https://crad.ict.ac.cn
62(8): 2040−2057, 2025

基于大语言模型的自动代码修复综述

许鹏宇[1] 况博裕[1] 苏 锰[2] 付安民[1,2]
[1](南京理工大学网络空间安全学院 南京 210094)
[2](南京理工大学计算机科学与工程学院 南京 210094)
(xupengyu@njust.edu.cn)

Survey of Large-Language-Model-Based Automated Program Repair

Xu Pengyu[1], Kuang Boyu[1], Su Mang[2], and Fu Anmin[1,2]
[1](*School of Cyber Science and Engineering*, *Nanjing University of Science and Technology*, *Nanjing* 210094)
[2](*School of Computer Science and Engineering*, *Nanjing University of Science and Technology*, *Nanjing* 210094)

Abstract Software systems play an indispensable role across various industries, handling large-scale and high-density data. However, the numerous defects within these systems have troubled developers for a long time, constantly threatening the security of data elements. Automated program repair (APR) technology aims to assist developers in automatically fixing defects in code during software development process, thereby saving costs in software system development and maintenance, enhancing the confidentiality, availability, and integrity of data elements within software systems. With the development of large language model (LLM) technology, many powerful code large language models have emerged. These models have demonstrated strong repair capabilities in the APR field, while also addressing shortcomings of traditional approaches in code comprehension and patch generation capabilities, further elevating the level of program repair tools. We thoroughly survey high-quality papers related to APR in recent years, summarizing the latest developments in the field. We then systematically categorize two types of LLM-based APR techniques: cloze style and neural machine translation style. We also conduct an in-depth comparison from various perspectives such as model usage, model size, types of defects repaired, programming languages involved, and the pros and cons of repair approaches. Additionally, we discuss the widely adopted APR datasets and metrics, and outline existing empirical studies. Finally, we summarize current challenges in the APR field along with future research directions.

Key words automated program repair (APR); large language model (LLM); code large language model; software defect; patch generation

摘 要 软件系统在各行各业中发挥着不可忽视的作用,承载着大规模、高密度的数据,但软件系统中存在的种种缺陷一直以来困扰着系统的开发者,时刻威胁着系统数据要素的安全.自动代码修复(automated program repair, APR)技术旨在帮助开发者在软件系统的开发过程中自动地修复代码中存在的缺陷,节约软件系统开发和维护成本,提高软件系统中数据要素的保密性、可用性和完整性.随着大语言模型(large language model, LLM)技术的发展,涌现出许多能力强大的代码大语言模型,并且代码

收稿日期: 2024-06-05; **修回日期**: 2024-10-22
基金项目: 国家自然科学基金项目(62372236); 江苏省青蓝工程项目; 江苏省卓越博士后计划项目; 国家资助博士后研究人员计划项目(GZB20240982)
This work was supported by the National Natural Science Foundation of China (62372236), the Qing Lan Project of Jiangsu Province, the Jiangsu Funding Program for Excellent Postdoctoral Talent, and the Postdoctoral Fellowship Program of CPSF (GZB20240982).
通信作者: 付安民(fuam@njust.edu.cn)

LLM 在 APR 领域的应用中表现出了强大的修复能力，弥补了传统方案对于代码理解能力、补丁生成能力方面的不足，进一步提高了代码修复工具的水平. 全面调研分析了近年 APR 相关的高水平论文，总结了 APR 领域的最新发展，系统归纳了完形填空模式和神经机器翻译模式 2 类基于 LLM 的 APR 技术，并从模型类型、模型规模、修复的缺陷类型、修复的编程语言和修复方案优缺点等角度进行全方位的对比与研讨. 同时，对 APR 数据集和评价 APR 修复能力的指标进行了梳理和分析，并且对现有的实证研究展开深入探讨. 最后，分析了当前 APR 领域存在的挑战及未来的研究方向.

