覆盖率制导的灰盒模糊测试研究综述

崔展齐[1)] 张家铭[1),2)] 郑丽伟[1)] 陈 翔[3)]

[1)](北京信息科技大学计算机学院 北京 100101)

[2)](北京科技大学计算机与通信工程学院 北京 100083)

[3)](南通大学信息科学技术学院 江苏 南通 226019)

摘 要 由于部署简单、可扩展性强、挖掘到大量真实漏洞等原因,模糊测试得到了科研和工程人员的广泛关注.其中,覆盖率制导的灰盒模糊测试(Coverage-guided Greybox Fuzzing,简称CGF)以被测程序代码覆盖率为反馈信息,可对软件进行较为充分的自动化测试,有效地保障软件质量,是目前最为流行的一类模糊测试技术.研究人员为改进CGF投入了大量精力,产生了许多研究成果.然而,目前并未有研究针对CGF的已有研究工作进行系统性综述.为此,本文分析了近年来CGF的相关重要研究成果,将CGF流程划分为4个阶段:预处理、测试用例选择、测试用例演化和测试用例评估,并系统地分阶段分析了已有研究进展.此外,针对现有工作中评估分析设置不一致的问题,本文整理了CGF领域中常用的测试对象、实验设置及评估指标.最后,基于对已有研究进展的分析,阐述了CGF目前在预处理、测试用例选择等阶段存在的局限性、可能的解决方案以及未来的研究方向.

关键词 模糊测试;灰盒测试;覆盖率;测试用例

中图法分类号 TP311 **DOI号** 10.11897/SP.J.1016.2024.01665

A Survey of Research on Coverage-Guided Greybox Fuzzing

CUI Zhan-Qi[1)] ZHANG Jia-Ming[1),2)] ZHENG Li-Wei[1)] CHEN Xiang[3)]

[1)](*School of Computer Science, Beijing Information Science and Technology University, Beijing* 100101)

[2)](*School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing* 100083)

[3)](*School of Information Science and Technology, Nantong University, Nantong, Jiangsu* 226019)

Abstract Due to simple deployment, high scalability, and many real-world vulnerabilities are successfully detected, fuzz testing has attracted the attention of many scientific researchers and industry engineers. Among them, Coverage-guided Greybox Fuzzing (CGF) has become one of the most popular fuzzing techniques. It uses coverage of the program under test as feedback information, which can perform adequate software testing automatically and ensure the quality of software effectively. As a result, researchers have invested considerable efforts into improving CGF, resulting in numerous achievements. However, there is still no systematic survey of the existing CGF research work. For this reason, this paper analyzes the critical research achievements of CGF in recent years, divides the CGF process into four stages: preprocessing, test case selection, test case evolution, and test case evaluation, and systematically summarizes research progress in different stages. Furthermore, to address the inconsistency of evaluation settings in existing works, this paper provides statistics on the commonly used benchmarks,

收稿日期:2023-05-12;在线发布日期:2024-04-09. 本课题得到江苏省前沿引领技术基础研究专项(BK20202001)、国家自然科学基金项目(No. 61702041)、北京信息科技大学"勤信人才"培育计划项目(No. QXTCP C201906)资助. **崔展齐**,博士,教授,中国计算机学会(CCF)高级会员,主要研究领域为智能软件工程及可信人工智能技术. E-mail:czq@bistu.edu.cn. **张家铭**,博士研究生,中国计算机学会(CCF)学生会员,主要研究领域为智能软件分析及测试技术. **郑丽伟**,博士,副教授,中国计算机学会(CCF)会员,主要研究领域为环境驱动的需求工程、智能软件工程和群智算法. **陈 翔**(通信作者),博士,副教授,中国计算机学会(CCF)高级会员,主要研究领域为软件测试、软件维护、经验软件工程和软件仓库挖掘. E-mail:xchencs@ntu.edu.cn.

experimental settings, and evaluation metrics in the CGF field. Lastly, based on an analysis of the current research progresses, this paper discusses the limitations, potential solutions, and future research directions of CGF in different stages, such as preprocessing and test case selection.

Keywords fuzz testing; greybox testing; test coverage; test case

1 引 言

软件漏洞是计算机安全问题的根源[1],如果在软件开发过程未及时发现漏洞,可能会导致严重后果.例如:心脏出血①漏洞(Heartbleed,CVE-2014-0160)在 2011 年 12 月被引入 OpenSSL②,但直至 2014 年 4 月该漏洞才被首次公开,造成了大量经济损失.截至 2020 年 11 月,仍有 20 万台计算机易受到心脏出血漏洞影响③.因此,如何挖掘软件中的未知漏洞成为了研究者们关注的焦点.软件测试是软件生命周期中的一项重要活动,其目的主要包括考察软件是否正确实现了预期功能、检测程序中是否存在潜在的缺陷、检验新版本软件是否引入了新的缺陷等[2].软件测试可有效地提高软件的质量,以确保软件在发布后可以正常运行[3].传统的软件测试方法通常需要手动构建大量测试用例,才能对被测软件进行较为充分的测试,因此会产生大量人力开销.Miller 等人[4]提出的模糊测试有效地缓解了传统软件测试过程中需要消耗大量人力资源以构建测试用例集的问题.经过多年的发展,模糊测试已经发展成为一种可扩展性强、实用性强且自动化程度高的测试技术,在各类软件中成功挖掘到大量漏洞,备受学术界与工业界的关注.

模糊测试是一种将随机生成的测试用例发送至被测程序(Program Under Test,PUT),并观测 PUT 的行为是否符合预期,以挖掘其中潜在漏洞的测试技术.根据对 PUT 内部逻辑了解程度不同,模糊测试可以分为黑盒模糊测试[5,6]、白盒模糊测试[7,8]与灰盒模糊测试[9,10].黑盒模糊测试常用于 Web 应用测试、协议测试、物联网设备测试等场景,代表工作有 Peach④、Sulley⑤、IoTFuzzer[11]等.白盒模糊测试通常与符号执行相结合,利用符号执行帮助模糊测试通过判定条件难以满足的分支,代表工作有 TaintScope[12]、Driller[13]等.灰盒模糊测试对 PUT 进行轻量级的分析或程序插桩,在测试过程中收集部分执行信息(例如测试用例覆盖信息)并利用启发式算法来辅助生成测试用例,代表工作有 EFS[14]、libfuzzer⑥、Honggfuzz⑦、AFL⑧等.

黑盒模糊测试在不考虑 PUT 的内部逻辑的情况下生成测试用例,尽管这样做可以提高模糊测试的吞吐量,但也可能会产生很多质量较低的测试用例.而白盒模糊测试则需要对 PUT 进行大量分析,在充分了解 PUT 内部逻辑的情况下生成测试用例.这样虽然可在测试过程中根据 PUT 的逻辑更有针对性地生成高质量测试用例,但也会大幅降低测试效率.此外,白盒模糊测试通常使用符号执行等程序分析技术,因此也存在可扩展性较低等缺点.灰盒模糊测试结合了黑盒模糊测试和白盒模糊测试的优点,对 PUT 内部逻辑进行轻量级分析后进行模糊测试,可以在保持较高测试吞吐量的同时对 PUT 进行有针对性的测试.迄今为止,灰盒模糊测试已经在 sqlite⑨、exiv2⑩、libxml2⑪等开源项目上发现了大量漏洞,受到了研究人员的广泛关注.

2013 年以来,灰盒模糊测试技术取得了快速发展,产生了大量研究成果.根据制导方式的不同,灰盒模糊测试可以分为两类:一类是以高代码覆盖率为目标的覆盖率制导灰盒模糊测试技术(Coverage-guided Greybox Fuzzing,简称 CGF),代表工作有 AFL、MOpt[15]、MTFuzz[16]等;另一类是以快速覆盖指定代码块为目标,可应用于漏洞验证、漏洞复现与补丁测试的定向灰盒模糊测试技术(Directed Greybox Fuzzing,简称 DGF),代表工作有 AFLGo[17]、

① https://heartbleed.com/
② https://www.openssl.org/
③ https://tuxcare.com/why-your-servers-can-still-suffer-from-a-heartbleed-and-what-to-do/
④ http://www.peachfuzzer.com
⑤ https://github.com/OpenRCE/sulley
⑥ https://llvm.org/docs/LibFuzzer.html
⑦ https://github.com/google/honggfuzz
⑧ https://lcamtuf.coredump.cx/afl/
⑨ https://www.sqlite.org/index.html
⑩ https://exiv2.org/
⑪ https://gitlab.gnome.org/GNOME/libxml2

Beacon[18]、DeltaFuzz[19].尽管模糊测试有着多样的应用场景,但大部分模糊测试技术主要关注挖掘软件中的安全漏洞[20].此外,模糊测试只能发现所覆盖到区域中的漏洞,其漏洞挖掘能力与所能达到的覆盖率密切相关,因此 CGF 是目前研究和应用最为广泛的模糊测试技术.近年来,CGF 技术发展迅速,不仅可以依靠代码覆盖率与收集测试用例执行信息来制导测试用例生成,还可以结合机器学习[16,21]、强化学习算法[22,23]、污点分析[24,25]等技术来提高模糊测试的效率与覆盖率.

目前,在模糊测试领域中,已有部分工作总结了研究进展,如:Manès 等人[20]在 2019 年提出了一个通用的模糊测试模型,对已有的模糊测试工作进行了分类与总结;Wang 等人[26]在 2020 年对已有的 DGF 工作进行了系统化的研究,并总结了 DGF 面临的挑战,为 DGF 的未来发展提出了建议;Saavedra 等人[27]在 2019 年对已有的基于机器学习的模糊测试工作进行了总结,并探讨了将机器学习应用到模糊测试面临的挑战.与已有的工作相比,我们首次针对 CGF 的研究进展展开综述,填补了模糊测试领域中因缺少 CGF 综述而产生的空缺,并根据各项工作的流程归纳了 CGF 技术的框架,为未来的 CGF 工作提供参考.

由于 Manès 等人[20]已在 2019 年为模糊测试工作进行了较为全面总结,因此本文主要关注了 2018 年至今顶级期刊和会议上发表的研究成果.此外,在收集过程中我们还参考了模糊测试论文库 FuzzingPaper①,以尽可能地减少我们在收集过程中可能产生的遗漏.

与已有的研究工作相比,我们的研究工作主要存在如下不同:

一方面,本文补充了重要的研究进展,并总结了 CGF 的工作流程框架.最新的模糊测试综述[28]于 2022 年 9 月发表,其中总结了 2008 年 1 月至 2021 年 5 月的模糊测试工作.在 2021 年 5 月后,有 91 篇新的 CGF 研究工作发表在顶级期刊和会议上.本文在已有综述的基础上对 CGF 领域的最新研究工作进行了总结,如 HashFuzz(TSE, 2022)、SLIME(ISSTA, 2022)、ZAFL(USENIX Security, 2021)等.同时,本文基于 159 篇 CGF 工作归纳了 CGF 工作流程框架,该框架可适用于目前已知的 CGF 研究工作.

另一方面,Manès 等人为规范化模糊测试工作,在 2019 年对模糊测试领域中使用的术语、模糊测试的流程、模糊测试工具的分类等内容进行了总结.本文的研究工作与其相比存在如下区别:1)Manès 等人根据 2019 年之前的模糊测试工作提炼了研究框架,但他们的框架无法适用于在 2019 年后新增的部分 CGF 研究工作,例如在预处理阶段使用静态分析[29,30]或生成初始种子测试用例[31,32]的研究工作,而本文所总结的框架可适用于目前已知的 CGF 研究工作;2)本文总结了 CGF 领域最新的研究进展,并针对 CGF 应用领域的多样性,归纳了各 CGF 工具的应用场景,以便为未来的研究工作中模糊测试工具的选取提供参考;3)本文统计并总结了适合评估 CGF 工作的测试对象、实验设置和评估指标,为未来对 CGF 研究工作进行评估分析的设置提供参考,并为 CGF 未来的研究方向提供建议.

本文的主要贡献如下:

(1)调研了近 6 年 159 篇与 CGF 相关的工作,将 CGF 技术框架划分为 4 个阶段:预处理、测试用例选择、测试用例演化和测试用例评估,并分阶段对已有工作进行了总结,为后续研究工作提供参考.

(2)针对现有工作中实验评估设置不一致的问题,为方便研究人员在未来可以更加客观地评估 CGF 工作,本文统计并整理了 CGF 工作常用的测试对象、实验设置和评估指标,为未来对 CGF 的研究提供实验对象、实验设置及评估指标的参考.

(3)基于对已有工作的调研情况,本文分析与探讨了目前 CGF 工作所面临的挑战和存在的局限性,探讨了可能的改进方案,并对未来的发展趋势进行了展望,以供研究人员进行参考.

本文剩余内容结构安排如下:第 2 节介绍相关 CGF 研究成果的收集过程;第 3 节对 CGF 进行概述,将 CGF 的流程进行划分并依次介绍各阶段的研究进展;第 4 节根据应用场景对 CGF 相关研究成果进行分类;第 5 节介绍 CGF 中常用的测试对象、实验设置和评估指标;最后,第 6 节总结全文,并对 CGF 的发展趋势进行展望.

2　相关论文收集与分析

为了解 CGF 领域的最新进展,我们通过以下步骤收集了相关研究成果.

(1)为确保研究对象具有较高的质量和可信度,使我们可以了解到 CGF 领域的最新研究进展,我们

① https://github.com/wcventure/FuzzingPaper

检索了 2018 年至今在 CCF 推荐 A 类国际期刊/会议、A 类中文期刊上发表的研究成果，包括 Journal of Cryptology、TOPLAS、TOSEM、TSE、TDSC、TIFS、CCS、S&P、ISSTA、NDSS、ICSE、ESEC/FSE、ASE、USENIX Security、《计算机学报》、《软件学报》、《中国科学：信息科学》和《计算机研究与发展》等. 在检索过程中，筛选出文章标题中包含单词“fuzz”、“fuzzing”或“fuzzer”的英文论文、文章标题中包含“模糊测试”的中文论文以及模糊测试相关专题的论文. 筛选结束后，我们收集到 350 篇论文.

(2)为避免在步骤(1)的过程中产生遗漏，我们还参考了模糊测试论文的数据库 FuzzingPaper 中所记录的论文，并从库中额外获取到了 35 篇国际权威会议和期刊上发表的模糊测试论文，作为对步骤(1)的补充.

(3)将步骤(2)获得的论文补充至步骤(1)获得的论文中，共得到 385 篇论文. 然后，下载了所有可以获取到原文的论文，并保留论文内容中包含“coverage-guided”或“coverage-based”的 278 篇论文.

(4)调研了 278 篇论文，并根据以下规则过滤与 CGF 无关的论文：

过滤规则 1：文中明确说明所提出的方法不属于 CGF 领域. 通过此条规则，有 79 篇论文被过滤.

过滤规则 2：文中指出所提方法属于黑盒模糊测试技术或白盒模糊测试技术. 通过此条规则，有 8 篇论文被过滤.

过滤规则 3：论文所提方法使用了白盒程序分析技术(例如静态符号执行或动态符号执行)来辅助模糊测试. 通过此条规则，有 33 篇论文被过滤.

过滤规则 4：论文的篇幅过短(少于 4 页)，未能有效地描述所提方法的框架. 通过此条规则，有 2 篇论文被过滤.

过滤规则 5：当同一项工作存在多个版本时，例如同时在会议和期刊上发表或同时作为工具论文与研究论文发表时，仅将对方法描述最全面的一篇论文保留作为研究对象. 通过此条规则，有 2 篇论文被过滤.

(5)过滤后，我们共得到 154 篇论文作为研究对象. 为了进一步弥补遗漏，我们还利用雪球文献搜索法[33]，对上述研究论文及近年发表的模糊测试综述[20,28,34-39]所引用的参考文献进行了人工检查，并保留了 2018 年至 2023 年间发表的与 CGF 相关的研究论文. 通过此步骤，我们得到了 5 篇论文作为对步骤(4)的补充.

(6)最后，我们对收集的 159 篇论文进一步进行深入分析和研究. 我们从这些论文中提取以下信息，具体包括：论文的发表源与发表年份；论文在 CGF 流程中的哪些方面提出了创新；基于论文中所描述方法实现的工具是否开源、是在哪个已有工具的基础上实现的；是否支持在没有被测项目源代码的情况下进行模糊测试；论文中为评估所提出的工作选用了哪些测试对象、实验设置如何(包括实验重复轮次及实验时间)、以及使用了哪些评估指标.

近 6 年 CGF 相关论文的数量分布情况如图 1 所示. 从图 1 中可以看出，自 2019 年开始，CGF 的论文数量增长迅速，2019 年的 CGF 论文数量与 2018 年相比增长了 250%，2020 年的 CGF 论文数量与 2019 年相比增长了 52.38%. 从 2020 年至 2023 年，CGF 的论文数量虽然整体上略有增加，但每年的论文数量总体保持稳定. CGF 相关论文在各权威期刊和会议的分布情况如表 1 所示，表中数据已按照研究论文数量降序排序. 从表中可以看出，近 6 年发表 CGF 研究论文数量最多的会议为网络信息与安全领域的权威会议 USENIX Security，共发

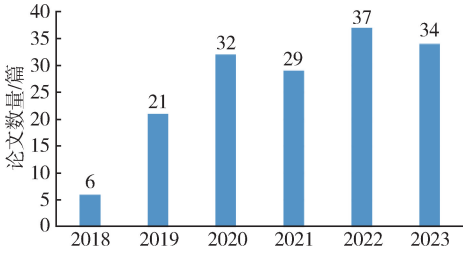

图 1 2018 至 2023 年 CGF 相关论文数量分布

表 1 各权威期刊和会议上发表的 CGF 相关论文数量分布

发表源	类型	论文数量/篇
USENIX Security	会议	36
ICSE	会议	22
ISSTA	会议	19
S&P	会议	18
ASE	会议	17
NDSS	会议	13
FSE	会议	12
CCS	会议	7
TSE	期刊	4
TDSC	期刊	2
USENIX ATC	会议	2
USENIX WOOT	会议	2
TIFS	期刊	2
TOSEM	期刊	2
计算机研究与发展	期刊	1

表了 36 篇,其次为软件工程领域的权威会议 ICSE,共发表了 22 篇. 除此之外,S&P、TSE 等安全与软件工程方面的权威会议与期刊也屡有高质量研究论文发表. 从图 1 和表 1 中可以看出,CGF 近年来受到了广泛关注,产生了大量的研究成果. 然而,目前尚未有一份较为完整的综述总结现有 CGF 工作. 因此,对 CGF 的研究进展与现状进行分析与归纳有着较为重要的意义.

