计 算 机 学 报
CHINESE JOURNAL OF COMPUTERS

定向灰盒模糊测试技术研究进展

徐邑江[1),2),3)] 高庆[1),2),3)] 陈立果[1),2),3)] 张世琨[1),2),3)] 吴中海[1),2),3)]

[1)](北京大学 软件与微电子学院, 北京 102600)

[2)](北京大学 软件工程国家工程研究中心, 北京 100871)

[3)](高可信软件技术教育部重点实验室(北京大学), 北京 100871)

摘 要 随着软件的复杂性和规模的增加，软件的安全问题日益突出。由于能够自动化生成和优化测试用例，帮助开发者检测和发现软件中的漏洞，模糊测试成为了保证软件安全的一种重要技术。与黑盒和白盒模糊测试技术相比，灰盒模糊测试技术兼顾了黑盒关注的软件的外部行为和白盒关注的实现细节，能够兼顾潜在安全漏洞检测的能力和效率，成为模糊测试技术领域的研究热点。但是，一般的灰盒模糊测试技术只能用于发现软件中尽可能多的未知漏洞，无法高效适用日益增长的静态分析报告验证、软件补丁回归测试、崩溃重现等特定目标探索需求场景。2017 年提出的定向灰盒模糊测试技术，聚焦于目标代码区域或目标类型漏洞的触发，能够实现有向的探索，受到广泛关注。为此，本文从定向灰盒模糊测试技术的流程出发，围绕测试目标的选取、目标定向的预处理和定向模糊测试的性能优化等 3 个关键环节，分析定向灰盒模糊测试技术的有效性及其面临的挑战性问题，归纳和总结了国内外的相关研究进展，包括：目标的手动、自动化选取和多目标优化方法，基于距离最小化、输入可达性和序列的代码路径定向引导方法，以及基于全局能量调度、自适应种子突变、多级优先级队列等目标触发性能提升技术，并展望了未来的发展和研究方向。

关键词 定向灰盒模糊测试；软件安全；漏洞检测；测试目标选取；目标定向；模糊测试性能优化

中图法分类号 TP309

Research Progress on Directed Grey-box Fuzzing

XU Yi-Jiang[1),2),3)] GAO Qing[1),2),3)] CHEN Li-Guo[1),2),3)] ZHANG Shi-Kun[1),2),3)] WU Zhong-Hai[1),2),3)]

[1)](School of Software and Microelectronics, Peking University, Beijing 102600)

[2)](National Engineering Research Center for Software Engineering, Peking University, Beijing 100871)

[3)](Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University), Beijing 100871)

Abstract With the increasing complexity and scale of software, security issues have become more prominent than ever. Fuzz testing has emerged as a crucial technique for ensuring software security due to its ability to automatically generate and optimize test cases, assisting developers in detecting and identifying vulnerabilities within software systems. Compared to black-box and white-box fuzzing techniques, grey-box fuzzing achieves a balance between these two approaches by incorporating both the external behavior analysis of black-box testing and the internal implementation details assessment of white-box testing. This hybrid approach allows grey-box fuzzing to combine the advantages of both techniques, striking a compromise between vulnerability detection capabilities and testing efficiency, making it a highly researched and widely adopted technique in the field of fuzz testing.

———————————————

本课题得到国家重点研发计划(2021YFB3101802)资助. 徐邑江，博士研究生，主要研究领域为软件分析、漏洞检测.高庆(**通信作者**)，男，1989年生，博士，助理研究员，主要研究领域为软件分析、漏洞检测. 陈立果，博士研究生，主要研究领域为软件分析、大模型评估. 张世琨，博士，研究员，博士生导师，中国计算机学会(CCF)高级会员，主要研究领域为软件工程、网络安全、知识计算. 吴中海，博士，教授，博士生导师，中国计算机学会(CCF)杰出会员，主要研究领域为软件工程，大数据系统与分析,大数据与云安全，嵌入式系统.

However, conventional grey-box fuzzing techniques primarily focus on discovering as many unknown vulnerabilities as possible within a software system. While this approach is effective for general vulnerability detection, it lacks efficiency when applied to specific target-driven exploration scenarios that are becoming increasingly critical in modern software security analysis. These scenarios include verifying static analysis reports, regression testing of software patches, and reproducing software crashes, where a more focused and target-oriented testing approach is required. To address these limitations, the concept of directed grey-box fuzzing was introduced in 2017. Unlike traditional fuzzing techniques that aim to maximize the overall vulnerability discovery rate, directed grey-box fuzzing emphasizes the targeted triggering of specific code regions or particular types of vulnerabilities. By directing the fuzzing process towards predefined targets, directed grey-box fuzzing enhances the efficiency and effectiveness of security testing in targeted applications, attracting widespread attention from both researchers and industry practitioners due to its orientation ability.

To gain a deeper understanding of the effectiveness and challenges of directed grey-box fuzzing, this paper explores its technical workflow by analyzing three key components: the selection of testing targets, the preprocessing of target-oriented guidance, and the performance optimization of directed fuzzing execution. Through this analysis, we assess the capabilities of directed grey-box fuzzing techniques while identifying the key obstacles that remain in this field. We summarize and synthesize relevant research advancements from both domestic and international studies. These advancements include various methods for selecting fuzzing targets, which can be categorized into manual selection, automated selection, and multi-objective optimization approaches. Additionally, we discuss essential techniques for guiding fuzzing towards target-specific code regions, such as distance minimization strategies, input reachability analysis, and sequence-based code path guidance mechanisms, all of which play a vital role in improving the overall effectiveness of security testing. Furthermore, we examine performance enhancement techniques aimed at improving the efficiency of triggering target vulnerabilities. These include global energy scheduling strategies, adaptive seed mutation techniques, and multi-level priority queue mechanisms designed to optimize the fuzzing process. By systematically summarizing these advancements, we provide a comprehensive overview of the current state of directed grey-box fuzzing research, identifying both its strengths and areas for future improvement. Finally, we discuss potential directions for further research in this domain, including multi-type crash detection, multi-metric fuzz testing, support for multiple programming languages, and leveraging large language models to enhance directed grey-box fuzzing to further refine fuzz testing approaches.

Key words directed grey-box fuzzing; software security; vulnerability detection; selection of testing targets; target orientation; fuzz testing performance optimization

1 引 言

近年来，随着软件技术的快速发展和软件体量的不断增长，软件安全问题日益严峻，恶意攻击者不断寻找新的漏洞和攻击方法，以便窃取敏感信息、破坏系统或进行其他不法行为[1-2]。根据知名的CVE（Common Vulnerabilities and Exposures）详细信息网站 CVEdetails[1][3]统计，仅 2021 年 1 月-2025 年1月，累计公开的CVE漏洞数量已经达到119,401个。因此，确保软件的安全性已成为软件开发的核心关注点。

软件测试是检测软件安全性的重要手段，其主要目的是验证软件是否正确实现预期功能、检查程序中是否存在潜在缺陷、检验新版本是否引入了新的漏洞等[4-6]。但是，传统的软件测试方法往往需要手动构建大量测试用例，以确保对被测软件进行全面测试[7]。针对这一问题，Miller 等人[8]提出了模糊测试（Fuzz Testing）方法，通过自动化生成大规模随机或变异的测试输入，以缓解手动构建测试用例所带来的开销，进而帮助开发者更早地识别和修复

1 CVEDetails 提供了关于 MITRE 公司维护的 CVE 详细信息,包含相关数据统计信息、补丁和解决方案等.

缺陷或漏洞，降低后期维护成本[9-10]。所以，模糊测试成为保障软件安全性的一种重要软件测试技术。

在模糊测试领域，黑盒模糊测试[11-13]是一种通过随机生成测试用例触发程序异常的测试方法，它关注软件的外部行为，不考虑程序的内部结构和实现细节，无法发现一些特定于实现的缺陷或安全问题。白盒模糊测试[14-16]是一种利用程序内部信息生成针对性测试用例的测试方法，它关注软件的内部结构和实现细节，可以更有针对性的检测程序的不同路径和边界条件，但是分析的求解难度较大，需要更大的时间开销和更多的计算资源。灰盒模糊测试[17-21]是一种结合程序的外部行为和有限的内部结构信息的测试方法，它结合了前两者的优势，允许测试者在了解部分程序内部信息的基础上，更加有效地设计测试用例生成方法并检测潜在的安全漏洞。

灰盒模糊测试相比传统模糊测试，引入了程序执行反馈机制，使测试输入的生成和变异更加有针对性，从而提高代码覆盖率和漏洞发现效率。目前主流的基于覆盖率引导的灰盒模糊测试（Coverage-based Grey-box Fuzzing, CGF）[24-29]，通过实时反馈来提高目标程序的代码覆盖率，进而增加检测到缺陷和漏洞的可能性。例如人们使用AFL[30]工具已经发现了数百个高影响漏洞[31]，效果显著。

但是，基于覆盖率引导的灰盒模糊测试仅用于发现程序中尽可能多的未知漏洞，无法高效适用特定目标探索场景。随着各类开源软件和商业软件被报告的缺陷和漏洞数量越来越多，静态分析报告验证[32]、软件补丁回归测试[33-34]、崩溃重现[35-36]等特定目标场景下的模糊测试需求越来越迫切。2017年，M Böhme 等人[37]首次提出了针对灰盒模糊测试的定向引导工具 AFLGo，即定向灰盒模糊测试（Directed Grey-box Fuzzing, DGF），旨在通过生成有针对性的输入，提高对特定目标（如公开的 CVE 漏洞或特定的代码行位置等）程序测试的效率和覆盖率。该方法的关键在于它能够识别出程序中目标位置区域，并集中资源进行定向深入测试。

定向灰盒模糊测试具有软件测试技术中演化测试技术[38-39]与逻辑覆盖测试技术[40]的优点，融合了演化测试在测试用例生成上的目标导向特性和逻辑覆盖测试在测试效果评价上的覆盖率指标。演化测试是一种基于搜索的软件测试方法[41-43]，其侧重于利用搜索策略（如遗传算法[38-39]等）探索输入空间，以动态优化测试输入，逐步逼近目标区域，并生成高效的测试用例[38-39]。逻辑覆盖测试则通过分析测试用例对程序分支、路径或基本块的覆盖情况，利用覆盖率指标综合评价测试充分性[44]。因此，定向灰盒模糊测试具备演化测试生成导向性测试用例的能力，采用逻辑覆盖测试对测试效果的评价机制，通过插桩技术实时收集程序的覆盖信息，使测试既能精准导向目标区域，又能在复杂输入空间中生成高效测试用例，从而提高触发漏洞的效率。

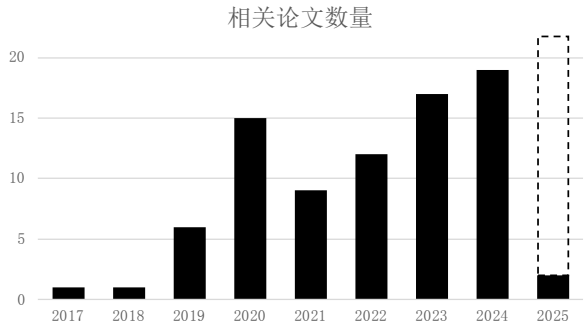

图 1　论文发表的年份分布

由于定向灰盒模糊测试在特定目标漏洞和缺陷分析方面的能力突出，近年来吸引了越来越多的研究人员的关注。本文广泛搜集了当前定向灰盒模糊测试的相关研究工作论文，通过使用“directed/guided/grey-box/fuzz/fuzzing”等关键词查询了 Google Scholar、DBLP、arXiv、IEEE Xplore、ACM Digital Library 以及中国知网 CNKI 等学术搜索引擎和数据库。本文主要选取了 CCF 分级较高的期刊和会议的正式论文，包括 CCF-A 类、CCF-B 类和部分高质量的 CCF-C 类、arXiv 论文。

经统计发现，自 2017 年提出至 2025 年 1 月，定向灰盒模糊测试领域的研究工作论文数量呈现逐年递增的趋势（见图 1），在包括 TDSC、ICSE、CCS、USENIX SECURITY、S&P、NDSS 等顶刊顶会在内的相关高水平学术论文达到 82 篇。其中，定向灰盒模糊测试技术在内核和嵌入式等环境中近年也有一些研究[45-48]，而本文主要围绕应用程序定向模糊测试存在的定向性能和准确性问题开展研究，主要包括测试目标的选取有效性不足、目标定向预处理的准确性不足和定向模糊测试的性能瓶颈。

因此，在定向灰盒模糊测试的研究中，有效选取测试目标、进行目标定向的预处理，以及优化定向模糊测试的性能是提升测试效率和能力的关键。

一是，测试目标的有效选取是定向灰盒模糊测试区别于覆盖率引导灰盒模糊测试的关键所在。但在测试目标选取过程中，除了需要从公开的漏洞报告和分析结果中快速分析和找出测试目标的位置信息以外[37,49]，为解决多个不同测试目标重复测试带来的资源消耗大的问题，还需要有效划分多个测试目标及其关联性[50,51]，这使得目标选取的有效性和效率问题成为学术界关注的重要方向。本文通过整理和归纳相关研究工作，将这个方向总结为目标的选取方式、多目标优化的方法。

二是，目标定向预处理是定向灰盒模糊测试提升目标定向准确性的重要技术手段。但在预处理过程中，如何对被测程序执行路径和测试目标之间的关系进行有效的度量[37,52]，减少测试时在不相关路径上的探索，实现向目标更快靠近[49,53]，或者实现依赖特定序列的漏洞目标定向[54-55]，成为了学术界关注的难点问题。本文总结和梳理了针对这类问题的相关研究工作，将它们归纳为基于距离最小化的方法、基于输入可达性的方法和基于序列导向的方法。

三是，定向模糊测试的性能是制约定向灰盒模糊测试方法可用性的瓶颈问题。在模糊测试的动态跟踪代码执行路径和状态变化过程中，现有覆盖率引导的方法缺乏对触发目标崩溃能力的反馈，难以有效提升测试的定向能力。如何根据测试目标选取和目标定向的预处理所得到的关键信息，提升模糊测试阶段的定向测试性能，是现有研究工作所关注的重点[37,52,56]。本文总结了相关研究工作，主要包括全局能量调度算法、启发式种子突变策略和多级优先级队列等方法。

近年来，国内外学者研究了灰盒模糊测试相关技术在漏洞检测、性能优化等方面的应用，形成了一些综述性的论文。其中，Cui 等人[57]对覆盖率引导的灰盒模糊测试工作进行了系统化的分析，总结了相关重要研究成果；Schloegel 等人[58]对模糊测试技术进行了综述，其中包括对相关论文实现的灰盒模糊器的可复现性、评估方法进行了细化分析；Manès 等人[10]对灰盒模糊测试在不同阶段的设计方法和策略创新进行了总结，系统地分析了不同模糊器的优化方案。然而，上述综述工作均没有对定向灰盒模糊测试领域研究工作进行明确的分类与总结。Wang 等人[59]对定向灰盒模糊测试进行了综述，但该工作重点关注 2021 年之前的研究工作，而且主要聚焦在不同类型研究工作的统计分析，以及相关方法的局限性分析和研究挑战性问题的讨论上。本文则是从定向灰盒模糊测试技术的流程出发，围绕测试目标的选取、目标定向的预处理和定向模糊测试的性能优化等 3 个关键环节，分析定向灰盒模糊测试技术的有效性及其面临的挑战性问题。本文系统性地归纳和总结了定向灰盒模糊测试领域的相关研究主要进展，包括：目标选取方式和多目标优化方法；基于距离最小化、输入可达性和序列导向的代码路径定向引导方法，以及基于全局能量调度、自适应种子突变、多级优先级队列等目标触发的性能提升技术，并展望了未来的发展和研究方向。

如图 2 所示，本文第 2 章介绍了定向灰盒模糊测试技术基本概念、具体流程及主要的工作原理；第 3 章梳理了定向灰盒模糊测试技术的应用现状，分析了面临的挑战性问题，总结了研究方向；第 4 章、第 5 章和第 6 章分别对现有工作中测试目标选取、目标定向预处理和定向模糊测试性能优化等方法进行了详细的分析和总结；第 7 章对现有工作中主要使用的测试对象进行了统计和分析，以帮助从事相关领域研究的人员提供参考；最后，在第 8 章展望了定向灰盒模糊测试技术的未来发展方向。

