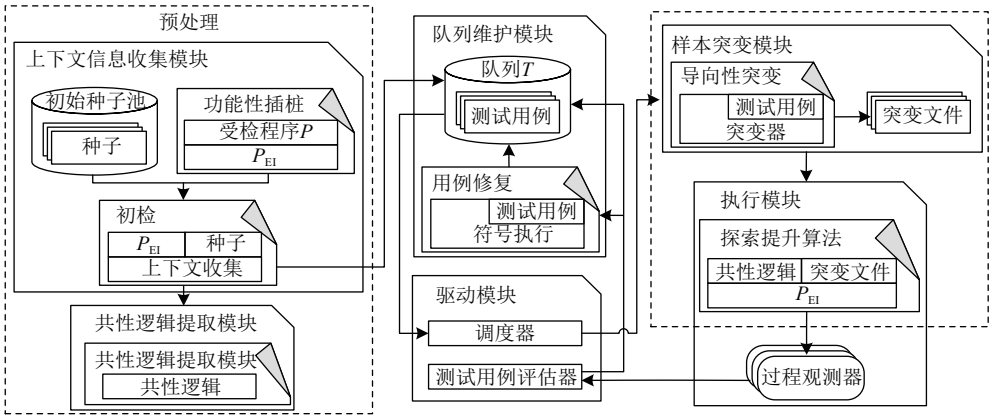


Fig. 5 Functional structure of CBFuzzer

图 5 CBFuzzer 功能结构

利用实验对比验证以下 4 个方面的问题:

1)整体方法的有效性验证.

2)相关关键模块的有效性验证.

3)对不同类型漏洞的检测能力.

4)工具运行性能的分析.

第一, 对于整体方法的有效性验证, 与 AFL[31] 及其优化方案 FairFuzz[19] 的测试表现进行对比分析. 由于本文模糊测试最终目的在于对程序脆弱点进行暴露, 因此通过对目标程序的脆弱点的暴露能力来反映方法的有效性. 表 2 是 CBFuzzer, AFL, FairFuzz 对有代表性的 6 个受检项目 otfcc, swftools, xpdf, swfmill, advancecomp, pngdec 的脆弱点暴露情况.

Table 2 Exposure of Vulnerability Points on Six Test Items for CBFuzzer, AFL and FairFuzz

表 2 CBFuzzer 与 AFL 和 FairFuzz 在 6 个测试项目上的脆弱点暴露情况

受检项目	AFL		FairFuzz		CBFuzzer (本文)					
	Crash 数量	unique Crash 数量	Crash 数量	unique Crash 数量	Crash 数量	对 AFL 的提升率/%	对 FairFuzz 的提升率/%	unique Crash 数量	对 AFL 的提升率/%	对 FairFuzz 的提升率/%
otfcc	136	37	149	42	186	36.76	24.83	49	32.43	16.67
swftools	147	21	161	25	179	21.77	11.18	29	38.10	16.00
xpdf	179	6	196	8	215	20.11	19.79	10	66.67	25.00
swfmill	103	6	113	6	124	20.39	9.73	8	33.33	33.33
advancecomp	134	5	147	6	159	18.66	6.80	7	40.00	16.67
pngdec	56	6	61	6	68	21.43	11.48	7	16.67	16.67

表 2 中, Crash 表示当前测试用例可导致目标程序运行过程中出现安全违例事件, 这些事件由脆弱点被触发而导致. unique Crash 则是去重后的 Crash. 模糊测试方法对于同一处脆弱点的不同触发路径视为不同的脆弱点. 换言之, Crash 指标表示模糊测试工具对于脆弱点的触发能力, Crash 的发现数量是衡量模糊测试工具在测试过程中模块表现的指标, unique Crash 数量是实际脆弱点的检出数量.

根据表 2, 在 6 个受检项目的 Crash 和 unique Crash 的发现数量上, CBFuzzer 相比 AFL 和 FairFuzz 均有不同程度的提升. 与 AFL 相比, CBFuzzer 在 otfcc 项目上的 Crash 发现数量较为显著, 提升率达到 36.76%, 在其他受检项目上提升率均在 20% 浮动, 最低的达到 18.66%. 同时, 在 xpdf 项目上的 unique Crash 发现数量上表现优异, 提升率达到 66.67%. 与 FairFuzz 在 6 个受检项目中的实验数据表现相比, CBFuzzer 在 Crash 发现数量上有不错的提升, 在 otfcc 项目中提升比例达到了 24.83%. 另外, CBFuzzer 在 unique Crash 发现数量指标上相比于 FairFuzz, 提升比例均在 16% 以上. 基于以上实验结果, 在整体的脆弱点暴露情况方面, CBFuzzer 相比于 AFL 和 FairFuzz 均有着很好的程序脆弱点暴露能力的提升, 验证了整体方法的有效性.