3　研究进展

3.1　概　述

从 1990 年提出至今,模糊测试已成为应用最广泛的软件安全性保障技术之一[40]. 模糊测试的主要目的是为了挖掘软件中所存在的漏洞,以提升软件质量,而覆盖率与模糊测试工具发现漏洞的能力之间密切相关. Miller 的研究①中指出,代码覆盖率每增加 1%,发现错误的概率就会增加 0.92%. 此外,灰盒模糊测试在黑盒和白盒模糊测试之间进行了平衡,在产生较小程序分析开销的情况下可以保持较高的吞吐量. 因此,CGF 是目前研究和应用最为广泛的一类模糊测试技术.

CGF通过对被测程序进行轻量级的程序分析和插桩来追踪测试用例的覆盖信息, 并在测试过程中利用覆盖信息制导测试用例种群的演化,其目的是通过尽可能提高代码覆盖率来检测程序中存在的漏洞. 如今,CGF 已经在 OpenCV②、MySQL③、FFmpeg④等多种软件上发现了重要的安全漏洞. CGF 的提出最早可以追溯至 2007 年的 EFS[14],EFS 利用调试器在被测程序运行过程中追踪执行信息以获取覆盖率,并结合进化算法与各种启发式变异方式生成子代测试用例以制导模糊测试达到更高的覆盖率. 2013 年,CGF 领域出现重要进展,来自谷歌的 Zalewski 发布了 AFL 并将其进行了开源. AFL 由于其部署简单、运行效率高、自动化程度高等优点受到了软件工程和安全领域研究人员的广泛关注. 如今,各种优秀的 CGF 工具层出不穷,各种基于 AFL 的模糊测试工作也推动了 CGF 领域的发展.

自 2013 年至今,关于覆盖率制导的模糊测试工作在逐年增加. CGF 的应用范围也在不断扩展,例如应用于 JS 引擎[41,42]、二进制文件测试[43,44]、区块链智能合约[45]、操作系统内核[46,47]等. 除 AFL 外,还诞生了许多易用且高效的模糊测试工具,如 libfuzzer、Honggfuzz、Syzkaller 等. 为了更好地对 CGF 的发展情况进行研究,我们对 CGF 的已有工作进行了系统梳理,将 CGF 的关键步骤分为 4 步:预处理、测试用例选择、测试用例演化和测试用评估,并提出了如图 2 所示的研究框架.

图 2　覆盖率制导的灰盒模糊测试框架图

由于 AFL 及各种基于 AFL 的 CGF 方法被广泛地应用,因此本文以 AFL 的模糊测试过程为例来描述 CGF 的过程. 如算法 1 所示,AFL 的模糊测试过程可分为四步,下面将依次进行介绍.

第一步,AFL 对被测项目进行预处理,使用一种轻量级的方式对被测项目进行程序插桩,在插桩过程中每个基本块的起始位置会被插入一些指令,以便在测试用例执行过程中收集其边覆盖,插桩完成后会生成已插桩被测程序 *PUT*(第 1 行). 然后,

① https://www.ise.io/wp-content/uploads/2019/11/cmiller_cansecwest2008.pdf
② https://opencv.org/
③ https://www.mysql.com/
④ https://ffmpeg.org/

将初始种子测试用例加入种子队列 q(第 2 行),并对测试报告进行初始化(第 3 行).预处理结束后,将进入模糊测试的核心流程(第 4～18 行),部分研究工作中,例如 Rawat 等人[48]、Li 等人[49]、Xu 等人[50]也称该流程为模糊测试循环(Fuzzing Loop).

第二步,AFL 从存储种子测试用例的队列中根据各种子的覆盖信息、运行时间等因素选择一个种子 s(第 5 行),s 将会作为父种子以生成子代测试用例.

第三步,AFL 对 s 进行打分,并根据 s 得分的高低确定其生成的子代测试用例的数量,得分越高的种子测试用例将会生成越多的子代测试用例(第 6-8 行).

算法 1. 覆盖率制导的灰盒模糊测试.

```
输入:被测项目 proj, 初始种子 init_seeds
输出:测试报告 report
1  PUT:=instrument(proj);
2  q:=init_seeds;
3  report.init();
4  WHILE 停止条件未满足 DO
5    s:=chooseOneFrom(q);
6    score:=calculateScore(s);
7    FOR i from 1 to score DO
8      s':=mutate(s)
9      feedback:=execute(s', PUT);
10     IF feedback==CRASH THEN
11       add s' to S_C;
12     ELSE IF feedback==INTERESTING THEN
13       add s' to q;
14     END IF
15     report.update();
16   END FOR
17 END WHILE
18 RETURN report;
```

第四步,AFL 将新生成的子代测试用例 s' 发送至被测程序运行,收集 s' 的运行信息,存储至 $feedback$(第 9 行);如果 s' 在运行过程中触发了被测程序的崩溃,则将其加入至存储触发程序崩溃的测试用例集合 S_C(第 10～11 行),如果 s' 没有使被测程序崩溃并且还覆盖了新的边,则将其加入种子队列 q 作为新的种子测试用例(第 12～13 行).子代测试用例执行完毕后,模糊测试的运行信息被更新至 $report$(第 15 行).

最后,当停止条件未满足时,AFL 会反复执行第二步至第四步,持续对测试用例种群进行演化,不断将可以覆盖到新边的子代测试用例加入种群,并对种群内的种子进行变异以尽可能地覆盖到更多的边来提高覆盖率.

尽管像 AFL 这种利用覆盖反馈来制导模糊测试的工具在测试数据库[51,52]、操作系统内核[53,54]、虚拟机监视器(Hypervisor)[55,56]等不同类型软件上取得了成功,但也存在一些限制.例如,AFL 在插桩过程中对被测项目的所有基本块进行插桩,这样可能会导致冗余的插桩并在测试过程中因为追踪测试用例的执行轨迹而产生过多的开销,Nagy 等人[57]提出通过仅追踪增加覆盖率的测试用例来减少程序插桩产生的开销,Hsu 等人[58]提出在不影响覆盖反馈效果的情况下减少插桩基本块的数量,提高 AFL 在测试过程中的吞吐量;又如,AFL 所使用的随机变异方式难以通过硬比较(hard comparisons,例如魔法字节),Liang 等人[24]、Chen 等人[25]提出通过动态污点分析生成高质量种子测试用例并同步至 AFL 的种子测试用例队列以通过难以满足的约束;再如,AFL 中对种子测试用例进行随机变异有较高的不确定性,可能会导致 AFL 将大量资源消耗在非关键字节上,Liang 等人[24]提出通过使用污点分析来判断种子测试用例的关键字节、Aschermann 等人[59]和 Gan 等人[60]提出通过变异推断字节与变量之间的关系,使 AFL 有更大的概率对关键字节进行变异,以探索被测程序中的稀有分支.

3.2 预处理

CGF 通常需要在测试开始前对被测程序进行预处理,使模糊测试工具在测试过程中收集种子测试用例的覆盖信息,并利用覆盖信息制导子代测试用例生成,以提高被测程序的代码覆盖率.以被广泛应用的 CGF 工具 AFL 为例,AFL 在测试开始前会对被测程序进行预处理,执行程序插桩,向被测程序中插入一些额外指令.在测试过程中,当测试用例执行了 AFL 插入的指令时,共享内存中会记录测试用例的覆盖信息,以便 AFL 制导生成子代测试用例.

然而,目前应用广泛的 CGF 工具通常不会对 PUT 进行静态分析,可能会导致测试资源利用率较低,而插桩则可能会产生较多的运行开销,从而降低 CGF 的测试效率.针对这些问题,现有工作主要通过增加静态分析或优化插桩算法这两种方式来改进预处理阶段的工作,以为后续模糊测试过程提供帮助,例如在预处理过程中利用静态分析技术从 PUT 中收集必要信息,以引导 CGF 在相同的测试时间内达到更高的覆盖率,或根据 PUT 的源码是否可获取来有针对性地改进程序插桩算法,以提高测试吞

吐量.

3.2.1　静态分析

已有研究工作通过添加对 PUT 进行静态分析,根据所收集的程序信息来指导预处理或模糊测试过程,例如 AFLTeam[61]通过静态分析提取 PUT 的函数调用图,以提升 CGF 的代码覆盖率和挖掘潜在漏洞的能力;*K*-Scheduler[62]通过静态分析提取 PUT 的控制流图,并基于种子在控制流的边覆盖信息为其分配能量,以提升边覆盖率.与在测试过程中对 PUT 进行动态分析相比,在测试开始前进行静态分析的好处是 CGF 工具在测试过程中不再需要消耗计算资源对 PUT 进行动态分析,从而将计算资源集中在对 PUT 进行测试上,并且可以利用静态分析结果制导模糊测试,在不降低测试效率的同时提高测试的针对性.

根据使用静态分析结果的阶段不同,已有工作可分为两类:一类是将静态分析结果用于测试开始前的预处理阶段,另一类是将静态分析结果用于测试过程中的测试用例选择、测试用例演化或测试用例评估阶段.

部分工作将静态分析结果用于测试开始前的预处理阶段,可利用静态分析结果指导初始种子测试用例生成,或改进插桩技术等.例如:Brennan 等人[63]利用静态分析技术识别秘密依赖分支(Secret-Dependent Branch Detection),辅助生成高质量的初始种子测试用例,从而更高效地挖掘 Java 程序中存在的侧信道(side-channel)漏洞;FuzzGen[64]通过生成被测项目的抽象 API(Application Programming Interface)依赖图(Abstract API Dependence Graph,A^2DG),并利用 A^2DG 生成模糊测试驱动程序,以更高效地挖掘库中隐藏较深的漏洞;InstruGuard[65]通过静态分析技术检测被测程序中的插桩错误,并指导使用二进制重写技术修复错误,以提升 CGF 的测试效率,挖掘更多潜在的漏洞;RIFF[66]执行控制流分析和过程间分析来预计算所有可能的控制流边,并利用分析结果简化插桩,减少测试过程中因收集覆盖信息而产生的开销.

另一部分工作将静态分析结果应用于制导模糊测试过程.与在测试过程中对 PUT 进行动态分析相比,这种方法虽然在分析准确度上略有损失,但可以在提高测试针对性的同时保持测试效率.当一个 PUT 需要被反复进行测试时,这种因对程序分析而产生的开销可明显减少.从已有工作来看,静态分析结果可以用于指导测试过程中的测试用例选择、测试用例演化和测试用例评估中的任意一个或多个阶段.例如:Cerebro[29]利用静态分析技术计算每个函数的复杂度分数,在模糊测试过程中基于静态分析结果计算种子的得分,并优先选择具有较高得分的种子进行变异,得分越高的种子会被分配更多的能量以生成更多的子代测试用例,从而在有限的测试时间内覆盖程序中更多边,挖掘更多漏洞;PATA[24]利用静态分析构建代表变量序列(Representative Variable Sequence),在模糊测试过程中会基于代表变量序列确定种子的关键字节,并对关键字节进行变异以覆盖程序的更多路径;DIE[41]、Token-Level AFL[67]、LawBreaker[68]等工具利用静态分析结果制导模糊测试生成语义性更强、结构化程度更高的子代测试用例,从而有效地探索 PUT 的深层代码;Razzer[30]则是根据静态分析获得的潜在数据竞争位置,制导模糊测试在执行过程中挖掘数据竞争漏洞.

静态分析通常会在处理阶段仅运行一次,所产生的时间开销会受到多种因素的影响,例如 PUT 的规模、复杂度等.根据文献[69-71]的内容可知,若 PUT 规模较小,静态分析所产生的时间可能仅需几分钟或几秒钟;若 PUT 规模较大,静态分析的时间可能会需要几个小时.然而,在验证 CGF 方法有效性时,模糊测试的运行时间通常设置为 24 小时,且会重复运行实验多次以减少随机性所带来的影响,模糊测试所产生的时间开销远远多于静态分析所产生的时间开销.因此,静态分析所产生的时间开销对 CGF 整体性能的影响较小,相关研究通常未对静态分析的时间开销进行统计.在研究过程中,可根据 PUT 的规模与复杂度、静态分析是否会带来正面影响等因素来综合考虑是否要使用静态分析来提升 CGF 的测试效果.

3.2.2　程序插桩

程序插桩是 CGF 中为模糊测试提供反馈制导的一个重要步骤,其目的是通过向被测程序中插入一些额外指令,为 CGF 提供反馈,进而指导 CGF 覆盖更多的路径.现有工作中,通常是对所有基本块进行程序插桩,以达到完整的覆盖反馈.然而,对所有基本块进行插桩可能会导致测试效率较低等问题,因此,CGF 的插桩阶段有较大改进空间,例如 Chen 等人[72]和 Dinesh 等人[73]通过优化插桩算法来提高模糊测试的效率.

在已有的 CGF 相关工作中,程序插桩通常在测试开始前,即预处理阶段进行.根据被测项目源代码是否可用,插桩可以分为基于源代码和基于二进制

文件进行插桩.

基于源代码插桩是在 PUT 源代码可用时执行的程序插桩技术,虽然通过这种方式可以有效地为 CGF 提供精确的反馈制导,然而该方法需要在可获取 PUT 源代码的情况下才能进行,难以对不开源的被测项目进行充分有效的测试.此外,基于源代码的程序插桩所产生的开销也是不可忽视的,Zhou 等人[74]的研究表明,与未经过插桩的 PUT 相比,经过插桩的 PUT 会平均增加 71.85%的运行开销,并且随着程序规模的增加,因程序插桩而产生的运行开销会越来越明显.针对此问题,Hsu 等人[58]和 Liu 等人[65]在不影响覆盖率收集的同时,通过优化插桩算法减少模糊测试过程中因程序插桩而产生的开销,提升 CGF 的测试效率,以达到在相同测试时间内覆盖更多路径、挖掘到更多安全漏洞的目的.

基于二进制文件进行插桩则是在没有源码的情况下,对 PUT 的可执行文件进行插桩以制导模糊测试.例如 Honggfuzz 可利用 QEMU[75]对无源码的可执行文件进行动态插桩并收集覆盖率制导模糊测试,WinAFL 使用 DynamoRIO① 插桩闭源程序收集代码覆盖率信息并制导模糊测试.此外,Gao 等人[44]还使用二进制重写技术对 PUT 进行重写,将插桩语句插入至 PUT 中,实现与基于源代码插桩同样的效果.与基于源代码的插桩相比,基于二进制文件的插桩虽然对 PUT 源代码的需求有所减少,但也会导致 CGF 的测试效率进一步降低.已有研究[76]表明,基于二进制的插桩与基于源代码的插桩相比会导致模糊测试的运行开销增加 0.25-100 倍,这会大幅降低 CGF 的测试效率.因此,Dinesh 等人[73]、Nagy 等人[76]在此基础上对二进制重写技术进行优化,使得在无源码情况下插桩获得的 PUT 可以达到与在源代码上插桩获得的 PUT 具有相似的吞吐量.

除基于源代码的插桩与基于二进制文件的插桩外,部分研究工作[46,77,78]还会在不影响模糊测试工具覆盖率收集的前提下,向 PUT 中插入额外指令内容,收集更多种子测试用例执行信息,进而更好地制导或优化模糊测试流程.例如:Memlock[77]向 PUT 中插入指令来收集种子测试用例的内存消耗量,制导模糊测试生成内存消耗量更大的测试用例,并挖掘与内存消耗有关的漏洞;CollAFL[79]对 AFL 程序插桩策略进行改进,以解决模糊测试在收集覆盖信息过程中产生的哈希冲突,获得更准确的覆盖信息;datAFLow[80]则是识别 PUT 中对变量进行定义或使用的点,并对这些指令进行插桩,跟踪测试用例的数据流覆盖,进而制导模糊测试获得更高的覆盖率.

3.2.3 其他技术

除了在静态分析和插桩方面做出改进外,已有的研究工作还在其他方面对预处理阶段进行了优化,例如 HWFP[81]硬件翻译成可以被编译、执行且功能相当的软件模型,并设计了相应语法规则来生成高质量初始种子测试用例,以实现对硬件进行高效的模糊测试;FuzzGen[64]、RULF[82]等工具通过生成模糊测试驱动程序来提高 CGF 的测试针对性.本节对最常用的两种方案进行介绍,即生成高质量初始种子测试用例[31,32,56,83]和生成模糊测试驱动程序[64,84-86].

初始种子测试用例对 CGF 的影响很大,一个高质量的初始种子测试用例会帮助 CGF 突破分支约束,探索程序中的更多路径,相反,一个低质量的初始种子测试用例则会使 CGF 耗费较多资源在通过分支条件上,导致 CGF 的测试效率变低.因此,为 CGF 提供一个高质量的初始种子测试用例是极为关键的步骤,人工设计初始种子测试用例会造成较多的人力开销,因为这需要经验丰富的工程师在熟悉 PUT 逻辑结构的情况下构建.针对此问题,研究人员尝试通过自动生成高质量的初始种子测试用例来提高 CGF 效率,例如:JFS[83]先根据 PUT 中自由变量的数据类型构建存储特殊值(如 NaN)和特定类型常数值的集合,然后根据集合中的内容进行排列组合,并随机抽取一定数量的种子作为初始种子,进一步提高模糊测试的覆盖率;HotFuzz[31]则利用标识值实例化(Identity Value Instantiation)和小型递归实例化(Small Recursive Instantiation)为模糊测试生成合适的种子输入,以高效地挖掘 Java 库中存在的算法复杂性漏洞.

