软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007323] [CSTR: 32375.14.jos.007323]

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

基于大语言模型的模糊测试研究综述*

李 岩[1], 杨文章[2], 张 翼[1], 薛吟兴[2,3]

[1](中国科学技术大学 软件学院, 安徽 合肥 230026)
[2](中国科学技术大学 计算机科学与技术学院, 安徽 合肥 230027)
[3](中国科学技术大学 苏州高等研究院, 江苏 苏州 215123)
通信作者: 薛吟兴, E-mail: yxxue@ustc.edu.cn

摘 要: 模糊测试是一种自动化的软件测试方法, 通过向目标软件系统输入大量自动生成的测试数据, 以发现系统潜在的安全漏洞、软件缺陷或异常行为. 然而, 传统模糊测试技术受限于自动化程度低、测试效率低、代码覆盖率低等因素, 无法应对现代的大型软件系统. 近年来, 大语言模型的迅猛发展不仅为自然语言处理领域带来重大突破, 也为模糊测试领域带来了新的自动化方案. 因此, 为了更好地提升模糊测试技术的效果, 现有的工作提出了多种结合大语言模型的模糊测试方法, 涵盖了测试输入生成、缺陷检测、后模糊处理等模块. 但是现有工作缺乏对基于大语言模型的模糊测试技术的系统性调研和梳理讨论, 为了填补上述综述方面的空白, 对现有的基于大语言模型的模糊测试技术的研究发展现状进行全面的分析和总结. 主要内容包括: (1) 概述模糊测试的整体流程和模糊测试研究中常用的大语言模型相关技术; (2) 讨论大模型时代之前的基于深度学习的模糊测试方法的局限性; (3) 分析大语言模型在模糊测试方法中不同环节的应用方式; (4) 探讨大语言模型技术在模糊测试中的主要挑战和今后可能的发展方向.

关键词: 大语言模型; 模糊测试; 测试输入生成; 缺陷检测; 后模糊处理
中图法分类号: TP311

中文引用格式: 李岩, 杨文章, 张翼, 薛吟兴. 基于大语言模型的模糊测试研究综述. 软件学报. http://www.jos.org.cn/1000-9825/7323.htm
英文引用格式: Li Y, Yang WZ, Zhang Y, Xue YX. Survey on Fuzzing Based on Large Language Model. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7323.htm

Survey on Fuzzing Based on Large Language Model

LI Yan[1], YANG Wen-Zhang[2], ZHANG Yi[1], XUE Yin-Xing[2,3]

[1](School of Software Engineering, University of Science and Technology of China, Hefei 230026, China)
[2](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)
[3](Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou 215123, China)

Abstract: Fuzzing, as an automated software testing method, aims to detect potential security vulnerabilities, software defects, or abnormal behaviors by inputting a large quantity of automatically generated test data into the target software system. However, traditional fuzzing techniques are restricted by such factors as low automation level, low testing efficiency, and low code coverage, being unable to handle modern large-scale software systems. In recent years, the rapid development of large language models has not only brought significant breakthroughs to the field of natural language processing but also introduced new automation solutions to the field of fuzzing. Therefore, to better enhance the effectiveness of fuzzing technology, existing works have proposed various fuzzing methods combined with large language models, covering modules like test input generation, defect detection, and post-fuzzing. Nevertheless, the existing works lack

* 基金项目: 国家自然科学基金 (61972373)
本文由“大模型下的软件质量保障”专题特约编辑王赞教授、王莹副教授、陈碧欢副教授、姚远副教授、张敏灵教授推荐.
收稿时间: 2024-07-17; 修改时间: 2024-10-14; 采用时间: 2024-11-25; jos 在线出版时间: 2024-12-10

systematic investigation and discussion on fuzzing techniques based on large language models. To fill the above-mentioned gaps in the review, this study comprehensively analyzes and summarizes the current research and development status of fuzzing techniques based on large language models. The main contents include (1) summarizing the overall process of fuzzing and the relevant technologies related to large language models commonly used in fuzzing research; (2) discussing the limitations of deep learning based fuzzing methods before the era of large language model (LLM); (3) analyzing the application methods of large language models in different stages of fuzzing; (4) exploring the main challenges and possible future development directions of large language model technology in fuzzing.

Key words: large language model (LLM); fuzzing; test input generation; defect detection; post-fuzzing

1 引 言

随着信息技术的快速发展, 软件系统与日常生活的融合程度不断提高, 极大地便利了公众的日常生活和工作. 与此同时, 软件系统的应用也面临着许多安全问题. 软件中的潜在缺陷会导致通讯、支付等数据的泄露, 给个人和社会带来较为严重的经济损失和安全风险. 因此, 确保软件系统在部署的环境中安全稳定运行愈发重要.

模糊测试作为一种发现软件安全漏洞的有效手段, 尤其适用于发现未知的、隐蔽性较强的底层缺陷, 并通过及时修复提高软件系统的安全性. 模糊测试的核心思想是通过大量输入数据对目标系统进行测试, 以触发其内部异常处理机制, 并通过观察系统响应和行为来发现潜在的漏洞和错误. 尽管模糊测试已在一定程度上实现了自动化, 但对于具有复杂功能和结构的软件系统, 即使利用基于约束[1]、基于随机[2]、基于学习[3]等辅助手段, 设计高效的模糊测试程序仍然是一项具有挑战性的任务.

近年来, 人工智能领域涌现出拥有庞大模型参数规模的大语言模型 (large language model, LLM), 如 ChatGPT[4]、Llama 2[5]. 这些模型通过大量数据训练, 在机器翻译、文本生成、文本分类等自然语言处理 (natural language processing, NLP) 任务中表现出良好性能, 并可成功应用于各种软件工程任务, 为模糊测试方法研究带来新机遇. 并且, 随着 LLM 的不断发展, 也出现了专用于程序语言处理任务的代码 LLM, 如 CodeGen2[6]、DeepSeek Coder[7]等. 经过大量程序源代码的训练, 这些模型可以更加正确地完成代码生成、程序修复等程序语言处理任务, 并能充分适配模糊测试中的测试输入生成和缺陷检测等环节.

将 LLM 应用于模糊测试领域有利于提升模糊测试自动化水平, 分析被测试软件的潜在漏洞, 从而实现对系统缺陷的高效检测. 一方面, LLM 经过大量语料库训练, 只需进行较少样本学习即可了解目标程序, 并自动生成符合模糊测试要求的测试用例. 另一方面, LLM 作为具有大量参数的语言模型, 具有强大的学习推理能力, 可深入分析具有复杂调用关系的程序漏洞. 尽管 LLM 无法保证结果的完全正确性, 且存在因对软件理解不足和处理能力有限而生成错误测试用例的情况, 但是这并不会为软件系统引入新的安全隐患. 此外, 在模糊测试中, 测试用例的多样性有助于提高测试的充分性, 因此即使是错误的测试用例, 仍有可能帮助发现意外的软件漏洞[8].

本文在公开的期刊以及会议论文中检索了目前基于 LLM 的模糊测试研究中提出的新方法, 并根据以下 3 个步骤在文献库中检索和选取文献.

(1) 本文选用 IEEE Xplore、ACM Digital Library、Google 学术等文献数据库和搜索引擎进行原始搜索. 针对论文的标题、摘要、关键字和索引, 使用关键字 fuzzing、testing、LLM、models、GPT 等进行检索.

(2) 为了避免遗漏相关研究, 本文在上述步骤的基础上, 根据每篇文献的参考文献列表寻找与基于 LLM 的模糊测试问题相关的研究文献, 并将遗漏的相关文献添加到本文的研究文献中.

(3) 在文献的选取过程中, 本文将以下研究工作排除在外: ① 文献没有使用 LLM 驱动测试任务; ② 文献不涉及模糊测试任务, 例如仅使用 LLM 用于代码生成; ③ 文献的研究重点为对 LLM 的评估, 例如对 LLM 本身安全性或生成能力进行测试; ④ 文献仅在展望中提及 LLM, 没有将 LLM 应用到具体研究中.

根据上述步骤和选取原则, 本文最终收集了 51 篇文献作为综述总结的相关文献. 图 1 展示了 2020 年 (GPT-3 发布) 至 2024 年 6 月每年关于 LLM 驱动的模糊测试研究的文献发表统计情况. 从总体趋势来看, LLM 驱动的模糊测试研究从 2021 年就开始出现, 并在 2023 年出现了明显的数量激增, 这与 LLM 的快速发展, 特别是 ChatGPT 的出现和应用密切相关.

此外, 如图 2 所示, 本文根据中国计算机学会 (CCF) 推荐的国际学术会议和期刊列表, 进一步对文献发表的会议和期刊情况进行了统计. 其中, 发表在 CCF 评级 A 类期刊/会议的文献包括: ICSE (13 篇), FSE (5 篇), ISSTA (2 篇), NDSS (1 篇), TSE (1 篇), S&P (1 篇), USENIX Security (1 篇) 发表在非 CCF 评级 A 类期刊/会议 (即“Others”分类) 的文献有 26 篇.

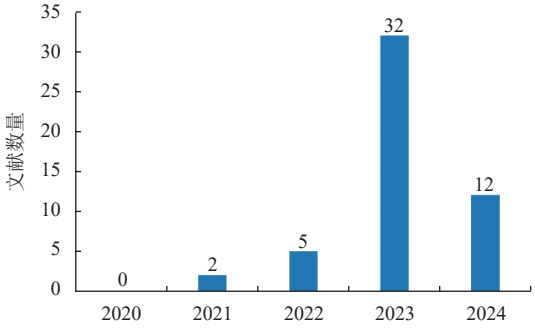

图 1 文献发表年份统计

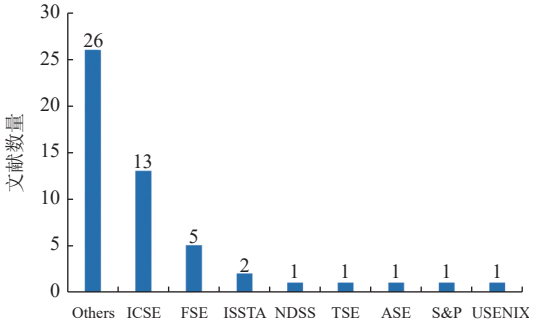

图 2 基于 LLM 的模糊测试文献分布

本文主要对基于大语言模型的模糊测试研究与应用进展进行综述. 第 2 节简要介绍研究背景, 包括模糊测试的分类、基本工作流程, 并探讨在模糊测试中常用的 LLM 技术. 第 3 节对大模型时代前的基于深度学习的模糊测试方法及其局限性进行总结. 第 4 节重点分析基于 LLM 的模糊测试方法的研究现状. 最后, 第 5 节对本文进行总结并展望未来研究方向.

2 相关背景

2.1 模糊测试

模糊测试是目前检测程序中安全缺陷最成功的技术之一, 其原理是向被测系统随机或定向生成输入, 并监视异常结果来发现系统漏洞[9].

如图 3 所示, 模糊测试的一般工作流程可分为 5 个基本步骤, 即预处理、测试输入生成、测试执行、缺陷检测和后模糊处理.

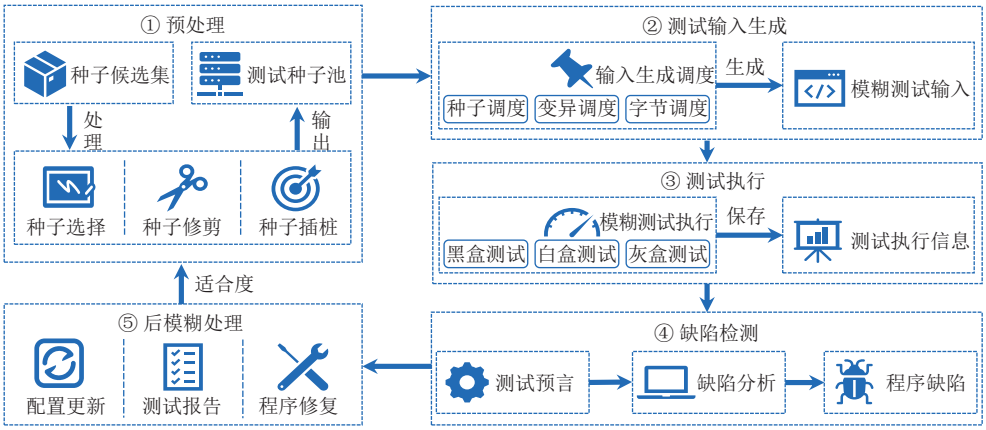

图 3 模糊测试的一般流程

(1) 模糊测试的预处理阶段是指在测试输入生成和执行之前, 对目标系统或应用程序进行准备工作的过程. 这个阶段的目的是为模糊测试配置基础环境, 确保测试过程能够高效、有序地进行, 主要包括种子的选择、修剪和

插桩.

(2) 测试输入生成阶段的目的是生成大量的异常、畸形或随机的测试输入数据. 这些测试输入可以通过基于变异的方法生成, 即从已知的数据样本出发, 依次完成种子调度、变异调度和字节调度, 分别确定待变异的种子程序、变异算子和待变异的字节位置, 通过变异产生新的测试用例; 或者基于生成的方法, 根据特定的协议或接口规范建模并生成测试用例.

(3) 执行模块的职责是根据模糊测试输入运行目标测试程序, 将生成的测试用例输入到被测系统中进行测试, 根据执行过程中所获得的代码信息, 模糊测试可以分为黑盒模糊测试、灰盒模糊测试和白盒模糊测试.

(4) 缺陷检测模块监视执行情况以检查是否发现新的执行状态或缺陷.

(5) 后模糊处理的目的是整理模糊测试执行信息, 生成测试报告, 并根据缺陷检测信息更新模糊测试的配置. 从更广义的视角来看, 程序修复等操作也属于模糊测试的后模糊处理任务, 有助于进一步完善测试框架.

2.2 基于 LLM 的模糊测试常用技术

目前, 在基于 LLM 的模糊测试中, 主流的技术手段包括微调、提示工程和传统算法融合. 如表 1 所示, 微调技术能够明显提高 LLM 对特定任务的性能, 且微调过程与深度学习训练过程类似较为简单, 但是要求所使用的 LLM 必须开源, 且微调过程需要较高的硬件资源. 而提示工程是 LLM 的一种特殊优化方法, 无需对 LLM 再次训练即可获得较好的任务适应能力, 但是由于提示策略多样, 需要反复调整和实验才能获得较好的提示内容. 而传统算法融合则是 LLM 的辅助方法, 弥补单一 LLM 的固有缺陷, 但 LLM 和传统算法的结合需要设计专门的组件, 设计较为复杂.

表 1 基于 LLM 的模糊测试常用技术优缺点分析

分类	优点	缺点
微调	(1) 微调过程较为简单 (2) 能够提高LLM对特定任务的性能	(1) LLM必须为开源模型 (2) 微调需要较高的硬件配置 (3) 微调过程训练时间长 (4) 微调数据收集繁琐
提示工程	无需对模型进行再次训练	(1) 提示模板设计繁琐 (2) 提示策略选择需要反复尝试
传统算法融合	弥补单一LLM的固有缺陷	与传统算法结合设计复杂

2.2.1 微 调

微调是指在预训练模型的基础上对模型参数进行微小调整, 以适应特定任务的需求. 通过微调, 预训练模型可以更好地适应目标任务, 从而实现更好的性能. 微调利用 LLM 已经学到的知识对模型进行再次训练, 而并非从头开始学习, 这可被视为 LLM 应用迁移学习的一种方式.

为了使预训练的模型适应下游任务, 微调方法以监督的方式训练 LLM. 具体来说, 给定一个由特定任务样本 X 和相应标签 Y 组成的数据集, 微调的目标是为模型找到一组合适的参数 θ, 使得 $\theta = \text{argmin}_{\theta} P(Y|X;\theta)$ [10]. 基于上述原理, LLM 微调常包括以下步骤: 选择开源的 LLM 模型; 准备并预处理微调 LLM 的数据集; 加载数据集训练 LLM. 目前主流的 LLM 微调方法为: Freeze 方法[11]、P-tuning 方法[12]、LoRA 方法[13]和 QLoRA 方法[14].