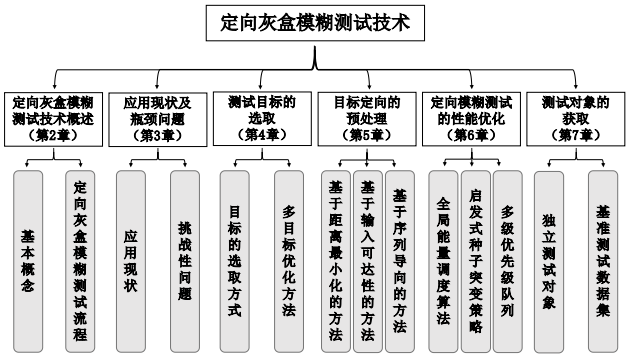

图 2 文章结构

2 定向灰盒模糊测试技术概述

2.1 基本概念

DGF[37]是一种结合目标导向搜索和灰盒测试特性的模糊测试技术，旨在通过引导测试输入生成过程，使其快速聚焦于特定的目标区域（如潜在漏洞、特定代码片段或关键路径）。DGF 通过综合利用代码覆盖信息、目标导向性机制和测试用例生成策略，提升指定漏洞的检测效率，在静态分析报告验证[32]、软件补丁回归测试[33-34]、崩溃重现[35-36]等特定目标场景下（具体见 3.1 节）效果显著。文

中使用的主要术语及英文全称和缩写参见附录 1。

2.2 定向灰盒模糊测试技术流程

本文通过对多篇相关工作[37,49,50,52,54,56]的总结，将 DGF 的执行过程归纳为 3 个主要阶段：测试目标选取、定向预处理及定向模糊测试循环，如图 3 所示，其中：

（1）测试目标选取。选取测试目标是 DGF 流程的首要步骤，具体的方法分为手动定义目标和自动生成目标。手动定义目标关注的是具体的 CVE 编号或目标在程序中的具体位置[37,52]；自动生成目标关注的是疑似存在问题的区域，通常使用自动化的工具分析提取[60,61]。一些工作还会针对选取的不同目标间的关联性，设计多目标优化方法，以期望在少量的测试中触发更多的目标[50,51]。

（2）定向预处理。定向预处理过程主要通过静态分析等程序分析技术，生成测试对象程序调用图等关键信息。利用这些信息设计目标相关的有效度量方法，主要有基于距离最小化的方法[37,52]、基于输入可达性的方法[49,53]和基于序列导向的方法[54-55]。DGF 在编译阶段进行度量相关的插桩，以便在定向模糊测试的过程中实时监测程序执行情况。编译后生成的二进制文件作为模糊测试循环的输入。

（3）定向模糊测试循环。DGF 在定向模糊测试阶段，主要包含能量调度、种子突变和种子优先级排序这 3 个核心模块[52]。在测试过程中，模糊器会维护一个种子队列，每次从队列中选取第一个种子进行测试。针对选定的种子，模糊器首先进行能量调度规划，依据程序分析结果和测试循环中收集的实时信息来规划该种子需要进行的突变次数[37,62]。接着，执行规划次数的突变操作。这些突变可能是随机的，也可能是基于启发式的策略，以增强测试的针对性[51,52]。突变后的种子被作为测试用例来执行程序，如果执行过程中触发了程序崩溃，该测试用例便被记录下来；如果没有触发崩溃，则利用预处理阶段的插桩信息评估该突变后种子的抵达目标和触发崩溃的潜力，将更有希望触发崩溃的种子赋予更高的优先级，加入种子优先级队列[52,56]。模糊测试工具根据上述规则持续地进行测试，直到触发指定崩溃或时间限制截止。

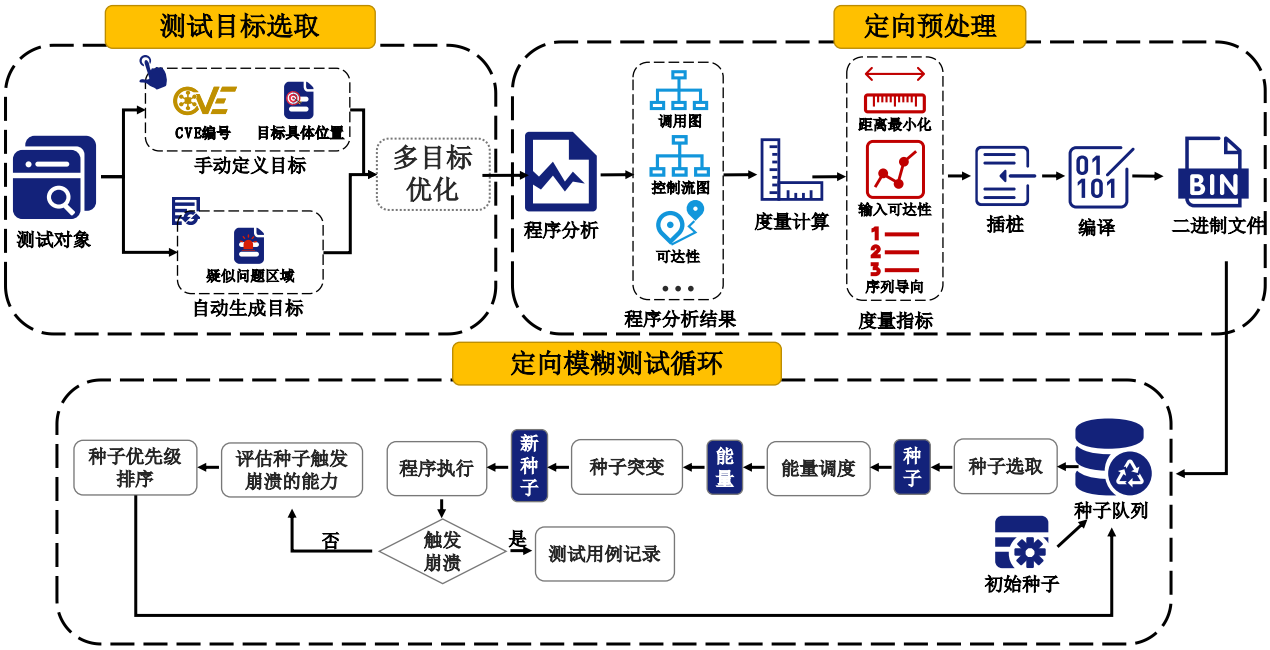

图 3 定向灰盒模糊测试的基本流程

3 应用现状及挑战性问题

3.1 应用现状

选择合适的应用场景，往往是从事 DGF 技术研究人员所关注的第一要素。在此，本文总结了 3 个典型场景，展示了 DGF 技术在软件开发和维护中可发挥的关键作用。

（1）静态分析报告验证[32]。项目的开发团队通常在软件开发过程中利用静态分析工具来检测源代码中的潜在错误。静态分析工具在编译阶段分析代码，能够生成关于可能错误的详细诊断报告，不仅指出潜在的崩溃位置，还会描述可能导致问题

的特定数据条件[63]。但由于其较高的误报率，开发人员通常不会对结果进行逐一验证，直到相关的错误在真实环境中引起崩溃[64]。DGF 可以通过指定目标进行模糊测试的方式，生成大量目标相关的测试用例，模拟各种可能的执行路径以验证静态分析的结果。DGF 可以实际触发代码中的潜在错误，帮助开发人员确认静态分析报告中的问题是否真实存在，从而在软件发布前修复这些问题，进而提升软件产品的可靠性和用户满意度。

（2）软件补丁回归测试[33-34]。回归测试是软件开发和维护过程中的关键环节，它为了确保软件在经历必要的迭代和更新后仍保持稳定和安全[65,66]。软件开发的过程需要不断地迭代，当软件产生安全漏洞修复、功能改进、性能优化、兼容性更新等新需求时，会对原有软件添加补丁。然而，补丁旨在解决特定问题，可能只考虑了程序的局部，从而可能引入新的漏洞[37]。传统的回归测试技术通常用于验证补丁修改后代码功能的正确性，而 DGF 可以通过其导向性的检测能力，进一步优化这一过程。DGF 利用程序分析技术识别补丁影响的代码区域，结合代码逻辑和结构生成有针对性的测试输入，聚焦这些区域进行深入模糊测试，从而更高效地验证补丁的正确性。

（3）崩溃重现[35-36]。开发人员在进行软件维护时，经常需要处理来自用户或其他开发人员的崩溃报告，报告中通常包含单个概念验证（Proof of Concept, PoC）输入和详细的崩溃转储[67-68]。然而，单个 PoC 输入只代表了一个具体的执行路径，可能无法涵盖引起崩溃的所有潜在原因，即使开发人员修改了代码使得特定的 PoC 输入不再导致崩溃，也难以确信是否彻底解决了根本的问题。为了更深入地理解和验证崩溃的根本原因是否已经被修复，开发人员可以利用 DGF，通过设置目标测试点为已知的崩溃位置及其相关代码区域，生成多种可能的输入场景，模拟不同的执行路径以重现崩溃，更有效地指导程序的改进。

3.2 挑战性问题

DGF 技术目前被广泛地使用到多个场景，同时，DGF 技术在应用中也暴露出了诸多挑战性问题。如何在实际应用中提升定向灰盒模糊器的性能和准确率，也成为学术界的研究热点。本文对 DGF 技术的研究工作进行了梳理和总结，将其挑战性问题总结为以下 3 个方面。

（1）测试目标的选取方法的有效性不足。在测试目标选取过程中，从公开的漏洞报告或缺陷分析结果中难以快速分析和找出定向测试目标的位置信息；并且，在多目标同时选取和处理的需求下，没有有效地划分多个目标之间的关联，可能导致定向测试退化为无向探索。

（2）测试目标定向的预处理方法准确性不足。在预处理过程中，对被测程序执行路径和测试目标之间的关系缺乏有效的度量，可能导致测试时，将资源浪费在不相关路径上的探索，难以准确地引导测试向目标靠近。

（3）定向模糊测试循环的性能存在瓶颈。在模糊测试的动态跟踪代码执行路径和状态变化过程中，现有覆盖率引导的种子能量调度、种子突变策略和优先级排序方法，缺乏对触发目标崩溃能力的反馈，难以有效地提升测试的定向能力。

针对上述挑战性问题，学术界和工业界进行了详细的分析和研究，并提出了相应的解决方案。本文对检索到的 2017 年至 2025 年 1 月期间发表的软件工程、网络与信息安全等领域的顶刊顶会在内的相关论文工作（见图 1）进行了总结和分类，如表 1 所示，包括以下 3 类。

（1）测试目标的选择。总结现有工作中针对测试目标的手动定义和自动生成方法，并重点分析多目标的优化方法。目前，针对测试目标选取的方法分为目标的选取方式和多目标优化的方法。

（2）目标定向的预处理。总结现有工作中提升 DGF 定向引导能力度量的预处理方法，分析对比不同方法的有效性。目前，针对目标定向的预处理方法分为基于距离最小化的方法、基于输入可达性的方法和基于序列导向的方法。

（3）定向模糊测试的性能优化。总结现有工作中定向模糊测试循环阶段的目标触发性能提升方法，分析不同模糊测试阶段的目标触发反馈评估和性能优化方向。目前，针对定向模糊测试性能优化的方法分为全局能量调度算法、自适应种子突变策略和多级优先级队列的方法。

表 1 相关工作总结与分类

工作分类	主要工作	研究方法	相关工作
测试目标的选取	测试目标选取方式和多目标优化方法	(1)目标选取方式	AFLGo[37], BEACON[49], Hawkeye[52], MC^2[72], ParmeSan[60], FishFuzz[61]等
		(2)多目标优化方法	Titan[51]，LeoFuzz[50]，Prospector[69]，AFLRUN[70]，WAFLGO[71]，SAFuzz[73]，ParmeSan[60]，FishFuzz[61]等
目标定向的预处理	增强定向引导能力度量的预处理方法	(1)基于距离最小化的方法	AFLGo[37]，Hawkeye[52]，WindRanger[56]，DAFL[62]，HyperGo[74]，ParmeSan[60]，FishFuzz[61]和 WAFLGO[71]，DirectFuzz[75]，SyzDirect[45]，G-Fuzz[46]等
		(2)基于输入可达性的方法	FuzzGuard[76]，MC^2[72]，Halo[77]，BEACON[49]，SelectFuzz[78]，TOPr[79]，SieveFuzz[53]，CONFF[80]，FGo[81]，PDGF[82]，DDGF[83]，G-Fuzz[46]，VULSEYE[142]等
		(3)基于序列导向的方法	UAFUZZ[55]，UAFL[84]，MDFuzz[85]，LOLLY[86]，Berry[54]，LTL-Fuzzer[87]，DeepGo[88]，SDFUZZ[89]，CAFL[90]等
定向模糊测试的性能优化	模糊测试循环阶段的目标触发性能提升方法	(1)全局能量调度算法	AFLGo[37]，GREYHOUND[91]，FishFuzz[61]，AFLRUN[70]，LeoFuzz[50]，Prospector[69]，SAFuzz[73]，DAFL[62]，UAFL[84]，UAFUZZ[55]，Hawkeye[52]，VCov[92]，ODDFUZZ[93]，DirectFuzz[75]等
		(2)自适应种子突变策略	Hawkeye[52], WindRanger[56], SAFuzz[73], Titan[51], UAFL[84], CONFF[80], Prospector[69], G-Fuzz[46], SyzDirect[45], GREYHOUND[91], ODDFUZZ[93]等
		(3)多级优先级队列	WindRanger[56], ODDFUZZ[93], FishFuzz[61], DirectFuzz[75], Hawkeye[52], Berry[54]，UAFL[84]，MDFuzz[85]等

4 测试目标的选取

与传统的模糊测试相比，DGF 采用了更为专注和精准的策略，不是盲目地尝试触发所有潜在的未知崩溃，而是集中资源对程序中的特定片段或函数进行深入测试。测试主要关注那些已知或高风险的代码区域，目的是在这些区域内发现、理解并修复错误，从而提升模糊测试的有效性。实施 DGF 的关键在于合理地选择测试目标，然后对不同的测试目标进行测试以验证定向模糊器的能力。如何选择合适的测试目标来验证定向灰盒模糊器的能力，是研究人员所关注的重点问题。在现有工作中，目标的选取方式通常分为 2 种：手动定义目标和自动生成目标。为了解决多个不同测试目标重复测试带来的资源消耗大问题，多目标优化方法受到关注，可以分为 2 种：种子与多目标的相关性分析的方法，以及提高单次执行的多目标覆盖能力的方法。

4.1 目标的选取方式

在 DGF 的过程中，如何选取合适的测试目标来检测其定向能力，是研究人员们所关注的重点问题。本节总结了现有工作中 2 种主要的目标选取方式：手动定义目标和自动生成目标。其中，手动定义目标依赖于测试人员的经验和专业知识，能够精确定位关键点；自动生成目标则利用算法和工具自动识别潜在目标，具有广泛覆盖和节省人力的优势。

（1）手动定义目标。手动定义测试目标适用于目标明确且依赖专业判断的测试场景，通常聚焦于复现已知崩溃的情况，依赖测试者对软件及其历史问题的深入理解来直接指定测试的焦点。例如，某个软件组件在过去的操作中发生了崩溃，测试者可以手动选择该崩溃位置或相关的代码路径作为模糊测试的直接目标[37,53]。通过这种方式，测试者可以有针对性地识别和利用已知的漏洞，利用程序分析等技术，生成能够触发特定漏洞的输入，从而实现高效的漏洞复现和分析。

首先，CVE 是 DGF 工作中最常用的目标选取指标。CVE 作为公开的漏洞和安全风险的标识系统，每个 CVE 标识对应一个具体的安全漏洞或风险[22]。大多数工作[37,45,49-56,62,69,70,72-74,78,80-81,83-87,92,94]使用已知 CVE 作为测试目标，通过提升重现对应崩溃的速度来验证方法的有效性。

其次，为了增强测试的精确性和针对性，一些

工作[52,56,71-72,78-80,82,90]使用具体的目标位置（如具体目标文件中的代码行号）作为测试目标，使测试不仅局限于触发已知的 CVE 漏洞，而是更加深入地针对特定代码段进行细化分析。

另外，还有一些特定领域的工作选择了对应领域的测试目标，如 DirectFuzz[75]使用了 RTL 硬件设计[95-96]中的具体模块实例作为目标，G-Fuzz[46]随机选取了 gVisor[97]中 50 个代码位置作为目标，ODDFUZZ[93]使用了常见 Java 库中发现的 34 个已知调用链（gadget chain）的集合作为目标。

（2）自动生成目标。自动生成目标的方法依靠自动化工具，根据实时数据动态调整测试焦点，更适合需要广泛覆盖和快速适应变化的测试环境。例如，使用 Sanitizer[98-99]等自动化工具在代码执行时实时检测并记录各种运行时错误，如内存泄漏、缓冲区溢出等；或使用深度学习模型对可能存在问题的目标进行预测。将这些结果作为 DGF 的目标，引导测试用例的生成，确保测试能够聚焦于最可能出现问题的区域。

现有的自动生成目标的 DGF 工作中，ParmeSan[60]和 FishFuzz[61]使用了 Sanitizer 检测出的潜在缺陷作为测试目标。Sanitizer 是一类运行时验证工具，可以检测出程序中的各种内存错误和未定义行为，如缓冲区溢出和使用后释放等问题[23]。ParmeSan[60]和 FishFuzz[61]将这些由 Sanitizer 检测到的潜在缺陷作为自动化测试的目标，从而确保测试可以集中在那些最有可能出现安全漏洞的代码区域；DeFuzz[100]构建了深度学习预测模型，识别潜在的易受攻击的位置作为测试目标。通过训练模型识别代码中的模式和特征，预测高风险区域并进行定向模糊测试，提升了定向模糊器对未知漏洞的定向探索能力。