在软件开发过程中,对整个 PUT 进行测试的效率是较低的,因此开发人员通常更希望对某一个函数进行针对性的测试,其中一种解决方案是手动构建模糊测试驱动程序.例如 AFL 的 persistent mode 可通过测试人员手动编写驱动程序,以对指定函数进行测试.然而,手动编写模糊测试驱动程序需要测试人员熟悉 PUT 源代码的内部逻辑,这既会导致较多的人力开销,也容易使测试人员在构建模糊测试驱动程序的过程中遇到困难.为此,部分工

① https://github.com/DynamoRIO/dynamorio

作[64,84-86]通过自动构建模糊测试驱动程序来提高模糊测试的针对性和有效性,例如:FuzzGen[64]在平衡了探索抽象 API 依赖图深度及广度的基础上合成模糊测试驱动程序,以有效地对特定环境中较为复杂的库函数进行模糊测试; IntelliGen[84]则首先确定一组入口函数,评估它们出现漏洞的概率,然后通过层次化参数替换和类型推理为指定函数生成模糊测试驱动程序,从而减少人工合成模糊测试驱动程序所产生的开销,提升自动合成的驱动程序的质量.

3.2.4　小　结

CGF 会在模糊测试开始前对被测项目进行预处理,例如对被测项目进行静态分析、程序插桩、生成初始种子测试用例等工作,以指导模糊测试覆盖更多的程序路径或探索更多的程序状态.然而,目前的 CGF 工作中在预处理阶段仍存在部分局限性,具体如下.

静态分析可以为 CGF 带来提高测试效率、覆盖更多路径等效果,但是静态分析可能会产生误报或漏报,这会导致静态分析无法准确地理解程序的行为与逻辑,进而无法向 CGF 提供有效的制导信息.此外,随着程序规模和复杂度的增加,静态分析所需的时间和空间也会增加,甚至可能会超出可以接受的范围,使 CGF 在预处理阶段产生大量的时间开销.

程序插桩可以为 CGF 提供覆盖信息反馈以及详细的测试用例执行信息,但是程序插桩通常会向被测程序中插入部分指令以收集运行信息.当需要收集更多的执行信息时,插入的指令数量也会增多,进而增加程序运行的开销,导致程序的运行效率下降.除降低被测程序运行效率外,若插入的指令改变了被测程序的行为或逻辑,可能会引入新的错误或隐藏程序中原有的错误,导致 CGF 的测试结果不可靠.

自动生成初始种子测试用例虽然可以使 CGF 更快地探索被测程序中的深层区域,但是当被测程序对输入的格式要求较高时,可能需要花费大量的时间生成初始种子测试用例,进而增加预处理期间的时间开销.自动构建模糊测试驱动程序虽可以为测试人员减少手动构建驱动程序减少的开销,但是目前的研究工作未考虑不同被测程序的特点,难以为不同类型的被测程序生成模糊测试驱动程序.

3.3　测试用例选择

CGF 通常会将所有种子测试用例存储在种子池中,种子池中所有种子的覆盖互不相同.在已有的 CGF 工作中,种子池通常会被定义为一个队列,队列中的每一个节点存储一个种子.在每次迭代开始前,模糊测试工具首先会根据种子队列里各种子的运行信息、覆盖信息等因素确定哪些种子是高质量种子.然后,依次选择队列中的种子进行变异,如果一个种子是高质量种子,那么它就会有较大的概率被选择进行变异,否则,会有较小的概率被选择进行变异.当队列中的种子遍历完毕后,模糊测试工具再次根据队列的内容确定哪些种子是高质量种子,并依次选择队列内存储的种子,开始下一轮迭代.

根据测试目的不同,测试用例选择策略也会存在一定差异,不同的选择策略会使模糊测试产生不同测试效果.CGF 通常会判断种子是否有较高的概率生成可覆盖新边的子代测试用例,并基于判断结果决定是否选择该种子进行测试用例演化,以达到对 PUT 进行充分测试的目的,因此 CGF 会覆盖 PUT 中较多的路径.DGF 则是在模糊测试过程中尝试尽快覆盖指定目标点,部分 DGF 研究工作[87,88]会根据种子与目标点之间的距离决定是否选取该种子进行演化,以达到对指定目标点进行充分测试或快速复现指定漏洞的目的.因此,虽然 DGF 不一定能比 CGF 覆盖 PUT 中更多的路径,但是会比 CGF 具有更强的针对性,可对 PUT 中的指定区域进行充分测试.

然而,在模糊测试过程中,测试时间通常有限,若高质量种子未得到及时变异或在质量较低的种子上耗费过多的测试资源,会降低模糊测试效率.因此,为最大限度发挥模糊测试能力,已有工作通常会使模糊测试选择更高质量的种子进行变异,因为高质量的种子有更大的概率生成高质量的子代测试用例,从而可以探索 PUT 的更多状态.目前常用的对测试用例选择步骤进行改进的方法主要有改进种子池的存储结构和优化种子的选择策略两类.

3.3.1　种子池存储结构改进

将种子池定义为线性数据结构可以在时间开销较小的情况下选择一个种子,然而这种选择方式存在一定局限性,例如通常只能按照存储顺序从种子池中选择种子,当质量高的种子被存储在队尾时,会导致其无法及时地进行变异;当种子队列的规模过大时,可能还会导致没有足够时间对高质量种子进行变异,从而对 CGF 产生负面影响.针对这些问题,V-Shuttle[89]、SLIME[23]将种子池的存储结构进行改进,从单队列改进为多队列,并根据模糊测试的执

行情况从合适的队列中选择种子进行变异，以提高CGF的效率. AFL-Hier/AFL++-Hier[90]则是通过更改种子池的存储结构，将队列更改为树，并根据多臂老虎机算法选择存储种子的叶子节点以达到更高的覆盖率.

3.3.2 种子选择策略

当种子池中存储了大量种子时，选取每一个种子并进行变异是非常耗时的一个过程，因此，CGF工具通常会从种子池中选取质量较高的部分种子组成一个子集，并优先对子集中的种子进行变异. 例如AFL会根据种子的运行时间、字节数量和覆盖信息筛选高质量种子，Skyfire[91]通过分析基本块覆盖情况从种子池中挑选高质量的种子. 具体来说，AFL会遍历每一个已被覆盖的边，并在覆盖了该边的所有种子中将运行速度快、字节数量少的种子标记为偏好种子. 在测试过程中，AFL有更高的概率选择偏好种子进行变异，因为这样的种子可以保持CGF的吞吐量. 然而已有研究工作中的种子选择策略比较经验化，已有研究表明运行速度快或字节数量少的种子并非总能在不同的PUT上达到理想效果[23]因此，部分工作会改进种子选择策略以提升CGF的效果，例如Memlock[77]会根据种子的覆盖率及内存消耗确定偏好种子，挖掘PUT中与内存消耗量有关的漏洞；Truzz[92]会优先选择产生更多新边覆盖的种子，以提高PUT的代码覆盖率；K-Scheduler[62]则是通过计算每个种子的中心性为其赋予不同的被选择概率，并优先选择高质量的种子进行变异，以覆盖PUT中更多的边.

3.3.3 小 结

在CGF的迭代过程中，CGF会从种子池中选择一个种子测试用例并对其进行变异、交叉等操作以生成子代测试用例. 在测试时间有限的情况下，为了提高测试效率，通常希望从种子池中选择质量更高的种子测试用例，因为对它进行变异更有可能生成高质量的子代测试用例，从而对PUT的程序空间进行充分探索.

虽然改进种子池的数据结构可提高种子池的读写性能、空间利用率、索引能力等，但将种子池的结构改为哈希表、树等数据结构后，选择种子时会增加一定的开销和限制，可能会降低CGF的稳定性和可扩展性，或增加CGF的时间复杂度、空间复杂度，从而降低CGF的测试效率. 此外，优化后的种子池结构需要与模糊测试引擎、被测程序进行适配和协调，可能会导致模糊测试的兼容性和通用性降低.

现有研究工作中的种子选择策略通常是基于启发式设计的，难以在不同的应用场景下均能有效地从种子池中选择多样性高的子集进行变异. 此外，种子选择策略缺乏对测试用例演化过程中产生的变异体进行动态反馈，即在变异过程中根据变异体的运行情况动态地调整种子选择策略，可能会将过多的测试资源分配给质量较低的种子测试用例.

3.4 测试用例演化

测试用例演化是CGF过程中探索PUT状态及挖掘潜在漏洞的一个重要阶段. 在测试用例演化过程中，首先，根据运行情况与覆盖信息对种子测试用例进行评分，质量越高的种子测试用例会得到越高的评分；然后，CGF工具会根据评分对种子测试用例使用各种变异方式，生成相应数量的子代测试用例. 然而，CGF工具中的评分标准通常较为经验化，可能会导致测试资源分配不合理，且对种子进行随机变异会使CGF难以通过诸如魔法字节之类的分支约束条件. 针对这些问题，现有工作通过改进评分标准来提高模糊测试的覆盖率，或通过添加、修改测试用例生成策略来提高CGF通过约束条件的概率，提高对PUT的覆盖率.

3.4.1 能量分配

在大多数CGF工作中，测试用例的演化会采用遗传算法. 具体来说，CGF会维护一个测试用例种群，种群中的每一个测试用例种子都有其自己的适应度. 当选定种群中的一个种子后，会根据所选种子的适应度值为其分配相应的测试资源. 在CGF中通常将测试资源称为能量(Energy). 当种子被分配好能量后，遗传算法就会将其进行变异或交叉，以生成子代测试用例. 一个种子测试用例被分配到的能量越多，那么它生成的子代测试用例也越多，反之，一个种子测试用例被分配到的能量越少，那么它生成的子代测试用例数量也越少.

已有工作中的能量分配策略通常基于启发式设计，例如AFL会为边覆盖数量多且执行速度快的种子分配更多的能量，libfuzzer会将更多的能量分配给在模糊测试过程中较晚保存的种子. 然而基于启发式设计的能量分配策略并非在任何情况下都能达到理想效果，可能会不合理地为种子测试用例分配能量. 因此，部分工作通过对能量分配策略进行改进以帮助CGF在相同的测试时间内达到更高的覆盖率. 例如：Cerebro[29]会根据种子执行轨迹的复杂度和执行轨迹附近未覆盖代码的复杂度计算种子综合得分，并根据综合得分将更多的能量分配给可能触

发错误或覆盖新边的种子;AFLSmart[93]则会计算种子的有效性比率,有效性程度高的种子会比有效性程度低的种子分配到更多的能量,从而有更多的机会产生有效的子代测试用例.

此外,已有研究表明 AFL 的能量分配策略会导致模糊测试花费较多的测试资源在高频路径上[94],即生成的子代测试用例大部分会执行高频路径而难以执行低频路径,进而使模糊测试陷入一种局部最优状态.针对这个问题,EcoFuzz[10]提出一种自适应的能量分配策略,该策略可通过评估先前的能量分配策略来调整下次为该种子分配的能量,从而达到有效利用资源,在有限的执行时间内实现路径覆盖最大化的目的;Entropic[95]提出一种基于熵的能量分配算法,利用香农熵[96]量化每个测试用例的质量,并将大部分能量分配给更有可能提高覆盖率的种子,以提高 CGF 对 PUT 的代码覆盖率及错误发现能力.

3.4.2　测试用例生成

子代测试用例的质量决定了是否能覆盖新路径或是否能触发新崩溃,因此子代测试用例生成是 CGF 流程中最重要的步骤之一.根据子代测试用例生成过程的不同,CGF 可以分为基于变异的模糊测试和基于生成的模糊测试.

基于变异的模糊测试设计思路很直接,通过扰乱测试用例中的字节内容达到随机变异的目的,以挖掘被测程序中的漏洞.但这种方法由于具有较高的随机性,可能会导致子代测试用例的有效性较低.假设有一条简单的判定语句:$if\ (input==23)$,如果输入是一个 32 bit 的整数,随机变异输入变量 *input* 以满足该约束条件的概率是 $1/2^{32}$,当输入是一个 64 bit 整数时,随机变异 *input* 以满足该约束条件的概率为 $1/2^{64}$.此外,当 PUT 对输入具有较高的结构要求时,例如要求输入的数据为图片文件或者音频文件时,通过随机变异生成有效测试用例的概率会更低.因为生成的测试用例很有可能在探索 PUT 更深层次的区域前就因为语义无效而被 PUT 拒绝解析,从而导致模糊测试陷入局部最优状态.针对此问题,部分工作通过在变异策略方面提出创新,使生成的子代测试用例具有更高的有效性,例如 Angora[25]跟踪字节流入的路径约束,重点对这些字节进行变异,并使用梯度下降算法解决路径约束;GreyOne[60]则通过监控对种子测试用例进行变异后 PUT 中变量值的变化情况,来推断 PUT 中的污点信息,并利用污点信息制导 CGF 生成更有效的测试用例.

基于生成的模糊测试(也称为基于语义的模糊测试)则是根据某个模板生成有效性较高的测试用例,通常会用于测试需要高度结构化输入的程序,例如 JS 引擎、编译器等.通过这种方式生成的测试用例会更有可能通过 PUT 的输入解析阶段,以探索 PUT 的核心功能.与基于变异的模糊测试相比,基于生成的模糊测试通常不需要种子输入,因为模糊测试工具会根据所提供的模板生成结构化的测试用例.然而这种方法也有一定的局限性,例如所生成的测试用例尽管在语法上有效,但在语义上可能无效,假设测试用例是一个语法上正确的 C 语言文件,如果文件中用到了没有定义的变量,那么也会导致 PUT 拒绝对测试用例进行解析.另外,通常需要有经验的测试工程师或开发工程师手动编写模板,这也会造成较大的人力开销.针对这些问题,MoFuzz[97]通过结合基于图语法的模糊测试和基于变异的覆盖制导模糊测试将 CGF 应用于测试模型驱动软件工程(Model-Driven Software Engineering,MDSE)工具 EMF2GraphViz①、UML2OWL② 等,基于图语法的模糊测试可以尽可能快地覆盖输入域的不同方面,而基于变异的覆盖制导模糊测试可对测试用例进行演化,以探索 MDSE 工具中更深层次的路径;DIE[41]通过分析 JS 格式的种子文件,生成与其相对应的抽象语法树(Abstract Syntax Tree,AST),并在测试过程中通过在 AST 中插入或替换节点来生成新的测试用例,以保护测试用例在变异过程中的结构不被破坏.

3.4.3　小　结

在测试用例演化阶段,CGF 会根据种子的质量为其分配相应的能量,质量越高的种子会被分配越多的能量,生成更多的子代测试用例,并发送至 PUT 以挖掘其中潜在的漏洞.

能量分配在已有的 CGF 研究工作中具有多种方法和策略,但是已有的能量分配策略关注的种子测试用例执行信息较少,这可能会导致高质量的种子测试用例被忽略,而质量低的种子测试用例被过度关注.若关注过多的执行信息会导致 CGF 在能量分配阶段产生过多的开销,从而降低 CGF 的测试效

① http://emftools.tuxfamily.org/wiki/doku.php?id=emf2gv:start

② https://www.eclipse.org/atl/atlTransformations/#UML2OWL

率.因此,未来工作需要根据不同的测试场景收集适量的运行信息,在保持 CGF 测试效率的同时为种子测试用例分配更合理的能量.此外,合适的能量分配策略通常需要调整多个参数,难以设置一个通用的参数以适应不同的测试场景或被测项目.

针对测试用例生成存在的局限性,目前相关研究工作已提出并实现了相应的解决方案.然而目前仍然存在一些不足,例如在基于变异的模糊测试中,收集更多测试用例的执行信息并确定其关键字节会导致测试用例执行速度变慢;在基于生成的模糊测试中,需要耗费较高的成本来编写模板,并且仍然可能会生成较低质量的测试用例,导致模糊测试效率变低,遗漏潜在的安全漏洞.

3.5 测试用例评估

当子代测试用例生成后,CGF 工具会将子代测试用例发送至被测程序,并查看子代测试用例的运行情况,以决定将其保留或者丢弃.输入评估阶段可分为两步:测试用例执行和反馈分析.部分 CGF 工具中的测试用例执行方式可能难以应用于任何场景,且反馈分析方法较为经验化.针对这些问题,现有工作通过改变测试用例的运行方式以提升在不同场景下 CGF 的吞吐量,或通过改进反馈情况的检查标准,将符合标准的子代测试用例加入种群以提高种群中种子的整体质量,进而提升 CGF 的测试覆盖率和漏洞挖掘能力.

3.5.1 测试用例执行

生成的测试用例会用于驱动 PUT 执行,以挖掘被测程序中存在的漏洞.在早期工作中,模糊测试工具运行子代测试用例的方式较为简单,即每运行一个测试用例都调用一次 *execve*()[①]函数来将测试用例发送至 PUT.这种方式在当时受到了研究人员的青睐,因为它不需要熟悉 PUT 的内部逻辑或说明文档,简单地将测试用例发送至 PUT 执行即可.此外,调用 *execve*()函数后等同于让 PUT 在一个单独的进程中运行,并且每发送一个测试用例前都会重新启动并初始化 PUT,通过这种方式就不必担心 PUT 的行为会影响到模糊测试工具本身的状态.然而,这样的执行方式存在一定的局限性,例如模糊测试过程中会消耗较多时间在 *execve*()函数、链接器和库的初始化中,降低测试效率.