相比从头开始训练 LLM, 微调 LLM 可以节省大量时间和硬件资源. 同时可以减少数据需求, 即使数据量较少, 微调也可以实现良好的性能. 针对软件测试等特定任务, LLM 微调可以根据用户需求进行定制化训练, 并且随着新数据的产生, 可以帮助 LLM 实现持续学习.

2.2.2 提示工程

虽然微调 LLM 能有效提高 LLM 解决特定任务的能力, 但是微调要求 LLM 必须开源. 然而, 由于成本高昂且具有较高的商业价值, 很多 LLM 仍处于闭源状态. 同时, 由于 LLM 的结构复杂且包含大量参数, 对 LLM 进行微调所需的硬件资源和时间成本也较高. 已有研究成果表明, 提示调优方法优于对 LLM 进行数据微调[10]. 目前主流的 LLM 提示方法包括: 零样本学习、少样本学习、思维链等.

零样本学习 (zero-shot learning)[15]是指在对 LLM 进行提示时, 不提供样本示例. 在基于 LLM 的代码生成任务中, 零样本学习是通过将任务文本直接输入给模型并询问结果来实现的, 通常有有指导和无指导两种方式. 有指导的方法, 如后续第 4.1.3 节中提到的 ChatUniTest, 会在提示中为 LLM 提供“请帮助我为特定 Java 方法生成 JUnit 测试... ”的说明, 以促进模糊测试用例的生成. 相比之下, 无指导的方式, 如文献 [16], 仅提供单元测试用例的代码头, 例如“class className Test”, LLM 将自动执行单元测试用例的生成. 一般来说, 有指导的方式带来更准确的输出, 而无指导的方式则适用于更加具体的应用场景.

相比零样本学习, 少样本学习 (few-shot learning)[17]使用的是少量但高质量的具有标签的样本数据进行学习. 在 LLM 执行代码生成任务时, 少样本学习的方式是首先提供一组高质量的代码示例, 每个示例都包含目标任务的输入和所需输出. 当 LLM 学习这些示例后, 它可以更好地理解人类的意图和对答案的期望标准. 例如, 在 LiBro[18]方法中, 为了使 LLM 能完成根据常见错误报告自动生成测试用例的任务, 研究人员提供了错误报告和使对应错误重现的测试用例.

思维链 (chain-of-thought, CoT)[19]提示是一种无梯度的提示技术, 可以引导 LLM 生成一系列短句来逐步描述推理逻辑, 最终得出答案. Wei 等人[19]最先研究了语言模型中 CoT 提示的问题, 该技术促使 LLM 生成一系列连贯的中间推理步骤, 从而得出问题的最终答案. 他们的研究表明, LLM 可以通过零样本提示 (zero-shot-CoT) 或手动编写的少样本演示 (Manual-CoT) 来执行 CoT 推理. zero-shot-CoT 方法是在测试相关问题后添加一个提示, 例如“首先我们考虑, 其次我们分析”或设计类似的步骤清单等, 以促使 LLM 生成对目标任务的思维链. 由于这种提示范例与任务无关, 无需输入演示, 因此是 zero-shot 的方式. 而 Manual-CoT 由人工为 LLM 设计一定的演示, 每个演示都有一个问题和对应的思维链, 包括一系列中间推理步骤和预期答案. 为了进一步减少人工工作量, 最近也出现了 Auto-CoT 的研究工作[20]. 例如, Yin 等人[21]探索了如何将 Auto-CoT 应用于问答任务, 根据时间顺序查找每个时间范围内的相关信息, 对 LLM 逐步提问, 激发 LLM 在理解时间顺序概念的基础上完成相关问答任务的潜力.

自动提示 (auto prompt)[22]是一种自动生成提示信息的机制, 旨在消除人工设计提示模板和手动组织提示信息的需求. 在模糊测试领域, 自动提示通常涉及两个甚至更多 LLM 的应用. 通常负责生成提示信息的模型需要深入分析当前任务及其上下文, 提取关键特征, 并生成结构化的候选提示内容. 这些提示信息随后被传递给另一个专注于生成模糊测试输入的模型, 后者利用这些提示高效生成测试数据. Kojima 等人[23]的研究指出, 自动提示方法是一种比人工设计更具优势的零样本提示策略, 因为它能够自适应任务需求, 自动优化提示结构, 从而为模糊测试生成更有效的输入数据.

2.2.3 传统算法融合

尽管 LLM 在许多自然语言处理任务中表现出色, 但由于不同的模型架构和参数设计, 不同的 LLM 在代码生成、上下文学习和任务填充等方面的性能有所不同, 例如 GPT-4[24]擅长总结上下信息、StarCoder[25]更擅长生成代码程序[26]. 因此, 单一的 LLM 在模糊测试任务中往往很难达到最佳效果, 难以全面应对多样化的任务需求. 为了克服这些挑战, 研究人员尝试将 LLM 与传统的模糊测试技术相结合, 以充分发挥各自的优势. 一种常见的方法是利用反馈驱动的验证机制对 LLM 生成的测试用例进行优化和迭代. 通过引入程序分析技术, LLM 可以根据被测试程序的结构和行为特征, 生成更具针对性的输入, 从而提高测试的覆盖率. 同时, 变异算子被用于对生成的输入进行多样化处理, 以探索更多的程序路径, 并通过统计分析技术对测试结果进行系统性评估.

3 大模型时代前的基于深度学习的模糊测试

相比于 LLM, 研究人员更早接触深度学习, 因此也更早开展了基于深度学习优化模糊测试的研究. 本节将介绍大模型时代之前的基于深度学习的模糊测试方法的研究现状, 并分析深度学习方法的不足.

3.1 大模型时代前的基于深度学习的模糊测试应用方式

深度学习作为一种强大的机器学习方法, 在图像识别、自然语言处理等领域取得了显著的成功. 例如, 在图像

识别领域, 深度学习技术已经在诸如 ImageNet[27]大规模视觉识别挑战赛中展现了卓越的性能, 能够准确地从大量图片中识别出细微的特征差异, 从而实现了对物体、场景等的高效分类与检测. 而在自然语言处理方面, 深度学习的长短时记忆网络 (LSTM)[28]架构及其变体通过捕捉长距离依赖关系, 能够更准确地判断文本的情绪色彩, 进而实现对社交媒体监控、市场趋势分析[29]; 深度学习的变压器 (Transformer)[30]模型由于其并行化处理机制和自注意力机制, 能够显著提高翻译任务质量和效率, 在多项基准测试中表现出更低的错误率[31]. 这些成果表明, 深度学习技术能够学习到数据的深层次特征, 可以为提升模糊测试等软件测试技术的效率和智能化水平提供较为智能的途径. 目前, 深度学习已经被应用到了模糊测试的测试输入生成、缺陷检测和后模糊处理等阶段.

3.1.1 基于深度学习的测试输入生成

基于深度学习的测试输入生成方法通常利用神经网络的学习能力, 通过训练得到能够产生有效测试用例的网络模型. 这些模型通常采用如卷积神经网络 (CNN)[32]、循环神经网络 (RNN)[33]等深度学习架构, 以捕获程序输入的内在规律性和复杂性.

例如, 在对文件解析工具进行模糊测试时, Learn&Fuzz[34]将 PDF 对象视为字符序列, 通过训练基于 RNN 的 Char-RNN 模型, 实现了对 PDF 格式文件解析工具的模糊测试. FuzzGuard[35]则根据 CNN 的特点设计了一种基于 3 层 CNN 模型的深度学习预测模型, 可以在执行目标应用程序之前根据程序的数据流结构预测种子的目标可达性, 该方法有助于定向灰盒模糊测试过滤掉不可达的输入, 从而提高模糊测试性能.

3.1.2 基于深度学习的缺陷检测

在模糊测试中, 缺陷检测环节主要包括测试预言生成和缺陷分析两个部分.

基于深度学习的测试预言生成方法通常利用了深度神经网络在特征提取方面的优势, 自动地从大量历史测试数据中学习出有效的预言. 例如, Monsefi 等人[36]将足够数量的软件输入及其相应的期望输出进行编码得到训练数据, 利用这些数据他们训练得到了用于生成测试预言的深度神经网络.

同样地, 基于深度学习的缺陷检测方法通常也是利用大量的数据来训练模型, 使得模型可以更有效的提取程序源代码的语义信息并识别潜在的代码缺陷. 同时, 为了使深度学习模型可以充分理解代码结构, 部分方法尝试将程序结构表示为向量对模型进行训练. 例如, Li 等人[37]首先将程序转换为抽象语法树, 再通过映射和词嵌入将其编码为数字向量, 仅将数字向量输入卷积神经网络 (CNN)[32]进行训练, 以自动学习程序的语义和结构特征, 实现对程序缺陷的识别和检测.

3.1.3 基于深度学习的后模糊处理

在模糊测试中, 后模糊处理阶段包含的任务较为多样, 包括测试报告的生成[38,39]、模糊测试的更新[40]等. 但目前深度学习系统更多地侧重于后模糊处理的程序修复任务, 通过学习修复前后的代码片段对模型进行训练, 以对程序的自动修复. 例如 Li 等人[41]提出的 DEAR 方法, 该方法深度学习和数据流分析相结合, 通过修复前后源代码的抽象语法树对 LSTM[27]模型进行训练, 可以对程序中的故障位置进行定位和自动修复.

3.2 大模型时代前的基于深度学习的模糊测试不足

尽管基于深度学习的模糊测试能够提高测试效率和测试有效性, 但是基于深度学习的模糊测试方法仍然存在较多不足. 如表 2 所示, 与 LLM 相比, 基于深度学习的模糊测试方法主要存在两类不足: 深度学习系统的固有局限性、深度学习系统的应用局限性.

表 2 基于深度学习的模糊测试方法的局限性

分类	局限性	解释
深度学习系统固有局限性	训练数据的局限性	训练数据质量差
	特征学习和推理能力的局限性	特征学习和推理能力差
深度学习系统应用局限性	测试输入生成的局限性	深度学习系统生成无效或低效的测试输入
	缺陷检测的局限性	测试预言知识理解不足、程序缺陷识别能力差
	后模糊处理缺陷的局限性	程序修复性能差、智能化和自动化程度较低

3.2.1 深度学习系统的固有局限性

(1) 训练数据局限性

基于深度学习的模糊测试, 其测试效果不仅与模型架构相关, 同时还依赖于模型训练集. 合格的训练集需要包含足够数量同时比例恰当的正负样本. 然而, 在模糊测试任务中, 由于涉及不同编程语言和测试任务, 部分情景缺乏足够的训练数据样本. 现有方法通常是使用网络爬虫方法提取或利用其他模糊测试工具生成的数据来构建当前任务的数据集.

现有的一些公共数据集, 往往存在样本类别少、特征不足的问题, 且用于训练的数据不平衡. 研究表明, 基于深度学习的模糊测试工具经常面临训练数据不平衡的问题, 导致模型仅基于统计证据盲目地将分支预测为“未发现”[42]. 相反, LLM 是在大量丰富的语料库上进行训练的, 通常只需进行小样本学习甚至零样本学习, 就能够很好地理解测试用例和目标程序, 减少了训练数据对模型的影响.

(2) 特征学习和推理能力局限性

尽管深度学习在语音识别、图片特征提取等领域展现出强大的特征学习和推理能力, 但当使用深度学习模型分析具有复杂调用关系的程序时, 如多点触发的漏洞和多重嵌套函数[43], 基于深度学习的算法性能表现往往较差. 深度学习模型通过多层次的特征学习来自动提取数据中的关键特征, 其黑箱特性也导致其可解释性不足. 在 Li 等人[25]的研究中, 他们发现基于深度学习的模糊测试方法的性能并没有明显优于其他模糊测试方法, 利用神经网络模型来指导变异在某些项目中改进并不显著.

3.2.2 深度学习系统的应用缺陷

(1) 测试输入生成局限性

尽管基于深度学习的测试输入生成方法在提高模糊测试的效率和有效性方面取得了显著进展, 但它们仍存在一些不足之处, 特别是基于深度学习的模糊测试方法可能会生成无效或低效的测试用例, 可能导致后续的编译问题或浪费测试执行时间. 例如, DeepXplore[44]在生成测试输入时只能修改现有的输入, 它们的性能取决于初始种子的质量, 同时由于该方法只是对现有的有效输入施加较小的扰动, 新的测试输入可能仍然局限于从原来测试输入可到达的区域, 导致许多其他有效输入区域未经测试.

(2) 缺陷检测局限性

首先, 针对测试预言的生成任务, 模型对于测试预言的理解不足. 一方面, 对测试预言的理解受限于模型的学习能力, 尤其是在面对复杂软件行为时, 单一深度学习模型的性能往往较差. 另一方面, 现有的深度学习预言生成方法往往需要大量的标注数据, 而这些数据的获取通常既昂贵又耗时. 此外, 由于深度学习模型的黑盒特性, 即使预言能够正确分类测试用例, 解释其决策过程也是一个难题.

其次, 基于深度学习的缺陷检测技术对于程序的修复能力仍然存在局限性. 首先, 深度学习模型的性能很大程度上取决于训练数据的质量和多样性. 如果训练数据不足以涵盖所有可能的缺陷类型, 模型可能无法准确检测某些类型的缺陷. 其次, 深度学习模型可能在面对新型或少见的缺陷时表现不佳.

此外, 利用深度学习系统检测代码漏洞的能力边界并不明确. 例如, 根据通用弱点枚举 (common weakness enumerations, CWE)[45]分类体系, 缺陷被划分为 10 个主要类别, 而这些主类别下的次级分类则多达上百种, 每种缺陷的特点和表现形式各不相同. 传统的方法会为每个特定类型的缺陷定制规则, 但现有的基于深度学习的缺陷检测手段往往忽略了不同缺陷类型之间的区别, 通常只是简单地将它们视为二分类或多元分类的问题.

同时, 目前关于深度学习在源代码缺陷检测领域的探索大多还停留在理论研究层面, 通常只是专注于如何对数据集中的项目进行分类. 然而, 源代码缺陷检测是软件开发流程中不可或缺的一部分, 具有重要的实用价值. 因此, 这项研究也应当结合工业应用的实际需求来进行, 即从大规模工程项目的角度考虑, 确保整个过程的有效性和技术方案的可行性. 但目前该部分的研究依然属于空白阶段[46].

(3) 后模糊处理局限性

虽然深度学习方法已经可以实现对程序的自动修复, 但是在目前的研究成果中, 基于深度学习的程序修复方法的修复性能依然较差. 例如, 在目前基于深度学习的程序修复方法中, 许多训练数据集包含噪声 (例如, CoCoNuT[47]

包含许多重复的样本), 这可能会降低模型的性能.

同时基于深度学习方法的程序修复方法, 智能化和自动化程度较低. 比如, 有研究表明[48], 深度学习方法需要针对每个打补丁的软件程序执行所有可用的功能测试套件, 无法进行智能化选择, 这将大量消耗执行时间; 同时, 由于过度拟合问题, 开发人员需要进一步执行人工检查来评估合理补丁的正确性, 不仅自动化程度低, 而且人工评估过程极易出错.

综上所述, 基于深度学习的模糊测试方法通过学习模糊测试输入和程序缺陷的特征, 可以实现自动化地生成测试输入和缺陷检测. 但是, 由于深度学习方法本身存在的固有缺陷, 以及它在模糊测试应用中存在较大的局限性, 基于深度学习的模糊测试方法在缺陷检测性能方面存在严重不足. 因此, 研究人员尝试利用 LLM 指导模糊测试, 并出现了较多研究成果, 在第 4 节中, 本文将对基于 LLM 的模糊测试方法进行总结.

4 基于 LLM 的模糊测试

随着 LLM 解决代码生成和推理分析等任务的能力不断增强, 研究人员对基于 LLM 的模糊测试方法了解也越来越深入.