4.2 多目标优化方法

DGF 通过专注于测试特定的程序位置来提高程序行为的检测效率。然而，随着软件系统规模的扩展，每次运行的 DGF 通常需要同时处理大量目标。例如，静态分析器可以报告 1,000 多个潜在目标，供开发人员在项目的单个版本中进行验证[101]。当程序中面对如此大量的目标时，一个潜在的选择是利用额外的计算资源提来换取更快的复制速度，使用单独的 CPU 核心为每个目标部署多个并发的模糊器实例[102]。然而，这对开发人员来说可能不切实际。上述的大多数工作，如[49,52,76,90]都不支持同时测试多个目标。它们的成功取决于设计更精确地分析单个目标的细粒度反馈。

因此，一些工作专注于提升多目标检测能力。AFLGo[37]、MC^2[72]和 DeFuzz[100]提供了多目标的检测支持，通过指定的多个目标，使用距离算法得到的所有目标的平均距离作为种子距离，优先考虑能够接近更多目标的种子。虽然它们尝试使用一般反馈（如平均控制流距离）同时处理多个目标，但与处理单个目标相比，它们在处理多个目标时的有效性显著降低[51]。当前定向灰盒模糊器在同时分析程序中的多个目标时经常面临可扩展性的问题，从而可能退化为目标无向方法。Huang 等人[51]在 Magma 数据集[106]中对 AFLGo 的多目标检测能力进行了测试，AFLGo 在使用多目标模糊测试重现相同数量目标时，与使用多个并行模糊器实例分别测试单个目标相比，重现崩溃所需的时间增加了 3.2 倍。

针对一般反馈的效率问题，目前在多目标优化方向上的研究工作主要包括以下 2 类：

（1）增强种子与多目标相关性的方法。这类方法通过分析种子与测试目标之间的关系，优化种子选择和资源分配，以在多目标测试中提高测试效率和目标覆盖率。Titan[51]提出了一种多目标相关性驱动的方法，识别并分类程序中各个目标之间的相关性，旨在提升种子的多目标覆盖能力，实现在少量的执行中覆盖更多的测试目标。Titan 设计了一种前提条件推断分析方法，使用静态分析推断出每个测试目标的前提条件，并基于路径条件计算各个目标之间的相关性。Titan 将程序目标之间的相关性细分为 3 种类型：条件重叠、条件冲突和条件独立。如图 4 所示，多目标案例中存在 3 个目标，其中，目标 1 与目标 2 在 a 条件下重叠，在 b 条件下冲突，目标 1 与目标 3 的条件相互独立。Titan 在条件重叠条件下，识别可能同时触及多个目标的种子，并优先调度；在条件冲突条件下，区分并独立处理这些冲突条件；在独立条件下，专注于改变不会影响触发其他目标的独立字节，从而减少种子为覆盖多个目标而产生的额外执行次数。

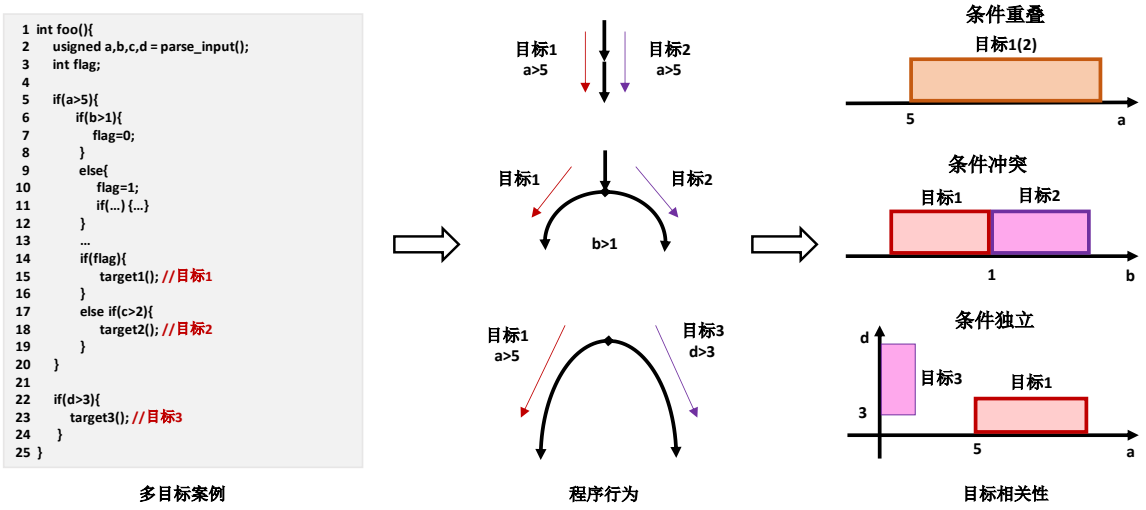

图 4 Titan 多目标相关性[51]

LeoFuzz[50]区别于 Titan 的目标间相关性分析，侧重于种子与已知目标之间的相关性。它提出了自适应调整探索（Exploration）和利用（Exploitation）阶段的方法，设计了 2 个队列，分别存储有助于达到目标的种子和增加代码覆盖的种子，根据覆盖种子的比例和代码覆盖的信息量来动态调整探索与利用阶段的转换。同时，设计了一种多目标能量调度策略，通过考虑种子与多个目标序列之间的关系，包括种子的序列覆盖率、目标序列的优先级和目标序列的全局最大覆盖率，实现细化能量分配；Prospector[69]则提出了一种专注于“聚焦目标”的多目标 DGF 方法，通过 Sanitizer 构建测试目标，并结合静态和动态优先级分析对多个目标进行排序。静态优先级通过 Louvain 算法[103]对函数调用图进行划分，结合目标密度、函数中心性和目标关联性，筛选可能更易触发漏洞的目标。动态优先级根据路径覆盖率和资源分配实时调整“聚焦目标”。在此基础上，通过探索与利用阶段的调度，优化种子选择和字节突变策略，优先测试与目标更相关的种子和字节，从而提升目标覆盖效率和漏洞触发能力。

Titan、LeoFuzz 和 Prospector 通过优化种子与目标的相关性和资源分配，显著提升了多目标模糊测试的效率和漏洞检测能力。

（2）提高单次执行的多目标覆盖能力方法。这类方法主要通过优化测试策略，使得每次执行尽可能地覆盖更多的目标。AFLRUN[70]主要通过目标路径多样性度量和无偏能量分配策略来优化测试过程。它维护了每个目标的覆盖状态图，并通过额外的映射来评估种子在不同目标上的多样化能力。无偏能量分配确保每个目标都能被公平测试，避免了测试资源集中于某些目标而忽略其他目标的情况；相对而言，WAFLGO[71]采用了一种专注于检测代码提交（commit）引入的潜在漏洞的方法。它通过识别路径前缀代码（Path-prefix Code）和数据后缀代码（Data-suffix Code），利用这些关键代码引导输入生成到达代码变更的位置，并进一步增加输入的多样性。它的优势在于其轻量级的多目标距离度量和无偏能量分配策略，通过合并同一函数中的变更点为整体目标，并计算每个目标块的距离，以确保每个目标都得到平等处理。这种方法适用于动态变化的代码库环境，能够有效应对频繁的代码变更；SAFuzz[73]则通过静态分析设计目标定向基本块，优先测试关键路径节点。结合动态目标覆盖反馈和自适应能量调度，确保测试资源集中于未覆盖或难以触发的目标。该方法还采用了分支敏感突变[104]和多臂老虎机模型[105]优化突变策略，生成更高效的多目标测试用例。

区别于 AFLRUN、WAFLGO 和 SAFuzz，ParmeSan[60]和 FishFuzz[61]采用了 Sanitizer 引导的模糊测试策略。其中，ParmeSan 通过在目标程序的中间表示上进行 Sanitizer 插桩，能够更精准地检测内存错误和未定义行为。它通过启发式剪枝减少测试目标的数量，特别是对复杂函数和程序热路径的优先测试，来提高测试效率；FishFuzz 则进一步优化了多目标模糊测试的策略，设计了一种多目标距离度量算法来计算每个种子与目标之间的独立距离。这种方法不仅能够引导模糊器触发更多的测试目标，还通过动态调整目标优先级，避免对单一目标的过度检测。当某个目标被充分测试且不太可能再发现新错误时，会降低其优先级，从而将更多测试

资源分配给尚未充分探索的目标。

总体来看，这些方法各有侧重，AFLRUN 强调路径多样性和公平能量分配，WAFLGO 注重代码变更影响和关键代码引导，SAFuzz 专注目标覆盖与突变优化策略；ParmeSan 和 FishFuzz 聚焦于 Sanitizer 驱动的精确测试，FishFuzz 进一步通过多距离度量和优先级动态调整优化目标覆盖。

4.3 小结

本节对 DGF 中测试目标的选取方法和多目标优化方案的能力分别进行了总结和分析。首先将目标选取的方式分为手动定义目标和自动生成目标 2 类，对不同选取方式的优劣势进行了总结和归纳，具体见表 2。通过分析这 2 种选取方式与其优劣势，总结为以下 3 个方面。

（1）手动定义目标和自动生成目标的目标都是为了验证 DGF 的定向检测能力，而手动定义目标关注的是已知错误的重现检测，自动生成目标关注的是对程序内潜在漏洞的定向检测。

（2）手动定义目标依赖于测试人员的经验和专业知识，能够根据任务的需求精确设定测试目标。

（3）自动生成目标使用算法和自动化工具识别潜在目标，可以系统性地大范围识别多个测试目标，并减少人为选取的偏见。

表 2 DGF 测试目标选取方式对比

测试目标选取方式	测试目标选取手段	主要优点	主要缺点	相关工作
手动定义目标	CVE 编号、具体目标位置或特殊领域测试目标	（1）精准关注已知的风险点或关键功能； （2）可以根据任务的需求设定明确的目标	（1）可能忽略未知的风险区域； （2）在大型项目或频繁变更的代码库中难以扩展	AFLGo[37]，BEACON[49]，Hawkeye[52]，MC^2[72]，ParmeSan[60]，FishFuzz[61]等
自动生成目标	Sanitizer 检测结果或深度学习预测	（1）系统性地分析整个代码库，识别出潜在的风险点； （2）减少人为选择的偏见	（1）难以根据特定的业务需求，生成具体的测试目标； （2）无法保证生成的目标一定存在漏洞，进而影响测试效率	Titan[51]，LeoFuzz[50]，Prospector[69]，AFLRUN[70]，WAFLGO[71]，SAFuzz[73]，ParmeSan[60]，FishFuzz[61]等

然后，本节将多目标优化的方法，分为种子与多目标的相关性分析方法和提高单次执行的多目标覆盖能力方法。为了更加直观地对比不同多目标优化方案的性能提升（相比 AFLGo[37]）和未知漏洞的检测能力，本节给出总结（见表 3）。通过分析这 2 类方案与其多目标检测能力，总结为以下 3 个方面。

（1）增强种子与多目标相关性的方法和提高单次执行的多目标覆盖能力方法的目标都是一致的，都是为了提升 DGF 在有限执行次数内触发更多的目标崩溃。

（2）增强种子与多目标相关性的方法关注于种子与不同目标间的联系，通过相关性分析提升种子对不同目标的触发能力。

（3）提高单次执行的多目标覆盖能力方法关注于每次执行对不同目标的覆盖程度，通过对不同目标的统一规划，提升种子的多目标覆盖率。

表 3 DGF 多目标优化方案对比

方案类型	主要方案	研究工作	主要方法	数据集	性能提升（相比 AFLGo）	未知漏洞检测数量	CVE 编号分配数量
增强种子与多目标相关性	对种子与多目标相关性进行分析，改进模糊测试策略	Titan[51]	目标间相关性分析	Magma[106]	28.7 倍	9 个	2 个
		LeoFuzz[50]	种子与目标相关性分析	手动选取的 31 个漏洞	4.63 倍	23 个	12 个
		Prospector[69]	多目标的动态排序	UniBench[107]	N/A	6 个	5 个
提高单次执行的多目标覆盖能力	以覆盖更多的目标为导向，旨在一次执行中覆	AFLRUN[70]	目标路径多样性度量	Magma[106]	1.09 倍	29 个	8 个
		WAFLGO[71]	多目标距离度量	手动选取的 30 个漏洞	9.1 倍	7 个	4 个
		SAFuzz[73]	目标定向基本块设计	UniBench[107]	16.35 倍	7 个	N/A

	盖更多目标	ParmeSan[60]	关注复杂函数和热路径	Google fuzzer-test-suite [108]	2.88 倍	N/A	N/A
		FishFuzz[61]	多目标距离度量	手动选取的 44 个测试对象	N/A	25 个	18 个

5 目标定向的预处理

在 DGF 的过程中，预处理方法直接关系到定向目标的测试效果。在现有的覆盖率引导的灰盒模糊测试的工作中，关注的重点在于如何高效地提升被测程序的覆盖率，通过实时监控程序执行过程中的路径变化，以自动化的方式快速发现程序中的错误或漏洞。然而，这种分析方式难以适用于 DGF，因为程序的覆盖率与目标的覆盖率并不等价。静态分析作为一种软件分析中漏洞检测的有效方式[109-111]，大多 DGF 工作在动态分析的同时，使用了静态分析来提取必要的信息，以指导测试的定向执行。这些工作通常会使用静态分析所得到的程序调用图、控制流图、数据流图等作为辅助，引导模糊器逐步向目标位置靠近。通过动态分析和静态分析相结合的方式，不仅可以更精确地识别并触发软件中的关键部分，还可以生成更有针对性的输入来提高测试用例的有效性，提升模糊器对特定目标的测试效率。

通过总结现有工作，本章将 DGF 中预处理阶段的优化工作分为 3 类：基于距离最小化的方法、基于输入可达性的方法和基于序列导向的方法。

5.1 基于距离最小化的方法

预处理过程中的一大主流方案是利用距离度量来引导输入的生成。基于距离的引导机制可以高效地导向模糊测试，避免了在程序的执行路径中进行盲目探索。在预处理方法的设计过程中，研究人员往往会使用控制流等信息定义距离算法，综合考虑路径长度、分支条件、特定函数调用等因素，定义一种量化种子与测试目标之间接近程度的度量，评估种子抵达目标的能力。通过这种方式，模糊器不断最小化这个度量，可以更有效地集中资源，优先执行更有可能触发目标位置或行为的种子，进而提升模糊测试的效率。

AFLGo[37]是第一个提出 DGF 概念的工作，它将检测特定目标的理念引入了基于覆盖率引导的灰盒模糊测试，并在 AFL[30]的基础上优化了种子选取，使得模糊器专注于特定的目标位置。AFLGo 在针对目标定向的预处理上，使用了静态分析技术，采用了启发式方法提升执行轨迹和测试目标之间距离较短的种子的利用率。通过构建程序调用图和过程间控制流图 ICFG，计算程序中函数间距离和基本块间距离。其中，函数间距离使用调和平均数距离，区分在多目标场景下的不同目标的距离度量。基本块间的距离使用 Dijkstra 最短路径算法[112]定义种子与目标的距离。然后，对函数间的距离和基本块间的距离进行归一化处理得到种子的最终距离。每次优先使用平均距离最小的种子优先执行，以实现对目标的定向能力。如图 5 所示，图中存在 2 个测试目标，每个节点代表一个基本块，灰色节点表示不同种子执行所覆盖的基本块，节点旁边的数字表示当前基本块到目标的距离。种子 1 和种子 2 的距离分别为（12/7+3+2）/3≈2.24 和（12/7+6/5+2/3+1）/4≈1.15，种子 2 的距离更小，在模糊测试的过程中 AFLGo 会相比种子 1 优先使用种子 2 进行测试。

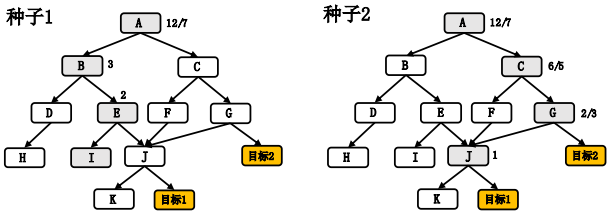

图 5 AFLGo 距离计算示例

但是，AFLGo 的基于基本块和目标间最小距离的种子选择方法准确性存在一定的不足，后续出现了一系列改进工作，主要包括以下 3 类：

（1）增强目标定向能力的距离度量算法。在增强目标定向能力的距离度量算法中，Hawkeye[52]、WindRanger[56]、DAFL[62]和 HyperGo[74]均对种子与目标的距离度量进行了优化。其中，Hawkeye 改进了 ICFG 的构建精度，利用包含关系的指针分析[113]来识别间接调用，生成更精细的控制流图。这种方法能够避免 AFLGo 方法中忽略较长路径可能导致的错误遗漏。它通过计算函数级和基本块级的距离来模拟函数之间的密切程度，从而更准确地评估种子到目标的接近程度；WindRanger 侧重于基本块的