针对这个问题,AFL 所采用的解决方案是使用“forkserver”,通过向 PUT 中注入一段代码,使 PUT 的运行状态停留在入口处.当需要运行测试用例时,AFL 会复制(fork)一份相同的进程来执行测试用例.通过这种方式,CGF 的吞吐量可以提升 1.5 至 2 倍左右.当然这种方法也存在使用场景有限等问题,通常仅支持在 Linux 平台上实现.Winnie[85]针对此问题在 WinAFL[②] 中实现了 fork 功能来提高 CGF 在 Windows 平台上进行测试的效率.除 forkserver 外,部分研究工作还利用了其他技术来提高测试效率,例如 UnicoreFuzz[98] 通过使用 UnicoreEngine 来提高对操作系统内核进行测试时的测试用例执行效率,Nyx[99] 则利用 hypercall 将测试用例发送至虚拟机监视器,实现对虚拟机监视器的高效模糊测试.

3.5.2 反馈结果分析

在执行完新生成的测试用例后,CGF 工具需要对测试用例的运行结果进行分析,并根据反馈分析结果制导模糊测试.在 CGF 的运行过程中,如果新测试用例触发了 PUT 崩溃、导致 PUT 运行时间超时、或者可以提升 PUT 的整体覆盖率,对应测试用例将会被保存,否则,对应测试用例将会被丢弃.

大多数工作通常会基于控制流边覆盖情况对 CGF 进行反馈制导,因为通过这种方式会比基于基本块覆盖收集到更多的信息.AFL 利用遗传算法和控制流边覆盖制导模糊测试,并且会将覆盖到新控制流边的测试用例保存至队列或者本地,其判断测试用例是否覆盖到新的边借助了位图(bitmap).具体来说,AFL 利用 3 段长度为 64 KB 的位图来存储触发程序崩溃的测试用例、超时测试用例和种子测试用例的覆盖情况,位图中每一个元素表示一个控制流边的覆盖情况.以种子测试用例为例,判断一条测试用例是否应当被加入队列以保存为新种子时,AFL 会首先获取它的覆盖信息,并将覆盖信息与种子覆盖位图进行比较,如果该测试用例覆盖到了之前未被覆盖过的边或已覆盖边的执行次数触发了不同的计数桶[③],则将其加入种子队列中.这样的种子被 AFL 称为执行了独特路径(unique path).同理,对于触发程序崩溃的测试用例和使 PUT 超时的测试用例,也会分别将其覆盖信息与崩溃覆盖位图和超时覆盖位图进行比较,当测试用例覆盖了新边或命中了新的计数桶,则它们会分别被保存下来,这样的测试用例在 AFL 中分别被称为触发了独特崩溃

① *execve*()函数可用于在 Linux 环境下执行指定的可执行二进制文件

② https://github.com/googleprojectzero/winafl

③ AFL 对边的执行次数按照 9 个不同的计数桶进行统计,即 0、1、2、3、4-7、8-15、16-31、32-127、128-255

(unique crash)和导致独特超时(unique hang).

尽管 AFL 所采用的覆盖反馈制导方法在各项研究中已取得了较好的效果,但其仍存在可提升空间.例如 Memlock[77]针对 PUT 的覆盖和内存消耗情况制导模糊测试,除了会将覆盖到新边的子代测试用例加入种子队列外,还会将具有更大内存消耗的子代测试用例替换种子测试用例,以挖掘更多与内存消耗量有关的漏洞;DIE[41]针对 JS 引擎的特点,舍弃了覆盖位图中各边的命中计数,改为判断某一边是否被覆盖,因此 DIE 可以存储的覆盖信息是 AFL 的 8 倍;MTFuzz[16]则是对变异的输入进行采样,将所收集的信息用于重新训练模型,以制导 CGF 达到更高的覆盖率.

3.5.3　小　结

在测试用例评估阶段,CGF 会将子代测试用例发送至 PUT 运行,并根据子代测试用例的运行结果决定是否将其保留.

尽管目前 CGF 研究工作中的测试用例执行已发展得较为成熟,但是仍存在部分因素可能会限制其稳定性,例如若测试用例执行过程中出现环境变化、系统重启等情况,那么模糊测试就会受到影响,进而无法保证测试用例的可重复性.

在反馈结果分析过程中,根据应用场景的不同,CGF 会需要处理大量的数据.若这些数据的解析过程较为复杂,则需要消耗大量的时间与资源,影响 CGF 的效率.此外,反馈结果分析还需要适应不同的系统和场景,例如不同的操作系统、编程语言等,这些系统和场景会有不同的特点,需要不同的分析方法.如果处理反馈结果的方法缺乏灵活性,会导致 CGF 方法难以适应多变的需求,进而影响其通用性.

3.6　总　结

表 2 是我们对所收集到的 CGF 工作进行的总结,并已按照发表年份顺序进行升序排序,每项工作根据所划分的 4 个阶段进行总结.此外,还有一部分用于对相关工作进行描述的其他信息,例如该项工作是否开源、是否支持在无源码的情况下进行模糊测试等.在表中,第 1 列是所描述工作的名称;第 2 至 4 列为该项工作在预处理阶段的哪些方面做出改进,例如使用静态分析辅助模糊测试计算种子测试用例的能量、优化插桩算法以提升 PUT 的运行效率、自动生成模糊测试驱动程序以高效地挖掘漏洞等;第 5 至 6 列为该项工作是否在测试用例选择方面做出改进,例如使用非线性结构存储种子以高效地选择种子进行变异、优化种子选择策略以在有限的测试时间内更多地选择高质量种子等;第 7 至 8 列为该项工作是否在测试用例演化阶段的能量分配或测试用例生成方面做出改进,例如基于启发式设计能量分配策略以合理地分配测试资源、识别种子中的关键字节以指导变异体生成等;第 9 至 10 列为该项工作是否在测试用例评估阶段的测试用例执行和反馈分析方面做出改进,例如优化测试用例执行方式以提高测试效率、收集种子的更多运行信息以制导模糊测试等;第 11 至 17 列为各项工作的其他信息,包含该项工作来源、发表年份、开源情况、代码存储库的 fork 和 star 数量、是否支持无源码模糊测试、是否基于先前的某项工作和研究工作中使用了哪些基线工具.

在第 2 至 10 列中,各项研究工作在各阶段进行改进的基准为“父类工具或基线工具”列所展示的工作,例如:FairFuzz 基于 AFL 在测试用例演化阶段的测试用例生成部分进行了改进.当论文中未提及研究工作是否在先前某项工作的基础上实现时,则分析该项研究工作与其实验中所使用的基线工具相比,在何阶段进行了改进,例如:Angora 与 AFL、FUZZER、SES、VUzzer 和 Steelix 相比,在测试用例演化阶段的测试用例生成部分、测试用例评估阶段的反馈分析部分进行了改进.为便于与基于先前某项工作的研究进行区分,我们在表格中使用加粗字体表示该项研究中所使用的基线工具.

从表 2 中可以看出,我们所搜集的 CGF 研究成果中,分别有 83、87、86 项研究成果在预处理、测试用例演化和测试用例评估阶段进行了改进,而在测试用例选择阶段进行改进的工作较少,仅 30 项.在是否开源方面,将 CGF 工作开源逐渐成为一种趋势,在所调研的项工作中共有 102 项工作对所提出的方法进行了开源,占比达 64.15%.在开源的 CGF 工具中,由于可以对 PUT 进行较为充分的测试、测试效率高、支持模块自定义等原因,AFL++在所有的开源 CGF 工具中获得了最多的 fork 和 star 次数,分别为 802 次和 3944 次,其次为 Echidna,它由于具有可扩展性强、可发现隐藏较深的错误等优势,被广泛应用于对智能合约进行模糊测试,fork 和 star 次数分别为 292 和 2312.此外,Fuzzilli、Angora 等工具也由于易于使用、漏洞挖掘能力较强等原因在 CGF 领域也具有较高的影响力, fork 和 star 次数也较多.在是否支持无源码测试方面,目前能支持无源码模糊测试的工具较少,在所调研的工作中仅有 39 项工作支持, 占比 24.53%. 在所基于的已有

表 2 近 6 年的 CGF 研究成果总结

工作	预处理			测试用例选择		测试用例演化		测试用例评估		其他信息						
	静态分析	插桩	其他	种子池结构	种子选择策略	能量分配	测试用例生成	测试用例执行	反馈分析	发表源[α]	年份	开源	fork 数量/次	star 数量/次	支持无源码	父类工具或基线工具
FairFuzz[100]							√			A	2018	√	49	231		AFL
PAFL[101]					√				√	F	2018					AFLFast+FairFuzz
PerfFuzz[102]					√				√	IS	2018	√	10	128		AFL
InsTrim[58]	√									N	2018	√	5	81		AFL
Angora[25]							√		√	S	2018	√	167	876		**AFL+FUZZER+SES+VUzzer+Steelix**
MoonShine[47]					√					US	2018					Syzkaller
Matryoshka[103]							√			C	2019					Angora
Cerebro[29]	√				√	√				F	2019					FOT
FUDGE[104]	√		√							F	2019					libfuzzer
JFS[83]			√							F	2019	√	18	232		libfuzzer
SLF[105]					√					IC	2019					AFL
Superion[106]							√			IC	2019	√	20	103	√	AFL
DeepHunter[107]					√		√		√	IS	2019					**DeepTest+TensorFuzz**
Zest[108]							√		√	IS	2019	√	97	578		JQF
PeriFuzz[109]			√						√	N	2019	√	12	62		AFL
Redqueen[59]							√	√	√	N	2019	√	59	341	√	kAFL
Janus[53]							√	√		S	2019	√	26	190		AFL
NEUZZ[110]							√			S	2019	√	103	374		AFL
ProFuzzer[111]							√			S	2019				√	AFL
Razzer[30]	√							√		S	2019	√	59	344		Syzkaller
UnTracer[57]		√								S	2019	√	29	119	√	AFL
AFLSmart[93]						√				TS	2019	√	92	480	√	AFL
Agamotto[54]								√		US	2019	√	28	114		QEMU+Syzkaller+AFL
Firm-AFL[112]								√		US	2019	√	88	383	√	AFL
Grimoire[113]						√				US	2019	√	21	117	√	Redqueen
MOpt[15]							√			US	2019	√	45	184		AFL
Unicorefuzz[98]								√		UW	2019	√	35	289	√	AFL-Unicorn
BigFuzz[114]							√		√	A	2020	√	5	11		JQF
MoFuzz[97]							√			A	2020	√	2	5		JQF
Zeror[74]		√						√		A	2020					**AFL+InsTrim+UnTracer+MOpt**
FreeDom[115]			√				√			C	2020	√	25	121		**Dharma+Domato**
Squirrel[52]							√			C	2020					AFL
CrFuzz[116]								√		F	2020				√	**AFL+QSYM+MOpt**
Entropic[95]						√			√	F	2020					libfuzzer
Harvey[45]							√		√	F	2020					**Harvey 的不同配置**
MTFuzz[16]							√		√	F	2020	√	10	33		**AFL+AFLFast+FairFuzz+Angora+NEUZZ**
Ankou[117]									√	IC	2020	√	12	55		AFL
Brennan 等人[63]	√	√	√				√			IC	2020					Kelinci+AFL
Memlock[77]		√			√				√	IC	2020	√	12	61		AFL
sFuzz[118]			√		√		√			IC	2020	√	26	73		AFL

（续表）

工作	预处理			测试用例选择		测试用例演化		测试用例评估		其他信息						
	静态分析	插桩	其他	种子池结构	种子选择策略	能量分配	测试用例生成	测试用例执行	反馈分析	发表源[α]	年份	开源	fork数量/次	star数量/次	支持无源码	父类工具或基线工具
Echidna[119]	√						√		√	IS	2020	√	292	2312		**solfuzz**
lFuzzer[120]			√							IS	2020	√	4	32		pFuzzer
WEIZZ[121]							√			IS	2020	√	13	88	√	AFL+QEMU
HotFuzz[31]			√						√	N	2020					**HotFuzz 的不同配置**
TortoiseFuzz[78]		√							√	N	2020	√	17	80		AFL
DIE[41]	√						√		√	S	2020	√	41	196		AFL
IJON[122]			√							S	2020	√	37	171		AFL
Krace[46]		√			√		√		√	S	2020					QEMU
RetroWrite[73]		√								S	2020	√	71	580	√	AFL
CollAFL[79]		√			√					TD	2020				√	AFL
EcoFuzz[10]					√	√				US	2020	√	12	38	√	AFL
FIFuzz[123]	√						√		√	US	2020					**AFL+AFLFast+AFLSmart+FairFuzz**
Frankenstein[124]							√			US	2020	√	65	394	√	QEMU
FuzzGen[64]	√		√							US	2020	√	56	275		libfuzzer
GreyOne[60]					√		√		√	US	2020					AFL
Muzz[72]		√			√			√		US	2020					SVF+AFL+ClusterFuzz
USBFuzz[125]							√		√	US	2020	√	19	96		QEMU+AFL
AFL++[126]			√			√	√			UW	2020	√	802	3944	√	AFL
AFLTeam[61]	√				√			√		A	2021	√	10	68		AFL
E9AFL[44]		√								A	2021	√	21	261	√	AFL
InstruGuard[65]	√	√								A	2021	√	1	8	√	**AFL+FairFuzz+MOpt+Memlock**
RULF[82]			√							A	2021	√	5	34		**FUDGE**
VisFuzz[127]		√							√	A	2021	√	10	61		**AFL**
HexCite[43]		√							√	C	2021	√	3	18	√	ZAFL+AFL
V-Shuttle[89]	√			√						C	2021	√	16	75		AFL
SoFi[42]			√				√	√		C	2021					AFL
HeteroFuzz[128]					√		√	√	√	F	2021					AFL
bonsai-fuzzing[129]			√				√			IC	2021	√	1	24		Zest
IntelliGen[84]			√							IC	2021					**FuzzGen+FUDGE**
Luo 等人[130]							√			IC	2021					**所提方法的不同配置**
Ratel[51]							√		√	IC	2021					**SQLsmith+SQLancer+Squirrel**
Gramatron[131]			√				√			IS	2021	√	3	51		AFL++
QFuzz[132]									√	IS	2021	√	1	13		Kelinci
AFL-Hier AFL++-Hier[90]				√		√			√	N	2021	√	10	49	√	AFL/AFL++
Winnie[85]			√					√		N	2021	√	74	479	√	WinAFL
DifuzzRTL[133]							√	√	√	S	2021	√	12	58		**RFuzz**
PATA[24]	√						√			S	2021					AFL
StochFuzz[134]									√	S	2021	√	9	175	√	AFL
RIFF[66]	√	√								UA	2021					AFL
TCP-Fuzz[135]							√		√	UA	2021					**AFL+Syzkaller+boofuzz+Fuzzotron+AFLNet**
InvsCov[136]	√	√								US	2021	√	6	65		AFL++

（续表）

工作	预处理			测试用例选择		测试用例演化		测试用例评估		其他信息						
	静态分析	插桩	其他	种子池结构	种子选择策略	能量分配	测试用例生成	测试用例执行	反馈分析	发表源[α]	年份	开源	fork数量/次	star数量/次	支持无源码	父类工具或基线工具
Nyx[99]								√		US	2021	√	19	172	√	QEMU-PT+KVM-PT
SyzVegas[137]					√					US	2021	√	10	21		Syzkaller
Token-Level AFL[67]	√						√			US	2021					AFL
ZAFL[76]		√								US	2021	√	1	3	√	AFL
Chen 等人[138]	√	√				√				A	2022					AFL+AFLFast
Chen 等人[139]		√							√	A	2022					AFL
Griffin[140]							√		√	A	2022					AFL++
HTFuzz[141]		√			√		√		√	A	2022	√	2	11		AFL
LawBreaker[68]	√		√				√			A	2022					**Lawbreaker 的不同配置**
QATest[142]					√		√		√	A	2022					**MT4MRC+QAAskeR**
RLF[143]							√		√	A	2022					ILF
Jit-Picker[144]								√	√	C	2022					Fuzzilli
glibFuzzer[145]				√	√	√				F	2022					libfuzzer
Minerva[146]			√							F	2022					**Domato+Favocado+FreeDom**
SEDiff[147]	√		√						√	F	2022					AFL
BeDivFuzz[148]							√		√	IC	2022	√	2	12		Zest
FreeFuzz[149]		√					√			IC	2022	√	13	56		**LEMON+CRADLE**
GraphFuzz[150]							√		√	IC	2022	√	22	7		libfuzzer
$Havoc_{MAB}$[22]							√			IC	2022	√	1	7		Havoc
μAFL[151]								√	√	IC	2022	√	5	36	√	AFL
Muffin[152]							√		√	IC	2022	√	3	23		**LEMON**
PreFuzz[21]							√			IC	2022					NEUZZ+MTFuzz+AFL
Truzz[92]					√		√			IC	2022	√	1	2		**NEUZZ+GreyOne+AFL+AFLFast+MOpt+FuzzFactory**
EQUAFL[153]			√							IS	2022					AFL
PrIntFuzz[86]	√		√				√			IS	2022	√	9	42		**VIA+QEMU**
SLIME[23]				√					√	IS	2022	√	2	17		MOpt
SnapFuzz[154]								√		IS	2022					AFLNet
Unicorn[155]							√		√	IS	2022					**SQLsmith+SQLancer**
datAFLow[80]		√							√	N	2022	√	8	89		AFL++
Dr. Fuzz[156]			√							N	2022	√	2	21		QEMU+Syzkaller
EMS[157]							√			N	2022	√	4	31		MOpt
FirmWire[158]			√							N	2022	√	72	659	√	AFL++
MobFuzz[159]		√				√			√	N	2022					AFL
K-Scheduler[62]	√				√	√				S	2022	√	19	102		**Entropic+libfuzzer+AFL+AFLFast+FairFuzz+EcoFuzz+TortoiseFuzz**
CSI-Fuzz[160]		√								TD	2022	√	7	4	√	AFL
SNPSFuzzer[161]					√			√		TI	2022					AFLNet
CAGFuzz[162]							√		√	TS	2022					**FGSM+DeepHunter+DeepXplore+PGD+CW+ELSR+CutMix**
HashFuzz[163]			√							TS	2022					AFL+AFLFast+FairFuzz+libfuzzer
UltraFuzz[164]				√					√	TS	2022	√	0	2		AFL