第二, 从 CBFuzzer 实施流程角度对实验数据进行分析, 验证 CBFuzzer 关键模块的有效性, 主要包括对探索提升模块和样本突变模块的验证. 其中, 探索提升模块的验证依赖于探索提升算法实现对目标程序内部结构的快速探索, 对于某些探索成功的部分, 可能 AFL 并未对其进行覆盖, 这将导致 Crash 可能存在新的触发路径或新的脆弱点. CBFuzzer 在 Crash 发

现数量上有 2 种来源：一是在生成新测试用例的突变阶段，由突变策略指导的导向性突变；二是由于探索提升算法的介入，导致新 Crash 发现或者新的执行位置被纳入测试范围，从而导致 Crash 有新的触发路径. 样本突变模块的验证可以通过引发程序 Crash 对应的测试用例是否经过用例修复环节来区分其来源.

图 6 和图 7 是 CBFuzzer 与 AFL 在 13 个受检项目中的 Crash 和 unique Crash 对应测试用例的来源构成. 对于 CBFuzzer 的发现，算法 2 和算法 3 决定了 CBFuzzer 测试用例的来源是否经过用例修复环节分为 mutate generate 和 after repair 这 2 个部分. 其中 mutate generate 部分是由 CBFuzzer 突变模块生成的测试用例导致的 Crash, after repair 部分意味着实施过上下文干涉并且经用例修复成功的 Crash 测试用例. AFL 的 Crash 样本来源单一，只有突变阶段的生成来源，这部分有着与 CBFuzzer 在样本突变模块相似的行为，存在一定的可比较性.

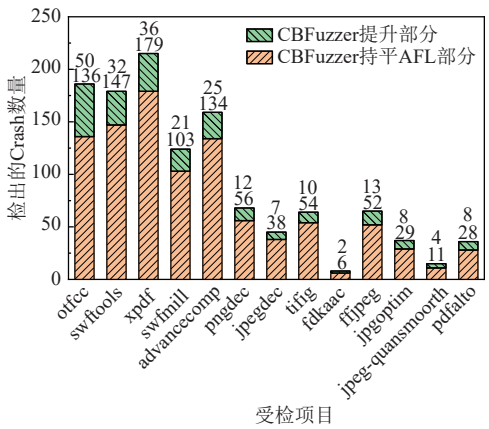


Fig. 6 Comparison of CBFuzzer and AFL on Crash

图 6 CBFuzzer 和 AFL 在 Crash 上的对比

图 6 展示的是 CBFuzzer 与 AFL 的 Crash 发现数量对比. CBFuzzer 相对于 AFL 的 Crash 发现数量在 13 个受检项目中有着不同程度的提升，提升较为显著的是 otfcc, swftools, xpdf, swfmill, advancecomp, 对应的提升数值分别为 50, 32, 36, 21, 25.尽管受检程序的代码体量与测试的结果表现有一定的关系，但同时也说明了 CBFuzzer 通过探索提升环节模块的设置及样本突变模块中对突变策略的调整是有效的，并且相比 AFL 有着更好的测试提升表现.

图 7 是对 CBFuzzer 发现的 unique Crash 进行分类. 其中以增值形式体现的用例修复部分是用例修复环节对于探索提升阶段有价值的发现，并且是成功修复的部分. 该部分数据的存在表明探索提升环节结合用例修复环节在测试阶段正常发挥作用，实现了程序内部快速探索（包括对保护机制的突破）以及模拟非常规程序状态转换的预期目标，并且其中一部分成功修复的样本属于 unique Crash.在模糊测试中往往更注重对 unique Crash 的发掘，图 7 数据验证了探索提升环节对 unique Crash 有着针对性的发掘. 综上，我们得出 CBFuzzer 对于脆弱点的触发能力和暴露能力均优于 AFL，同时也验证了 CBFuzzer 关键模块的有效性.