偏差分析，针对 AFLGo 仅使用控制流信息进行距离计算的局限性，该方法结合数据流分析，识别执行过程中可能偏离目标的偏差基本块。它通过静态分析和污点分析，确定哪些字节会影响给定分支约束的数据流信息，并提出改进的距离计算方法，以更精确地引导模糊器集中测试关键路径；DAFL 则采用基于定义使用图的语义性评分算法，进一步提升了种子与目标之间距离计算的精度，在处理复杂结构的程序时表现出色。该方法还通过选择性覆盖检测方法，只收集与目标执行相关的代码覆盖信息，减少负面反馈，有效地消除了控制流图中的复杂结构（如循环），优先执行与目标位置相关性更强的输入；HyperGo 在 AFLGo 距离度量的基础上，引入了路径概率，动态评估路径到达目标的可能性。该方法根据路径中分支命中的统计概率计算路径到达目标的可能性，并将其与基本块距离结合，形成概率基础距离。将更多资源分配给路径复杂度更低（路径约束更少），且距离更短的种子，以实现更高效的漏洞触发。

（2）支持多目标同时处理能力的距离度量算法。ParmeSan[60]、FishFuzz[61]和 WAFLGO[71]针对多目标处理的需求，设计了高效的距离算法。其中，ParmeSan 通过运行时 Sanitizer 插入的检查来识别潜在漏洞作为测试目标，结合动态控制流图和数据流分析，实时调整种子与目标的距离计算，以适应程序行为的变化。该方法通过递归计算每个基本块到不同目标的距离，并根据路径的条件分支数量进行权重分配，实现了精确的多目标距离度量；FishFuzz 同样使用 Sanitizer 来生成测试目标，但其创新在于提出了一种多距离度量算法，计算种子与每个目标之间的独立距离，并通过动态调整目标优先级，减少对已触发目标的种子的关注，优先测试尚未触发的目标，从而优化测试广度，确保覆盖更多的目标；WAFLGO 则针对代码提交可能引入的漏洞，提出了一种多目标距离度量算法。通过构建支配树将同一函数内的多个变更位置合并为一个目标节点，简化了目标管理，并使用基本块间的最短路径计算合并目标的距离。在多目标测试中，该方法能够根据种子输入的执行轨迹来生成更具针对性的输入，提高测试的精度和效率。

（3）适用特定应用领域的距离度量算法。DirectFuzz[75]、SyzDirect[45]和 G-Fuzz[46]针对不同应用场景的需求，提出了专门的优化策略来提高模糊测试的精度和效率。其中，DirectFuzz 专注于硬件寄存器传输层（Register Transfer Level, RTL）设计验证。区别于 AFLGo 中基本块间的距离度量，在硬件设计场景下，该方法通过从 RTL 设计中抽取出的模块实例连接图，使用种子影响到的所有模块实例到目标实例的最短路径长度的平均值作为距离。基于图的距离度量方法更加精确地定位需要测试的硬件组件，适应不同复杂度的 RTL 设计，提供了灵活高效的硬件验证方案；SyzDirect 针对 Linux 内核的复杂性，改进了 AFLGo 的距离计算方法，结合控制流和数据流信息，并通过增加可达性分析减少误报。同时，该方法还引入基于类型的多层次类型分析（MLTA）算法[114]，通过解析间接调用生成更精确的调用图和控制流图，提高距离计算的准确性，从而优化内核模糊测试用例的生成；G-Fuzz 则针对应用级内核 gVisor[97]提出了一种专门的 go 语言 DGF 方法。它采用快速类型分析（Rapid Type Analysis, RTA）[115]，结合定制的 go-callvis[116]工具，识别 gVisor 中的间接调用。与传统的指针分析相比，该方法的调用关系分析更快更准，同时利用广度优先搜索算法替代 AFLGo 的 Dijkstra 算法，以降低计算复杂性和提高执行速度，从而更有效地适应 gVisor 的测试需求。

5.2 基于输入可达性的方法

DGF 通常是在覆盖率引导的模糊测试的基础上进行创新和优化，但覆盖率引导的方式会导致 DGF 运行过程中产生路径发散问题，使得探索无法有效抵达特定目标，导致浪费计算资源并影响测试效率。FuzzGuard[76]中统计的数据显示，AFLGo 在执行过程中约 91.7%的输入都无法到达目标，反映出现有定向灰盒模糊器在可达性上的不足。为了提升对特定目标的探索能力，研究者们针对模糊器的输入可达性进行了多方面的优化，采用了启发式的从执行反馈中收集相关信息的方法，优化种子的选择。

针对不可达路径的执行会严重影响目标定向测试的效率问题，目前在输入可达性方向上的研究工作主要包括以下 4 类：

（1）对输入的可达性进行分析预测。这类方法通过评估哪些输入能够到达目标代码路径，来优化输入选择和生成策略。FuzzGuard[76]利用深度学习技术来预测新生成输入的可达性，通过在执行目标程序之前过滤掉目标不可达的输入，从而提高整体性能。它对每个测试项目训练专用模型，使用输入执行后的可达性数据，并引入预主导节点来解决

数据不平衡问题。此外，该方法还设计了一种代表性种子选择方法，最小化采样数据的数量，优化训练时间，并且可以与其他模糊测试算法结合使用，进一步提升测试效率；MC^2[72]将模糊测试视为一个由预测模型引导的搜索问题，通过预测模型提供的输入空间信息来优化搜索策略，减少达到目标所需的输入数量。它提出了一种随机化搜索算法，根据预测模型的反馈动态调整搜索范围，以对数级查询复杂度快速接近目标，显著提高了模糊测试的效率；Halo[77]则侧重于约束输入生成，推导出能到达目标的不变条件，通过历史输入的分析，限制生成不可达的输入，采用基于距离和相似性的策略优化输入选择。

（2）对被测程序中目标无关的代码进行剪枝。针对现有的模糊器都存在路径爆炸的问题，定向灰盒模糊器旨在测试程序的特定目标，应该尽早拒绝不可达的执行路径。而类似于 AFLGo 的基于距离最小化的工作一般不考虑拒绝不可达路径，它们使用轻量级的启发式距离算法来促进模糊器的定向执行。

对被测程序中目标无关的代码进行剪枝的方法通过去除不会影响测试目标的代码路径，减少不必要的路径探索，从而提高模糊测试的效率。BEACON[49]是首个在 DGF 的预处理阶段修剪不可达路径的方法，通过静态分析程序的 ICFG，删除无法到达目标代码的基本块，并通过反向区间分析进一步剔除更多不可达路径，专注于减少测试过程中不必要的路径探索；SelectFuzz[78]进一步细化了剪枝方法，将可达路径划分为目标相关路径和不相关路径，对不相关路径进行细化修剪，作者经过统计分析发现，87.67%可达路径实际上是目标不相关的。该方法通过定义路径发散代码和数据依赖代码，使用新的距离度量和反向数据流分析，精确识别和剔除与目标无关的路径；TOPr[79]则针对一般剪枝方法无法恢复间接控制流的问题，设计了增强目标导向的剪枝策略，利用编译器提供的默认分析恢复所有直接传输的控制流，无需额外的启发式加强。然后，通过函数签名匹配，恢复和定位间接控制流，构建间接调用关系。最后，对程序中所有相关基本块进行标记，并对目标位置无关的基本块进行剪枝，通过避免对无关路径的探索。

（3）通过有效的实时反馈减少对无关路径的探索。这类方法通过动态调整测试路径，确保测试资源集中在最有可能发现漏洞的区域。SieveFuzz[53]提出了一种基于“拌绳”（Tripwiring）的优化策略，通过静态分析构建调用图和控制流图，识别所有潜在的目标可达路径，并在不可达的代码区域设置拌绳，以便实时终止不必要的测试。当新的可达路径被发现时，拌绳机制能够动态调整，添加或移除拌绳，使测试聚焦于相关路径；CONFF[80]采用了一种基于动态约束过滤和聚焦的方法，通过追踪程序执行中的所有路径约束，使用优先级系统筛选出最接近目标的关键约束，然后利用多种突变策略逐步满足这些约束，减少对无关路径的探索；FGo[81]引入了概率指数级止损策略，通过利用 ICFG 中的不可达信息，动态调整测试用例的终止概率，使得测试用例在进入不可达区段时更有可能被提前终止，从而节省计算资源并提高测试效率；PDGF[82]提出了一种前驱感知的路径搜索方法，将 DGF 的问题重新定义为在目标节点前驱区域内的路径探索。该方法通过轻量级静态分析初步识别目标前驱区域，结合动态执行中的覆盖追踪不断扩展前驱集合，使得测试有效的减少了目标无关路径的探索；DDGF[83]提出了一种动态定向的路径探索方法，将 DGF 的问题重新定义为基于路径频率分布的动态路径引导。通过 Ball-Larus 路径分析算法[117]对路径进行高效编码，结合实时路径频率分析，帮助用户识别和标记关键路径。该方法利用轻量级动态路径分析实时显示路径覆盖状态，并通过用户交互不断调整测试方向，使得模糊测试器聚焦于目标区域内的漏洞触发路径。

（4）针对特定领域需求，对输入可达性方法进行专门优化。这类方法注重结合领域特性，改进模糊测试中的输入选择策略。Wüstholz 等人[118]针对智能合约，提出了一种静态前瞻分析方法，通过分析新生成输入的路径前缀来判断其是否能到达目标代码，在线实时过滤不可达的路径前缀，以避免不必要的测试，从而节省资源；VULSEYE[142]针对智能合约，设计了一种基于状态的定向灰盒模糊器，它通过静态分析识别漏洞敏感的代码目标，利用逆向分析确定能够引发漏洞的合约状态目标，从而确保测试聚焦于可能导致漏洞的状态目标，避免无关区域探索所导致的资源浪费；G-Fuzz[46]则面向应用级内核 gVisor，通过静态分析技术构建 gVisor 的调用图和控制流图，识别与测试目标相关的代码路径，进行可达性分析，使用直接或间接调用到的系统调用集合构建测试用例，进而提升模糊测试的输入可达性。

现有研究工作中，基于输入可达性的方法可以有效地提高 DGF 的效率。本节对上述工作中前 3 类方法进行了总结（见表 4），主要归纳了它们的优化方案、改进后可达性相关的效果、相比 AFLGo 的性能提升和未知漏洞检测能力。其中，由于第 4 类研究方法针对特定领域，无法和前 3 类方法在上述指标中进行有效对比，因此在表中没有体现。

表 4 针对输入可达性的改进方法对比

优化方案	研究工作	主要方法	数据集	可达性相关的效果	性能提升（相比 AFLGo）	未知漏洞检测数量	CVE 编号分配数量
输入可达性预测	FuzzGuard[76]	深度学习模型预测	手动选取的 45 个漏洞	过滤 65.1%的输入	5.1 倍	N/A	N/A
	MC^2[72]	复杂性理论框架预测	Magma[106]	N/A	134 倍	15 个	N/A
	Halo[77]	推导目标可达的不变条件	Magma[106]	可达输入比率 57.0%	28.9 倍	10 个	N/A
目标无关代码剪枝	BEACON[49]	不可达路径剪枝	手动选取的 51 个漏洞	修剪 82.94%的执行路径	11.5 倍	8 个	10 个
	SelectFuzz[78]	不相关路径剪枝	Google fuzzer-test-suite[108]	修剪 98.04%的不相关基本块	10.69 倍	14 个	6 个
	TOPr[79]	增强目标导向的剪枝	手动选取的 24 个补丁	N/A	1.08 倍	24 个	N/A
减少无关路径探索	SieveFuzz[53]	实时终止目标不可达的测试	Magma[106]	减少 29%的测试范围	1.42 倍	N/A	N/A
	CONFF[80]	动态约束过滤和聚焦	Lava[119]	N/A	27.3 倍	N/A	N/A
	FGo[81]	概率指数级止损策略	手动选取的 4 个漏洞	N/A	1.06 倍	N/A	N/A
	PDGF[82]	前驱感知区域覆盖优化	UniBench[107]	目标可达路径的覆盖率平均提升 5.63 倍	1.34 倍	9 个	6 个
	DDGF[83]	动态路径标记聚焦关键路径	Magma[106]	N/A	N/A	4 个	2 个

5.3 基于序列导向的方法

尽管 DGF 在一些场景下表现优异，但其在处理特定触发序列的漏洞时仍然存在不足，如释放后使用（use-after-free, UAF）、双重释放（double free）等依赖特定操作顺序的漏洞，仅仅到达目标位置并不能保证漏洞的触发，而是需要满足一定的执行序列。例如 UAF 漏洞，必须先执行内存释放操作，再进行内存使用操作，才能成功引发错误[55,84]。近年来，一些研究工作采用序列导向的策略来增强模糊器对漏洞触发能力。通过分析测试目标类型及其相关数据条件，并将它们整合为序列化的约束，以序列导向的方式引导模糊器的执行。

针对执行序列对触发目标崩溃的影响，目前在序列导向方向上的研究工作主要包括以下 3 类：

（1）专门针对 UAF 漏洞进行有效检测。这类方法通过分析内存操作序列和类型状态，以提升 UAF 漏洞的检测效率。UAFUZZ[55]作为首个专门用于检测 UAF 漏洞的定向灰盒模糊器，通过动态监控程序的内存分配、释放和使用操作，生成内存事件序列，并将这些序列与已知的 UAF 漏洞模式进行比对，生成针对性的测试输入。它还采用序列相似度评估机制，优先选择与已知 UAF 序列高度匹配的种子进行测试，持续优化测试策略；与 UAFUZZ 不同，UAFL[84]采用类型状态分析作为主导，关注于追踪程序中的内存操作序列（malloc->free->use），以此为基础来生成能够触发 UAF 漏洞的测试用例。该方法利用类型状态分析（typestate analysis）来识别可能违反类型状态属性的操作序列，通过追踪在程序执行过程中状态的变化，构建映射对象状态转换模型，从而精确地捕捉到潜在的 UAF 漏洞触发点。如图 6，我们给出了 CVE-2018-20623 的 UAF 漏洞简化案例的序列引导流程。使用 UAFL 的类型状态分析方法对代码进行解析，得到漏洞所需的操作顺序为 4->7->10->14，图中每个节点对应着代码的行号。在执行序列的引导下，UAFL 生成测试用例，逐步向序列靠近并触发指定漏洞；MDFuzz[85]则采用多层次的处理方法，通过静态分析识别内存操作的关键位置，并计算各基本块到这些目标的距离，结合概率权重的种子选

择策略，动态调整测试过程中的种子选择和突变，迅速发现了二进制代码中的复杂 UAF 漏洞。

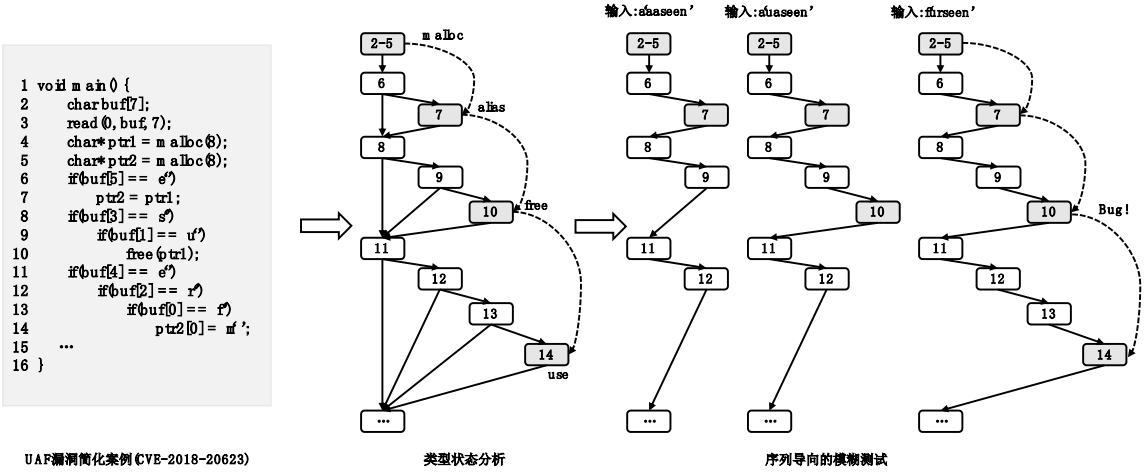

图 6 UAFL 序列引导流程[84]