（续表）

工作	预处理			测试用例选择		测试用例演化		测试用例评估		其他信息						
	静态分析	插桩	其他	种子池结构	种子选择策略	能量分配	测试用例生成	测试用例执行	反馈分析	发表源[α]	年份	开源	fork数量/次	star数量/次	支持无源码	基线工具或父类工具
HWFP[81]			√							US	2022	√	11	57		**RFuzz**
Morphuzz[55]			√				√			US	2022					QEMU
MundoFuzz[56]			√				√		√	US	2022					AFL
SGFuzz[165]						√	√		√	US	2022	√	14	57		libfuzzer
SGXFuzz[166]			√				√	√		US	2022					kAFL+Nyx
TheHuzz[32]			√				√		√	US	2022					**DifuzzRTL**
MOTIF[167]			√							A	2023					AFL++
GCMiner[168]	√		√							IC	2023	√	2	23	√	JQF
JITfuzz[169]						√	√			IC	2023	√	0	4		**Classming+JavaTailor**
▽Fuzz[170]							√		√	IC	2023	√	0	12		FreeFuzz
TypeOracle[171]			√				√		√	IC	2023	√	0	1	√	Gramatron+Favocado+Cooper
GDBFuzz[172]			√						√	IS	2023	√	7	76	√	libfuzzer
GrayC[173]							√			IS	2023	√	2	32	√	libfuzzer
HirGen[174]							√		√	IS	2023	√	0	3		**TVMfuzz+MT-DLComp+LEMON+NNSmith**
Icicle[175]		√								IS	2023	√	9	140	√	Fuzzware+AFL++
ItyFuzz[176]									√	IS	2023	√	48	385		LibAFL
TitanFuzz[177]			√		√	√	√			IS	2023	√	0	9		**FreeFuzz+DeepREL+LEMON+Muffin**
Fuzzilli[178]							√		√	N	2023	√	304	1677		Superion
FuzzNG[179]									√	N	2023	√	1	10	√	libfuzzer
SegFuzz[180]		√					√	√	√	S	2023	√	2	13		Syzkaller
UTopia[181]	√		√							S	2023	√	16	118		**OSS-Fuzz**
ViDeZZo[182]							√			S	2023	√	2	13		libfuzzer
Witcher[183]		√					√		√	S	2023	√	8	40	√	AFL
IR-Fuzz[184]	√		√			√	√			TI	2023	√	5	31		sFuzz
ConfigFuzz[185]			√							TO	2023					**AFL+AFL++**
μFUZZ[186]					√	√		√		US	2023	√	0	27		AFL++
AIFORE[187]						√	√		√	US	2023				√	VUzzer+libdft+Angr
CarpetFuzz[188]	√	√						√	√	US	2023	√	8	27		AFL++
DynSQL[189]								√	√	US	2023					AFL
FuzzJIT[190]							√	√		US	2023	√	5	41		Fuzzilli
Fuzztruction[191]			√		√		√		√	US	2023	√	10	82	√	AFL++
Hoedur[192]							√		√	US	2023	√	4	13	√	QEMU+libfuzzer+AFL
KextFuzz[193]	√	√								US	2023	√	1	26	√	Syzkaller
MorFuzz[194]							√	√	√	US	2023	√	1	7		**DifuzzRTL+riscv-dv+riscv-torture**
PolyFuzz[195]	√	√	√							US	2023	√	3	12		AFL++
Rubick[196]	√		√							US	2023					**FuzzGen**
SafireFuzz[197]		√							√	US	2023	√	3	70	√	LibAFL
WinFuzz[198]			√						√	US	2023	√			√	Winnie
HeapAFL[199]	√	√			√				√	计研	2023	√	2	2		AFL

[α] A：ASE，C：CCS，F：FSE，IC：ICSE，IS：ISSTA，N：NDSS，S：S & P，TD：TDSC，TI：TIFS，TO：TOSEM，TS：TSE，UA：USENIX ATC，US：USENIX Security，UW：USENIX WOOT，计研：计算机研究与发展

工作方面,AFL 因其可扩展性强、部署简单、测试效率高等优点成为近年来最流行的工具,在所调研的工作中有 68 项工作直接或间接在 AFL 的基础上进行改进,占比达 42.77%(不含仅将 AFL 作为基线工具,而未进行改进的研究工作).

4 应用场景

随着 CGF 技术的进步,CGF 的应用领域也不断被拓展.除最为常用的对 C/C++程序进行测试外,目前的 CGF 方法还可应用于 Java 程序测试、智能合约测试、操作系统内核测试等场景.例如 CGF 可帮助 C/C++开发者检测内存泄漏、缓冲区溢出、空指针解引用等漏洞;可帮助 Java 开发者检测空指针异常、数组越界异常、类型转换异常等漏洞;可帮助智能合约开发者检测整型溢出、重入、拒绝服务等漏洞.

在不同的应用场景下,研究人员关注的重点通常不同,例如针对闭源程序测试,通常更关注如何在源码不可用的情况下准确地进行程序插桩、提高对闭源程序的测试吞吐量等;针对结构化输入程序的测试,通常更关注如何生成格式有效性更高的测试用例,以通过 PUT 的输入解析阶段,对 PUT 的核心功能进行测试.测试人员需要针对具体应用场景选取合适的 CGF 工具,以进行充分且有效的测试.研究人员也需要从众多的 CGF 研究工作中挑选合适的基线工具,以验证自己的研究工作是否更加有效.然而,现有研究综述中未对 CGF 方法的应用场景进行总结,在 CGF 可用工具数量激增的情况下,测试人员和研究人员在选择适当的 CGF 工具进行测试或实验时存在一定困难.

针对此问题,我们调研了 159 篇 CGF 工作的应用场景,以为后续 CGF 工作中测试工具的选取提供参考,如表 3 所示.从表 3 中可以看出,在近年的 CGF 研究工作中,对 C/C++程序进行模糊测试仍然是最常见的应用场景,在 159 项 CGF 研究工作中共有 51 项研究工作应用于 C/C++程序测试,占 32.08%.其次是对结构化输入程序进行模糊测试,即将 CGF 应用于对输入有高度结构化要求的程序,例如 JS 引擎、编译器等.共有 19 项工作针对结构化输入程序测试场景对 CGF 进行了优化和改进,占 11.95%.此外,操作系统内核测试、闭源程序测试、库函数测试、虚拟设备测试等也是在近年来 CGF 研究领域中关注度较高的应用场景.

表 3 CGF 研究工作应用场景分类

应用场景	研究工作	数量/项
C/C++程序测试	文献[10,15,16,21-25,29,57-60,62,66,74,77-80,84,90,92,95,100,102-105,110,111,116,117,122,123,126,127,136,138,139,141,148,157,159,160,163,167,181,185,188,199]	51
结构化输入程序测试	文献[41,42,67,93,106,113,120,121,129,131,144,169,171,173,174,178,187,190,191]	19
操作系统内核测试	文献[30,46,47,53,54,98,109,137,179,180]	10
库函数测试	文献[31,64,82,132,149,150,152,170,177,196]	10
闭源程序测试	文献[43,44,65,73,79,85,134,168,198]	9
智能合约测试	文献[45,118,119,143,176,184]	6
固件测试	文献[124,151,158,175,192,197]	6
DBMS 系统测试	文献[51,52,140,155,189]	5
并行模糊测试	文献[61,101,145,164,186]	5
虚拟设备测试	文献[55,56,89,99,182]	5
硬件测试	文献[32,81,133,194]	4
驱动程序测试	文献[86,125,156,193]	4
DNN 测试	文献[107,162]	2
物联网设备测试	文献[112,153]	2
Java 程序测试	文献[63,108]	2
协议测试	文献[161,165]	2
DISC 应用程序测试	文献[114]	1
DOM 测试	文献[115]	1
微控制器测试	文献[172]	1
DL 推理引擎测试	文献[130]	1

(续表)

应用场景	研究工作	数量/项
异构应用测试	文献[128]	1
解决 SMT 公式	文献[83]	1
自动驾驶测试	文献[68]	1
浏览器 API 测试	文献[146]	1
MDSE 工具测试	文献[97]	1
多线程程序测试	文献[72]	1
多语言模糊测试	文献[195]	1
问答系统测试	文献[142]	1
符号执行引擎测试	文献[147]	1
飞地测试	文献[166]	1
网络应用程序测试	文献[154]	1
TCP 堆栈测试	文献[135]	1
Web 应用测试	文献[183]	1

5　评估分析

对 CGF 的有效性进行评估和分析通常可分为三个步骤:选取合适的测试对象、设计合理的实验设置和使用合理的评估指标. 首先,需要选取广泛应用,且尽可能多样的测试对象,以更全面地评估 CGF 在不同类型软件上的有效性. 然后,需要设计合理且有效的实验设置,以充分地对 CGF 方法进行实验. 最后,需要使用能够准确评估 CGF 方法特点的评估指标,以得出客观结论. 为此,本节总结了在 CGF 领域中常被使用的测试对象、实验设置和评估指标,为研究人员在未来的 CGF 工作中提供参考.

5.1　测试对象

随着近几年 CGF 工作数量的激增,已有研究工作在测试对象(Benchmark)的选取方面暴露出许多问题,主要表现在:第一,对于测试对象的选取较为随意且没有统一的选取准则,例如 Lyu 等人[23]从 UniFuzz[200]和多篇最新的模糊测试论文中随机选取了 9 个广泛使用的程序对模糊测试工具进行评估,She 等人[62]从 FuzzBench[201]测试集中选取了 12 个在模糊测试领域常用的测试对象,Gan 等人[60]则根据流行程度、测试频率、功能多样性等因素选取了 19 个 Linux 开源应用程序作为测试对象,这种随机选取测试对象或使用不统一的准则选取测试对象的方式可能会使实验结果和研究论文的结论出现偏见;例如:Yue 等人[10]和 She 等人[62]使用不同的测试对象对 EcoFuzz 和 FairFuzz 的覆盖能力进行了评估,在 Yue 等人的实验中,EcoFuzz 比 FairFuzz 覆盖了更多的边,而在 She 等人的实验中,则是 FairFuzz 比 EcoFuzz 覆盖了更多的边,除了测试对象不统一导致出现结论相悖外,实验设置、实验运行环境、模糊测试的随机性等因素也可能是影响实验结论不一致的原因;第二,在描述测试对象时,部分工作中存在命名不一致的情况,例如:XML 文件解析库 libxml2 在文献[21]中被称为 libxml,而在文献[10]中被称为 libxml2;图像处理工具包 JasPer 在文献[117]中被称为 jasper,而在文献[93]中被称为 LibJasper;最后,除命名不一致外,还存在测试对象名称不规范的问题,例如:文献[43]中选取了 mjs 作为测试对象,同样的程序在文献[29]中被称为 MJS,而在其项目文档①中的命名则是 mJS;PNG 图像处理库 libpng 在文献[25]中被称为 libpng,而在文献[61]中被称为 LibPNG. 从长远来看,这些问题可能会使研究人员在未来对于 CGF 领域测试对象的选取、名称等内容产生困扰,进而对 CGF 未来的发展产生负面影响.

针对这些问题,本文收集并整理了 159 篇 CGF 工作中使用的测试对象,并根据各测试对象被选择作为实验对象的次数进行了降序排序,如表 4 所示. 由于 CGF 的应用场景非常广泛,被广泛应用于库函数测试[82]、操作系统内核测试[46,47]、驱动程序测试[86]等,曾被使用过的测试对象众多,将所有的测试对象选取信息全部列在本文中将会占用较大篇幅,因此本文仅列出被选取次数超过 5 次的测试对象. 针对命名不一致的问题,我们通过访问测试对象的网页地址来确定其准确名称. 表 4 中分别统计了测试对象的名称、被选择作为实验对象的次数、下载

① https://github.com/cesanta/mjs/blob/master/README.md

地址和输入格式,我们希望表中所整理的信息可以为未来的 CGF 工作中数据集的选取提供参考.

从表 4 中可以看出,用于评估 CGF 工作的测试对象主要可分为两种,一种是使用真实软件项目组成的程序集进行评估,例如 fuzzer-test-suite、Binutils 等;另一种则是使用人工植入漏洞的程序组成的程序集进行评估,例如 LAVA-M. 在 CGF 领域中,被选择作为测试对象次数最多的是 GNU 二进制工具集 Binutils,其次是 XML 解析库 libxml2. 此外,libpng、libming、libarchive 等 PNG 文件处理程序、Flash 文件处理程序和压缩文件处理程序也常在 CGF 领域中被选为测试对象.

表 4 被选用次数超过 5 次的测试对象

测试对象名称	被选用次数/次	下载地址	输入格式
Binutils	38	https://www.gnu.org/software/binutils/	ELF
libxml2	23	https://gitlab.gnome.org/GNOME/libxml2	XML
libpng	21	http://www.libpng.org/pub/png/libpng.html	PNG
LibTIFF	18	http://www.libtiff.org/	TIFF
libjpeg	15	https://libjpeg.sourceforge.net/	JPEG
tcpdump	14	https://www.tcpdump.org/	PCAP
LAVA-M[203]	13	http://panda.moyix.net/~moyix/lava_corpus.tar.xz	FILE
NASM	12	https://www.nasm.us/	ASM
JasPer	11	https://jasper-software.github.io/jasper/	JP2、PNG 等
JHEAD	10	https://github.com/Matthias-Wandel/jhead	JPEG
HarfBuzz	10	https://github.com/harfbuzz/harfbuzz	TTF
FreeType	10	https://freetype.org/	TTF
FFmpeg	10	https://ffmpeg.org/	MP4、MP3 等
SQLite	9	https://www.sqlite.org/	DB
OpenSSL	9	https://www.openssl.org/	PEM
libjpeg-turbo	9	https://libjpeg-turbo.org/	JPEG
Linux 内核	8	https://github.com/torvalds/linux	-
libarchive	8	https://www.libarchive.org/	TAR
Exiv2	8	https://exiv2.org/	JP2、JPEG 等
Bison	8	https://www.gnu.org/software/bison/	LEX & YACC
Xpdf	7	http://www.xpdfreader.com/	PDF
RE2	7	https://github.com/google/re2	TXT
PCRE2	7	https://github.com/PCRE2Project/pcre2	TXT
NCURSES	7	https://invisible-island.net/ncurses/announce.html	TXT
MuPDF	7	https://mupdf.com/	PDF
libming	7	https://github.com/libming/libming	SWF
ImageMagick	7	https://imagemagick.org/index.php	PNG
fuzzer-test-suite	7	https://github.com/google/fuzzer-test-suite	XML、PNG 等
cflow	7	https://www.gnu.org/software/cflow/	C
V8	6	https://v8.dev/	JS
mJS	6	https://github.com/cesanta/mjs	JS
Libav	6	https://github.com/libav/libav	MP4、MP3 等
JavaScriptCore	6	https://github.com/apple-opensource/JavaScriptCore	JS
catdoc	6	https://github.com/petewarden/catdoc	MS-WORD

基于所统计的结果,我们发现现有工作中选取测试对象数量的中位数为 6,即通过 6 个不同的测试对象来验证 CGF 方法的有效性. 然而,在部分应用场景下选择少量的测试对象进行评估也是合理的,例如在内核模糊测试场景下难以获取多个测试对象(即操作系统内核)来评估 CGF 方法的有效性,此时对单个测试对象进行评估的结果同样具有一定的参考价值.

Hazimeh 等人[202] 在 2020 年针对已有模糊测试工作缺乏统一的指标与基准、难以进行评估与比较的问题,开发并发布了测试集 Magma. Magma 由输入类型不同的 7 个真实项目组成,Hazimeh 等人

将 7 个项目历史版本中的 118 个错误注入至新版本中,并同时为每个错误插入测试预言,以检测模糊测试工具是否在测试过程中到达或触发了错误. 与 Hazimeh 等人的工作不同的是,我们对近年来 CGF 研究领域常用的测试对象的流行程度进行了量化,并为测试对象的规范命名与选取提供了依据.

5.2　实验设置

在选取合适的数据集后,需要设计合理的实验设置,以对 CGF 方法进行评估. 目前,CGF 工作中的实验设置主要包括测试轮数及测试时间两方面. 合适的测试轮数和测试时间有助于客观地评估现有工作,过多的测试轮数或过长的测试时间虽然可以大幅减弱 CGF 过程中因随机性产生的影响,但也会使研究过程中的大部分时间花费在实验运行上,而过少的测试轮数及过短的测试时间虽然会耗费较少的硬件资源,节省实验运行时间,但可能会使实验结果或结论出现偏见.

已有的 CGF 工作中通常未对实验设置做出解释,并且实验设置不一致的问题较为明显,例如:Angora[25] 中所使用的实验设置是重复 5 次 5 小时的实验;CrFuzz[116] 中所使用的实验设置是重复 5 次 48 小时的实验;而 Gramatron[131] 中所使用的实验设置则是重复 10 次 24 小时的实验. 实验设置不一致的问题使研究人员难以在不同的论文之间直接进行比较. 此外,因实验设置不一致导致结论相悖的情况在部分论文中已有体现,例如:Redqueen[59] 和 NEUZZ[110] 同时将二进制工具集 Binutils 中的 size 当作评估对象,在 Redqueen 中的实验设置为重复 5 次10 小时的测试,其实验结论为 laf-intel 比 AFLFast 覆盖了更多的边,在 NEUZZ 中,实验设置为进行 24 小时的测试,未提及重复测试轮数,而其实验结论为 AFLFast 比 laf-intel 在 size 上覆盖了更多的边;EcoFuzz[10] 和 FairFuzz[100] 同时将 Binutils 中的 nm 当作评估对象,在 EcoFuzz 中的实验设置为重复5 次24 小时的实验,其实验结论为 FidgetyAFL① 比 FairFuzz 覆盖了更多的边,在 FairFuzz 中的实验设置为重复 20 次 24 小时的实验,其实验结论为 FairFuzz 比 FidgetyAFL 覆盖了更多的边. 产生这种现象一种可能的原因是测试用例演化的随机性会对实验结果产生部分影响,另一种可能的原因则是测试时间、测试轮数等实验设置的不一致使结论产生了偏差. 这种因实验设置不一致导致结论相悖的情况会使研究人员在研究过程中产生困扰,研究人员会难以决定应当相信哪项工作的结论,并且难以在不同的工作之间进行间接比较,从而对 CGF 未来的发展产生阻碍.