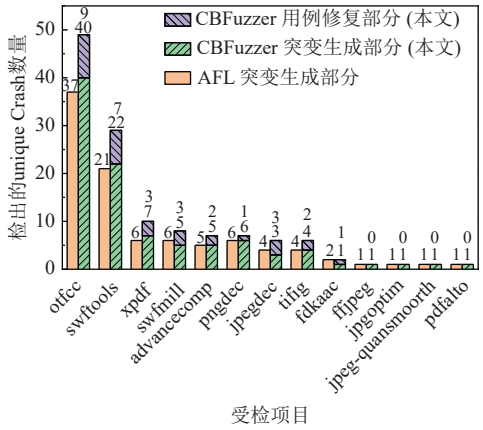


Fig. 7 Comparison of CBFuzzer and AFL on unique Crash

图 7 CBFuzzer 和 AFL 在 unique Crash 上的对比

第三，根据实验过程中检出的漏洞按照漏洞类型进行分类，验证 CBFuzzer 对常规类型脆弱点以及高隐蔽性脆弱点的检出能力. 原型工具 CBFuzzer 在 13 个测试任务中共发现了 126 个新的漏洞，相关信息已报告给软件开发者，并已提交给 CVE®组织[①]，如附录 A 表 A1 所示.

对已检出的脆弱点根据类型进行分类，实验获得了 CBFuzzer 对不同类型脆弱点的暴露能力，如图 8 所示. CBFuzzer 对已分类的 7 种漏洞类型的脆弱点暴露能力均存在着不同程度的提升. 其中 heap buffer overflow 与 SEGV 漏洞类型属于常见漏洞类型，CBFuzzer 在这类常规漏洞类型的检出数量上高于 AFL，检出数量的提升值分别为 21 和 9. 对于其余隐匿性较高的 5 种漏洞类型，CBFuzzer 也有着不错的表现，其中对于 use after free 类型，AFL 在 13 个测试任务中甚至没有检出，而 CBFuzzer 发现了 2 处. 实验结果表明 CBFuzzer 对于常规漏洞类型和隐匿性较高的漏洞类型，均有着更高的检测能力.

第四，在工具的运行性能上，由于 CBFuzzer 附加了额外的功能，在具体实施过程中会增加额外的开

① https://cve.mitre.org/

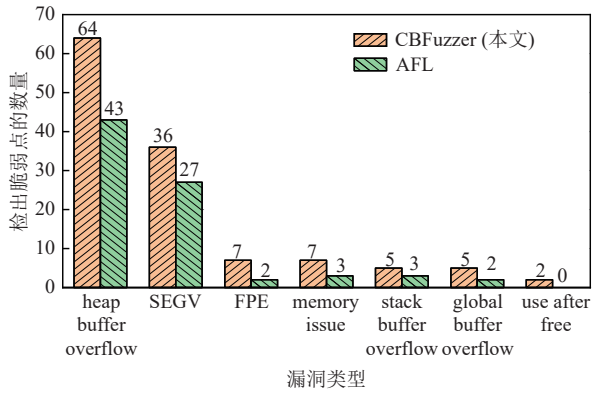


Fig. 8 Exposure ability of CBFuzzer and AFL for different types of vulnerability points

图 8 CBFuzzer 和 AFL 对不同类型的脆弱点的暴露能力

销, 已预期可能使得模糊测试工具的运行效率有一定的下滑.

图 9 描述的是 CBFuzzer, AFL, FairFuzz 这 3 个工具的整体运行性能比较. 其中, 平均执行速度是模糊测试工具中用于衡量测试工具效率的指标, 表示模糊测试工具单位时间内的测试项执行数, 该指标根据测试项和测试方法的复杂程度不同表现存在着差异. 图 9(a)展示的是在不同测试项目上的平均执行速度的对比情况. CBFuzzer 和 FairFuzz 由于额外功能的附加, 增加了二者在测试过程中的复杂度, 使得整体的平均执行速度出现下滑, CBFuzzer 的下滑比例在 21.8% 左右. 图 9(b)显示了 CBFuzzer 对于 CPU 的使用率整体在 AFL 和 FairFuzz 之上, 从实验结果可以看出 3 个工具在实验阶段对 CPU 资源的依赖程度上是 CBFuzzer > FairFuzz > AFL.性能出现一定程度的下滑与系统资源的消耗增大的问题在方法设计初期是可预见的.