实验证明, LLM 的自身特点能够有效克服深度学习在模糊测试中的固有缺陷. 首先, LLM 是在大量丰富的语料库上进行训练的, 通常只需进行小样本学习甚至零样本学习, 就能够很好地理解测试用例和目标程序, 减少了训练数据对模型的影响. 其次, LLM 作为具有大量参数的语言模型, 具备强大的推理能力, 能够更好地学习和生成自然语言和程序代码, 甚至可以模拟人类的思考过程, 可以推动模糊测试任务的学习和实现.

基于 LLM 在 NLP 领域表现出的强大能力, 将 LLM 应用于模糊测试的研究快速发展. 由于模糊测试的预处理和测试执行模块尚未出现 LLM 的应用工作, 因此本节将针对深度学习方法在模糊测试的应用缺陷, 仅详细介绍 LLM 在模糊测试的测试输入生成、缺陷检测和后模糊处理这 3 个方面的研究现状.

4.1 LLM 驱动测试输入生成

目前已经有很多领域引入 LLM, 将 LLM 与领域知识结合实现应用. 由于 LLM 在代码生成方面的适应性较强, 代码生成领域已经有较多研究成果. 基于现有的研究成果, 部分研究人员开始应用 LLM 进一步实现测试输入的自动化生成. 相比深度学习系统, LLM 在预训练过程中学习了更加丰富且多样的代码数据, 能够更加充分理解测试输入生成任务的要求, 有效克服深度学习系统生成测试输入低效或无效的缺点, 提升模糊测试的测试效率. 在目前的研究中, 相比模糊测试的其他任务, 利用 LLM 生成测试输入是研究成果最多的部分, 已经发展出了基于微调、基于提示工程、和传统算法融合这 3 类测试输入生成方法.

LLM 驱动测试输入生成的基本流程如图 4 所示. 为了提高 LLM 生成测试输入的有效性, 研究人员将采取一些措施指导 LLM 生成测试输入的候选集. 常用的措施有结合训练数据微调 LLM[49−54]、根据种子程序和目标程序生成提示[18,55−64]、结合传统算法策略[65−80]等. 由于 LLM 生成的测试输入可能会导致编译错误, 因此研究人员需要设计输入验证器对生成的测试输入进行验证. 输入验证器将通过执行编译判断测试输入的正确性, 若测试输入出现编译错误, 输入验证器则将编译过程的错误信息发送至提示生成器或 LLM, 错误信息将作为提示的一部分重新指导 LLM 生成新的测试输入.

4.1.1 基于微调的方法

早期 LLM 驱动测试输入生成通常采用微调的方式, 从而提高 LLM 解决特定任务的能力, 一般是使用任务相关的数据对 LLM 进行再次训练. 例如, Alagarsamy 等人[49]提出的 A3Test 方法, 他们采用自监督的方式利用 Java 语言料库中的断言语句训练语言模型, 使模型具有更强的断言基础知识.

除了直接使用与代码生成任务相关的现有数据集对 LLM 进行微调, Deng 等人[50]在提出的 FuzzGPT 中通过历史错误代码片段数据集对 LLM 进行微调. FuzzGPT 利用历史错误代码样本中的 API 名称、错误描述以及错误触发的代码片段等信息, 让 LLM 进行自动回归预测, 使用梯度下降不断更新权重, 从而使 LLM 适应模糊测试任务的模型参数得到了改进. 类似地, LLAMAFUZZ[51]利用 AFL 实验过程中的历史测试输入数据集对 Llama 模型进

行微调, 然后利用微调后的模型进行测试输入得生成和变异, 同时为了保证测试输入的多样性, LLAMAFUZZ 也同时保留了随机变异的组件, 以得到充足的测试输入.

在实践过程中由于 LLM 生成的测试用例通常难以被人类理解, Hashtroudi 等人[52]提出了一个自动化的测试框架. 该框架在对 CodeT5 进行微调时使用了 Methods2Test 数据集[53], 其中包含源代码、相关的测试代码以及源代码的上下文 (如类名称、构造函数方法的签名、公共变量和字段以及类中所有其他方法的签名等信息). 结合数据集中的上下文内容设计提示信息, LLM 能更好地理解测试用例特征, 从而输出正确的测试用例.

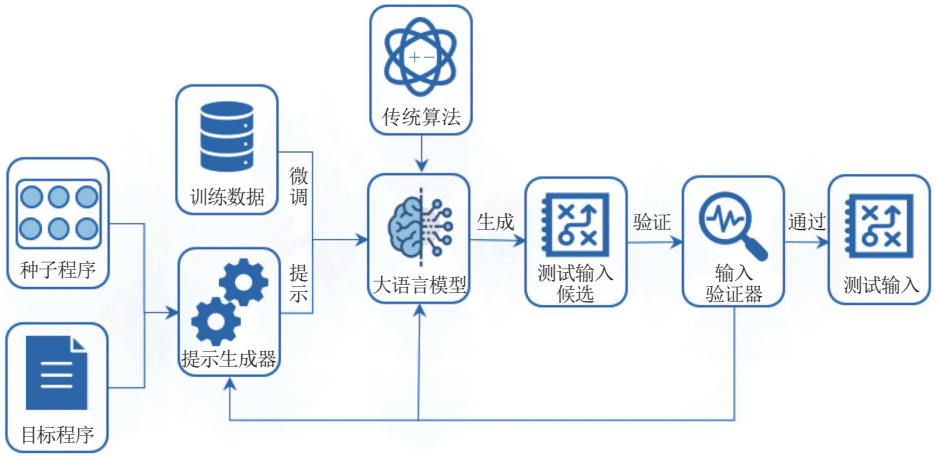

图 4 LLM 驱动测试输入生成的一般流程

上述的微调方法是对 LLM 进行指令微调, 即通过模拟提问和回答的过程进行训练. 除此之外, 基于强化学习的微调也被用于模糊测试任务中. 例如, Jueon 等人提出的 CovRL 方法[54], 它通过统计模糊测试过程中输入的对应的覆盖率信息, 结合种子频率作为权重, 得到的基于覆盖率的奖励策略, 并通过 PPO 算法[81]进行强化学习, 实现对基于 LLM 的输入变异的不断增强.

4.1.2 基于提示工程的方法

由于微调方法在模型开源性、数据集选择等方面的局限, 后期 LLM 驱动测试输入生成更多关注基于提示的方式. 为了获得更好的生成效果, 基于提示的方法需要引入模糊测试领域知识、提供清晰明确的问题以及合适长度的上下文信息, 以引导模型生成与特定功能相关的测试用例. 目前, 应用 LLM 时主流的提示方法包括: 零样本学习、少样本学习、思维链和自动提示方法.

(1) 零样本学习方法

零样本学习是指在对 LLM 的提示内容中不提供显式学习示例的提示方法, 让 LLM 从先前学到的知识中推断、联想, 并通过简单地将任务文本输入模型中获得结果. 为了获得较好的生成效果, 该方法需要设计清晰、具体的提示内容, 以激发模型对任务的正确理解.

提示内容可以从 GUI 界面中进行提取. 比如 Liu 等人[55]提出了 QTypist 方法用于测试移动端应用的 GUI. 首先, 该方法根据上下文信息提取 GUI 测试相关的组件信息, 并进行预处理, 例如通过词性标注处理提取的信息, 考虑下划线和驼峰式命名法. 在生成具体测试时, QTypist 根据提取到的信息从人工设计的语言模式中选择适当的范式, 并组合成对 LLM 的提示内容.

提示内容也可以直接选择代码程序. 比如 Deng 等人[56]在 TitanFuzz 中直接向生成式 LLM 提供包含目标库和目标 API 签名信息的提示, 以生成相应的测试用例. 除了直接使用目标代码程序, 可以根据与目标程序类似的代码设计提示. 例如 Li 等人[57]提出的 AceCoder 提示技术, 在向 LLM 模型提供提示之前, 他们使用检索器和选择器获取与目标程序类似的代码程序, 生成对 LLM 的提示信息, 实现在无样本学习的情况下对 LLM 进行提示. 对于

LLM 预训练过程中已经包含的被测试目标, 提示信息可以更加简单, 比如 Asmita 等人[58]认为 GPT-4 已经了解 BusyBox awk 小程序的数据格式, 因此在没有微调和样本提示的情况下, 通过直接提示“Generate initial seed to fuzz BusyBox awk applet”, 得到了大量的测试输入.

(2) 少样本学习方法

与零样本学习方法不同, 小样本学习允许在对 LLM 的提示内容中提供少量的学习示例, 从而使模型能够从有限的示例中学习并适应到新任务和领域.

在模糊测试任务中, 一般采用在对 LLM 提示中增加测试用例的示例程序, 引导 LLM 生成符合格式要求、满足功能需求的测试用例程序. 例如, Nashid 等人[59]的 CEDAR 方法在构建提示时设计了代码演示实例, 帮助模型明确开发人员期望的预期响应. 在 ChatAFL 方法[60]中也设计了对 LLM 的提示模板, 内容包括执行内容、期望内容 (Shot-1; Shot-2), 其中 Shot-1 和 Shot-2 即为代码示例代码.

为了实现基于错误报告设计提示中的样本示例, Kang 等人[18]在提出的 LiBro 方法中提取了错误报告中的代码、报错信息等内容, 并构造了一个 Markdown 文档, 并在文档中添加了一些独特的部分 (例如 LLM 的指令、代码块的开头, 以及诱导 LLM 输出测试方法的标记“public void test”) 作为完整的提示内容.

为进一步提高小样本学习方式对 LLM 的提示效果, Ahmed 等人[61]还引入了静态分析方法. Ahmed 等人提出的 ASAP 方法使用从源代码中自动提取的语义事实来增强提示, 即在提供给 LLM 的少量样本中添加参数名称、局部变量名称、调用的方法和数据量等语义信息.

(3) 思维链方法

思维链方法是模拟人类思考步骤的提示方法, 它强调在提示中逐步建设上下文, 确保生成的内容具有前后文的关联性. 这种逐步引导有助于确保生成的内容保持连贯性, 符合逻辑结构.

为了模拟测试用例的设计步骤, Vikran 等人[62]在 PBT-GPT 方法中设计了一个包括 4 步提示的提示策略. 提示内容包括: 告知 LLM 它是专家级 Python 程序员、查看 API 文档并生成 PBT 组件、输入 API 方法文档以及提示所需的输出格式. 通过引入思维链方式, LLM 生成基于属性的测试用例可以实现 98% 的属性有效性. 同样地, Ackerman 等人[63]针对模糊测试的种子生成任务, 也采用四步骤提示方法, 包括理解、框架、形成和变形这 4 个步骤. 这使得 LLM 可以递归地检查自然语言格式规范, 在生成和变形步骤中变异种子程序. 初步实验结果表明, 这种方法优于简单的突变模糊测试和随机模糊测试.

(4) 自动提示方法

在模糊测试任务中, 自动提示方法是利用 LLM 分析上下文信息, 自动生成提示内容的方法, 该方法一般使用多个 LLM. Xia 等人[26]设计了基于两个 LLM 的自动提示方法, 用于生成测试用例. 在 Fuzz4All[26]中, 该方法利用其中一个 LLM 接受相关内容输入, 将其提炼为简洁但信息丰富的模糊测试提示, 并将生成的提示输入给另一个 LLM. 类似地, WhiteFox 方法[64]也使用了双模型框架, 其中负责分析的 LLM 分析代码程序并总结测试需求, 而具体生成用例的 LLM 则根据总结的要求制定测试程序.

4.1.3 传统算法融合

如第 2.2.3 节中所述, 单一 LLM 并没有在模糊测试的各个阶段任务中达到全能的状态. 比如, LLM 仍然存在代码生成错误、语义理解不足的局限性, 因此, 将 LLM 与反馈验证策略结合可以不断改正错误的代码, 与程序分析技术相结合可以缓解语义理解的劣势. 此外, 即使 LLM 能够生成正确的代码程序, 但是 LLM 对于模糊测试的种子选择和种子优劣并不了解, 因此需要结合传统模糊测中的变异算子和统计分析等方法进行优化.

(1) 反馈验证

由于 LLM 生成的测试用例可能存在语法错误、语义错误等编译问题, 为了获得适用于软件测试任务的测试输入, 必须对 LLM 生成的结果进行验证和修复. 在现有的 LLM 驱动的软件测试输入生成技术中, 为确保生成的测试输入能够正确编译, 大多数技术框架都采用了验证-反馈范式.

例如, Chen 等人提出的 ChatUniTest 方法[65]. 该方法设计了基于 ChatGPT 的软件测试框架, 分为生成、验证和修复这 3 个阶段. 在完成生成任务后, ChatUniTest 会提取并验证测试; 如果测试无法编译或在执行过程中遇到

错误, 则会进入修复过程, 使用基于规则或基于 ChatGPT 的修复方法, 反复迭代以获得正确的生成结果.

与 ChatUniTest 类似, Yuan 等人[66]也基于 ChatGPT 模型和验证-反馈范式提出了 ChatTester. ChatTester 方法包含一个迭代测试改进模块, 该模块迭代地修复初始测试生成的测试用例中的编译错误. 首先, 该方法在验证环境中编译生成的测试来验证其有效性, 然后根据编译时的错误信息以及与编译错误相关的额外代码上下文构建提示, 并将新的提示交给 ChatGPT, 以获得正确的测试程序.

在 Schäfer 等人[67]提出的 TESTPILOT 中, 同样借鉴了验证-反馈的思想. 然而, 与前述的 ChatUniTest 和 ChatTester 不同, TESTPILOT 并未直接修复 LLM 生成的测试用例. 相反, TESTPILOT 将生成的测试用例传递给验证组件, 该组件直接运行程序以确定测试是否通过, 以及具体的失败类型. 随后, 验证组件将错误信息提供给提示改进器, 优化对模型的提示信息, 以便 LLM 能够重新生成更为准确的测试用例. 同样地, InputBlaster[68]方法也是在测试输入生成过程中, 将执行信息反馈给提示生成组件, 用于引导 LLM 减少生成错误的测试输入.

(2) 程序分析

由于软件代码具有复杂的结构信息, 而 LLM 在执行测试用例生成时对代码结的构信息的理解不足. 因此研究人员尝试结合程序分析技术从而有效提高 LLM 对代码结构地理解.

例如, Mahbub 等人[69]提出的 Bugsplainer 方法采用程序分析方法将目标程序的代码构建为抽象语法树, 以便有效地表示程序的代码结构, 并提高 LLM 准确理解代码的能力. 基于程序分析方法, 还可以实现从其他代码文件中提取额外的代码上下文, 以便模型能够专注于代码库中与任务最相关的信息, 并带来更准确的预测. 这一思路的典型案例是 ChatTester[66]和 LLM4Fuzz[70], 它们在准备阶段会通过静态分析方法全面总结了关于焦点方法的上下文信息, 如类声明、构造函数签名等上下文信息.

在 Linux 内核的模糊测试中, 即使有丰富的文档和示例, 直接应用 LLM 进行测试依然面临效率低下的挑战[71−73]. 主要原因是内核代码规模庞大且高度复杂, 涉及众多模块, 难以全面覆盖所有功能. 此外, 内核的状态空间庞大, 模糊测试工具难以有效探索深层次的状态交互. 内核级模糊测试需要模拟接近实际运行的环境, 构建和维护这些高开销的环境会显著增加测试的复杂度和时间成本. 同时, 内核对输入有严格的验证逻辑, 随机生成的输入经常被过滤, 无法深入测试核心功能. 特别是, 内核中的许多漏洞隐藏在特殊的边界条件下, 普通模糊测试工具难以生成足够精确的输入来触发这些条件. 因此 KernelGPT[74]对源代码进行了静态分析, 提取内核相关的操作符、内核功能和参数定义等信息, 利用 LLM 的推理能力生成特定的测试规范, 以增强内核模糊测试.