针对这些问题,本文收集并整理了 159 项 CGF 工作中各项工作的实验设置,包含实验时间和重复轮次,如表 5 所示. 由于篇幅原因,本文在此仅展示被选择作为实验设置最为频繁的前 6 种设置. 从表中可以看出,近 6 年来最为常用的实验设置为重复 5 次 24 小时的实验,其次为重复 10 次 24 小时的实验.

考虑到模糊测试过程中具有较高的随机性,实验时间较长且重复多轮才能较为客观地评价 CGF 方法. 目前 CGF 工作中最为流行的测试轮次为重复 5 次,因此,我们建议后续的 CGF 工作中采用至少重复 5 次,每次 24 小时测试的实验设置. 当测试重复 5 次或以上时,可以在一定程度上提升实验结果的可靠性,实验结果也会更加具有统计学意义. 此外,当测试时间设置为 24 小时或以上时,CGF 过程中因随机性而产生的影响也可以有效降低,可以更有效地说明不同工具间的性能差异[203].

表 5　CGF 工作中常用的实验设置

时间/h	测试轮次/轮	使用次数/次
24	5	33
24	10	21
5	5	6
48	5	6
24	20	6
24	3	5

5.3　评估指标

恰当的评估指标可以客观地对 CGF 方法进行评价,反映其真实能力. 在现阶段的 CGF 工作中,常用的评估指标主要有 3 种:覆盖率、崩溃数量和统计检验.

(1)覆盖率. CGF 的主要目标之一是达到更高的测试覆盖率,因此大部分 CGF 工作都会使用覆盖率来评估其有效性. 覆盖率越高,表明 CGF 方法可以更全面地探索 PUT,对 PUT 的测试也就越充分. 部分工作[15,60,157]沿用 AFL 中的独特路径数量作为评估指标,还有一部分工作使用代码行覆盖率、基本块覆盖率等信息进行评估. 此外,gcov②、llvm-cov③、

① https://groups.google.com/g/afl-users/c/fOPeb62FZUg/m/CES5lhznDgAJ

② https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

③ https://llvm.org/docs/CommandGuide/llvm-cov.html

afl-cov① 等覆盖率统计工具在 CGF 领域也取得了广泛应用,例如文献[25]使用 gcov 记录 PUT 中所有被覆盖的代码行与分支,文献[84]使用 llvm-cov 收集 PUT 的基本快覆盖与路径覆盖信息,文献[106]使用 afl-cov 度量 CGF 的代码行覆盖率与函数覆盖率.随着覆盖率统计工具准确度和易用性的不断增强,研究人员在未来可以更容易地对 CGF 方法在 PUT 上所达到的覆盖率进行准确评估.

(2)崩溃数量.除覆盖率外,所挖掘到的崩溃数量也可以作为评估 CGF 方法有效性的一种指标.当模糊测试工具所挖掘到的独特崩溃数量较多时,也间接说明其覆盖了较多的边.部分工作[10,29,117]沿用 AFL 的独特崩溃数量作为评估指标,还有一部分工作使用堆栈跟踪来对崩溃进行去重,并将去重后的崩溃数量作为评估指标.例如使用 AddressSanitizer② 编译被测项目生成 PUT,然后将触发程序崩溃的测试用例发送至 PUT,获取 PUT 在崩溃时的函数调用堆栈,并根据堆栈信息对崩溃进行分类与去重.已有研究工作会基于堆栈的前 N 层信息对崩溃进行分类与去重,例如文献[15,23]中 N 的取值为 3,文献[117]中 N 的取值为 5,而文献[110,141]中 N 的取值为无穷大.N 的取值对崩溃分类与去重的结果会产生较大的影响,当 N 的取值较大时,会为崩溃产生较为精确的分类,但是会在手动分析崩溃产生原因时为维护人员带来较大的负担;当 N 的取值较小时,则可能会在崩溃分类时产生较多的误分类.因此,需要根据崩溃数量、修复成本等多种因素决定 N 的取值.

(3)统计检验.统计检验可用于评估和分析随机算法(例如模糊测试)的有效性[205].Mann-Whitney U Test(U 检验)[206] 和 Vargha-Delaney statistic ($\hat{A}_{12}$)[207]是文献[205]中推荐的两种用于评估随机测试的统计检验方法,同时它们也常用于评估 CGF 工作的有效性.U 检验是一种非参数检验,可用于评价新方法和原方法之间是否存在差异(例如新方法是否显著优于原方法).在 U 检验中,零假设 H_0 通常定义为新方法与原方法之间没有差异,当 p 值小于等于某一阈值 α(通常为 0.01 或 0.05)时,则拒绝 H_0,即认为新方法与原方法相比有明显的差异;当 p 值大于 α 时,则接受 H_0,即认为新方法与原方法相比差异不显著.$\hat{A}_{12}$ 表示新方法比原方法可以获得更好效果的概率,取值范围为[0, 1],例如:当 $\hat{A}_{12}$ 的值为 0.7 时,意味着新方法有 70%的概率比原方法获得更好的效果.U 检验与 $\hat{A}_{12}$ 可用于比较不同 CGF 方法在覆盖率或触发崩溃数量方面是否存在显著差异,例如文献[78,85]基于新方法与基线方法的边覆盖实验结果使用 U 检验计算 p 值,并判断新方法的性能是否显著优于基线方法;文献[77,163]基于崩溃触发数量计算新方法与基线方法的 $\hat{A}_{12}$,并根据计算结果判断新方法是否有较高的概率比基线方法取得更好的测试效果.

此外,Fioraldi 等人[208]针对已有模糊测试工作常基于 AFL 进行开发,但未深入探讨 AFL 内部机制的问题,对 AFL 的不同组件进行了实验与分析,包括 AFL 的覆盖信息收集、种子选择策略、能量分配策略等,以充分了解 AFL 的各个组件是否均有助于提升模糊测试的效果.实验结果表明,虽然 AFL 的各个组件在部分场景下会产生正面影响,但在不同的测试场景下,这些组件也可能会带来负面影响.基于此现象,Fioraldi 等人认为仅关注崩溃数量和覆盖率往往难以理解指定方法为模糊测试所带来的效果和影响,应当在评估过程中针对测试场景或测试目标的特点,使用更细粒度的指标(例如超时测试用例数量、吞吐量等)来评估模糊测试方法的有效性.

6 总结与展望

作为目前最为有效的漏洞挖掘技术之一,模糊测试受到了工业界和学术界的广泛关注.其中,CGF 作为目前模糊测试的主流技术之一在近年来取得了较快发展,污点分析、机器学习等技术被应用于 CGF 并取得了良好效果,AFL、libfuzzer 等开源的 CGF 工具也因部署简单、可扩展性强、测试效率高等原因在多种场景下获得了成功应用.本文收集并分析了近 6 年与 CGF 相关的文献,按照预处理、测试用例选择、测试用例演化、测试用例评估 4 个阶段对已有工作进行总结,并分析了各类工作所做出的贡献和存在的不足.除此之外,本文还针对模糊测试评估过程中所存在的问题进行了研究与探讨,总结了评估 CGF 工作常用的测试对象、实验设置及评估指标.希望本文工作可以为未来的 CGF 研究提供参考.

基于本文对 CGF 研究进展的分析可知,CGF 在预处理、测试用例选择等不同阶段中仍存在可提升空间.本文为未来 CGF 方法的发展方向进行了展

① https://github.com/mrash/afl-cov

② https://github.com/google/sanitizers

望,供研究人员进行参考:

(1)提升程序分析的准确度.在 CGF 的预处理阶段,现有大部分 CGF 工作会在预处理阶段对 PUT 进行静态分析,例如生成 PUT 的控制流图、获取 PUT 中漏洞潜在位置等.利用静态分析可以高效地分析 PUT 并获取所需信息,然而静态分析结果可能会存在误报、漏报等问题,例如 AFLTeam 等工具中使用的静态分析技术难以将函数指针准确识别为函数调用,这可能会影响到 CGF 工具的测试效果.在未来可考虑将静态分析与动态分析进行结合,在测试用例执行过程中使用动态分析技术更新所获取的程序信息,以提高程序分析的准确性,并制导 CGF 达到更高的覆盖率.

(2)优化种子池的存储结构.在 CGF 的测试用例选择阶段,现有的大部分 CGF 工作在种子存储结构方面所使用的数据结构为线性数据结构,例如使用队列或链表存储种子.使用线性数据结构存储种子的优点是实现简单、插入效率高,但也存在一定局限性,例如在线性数据结构中查找所需种子渐进时间复杂度为 $O(n)$(n 为种子池中的种子数量),当种子池中的种子较多时,查找所需种子会产生较大时间开销.未来可将存储种子的数据结构进行改进,例如可使用非线性数据结构存储种子,并根据非线性数据结构的特点改进 CGF 测试流程,进一步降低选择种子、替换种子等各项操作的时间复杂度,提高 CGF 的测试效率.

(3)提升测试用例演化过程中的测试资源利用率.在 CGF 的测试用例演化阶段,CGF 在测试用例演化过程中通常会沿用随机变异的方式,尽管这种方式可以快速生成大量测试用例,但所生成的测试用例有效性较低,难以通过 PUT 中的特定约束条件,从而导致 CGF 难以充分地探索 PUT 的程序空间.针对此问题,Redqueen 通过对测试用例的字节进行染色并根据染色结果进行变异,以通过与魔法字节相关的约束;PATA 利用动态污点分析确定测试用例中的关键字节,并将测试资源集中在关键字节的变异上.除动态分析外,未来还可使用强化学习算法(如多臂老虎机、Q-learning)或其他机器学习算法调整测试用例演化过程中的能量分配和测试用例生成方式,制导 CGF 生成更高质量的子代测试用例,以提高测试资源的利用率.

(4)提升分析测试用例覆盖信息的准确性.在 CGF 的测试用例评估阶段,CGF 通常通过遍历位图来分析测试用例的边覆盖信息,以评估测试用例质量.位图的容量越大就可以存储越多的边覆盖信息,CGF 对测试用例覆盖信息的评估也就越准确,然而这也会使 CGF 在遍历位图上耗费较多时间,影响测试效率.当位图容量过小时,会频繁产生哈希冲突,进而导致 CGF 对测试用例覆盖信息评估的准确性下降.AFL 在哈希冲突和测试效率之间进行了权衡,将位图的大小设置为 64 KB,大多数基于 AFL 的工作也沿用了相同设置,但该容量对于大部分项目而言都会发生哈希冲突[79].未来研究工作可针对不同被测试程序的特点,改进 CGF 工具位图中记录测试用例边覆盖的方式,或利用其他数据结构代替位图来存储边覆盖信息,以更有效地平衡哈希冲突和测试效率,提高反馈分析的准确性.

(5)提升模糊测试工具的健壮性.近年来,模糊测试工具在数量逐渐增多,然而数量众多的模糊测试工具却难以保证其自身不存在漏洞.目前还没有研究针对模糊测试工具的特点进行健壮性测试或漏洞挖掘.未来可考虑将 CGF 方法应用于对模糊测试工具进行有针对性的测试,例如将差分测试与 CGF 方法进行结合,以对不同的模糊测试工具进行测试,挖掘工具中存在的漏洞.通过这种方式,可提高模糊测试工具的健壮性.

(6)构建更加多样的测试对象集.CGF 已被广泛应用于操作系统内核测试、数据库管理系统测试、虚拟机监视器测试等场景.然而,并非在各类场景下都有认可程度高且完善的基准程序集.在开源软件模糊测试场景下,目前有 FuzzBench、fuzzer-test-suite 等由真实软件项目组成的测试对象集,以及 LAVA-M、DARPA CGC 等通过人工植入漏洞构建的基准测试集.然而,在闭源软件测试、数据库管理系统测试等场景下仍然缺乏一个类似于 FuzzBench 的测试对象集.未来可考虑在已有工作的基础上,构建在不同测试场景下适用的 CGF 测试对象,以覆盖更加多样的测试场景,为后续的研究工作提供实验对象支撑.

致　谢　在此,我们向对本文工作给予支持和提出宝贵建议的评审老师和同行表示衷心的感谢!

参　考　文　献

[1] Liu B, Shi L, Cai Z, et al. Software vulnerability discovery techniques: A survey// Proceedings of the 2012 Fourth International Conference on Multimedia Information Networking

and Security. Nanjing, China, 2012: 152-156

[2] Li Zhou-Jun, Zhang Jun-Xian, Liao Xiang-Ke, et al. Survey of software vulnerability detection techniques. Chinese Journal of Computers, 2015, 38(4): 717-732 (in Chinese)
(李舟军, 张俊贤, 廖湘科, 等. 软件安全漏洞检测技术. 计算机学报, 2015, 38(4): 717-732)

[3] Orso A, Rothermel G. Software testing: A research travelogue (2000-2014)// Future of Software Engineering Proceedings. Hyderabad, India, 2014: 117-132

[4] Miller B, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. Communications of the ACM, 1990, 33(12): 32-44

[5] Pang Q, Yuan Y, Wang S. MDPFuzz: Testing models solving markov decision processes// Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. Korea, 2022: 378-390

[6] Walz A, Sikora A. Exploiting dissent: Towards fuzzing-based differential black-box testing of TLS implementations. IEEE Transactions on Dependable and Secure Computing, 2017, 17(2): 278-291

[7] Wang M, Liang J, Chen Y, et al. SAFL: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing// Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings. Gothenburg, Sweden, 2018: 61-64

[8] Cho M, Kim S, Kwon T. Intriguer: Field-level constraint solving for hybrid fuzzing// Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. London, UK, 2019: 515-530

[9] Zhu X, Böhme M. Regression greybox fuzzing// Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. Korea, 2021: 2169-2182

[10] Yue T, Wang P, Tang Y, et al. EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit// Proceedings of the 29th USENIX Security Symposium. Boston, USA, 2020: 2307-2324

[11] Chen J, Diao W, Zhao Q, et al. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing// Proceedings of the 2018 Network and Distributed System Security Symposium. San Diego, USA, 2018: 1-15

[12] Wang T, Wei T, Gu G, et al. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection// Proceedings of the 2010 IEEE Symposium on Security and Privacy. Oakland, USA, 2010: 497-512

[13] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting fuzzing through selective symbolic execution// Proceedings of the 2016 Network and Distributed Systems Security Symposium. San Diego, USA, 2016: 1-16

[14] DeMott J, Enbody R, Punch W. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing// Proceedings of the BlackHat and Defcon. Las Vegas, USA, 2007: 1-17

[15] Lyu C, Ji S, Zhang C, et al. MOpt: Optimized mutation scheduling for fuzzers// Proceedings of the 28th USENIX Security Symposium. Santa Clara, USA, 2019: 1949-1966

[16] She D, Krishna R, Yan L, et al. MTFuzz: Fuzzing with a multi-task neural network// Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. USA, 2020: 737-749

[17] Böhme M, Pham V, Nguyen M, et al. Directed greybox fuzzing// Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas, USA, 2017: 2329-2344

[18] Huang H, Guo Y, Shi Q, et al. Beacon: Directed grey-box fuzzing with provable path pruning// Proceedings of the 2022 IEEE Symposium on Security and Privacy. San Francisco, USA, 2022: 36-50

[19] Zhang J, Cui Z, Chen X, et al. DeltaFuzz: Historical version information guided fuzz testing. Journal of Computer Science and Technology, 2022, 37(1): 29-49

[20] Manès V, Han H, Han C, et al. The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering, 2019, 47(11): 2312-2331

[21] Wu M, Jiang L, Xiang J, et al. Evaluating and improving neural program-smoothing-based fuzzing// Proceedings of the 44th International Conference on Software Engineering. Pittsburgh, USA, 2022: 847-858

[22] Wu M, Jiang L, Xiang J, et al. One fuzzing strategy to rule them all// Proceedings of the 44th International Conference on Software Engineering. Pittsburgh, USA, 2022: 1634-1645

[23] Lyu C, Liang H, Ji S, et al. SLIME: Program-sensitive energy allocation for fuzzing// Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. Korea, 2022: 365-377

[24] Liang J, Wang M, Zhou C, et al. PATA: Fuzzing with path aware taint analysis// Proceedings of the 2022 IEEE Symposium on Security and Privacy. San Francisco, USA, 2022: 154-170

[25] Chen P, Chen H. Angora: Efficient fuzzing by principled search// Proceedings of the 2018 IEEE Symposium on Security and Privacy. San Francisco, USA, 2018: 711-725

[26] Wang P, Zhou X, Lu K, et al. SoK: The progress, challenges, and perspectives of directed greybox fuzzing. arXiv preprint arXiv:2005.11907, 2020

[27] Saavedra G, Rodhouse K, Dunlavy D, et al. A review of machine learning applications in fuzzing. arXiv preprint arXiv: 1906.11133, 2019

[28] Zhu X, Wen S, Camtepe S, et al. Fuzzing: A survey for roadmap. ACM Computing Surveys, 2022, 54(11s): 230:1-230:36