性能在可接受范围内出现折中的主要原因是: CBFuzzer 在实现相关功能时, 相比其他模糊测试方法引入更多的信息, 进行了一些策略调整, 比如引入符号执行技术对实施干涉程序位置对应输入值的求解以及对突变操作进行的位置进行评估等, 这些都是导致整体性能出现下滑的原因. 不过, 相比运行性能上的折中, 我们关注实验阶段中 CBFuzzer 的表现结果, 认为这部分测试资源的投入是有价值并且必要的. 另外, 与 FairFuzz 相比, CBFuzzer 在平均执行速度指标上有少量下滑, 主要原因在于测试流程中纳入了其他测试方案所不考量的因素, 例如样本突变模块中对于历史突变操作进行位置的记录以及用例修复中引入的符号执行技术等. 这种对测试过程中的信息进行捕获和运用, 以及外部技术的引入会增

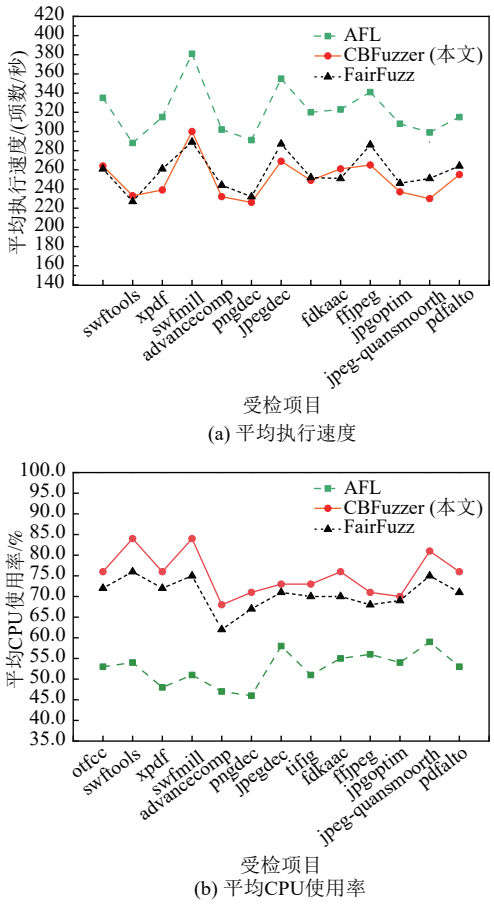


Fig. 9 Comparison of operational performance for CBFuzzer, AFL and FairFuzz

图 9 CBFuzzer 与 AFL、FairFuzz 在操作性能上的比较

加相应测试任务的负累. 相应的解决方案可对相应策略进行优化, 例如用例修复环节可能存在部分约束条件明确无解的情况, 这意味着存在某程序状态转移链路是不可达的, 即对这部分位置实施上下文干涉的测试结果是无法完成测试结果可持久化的, 通过识别并标记这些位置, 在后续测试中不再对它们实施干涉, 避免算力的无效投入.

5 结 论

基于程序面向不同输入所对应的处理逻辑不同或者相同处理逻辑处的执行上下文的不同, 设计相应的方法、模块在这些逻辑位置的上下文信息进行捕获、分析及利用, 提出了一种基于执行上下文信息导向的程序缺陷模糊测试方法, 实现了对目标程序内部结构的快速探索以及样本突变策略的优化. 延续 AFL 的思想, 对本文方法涉及的功能模块进行设计与开发, 形成了基于执行上下文信息导向的程序缺陷模糊测试方法的原型工具 CBFuzzer.通过实验,

在受检的13个受检项目中验证了方法和工具的有效性.

对比AFL, FairFuzz, CBFuzzer的测试表现，CBFuzzer对于受检程序的整体脆弱点暴露能力上优于AFL和FairFuzz,并且对隐匿性较强的脆弱类型的暴露能力也有所提升.在性能方面,CBFuzzer在衡量测试速率的平均执行速度指标上相比AFL下降幅度在21.8%左右,对于CPU的使用率方面,CBFuzzer整体在AFL之上,说明CBFuzzer对于系统资源的依赖程度相较AFL明显.综合考虑测试过程中的资源开销增加与脆弱点的暴露能力提升,CBFuzzer额外的开销是有必要且存在价值的.

下一步工作考虑关联更多程序指令的执行特点与执行上下文之间的关系,对基于执行上下文信息导向的程序缺陷模糊测试方法进行扩展,以提升模糊测试工具对程序内部的探索效率以及更高的脆弱点暴露能力.同时,对相应模块进行设计优化,在保证检出效果的前提下,降低工具对性能和资源的依赖.