(3) 变异算子

将 LLM 与模糊测试中的变异算子方法结合可以增加模糊测试的初始种子数量和种类, 使得模糊测试对被测程序的测试更加全面. 为了得到更好的种子程序, 可以直接利用 LLM 生成种子的突变体. 例如 Hu 等人[75]提出的 ChatFuzz 系统, 该系统中包含了一个“chat mutator”组件, 它会从种子池中挑选一个种子程序, 并提示 ChatGPT 模型生成与示例种子程序类似的输入, 以得到更多的变异种子程序. 除此之外, 也可以通过调整提示得到种子程序的变异体. 例如, Li 等人[76]在 CCTEST 方法中, 对提示 p_0 进行变异, 得到一组变异的提示 $\{p_1,\ldots,p_k\}$. CCTEST 通过将不同的提示信息提供给 LLM, 可以得到不同的种子程序.

上述利用 LLM 结合变异算子的方法仅是利用 LLM 的生成特点, 得到一组类似的种子程序, 但并没有考虑到种子程序对代码覆盖率的影响. 因此, SearchGEM5 方法[77]和 CodaMosa[78]选择利用基于搜索的模糊测试执行过程中的覆盖率信息, 来生成有利于提高覆盖率的变异种子. 它们首先利用 LLM 生成初始的种子程序并执行, 当注意到测试覆盖率停滞时, 它们会识别被测程序中覆盖率较低的代码区域, 并利用变异方法生成新的测试用例, 以引导搜索测试到这些区域. 不同的是, SearchGEM5 利用传统 AFL++方法[38]实现变异, 而 CodaMosa 是利用 LLM 生成未覆盖区域的种子程序.

(4) 统计分析

上述方法将 LLM 与程序分析方法和变异算子方法结合, 有助于生成更多的测试用例. 尽管 LLM 能够生成大量的测试用例, 但这些用例在输入格式或语义上可能高度相似, 在测试过程中实际触发的程序路径或状态变化也往往是重复的, 无法有效覆盖新的边界或漏洞. 此外, 即使某些测试用例在形式上有所差异, 它们的“测试能力”, 即

发现问题的潜力和触发不同程序行为的能力, 也可能相差很大. 有些用例能够触发重要的边界条件, 而其他用例则几乎不影响测试结果, 导致执行了大量冗余、低效的测试输入, 从而降低模糊测试的整体效率. 因此, 可以结合统计分析的方法, 对 LLM 生成的测试用例进行排名和聚类, 以适当减少模糊测试阶段的时间负担. 例如, Fakhoury 等人提出的 TICODER[79]对生成的测试候选和代码候选进行排序, 并根据最后的排序情况选择最佳的测试程序. 同样, Xia 等人在使用 LLM 进行自动修复时, 通过计算熵值对生成的补丁进行了排序和选择[80].

4.2 LLM 驱动缺陷检测

在模糊测试的缺陷检测模块中包括测试预言生成和缺陷分析两部分. 其中, 实现对软件缺陷的准确识别需要设定合理的测试预言. 大多数预言生成方法侧重于测试断言的生成, 并据此检测出被测程序中的潜在问题. 而缺陷分析是通过对过程信息进行分析及时识别软件缺陷, 输出被测程序的潜在漏洞, 并服务于模糊测试的后模糊处理阶段. 与测试输入生成任务类似, 相比深度学习系统, LLM 凭借预训练过程中庞大的代码数据, 具备更丰富的断言知识, 能够生成准确的测试断言. 同时, 由于 LLM 架构包含巨量的参数, LLM 对程序缺陷特征的分析和识别能力也更强, 可以缓解深度学习系统对程序缺陷识别效果较差的不足. 目前, 基于 LLM 的模糊测试缺陷检测已经发展出了基于微调、基于提示工程和与传统算法融合的缺陷检测方法.

如图 5 所示, 在基于 LLM 驱动的缺陷检测一般流程中, 一种策略是利用断言数据或缺陷数据对 LLM 进行微调, 提高 LLM 对断言语句结构的识别能力和代码缺陷的识别能力[82−86]; 另一种策略是利用结合断言的示例程序或执行信息设计对 LLM 的提示, 引导 LLM 生成符合要求的测试预言, 完成对缺陷的识别定位[59,87−93].

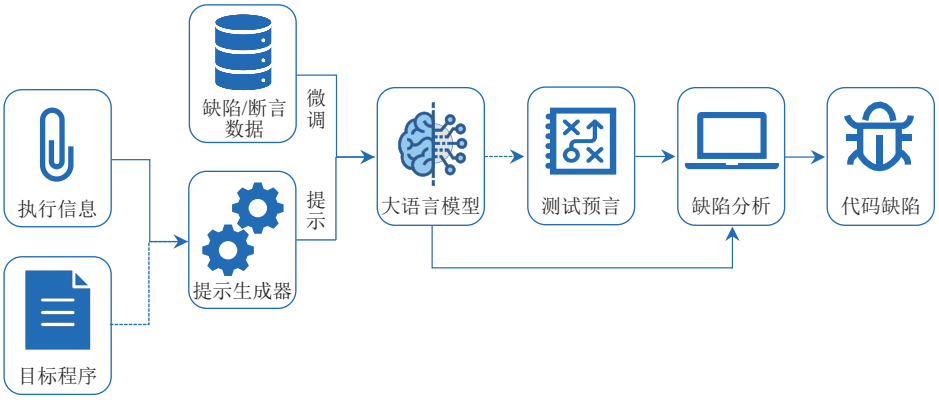

图 5 LLM 驱动缺陷检测的一般流程

4.2.1 测试断言生成

(1) 基于微调的方法

在实践中, 为了使 LLM 生成更高效的断言, 通常采用微调的方法. 例如, Mastropaolo 等人[82]使用由英语文本和源代码组成的数据集对 T5 模型进行微调. 随后, 通过重用修复错误、生成断言语句等数据集, 进一步微调模型, 以提升其生成断言语句的性能. 类似地, Tufano 等人[83]也使用英语语料库和代码语料库对 LLM 进行预训练, 然后在断言数据集上采用测试方法、焦点方法和断言进行微调, 以进一步提升 LLM 的断言生成性能.

(2) 基于提示工程的方法

除微调方法外, 现在也出现了基于提示的断言生成方式. 在 DIVAS 框架[87]中, 该方法利用 LLM 的知识库来识别与被测片上系统最相关的通用弱点枚举 CWE. 然后, 根据 LLM 生成的 CWE 映射生成等效的断言. Nashid 等人[59]提出的方法在构建提示时会自动检索与任务相似的代码演示. 只需少量相关代码演示, 就能实现 76% 的精确匹配准确率.

(3) 传统算法融合方法

在测试预言的应用中, 差异测试思想较多的应用于模糊测试当中. 例如, TitanFuzz[56]在测试的最后阶段, 将 LLM

生成的测试用例分别在 CPU 和 GPU 上执行, 对深度学习库函数进行测试. 通过对比 CPU 和 GPU 的表现差异, 判断出被测程序中存在的缺陷. 与深度学习库的测试不同, 针对 C++和 Python 程序的测试预言, Liu 等人[94]提出的 AID 框架会首先通过分析程序规范获得不同的程序变体, 之后结合多样性原则选出一组程序并提供相同的测试输入, 并通过比较输出的差异剔除掉错误的测试预言.

除了直接利用差异测试外, 还可以尝试对 LLM 提出不同任务要求, 通过比较不同任务的差异, 间接查找潜在的程序缺陷. 例如 Li 等人[95]在测试 ChatGPT 识别错误测试用例的能力时, 差异提示会分别要求 ChatGPT 推断被测程序的意图, 并生成多个与被测程序相同意图的可编译程序. 当程序执行效果和推断的意图存在差异时, 该测试用例可能存在潜在问题.

4.2.2 缺陷分析

(1) 基于微调的方法

针对缺陷驱动检测任务, 微调方法是较好的实现策略. 为了提高 LLM 检测缺陷的能力, Paul 等人[84]在研究中使用了两个数据集对 LLM 进行微调, 其中包含有缺陷的代码片段以及对应的代码注释, 提高 LLM 对代码缺陷的识别检测能力. 实验结果表明, 经过微调后的 LLM 性能有明显提升. 除了直接使用代码数据集, Zhang 等人[86]尝试通过英语语料库对 BART 模型[85]进行微调, 使其能够根据错误描述自动生成错误标题, 从而简化了程序缺陷分类和后续程序修复过程.

(2) 基于提示工程的方法

利用缺陷数据不仅可以实现对 LLM 的微调, 而且还可以基于提示对 LLM 进行上下文学习. 例如, Liu 等人[88]提出的 VUL-GPT 方法, 首先通过代码检索方法从数据集中检索与给定测试代码片段相似的代码样本信息, 然后将相似代码、相似代码的漏洞信息、测试代码组合为提示信息输入到 GPT 模型中进行上下文学习, 实现利用 LLM 有效地利用上述信息进行漏洞检测.

基于提示工程思想, Kang 等人[89]提出了基于提示工程的 AutoSD 方法. 在该方法的具体实现中, AutoSD 首先会提示 LLM 生成有关导致错误原因的假设以及用于测试假设的调试器脚本. 然后, AutoSD 执行调试命令并向 LLM 提供调试的执行结果, LLM 最终决定是否满足假设, 预测是否完成调试过程, 或者是否需要进行额外分析, 通过不断的假设循环实现对软件缺陷的检测.

在缺陷检测中常用的一种方法是污点分析, 但这种方法是一个部分自动化的过程, 需要大量人力来执行. 因此, Liu 等人[90]提出了 LATTE, 实现了基于 LLM 的静态二进制污点分析方法. 该方法将污点分析方法识别的危险数据流与特定的分析任务相结合, 构建对 LLM 的提示信息, 然后使用生成的提示与 LLM 进行通信, 逐步引导 LLM 执行各种分析任务, 以检查二进制文件中的潜在漏洞, 实现了污点分析的自动化缺陷检测.

在缺陷的检测过程中, 缺陷的定位是调试修复缺陷的重要步骤. 因此, Kang 等人[91]首次提出了基于 LLM 的故障定位方法——AutoFL. 在对 LLM 进行提示时, AutoFL 采用了两步思维链的方式. 第 1 步是询问导致故障的根本原因, 第 2 步是请求输出有关故障位置信息. 通过使用两步思维链, AutoFL 实现了逐步引导 GPT-3.5 对故障问题进行位置定位, 能够获得更高的准确率.

为了实现较为完整的缺陷检测和修复框架, Bui 等人[92]首先提出了一个基于 LLM 的 Detect-Localize-Repair 框架进行调试. 该框架首先确定给定代码片段是否有错误, 然后识别有错误的行, 并将有错误的代码翻译成正确版本. 随后, 在 SkipAnalyzer[93]方法中, 也提出了基于 LLM 的缺陷检测、过滤和修复的 3 组件框架. 该方法使用包含各类错误类型、错误代码片段的提示让 LLM 进行学习, 以提高 LLM 在该任务上的准确性.

4.3 LLM 驱动后模糊处理

在模糊测试的后模糊处理环节, 通常包含测试报告生成、程序修复和配置更新. 但是对于测试报告生成和配置更新, 传统的模糊测试已经实现了较为自动化的方法, 因此尚未有基于深度学习的模糊测试方法对该两类任务进行优化. 而 LLM 具备优秀的信息总结能力, 能够生成更易读的测试报告. 此外, 如图 6 所示, LLM 还能根据测试报告的结果调整对适应度的评估策略, 从而间接实现对模糊测试的配置更新. 例如, CHEMFUZZ[96]方法在测试的

结果分析阶段利用 LLM 来协助识别化学软件计算结果中潜在的异常数据, 并将数据中的警告数量 W 作为种子适应度的重要计算依据, 实现对模糊测试的配置更新. 但是此类工作, 在基于 LLM 的后模糊处理方法中应用依然较少.

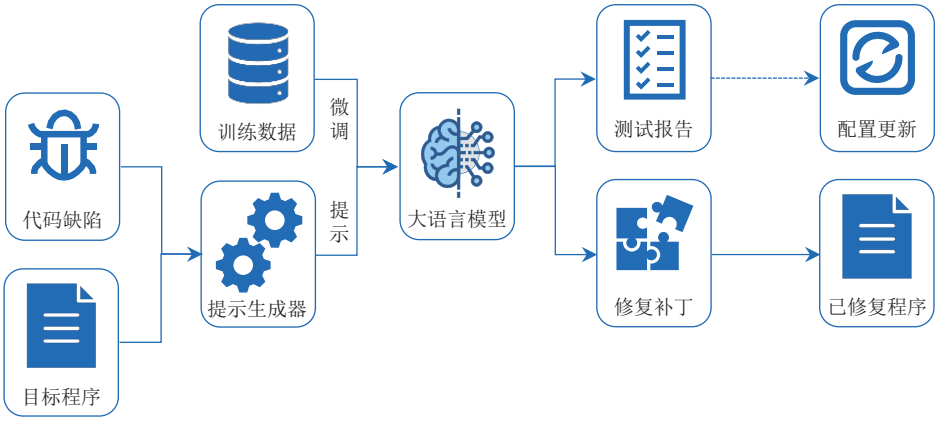

图 6 LLM 驱动后模糊处理的一般流程

在程序修复方面, LLM 凭借其先进的自然语言处理和理解能力, 能够高效处理和分析源代码, 生成有效的程序修复补丁, 使其成为修复错误的理想工具, 成为利用 LLM 后模糊处理的重点, 其代码修复能力相比深度学习系统准确率更高. 同样地, 基于 LLM 的程序修复方法也已经发展出了基于微调[97−100]、基于提示工程[101,102]和与传统算法融合[103]的缺陷检测方法.

4.3.1 基于微调的方法

在将 LLM 应用于程序修复任务时, 可以利用错误程序以及对错误程序的修复数据对 LLM 进行微调, 提高 LLM 在生成代码补丁和修复程序的能力. 研究表明[104], 微调后的 LLM 可以显著优于传统程序修复工具.

例如, Zhang 等人[97]提出用于修复神经网络程序的 REPEATNPR 框架. 为了使 LLM 在 REPEATNPR 中学习通用的错误修复表示, 研究人员通过 BFP 数据集 D 的训练语料库对 LLM 进行微调, 数据集 D 的每个实例形式为错误修复对 $d_i = \{b, f\}$, 其中 $b = (c, m, s)$ 包括全局上下文 c、从错误的方法 m 和错误的语句 s 中切出的过程上下文, f 表示人工编写的程序补丁.

基于同样的设计思想, Mashhadi 等人[98]在 ManySStuBs4J 数据集上微调 LLM, 微调过程仅需将存在程序缺陷的代码示例作为训练数据对 LLM 进行再次训练. 经过微调的模型, 不仅可以生成不同长度的修复, 而且适用于修复不同类型的代码错误.

在对 LLM 进行微调的过程中, 微调数据的选择至关重要. 为了实现自动化的微调数据设置, Jin 等人[99]提出了 InferFix, 该方法会从语义上搜索等效的错误和相应的修复, 并将搜索到的错误及修复数据对生成式 LLM 进行微调. 通过让 LLM 学习错误修复数据, LLM 对于程序错误的修复能力能够得到明显提升.

除了检索微调数据外, 可以通过编译方法扩充原有数据集. 因此, Hao 等人[100]设计了 APRFiT 方法对微调数据进行增强, 他们使用一组代码增强操作符自动增强修复错误的代码数据, 例如使用 Change If Statement 算子将单个 if 语句更改为条件表达式语句, 并将新的代码补充到微调数据中. 例如, 把原始的 if 语句“if (condition) {statement1; } else {statement2; }”转换为“statement1 = (condition) ? statement1 : statement2;”. 这样的转换不仅丰富了数据集的多样性, 还帮助模型学习到不同语法结构之间的对应关系. 实验结果表明, 使用 APRFiT 增强后的数据集进行微调, LLM 在生成程序补丁的性能得到了显著提升, 这证明了 APRFiT 方法在提高程序修复成功率方面的有效性.