（2）仅关注于测试目标相关的执行序列，不区分目标漏洞类型。聚焦特定执行路径的优化策略旨在关注特定的目标执行序列，而不考虑漏洞的具体类型。LOLLY[86]作为首个序列导向的灰盒模糊器，通过插桩技术监控程序运行时的目标基本块覆盖情况，利用共享内存记录执行轨迹，并根据种子的序列覆盖率动态分配能量，从而生成高质量的测试输入；为解决 LOLLY 在随机突变下难以覆盖复杂路径的问题，Berry[54]进一步提出了一种更细致的序列导向方法。该方法通过构建控制流图和支配树，扩展了原有的目标序列，称为增强目标序列（Enhanced Target Sequence, ETS），确保了在到达目标序列的任何一点之前，必须先访问这些节点。Berry 可以在相同于 LOLLY 的场景下，进一步提升了模糊测试的有效性与效率。

LTL-Fuzzer[87]采用线性时序逻辑（Linear-time Temporal Logic, LTL）来引导输入生成，通过比对程序行为和预定义的时序逻辑约束，确保生成的输入序列能够探索可能违反 LTL 属性的执行路径。它利用 Büchi 自动机[120]监测程序状态和事件序列，不仅能发现简单的内存错误，还能揭示复杂的基于事件顺序的漏洞；DeepGo[88]则通过结合历史路径信息和深度学习模型，设计了序列化导向的方法，预测未来的路径转换，以优化路径选择。为了预测未来可能的路径转换及其对应的奖励，该方法设计了一种虚拟集成环境（Virtual Ensemble Environment, VEE），利用深度神经网络模拟和预测路径转换过程。该方法还通过强化学习的方法，进一步开发了一种针对模糊测试的强化学习模型，根据历史和预测的路径转换信息，生成具有更高序列奖励的路径转换序列，提升模糊器触发目标的效率；SDFUZZ[89]提出了一种基于目标状态驱动的方法，利用崩溃转储和 SVF 工具[121]的静态分析的结果提取漏洞触发的特定执行序列。它通过双维度反馈机制（目标状态反馈和改进的距离度量反馈），评估当前测试用例与目标状态的相似性和函数调用的可能性，主动引导测试朝着最有可能触发漏洞的方向进行。

（3）以不同漏洞类型为导向，支持多种类型的漏洞定向检测。CAFL[90]提出了约束引导的方法，基于漏洞类型的序列化对模糊器进行引导。该方法设计了一种利用崩溃转储和补丁日志自动生成约束的方法，生成具体的目标和其数据条件，形成一系列需要满足的约束序列。通过上述的信息和静态分析的结果，该方法提出了一种综合考虑控制流和数据流信息的距离算法，使其有序且高效地触发目标漏洞，其中，控制流信息体现种子与目标的接近程度，数据流信息体现种子满足目标条件的能力。CAFL 能够检测包括 use-after-free、double-free、use-of-uninitialized-value、stack-buffer-overflow、heap-buffer-overflow、assertion-failure 以及 divide-by-zero 共 7 种类型的错误。

5.4 小结

本节总结的以上 3 种目标定向的预处理方法概括了目前 DGF 技术的预处理改进方案，对每一种技术的优缺点和性能进行了总结和归纳，具体见表 5。通过分析这 3 种预处理方法与其优劣性，可以将这 3 种技术的研究总结为以下 4 个方面。

（1）DGF 的总目标是快速触发指定崩溃。基于距离最小化和输入可达性的方法的目的都是为

了快速抵达目标，基于序列导向的方法目的是按特定的执行序列触发指定崩溃。

（2）基于距离最小化的方法使用启发式的度量方法，引导模糊器向目标靠近。主要思想是离目标越近的执行，越有可能抵达并触发目标崩溃。

（3）基于输入可达性的方法对种子的可达性进行了划分，优先执行更有机会抵达目标的输入。主要思想是减少模糊测试中大量的目标无关输入，提升输入的可达性。

（4）基于序列导向的方法对目标崩溃的执行序列进行分析，引导模糊器按顺序执行序列并触发崩溃。主要思想是使用目标相关的执行序列约束模糊器，避免可达但不可触发的执行带来的时间和资源损耗。

表 5 DGF 预处理阶段优化方案

方案类型	主要方案	代表性工作	主要优点	主要缺点
基于距离最小化的方法	增强目标定向能力	Hawkeye[52]等	（1）精准的目标导向，快速抵达目标；（2）实时的反馈机制，动态调整测试的方向	（1）搜索无关路径带来额外开销；（2）抵达目标但不一定能有效触发崩溃
	支持多目标同时处理	ParmeSan[60]等		
	适用特定应用领域	DirectFuzz[75]等		
基于输入可达性的方法	输入可达性预测	FuzzGuard[76]等	（1）提升目标相关代码的覆盖率；（2）降低不可达路径带来的性能开销	（1）可达性分析成本较高；（2）抵达目标但不一定能有效触发崩溃
	目标无关代码剪枝	BEACON[49]等		
	减少无关路径探索	SieveFuzz[53]等		
	特定领域可达性优化	G-Fuzz[46]等		
基于序列导向的方法	针对 UAF 漏洞检测	UAFUZZ[55]等	（1）对相同类型的漏洞，复现能力更强；（2）适用于复杂漏洞，确保必要节点被充分测试	（1）无法高效应对已知漏洞的未知执行路径；（2）不同类型的漏洞检测可能需要额外人工开发
	关注目标相关的执行序列	LOLLY[86]等		
	以不同漏洞类型为导向	CAFL[90]等		

6 定向模糊测试的性能优化

在模糊测试的执行阶段，定向灰盒模糊器通常基于测试目标选取和预处理阶段的分析结果，用于调整和优化测试流程。DGF 在模糊测试阶段主要沿用了覆盖率引导的策略，通过动态追踪代码执行路径和状态变化，增加代码覆盖率，以实现更快的目标抵达速度，提高测试的效率和精准度。

在定向模糊试循环的过程中，有 3 个核心模块（见图 2）。

（1）能量调度。定向模糊测试循环中，会始终维护着一个种子队列，每次选取队列的第一个种子进行测试。能量调度算法决定了种子的突变次数，对更有可能触发目标崩溃的种子给予更大的能量。

（2）种子突变。对于精心设计的被测程序，随机输入的种子很难触发程序的崩溃。种子突变策略的设计目标，是对未触发崩溃的种子进行改变（如字节翻转、字节拼接等），试图更有效地触发目标崩溃。

（3）优先级队列排序。突变后的种子会被作为测试用例执行程序，对于仍未触发目标崩溃的种子，需要进入定向模糊测试循环进一步测试。优先级队列通过评估种子触发崩溃的能力，将更优质的种子赋予更高的优先级，在后续的过程中优先执行，提升有效种子的利用率。

综上所述，这 3 个核心模块是决定定向模糊测试执行阶段触发目标崩溃能力的关键。本章将从现有 DGF 工作中的全局能量调度算法、自适应种子突变策略和多级优先级队列这 3 个角度进行归纳和总结。

6.1 全局能量调度算法

在模糊测试过程中，能量调度决定了不同种子所分配的系统资源，其目的是优化测试过程，确保有限的资源能够高效地使用，以最大化发现软件缺陷的效率。这一过程通常涵盖探索（Exploration）和利用（Exploitation）两个阶段，每个阶段针对的目标和策略有所不同。

• 探索阶段：目的是广泛地搜索测试空间以发现新的、未被探索的代码路径。在这一阶段，模糊器会尽可能地生成多样化的输入数据，以覆盖尽可能多的程序行为和状态。能量调度在此阶段会更加

倾向于均匀地分配资源给各种潜在的输入或种子，以避免过早地聚焦于某些特定的路径或输入，从而错过其他重要的测试区域。

• 利用阶段：一旦在探索阶段发现了值得进一步探索的路径，模糊测试便进入利用阶段。在这一阶段，能量调度的重点转变为深入挖掘已知的有潜力路径，提升发现漏洞的能力。能量调度策略会倾向于将更多的资源集中在那些已经显示出高潜力的种子上，或是那些能触发罕见或复杂代码路径的种子上。

在现有的 DGF 工作中，能量调度算法通常采用动态调整机制，根据实时反馈调整资源分配策略。全局能量调度算法的优化主要包括以下 2 类：

（1）对探索和利用阶段的能量分配进行灵活调整。通过调整能量分配策略来优化探索和利用阶段的测试效果，不同方法在能量分配的灵活性和优先级上各有侧重。AFLGo[37]引入了模拟退火算法（Simulated Annealing, SA）[122]来优化种子的能量分配，从而更有效地触发目标代码位置。模拟退火是一种通用的概率优化算法[37]，其灵感来源于材料科学中的退火过程，通过控制温度逐步降低来找到材料的最低能态，最终目的是向全局最优解渐近收敛。在模糊测试开始时，AFLGo 将相同的能量分配给距离不同的种子。随着时间的推移，越有希望触发目标代码位置的种子被分配越来越多的能量。AFLGo 通过定义一个时间阈值，规划退火过程是否经历了足够的探索阶段，判断其是否进入利用阶段。AFLGo 使用模拟退火算法，将更多的时效开销集中在更有触发目标崩溃的输入上，主要优化那些已经接近目标位置的种子的变异，以细化探索可能导致程序错误的具体输入。后续的很多 DGF 相关工作[45-46,50,54,56,71,78,82,86,87]也沿用了 AFLGo 中基于模拟退火的全局能量调度算法。

除了基于模拟退火的调度算法，一些工作还采用了不同的策略用于不同阶段的能量调度。GREYHOUND[91]采用成本函数和粒子群优化（Particle Swarm Optimization, PSO）策略，通过定义多个成本函数（如漏洞发现数量、状态转换数、测试迭代时间等）来引导能量分配，使得测试过程更聚焦于可能存在问题的协议层或状态；FishFuzz[61]则引入多距离度量和动态目标排序策略，通过两种探索方式在函数间和函数内最大化目标触发，在利用阶段则根据目标的优先级来分配种子能量。与 GREYHOUND 不同，FishFuzz 侧重于对目标之间的距离测量和动态排序来进行灵活调整，重点在于最大化目标的覆盖和种子的有效利用；AFLRUN[70]更加注重平衡初期的能量分配，而不依赖复杂的距离度量和排序策略。该方法提出的无偏能量分配策略相对简单，为每个目标代码块分配相同的初始权重，保证各目标在测试初期得到均等对待。其能量分配机制通过将目标的权重传播给种子，依据种子与目标的距离和关键性进行动态调整。

LeoFuzz[50]采用了一种自适应调整的能量分配策略，包括探索与利用阶段的协调 CEE（coordinate exploration and exploitation stages）和多级能量调度策略 MES（Multiple Energy Scheduling）。不同于上述方法通过动态调整阈值和分配能量的多级策略，该方法专注于如何在不同阶段灵活切换和调整能量，初期优先高优先级目标，但会随着测试的深入逐渐平衡到低覆盖度的目标，实现了更全面的多目标处理；Prospector[69]在探索阶段通过静态和动态目标优先级分析，优先选择离聚焦目标最近的种子，并利用路径覆盖信息优化函数内外的关键路径探索；在利用阶段则集中资源触发已识别目标，选择最快到达目标的种子，并通过突变高相关性的字节生成多样化测试用例；SAFuzz[73]在探索阶段侧重于发现新路径和未覆盖目标，通过距离、目标定向基本块的数量以及目标未覆盖程度动态分配能量，快速接近更多目标。在利用阶段，注重对已覆盖目标路径的深入挖掘，优先为触发目标的种子分配资源，并结合目标覆盖反馈减少对已充分探索路径的能量浪费。

（2）根据种子的质量，定义种子的能量，决定哪些种子应该更频繁地被突变。DAFL[62]和 UAFL[84]通过分析程序的静态特征来优化种子的能量分配。DAFL 利用定义使用图计算语义相关性评分，以评估种子与目标之间的接近程度，从而调整种子的突变频率，确保能量的精准分配；UAFL 则专注于检测 UAF 漏洞，通过静态类型状态分析识别关键操作序列，将更多能量分配给覆盖这些序列的种子，以增加触发漏洞的机会。

UAFUZZ[55]和 Hawkeye[52]，注重利用种子的执行路径和相似性来动态调整能量分配。UAFUZZ 结合前缀目标相似性、种子到目标路径的距离和边缘覆盖度量，综合评估种子的质量，并根据这些指标灵活调整能量分配，以更有效地检测复杂的 UAF 漏洞；Hawkeye 则通过计算基本块距离和覆盖函数

相似度，优先给更接近目标的种子分配能量。

VCov[92]、ODDFUZZ[93]和 DirectFuzz[75]，侧重于基于距离和覆盖率的信息进行能量调度。VCov 动态分配资源，特别是当种子触发新的控制流覆盖时，增加资源投入以进一步探索相关路径；ODDFUZZ 结合种子距离和调用链覆盖率，利用 AFLGo 的距离算法来靠近目标，同时优先处理覆盖更多分支的种子，平衡了距离和覆盖之间的关系；DirectFuzz 通过评估输入与目标模块之间的层级和连接距离来分配能量，同时采用随机输入调度机制避免测试陷入局部最小值，确保测试过程的多样性和覆盖广度。

6.2 启发式种子突变策略

种子突变是通过对输入数据的有意修改来诱发软件错误或崩溃的手段。种子突变的方法通常包括对原始种子进行系统性的、随机的或有规则的改动，以探索软件的不同执行路径。此外，种子突变还能帮助开发团队理解软件在面对非标准或意外输入时的行为表现，从而优化错误处理和输入验证机制，增强软件的鲁棒性和安全性。

具体来说，在模糊测试中常用的种子突变策略主要包括以下 6 种方式[30]：

- 比特翻转（Bit-flips）：改变输入数据的单个比特，例如将 0 变为 1 或将 1 变为 0。
- 简单算数运算（Simple arithmetic）：对输入数据执行基本的算术运算，例如加减一定数值。
- 覆盖（Over-writing）：使用新的数据覆盖选定的数据。
- 插入（Inserting）：在输入数据中加入新的字节或数据块。
- 删除（Deleting）：删除输入数据中的一部分。
- 拼接（Splice）：将两个或多个种子输入数据合并或重组，形成新的输入。

但是这种较为随机的普适性突变策略，可能会生成大量非法或低效的测试用例，尤其是在 DGF 这种具有特定测试目标的场景下。这不仅会导致消耗大量的计算资源，还可能增加找到实际漏洞的时间，进而影响模糊测试的效率。由此，以下的 DGF 工作在种子突变策略上进行了改进，本节对不同的启发式种子突变策略分为以下 3 类。

（1）对探索和利用阶段分别进行突变策略优化。探索和利用阶段的突变策略优化通过在不同测试阶段采用不同的突变方法来提高模糊测试的效率和漏洞检测能力。具体来说，Hawkeye[52]提出了一种自适应种子突变方法，根据测试阶段和种子特性灵活选择粗粒度和细粒度突变策略。在探索阶段，该方法使用粗粒度突变（如 Mixed havoc、Semantic mutation 和 Splice），以便快速扩展搜索空间，让种子更快靠近目标。在利用阶段，则采用细粒度突变（如 Bit/byte Flippings 和 Arithmetics on Some Bytes），旨在微调种子以触发潜在的目标崩溃；WindRanger[56]在探索阶段标记所有与约束变量相关的输入字节为高优先级，而在利用阶段，则专注于识别偏移基本块约束变量相关的输入字节。无论在哪个阶段，高优先级字节在随机突变期间都有更大的突变机会。此外，该方法还特别处理连续输入字节的情况，如果这些字节共享相同的值，它会尝试替换为约束的其他比较操作数，以更有效地探索和开发目标路径；SAFuzz[73]在探索阶段注重路径的广度，通过分支敏感分析和多臂老虎机模型生成多样化的测试用例以覆盖新路径。在利用阶段则聚焦路径深度，优先突变关键字节以触发目标漏洞，并动态调整突变操作以提升测试效率。

（2）不区分执行阶段改进种子突变策略。一些工作在探索和利用阶段使用统一的改进突变策略，提升模糊测试效果。Titan[51]利用种子相关性来突变种子中与不同目标间独立变量相关的多个字节，基于推理的污染分析动态推断可能影响独立相关性变量的输入字节[27,104,123]，逐步改变这些字节，使模糊器能以更少的突变接近更多目标；UAFL[84]则提出了一种基于信息流的突变策略，通过分析每个输入字节与程序变量间的信息流强度来确定其对程序状态的影响，归一化计算各字节的突变概率，信息流强度越高的字节变异频率越高，从而提高对关键区域的测试覆盖率；CONFF[80]采用策略性变异方法，确定路径约束与种子输入字节的映射关系后，快速满足路径约束。该方法将路径约束划分为双路径和多路径约束，并根据数据条件类型（fixed-to-fixed、fixed-to-mapped、mapped-to-mapped）应用相应的突变策略，如直接复制、二进制枚举和单点变异，以提高测试效率；Prospector[69]在探索和利用阶段，根据种子每个字节触发新路径或到达目标的能力对字节进行评分，并采用 AFLChurn[124]的策略，优先突变相关性较高的字节。