作者贡献声明: 唐成华提出了概念、模型和算法思路,完成论文修改;蔡维嘉负责数据整理、算法实现与验证并撰写论文;杨萌萌开展数据及实验项目分析;强保华负责论文审阅与修订.

参 考 文 献

[1] Ribeiro V V, Cruzes D S, Travassos G H. Moderator factors of software security and performance verification[J]. Journal of Systems and Software, 2022, 184: 111137

[2] Ren Zezhong, Zheng Han, Zhang Jiayuan, et al. A review of fuzzing techniques[J]. Journal of Computer Research and Development, 2021, 58(5): 944−963 (in Chinese)
(任泽众, 郑晗, 张嘉元, 等. 模糊测试技术综述[J]. 计算机研究与发展, 2021, 58(5): 944−963)

[3] Peng Hui, Shoshitaishvili Y, Payer M. T-Fuzz: Fuzzing by program transformation[C]//Proc of the 39th IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2018: 697−710

[4] Wang Wenpeng, Chen Zhixiang, Zheng Ziyang, et al. An adaptive fuzzing method based on transformer and protocol similarity mutation[J]. Computers and Security, 2023, 129: 103197

[5] Wu Qingyang, Ning Yuqiao, Guo Zhen, et al. A reflective XSS vulnerability detection method based on fuzzing test[C]//Proc of the Int Conf on Algorithms, Data Mining, and Information Technology. Piscataway, NJ: IEEE, 2022: 25−31

[6] Weng Le, Feng Chao, Shi Zhiyuan, et al. FASSFuzzer-An automated vulnerability detection system for android system services[J]. Journal of Computers (Taiwan), 2022, 22(2): 189−200

[7] Moukahal L J, Zulkernine M, Soukup M. Vulnerability-oriented fuzz testing for connected autonomous vehicle systems[J]. IEEE Transactions on Reliability, 2021, 70(4): 1422−1437

[8] Shu Zhan, Yang Guanhua. IoTInfer: Automated blackbox fuzz testing of IoT network protocols guided by finite state machine inference[J]. IEEE Internet of Things Journal, 2022, 9(22): 22737−22751

[9] Alsaedi A, Alhuzal A, Bamasag O. Black-box fuzzing approaches to secure Web applications: Survey[J]. International Journal of Advanced Computer Science and Applications, 2021, 12(5): 849−855

[10] Borzacchiello L, Coppa E, Demetrescu C. FUZZOLIC: Mixing fuzzing and concolic execution[J]. Computers and Security, 2021, 108: 102368

[11] Le H M. KLUZZER: Whitebox fuzzing on top of LLVM [G]//LNCS 11781: Proc of the 17th Int Symp on Automated Technology for Verification and Analysis. Berlin: Springer, 2019: 246−252

[12] Pham V T, Bohme M, Santosa A E, et al. Smart greybox fuzzing[J]. IEEE Transactions on Software Engineering, 2021, 47(9): 1980−1997

[13] Situ L Y, Zuo Zhiqiang, Guan Le, et al. Vulnerable region-aware greybox fuzzing[J]. Journal of Computer Science and Technology, 2021, 36(5): 1212−1228

[14] Österlund S, Razavi K, Bos H, et al. Parmesan: Sanitizer-guided greybox fuzzing[C]//Proc of the 29th USENIX Security Symp. Berkeley, CA: USENIX Association, 2020: 2289−2306

[15] Zheng Wei, Deng Peiran, Gui Kui, et al. An abstract syntax tree based static fuzzing mutation for vulnerability evolution analysis[J]. Information and Software Technology, 2023, 158: 107194

[16] Ma Rui, He Jinyuan, Wang Xuefei, et al. Mutation scheme for fuzzing based on Thompson sampling[J]. Transactions of Beijing Institute of Technology, 2020, 40(12): 1307−1313 (in Chinese)
(马锐, 贺金媛, 王雪霏, 等. 基于汤普森采样的模糊测试用例变异方法[J]. 北京理工大学学报, 2020, 40(12): 1307−1313)

[17] Pang Chengbin, Liu Hongbin, Wang Yifan, et al. Generation-based fuzzing? Don't build a new generator, reuse![J]. Computers and Security, 2023, 129: 103178