4.3.2 基于提示工程的方法

对 LLM 进行修复任务的微调需要检索大量的错误程序以及应对的修复程序, 而为所有编程语言和模糊测试

对象准备充分的微调数据是不可行的. 因此, Pearce 等人[101]设计了基于 LLM 提示工程的方法, 用于对网络安全漏洞的修复. 该方法首先使用现有的安全工具识别程序中的错误, 结合错误信息设计对 LLM 的提示信息, 由此提升 LLM 在网络应用程序漏洞修复方面的能力.

上述基于提示的方法直接使用代码程序作为示例, 而 Fakhoury 等人[102]基于 LLM 在 NLP 领域的出现表现, 验证了 LLM 根据自然语言描述生成功能正确的代码的能力. 因此, 他们将代码修改的自然语言描述 (即存储库中问题报告中描述的错误修复) 作为提示信息提供给 LLM, 要求 LLM 将其转换为正确的代码修复, 实现了根据自然语言描述修复错误的方法.

4.3.3 传统算法融合

除了微调和提示工程方法, 研究人员还尝试结合程序分析方法, 对代码进行基于结构的子集化处理, 限制 LLM 对软件潜在问题的分析范围. 例如, 在 Zhang 等人[105]提出的 PyDex 方法中, 在处理程序中的语义问题时, PyDex 方法对程序进行分块, 由程序分块器对原始有缺陷程序中的每个语法错误进行分析, 得到包含错误位置信息和控制流语句的行子集, 从而有效减小 LLM 关注的范围, 更好地修复错误程序.

4.4 方法总结

根据 LLM 在模糊测试不同阶段的应用, 上文分别对各类方法进行性详细地综述. 为了进一步理清 LLM 在不用应用任务、不用应用策略、不同测试对象、测试类型、测试语言以及使用的模型的使用详情, 表 3 对上文进行了如下统计. 基于 LLM 的模糊测试方法呈现出以下特点.

表 3 基于 LLM 的模糊测试方法总结

文献	应用任务	应用策略	测试对象	测试类型	测试语言	使用模型
LiBro[18]	TG	PE (F)	常规软件	●	Java	Codex
A3Test[49]	TG	FT	常规软件	◐	Java	PLBART
FuzzGPT[50]	TG	FT	深度学习库	●	Python	Codex, CodeGen
LLAMAFUZZ[51]	TG	FT	常规软件	◐	C/C++	Llama
Hashtroudi等人[52]	TG	FT	常规软件	◐	Java	CodeT5
CovRL[54]	TG	FT	JS引擎	◐	JavaScript	CodeT5
QTypist[55]	TG	PE (Z)	图形界面软件	●	Python	GPT-3
TitanFuzz[56]	TG DD	PE (Z)	深度学习库	●	General	Codex Incoder
AceCoder[57]	TG	PE (Z)	常规软件	◐	General	CodeGeex等
Asmita等人[58]	TG	PE (Z)	嵌入式软件	◐	C/C++	GPT-4
CEDAR[59]	TG DD	PE (F)	常规软件	◐	General	Codex
ChatAFL[60]	TG	PE (F)	协议软件	◐	C/C++	GPT-3.5-Turbo等
ASAP[61]	TG	PE (F)	常规软件	○	General	GPT-3.5-Turbo
PBT-GPT[62]	TG	PE (C)	常规软件	○	Python	GPT-4
Ackerman等人[63]	TG	PE (C)	解析器	◐	Python	GPT-4
Fuzz4All[26]	TG	PE (A)	常规软件	●	General	GPT-4, StarCoder
WhiteFox[64]	TG	PE (A)	编译器	○	C/C++	GPT-4, StarCoder
ChatUniTest[65]	TG	FU	常规软件	○	General	ChatGPT
ChatTester[66]	TG	FU	常规软件	○	Java	ChatGPT
TESTPILOT[67]	TG	FU	常规软件	○	JavaScript	GPT-3.5-Turbo
InputBlaster[68]	TG	FU	移动端软件	●	General	ChatGPT
Bugsplainer[69]	TG	FU	常规软件	◐	Python	CodeT5
LLM4Fuzz[70]	TG	FU	智能合约	◐	C/C++	GPT-4
KernelGPT[74]	TG	FU	系统内核	◐	C/C++	GPT-4
ChatFuzz[75]	TG	FU	常规软件	◐	General	ChatGPT

表 3 基于 LLM 的模糊测试方法总结 (续)

文献	应用任务	应用策略	测试对象	测试类型	测试语言	使用模型
CCTEST[76]	TG	FU	常规软件	◐	Python	GPT-Neo CodeGen
SearchGEM5[77]	TG	FU	模拟仿真软件	◐	C/C++	GPT-3.5-Turbo
CodaMosa[78]	TG	FU	常规软件	◐	Python	Codex
TICODER[79]	TG	FU	常规软件	◐	Python	Codex
Xia等人[80]	TG	FU	常规软件	◐	General	GPT-3.5-Turbo
Mastropaolo等人[82]	DD	FT	常规软件	◐	General	T5
Tufano等人[83]	DD	FT	常规软件	◐	Java	T5
DIVAS[87]	DD	PE (Z)	片上系统	◐	General	ChatGPT, BART
AID[94]	DD	FU	常规软件	◐	Python C++	ChatGPT
Li等人[95]	DD	FU	常规软件	●	Python	ChatGPT
Paul等人[84]	DD	FT	常规软件	◐	General	PLBART, CodeT5
ITiger[86]	DD	FT	常规软件	◐	General	BART
VUL-GPT[88]	DD	PE (F)	常规软件	◐	General	GPT-3.5-Turbo
AutoSD[89]	DD	PE (F)	常规软件	◐	Java	ChatGPT
LATTE[90]	DD	PE (C)	常规软件	◐	C/C++	GPT-4
AutoFL[91]	DD	PE (C)	常规软件	◐	Java	ChatGPT
Bui等人[92]	DD PF	PE (C)	常规软件	◐	General	PLBART, CodeT5等
SkipAnalyzer[93]	DD PF	PE (F)	常规软件	◐	Java	GPT-4, GPT-3.5-Turbo
CHEMFUZZ[96]	PF	PE (Z)	常规软件	◐	C++ Fortran	GPT-3.5等
REPEATNPR[97]	PF	FT	常规软件	◐	General	CodeT5等
Mashhadi等人[98]	PF	FT	常规软件	◐	Java	CodeBERT
InferFix[99]	PF	FT	常规软件	◐	General	Codex
Hao等人[100]	PF	FT	常规软件	◐	Java	CodeT5等
Pearce等人[101]	PF	PE (F)	常规软件	◐	C/C++	Codex等
NL2Fix[102]	PF	PE (Z)	常规软件	◐	Java	Codex等
PyDex[105]	PF	FU	常规软件	◐	Python	Codex

注: 应用任务中TG表示测试输入生成, DD表示缺陷检测, PF表示后模糊处理; 应用策略中FT表示微调, PE表示提示工程 (Z表示零样本学习, F表示少样本学习, C表示思维链, A表示自动提示), FU表示传统算法融合; 测试类型中●表示黑盒模糊测试, ○表示白盒模糊测试, ◐表示灰盒模糊测试; 在测试语言中General表示该方法可以适用于多种编程语言

4.4.1 应用任务与应用策略分析

从应用任务的角度分析, 当前 LLM 与模糊测试结合的方式主要分为 3 类: LLM 驱动的测试输入生成、缺陷检测和后模糊处理. 现有研究的重点集中在 LLM 用于测试输入生成, 这是因为测试输入的质量直接影响测试的覆盖率和有效性, 生成高质量的测试输入是提高测试效果较为有效的途径[106]. 相比之下, 缺陷检测和后模糊处理的研究较少, 原因可能在于这些模块更多依赖于测试执行后的数据分析, 而目前, LLM 更擅长基于上下文进行生成, 直接应用于检测和处理模块的效果尚不突出. 至于模糊测试的预处理模块和执行模块, 目前尚未有研究尝试将 LLM 应用于这两个环节. 可能的原因在于, 预处理和执行通常需要强大的系统层面控制能力和对底层结构的直接操作, 而 LLM 的工作原理更侧重于较高层次的语义层面的理解, 难以与此类底层任务高效结合. 因此, LLM 在这些环节的应用受到限制.

从应用 LLM 的策略来看, 常用策略包括数据微调、提示工程和传统算法融合这 3 类. 其中提示工程是目前最常见的策略, 原因在于其简便性. 提示工程不需要大量的额外数据集或复杂的模型微调, 能够直接通过设计提示来引导 LLM 生成期望的输出, 节省了开发成本和时间. 与数据微调相比, 提示工程可以在无需重新训练模型的情况下实现良好的效果, 尤其适用于实际应用场景中快速迭代的需求. 在提示工程中, 少样本提示被广泛采用, 这是因

为少样本提示能够通过几个示例就有效引导 LLM 输出符合需求的结果, 特别是在复杂任务中, 少样本提示能更精确地控制输出内容. 而零样本提示更多用于简单任务, 这是因为零样本提示依赖于 LLM 的通用能力, 对于复杂的、任务特定的输出, 零样本提示的有效性较低.

综上所述, 现有 LLM 与模糊测试的结合方式和策略主要受限于 LLM 的生成能力和任务复杂性的匹配度, 当前的研究重点和应用偏好反映了 LLM 在测试输入生成方面的优势, 以及提示工程在实际应用中的高效性.

4.4.2 测试对象与测试语言分析

从测试对象的角度来看, 基于 LLM 的模糊测试方法几乎涵盖了所有类型的软件, 包括常规软件、嵌入式软件、移动端软件、图形界面软件、深度学习库、系统内核、智能合约以及片上系统等多种复杂系统. 尽管这些研究对象范围广泛, 现有研究仍然主要集中在常规的开源软件和封装库上, 特别是那些拥有公开可用的数据集和代码库的软件系统. 这些开源资源为模糊测试提供了丰富的测试数据和场景, 降低了测试环境的构建成本, 并提高了测试的可重复性和验证能力. 然而, 这种依赖开源项目的倾向可能导致其他软件类型, 尤其是闭源商业软件、定制嵌入式系统和智能合约等领域的研究相对不足, 从而限制了 LLM 模糊测试方法在这些特殊领域的广泛适用性和推广.

从编程语言的角度分析, 基于 LLM 的模糊测试方法支持多种编程语言的测试, 这使得其能够适用于不同开发场景中的各种软件项目. 其中, Java、Python 和 C/C++是被测试最多的编程语言, 这与这些语言在软件开发中的广泛使用相一致. Java 和 Python 在企业级应用、数据科学和人工智能领域具有广泛的应用, C/C++则主要用于系统级开发和高性能计算场景. LLM 的语言支持不仅涵盖了这些主流语言, 还拓展到其他常见语言例如 Go 模糊测试[26]、JavaScript 模糊测试[54]等, 从而扩展了模糊测试在不同编程生态中的应用范围. 然而, 对于一些领域特定的编程语言 (如智能合约中的 Solidity 或 Verilog 等硬件描述语言), LLM 的模糊测试方法尚处于探索阶段, 相关研究仍需进一步深入, 以应对这些语言的特性及其对模糊测试带来的独特挑战.

进一步分析, LLM 模糊测试方法在不同编程语言和软件类型中的应用效果, 可能会受到软件系统复杂度、语言特性以及上下文窗口大小等因素的影响. 例如, 具备汇编特性的编程语言如 C/C++通常涉及复杂的内存管理和指针操作, 这对 LLM 的上下文理解提出了更高要求. 相反, Python 等编程语言因其动态特性和自动内存管理, 可能更适合利用 LLM 的推理能力实现更好的测试效果.

4.4.3 测试类型分析

根据测试方法对被测系统内部结构的了解程度, 基于 LLM 的模糊测试方法中大多数倾向于采用灰盒模糊测试 (占 74.5%), 仅有少部分为黑盒模糊测试 (占 13.7%) 和白盒模糊测试 (占 11.8%). 由于 LLM 在训练时并未专门针对特定的被测试软件进行学习, 因此在完全不提供任何系统内部结构信息的情况下, 黑盒测试效果往往不佳. 此外, LLM 的上下文窗口存在长度限制, 难以完整理解被测试系统的全貌, 白盒测试难以实现. 因此, 大多数研究者选择灰盒测试方法, 通过在提示中提供少量的内部结构信息, 以在上下文窗口和内部信息之间取得平衡.

黑盒方法虽然不直接分析被测程序的内部结构, 但可以通过参考历史文档或相似程序来提高程序理解. 例如 LiBro[18]利用基于错误报告构造的 Markdown 文档、FuzzGPT[50]则通过微调相似的代码片段来增强 LLM 的测试能力. 而使用最少的白盒方法通常借助 LLM 之外的工具对被测程序进行内部结构分析. 并根据上下文窗口的长度选择部分关键信息作为提示. 以克服上下文窗口的长度限制. 例如 ChatTester[61]通过静态分析提取函数的完整上下文, 并不断调整所选择的提示内容, 以在有限的上下文窗口内最大化信息量.

4.4.4 使用模型分析

从上述方法选择的模型角度分析, GPT 系列是应用最为广泛的模型. 在各项研究中, GPT-3.5-Turbo[107]和 GPT-4[108]均展现出卓越的生成和推理能力. 例如, Fuzz4All[26]利用先进的 GPT-4 较为准确地实现了对提示信息上下文的推理总结. 其次是 Codex 模型[109]和 T5 系列的模型. Codex 是基于 GPT-3 开发并于 2021 年 9 月发布的模型, 由于推出较早, 因此在早期研究工作应用较多. 而 T5 系列[110]是开源 LLM, 由于其开源, 研究人员可以轻松地使用特定领域的数据进行预训练和微调, 以获得更好的性能, 同时对于其执行速度较快, 被广泛用于测试输入和测试程序的掩码变异环节. 例如 CovRL[54]使用 T5 模型对输入中的某些部分进行遮盖 (即添加掩码 MASK), 并在掩码处选择变

异算子完成预测填补, 来生成新的测试输入.

5 挑战与展望

针对传统模糊测试方法和基于深度学习的模糊测试方法存在的不足, 研究人员利用 LLM 驱动模糊测试, 在一定程度上弥补了原有方法的不足. 本文对目前工作的研究成果进行如下总结.

首先, 传统算法和基于深度学习方法生成的测试用例存在一些局限性: 通过编译的测试用例较少、测试用例的测试效果较差. 基于 LLM 的模糊测试方法通过微调和提示, 融合传统算法, 从而指导生成正确性更高且测试效果更好的测试用例.

其次, 考虑到传统算法和基于深度学习方法对预言知识理解不足以及缺陷检测能力差的情况, 基于 LLM 的模糊测试方法通过断言数据训练和构建提示的方式, 能够引导 LLM 生成符合要求的测试预言. 同时, 基于 LLM 的模糊测试方法具备学习缺陷特征的能力, 并利用 LLM 强大的推理能力, 实现根据执行信息检测甚至预测程序的潜在缺陷.

最后, 为了提高模糊测试的自动化水平, 基于 LLM 的模糊测试能够充分利用执行过程和缺陷检测过程的信息, 实现模糊测试的后模糊处理, 包括测试报告生成、模糊测试配置更新以及程序修复任务.

虽然 LLM 具有较强的生成和推理能力, 但将其应用到模糊测试领域仍存在巨大的提升空间. 随着 LLM 的不断发展和模糊测试技术的提升, 以 LLM 为驱动的模糊测试具有良好的研究前景. 接下来, 本文将对基于 LLM 的模糊测试所面临的挑战进行总结, 并进行展望.

5.1 挑　战

经过对现有文献的深入分析, 当前基于 LLM 的模糊测试面临着一系列挑战, 这些挑战如表 4 所示.