（3）针对特殊领域的不同需求，设计种子突变策略。在不同领域下的模糊测试需求有所不同，一些工作针对特定领域的需求，设立了不同的启发

式突变策略。G-Fuzz[46]使用推断的系统调用来优化种子生成和突变过程，结合静态分析和专家知识，动态调整系统调用的选择概率，以减少误报的影响，并根据系统调用间的依赖关系调整插入顺序，确保生成语义上正确的输入；SyzDirect[45]从系统调用和相关参数两个维度优化突变策略，确保突变生成的测试用例既包含入口系统调用，也遵循模板中定义的参数约束条件，从而准确定位 Linux 内核中的目标位置；GREYHOUND[91]利用 Wi-Fi 协议的状态机模型来推测设备状态，并生成和突变数据包，既发送格式良好的数据包，也有意制造协议错误，以诱导设备触发非标准行为或崩溃，暴露潜在的安全漏洞；ODDFUZZ[93]则采用步进种子突变方式，利用 JQF 模糊框架[125]将结构化输入映射为无类型化参数，在字节级别进行突变，并通过反馈机制根据覆盖率和执行路径数据指导突变过程，使每次突变都基于先前测试结果优化，增强探索新代码路径的能力。

6.3 多级优先级队列

模糊器经过种子突变后，会产生大量输入进入程序执行阶段，根据执行的结果判断种子触发目标崩溃的能力，如何有效地对这些种子进行队列排序优化，是优先级队列的目标。优先级队列通常会考虑多个因素，包括种子的历史行为、与测试目标的距离、测试目标可达性等。通过这种方式，模糊器不仅可以让具有高潜力的种子优先执行，还能有效地优化测试过程，避免资源浪费在无效输入上。

大部分 DGF 工作都使用了所提出的优化方法，对种子的优先级队列进行了单级队列排序，如[37,62]等，优先考虑最有可能触发目标的种子。单级队列的方式简化了种子管理流程，使得模糊器可以快速并直接地将测试资源集中到最有潜力的输入上，从而提高测试的有效性。

也有一些工作，在优先级队列的设定上，使用了多级队列的概念，以进一步细化种子的管理和选择过程。多级队列系统将种子根据其潜在价值和预期的效果分层分类，根据当前测试阶段的需求动态调整不同种子的优先级，有效地平衡不同阶段的需求，从而使得资源分配更为合理，提高测试的全面性和深入性。本节对不同的多级优先级队列方法分为以下 2 类。

（1）使用两级优先级队列来提升有效种子的利用率。WindRanger[56]通过对每个偏差基本块（DBB）生成一个可以覆盖该 DBB 的种子列表，并根据种子到 DBB 的距离对列表进行升序排序，将最优的种子放入优先级较高的队列中，其余种子放入低级队列，以确保最有可能触发新路径的种子优先被选中；ODDFUZZ[93]采用类似的策略，将距离相同但覆盖范围更广的种子放入高优先级队列，使得优质种子在突变过程中被优先使用，从而提高测试效率。

FishFuzz[61]通过设计队列剔除算法，动态调整目标的优先级，优先处理那些尚未被覆盖或探索较少的目标。这个策略根据目标探索状态的实时分析，动态调整测试资源分配，确保测试重点集中在最有潜力触发新漏洞的区域；DirectFuzz[75]针对 RTL 设计中的特定实例，改进了输入队列管理，创建了一个额外的优先级队列来存储涉及多路选择信号的测试输入。该方法优先从高优先级队列中选择输入，以便更快地覆盖目标模块实例，只有在高优先级队列为空时，才按照 FIFO 顺序使用常规队列中的输入。

（2）使用三级优先级队列来提升有效种子的利用率。Hawkeye[52]根据种子的潜在价值和与目标区域的相关性，优化种子的处理顺序和测试效率。它将覆盖新轨迹、与目标种子相似度更大、或能覆盖目标函数的种子放入一级队列，因为这些种子在突变后最有可能触发目标崩溃。将没有这些特性但为新生成的种子放入二级队列，突变后的原种子则被放入三级队列；Berry[54]根据种子的相似度和覆盖率进行分级，将相似度大于特定阈值且带来新覆盖的种子放入一级队列，相似度大于阈值或带来新覆盖的种子放入二级队列，其余的种子放入三级队列；UAFL[84]则根据种子的得分将其分类到不同的队列中，覆盖更多目标 UAF 序列的种子被放入一级队列，覆盖新的控制流边缘的种子被放入二级队列，其余种子被放入三级队列；MDFuzz[85]引入了基于概率的多级种子队列，针对 UAF 漏洞检测，优先选择包含内存分配和释放操作的种子，并将其放入最高级队列，覆盖内存分配的种子放入二级队列，其他种子则放入最低级队列。该方法通过概率分配增加高优先级队列的选择机会，同时确保低优先级队列不会被完全忽视，以防止测试过程中的饥饿现象。

6.4 小结

本节介绍了 DGF 在模糊测试阶段的 3 个主要流程的优化方法，并总结了现有 DGF 技术在这一阶段的性能提升方案和优缺点，具体见表 6。通过

对这 3 个阶段的优化方法进行分析，可以将这些研究总结为以下 4 个方面。

（1）在模糊测试阶段的全局能量调度算法、启发式种子突变策略和多级优先级队的优化方案的目标都是一致的，都是为了提升 DGF 发现目标漏洞的效率。

（2）全局能量调度算法通过全局的规划，为每个种子分配合适的突变能量，并将更多的能量集中于那些更有可能触发目标崩溃的种子，提升有效种子的利用率。

（3）启发式种子突变策略专注于调整和选择测试种子，通过分析种子的行为特征和历史数据来指导种子的突变过程，提升种子突变后触发目标崩溃的能力。

（4）多级优先级队列通过层次化管理种子，确保高优先级的种子获得充分的资源和关注，同时也保证低优先级的种子不被忽略，实现对测试资源的有效分配。

表 6 DGF 模糊测试阶段优化方案

方案类型	主要方案	代表性工作	主要优点	主要缺点
全局能量调度算法	对探索和利用阶段的能量分配进行调整	AFLGo[37]等	（1）避免能量调度陷入局部最优； （2）灵活实时调配调度资源	（1）算法复杂，需要额外的计算资源； （2）依赖算法的准确性
	根据种子质量分配能量	DAFL[62]等		
启发式种子突变策略	对探索和利用阶段分别进行突变策略优化	Hawkeye[52]等	（1）更有针对性地发现深层次的漏洞； （2）提升有效种子对目标的覆盖	（1）可能会忽略简单突变产生的有效种子； （2）泛化能力有限，部分启发式策略针对特定领域
	不区分执行阶段改进突变策略	Titan[51]等		
	特定领域突变策略优化	G-Fuzz[46]等		
多级优先级队列	两级优先级队列	WindRanger[56]等	（1）合理分配资源到最有可能触发漏洞的种子上； （2）细粒度地控制种子队列	（1）可能优化过度，忽略广泛性测试的重要性； （2）可能导致对低优先级种子的不公平调度
	三级优先级队列	Berry[54]等		

7 测试对象的获取

DGF 的测试对象是指包含测试目标的被测程序。研究人员通常基于软件的特定错误历史、代码复杂性，以及先前的测试结果等情况，决定是否将其作为测试对象。不同的研究团队和研究工作，会根据其具体需求选择不同的测试对象。本章收集了相关研究工作选取的独立测试对象和基准测试数据集，并进行了统计和分析。旨在为该领域的研究人员和实践者提供实用的参考，帮助他们在未来的工作中更加有效地选择测试对象。

7.1 独立测试对象

本节收集了现有 DGF 研究工作中评估的独立测试对象，根据测试对象的被使用频次进行了降序排序，如表 7 所示。由于 DGF 的应用范围非常广泛，不同的工作针对的场景有所不同，因此本节仅列出常用的独立测试对象。表 7 中分别统计了独立测试对象名称、功能描述、被选用次数、项目来源和对应的相关工作。从表 7 中可以看出，在现有的工作中，GNU 的二进制工具集 Binutils 被选做测试对象的次数最多，其次是 Flash 输出库 LibMing。此外，用于图形处理的 JasPer、libpng、LibTIFF、libjpeg、GIFLIB、ImageMagick，用于嵌入式系统和物联网设备的轻量级 JavaScript 引擎 MJS，用于程序压缩和归档的 lrzip，用于 XML 解析的 libxml2、XmlLINT，用于汇编的 NASM，用于安全通信处理的 OpenSSL，也被常用于 DGF 的独立测试对象。

表 7 常用独立测试对象的使用情况统计

独立测试对象名称	功能描述	被选用次数	项目来源	相关工作
Binutils	二进制工具集	16	https://www.gnu.org/software/binutils/	AFLGo[37]，BEACON[49]，LeoFuzz[50]，Hawkeye[52]，UAFUZZ[55]，WindRanger[56]，ParmeSan[60]，FishFuzz[61]，DAFL[62]，SelectFuzz[78]，TOPr[79]，CONFF[80]，UAFL[84]，MDFuzz[85]，DeepGo[88]，VCov[92]
LibMing	Flash 处理	14	https://github.com/libming/libming	AFLGo[37]，BEACON[49]，WindRanger[56]，DAFL[62]，FuzzGuard[76]，SelectFuzz[78]，TOPr[79]，CONFF[80]，FGo[81]，MDFuzz[85]，LOLLY[86]，DeepGo[88]，CAFL[90]，DeFuzz[100]
mJS	C/C++ 的嵌入式 Java Script 引擎	8	https://github.com/cesanta/mjs	LeoFuzz[50]，Hawkeye[52]，SieveFuzz[53]，UAFUZZ[55]，TOPr[79]，UAFL[84]，CAFL[90]，VCov[92]
JasPer	JPEG-2000 图像格式编解码	7	https://jasper-software.github.io/jasper/	SieveFuzz[53]，FishFuzz[61]，FuzzGuard[76]，TOPr[79]，CONFF[80]，MDFuzz[85]，CAFL[90]
lrzip	大型文件压缩	7	https://github.com/ckolivas/lrzip	BEACON[49]，UAFUZZ[55]，DAFL[62]，SelectFuzz[78]，TOPr[79]，UAFL[84]，DeepGo[88]
libxml2	XML 文档解析	6	https://github.com/GNOME/libxml2	AFLGo[37]，BEACON[49]，DAFL[62]，FuzzGuard[76]，TOPr[79]，TargetFuzz[94]
libpng	PNG 格式图像处理	6	http://www.libpng.org/pub/png/libpng.html	AFLGo[37]，BEACON[49]，Hawkeye[52]，DeepGo[88]，VCov[92]，TargetFuzz[94]
LibTIFF	TIFF 格式图像处理	5	http://www.libtiff.org/	LeoFuzz[50]，FishFuzz[61]，FuzzGuard[76]，CAFL[90]，TargetFuzz[94]
libjpeg	JPEG 格式图像压缩和解压	4	https://libjpeg.sourceforge.net/	BEACON[49]，Hawkeye[52]，DAFL[62]，VCov[92]
NASM	x86 架构的汇编器	4	https://www.nasm.us/	UAFUZZ[55]，FishFuzz[61]，CONFF[80]，UAFL[84]
GIFLIB	GIF 格式图像处理	4	https://giflib.sourceforge.net/	UAFUZZ[55]，FishFuzz[61]，TOPr[79]，DeFuzz[100]
OpenSSL	安全通信软件包	3	https://www.openssl.org/	ParmeSan[60]，LTL-Fuzzer[87]，TargetFuzz[94]
ImageMagick	图形处理工具	3	https://www.imagemagick.org/	FuzzGuard[76]，UAFL[84]，CAFL[90]
XmlLINT	XML 文档验证和格式化	3	https://xmllint.com/	FishFuzz[61]，SelectFuzz[78]，DeepGo[88]

7.2 基准测试数据集

除了指定特定的项目作为测试对象，也有一些工作[106-108,119,126]致力于构建一个可用于模糊测试的统一标准数据集。在现有的 DGF 研究工作中，主要使用的数据集有以下 5 个（见表 8）。

Google fuzzer-test-suite[108]是一套专为模糊测试引擎设计的基准测试集，用于在软件中发现安全漏洞和错误。这些测试对象来源于存在已知且具挑战性的错误、难以找到的代码路径或其他特殊情况的实际库，这使它们对于测试这些定向模糊测试工具的有效性非常有用。

Magma[106]数据集是一种用于精确评估和比较不同模糊测试工具性能的基准测试集，它通过在 7 个广泛使用的开源库和应用程序中人工重新引入 138 个已知的、真实的安全相关错误，创建了一个复杂的测试环境。每个错误都伴随着一种轻量级的监测机制，这些机制不仅能检测到错误是否被定向模糊器抵达，还能检测到错误被触发的具体情形。

UniBench[107]来自开源的模糊测试评估平台 UNIFUZZ[130]，专为模糊测试评估设计并实现了一个包含 20 个不同程序的数据集。UniBench 数据集中的程序支持 6 种不同的输入格式，包括图像、声音、视频、文本、二进制和网络相关输入。这些

多样化的输入格式为DGF提供了广泛的测试场景，可以有效地揭示不同类型输入下程序的漏洞触发规则。

Google OSS-Fuzz[126]是安全关键库和其他开源项目的连续测试平台。OSS-Fuzz以完全自动化的方式定期检出已注册的项目，对它们进行模糊测试。Bug报告会自动提交给项目维护者，一旦Bug被修补就会关闭。在新项目上线期间，维护者为OSS-Fuzz提供构建脚本并实现一个或多个测试驱动程序。在Google的服务器上，OSS-Fuzz每天处理十万亿级别（10^{13}）的输入[127]，截至2023年8月，OSS-Fuzz已帮助1,000个项目中[128]识别并修复了10,000多个漏洞和36,000多个错误[129]。

LAVA[119]数据集旨在帮助开发和测试寻找软件漏洞的工具，提供了大量的已知漏洞数据。LAVA通过自动向程序代码中注入数百万个现实合成的漏洞，这些漏洞深埋于程序中，且通过真实输入触发，为测试漏洞检测工具提供了真实的测试场景。这种方法有助于评估这些工具的有效性，因为它提供了一个具有已知变量的控制环境——即人为添加的漏洞，允许开发者准确评估工具在漏报率和误报率方面的性能。LAVA提供了详细的崩溃信息，包括导致崩溃的程序位置和导致崩溃的相关数据信息。

表8 常用基准测试数据集的使用情况统计

数据集名称	包含漏洞数量	被选用次数	项目来源	相关工作
Google fuzzer-test-suite	24	7	https://github.com/google/fuzzer-test-suite	Hawkeye[52], WindRanger[56], ParmeSan[60], MC^2[72], SelectFuzz[78], UAFL[84], VCov[92]
Magma	138	7	https://hexhive.epfl.ch/magma/	Titan[51], SieveFuzz[53], AFLRUN[70], MC^2[72], Halo[77], DDGF[83], TargetFuzz[94]
UniBench	20	6	https://github.com/unifuzz/unibench	WindRanger[56], Prospector[69], SAFuzz[73], HyperGo[74], PDGF[82], DeepGo[88]
Google OSS-Fuzz	持续集成	3	https://github.com/google/oss-fuzz	AFLGo[37], AFLRUN[70], TargetFuzz[94]
LAVA	百万级	3	https://github.com/panda-re/lava	Berry[54], CONFF[80], CAFL[90]

7.3 小结

本章对DGF中测试对象的获取方式进行了总结，归纳了现有工作实验所选取的测试对象，具体分为独立测试对象和基准测试数据集。针对独立测试对象，我们统计了现有工作中常用的独立测试对象信息，包括测试对象的名称、功能描述、被选用次数、项目来源和对应的相关工作；针对基准测试数据集，我们统计了工作中使用到的相关数据集信息，包括数据集名称、包含漏洞数量、被选用次数、项目来源和对应的相关工作。

8 未来研究展望

DGF技术的推出，为具有特定目标模糊测试需求的用户提供了有效的解决方案。本文从测试目标选取、目标定向的预处理和定向模糊测试的性能优化这3个方面，对DGF技术的研究现状进行了总结和归纳。但是，如何从上述3个方面更有效地提升DGF的能力，需要进一步的研究。

（1）测试目标的选取：如何选择合适的测试目标是DGF相关研究要解决的首要问题。选择合适的测试目标，对不同测试目标进行统一的处理，提升测试效能，增强模糊器验证不同漏洞的能力，依然是DGF技术应用研究的重要研究方向。具体的研究方向包括：目标优先级的动态调整、多目标间的权衡与并行测试机制等。

（2）目标定向的预处理方法：DGF区别于CGF最显著的特点就是其定向探索的能力，如何实现对测试目标有针对性的模糊测试进行探索，是DGF研究中最核心的研究问题，也是后续研究中最关键的研究方向。具体的研究方向包括：测试目标的静态分析与特征提取、目标覆盖与探索反馈机制、路径约束与目标引导策略优化等。