[18] Li Yuekang, Chen Bihuan, Chandramohan M, et al. Steelix: Program-state based binary fuzzing[C]//Proc of the 11th Joint Meeting on Foundations of Software Engineering. New York: ACM, 2017: 627−637

[19] Lemieux C, Sen K. FairFuzz-TC: A fuzzer targeting rare branches[J]. International Journal on Software Tools for Technology Transfer, 2021, 23(6): 863−866

[20] Wang Tielei, Wei Tao, Gu Guofei, et al. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection[C]//Proc of the 31st IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2010: 497−512

[21] Li Tiankai, Li Jianping, He Xi. An improvement of AFL based on the function call depth[C]//Proc of the 18th Int Computer Conf on Wavelet Active Media Technology and Information Processing. Piscataway, NJ: IEEE, 2021: 440−445

[22] Liu Yinxi, Meng Wei. DSFuzz: Detecting deep state bugs with dependent state exploration[C]//Proc of the 30th ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2023: 1242−1256

[23] Xiao Tian, Jiang Zhihao, Tang Peng, et al. High-performance directional fuzzing scheme based on deep reinforcement learning[J]. Chinese Journal of Network and Information Security, 2023, 9(2): 132−142 (in Chinese)
(肖天, 江智昊, 唐鹏, 等. 基于深度强化学习的高性能导向性模糊测试方案[J]. 网络与信息安全学报, 2023, 9(2): 132−142)

[24] Xie Xiaofei, Li Xiaohong, Chen Xiang, et al. Hybrid testing based on symbolic execution and fuzzing[J]. Journal of Software, 2019, 30(10): 3071−3089 (in Chinese)
(谢肖飞, 李晓红, 陈翔, 等. 基于符号执行与模糊测试的混合测试方法[J]. 软件学报, 2019, 30(10): 3071−3089)
[25] Wang Rong, Liu Shaoying, Sato Yuji. SIT-SE: A specification-based incremental testing method with symbolic execution[J]. IEEE Transactions on Reliability, 2021, 70(3): 1053−1070
[26] Noller Y, Kersten R, Pasareanu C S. Badger: Complexity analysis with fuzzing and symbolic execution[C]//Proc of the 27th ACM SIGSOFT Int Symp on Software Testing and Analysis. New York: ACM, 2018: 322−332
[27] Ognawala S, Kilger F, Pretschner A. Compositional fuzzing aided by targeted symbolic execution[J]. arXiv preprint, arXiv:1903.02981, 2019
[28] Sargsyan S, Hakobyan J, Nersisyan L, et al. Improving fuzzing efficiency based on extracted constant values[C]//Proc of the 2022 Ivannikov Open Conf. Piscataway, NJ: IEEE, 2022: 81−85
[29] Huang Heqing, Guo Yiyuan, Shi Qingkai, et al. BEACON: Directed grey-box fuzzing with provable path pruning[C]//Proc of the 43rd IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2022: 36−50
[30] Liu Huayuan, Su Yunfei, Li Ruilin, et al. Structure-statebased graybox fuzzing technique[J]. Journal of Xidian University, 2021, 48(1): 117−123 (in Chinese)
(刘华渊, 苏云飞, 李瑞林, 等. 结构状态覆盖导向的灰盒模糊测试技术[J]. 西安电子科技大学学报, 2021, 48(1): 117−123)
[31] Zalewski M. American fuzzy lop (2.52b)[EB/OL]. [2023-06-12]. https://lcamtuf.coredump.cx/afl