关键词 自动代码修复；大语言模型；代码大语言模型；软件缺陷；补丁生成

中图法分类号 TP311

DOI: 10.7544/issn1000-1239.202440467 **CSTR:** 32373.14.issn1000-1239.202440467

软件系统已经成为人类社会运行的基石，贯穿于人们生活的方方面面，在企业、政府层面发挥着不可或缺的作用，并承载着各行各业的重要数据，影响着数据要素的采集、生产、存储、传输、使用、流通、删除全生命周期的安全. 然而，软件系统中存在一个无法避免的难题，即代码缺陷. 不仅是人类开发者所编写的代码，随着人工智能技术的发展，由计算机所自动生成的代码中也同样存在着各种缺陷[1]，包括一些编程错误、软件 bug 和安全漏洞[2]. 这些代码缺陷会显著影响软件系统的表现，增加软件系统开发和维护成本，危害软件系统的功能，最终危害软件系统所有用户的数据安全[3]. 为了应对软件缺陷的存在，传统方式通过人工编写补丁来修复缺陷，但这种方式不仅非常依赖于人工的专家经验，而且效率低下、成本高昂. 于是，研究者们提出了自动代码修复（automated program repair, APR）技术[4-7]，通过计算机全自动化的方式为代码缺陷生成相应的补丁以修复这些缺陷.

APR 技术的核心是生成高质量的能够满足代码缺陷修复需求的补丁. 根据补丁生成技术的迭代，APR 技术的发展可以划分为 3 个阶段[5]：传统的 APR 技术、基于深度学习的 APR 技术和基于大语言模型（large language model, LLM）的 APR 技术. 传统的 APR 技术依靠遗传编程、进化计算等算法实现，无需依赖外界高质量数据集即可完成代码修复，但补丁种类局限且代码修复效率低. 基于深度学习的 APR 技术则借助循环神经网络等深度神经网络模型实现，能够借助海量开源代码自动化学习代码修复的规律，但难以理解逻辑复杂的缺陷代码，补丁多样性较低. 虽然文献 [4−8] 系统性地总结并分析当前传统的和基于深度学习的 APR 工作，但随着大模型技术的迅速发展，大量的工作将大模型技术应用于 APR 领域，以提升代码修复的效果. 这些研究发现大模型相比普通规模神经网络具备更强的代码理解能力和代码生成能力，能够修复难度更大的缺陷，生成更高效且丰富的补丁，例如 VulRepair[9] 借助 T5 大模型的自注意力机制和分词机制将安全漏洞的修复能力从 23% 提升至 44%. 这类基于大模型的 APR 的工作在目前的综述文献中基本没有体现，也缺乏系统深入的归纳分析.

在此背景下，为系统梳理目前大语言模型技术在 APR 领域的应用，深入探索 LLM 给 APR 带来的技术优势和未来的发展方向，本文收集了 2019 年 1 月至 2024 年 5 月涉及 APR 工作评级为 CCF-A 的国内和国际推荐会议和期刊论文. 从图 1 可以看出，在 2021 年，也即 LLM 刚起步的时候，已经有学者敏锐地发现利用 LLM 进行代码修复任务的潜力. 从 2022 年开始，基于 LLM 的研究数量超过了传统的 APR 和基于深度学习的 APR 论文数量，且呈现出逐年上升趋势. 可见 LLM 技术的应用已经成为未来 APR 研究的主流发展方向和重要推动力.

综上，本篇综述旨在系统地对目前 APR 领域 LLM 技术的应用进行梳理和总结. 主要贡献包括 3 个方面：

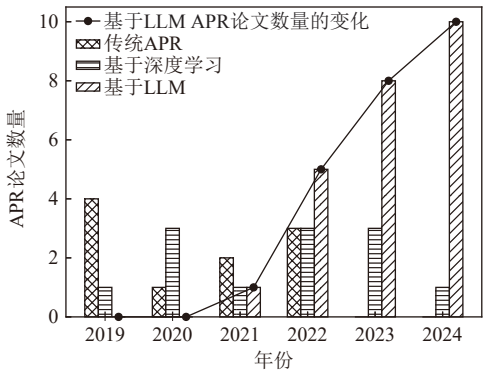


Fig. 1 Number and distribution of APR papers from 2019 to 2024

图 1 2019—2024 年 APR 论文的数量和分布

1)给出了APR技术的工作流程,结合目前基于大模型的APR研究的成果,从补丁生成算法的角度对APR技术进行了分类,总结了不同分类模式下的缺陷修复特点.