（3）定向模糊测试的性能优化：DGF的本质依旧是模糊测试，所以实现更高效的模糊测试能力仍是提升DGF能力的重要手段，如何对模糊测试中的流程进行更加适应性的改进，也将是后续研究中的热点问题。具体的研究方向包括：动态能量分配机制、高效种子突变策略优化、种子优先级的动态调整等。

DGF技术的发展在学术界和工业界都引起广泛关注，本文认为，DGF技术在未来的研究工作可

能会侧重以下几个方面。

（1）多类型崩溃探测

现有研究工作中，已有一些优质的工作[55,84-85]聚焦于 UAF 类型的漏洞，进行了深入的研究与分析。然而，DGF 测试所选取的测试目标通常涉及多种复杂的漏洞类型，单一的类型识别工作无法高效适用于大规模测试。为了提高测试的全面性和实用性，DGF 的研究需要进一步扩展，以包括更广泛类型的漏洞检测能力。例如，CAFL[90]支持了 7 种崩溃类型，如 UAF、double-free 等，验证了多类型崩溃探测的可行性与有效性。本文认为，DGF 未来的研究工作可以以单类型漏洞检测为基础，逐步迭代扩展到多个类型崩溃的支持，以实现不同类型崩溃重现的高效性，适应更广泛的漏洞检测需求。

（2）多指标模糊测试

大部分 DGF 的研究工作使用了测试目标的覆盖程度来引导测试，但单一的覆盖率指标可能会遗漏一些关键的非功能性问题，如内存使用和资源消耗。这些非功能问题在实际应用中可能导致性能下降或系统不稳定，严重时甚至可能引发系统崩溃。因此，单一指标的测试策略往往不能全面反映软件的实际运行状态和潜在风险。为了更全面地评估和提升软件的质量，未来的 DGF 应该向多指标优化的方向发展。不仅仅关注覆盖率这一单一指标，还可以从路径的复杂度、响应时间、资源消耗等方面，进行多维度的优化。如何做好多个指标之间的权衡，也是未来的研究方向之一。

（3）多程序语言支持

现有研究工作中 DGF 主要针对的程序语言是 C/C++，但是目前软件测试的需求越来越广泛，大型项目中的开发会使用不同的程序语言，例如：Java、Python 等语言会经常被同时使用。由于现有静态分析技术准确性的限制，DGF 难以直接同时应用到不同的语言中，因此需要对现有的分析和测试方法进行扩展和改进。为了适应多语言环境，DGF 可能需要引入更为灵活的静态分析框架，通过插件或模块化的方式支持多种语言，使得不同语言之间的集成和交互变得更加准确和高效。在多语言的软件项目中，由于各种语言特性和运行时行为的差异，DGF 还需优化其错误检测和性能分析工具，以确保能够准确识别和处理不同语言环境中的特定问题。

（4）应用大模型增强 DGF

现有的模糊测试技术在漏洞检测中仍存在一定的局限性，例如对代码结构、逻辑和上下文缺乏深入理解，导致在复杂漏洞的探索中效果不佳。为了解决这些问题，近年来一些研究[131-136]开始探索利用大模型（Large Language Model, LLM）来提升传统模糊测试的能力，这些研究工作利用 LLM 增强了测试对象的边缘覆盖率、优化了种子突变策略，生成了更高质量的种子，进而提升模糊测试的效能。同时，LLM 的强大语义理解能力同样可以应用于 DGF 技术[143-144]。未来的研究方向之一，可以聚焦于如何有效地利用 LLM 来增强 DGF 的效能，特别是在提高漏洞检测率、减少误报和加速测试过程方面。

9 结束语

本文对定向灰盒模糊测试技术（DGF）的研究进行了深入的分析与总结。首先，阐述了 DGF 技术的基本原理与具体流程，并探讨了其当前的应用情况与面临的挑战性问题；其次，详细归纳和分析了 DGF 技术在 3 个阶段进行的优化方法，包括：测试目标选取、目标定向的预处理和定向模糊测试的性能优化等方法；然后，对现有工作在实验中选取的测试对象进行了统计和分析；最后，对 DGF 技术未来的研究方向进行了展望。期望通过本文的工作，能为后续的研究者提供借鉴与参考，为推进 DGF 技术的进一步应用和发展做出贡献。

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

参 考 文 献

[1] Zahan N, Shohan S, Harris D, et al. Do software security practices yield fewer vulnerabilities?[C]//2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2023: 292-303.

[2] Wang W, Dumont F, Niu N, et al. Detecting software security vulnerabilities via requirements dependency analysis[J]. IEEE Transactions on Software Engineering, 2020, 48(5): 1665-1675.

[3] CVE Details. 2024. https://www.cvedetails.com

[4] Nie, C.-H. Concepts and Methods of Software Testing [M]. Tsinghua University Press, 2013. (in Chinese)
(聂长海 -.软件测试的概念与方法[M].清华大学出版社,2013.)

[5] Li ZJ, Zhang JX, Liao XK, et al. Survey of Software Vulnerability

Detection Techniques [J]. Ji Suan Ji Xue Bao/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[J]. 计算机学报, 2015, 38(4): 717-732.)

[6] Orso A, Rothermel G. Software testing: a research travelogue (2000–2014)[M]//Future of Software Engineering Proceedings. 2014: 117-132.

[7] Bertolino A. Software testing research: Achievements, challenges, dreams[C]//Future of Software Engineering (FOSE'07). IEEE, 2007: 85-103.

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

[9] Sutton M, Greene A, Amini P. Fuzzing: brute force vulnerability discovery[M]. Pearson Education, 2007.

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

[11] Woo M, Cha S K, Gottlieb S, et al. Scheduling black-box mutational fuzzing[C]//Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. 2013: 511-522.

[12] Chen K, Feng D G, Su P R, et al. Black-box testing based on colorful taint analysis[J]. Science China Information Sciences, 2012, 55: 171-183.

[13] Feng X, Sun R, Zhu X, et al. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference[C]//Proceedings of the 2021 ACM SIGSAC conference on computer and communications security. 2021: 337-350.

[14] Bounimova E, Godefroid P, Molnar D. Billions and billions of constraints: Whitebox fuzz testing in production[C]//2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013: 122-131.

[15] Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing[C]//2009 IEEE 31st International Conference on Software Engineering. IEEE, 2009: 474-484.

[16] Godefroid P, Kiezun A, Levin M Y. Grammar-based whitebox fuzzing[C]//Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation. 2008: 206-215.

[17] Wang J, Song C, Yin H. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing[J]. 2021.

[18] Chen H, Guo S, Xue Y, et al. {MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs[C]//29th USENIX Security Symposium (USENIX Security 20). 2020: 2325-2342.

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

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

[21] Ispoglou K, Austin D, Mohan V, et al. {FuzzGen}: Automatic fuzzer generation[C]//29th USENIX Security Symposium (USENIX Security 20). 2020: 2271-2287.

[22] CVE. 2024. https://cve.mitre.org/

[23] Song D, Lettner J, Rajasekaran P, et al. SoK: Sanitizing for security[C]//2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019: 1275-1295.

[24] LibFuzzer –A Library for Coverage-guided Fuzz Testing. 2024. https://llvm.org/docs/LibFuzzer.html.

[25] Li Y, Chen B, Chandramohan M, et al. Steelix: program-state based binary fuzzing[C]//Proceedings of the 2017 11th joint meeting on foundations of software engineering. 2017: 627-637.

[26] Chen P, Chen H. Angora: Efficient fuzzing by principled search[C]//2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018: 711-725.

[27] Gan S, Zhang C, Chen P, et al. {GREYONE}: Data flow sensitive fuzzing[C]//29th USENIX security symposium (USENIX Security 20). 2020: 2577-2594.

[28] Böhme M, Pham V T, Roychoudhury A. Coverage-based greybox fuzzing as markov chain[C]//Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 2016: 1032-1043.

[29] Li Y, Xue Y, Chen H, et al. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection[C]//Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2019: 533-544.

[30] American Fuzzy Lop. 2024. http://lcamtuf.coredump.cx/afl/technical_details.txt

[31] AFL Vulnerability Trophy Case. 2024. http://lcamtuf.coredump.cx/afl/#bugs

[32] Christakis M, Müller P, Wüstholz V. Guiding dynamic symbolic execution toward unverified program executions[C]//Proceedings of the 38th International Conference on Software Engineering. 2016: 144-155.

[33] Böhme M, Oliveira B C S, Roychoudhury A. Regression tests to expose change interaction errors[C]//Proceedings of the 2013 9th

Joint Meeting on Foundations of Software Engineering. 2013: 334-344.

[34] Marinescu P D, Cadar C. KATCH: High-coverage testing of software patches[C]//Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. 2013: 235-245.

[35] Jin W, Orso A. Bugredux: Reproducing field failures for in-house debugging[C]//2012 34th international conference on software engineering (ICSE). IEEE, 2012: 474-484.

[36] Pham V T, Ng W B, Rubinov K, et al. Hercules: Reproducing crashes in real-world application binaries[C]//2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE, 2015, 1: 891-901.

[37] Böhme M, Pham V T, Nguyen M D, et al. Directed greybox fuzzing[C]//Proceedings of the 2017 ACM SIGSAC conference on computer and communications security. 2017: 2329-2344.

[38] XIE Xiaoyuan, XU Lei, XU Baowen, et al. Survey of Evolutionary Testing. Journal of Frontiers of Computer Science and Technology, 2008, 2(5): 449–466. (in Chinese)

(谢晓园, 许蕾, 徐宝文, 等. 演化测试技术的研究[J]. 计算机科学与探索, 2008, 2(5): 449-466.)

[39] Xie XY, Xu BW, Shi L, Nie CH. Genetic Test Case Generation for Path-Oriented Testing. Journal of Software, 2009, 20(12): 3117–3136. http://www.jos.org.cn/1000-9825/580.htm (in English)

(谢晓园 [1, 徐宝文 [1, 史亮, 等. 面向路径覆盖的演化测试用例生成技术[J]. 软件学报, 2009, 20(12): 3117-3136.)

[40] Fang C R, Chen Z Y, Xu B W. Comparing logic coverage criteria on test case prioritization[J]. Science China Information Sciences, 2012, 55: 2826-2840.

[41] Harman M, Jia Y, Zhang Y. Achievements, open problems and challenges for search based software testing[C]//2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2015: 1-12.

[42] Li Z, Gong DW, Nie CH, Jiang H. Preface to the Special Issue on Search-Based Software Engineering [J]. Ruan Jian Xue Bao/Journal of Software, 2016, 27(4): 769–770. http://www.jos.org.cn/1000-9825/4976.html (in Chinese)

(李征, 巩敦卫, 聂长海, 等. 基于搜索的软件工程研究专题前言[J]. 软件学报, 2016, 27(4): 769-770.)

[43] Nie C, Wu H, Liang Y, et al. Search based combinatorial testing[C]//2012 19th Asia-Pacific Software Engineering Conference. IEEE, 2012, 1: 778-783.

[44] Khan M E, Khan F. A comparative study of white box, black box and grey box testing techniques[J]. International Journal of Advanced Computer Science and Applications, 2012, 3(6).

[45] Tan X, Zhang Y, Lu J, et al. SyzDirect: Directed Greybox Fuzzing for Linux Kernel[C]//Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. 2023: 1630-1644.

[46] Chen M, Mishra P. Property learning techniques for efficient generation of directed tests[J]. IEEE Transactions on Computers, 2011, 60(6): 852-864.

[47] Benahmed S, Qasem A, Lounis A, et al. Modularizing Directed Greybox Fuzzing for Binaries over Multiple CPU Architectures[C]//International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Cham: Springer Nature Switzerland, 2024: 84-103.

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

[49] Huang H, Guo Y, Shi Q, et al. Beacon: Directed grey-box fuzzing with provable path pruning[C]//2022 IEEE Symposium on Security and Privacy (SP). IEEE, 2022: 36-50.

[50] Liang H, Yu X, Cheng X, et al. Multiple targets directed greybox fuzzing[J]. IEEE Transactions on Dependable and Secure Computing, 2023, 21(1): 325-339.

[51] Huang H, Yao P, Chiu H C, et al. Titan: Efficient Multi-target Directed Greybox Fuzzing[C]//Proceedings of the 2024 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA. 2023: 20-22.

[52] Chen H, Xue Y, Li Y, et al. Hawkeye: Towards a desired directed grey-box fuzzer[C]//Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. 2018: 2095-2108.

[53] Srivastava P, Nagy S, Hicks M, et al. One fuzz doesn't fit all: Optimizing directed fuzzing via target-tailored program state restriction[C]//Proceedings of the 38th Annual Computer Security Applications Conference. 2022: 388-399.

[54] Liang H, Jiang L, Ai L, et al. Sequence directed hybrid fuzzing[C]//2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2020: 127-137.

[55] Nguyen M D, Bardin S, Bonichon R, et al. Binary-level directed fuzzing for {use-after-free} vulnerabilities[C]//23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020). 2020: 47-62.

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

[57] Cui ZQ, Zhang JM, Zheng LW, Chen X. A Survey of Research on Coverage-guided Greybox Fuzzing.Ji Suan Ji Xue Bao/CHINESE

JOURNAL OF COMPUTERS, 2024, 1-30. (in Chinese)
(崔展齐,张家铭,郑丽伟,等.覆盖率制导的灰盒模糊测试研究综述[J].计算机学报,2024,47(07):1665-1696.)

[58] Schloegel M, Bars N, Schiller N, et al. SoK: Prudent evaluation practices for fuzzing[C]//2024 IEEE Symposium on Security and Privacy (SP). IEEE, 2024: 1974-1993.

[59] Wang P, Zhou X, Yue T, et al. The progress, challenges, and perspectives of directed greybox fuzzing[J]. Software Testing, Verification and Reliability, 2024, 34(2): e1869.

[60] Österlund S, Razavi K, Bos H, et al. {ParmeSan}: Sanitizer-guided greybox fuzzing[C]//29th USENIX Security Symposium (USENIX Security 20). 2020: 2289-2306.

[61] Zheng H, Zhang J, Huang Y, et al. FishFuzz: Throwing larger nets to catch deeper bugs[J]. arXiv preprint arXiv:2207.13393, 2022.

[62] Kim T E, Choi J, Heo K, et al. {DAFL}: Directed Grey-box Fuzzing guided by Data Dependency[C]//32nd USENIX Security Symposium (USENIX Security 23). 2023: 4931-4948.

[63] Chess B, McGraw G. Static analysis for security[J]. IEEE security & privacy, 2004, 2(6): 76-79.

[64] Emanuelsson P, Nilsson U. A comparative study of industrial static analysis tools[J]. Electronic notes in theoretical computer science, 2008, 217: 5-21.

[65] Yoo S, Harman M. Regression testing minimization, selection and prioritization: a survey[J]. Software testing, verification and reliability, 2012, 22(2): 67-120.

[66] Leung H K N, White L. Insights into regression testing (software testing)[C]//Proceedings. Conference on Software Maintenance-1989. IEEE, 1989: 60-69.

[67] Soltani M, Panichella A, Van Deursen A. Evolutionary testing for crash reproduction[C]//Proceedings of the 9th International Workshop on Search-Based Software Testing. 2016: 1-4.

[68] Xuan J, Xie X, Monperrus M. Crash reproduction via test case mutation: Let existing test cases help[C]//Proceedings of the 2015 10th joint meeting on foundations of software engineering. 2015: 910-913.

[69] Zhang Z, Chen L, Wei H, et al. Prospector: Boosting Directed Greybox Fuzzing for Large-Scale Target Sets with Iterative Prioritization[C]//Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2024: 1351-1363.

[70] Rong H, You W, Wang X, et al. Toward Unbiased Multiple-Target Fuzzing with Path Diversity[J]. arXiv preprint arXiv:2310.12419, 2023.

[71] Xiang Y, Zhang X, Liu P, et al. Critical code guided directed greybox fuzzing for commits[C]//33rd USENIX Security Symposium (USENIX Security 24). 2024: 2459-2474.

[72] Shah A, She D, Sadhu S, et al. Mc2: Rigorous and efficient directed greybox fuzzing[C]//Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. 2022: 2595-2609.

[73] Yu X, Liang H, Wang C. Multiple Targets Directed Greybox Fuzzing: From Reachable to Exploited[C]//2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2024: 907-917.

[74] Lin P, Wang P, Zhou X, et al. HyperGo: Probability-based directed hybrid fuzzing[J]. Computers & Security, 2024, 142: 103851.

[75] Canakci S, Delshadtehrani L, Eris F, et al. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing[C]//2021 58th ACM/IEEE Design Automation Conference (DAC). IEEE, 2021: 529-534.

[76] Zong P, Lv T, Wang D, et al. {FuzzGuard}: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning[C]//29th USENIX security symposium (USENIX security 20). 2020: 2255-2269.