[32] Zhou Xu, Wang Pengfei, Liu Chenyifan, et al. UltraFuzz: Towards resource-saving in distributed fuzzing[J]. IEEE Transactions on Software Engineering, 2023, 49(4): 2394−2412
[33] Miyaki R, Yoshida N, Tsuzuki N, et al. Fuzz4B: A support tool for fuzzing with AFL[J]. Computer Software, 2022, 39(2): 124−142
[34] Ding Zhenyu, Le Goues C. An empirical study of OSS-Fuzz bugs[C]//Proc of the 18th IEEE/ACM Int Conf on Mining Software Repositories. Piscataway, NJ: IEEE, 2021: 131−142
[35] LLVM Project. LibFuzzer —A library for coverage-guided fuzz testing[EB/OL]. [2023-06-13]. https://llvm.org/docs/LibFuzzer.html
[36] Google. Honggfuzz[EB/OL]. [2023-05-28]. https://github.com/google/honggfuzz
[37] Godefroid P, Levin M Y, Molnar D. SAGE: Whitebox fuzzing for security testing[J]. Communications of the ACM, 2012, 55(3): 40−44
[38] Microsoft. OneFuzz[EB/OL]. [2023-06-10]. https://github.com/microsoft/onefuzz
[39] Wang Mingzhe, Jiang Yu, Sun Jiaguang. Static instrumentation techniques in fuzzing testing[J]. Journal of Computer Research and Development, 2023, 60(2): 262−273 (in Chinese)
(王明哲, 姜宇, 孙家广. 模糊测试中的静态插桩技术[J]. 计算机研究与发展, 2023, 60(2): 262−273)
[40] Engelke A, Schulz M. Instrew: Leveraging LLVM for high performance dynamic binary instrumentation[C]//Proc of the 16th ACM SIGPLAN/SIGOPS Int Conf on Virtual Execution Environments. New York: ACM, 2020: 172−184
[41] Shen Yingzhu, Gu Chunxiang, Chen Xi, et al. Vulnerability analysis of OpenVPN system based on model learning[J]. Journal of Software, 2019, 30(12): 3750−3764 (in Chinese)
(申莹珠, 顾纯祥, 陈熹, 等. 基于模型学习的 OpenVPN 系统脆弱性分析[J]. 软件学报, 2019, 30(12): 3750−3764)
[42] Doupé A, Cavedon L, Kruegel C, et al. Enemy of the state: A state-aware black-box web vulnerability scanner[C]//Proc of the 21st USENIX Security Symp. Berkeley, CA: USENIX Association, 2012: 523−538
[43] Alexi T, Pierre D B, Filip K, et al. Signatr: A data-driven fuzzing tool for R[C]//Proc of the 15th ACM SIGPLAN Int Conf on Software Language Engineering. New York: ACM, 2022: 216−221
[44] Cheng Yixuan, Fan Wenqing, Huang Wei, et al. PDFuzzerGen: Policy-driven black-box fuzzer generation for smart devices[J]. Security and Communication Networks, 2022, 2022: 9788219
[45] Redini N, Continella A, Das D, et al. Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for IoT devices[C]//Proc of the 42nd IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2021: 484−500
[46] Liu Xiaolong, Wei Qiang, Wang Qingxian, et al. CAFA: A checksum-aware fuzzing assistant tool for coverage improvement[J]. Security and Communication Networks, 2018, 2018: 9071065
[47] Lyu C, Ji Shouling, Zhang Chao, et al. MOPT: Optimized mutation scheduling for fuzzers[C]//Proc of the 28th USENIX Security Symp. Berkeley, CA: USENIX Association, 2019: 1949−1966