表 4　基于 LLM 的模糊测试的挑战

类型	文献	LL	PD	TL	UP	UC	CR	SL	OL	SA	BL
测试输入生成	A3Test[49]	○	○	●	●	●	●	●	◐	◐	◐
	FuzzGPT[50]	○	●	●	●	●	●	●	◐	◐	◐
	LLAMAFUZZ[51]	○	○	●	●	●	●	●	◐	◐	◐
	Hashtroudi等人[52]	○	○	●	●	●	●	●	◐	◐	◐
	CovRL[54]	○	○	●	●	●	●	●	◐	◐	◐
	QTypist[55]	○	●	○	●	●	●	●	◐	◐	◐
	TitanFuzz[56]	○	●	○	●	●	●	●	◐	◐	◐
	AceCoder[57]	○	●	○	●	●	●	●	◐	◐	◐
	LiBro[18]	○	●	○	●	●	●	●	◐	◐	◐
	Asmita等人[58]	○	●	○	●	●	●	●	◐	◐	◐
	CEDAR[59]	○	●	○	●	●	●	●	◐	◐	◐
	ChatAFL[60]	○	●	○	●	●	●	●	◐	◐	◐
	ASAP[61]	○	●	○	●	●	●	●	◐	◐	◐
	PBT-GPT[62]	○	●	○	●	●	●	●	◐	◐	◐
	Ackerman等人[63]	○	●	○	●	●	●	●	◐	◐	◐
	Fuzz4All[26]	○	●	○	●	●	●	●	◐	◐	◐
	WhiteFox[64]	○	●	○	●	●	●	●	◐	◐	◐
	ChatUniTest[65]	●	●	○	●	●	●	●	◐	◐	◐
	ChatTester[66]	○	●	○	●	●	●	●	◐	◐	◐
	TESTPILOT[67]	○	●	○	●	●	●	●	◐	◐	◐
	InputBlaster[68]	○	●	○	●	●	●	●	◐	◐	◐
	Bugsplainer[69]	○	○	●	●	○	○	○	◐	◐	◐

表 4 基于 LLM 的模糊测试的挑战 (续)

类型	文献	LL	PD	TL	UP	UC	CR	SL	OL	SA	BL
测试输入生成	LLM4Fuzz[70]	○	●	○	●	●	●	●	◐	◐	◐
	KernelGPT[74]	○	●	○	●	●	●	●	◐	◐	◐
	ChatFuzz[75]	○	●	○	●	●	●	●	◐	◐	◐
	CCTEST[76]	○	●	○	●	●	○	○	◐	◐	◐
	SearchGEM5[77]	○	●	○	●	●	●	●	◐	◐	◐
	CodaMosa[78]	○	●	○	●	●	●	●	◐	◐	◐
	TICODER[79]	○	●	○	●	●	●	●	◐	◐	◐
	Xia等人[80]	○	●	○	●	○	○	○	◐	◐	◐
缺陷检测	Mastropaolo等人[82]	○	○	●	●	●	○	○	◐	◐	◐
	Tufano等人[83]	○	○	●	●	●	○	○	◐	◐	◐
	DIVAS[87]	○	○	○	●	○	○	○	◐	◐	◐
	AID[94]	○	●	○	●	●	●	●	◐	◐	◐
	Li等人[95]	○	●	○	●	●	●	●	◐	◐	◐
	Paul等人[84]	○	○	●	●	○	●	●	◐	◐	◐
	ITiger[86]	○	○	●	●	○	○	○	◐	◐	◐
	VUL-GPT[88]	○	●	○	●	○	○	○	◐	◐	◐
	AutoSD[89]	○	●	○	●	○	○	○	◐	◐	◐
	LATTE[90]	○	●	○	●	○	○	○	◐	◐	◐
	AutoFL[91]	●	●	○	●	○	○	○	◐	◐	◐
	Bui等人[92]	○	○	●	●	○	○	○	◐	◐	◐
	SkipAnalyzer[93]	○	●	○	●	○	○	○	◐	◐	◐
后模糊处理	CHEMFUZZ[96]	○	●	○	●	●	●	○	◐	◐	◐
	REPEATNPR[97]	○	○	●	●	○	○	○	◐	◐	◐
	Mashhadi等人[98]	○	○	●	●	○	○	○	◐	◐	◐
	InferFix[99]	○	●	●	●	○	●	●	◐	◐	◐
	Hao等人[100]	○	○	●	●	○	○	○	◐	◐	◐
	Pearce等人[101]	○	●	○	●	○	○	○	◐	◐	◐
	NL2Fix[102]	○	●	○	●	●	●	●	◐	◐	◐
	PyDex[105]	○	●	○	●	●	○	○	◐	◐	◐

注: ○: 该方法可以有效解决该挑战, ◐: 目前虽然可以缓解该挑战但仍然未完全解决, ●: 该方法未涉及该挑战的解决; LL: LLM的Token长度限制, PD: LLM高效的提示设计方式, TL: LLM微调的软硬件限制, UP: LLM对被测程序代码理解不足, UC: LLM对测试输入结构理解不足, CR: LLM生成测试输入的正确性、可读性不足, SL: LLM生成测试输入的规范性、逻辑性不足, OL: 基于LLM的模糊测试对象范围有限, SA: 模糊测试中LLM应用的充分性不足, BL: 缺少统一的基准和评估方法

具体而言, 表 4 中的“LL”指的是 LLM 的 Token 长度限制, 各类 LLM 均存在一定的上下文长度限制, 而完整的程序代码和包含大量信息的提示通常超出了 LLM 的 Token 长度. “PD”表示在引导 LLM 完成特定任务的过程中, 需要设计高效的提示方式. “TL”则指在对 LLM 进行微调时, 存在软硬件的限制, 包括 LLM 必须开源、优质的训练数据以及足够的微调计算资源. “UP”是指 LLM 对被测程序的语法、语义、代码结构和函数调用关系等的理解存在不足. “UC”表示 LLM 在理解测试输入中的断言语句、验证机制和测试签名等方面存在欠缺. “SL”指 LLM 生成的输入在同时满足数据类型、输入格式和数据逻辑等多方面要求方面存在不足. 从表 4 基于 LLM 的模糊测试的挑战中可以看出, 以上 7 类问题在综述部分中已经得到有效解决.

然而, 仍存在 3 类问题尚未有较为综合详尽的解决方案. 首先, 基于 LLM 的模糊测试对象范围有限; 其次, LLM 在模糊测试执行各环节中应用不充分. 最后, 缺乏统一的基准和评估方法, 使得对 LLM 模糊测试性能的全面评估变得困难. 同时, 对于 LLM 本身, 现有的微调和提示方法仍需要进一步深入研究, 以更好地发挥 LLM 在模糊

测试中的性能. 对当前研究方法的调研显示, 这些问题仍然存在一定的局限性. 因此, 未来需要在这些方面展开深入的研究工作

5.2 展　望

随着 LLM 的不断发展, 将其应用于模糊测试领域的研究备受关注. 鉴于前文提到的挑战和相关文献的探讨, 图 7 展示了本文从 4 个角度对 LLM 驱动的模糊测试工作进行展望.

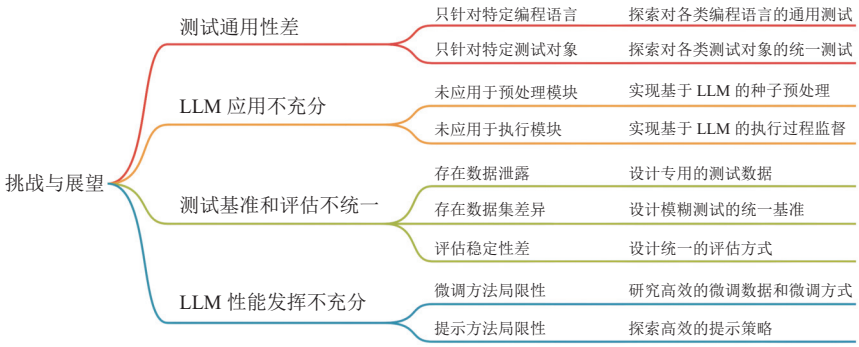

图 7　基于 LLM 的模糊测试的挑战与展望

5.2.1　拓展模糊测试下游任务范围

目前, 基于 LLM 的模糊测试仍存在一定的局限性.

在第 4 节中提到的大多数文献仅能针对特定编程语言 (如 Java、Python、C 等) 或者特定测试任务 (如深度学习库、移动应用程序界面等) 进行模糊测试. 目前, 仍缺少通用的基于 LLM 模糊测试方法, 能适用于常见的编程语言或多种测试对象.

考虑到现有的软件系统通常由多种编程语言编写, 会涉及不同类型的测试, 例如 Web 网站系统的后端采用 Java 语言, 需要进行 API 调用测试, 同时前端往往会由 JavaScript 语言编写, 需要进行图形化界面测试. 此外, 不同编程语言之间还存在调用关系, 例如 Python 可以使用 ctypes 库调用 C/C++编写的程序, Java 使用 JDBC (Java database connectivity) 驱动执行 SQL 语言对数据库进行增删改查的数据操作.

由于各编程语言和测试任务具有不同的特征, 传统模糊测试方法很难设计出通用的模糊测试工具. 然而, 随着 LLM 的不断发展, 逐渐具备理解不同编程语言和测试任务的能力. 基于 LLM 的模糊测试方法有望实现通用的模糊测试方法.

首先, 对编程语言进行通用测试. Fuzz4All[26]充分利用 GPT-4 强大的信息总结和提取能力, 通过 GPT 对各种编程语言上下文信息进行蒸馏提取, 并自动提供提示给另一个 LLM, 实现对基于 C、C++、Go、SMT2、Java 和 Python 等 6 种不同语言程序的测试. 评估结果表明, 通用模糊测试相对于特定语言模糊测试具有更好的覆盖率. 随着更强大的 LLM 的涌现, 其在各类编程语言程序的测试任务上可能表现更为出色.

其次, 对不同测试对象进行通用测试. 在基于 LLM 的模糊测试中, 对于不同的测试对象, 例如 Web 网站系统、具有 GUI 界面的移动应用等, 需要详细分析不同测试对象的特点, 并采用不同的微调或提示方式. 而之后的研究可以分析不同的测试对象, 设计通用模块, 生成符合要求的提示, 实现针对多种测试对象的通用测试.

5.2.2　LLM 应用于模糊测试的多个阶段

通过表 2 可以看出, 目前基于 LLM 的方法主要将 LLM 应用于测试输入、测试预言等生成任务, 较少的研究工作将 LLM 用于缺陷检测和程序修复等推理任务. 然而, 根据图 3 所示, 模糊测试的一般工作流程还包括预处理模块、测试执行模块中的多个模糊测试步骤. 因此, 未来的研究工作可以扩展 LLM 在模糊测试中的应用范围, 以全面提升模糊测试的智能化和自动化程度.

首先, 将 LLM 用于预处理阶段. 由于模糊测试尚未执行, 预处理阶段通常需要对种子进行静态分析. 考虑到

LLM 具有强大的程序分析和理解能力, 可通过使用插桩前后的程序代码示例对 LLM 进行微调, 或者根据插桩的执行过程向 LLM 提供提示, 使 LLM 分析程序结构并进行插桩. 相对于深度学习方法和传统方法, LLM 的推理能力更强, 有助于找到合适的插桩位置, 提高代码缺陷的检测率. 对于程序的选择和修剪, LLM 可以发现具有相似结构和功能的代码片段, 为模糊测试提供更多有价值的种子, 或结合统计分析方法指导种子的选择和修改, 提高模糊测试的执行效率.

其次, 将 LLM 应用于测试执行阶段. 尽管现有模糊测试技术已经能够实现较为自动化的执行和执行过程的检测, 但这仅是将测试输入的信息输入给测试程序, 保存执行过程中的信息, 待后续的缺陷检测阶段再进一步分析. 然而, 在模糊测试的一般工作流程中, 存在预处理、测试执行和其他多个步骤. 将 LLM 引入测试执行模块, LLM 不仅能够存储执行信息, 还能进一步分析. 对于模糊测试程序本身的执行错误, LLM 可以及时识别并进行修复和再次执行, 避免了只有在程序完全退出后才能识别和修复的问题. 此外, 对于存在交互的程序, LLM 的引入也有助于提高测试的自动化程度.

5.2.3 标准化测试基准与指标

在当前调研的文献中, 我们注意到各种研究尚未提出一套统一且可靠的测试基准. 在实验评估中, 由于缺乏专门针对基于 LLM 的模糊测试的实验基准, 研究工作通常采用相应领域或编程语言的评估基准. 例如, 在代码生成任务中, 使用了 HumanEval[103]基准; 缺陷定位任务采用了 Quixbugs[111]基准; Java 语言模糊测试任务使用了 Defects4J[112]基准, 而 Python 语言则可以使用 MBPP[113]作为模糊测试的基准. 然而, 使用非统一的评估基准可能导致对现有评估工作的误导.

首先, 我们要考虑数据泄露问题, 即 LLM 在预训练过程中可能学到了上述基准数据. 在机器学习中, 防止训练和测试数据的污染是基本原则. 数据泄露可能导致机器学习模型的评估错误. 研究表明, LLM 也会受到数据泄露的影响, 包括标签泄露和无标签泄露[114]. 为了研究 LLM 中的数据泄露问题, Aiyappa 等人[115]在研究中分析了 ChatGPT 存在数据泄露的原因. Jiang 等人[116]检查了 LLM 训练所使用的数据源 CodeSearchNet 和 BigQuery, 发现 Defects4J 基准测试所使用的 4 个存储库也包含在 CodeSearchNet 中, 整个 Defects4J 存储库也被 BigQuery 包含. 因此, LLM 可能在训练中已经学到了相关基准数据. 在 LLM 的提示中使用的零样本学习和少样本学习可能不准确, 从而可能导致基于 LLM 的模糊测试的应用效果并不如第 4 节中所介绍的方法评估结果所示. 因此, 有必要为基于 LLM 的模糊测试设计一个独立的评估基准, 以规避应用 LLM 时潜在的数据泄露问题.

其次, 当前方法使用的数据集之间存在差异. 在 Siddiq 等人的研究中[16], ChatGPT-3.5、CodeGen、Codex 在 HumanEval 上生成的测试正确率分别为 52.3%、23.9% 和 77.5%, 而在 SF110 数据集上的正确率却分别为 6.9%、30.2% 和 46.5%, 呈现明显的差异. 同样, 在使用 GPT-3.5-Turbo 生成测试用例时, 对于 gitlab-js 项目的测试通过率只有 9.9%, 而对于测试数据集中的 image-downloader 项目, 测试通过率却高达 80.0%[66]. 不同数据集之间的性能差异表明了当前基于 LLM 方法的局限性, 同时也表明无法客观地比较各种 LLM 的性能. 因此, 未来的工作需要设计一个统一的评估基准, 更好地衡量各种基于 LLM 方法的性能, 以帮助研究人员能够深入探讨不同方法之间的差异, 推动基于 LLM 的模糊测试技术的进步.

最后, 评估结果的稳定性较差. 当前研究表明, 提示中的简单修改可能会导致 LLM 性能的巨大变化. 由于在模糊测试中, 大多数提示的内容都存在差异, 甚至在相同模型、参数和数据集下, 生成的提示也可能不同, 因此评估结果很难具有稳定性. 鉴于 LLM 的特殊性, 未来的工作应该根据 LLM 的特点设计更科学的评估基准, 以避免因实验偶然性而对 LLM 性能进行错误评估.

5.2.4 提升 LLM 测试领域性能

当前的 LLM 已经展现出卓越的生成和推理能力, 然而, 如何充分发挥 LLM 在模糊测试中的性能仍然是一个未来值得研究的方向. 正如第 2.2 节中所述, 目前提高 LLM 在模糊测试中的表现的方法主要包括微调、提示以及与其他技术的结合.

尽管 LLM 的微调方法通常在其早期应用较为常见, 但在实际研究工作中, 一些研究者可能更倾向于避免使用商业的非开源 LLM, 以防止潜在的研究数据泄漏问题. 因此, 对于开源 LLM 模型的微调仍然在未来的 LLM 应用

研究中具有重要意义. 对未来 LLM 微调的研究可以从两个方面入手, 一是构建高质量的微调数据集, 二是设计更符合现有 LLM 架构的微调方法.