[29] Li Y, Xue Y, Chen H, et al. Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection// Proceedings

of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Tallinn, Estonia, 2019: 533-544

[30] Jeong D, Kim K, Shivakumar B, et al. Razzer: Finding kernel race bugs through fuzzing// Proceedings of the 2019 IEEE Symposium on Security and Privacy. San Francisco, USA, 2019: 754-768

[31] Blair W, Mambretti A, Arshad S, et al. HotFuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing// Proceedings of the 2020 Network and Distributed System Security Symposium. San Diego, USA, 2020: 1-18

[32] Kande R, Crump A, Persyn G, et al. TheHuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities// Proceedings of the 31st USENIX Security Symposium. Boston, USA, 2022: 3219-3236

[33] Kitchenham B, Brereton O, Budgen D, et al. Systematic literature reviews in software engineering-a systematic literature review. Information and Software Technology, 2009, 51(1): 7-15

[34] Böhme M, Cadar C, Roychoudhury A. Fuzzing: Challenges and reflections. IEEE Software, 2020, 38(3): 79-86

[35] Godefroid P. Fuzzing: Hack, art, and science. Communications of the ACM, 2020, 63(2): 70-76

[36] Wang Y, Jia P, Liu L, et al. A systematic review of fuzzing based on machine learning techniques. PLOS One, 2020, 15(8): e0237749

[37] Li J, Zhao B, Zhang C. Fuzzing: A survey. Cybersecurity, 2018, 1(1): 6:1-6:13

[38] Liang H, Pei X, Jia X, et al. Fuzzing: State of the art. IEEE Transactions on Reliability, 2018, 67(3): 1199-1218

[39] Eisele M, Maugeri M, Shriwas R, et al. Embedded fuzzing: A review of challenges, tools, and solutions. Cybersecurity, 2022, 5(1): 18:1-18:18

[40] Ren Ze-Zhong, Zheng Han, Zhang Jia-Yuan, et al. A review of fuzzing techniques. 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. 计算机研究与发展, 2021, 58(5): 944-963)

[41] Park S, Xu W, Yun I, et al. Fuzzing JavaScript engines with aspect-preserving mutation// Proceedings of the 2020 IEEE Symposium on Security and Privacy. San Francisco, USA, 2020: 1629-1642

[42] He X, Xie X, Li Y, et al. SoFi: Reflection-augmented fuzzing for JavaScript engines// Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. Virtual Event, Korea, 2021: 2229-2242

[43] Nagy S, Nguyen-Tuong A, Hiser J, et al. Same coverage, less bloat: Accelerating binary-only fuzzing with coverage-preserving coverage-guided tracing// Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. Korea, 2021: 351-365

[44] Gao X, Duck G, Roychoudhury A. Scalable fuzzing of program binaries with E9AFL// Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering. Melbourne, Australia, 2021: 1247-1251

[45] Wüstholz V, Christakis M. Harvey: A greybox fuzzer for smart contracts// Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. USA, 2020: 1398-1409

[46] Xu M, Kashyap S, Zhao H, et al. Krace: Data race fuzzing for kernel file systems// Proceedings of the 2020 IEEE Symposium on Security and Privacy. San Francisco, USA, 2020: 1643-1660

[47] Pailoor S, Aday A, Jana S. MoonShine: Optimizing OS fuzzer seed selection with trace distillation// Proceedings of the 27th USENIX Security Symposium. Baltimore, USA, 2018: 729-743

[48] Rawat S, Jain V, Kumar A, et al. VUzzer: Application-aware evolutionary fuzzing// Proceedings of the 2017 Network and Distributed Systems Security Symposium. San Diego, USA, 2017: 1-14

[49] Li Y, Chen B, Chandramohan M, et al. Steelix: Program-state based binary fuzzing// Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. Paderborn, Germany, 2017: 627-637

[50] Xu W, Kashyap S, Min C, et al. Designing new operating primitives to improve fuzzing performance// Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas, USA, 2017: 2313-2328

[51] Wang M, Wu Z, Xu X, et al. Industry practice of coverage-guided enterprise-level DBMS fuzzing// Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice. Madrid, Spain, 2021: 328-337

[52] Zhong R, Chen Y, Hu H, et al. Squirrel: Testing database management systems with language validity and coverage feedback// Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. USA, 2020: 955-970

[53] Xu W, Moon H, Kashyap S, et al. Fuzzing file systems via two-dimensional input space exploration// Proceedings of the 2019 IEEE Symposium on Security and Privacy. San Francisco, USA, 2019: 818-834

[54] Song D, Hetzelt F, Kim J, et al. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints// Proceedings of the 29th USENIX Security Symposium. Boston, USA, 2020: 2541-2557

[55] Bulekov A, Das B, Hajnoczi S, et al. Morphuzz: Bending input space to fuzz virtual devices// Proceedings of the 31st USENIX Security Symposium. Boston, USA, 2022: 1221-1238

[56] Myung C, Lee G, Lee B. MundoFuzz: Hypervisor fuzzing with statistical coverage testing and grammar inference// Proceedings of the 31st USENIX Security Symposium. Boston, USA, 2022: 1257-1274

[57] Nagy S, Hicks M. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing// Proceedings of the 2019 IEEE Symposium on Security and Privacy. San Francisco, USA, 2019: 787-802

[58] Hsu C, Wu C, Hsiao H, et al. InsTrim: Lightweight instrumentation for coverage-guided fuzzing// Symposium on Network and Distributed System Security, Workshop on Binary Analysis Research. San Diego, USA, 2018: 1-7

[59] Aschermann C, Schumilo S, Blazytko T, et al. Redqueen: Fuzzing with input-to-state correspondence// Proceedings of the 2019 Symposium on Network and Distributed System Security. San Diego, USA, 2019: 1-15

[60] Gan S, Zhang C, Chen P, et al. GreyOne: Data flow sensitive fuzzing// Proceedings of the 29th USENIX Security Symposium. Boston, USA, 2020: 2577-2594

[61] Pham V, Nguyen M, Ta Q, et al. Towards systematic and dynamic task allocation for collaborative parallel fuzzing // Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering. Melbourne, Australia, 2021: 1337-1341

[62] She D, Shah A, Jana S. Effective seed scheduling for fuzzing with graph centrality analysis// Proceedings of the 2022 IEEE Symposium on Security and Privacy. San Francisco, USA, 2022: 2194-2211

[63] Brennan T, Saha S, Bultan T. JVM fuzzing for jit-induced side-channel detection// Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. Korea, 2020: 1011-1023

[64] Ispoglou K, Austin D, Mohan V, et al. FuzzGen: Automatic fuzzer generation// Proceedings of the 29th USENIX Security Symposium. Boston, USA, 2020: 2271-2287

[65] Liu Y, Wang Y, Su P, et al. InstruGuard: Find and fix instrumentation errors for coverage-based greybox fuzzing // Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering. Melbourne, Australia, 2021: 568-580

[66] Wang M, Liang J, Zhou C, et al. RIFF: Reduced instruction footprint for coverage-guided fuzzing// Proceedings of the 2021 USENIX Annual Technical Conference. 2021: 147-159

[67] Salls C, Jindal C, Corina J, et al. Token-level fuzzing// Proceedings of the 30th USENIX Security Symposium. 2021: 2795-2809

[68] Sun Y, Poskitt C, Sun J, et al. LawBreaker: An approach for specifying traffic laws and fuzzing autonomous vehicles// Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. Rochester, USA, 2022: 62:1-62:12

[69] Chen J, Feng Y, Dillig I. Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic// Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas, USA, 2017: 875-890

[70] Gens D, Schmitt S, Davi L, et al. K-Miner: Uncovering memory corruption in Linux// Proceedings of the 2018 Network and Distributed System Security Symposium. San Diego, USA, 2018: 1-15

[71] Liu H, Liu C, Zhao W, et al. S-gram: Towards semantic-aware security auditing for ethereum smart contracts // Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. Montpellier, France, 2018: 814-819

[72] Chen H, Guo S, Xue Y, et al. Muzz: Thread-aware greybox fuzzing for effective bug hunting in multithreaded programs// Proceedings of the 29th USENIX Security Symposium. Boston, USA, 2020: 2325-2342

[73] Dinesh S, Burow N, Xu D, et al. RetroWrite: Statically instrumenting cots binaries for fuzzing and sanitization // Proceedings of the 2020 IEEE Symposium on Security and Privacy. San Francisco, USA, 2020: 1497-1511

[74] Zhou C, Wang M, Liang J, et al. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling// Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. Melbourne, Australia, 2020: 858-870

[75] Bellard F. QEMU, a fast and portable dynamic translator// Proceedings of the Annual Conference on USENIX Annual Technical Conference. Anaheim, USA, 2005: 41-46

[76] Nagy S, Nguyen-Tuong A, Hiser J, et al. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing// Proceedings of the 30th USENIX Security Symposium. Vancouver, Canada, 2021: 1683-1700

[77] Wen C, Wang H, Li Y, et al. Memlock: Memory usage guided fuzzing// Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. Seoul, Korea, 2020: 765-777

[78] Wang Y, Jia X, Liu Y, et al. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization// Proceedings of the 2020 Network and Distributed System Security Symposium. San Diego, USA, 2020: 1-17

[79] Gan S, Zhang C, Qin X, et al. Path sensitive fuzzing for native applications. IEEE Transactions on Dependable and Secure Computing, 2020, 19(3): 1544-1561

[80] Herrera A, Payer M, Hosking A. datAFLow: Toward a data-flow-guided fuzzer. ACM Transactions on Software Engineering and Methodology, 2023, 32(5): 132:1-132:31

[81] Trippel T, Shin K, Chernyakhovsky A, et al. Fuzzing hardware like software// Proceedings of the 31st USENIX Security Symposium. Boston, USA, 2022: 3237-3254

[82] Jiang J, Xu H, Zhou Y. RULF: Rust library fuzzing via API dependency graph traversal// Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software

Engineering. Melbourne, Australia, 2021: 581-592

[83] Liew D, Cadar C, Donaldson A, et al. Just fuzz it: Solving floating-point constraints using coverage-guided fuzzing// Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Tallinn, Estonia, 2019: 521-532

[84] Zhang M, Liu J, Ma F, et al. IntelliGen: Automatic driver synthesis for fuzz testing// Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice. Madrid, Spain, 2021: 318-327

[85] Jung J, Tong S, Hu H, et al. Winnie: Fuzzing windows applications with harness synthesis and fast cloning// Proceedings of the 2021 Network and Distributed System Security Symposium. Virtual Event, 2021: 1-17

[86] Ma Z, Zhao B, Ren L, et al. PrIntFuzz: Fuzzing Linux drivers via automated virtual device simulation// Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. Virtual, Korea, 2022: 404-416

[87] Chen H, Xue Y, Li Y, et al. Hawkeye: Towards a desired directed grey-box fuzzer// Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto, Canada, 2018: 2095-2108

[88] Du Z, Li Y, Liu Y, et al. WindRanger: A directed greybox fuzzer driven by deviation basic blocks// Proceedings of the 44th International Conference on Software Engineering. Pittsburgh, USA, 2022: 2440-2451

[89] Pan G, Lin X, Zhang X, et al. V-Shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing// Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. Korea, 2021: 2197-2213

[90] Wang J, Song C, Yin H. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing// Proceedings of the 2021 Network and Distributed System Security Symposium. 2021: 1-17

[91] Wang J, Chen B, Wei L, et al. Skyfire: Data-driven seed generation for fuzzing// Proceedings of the 2017 IEEE Symposium on Security and Privacy. San Jose, USA, 2017: 579-594

[92] Zhang K, Xiao X, Zhu X, et al. Path transitions tell more: Optimizing fuzzing schedules via runtime program states// Proceedings of the 44th International Conference on Software Engineering. Pittsburgh, USA, 2022: 1658-1668

[93] Pham V, Böhme M, Santosa A, et al. Smart greybox fuzzing. IEEE Transactions on Software Engineering, 2019, 47(9): 1980-1997

[94] Böhme M, Pham V, Roychoudhury A. Coverage-based greybox fuzzing as markov chain. IEEE Transactions on Software Engineering, 2017, 45(5): 489-506

[95] Böhme M, Manès V, Cha S. Boosting fuzzer efficiency: An information theoretic perspective// Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. USA, 2020: 678-689

[96] Shannon C. A mathematical theory of communication. The Bell System Technical Journal, 1948, 27(3): 379-423

[97] Nguyen H, Nassar N, Kehrer T, et al. MoFuzz: A fuzzer suite for testing model-driven software engineering tools// Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering. Melbourne, Australia, 2020: 1103-1115

[98] Maier D, Radtke B, Harren B. Unicorefuzz: On the viability of emulation for kernelspace fuzzing// Proceedings of the 13th USENIX Workshop on Offensive Technologies. Santa Clara, USA, 2019: 1-11

[99] Schumilo S, Aschermann C, Abbasi A, et al. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types// Proceedings of the 30th USENIX Security Symposium. 2021: 2597-2614

[100] 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, France, 2018: 475-485

[101] Liang J, Jiang Y, Chen Y, et al. PAFL: Extend fuzzing optimizations of single mode to industrial parallel mode// Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Lake Buena Vista, USA, 2018: 809-814

[102] Lemieux C, Padhye R, Sen K, et al. PerfFuzz: Automatically generating pathological inputs// Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. Amsterdam, The Netherlands, 2018: 254-265

[103] Chen P, Liu J, Chen H. Matryoshka: Fuzzing deeply nested branches// Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. London, UK, 2019: 499-513

[104] Babić D, Bucur S, Chen Y, et al. FUDGE: Fuzz driver generation at scale// Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Tallinn, Estonia, 2019: 975-985

[105] You W, Liu X, Ma S, et al. SLF: Fuzzing without valid seed inputs// Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering. Montreal, Canada, 2019: 712-723

[106] Wang J, Chen B, Wei L, et al. Superion: Grammar-aware greybox fuzzing// Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering. Montreal, Canada, 2019: 724-735

[107] Xie X, Ma L, Juefei-Xu F, et al. DeepHunter: A coverage-guided fuzz testing framework for deep neural networks//

Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. Beijing, China, 2019: 146-157

[108] Padhye R, Lemieux C, Sen K, et al. Semantic fuzzing with zest// Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. Beijing, China, 2019: 329-340

[109] Song D, Hetzelt F, Das D, et al. PeriScope: An effective probing and fuzzing framework for the hardware-os boundary// Proceedings of the 2019 Network and Distributed Systems Security Symposium. San Diego, USA, 2019: 1-15

[110] She D, Pei K, Epstein D, et al. NEUZZ: Efficient fuzzing with neural program smoothing// Proceedings of the 2019 IEEE Symposium on Security and Privacy. San Francisco, USA, 2019: 803-817

[111] You W, Wang X, Ma S, et al. ProFuzzer: On-the-fly input type probing for better zero-day vulnerability discovery // Proceedings of the 2019 IEEE Symposium on Security and Privacy. San Francisco, USA, 2019: 769-786

[112] Zheng Y, Davanian A, Yin H, et al. Firm-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation// Proceedings of the 28th USENIX Security Symposium. Santa Clara, USA, 2019: 1099-1114

[113] Blazytko T, Bishop M, Aschermann C, et al. Grimoire: Synthesizing structure while fuzzing// Proceedings of the 28th USENIX Security Symposium. Santa Clara, USA, 2019: 1985-2002

[114] Zhang Q, Wang J, Gulzar M, et al. BigFuzz: Efficient fuzz testing for data analytics using framework abstraction// Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering. Melbourne, Australia, 2020: 722-733

[115] Xu W, Park S, Kim T. FreeDom: Engineering a state-of-the-art DOM fuzzer// Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. USA, 2020: 971-986

[116] Song S, Song C, Jang Y, et al. CrFuzz: Fuzzing multi-purpose programs through input validation// Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. USA, 2020: 690-700

[117] Manès V, Kim S, Cha S. Ankou: Guiding grey-box fuzzing towards combinatorial difference// Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. Seoul, Korea, 2020: 1024-1036

[118] Nguyen T, Pham L, Sun J, et al. sFuzz: An efficient adaptive fuzzer for solidity smart contracts// Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. Seoul, Korea, 2020: 778-788

[119] Grieco G, Song W, Cygan A, et al. Echidna: Effective, usable, and fast fuzzing for smart contracts// Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. USA, 2020: 557-560

[120] Mathis B, Gopinath R, Zeller A. Learning input tokens for effective fuzzing// Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. USA, 2020: 27-37

[121] Fioraldi A, D'Elia D, Coppa E. WEIZZ: Automatic grey-box fuzzing for structured binary formats// Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. USA, 2020: 1-13

[122] Aschermann C, Schumilo S, Abbasi A, et al. IJON: Exploring deep state spaces via fuzzing// Proceedings of the 2020 IEEE Symposium on Security and Privacy. San Francisco, USA, 2020: 1597-1612

[123] Jiang Z, Bai J, Lu K, et al. Fuzzing error handling code using context-sensitive software fault injection// Proceedings of the 29th USENIX Security Symposium. Boston, USA, 2020: 2595-2612

[124] Ruge J, Classen J, Gringoli F, et al. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets// Proceedings of the 29th USENIX Security Symposium. Boston, USA, 2020: 19-36

[125] Peng H, Payer M. USBFuzz: A framework for fuzzing USB drivers by device emulation// Proceedings of the 29th USENIX Security Symposium. Boston, USA, 2020: 2559-2575

[126] Fioraldi A, Maier D, Eißfeldt H, et al. AFL++: Combining incremental steps of fuzzing research// Proceedings of the 14th USENIX Workshop on Offensive Technologies. Boston, USA, 2020: 1-12

[127] Zhou C, Wang M, Liang J, et al. VisFuzz: Understanding and intervening fuzzing with interactive visualization// Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering. San Diego, USA, 2019: 1078-1081

[128] Zhang Q, Wang J, Kim M. HeteroFuzz: Fuzz testing to detect platform dependent divergence for heterogeneous applications// Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Athens, Greece, 2021: 242-254