Tang Chenghua, born in 1974. PhD, professor. Member of CCF. His main research interests include network information security, software code audit, and security policy analysis.
唐成华, 1974 年生. 博士, 教授. CCF 会员. 主要研究方向为网络信息安全、软件代码审计、安全策略分析.



Cai Weijia, born in 1997. Master. His main research interests include network information security and code analysis.
蔡维嘉, 1997 年生. 硕士. 主要研究方向为网络信息安全、代码分析.



Yang Mengmeng, born in 1988. PhD. Her main research interests include network information security, privacy preserving, and data mining.
杨萌萌, 1988 年生. 博士. 主要研究方向为网络信息安全、隐私保护、数据挖掘.



Qiang Baohua, born in 1972. PhD, professor, PhD supervisor. Member of CCF. His main research interests include detection and analysis of network behavior, and data mining.
强保华, 1972 年生. 博士, 教授, 博士生导师. CCF 会员. 主要研究方向为网络行为检测与分析、数据挖掘.

附录 A

Table A1 CVEs Obtained by the CBFuzzer(As of August 10, 2023)

表 A1 CBFuzzer 获得的 CVE（截至 2023 年 8 月 10 日）

漏洞类型	CVE ID			
heap buffer overflow	CVE-2022-33 047(C)	CVE-2022-35 034(M)	CVE-2022-35 035(M)	CVE-2022-35 070(M)
	CVE-2022-35 036(M)	CVE-2022-35 037(M)	CVE-2022-35 038(M)	CVE-2022-35 088(M)
	CVE-2022-35 039(M)	CVE-2022-35 040(M)	CVE-2022-35 041(M)	CVE-2022-35 094(M)
	CVE-2022-35 042(M)	CVE-2022-35 043(M)	CVE-2022-35 044(M)	CVE-2022-35 104(M)
	CVE-2022-35 045(M)	CVE-2022-35 046(M)	CVE-2022-35 047(M)	CVE-2022-35 081(M)
	CVE-2022-35 048(M)	CVE-2022-35 049(M)	CVE-2022-35 050(M)	CVE-2022-38 228(H)
	CVE-2022-35 051(M)	CVE-2022-35 052(M)	CVE-2022-35 053(M)	CVE-2022-36 144(H)
	CVE-2022-35 054(M)	CVE-2022-35 055(M)	CVE-2022-35 056(M)	CVE-2022-35 020(M)
	CVE-2022-35 058(M)	CVE-2022-35 059(M)	CVE-2022-35 060(M)	CVE-2022-35 017(M)
	CVE-2022-35 061(M)	CVE-2022-35 062(M)	CVE-2022-35 063(M)	CVE-2022-35 012(M)
	CVE-2022-35 064(M)	CVE-2022-35 066(M)	CVE-2022-35 068(M)	CVE-2022-35 010(M)
	CVE-2022-35 069(M)	CVE-2022-35 098(M)	CVE-2022-35 113(M)	CVE-2022-35 090(M)
	CVE-2022-37 781(H)	CVE-2022-35 080(M)	CVE-2022-36 142(H)	CVE-2022-35 016(M)
	CVE-2022-32 324(C)	CVE-2022-35 109(M)	CVE-2022-36 139(H)	CVE-2022-36 150(M)
	CVE-2022-35 089(M)	CVE-2022-35 105(M)	CVE-2022-35 015(M)	CVE-2022-35 067(M)
	CVE-2022-35 096(M)	CVE-2022-38 229(H)	CVE-2022-35 007(M)	CVE-2022-36 143(H)
SEGV	CVE-2022-35 022(M)	CVE-2022-35 023(M)	CVE-2022-35 024(M)	CVE-2022-36 153(M)
	CVE-2022-35 025(M)	CVE-2022-35 026(M)	CVE-2022-35 027(M)	CVE-2022-36 145(M)
	CVE-2022-35 028(M)	CVE-2022-35 029(M)	CVE-2022-35 030(M)	CVE-2022-35 019(M)
	CVE-2022-35 031(M)	CVE-2022-35 032(M)	CVE-2022-35 065(M)	CVE-2022-38 234(M)
	CVE-2022-32 325(M)	CVE-2022-35 086(M)	CVE-2022-35 087(M)	CVE-2022-36 140(M)
	CVE-2022-35 092(M)	CVE-2022-35 095(M)	CVE-2022-35 097(M)	CVE-2022-35 014(M)
	CVE-2022-35 100(M)	CVE-2022-35 101(M)	CVE-2022-35 106(M)	CVE-2022-38 233(M)
	CVE-2022-35 108(M)	CVE-2022-35 114(M)	CVE-2022-38 235(M)	CVE-2022-36 141(M)
	CVE-2022-36 151(M)	CVE-2022-35 002(M)	CVE-2022-35 000(M)	CVE-2022-35 018(M)
memory issue	CVE-2022-35 433(M)	CVE-2022-35 085(M)	CVE-2022-35 110(M)	CVE-2022-36 155(M)
	CVE-2022-36 146(M)	CVE-2022-35 009(M)	CVE-2022-36 152(M)	
FPE	CVE-2022-35 091(M)	CVE-2022-38 230(M)	CVE-2022-35 434(M)	CVE-2022-34 999(M)
	CVE-2022-35 013(M)	CVE-2022-35 004(M)	CVE-2022-36 148(M)	
stack buffer overflow	CVE-2022-35 107(M)	CVE-2022-35 111(M)	CVE-2022-35 099(M)	CVE-2022-38 227(H)
	CVE-2022-35 008(M)			
global buffer overflow	CVE-2022-35 021(M)	CVE-2022-35 093(M)	CVE-2022-35 011(H)	CVE-2022-34 998(H)
	CVE-2022-35 003(H)			
use after free	CVE-2022-36 149(M)	CVE-2022-35 033*		

注：括号中内容为漏洞的安全等级，根据漏洞 CVSS 评估系统划定，L 代表低危，M 代表中危，H 代表高危，C 代表严重. 另外，带*标的 CVE 编号表示审查中状态.