在微调的数据集方面, 目前的方法通常直接使用已有数据集进行 LLM 的训练, 例如 Alagarsamy 等人[49]和 Hashtroudi 等人[52]在其研究中都采用了广泛引用的 Methods2Test 数据集进行模型微调. 然而, 这些方法未完全实现测试任务和微调数据的精准匹配. 鉴于手动创建特定要求的数据集耗时且繁琐, 未来的工作需要进一步研究自动获取微调数据集的方法, 例如尝试从代码仓库中提取目标信息作为微调数据. 例如, Utopia[117]利用从代码仓库中获得的单元测试数据来驱动程序的生成. 在微调方法方面, 第 2.2.1 节中描述的方法通常简单地微调少量或额外的模型参数, 而关于 LLM 如何理解输入数据的方面目前尚未有专门的研究, 例如在微调方法中如何将数据集输入 LLM, 使其更好地理解微调数据中代码的数据流和控制流.

对于提示方法, 除了第 2.2.2 节中介绍的 3 类提示工程方法外, 还有其他提示方式, 例如 Tree-of-thoughts[118]、Active-prompt[119]、ReAct-prompt[120]和 GraphPrompt[121]等. 在目前 LLM 提示方法的研究中, 未来的工作可以进一步深入研究提示模板, 例如 GraphPrompt 使用图形或视觉结构来表示信息, 将代码程序涉及的依赖关系、控制流、数据流、状态转换等信息转换为图形结构, 以便 LLM 更有效地理解被测软件. 随着各类多模态 LLM 的发展, 不仅可以利用图形, 还可以结合声音、视频等多种形式, 从而提高提示的自动化程度, 增强 LLM 的理解能力.

关于与其他技术结合的方法, 如第 4.1.3 节所述, 目前的方法已经将 LLM 与反馈验证、程序分析、变异算子、统计分析等方法结合起来. 随着软件工程和机器学习领域的发展, 各种问题都有了相关的技术, 这些技术均可以尝试与 LLM 结合, 发挥 LLM 和其他技术的共同优势. 为了帮助 LLM 更好地理解程序中 API 的调用关系, 可以引入编译中的 DU 链分析, 使用 DU 链将属性的定义和所有的访问连接起来, 跟踪每个 API 的使用情况, 以实现对程序数据流的充分理解. 除了传统的程序分析方法, 还可以结合机器学习的方法, 例如使用随机森林中的投票思想对种子程序进行分类, 从而选出最小的种子集.

6 总 结

本文针对基于 LLM 的模糊测试的现有研究进行了综述. 文章首先介绍了模糊测试的基本架构和模糊测试中常用的 LLM 技术. 结合基于深度学习的模糊测试方法的不足, 本文从而引出基于 LLM 的模糊测试方法, 并对相关文献进行分类和描述. 根据对相关文献的分析和总结, 本文系统性地给出了基于 LLM 的模糊测试所面临的挑战, 并提供了 4 个可以扩展的研究方向.

致谢 感谢 CCF-华为胡杨林基金对本课题的支持!

References:

[1] Xiao XS, Li SH, Xie T, Tillmann N. Characteristic studies of loop problems for structural test generation via symbolic execution. In: Proc. of the 28th IEEE/ACM Int'l Conf. on Automated Software Engineering. Silicon Valley: IEEE, 2013. 246–256. [doi: 10.1109/ASE.2013.6693084]

[2] Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-directed random test generation. In: Proc. of the 29th Int'l Conf. on Software Engineering. Minneapolis: IEEE, 2007. 75–84. [doi: 10.1109/ICSE.2007.37]

[3] Pan MX, Huang A, Wang GX, Zhang T, Li XD. Reinforcement learning based curiosity-driven testing of Android applications. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. New York: ACM, 2020. 153–164. [doi: 10.1145/3395363.3397354]

[4] OpenAI. ChatGPT. 2024. https://openai.com/chatgpt/download/

[5] Touvron H, Martin L, Stone K, *et al*. Llama 2: Open foundation and fine-tuned chat models. arXiv:2307.09288, 2023.

[6] Nijkamp E, Hayashi H, Xiong CM, Savarese S, Zhou YB. CodeGen2: Lessons for training LLMs on programming and natural languages. arXiv:2305.02309, 2023.

[7] DeepSeek-AI. DeepSeek Coder. 2023. https://github.com/deepseek-ai/DeepSeek-Coder

[8] Wang JJ, Huang YC, Chen CY, Liu Z, Wang S, Wang Q. Software testing with large language models: Survey, landscape, and vision.

IEEE Trans. on Software Engineering, 2024, 50(4): 911–936. [doi: 10.1109/TSE.2024.3368208]

[9] Zhao XQ, Qu HP, Xu JL, Li XH, Lv WJ, Wang GG. A systematic review of fuzzing. Soft Computing, 2024, 28(6): 5493–5522. [doi: 10.1007/s00500-023-09306-2]

[10] Wang CZ, Yang YH, Gao CY, Peng Y, Zhang HY, Lyu MR. No more fine-tuning? An experimental evaluation of prompt tuning in code intelligence. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Singapore: ACM, 2022. 382–394. [doi: 10.1145/3540250.3549113]

[11] Mo S, Cho M, Shin J. Freeze the discriminator: A simple baseline for fine-tuning GANs. arXiv:2002.10964, 2020.

[12] Li XL, Liang P. Prefix-tuning: Optimizing continuous prompts for generation. In: Proc. of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th Int'l Joint Conf. on Natural Language Processing. Association for Computational Linguistics, 2021. 4582–4597. [doi: 10.18653/v1/2021.acl-long.353]

[13] Hu EJ, Shen YL, Wallis P, Allen-Zhu Z, Li YZ, Wang SA, Wang L, Chen WZ. LoRA: Low-rank adaptation of large language models. In: Proc. of the 10th Int'l Conf. on Learning Representations. OpenReview.net, 2022.

[14] Dettmers T, Pagnoni A, Holtzman A, Zettlemoyer L. QLoRA: Efficient finetuning of quantized LLMs. In: Proc. of the 37th Conf. on Neural Information Processing Systems. New Orleans: NeurIPS, 2023. 3982–3992.

[15] Wang W, Zheng VW, Yu H, Miao CY. A survey of zero-shot learning: Settings, methods, and applications. ACM Trans. on Intelligent Systems and Technology (TIST), 2019, 10(2): 13. [doi: 10.1145/3293318]

[16] Siddiq ML, Santos JCS, Tanvir RH, Ulfat N, Al Rifat F, Lopes VC. Exploring the effectiveness of large language models in generating unit tests. arXiv:2305.00418, 2024.

[17] Wang YQ, Yao QM, Kwok JT, Ni LM. Generalizing from a few examples: A survey on few-shot learning. ACM Computing Surveys (CSUR), 2021, 53(3): 63. [doi: 10.1145/3386252]

[18] Kang S, Yoon J, Yoo S. Large language models are few-shot testers: Exploring LLM-based general bug reproduction. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 2312–2323. [doi: 10.1109/ICSE48619.2023.00194]

[19] Wei J, Wang XZ, Schuurmans D, Bosma M, Ichter B, Xia F, Chi EH, Le QV, Zhou D. Chain-of-thought prompting elicits reasoning in large language models. In: Proc. of the 36th Conf. on Neural Information Processing Systems. New Orleans: NeurIPS, 2022. 24824–24837.

[20] Zhang ZS, Zhang A, Li M, Smola A. Automatic chain of thought prompting in large language models. In: Proc. of the 11th Int'l Conf. on Learning Representations. Kigali: OpenReview.net, 2023.

[21] Yin BS, Hu NY. Time-CoT for enhancing time reasoning factual question answering in large language models. In: Proc. of the 2024 Int'l Joint Conf. on Neural Networks. Yokohama: IEEE, 2024. 1–8. [doi: 10.1109/IJCNN60899.2024.10650885]

[22] Zhou YC, Muresanu AI, Han ZW, Paster K, Pitis S, Chan H, Ba J. Large language models are human-level prompt engineers. In: Proc. of the 11th Int'l Conf. on Learning Representations. Kigali: OpenReview.net, 2023.

[23] Kojima T, Gu SS, Reid M, Matsuo Y, Iwasawa Y. Large language models are zero-shot reasoners. In: Proc. of the 36th Conf. on Neural Information Processing Systems. New Orleans: NeurIPS, 2022.

[24] Achiam J, Adler S, Agarwal S, *et al*. GPT-4 technical report. arXiv:2303.08774, 2024.

[25] Li R, Allal LB, Zi YT, *et al*. StarCoder: May the source be with you! Trans. on Machine Learning Research, 2023, 12: 1–43.

[26] Xia CS, Paltenghi M, Tian JL, Pradel M, Zhang LM. Fuzz4All: Universal fuzzing with large language models. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 126. [doi: 10.1145/3597503.3639121]

[27] Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks. Communications of the ACM, 2017, 60(6): 84–90. [doi: 10.1145/3065386]

[28] Graves A. Generating sequences with recurrent neural networks. arXiv:1308.0850, 2014.

[29] Yin ZY, Shao JS, Hussain MJ, Hao YJ, Chen Y, Zhang XF, Wang L. DPG-LSTM: An enhanced LSTM framework for sentiment analysis in social media text based on dependency parsing and GCN. Applied Sciences, 2023, 13(1): 354. [doi: 10.3390/app13010354]

[30] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proc. of the 31st Int'l Conf. on Neural Information Processing Systems. Long Beach: Curran Associates Inc., 2017. 6000–6010.

[31] Hassan H, Aue A, Chen C, Chowdhary V, Clark J, Federmann C, Huang XD, Junczys-Dowmunt M, Lewis W, Li M, Liu SJ, Liu TY, Luo RQ, Menezes A, Qin T, Seide F, Tan X, Tian F, Wu LJ, Wu SZ, Xia YC, Zhang DD, Zhang ZR, Zhou M. Achieving human parity on automatic Chinese to English news translation. arXiv:1803.05567, 2018.

[32] LeCun Y, Bottou L, Bengio Y, Haffner P. Gradient-based learning applied to document recognition. Proc. of the IEEE, 1998, 86(11): 2278–2324. [doi: 10.1109/5.726791]

[33] Graves A. Long short-term memory. In: Graves A, ed. Supervised Sequence Labelling with Recurrent Neural Networks. Berlin:

Springer, 2012. 37–45. [doi: 10.1007/978-3-642-24797-2_4]

[34] Godefroid P, Peleg H, Singh R. Learn&Fuzz: Machine learning for input fuzzing. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering. Urbana: IEEE, 2017. 50–59. [doi: 10.1109/ASE.2017.8115618]

[35] Zong PY, Lv T, Wang DW, Deng ZZ, Liang RG, Chen K. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In: Proc. of the 29th USENIX Conf. on Security Symp. USENIX Association, 2020. 127.

[36] Monsefi AK, Zakeri B, Samsam S, Khashehchi M. Performing software test oracle based on deep neural network with fuzzy inference system. In: Proc. of the 2019 Int'l Congress on High-performance Computing and Big Data Analysis. Tehran: Springer, 2019. 406–417. [doi: 10.1007/978-3-030-33495-6_31]

[37] Li J, He PJ, Zhu JM, Lyu MR. Software defect prediction via convolutional neural network. In: Proc. of the 2017 IEEE Int'l Conf. on Software Quality, Reliability and Security. Prague: IEEE, 2017. 318–328. [doi: 10.1109/QRS.2017.42]

[38] Fioraldi A, Maier D, Eißfeldt H, Heuse M. AFL++: Combining incremental steps of fuzzing research. In: Proc. of the 14th USENIX Conf. on Offensive Technologies. USENIX Association, 2020. 10: 1–10.

[39] Dias T, Batista A, Maia E, Praça I. TestLab: An intelligent automated software testing framework. In: Mehmood R, Alves V, Praça I, Wikarek J, Parra-Domínguez J, Loukanova R, de Miguel I, Pinto T, Nunes R, Ricca M, eds. Proc. of the 20th Int'l Conf. of Special Sessions I on Distributed Computing and Artificial Intelligence. Cham: Springer, 2023. 355–364. [doi: 10.1007/978-3-031-38318-2_35]

[40] Zhang ZN, Klees G, Wang E, Hicks M, Wei SY. Fuzzing configurations of program options. ACM Trans. on Software Engineering and Methodology, 2023, 32(2): 53. [doi: 10.1145/3580597]

[41] Li Y, Wang SH, Nguyen TN. DEAR: A novel deep learning-based approach for automated program repair. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 511–523. [doi: 10.1145/3510003.3510177]

[42] Li SQ, Xie XF, Lin Y, Li YK, Feng RT, Li XH, Ge WM, Dong JS. Deep learning for coverage-guided fuzzing: How far are we? IEEE Trans. on Dependable and Secure Computing. [doi: 10.1109/TDSC.2022.3200525]

[43] Miao SW, Wang J, Zhang C, Lin ZQ, Gong JX, Zhang XJ. Deep learning in fuzzing: A literature survey. In: Proc. of the 2nd IEEE Int'l Conf. on Electronic Technology, Communication and Information. Changchun: IEEE, 2022. 220–223. [doi: 10.1109/ICETCI55101.2022.9832143]

[44] Riccio V, Tonella P. When and why test generators for deep learning produce invalid inputs: An empirical study. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 1161–1173. [doi: 10.1109/ICSE48619.2023.00104]

[45] Common Weakness Enumeration. 2024. https://cwe.mitre.org/

[46] Deng X, Ye W, Xie R, Zhang SK. Survey of source code bug detection based on deep learning. Ruan Jian Xue Bao/Journal of Software, 2023, 34(2): 625–654 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6696.htm [doi: 10.13328/j.cnki.jos.006696]

[47] Lutellier T, Pham HV, Pang L, Li YT, Wei MS, Tan L. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. New York: ACM, 2020. 101–114. [doi: 10.1145/3395363.3397369]

[48] Zhang QJ, Fang CR, Ma YX, Sun WS, Chen ZY. A survey of learning-based automated program repair. ACM Trans. on Software Engineering and Methodology, 2024, 33(2): 55. [doi: 10.1145/3631974]

[49] Alagarsamy S, Tantithamthavorn C, Aleti A. A3Test: Assertion-augmented automated test case generation. Information and Software Technology, 2024, 176: 107565. [doi: 10.1016/j.infsof.2024.107565]

[50] Deng YL, Xia CS, Yang CY, Zhang SD, Yang SJ, Zhang LM. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 70. [doi: 10.1145/3597503.3623343]

[51] Zhang HX, Rong YY, He YF, Cehn H. LLAMAFUZZ: Large language model enhanced greybox fuzzing. arXiv:2406.07714, 2024.

[52] Hashtroudi S, Shin J, Hemmati H, Wang S. Automated test case generation using code models and domain adaptation. arXiv:2308.08033v1, 2024.

[53] Tufano M, Deng SK, Sundaresan N, Svyatkovskiy A. Methods2Test: A dataset of focal methods mapped to test cases. In: Proc. of the 19th Int'l Conf. on Mining Software Repositories. Pittsburgh: ACM, 2022. 299–303. [doi: 10.1145/3524842.3528009]

[54] Eom J, Jeong S, Kwon T. Fuzzing javascript interpreters with coverage-guided reinforcement learning for LLM-based mutation. In: Proc. of the 33rd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Vienna: ACM, 2024. 1656–1668. [doi: 10.1145/3650212.3680389]

[55] Liu Z, Chen CY, Wang JJ, Che X, Huang YK, Hu J, Wang Q. Fill in the blank: Context-aware automated text input generation for mobile GUI testing. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 1355–1367. [doi: 10.1109/ICSE48619.2023.00119]

[56] Deng YL, Xia CS, Peng HR, Yang CY, Zhang LM. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 423–435. [doi: 10.1145/3597926.3598067]

[57] Li J, Zhao YF, Li YM, Li G, Jin Z. AceCoder: Utilizing existing code to enhance code generation. arXiv:2303.17780, 2023.