[129] Vikram V, Padhye R, Sen K. Growing a test corpus with bonsai fuzzing// Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering. Madrid, Spain, 2021: 723-735

[130] Luo W, Chai D, Ruan X, et al. Graph-based fuzz testing for deep learning inference engines// Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering. Madrid, Spain, 2021: 288-299

[131] Srivastava P, Payer M. Gramatron: Effective grammar-aware fuzzing// Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. Denmark, 2021: 244-256

[132] Noller Y, Tizpaz-Niari S, QFuzz: Quantitative fuzzing for

side channels// Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. Denmark, 2021: 257-269

[133] Hur J, Song S, Kwon D, et al. DifuzzRTL: Differential fuzz testing to find CPU bugs// Proceedings of the 2021 IEEE Symposium on Security and Privacy. San Francisco, USA, 2021: 1286-1303

[134] Zhang Z, You W, Tao G, et al. StochFuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting// Proceedings of the 2021 IEEE Symposium on Security and Privacy. San Francisco, USA, 2021: 659-676

[135] Zou Y, Bai J, Zhou J, et al. TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing// Proceedings of the 2021 USENIX Annual Technical Conference. Virtual Event, 2021: 489-502

[136] Fioraldi A, D'Elia D, Balzarotti D. The use of likely invariants as feedback for fuzzers// Proceedings of the 30th USENIX Security Symposium. 2021: 2829-2846

[137] Wang D, Zhang Z, Zhang H, et al. SyzVegas: Beating kernel fuzzing odds with reinforcement learning// Proceedings of the 30th USENIX Security Symposium. 2021: 2741-2758

[138] Chen J, Wang S, Cai S, et al. A novel coverage-guided greybox fuzzing based on power schedule optimization with time complexity// Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. Rochester, USA, 2022: 172:1-172:5

[139] Chen H, Chen J. Coverage-based greybox fuzzing with pointer monitoring for C programs// Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. Rochester, USA, 2022: 121:1-121:4

[140] Fu J, Liang J, Wu Z, et al. Griffin: Grammar-free DBMS fuzzing// Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. Rochester, USA, 2022: 49:1-49:12

[141] Yu Y, Jia X, Liu Y, et al. HTFuzz: Heap operation sequence sensitive fuzzing// Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. Rochester, USA, 2022: 53:1-53:13

[142] Liu Z, Feng Y, Yin Y, et al. QATest: A uniform fuzzing framework for question answering systems// Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. Rochester, USA, 2022: 81:1-81:12

[143] Su J, Dai H, Zhao L, et al. Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing// Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. Rochester, USA, 2022: 36:1-36:12

[144] Bernhard L, Scharnowski T, Schloegel M, et al. Jit-Picker: Differential fuzzing of JavaScript engines// Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. Los Angeles, USA, 2022: 351-364

[145] Gu T, Li X, Lu S, et al. Group-based corpus scheduling for parallel fuzzing// Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Singapore, 2022: 1521-1532

[146] Zhou C, Zhang Q, Wang M, et al. Minerva: Browser API fuzzing with dynamic mod-ref analysis// Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Singapore, 2022: 1135-1147

[147] Li P, Meng W, Lu K. SEDiff: Scope-aware differential fuzzing to test internal function models in symbolic execution// Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Singapore, 2022: 57-69

[148] Nguyen H, Grunske L. BeDivFuzz: Integrating behavioral diversity into generator-based fuzzing// Proceedings of the 44th International Conference on Software Engineering. Pittsburgh, USA, 2022: 249-261

[149] Wei A, Deng Y, Yang C, et al. Free lunch for testing: Fuzzing deep-learning libraries from open source// Proceedings of the 44th International Conference on Software Engineering. Pittsburgh, USA, 2022: 995-1007

[150] Green H, Avgerinos T. GraphFuzz: Library API fuzzing with lifetime-aware dataflow graphs// Proceedings of the 44th International Conference on Software Engineering. Pittsburgh, USA, 2022: 1070-1081

[151] Li W, Shi J, Li F, et al. μAFL: Non-intrusive feedback-driven fuzzing for microcontroller firmware// Proceedings of the 44th International Conference on Software Engineering. Pittsburgh, USA, 2022: 1-12

[152] Gu J, Luo X, Zhou Y, et al. Muffin: Testing deep learning libraries via neural architecture fuzzing// Proceedings of the 44th International Conference on Software Engineering. Pittsburgh, USA, 2022: 1418-1430

[153] Zheng Y, Li Y, Zhang C, et al. Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation// Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. Korea, 2022: 417-428

[154] Andronidis A, Cadar C. SnapFuzz: High-throughput fuzzing of network applications// Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. Korea, 2022: 340-351

[155] Wu Z, Liang J, Wang M, et al. Unicorn: Detect runtime errors in time-series databases with hybrid input synthesis// Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. Korea, 2022: 251-262

[156] Zhao W, Lu K, Wu Q, et al. Semantic-Informed driver fuzzing without both the hardware devices and the emulators

//Proceedings of the 2022 Network and Distributed Systems Security Symposium. San Diego, USA, 2022: 1-16

[157] Lyu C, Ji S, Zhang X, et al. EMS: History-driven mutation for coverage-based fuzzing// Proceedings of the 2022 Network and Distributed System Security Symposium. San Diego, USA, 2022: 1-18

[158] Hernandez G, Muench M, Maier D, et al. FirmWire: Transparent dynamic analysis for cellular baseband firmware //Proceedings of the 2022 Network and Distributed System Security Symposium. San Diego, USA, 2022: 1-19

[159] Zhang G, Wang P, Yue T, et al. MobFuzz: Adaptive multi-objective optimization in gray-box fuzzing // Proceedings of the 2022 Network and Distributed Systems Security Symposium. San Diego, USA, 2022: 1-18

[160] Zhu X, Feng X, Meng X, et al. CSI-Fuzz: Full-speed edge tracing using coverage sensitive instrumentation. IEEE Transactions on Dependable and Secure Computing, 2020, 19(2): 912-923

[161] Li J, Li S, Sun G, et al. SNPSFuzzer: A fast greybox fuzzer for stateful network protocols using snapshots. IEEE Transactions on Information Forensics and Security, 2022, 17: 2673-2687

[162] Zhang P, Ren B, Dong H, et al. CAGFuzz: Coverage-guided adversarial generative fuzzing testing for image-based deep learning systems. IEEE Transactions on Software Engineering, 2021, 48(11): 4630-4646

[163] Menendez H, Clark D. Hashing fuzzing: Introducing input diversity to improve crash detection. IEEE Transactions on Software Engineering, 2021, 48(9): 3540-3553

[164] Zhou X, Wang P, Liu C, et al. UltraFuzz: Towards resource-saving in distributed fuzzing. IEEE Transactions on Software Engineering, 2022, 49(4): 2394-2412

[165] Ba J, Böhme M, Mirzamomen Z, et al. Stateful greybox fuzzing// Proceedings of the 31st USENIX Security Symposium. Boston, USA, 2022: 3255-3272

[166] Cloosters T, Willbold J, Holz T, et al. SGXFuzz: Efficiently synthesizing nested structures for SGX enclave fuzzing// Proceedings of the 31st USENIX Security Symposium. Boston, USA, 2022: 3147-3164

[167] Lee J, ViganòE, Cornejo O, et al. Fuzzing for CPS mutation testing. arXiv preprint arXiv:2308.07949, 2023

[168] Cao S, Sun X, Wu X, et al. Improving java deserialization gadget chain mining via overriding-guided object generation. arXiv preprint arXiv:2303.07593, 2023

[169] Wu M, Lu M, Cui H, et al. JITfuzz: Coverage-guided fuzzing for JVM just-in-time compilers// Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering. Melbourne, Australia, 2023: 56-68

[170] Yang C, Deng Y, Yao J, et al. Fuzzing automatic differentiation in deep-learning libraries. arXiv preprint arXiv:2302.04351, 2023

[171] Guo S, Wan X, You W, et al. Operand-variation-oriented differential analysis for fuzzing binding calls in PDF readers //Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering. Melbourne, Australia, 2023: 95-107

[172] Eisele M, Ebert D, Huth C, et al. Fuzzing embedded systems using debug interfaces// Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. Seattle, USA, 2023: 1031-1042

[173] Even-Mendoza K, Sharma A, Donaldson A, et al. GrayC: Greybox fuzzing of compilers and analysers for C// Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. Seattle, USA, 2023: 1219-1231

[174] Ma H, Shen Q, Tian Y, et al. Fuzzing deep learning compilers with HirGen// Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. Seattle, USA, 2023: 248-260

[175] Chesser M, Nepal S, Ranasinghe D. Icicle: A re-designed emulator for grey-box firmware fuzzing. arXiv preprint arXiv:2301.13346, 2023

[176] Shou C, Tan S, Sen K. ItyFuzz: Snapshot-based fuzzer for smart contract// Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. Seattle, USA, 2023: 322-333

[177] Deng Y, Xia C, Peng H, et al. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models// Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. Seattle, USA, 2023: 423-435

[178] Groß S, Koch S, Bernhard L, et al. Fuzzilli: Fuzzing for JavaScript JIT compiler vulnerabilities // Proceedings of the 2023 Network and Distributed Systems Security Symposium. San Diego, USA, 2023: 1-17

[179] Bulekov A, Das B, Hajnoczi S, et al. No grammar, No problem: Towards fuzzing the linux kernel without system-call descriptions// Proceedings of the 2023 Network and Distributed Systems Security Symposium. San Diego, USA, 2023: 1-16

[180] Jeong D, Lee B, Shin I, et al. SegFuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing// Proceedings of the 2023 IEEE Symposium on Security and Privacy. San Francisco, USA, 2023: 2104-2121

[181] Jeong B, Jang J, Yi H, et al. UTopia: Automatic generation of fuzz driver using unit tests// Proceedings of the 2023 IEEE Symposium on Security and Privacy. San Francisco, USA, 2023: 2676-2692

[182] Liu Q, Toffalini F, Zhou Y, et al. ViDeZZo: Dependency-aware virtual device fuzzing// Proceedings of the 2023 IEEE Symposium on Security and Privacy. San Francisco, USA, 2023: 3228-3245

[183] Trickel E, Pagani F, Zhu C, et al. Toss a fault to your

witcher: Applying grey-box coverage-guided mutational fuzzing to detect SQL and command injection vulnerabilities // Proceedings of the 2023 IEEE Symposium on Security and Privacy. San Francisco, USA, 2023: 2658-2675

[184] Liu Z, Qian P, Yang J, et al. Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting. IEEE Transactions on Information Forensics and Security, 2023, 18: 1237-1251

[185] Zhang Z, Klees G, Wang E, et al. Fuzzing configurations of program options. ACM Transactions on Software Engineering and Methodology, 2023, 32(2): 53:1-53:21

[186] Chen Y, Zhong R, Yang Y, et al. μFuzz: Redesign of parallel fuzzing using microservice architecture // Proceedings of the 32nd USENIX Security Symposium. Anaheim, USA, 2023: 1325-1342

[187] Shi J, Wang Z, Feng Z, et al. AIFORE: Smart fuzzing based on automatic input format reverse engineering // Proceedings of the 32nd USENIX Security Symposium. Anaheim, USA, 2023: 4967-4984

[188] Wang D, Li Y, Zhang Z, et al. CarpetFuzz: Automatic program option constraint extraction from documentation for fuzzing // Proceedings of the 32nd USENIX Security Symposium. Anaheim, USA, 2023: 1919-1936

[189] Jiang Z, Bai J, Su Z. DynSQL: Stateful fuzzing for database management systems with complex and valid SQL query generation // Proceedings of the 32nd USENIX Security Symposium. Anaheim, USA, 2023: 4949-4965

[190] Wang J, Zhang Z, Liu S, et al. FuzzJIT: Oracle-enhanced fuzzing for JavaScript engine JIT compiler // Proceedings of the 32nd USENIX Security Symposium. Anaheim, USA, 2023: 1865-1882

[191] Bars N, Schloegel M, Scharnowski T, et al. Fuzztruction: Using fault injection-based fuzzing to leverage implicit domain knowledge // Proceedings of the 32nd USENIX Security Symposium. Anaheim, USA, 2023: 1847-1864

[192] Scharnowski T, Wörner S, Buchmann F, et al. Hoedur: Embedded firmware fuzzing using multi-stream inputs // Proceedings of the 32nd USENIX Security Symposium. Anaheim, USA, 2023: 2885-2902

[193] Yin T, Gao Z, Xiao Z, et al. KextFuzz: Fuzzing macOS kernel extensions on apple silicon via exploiting mitigations // Proceedings of the 32nd USENIX Security Symposium. Anaheim, USA, 2023: 5039-5054

[194] Xu J, Liu Y, He S, et al. MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation // Proceedings of the 32nd USENIX Security Symposium. Anaheim, USA, 2023: 1307-1324

[195] Li W, Ruan J, Yi G, et al. PolyFuzz: Holistic greybox fuzzing of multi-language systems // Proceedings of the 32nd USENIX Security Symposium. Anaheim, USA, 2023: 1379-1396

[196] Zhang C, Li Y, Zhou H, et al. Automata-guided control-flow-sensitive fuzz driver generation // Proceedings of the 32nd USENIX Security Symposium. Anaheim, USA, 2023: 2867-2884

[197] Seidel L, Maier D, Muench M. Forming faster firmware fuzzers // Proceedings of the 32nd USENIX Security Symposium. Anaheim, USA, 2023: 2903-2920

[198] Stone L, Ranjan R, Nagy S, et al. No linux, No problem: Fast and correct windows binary fuzzing via target-embedded snapshotting // Proceedings of the 32nd USENIX Security Symposium. Anaheim, USA, 2023: 4913-4929

[199] Yu Yuan-Ping, Su Pu-Rui. HeapAFL: Heap-behavior guided greybox fuzzing. Journal of Computer Research and Development, 2023, 60(7): 1501-1513 (in Chinese)
(余媛萍, 苏璞睿. HeapAFL: 基于堆操作行为引导的灰盒模糊测试. 计算机研究与发展, 2023, 60(7): 1501-1513)

[200] Li Y, Ji S, Chen Y, et al. UniFuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers // Proceedings of the 30st USENIX Security Symposium. 2021: 2777-2794

[201] Metzman J, Szekeres L, Simon L, et al. FuzzBench: An open fuzzer benchmarking platform and service // Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Athens, Greece, 2021: 1393-1403

[202] Hazimeh A, Herrera A, Payer M. Magma: A ground-truth fuzzing benchmark. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2020, 4(3): 49: 1-49:29

[203] Dolan-Gavitt B, Hulin P, Kirda E, et al. Lava: Large-scale automated vulnerability addition // Proceedings of the 2016 IEEE Symposium on Security and Privacy. San Jose, USA, 2016: 110-121

[204] Klees G, Ruef A, Cooper B, et al. Evaluating fuzz testing // Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto, Canada, 2018: 2123-2138

[205] Arcuri A, Briand L. A practical guide for using statistical tests to assess randomized algorithms in software engineering // Proceedings of the 33rd International Conference on Software Engineering. Honolulu, USA, 2011: 1-10

[206] McKnight P, Najab J. Mann-Whitney u test. The Corsini Encyclopedia of Psychology, 2010: 1-1

[207] Vargha A, Delaney H. 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

[208] Fioraldi A, Mantovani A, Maier D, et al. Dissecting american fuzzy lop: A fuzzbench evaluation. ACM Transactions on Software Engineering and Methodology, 2023, 32(2): 52:1-52:26

CUI Zhan-Qi, Ph. D., professor. His research interests include intelligent software engineering and trustworthy artificial intelligence.

ZHANG Jia-Ming, Ph. D. candidate. His research interests include software analysis and testing.

ZHENG Li-Wei, Ph. D., associate professor. His research interests include environment driven requirement engineering, intelligent software engineering, and swarm intelligence algorithm.

CHEN Xiang, Ph. D., associate professor. His research interests include software testing, software maintenance, empirical software engineering, and mining software repository.

Background

This research belongs to the software engineering area, focusing on the research progress of Coverage-guided Greybox Fuzzing (CGF). Fuzzing (or fuzz testing) is a reliable software quality assurance technology. Its core idea is sending randomly generated test cases to the program under test (PUT) automatically or semi-automatically to improve the robustness of PUT. Among them, CGF has received extensive attention from scientific researchers and industrial engineers because of its simple deployment, strong scalability, and discovery of many vulnerabilities in real-world software.

In recent years, many research achievements have been made in the field of CGF. To enhance the code coverage and vulnerability discovery abilities of CGF, researchers combine CGF with other technologies, such as machine learning and taint analysis. Additionally, the application scenarios of CGF have been expanded to include kernel testing, network protocol testing, and IoT device testing to improve the robustness of various software. Unfortunately, although CGF has developed rapidly in recent years, some problems have also been exposed. For example, the evaluation settings in different papers are inconsistent, leading to contradictory conclusions. In addition, the selection of benchmarks is relatively random, and their names are not standardized, which confuses researchers about benchmark selection. Furthermore, there is still no survey that systematically summarizes the research achievements of CGF, which also makes it difficult for researchers to keep abreast of the latest research progress of CGF.

For this reason, we review the research achievements of CGF in recent years. First, we divide the process of CGF into four stages: preprocessing, test case selection, test case evolution, and test case evaluation. Then, we systematically summarize the research progress of CGF in different stages. Finally, we provide statistics on the commonly used evaluation settings in the CGF field and discuss the future research directions of CGF. Compared with existing works, we summarize the research progress of CGF for the first time and discuss the contributions and limitations of the current CGF techniques, which can serve as a reference for future CGF research.

This work was supported by the Jiangsu Provincial Frontier Leading Technology Fundamental Research Project (BK20202001), the National Natural Science Foundation of China (61702041), and the Beijing Information Science and Technology University "Qin-Xin Talent" Cultivation Project (QXTCP C201906).