[77] Huang H, Zhou A, Payer M, et al. Everything is Good for Something: Counterexample-Guided Directed Fuzzing via Likely Invariant Inference[C]//2024 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, 2024: 142-142.

[78] Luo C, Meng W, Li P. Selectfuzz: Efficient directed fuzzing with selective path exploration[C]//2023 IEEE Symposium on Security and Privacy (SP). IEEE, 2023: 2693-2707.

[79] Niddodi C, Nagy S, Marinov D, et al. TOPr: Enhanced Static Code Pruning for Fast and Precise Directed Fuzzing[J]. arXiv preprint arXiv:2309.09522, 2023.

[80] Li X, Li X, Lv G, et al. Directed Greybox Fuzzing with Stepwise Constraint Focusing[J]. arXiv preprint arXiv:2303.14895, 2023.

[81] Lau H. FGo: A Directed Grey-box Fuzzer with Probabilistic Exponential cut-the-loss Strategies[J]. arXiv preprint arXiv:2307.05961, 2023.

[82] Zhang Y, Liu Y, Xu J, et al. Predecessor-aware Directed Greybox Fuzzing[C]//2024 IEEE Symposium on Security and Privacy (SP). IEEE, 2024: 1884-1900.

[83] Fang H, Zhang K, Yu D, et al. DDGF: Dynamic Directed Greybox Fuzzing with Path Profiling[C]//Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2024: 832-843.

[84] Wang H, Xie X, Li Y, et al. Typestate-guided fuzzer for discovering use-after-free vulnerabilities[C]//Proceedings of the ACM/IEEE 42nd

International Conference on Software Engineering. 2020: 999-1010.

[85] Zhang Y, Wang Z, Yu W, et al. Multi-level directed fuzzing for detecting use-after-free vulnerabilities[C]//2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). IEEE, 2021: 569-576.

[86] Liang H, Zhang Y, Yu Y, et al. Sequence coverage directed greybox fuzzing[C]//2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC). IEEE Computer Society, 2019: 249-259.

[87] Meng R, Dong Z, Li J, et al. Linear-time temporal logic guided greybox fuzzing[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 1343-1355.

[88] Lin P, Wang P, Zhou X, et al. DeepGo: Predictive Directed Greybox Fuzzing[C]//Proceedings of the Network and Distributed System Security Symposium. 2024.

[89] Li P, Meng W, Zhang C. SDFuzz: Target States Driven Directed Fuzzing[C]//Proceedings of the 33rd USENIX Security Symposium (Security). Philadelphia, PA, USA. 2024.

[90] Lee G, Shim W, Lee B. Constraint-guided directed greybox fuzzing[C]//30th USENIX Security Symposium (USENIX Security 21). 2021: 3559-3576.

[91] Garbelini M E, Wang C, Chattopadhyay S. Greyhound: Directed greybox wi-fi fuzzing[J]. IEEE Transactions on Dependable and Secure Computing, 2020, 19(2): 817-834.

[92] Yang K, He YP, Ma HT, Cai CF, Xie Y, Dong K. Guiding Directed Grey-box Fuzzing by Target-oriented Valid Coverage. Ruan Jian Xue Bao/Journal of Software, 2022, 33(11): 3967-3982. (in Chinese)

(杨克, 贺也平, 马恒太, 等. 有效覆盖引导的定向灰盒模糊测试[J]. 软件学报, 2021, 33(11): 3967-3982.)

[93] Mukherjee R, Kroening D, Melham T. Hardware verification using software analyzers[C]//2015 IEEE Computer Society Annual Symposium on VLSI. IEEE, 2015: 7-12.

[94] Canakci S, Matyunin N, Graffi K, et al. Targetfuzz: Using darts to guide directed greybox fuzzers[C]//Proceedings of the 2022 ACM on Asia conference on computer and communications security. 2022: 561-573.

[95] Chen M, Mishra P. Property learning techniques for efficient generation of directed tests[J]. IEEE Transactions on Computers, 2011, 60(6): 852-864.

[96] Mukherjee R, Kroening D, Melham T. Hardware verification using software analyzers[C]//2015 IEEE Computer Society Annual Symposium on VLSI. IEEE, 2015: 7-12.

[97] Google gvisor. 2024. https://github.com/google/gvisor

[98] Jeon Y, Han W H, Burow N, et al. {FuZZan}: Efficient sanitizer metadata design for fuzzing[C]//2020 USENIX Annual Technical Conference (USENIX ATC 20). 2020: 249-263.

[99] Ba J, Duck G J, Roychoudhury A. Efficient greybox fuzzing to detect memory errors[C]//Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. 2022: 1-12.

[100] Zhu X, Liu S, Li X, et al. Defuzz: Deep learning guided directed fuzzing[J]. arXiv preprint arXiv:2010.12149, 2020.

[101] Raghothaman M, Kulkarni S, Heo K, et al. User-guided program reasoning using Bayesian inference[C]//Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2018: 722-735.

[102] Liang J, Jiang Y, Chen Y, et al. Pafl: extend fuzzing optimizations of single mode to industrial parallel mode[C]//Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. 2018: 809-814.

[103] Blondel V D, Guillaume J L, Lambiotte R, et al. Fast unfolding of communities in large networks[J]. Journal of statistical mechanics: theory and experiment, 2008, 2008(10): P10008.

[104] Lemieux C, Sen K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage[C]//Proceedings of the 33rd ACM/IEEE international conference on automated software engineering. 2018: 475-485.

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

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

[107] UniBench. 2024. https://github.com/unifuzz/unibench

[108] Fuzzer Test Suite. 2024. https://github.com/google/fuzzer-test-suite.

[109] Nadi S, Berger T, Kästner C, et al. Mining configuration constraints: Static analyses and empirical results[C]//Proceedings of the 36th international conference on software engineering. 2014: 140-151.

[110] Murali A, Mathews N, Alfadel M, et al. Fuzzslice: Pruning false positives in static analysis warnings through function-level fuzzing[C]//Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024: 1-13.

[111] Huang R, Motwani M, Martinez I, et al. Generating REST API Specifications through Static Analysis[C]//Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. 2024: 1-13.

[112] Hopcroft J E, Ullman J D, Aho A V. Data structures and algorithms[M]. Boston, MA, USA:: Addison-wesley, 1983.

[113] Andersen L O. Program analysis and specialization for the C programming language[J]. 1994.

[114] Lu K, Hu H. Where does it go? refining indirect-call targets with multi-layer type analysis[C]//Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019: 1867-1881.

[115] Rapid Type Analysis (RTA) for Go. 2024. https://pkg.go.dev/golang.org/x/tools/go/callgraph/rta

[116] Go-callvis. 2024.https://github.com/ofabry/go-callvis

[117] Ball T, Larus J R. Efficient path profiling[C]//Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29. IEEE, 1996: 46-57.

[118] Wüstholz V, Christakis M. Targeted greybox fuzzing with static lookahead analysis[C]//Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 2020: 789-800.

[119] Dolan-Gavitt B, Hulin P, Kirda E, et al. Lava: Large-scale automated vulnerability addition[C]//2016 IEEE symposium on security and privacy (SP). IEEE, 2016: 110-121.

[120] Vardi M Y, Wolper P. An automata-theoretic approach to automatic program verification[C]//1st Symposium in Logic in Computer Science (LICS). IEEE Computer Society, 1986.

[121] Sui Y, Ye D, Xue J. Detecting memory leaks statically with full-sparse value-flow analysis[J]. IEEE Transactions on Software Engineering, 2014, 40(2): 107-122.

[122] Kirkpatrick S, Gelatt Jr C D, Vecchi M P. Optimization by simulated annealing[J]. science, 1983, 220(4598): 671-680.

[123] Aschermann C, Schumilo S, Blazytko T, et al. REDQUEEN: Fuzzing with Input-to-State Correspondence[C]//NDSS. 2019, 19: 1-15.

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

[125] Padhye R, Lemieux C, Sen K. Jqf: Coverage-guided property-based testing in java[C]//Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2019: 398-401.

[126] OSS-Fuzz: Continuous Fuzzing Framework for Open-Source Projects. 2024. https://github.com/google/oss-fuzz.

[127] OSS-Fuzz: Five Months Later, and Rewarding Projects. 2024. https://testing.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html

[128] OSS-Fuzz Trophies. 2024. https://github.com/google/oss-fuzz/tree/master/projects

[129] OSS-Fuzz Bug Details. 2024. https://github.com/google/oss-fuzz

[130] Li Y, Ji S, Chen Y, et al. {UNIFUZZ}: A holistic and pragmatic {Metrics-Driven} platform for evaluating fuzzers[C]//30th USENIX Security Symposium (USENIX Security 21). 2021: 2777-2794.

[131] Huang L, Zhao P, Chen H, et al. Large language models based fuzzing techniques: A survey[J]. arXiv preprint arXiv:2402.00350, 2024.

[132] Xia C S, Paltenghi M, Le Tian J, et al. Fuzz4all: Universal fuzzing with large language models[C]//Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. 2024: 1-13.

[133] Deng Y, Xia C S, Peng H, et al. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models[C]//Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis. 2023: 423-435.

[134] Zhang C, Bai M, Zheng Y, et al. Understanding large language model based fuzz driver generation[J]. arXiv preprint arXiv:2307.12469, 2023.

[135] Hu J, Zhang Q, Yin H. Augmenting greybox fuzzing with generative ai[J]. arXiv preprint arXiv:2306.06782, 2023.

[136] Deng Y, Xia C S, Yang C, et al. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries[C]//Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024: 1-13.

[137] Wang Y, Zhang C, Xiang X, et al. Revery: From proof-of-concept to exploitable[C]//Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. 2018: 1914-1927.

[138] Serebryany K, Bruening D, Potapenko A, et al. {AddressSanitizer}: A fast address sanity checker[C]//2012 USENIX annual technical conference (USENIX ATC 12). 2012: 309-318.

[139] UndefinedBehaviorSanitizer. 2024. https://clang.llvm.org/docs/UndefinedBehaviorS anitizer.html

[140] Nielson F, Nielson H R, Hankin C. Principles of program analysis[M]. springer, 2015.

[141] Xu Y, Jia H, Chen L, et al. ISC4DGF: Enhancing Directed Grey-box Fuzzing with LLM-Driven Initial Seed Corpus Generation[J]. arXiv preprint arXiv:2409.14329, 2024.

[142] Liang R, Chen J, Wu C, et al. Vulseye: Detect smart contract vulnerabilities via stateful directed graybox fuzzing[J]. IEEE Transactions on Information Forensics and Security, 2025.

[143] Zhou Z, Yang Y, Wu S, et al. Magneto: A Step-Wise Approach to Exploit Vulnerabilities in Dependent Libraries via LLM-Empowered Directed Fuzzing[C]//Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. 2024: 1633-1644.

[144] Liu J, Lee S, Losiouk E, et al. Can LLM Generate Regression Tests for Software Commits?[J]. arXiv preprint arXiv:2501.11086, 2025.

附录 1：定向灰盒模糊测试技术相关术语定义表

术语	英文全称和缩写	描述
公共漏洞披露[22]	Common Vulnerabilities and Exposures (CVE)	已经过确认会对安全性造成负面影响的安全漏洞。公布时，CVE 条目中会包含 CVE ID（格式如 CVE-2024-26903）、安全漏洞的概要描述（包括引起风险的函数和被利用点）、漏洞类型、严重性评级、受影响产品、修复和缓解措施、相关链接等
基于覆盖率引导的灰盒模糊测试[24-29]	Coverage-based Grey-box Fuzzing (CGF)	以覆盖率为导向的一种灰盒模糊测试技术，主要通过提高目标程序的代码覆盖率，以增加检测到错误和漏洞的可能性
定向灰盒模糊测试[37,52,56,60,90]	Directed Grey-box Fuzzing (DGF)	结合了灰盒模糊测试和程序分析技术的自动化测试方法，旨在通过生成有针对性的输入，提高针对特定目标测试的有效性。其中特定目标，指软件开发、维护过程中报告的已发生崩溃的代码位置，如 decompile.c 文件中第 398 行[90]；或者指 CVE 漏洞描述中的缺陷函数，如 CVE-2017-8392 漏洞描述中的_bfd_dwarf2_find_nearest_line 函数[37]；或者是自动化工具生成的疑似缺陷代码位置[60]
概念验证[137]	Proof of Concept (PoC)	可以重现目标漏洞的测试用例输入
测试对象[26,52]	Program Under Test (PUT)	用于验证定向灰盒模糊器能力所使用的测试对象，具体表现为：不同软件项目的特定版本（如 libming 0.4.8）、或者是可用于模糊测试的基准测试集（如 Magma[106]）等
代码检测器[23]	Sanitizer	检测软件缺陷的动态分析工具，通过在编译时插入检查代码，在程序运行时动态监控程序行为，一旦发现异常情况就会报告错误信息，如 AddressSanitizer[138]用于检查内存错误问题， UndefinedBehaviorSanitizer[139]用于检测未定义问题
过程间控制流图[140]	Inter-procedural Control Flow Graph (ICFG)	描述了整个程序中所有函数之间的控制流，不仅考虑了每个单独函数内的控制流，还包含函数之间的调用和返回关系
距离度量[37]	Distance Metric	当前测试执行的程序节点与测试目标间的距离度量
输入可达性[76]	Input Reachability	程序的输入是否可以抵达或触发测试目标
序列导向[55,84]	Sequence Guided	以特定的执行顺序引导模糊器抵达或触发测试目标，例如关注 use-after-free 类型漏洞的模糊器的序列导向为 malloc->free ->use[55,84]
种子[141]	Seed	用于生成测试输入的初始数据样本，可以随机或用户指定
能量调度[52]	Power Scheduling	规划种子在模糊测试过程中的资源分配，指定种子突变的次数
种子突变[52]	Seed Mutation	对输入种子进行的一系列修改和变形，以产生新的测试用例
种子优先级[52]	Seed Prioritization	突变后的种子，根据触发测试目标崩溃的能力分配优先级，按优先级排序执行测试

XU Yi-Jiang, born in 1997, Ph.D. candidate. His current research interests include software analysis and vulnerability detection.

GAO Qing, born in 1989, Ph.D., research associate. His current research interests include software analysis and vulnerability detection.

CHEN Li-Guo, born in 1997, Ph.D. candidate. His current research interests include software analysis and large language model evaluation.

Zhang Shi-Kun, born in 1969, Ph.D., research fellow, Ph.D. supervisor. His current research interests software engineering, network security, and knowledge computing.

WU Zhong-Hai, born in 1968, Ph.D., professor, Ph.D. supervisor. His current research interests include software engineering, big data system and analysis technology, big data and cloud security, and highly dependable embedded system.

Background

As software systems grow in size and complexity, and as cyber-attacks become more frequent, the challenge of ensuring software security has intensified. Recent data from CVEdetails[3] reveals that over 119,000 CVE vulnerabilities were disclosed between January 2021 and January 2025, emphasizing the critical need for robust vulnerability detection mechanisms. Fuzz testing, a widely-used technique in software security, generates diverse inputs to test programs, aiming to uncover vulnerabilities by causing unexpected behaviors. Among fuzz testing methods, grey-box fuzzing, which balances the external focus of black-box testing and the internal insight of white-box testing, is particularly effective for identifying software vulnerabilities.

Coverage-based Grey-box Fuzzing (CGF), the dominant approach, seeks to maximize code coverage to detect as many errors as possible. However, CGF is most effective in undirected scenarios and less suitable for targeted exploration needed in specific contexts, such as patch testing or reproducing crashes. To address this, Directed Grey-box Fuzzing (DGF) was introduced to improve testing efficiency and coverage for specific targets, such as known vulnerabilities or certain code paths. DGF has gained significant research interest due to its ability to focus testing efforts on specific areas, enhancing vulnerability detection.

DGF is a fuzz testing technique that aims to efficiently test specific areas of a program to identify vulnerabilities. The process begins with selecting specific test targets within the program, such as known vulnerabilities, critical code paths, or high-risk functions, to focus the fuzzing efforts. This selection often involves using static and dynamic analysis techniques to determine which parts of the program are most likely to contain security flaws. Following target selection, target-oriented preprocessing is conducted to prioritize paths leading to the chosen targets, reducing the exploration of irrelevant paths and concentrating testing resources on reaching the desired areas more effectively. Finally, the fuzzing process is optimized through various strategies, including energy scheduling, mutation strategies, and the use of priority queues, to enhance the efficiency and effectiveness of the testing, thereby increasing the likelihood of discovering vulnerabilities in the targeted code regions.

This paper focuses on the current state of development in DGF, exploring its application scenarios, challenges, and various research approaches. We analyzed and summarized existing work within the DGF workflow, concentrating on three key procedures: the selection of testing targets, the preprocessing of target-oriented guidance, and the performance optimization of directed fuzzing execution. Additionally, the paper provides insights into future developments and potential research directions in this field.

This work is supported by the National Key Research and Development Program under Grant No. 2021YFB3101802.