[58] Asmita, Oliinyk Y, Scott M, Tsang R, Fang CZ, Homayoun H. Fuzzing BusyBox: Leveraging LLM and crash reuse for embedded bug unearthing. In: Proc. of the 33rd USENIX Security Symp. Philadelphia: USENIX Association, 2024.

[59] Nashid N, Sintaha M, Mesbah A. Retrieval-based prompt selection for code-related few-shot learning. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 2450–2462. [doi: 10.1109/ICSE48619.2023.00205]

[60] Meng RJ, Mirchev M, Böhme M, Roychoudhury A. Large language model guided protocol fuzzing. In: Proc. of the 31st Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2024. [doi: 10.14722/ndss.2024.24556]

[61] Ahmed T, Pai KS, Devanbu P, Devanbu P, Barr ET. Improving few-shot prompts with relevant static analysis products. arXiv:2304.06815v1, 2024.

[62] Vikram V, Lemieux C, Sunshine J, Padhye R. Can large language models write good property-based tests? arXiv:2307.04346, 2024.

[63] Ackerman J, Cybenko G. Large language models for fuzzing parsers (registered report). In: Proc. of the 2nd Int'l Fuzzing Workshop. Seattle: ACM, 2023. 31–38. [doi: 10.1145/3605157.3605173]

[64] Yang CY, Deng YL, Lu RY, Yao JY, Liu JW, Jabbarvand R, Zhang LM. WhiteFox: White-box compiler fuzzing empowered by large language models. Proc. of the ACM on Programming Languages, 2024, 8(OOPSLA2): 296. [doi: 10.1145/3689736]

[65] Chen YH, Hu ZH, Zhi C, Han JX, Deng SG, Yin JW. ChatUniTest: A framework for LLM-based test generation. In: Proc. of the 32nd ACM Int'l Conf. on the Foundations of Software Engineering. Porto de Galinhas: ACM, 2024. 572–576. [doi: 10.1145/3663529.3663801]

[66] Yuan ZQ, Liu MW, Ding SJ, Wang KX, Chen YX, Peng X, Lou YL. Evaluating and improving ChatGPT for unit test generation. Proc. of the ACM on Software Engineering, 2024, 1(FSE): 76. [doi: 10.1145/3660783]

[67] Schäfer M, Nadi S, Eghbali A, Tip F. An empirical evaluation of using large language models for automated unit test generation. IEEE Trans. on Software Engineering, 2024, 50(1): 85–105. [doi: 10.1109/TSE.2023.3334955]

[68] Liu Z, Chen CY, Wang JJ, Chen MZ, Wu BY, Tian ZL, Huang YK, Hu J, Wang Q. Testing the limits: Unusual text inputs generation for mobile APP crash detection with large language model. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 137. [doi: 10.1145/3597503.3639118]

[69] Mahbub P, Shuvo O, Rahman MM. Explaining software bugs leveraging code structures in neural machine translation. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 640–652. [doi: 10.1109/ICSE48619.2023.00063]

[70] Shou CF, Liu J, Lu DD, Sen K. LLM4Fuzz: Guided fuzzing of smart contracts with large language models. arXiv:2401.11108, 2024.

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

[72] Teplyuk PA, Yakunin AG, Sharlaev EV. Study of security flaws in the Linux kernel by fuzzing. In: Proc. of the 2020 Int'l Multi-conf. on Industrial Engineering and Modern Technologies. Vladivostok: IEEE, 2020. 1–5. [doi: 10.1109/FarEastCon50210.2020.9271516]

[73] Hao Y, Zhang H, Li GR, Du XY, Qian ZY, Sani AA. Demystifying the dependency challenge in kernel fuzzing. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 659–671. [doi: 10.1145/3510003.3510126]

[74] Yang CY, Zhao ZJ, Zhang LM. KernelGPT: Enhanced kernel fuzzing via large language models. arXiv:2401.00563, 2024.

[75] Hu J, Zhang Q, Yin H. Augmenting greybox fuzzing with generative AI. arXiv:2306.06782, 2023.

[76] Li ZJ, Wang CZ, Liu ZB, Wang HX, Chen D, Wang S, Gao CY. CCTEST: Testing and repairing code completion systems. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 1238–1250. [doi: 10.1109/ICSE48619.2023.00110]

[77] Dakhama A, Even-Mendoza K, Langdon WB, Menendez H, Petke J. SearchGEM5: Towards reliable GEM5 with search based software testing and large language models. In: Proc. of the 15th Int'l Symp. on Search Based Software Engineering. San Francisco: Springer, 2024. 160–166. [doi: 10.1007/978-3-031-48796-5_14]

[78] Lemieux C, Inala JP, Lahiri SK, Sen S. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 919–931. [doi: 10.1109/ICSE48619.2023.00085]

[79] Fakhoury S, Naik A, Sakkas G, Chakraborty S, Musuvathi M, Lahiri S. Exploring the effectiveness of LLM based test-driven interactive code generation: User study and empirical evaluation. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering: Companion Proc. Lisbon: ACM, 2024. 390–391. [doi: 10.1145/3639478.3643525]

[80] Xia CS, Wei YX, Zhang LM. Automated program repair in the era of large pre-trained language models. In: Proc. of the 45th

IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 1482–1494. [doi: 10.1109/ICSE48619.2023.00129]

[81] Engstrom L, Ilyas A, Santurkar S, Tsipras D, Janoos F, Rudolph L, Madry A. Implementation matters in deep RL: A case study on PPO and TRPO. In: Proc. of the 8th Int'l Conf. on Learning Representations. Addis Ababa: OpenReview.net, 20220.

[82] Mastropaolo A, Scalabrino S, Cooper N, Palacio DN, Poshyvanyk D, Oliveto R, Bavota G. Studying the usage of text-to-text transfer Transformer to support code-related tasks. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering. Madrid: IEEE, 2021. 336–347. [doi: 10.1109/ICSE43902.2021.00041]

[83] Tufano M, Drain D, Svyatkovskiy A, Sundaresan N. Generating accurate assert statements for unit test cases using pretrained Transformers. In: Proc. of the 3rd ACM/IEEE Int'l Conf. on Automation of Software Test. Pittsburgh: ACM, 2022. 54–64. [doi: 10.1145/3524481.3527220]

[84] Paul R, Hossain M, Siddiq ML, Hasan M, Iqbal A, Santos JCS. Enhancing automated program repair through fine-tuning and prompt engineering. arXiv:2304.07840, 2023.

[85] Lewis M, Liu YH, Goyal N, Ghazvininejad M, Mohamed A, Levy O, Stoyanov V, Zettlemoyer L. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proc. of the 58th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, 2020. 7871–7880. [doi: 10.18653/v1/2020.acl-main.703]

[86] Zhang T, Irsan IC, Thung F, Han D, Lo D, Jiang LX. ITiger: An automatic issue title generation tool. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Singapore: ACM, 2022. 1637–1641. [doi: 10.1145/3540250.3558934]

[87] Paria S, Dasgupta A, Bhunia S. DIVAS: An LLM-based end-to-end framework for SoC security analysis and policy-based protection. arXiv:2308.06932, 2023.

[88] Liu ZH, Liao Q, Gu WC, Gao CY. Software vulnerability detection with GPT and in-context learning. In: Proc. of the 8th Int'l Conf. on Data Science in Cyberspace. Hefei: IEEE, 2023. 229–236. [doi: 10.1109/DSC59305.2023.00041]

[89] Kang S, Chen B, Yoo S, Lou JC. Explainable automated debugging via large language model-driven scientific debugging. Empirical Software Engineering, 2025, 30(2): 45. [doi: 10.1007/s10664-024-10594-x]

[90] Liu PZ, Sun CN, Zheng YW, Feng X, Qin C, Wang YC, Xu ZY, Li Z, Di P, Jiang Y, Sun LM. Harnessing the power of LLM to support binary taint analysis. arXiv:2310.08275, 2025.

[91] Kang S, An GB, Yoo S. A quantitative and qualitative evaluation of LLM-based explainable fault localization. Proc. of the ACM on Software Engineering, 2024, 1(FSE): 1424–1446. [doi: 10.1145/3660771]

[92] Bui N, Wang Y, Hoi SCH. Detect-localize-repair: A unified framework for learning to debug with CodeT5. In: Proc. of the 2022 Findings of the Association for Computational Linguistics. Abu Dhabi: Association for Computational Linguistics, 2022. 812–823. [doi: 10.18653/v1/2022.findings-emnlp.57]

[93] Mohajer MM, Aleithan R, Harzevili NS, Wei MS, Belle AB, Pham HV, Wang S. SkipAnalyzer: An embodied agent for code analysis with large language models. arXiv:2310.18532v1, 2023.

[94] Liu KB, Liu YY, Chen ZP, Zhang JM, Han YD, Ma Y, Li G, Huang G. LLM-powered test case generation for detecting tricky bugs. arXiv:2404.10304, 2024.

[95] Li TO, Zong WX, Wang YB, Tian HY, Wang Y, Cheung SC. Finding failure-inducing test cases with ChatGPT. arXiv:2304.11686v1, 2023.

[96] Qiu F, Ji P, Hua BJ, Wang Y. CHEMFUZZ: Large language models-assisted fuzzing for quantum chemistry software bug detection. In: Proc. of the 23rd IEEE Int'l Conf. on Software Quality, Reliability, and Security Companion. Chiang Mai: IEEE, 2023. 103–112. [doi: 10.1109/QRS-C60940.2023.00104]

[97] Zhang YW, Li G, Jin Z, Xing Y. Neural program repair with program dependence analysis and effective filter mechanism. arXiv:2305.09315, 2023.

[98] Mashhadi E, Hemmati H. Applying CodeBERT for automated program repair of Java simple bugs. In: Proc. of the 18th IEEE/ACM Int'l Conf. on Mining Software Repositories. Madrid: IEEE, 2021. 505–509. [doi: 10.1109/MSR52588.2021.00063]

[99] Jin M, Shahriar S, Tufano M, Shi X, Lu S, Sundaresan N, Svyatkovskiy A. InferFix: End-to-end program repair with LLMs. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 1646–1656. [doi: 10.1145/3611643.3613892]

[100] Hao SC, Shi XJ, Liu HW, Shu YJ. Enhancing code language models for program repair by curricular fine-tuning framework. In: Proc. of the 2023 IEEE Int'l Conf. on Software Maintenance and Evolution. Bogotá: IEEE, 2023. 136–146. [doi: 10.1109/ICSME58846.2023.00024]

[101] Pearce H, Tan B, Ahmad B, Karri R, Dolan-Gavitt B. Examining zero-shot vulnerability repair with large language models. In: Proc. of the 2023 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2023. 2339–2356. [doi: 10.1109/SP46215.2023.10179324]

[102] Fakhoury S, Chakraborty S, Musuvathi M, Lahiri SK. NL2Fix: Generating functionally correct code edits from bug descriptions. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering: Companion Proc. Lisbon: ACM, 2024. pp. 410–411. [doi: 10.1145/3639478.3643526]

[103] Chen M, Tworek J, Jun H, *et al*. Evaluating large language models trained on code. arXiv:2107.03374, 2021.

[104] Huang K, Meng XX, Zhang J, Liu Y, Wang WJ, Li SH, Zhang YQ. An empirical study on fine-tuning large language models of code for automated program repair. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Automated Software Engineering. Luxembourg: IEEE, 2023. 1162–1174. [doi: 10.1109/ASE56229.2023.00181]

[105] Zhang JL, Cambronero JP, Gulwani S, Le V, Piskac R, Soares G, Verbruggen G. PyDex: Repairing bugs in introductory Python assignments using LLMs. Proc. of the ACM on Programming Languages, 2024, 8(OOPSLA1): 133. [doi: 10.1145/3649850]

[106] He YF, Wang JC, Rong YY, Chen H. Exploring fuzzing as data augmentation for neural test generation. arXiv:2406.08665v1, 2024.

[107] OpenAI. GPT-3.5-Turbo. 2024. https://freeopenaisora.com/deep-dive-gpt-3-5-turbo-versions/

[108] OpenAI. GPT-4. 2024. https://openai.com/index/gpt-4-research/

[109] OpenAI. Codex. 2021. https://openai.com/index/openai-codex/

[110] Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, Zhou YQ, Li W, Liu PJ. Exploring the limits of transfer learning with a unified text-to-text Transformer. The Journal of Machine Learning Research, 2020, 21(1): 140.

[111] Lin D, Koppel J, Chen A, Solar-Lezama A. QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge. In: Proc. of the Companion of the 2017 ACM SIGPLAN Int'l Conf. on Systems, Programming, Languages, and Applications: Software for Humanity. Vancouver: ACM, 2017. 55–56. [doi: 10.1145/3135932.3135941]

[112] Just R, Jalali D, Ernst MD. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proc. of the 2014 Int'l Symp. on Software Testing and Analysis. San Jose: ACM, 2014. 437–440. [doi: 10.1145/2610384.2628055]

[113] Austin J, Odena A, Nye M, Bosma M, Michalewski H, Dohan D, Jiang E, Cai C, Terry M, Le Q, Sutton C. Program synthesis with large language models. arXiv:2108.07732, 2021.

[114] Brown TB, Mann B, Ryder N, *et al*. Language models are few-shot learners. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 159. [doi: 10.5555/3495724.3495883]

[115] Aiyappa R, An JS, Kwak H, Ahn YY. Can we trust the evaluation on ChatGPT? arXiv:2303.12767, 2024.

[116] Jiang N, Liu K, Lutellier T, Tan L. Impact of code language models on automated program repair. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 1430–1442. [doi: 10.1109/ICSE48619.2023.00125]

[117] Jeong B, Jang J, Yi H, Moon J, Kim J, Jeon I, Kim T, Shim WC, Hwang YH. UTopia: Automatic generation of fuzz driver using unit tests. In: Proc. of the 2023 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2023. 2676–2692. [doi: 10.1109/SP46215.2023.10179394]

[118] Yao SY, Yu D, Zhao J, Shafran I, Griffiths TL, Cao Y, Narasimhan K. Tree of thoughts: Deliberate problem solving with large language models. In: Proc. of the 37th Int'l Conf. on Neural Information Processing Systems. New Orleans: Curran Associates Inc., 2024, 517. [doi: 10.5555/3666122.3666639]

[119] Diao SZ, Wang PC, Lin Y, Pan R, Liu X, Zhang T. Active prompting with chain-of-thought for large language models. In: Proc. of the 62nd Annual Meeting of the Association for Computational Linguistics. Bangkok: Association for Computational Linguistics, 2024. 1330–1350. [doi: 10.18653/v1/2024.acl-long.73]

[120] Yao SY, Zhao J, Yu D, Du N, Shafran I, Narasimhan KR, Cao Y. ReAct: Synergizing reasoning and acting in language models. In: Proc. of the 11th Int'l Conf. on Learning Representations. Kigali: OpenReview.net, 2023.

[121] Liu ZM, Yu XT, Fang Y, Zhang XM. GraphPrompt: Unifying pre-training and downstream tasks for graph neural networks. In: Proc. of the 2023 ACM Web Conf. Austin: ACM, 2023. 417–428. [doi: 10.1145/3543507.3583386]

附中文参考文献:

[46] 邓枭, 叶蔚, 谢睿, 张世琨. 基于深度学习的源代码缺陷检测研究综述. 软件学报, 2023, 34(2): 625–654. http://www.jos.org.cn/1000-9825/6696.htm [doi: 10.13328/j.cnki.jos.006696]

李岩(2000－), 男, 硕士生, 主要研究领域为模糊测试.

张翼(1998－), 女, 硕士生, 主要研究领域为自动驾驶, 车载网络测试.

杨文章(1994－), 男, 博士生, CCF 学生会员, 主要研究领域为软件工程, 程序设计语言.

薛吟兴(1982－), 男, 博士, 特任研究员, 博士生导师, CCF 专业会员, 主要研究领域为软件安全, 人工智能安全, 网络空间安全.