2)重点从“完形填空”和“神经机器翻译”两大自然语言处理领域的典型任务深入分析了现有APR研究进展,并对面向LLM的APR研究进行了对比分析与探讨.

3)结合LLM和APR技术发展特点,分析了现有基于LLM的APR研究工作存在的不足,并重点从提升APR的修复效率、提高APR的补丁质量、强化APR的泛化能力和构建全流程的缺陷修复共4个方面探讨了基于LLM的APR研究面临的挑战以及未来研究方向的展望.

1 LLM与APR

本节从LLM和APR 2个方面阐述其当前的发展情况,将首先介绍LLM在编程语言领域的表现和在2种范式下APR领域的应用,接着梳理软件缺陷的3种类别与APR的工作流程,最后将现有的3类APR方案进行深入对比和分析.

1.1 LLM

随着人工智能技术[10]的发展,基于大规模数据集训练出来的具有庞大参数量的人工智能大模型得到广泛应用和关注,并且在自然语言处理、计算机视觉、语音识别等许多科研领域展现出了前所未有的卓越能力[11].

其中,在编程语言领域,涌现出多种代码LLM[12]以实现代码生成、代码修复等编程语言领域的任务. 2020年,Microsoft基于GPT-2[13]LLM推出了第1款代码LLMGPT-C[14],通过使用编程语言源代码数据集进行无监督学习,使得LLM具有了理解、生成编程语言的能力.其后,学者基于其他LLM,例如T5[15],BERT[16],推出了更多的代码LLM,如2020年的CodeBERT[17]、2024年的CodeT5+[18],在解决编程领域的APR、代码生成、代码翻译等问题上都取得了显著的进展.

不过由于如今LLM的规模不断增大,其训练成本也随之增大,普通的用户难以承担如此大规模的模型训练任务.因此,目前的LLM提出了预训练模型的形式,由生产厂商将LLM先在指定的数据集上提前进行训练,使其事先具备一些任务的处理能力后再提供给用户使用.这种预训练模式削减了模型用户在重新下载数据集及重新训练LLM上所需的时间和金钱成本,降低了LLM的使用门槛.然而,虽然在经过预训练之后,LLM获得了较为强大的泛化能力,但是对特定任务的处理能力还不足.于是,为了进一步将模型适应更为具体的不同下游任务,需要对预训练LLM开展进一步的适应性工作,具体可以分为预训练-微调(pretrain-finetune)和提示工程(prompt engineering)两种范式.

1)预训练-微调.微调指将预训练得到的模型继续进行相关任务的训练,更新预训练得到的神经网络参数,从而增强模型针对特定任务的处理能力.

2)提示工程.提示工程指将待解决任务的描述作为输入提供给预训练模型,模型在理解输入任务的基础上激活自身现有的能力解决任务.

在APR领域,学者们借助预训练-微调和提示工程2种范式使已有的代码大语言模型具备补丁生成能力,以完成缺陷修复任务.例如Wu等人[19]对预训练的代码大语言模型进行了Java安全漏洞修复的微调,使得原本拥有较为泛化代码能力的大模型熟悉Java安全漏洞修复的任务.而AlphaRepair[20]借助提示工程范式将缺陷代码、缺陷位置等信息以代码提示的形式提供给预训练的代码大语言模型,LLM根据接收到的提示借助已有知识给出修复代码.

总而言之,LLM在多个领域有着可观的发展前景,特别是在APR领域,学者们借助代码大语言模型优秀的代码理解和生成能力,显著提升了生成补丁的效率和多样性,使得APR技术能够成功修复更加复杂的缺陷.

1.2 APR

APR技术能够自动地修复软件系统中的多种缺陷,当前软件缺陷可以具体分为编程错误(programming error)、软件bug(software bug)和安全漏洞(security vulnerability)三类[2].编程错误指开发者在编码过程中写出的错误代码,例如对某一变量执行了变量类型所支持范围之外的操作;软件bug指会导致程序功能性异常的软件缺陷,例如“if”语句中忽略了某种条件的考虑,导致本应无法通过if判断的输入通过了判断;安全漏洞指能够被攻击者利用的危害系统信息安全的缺陷,例如程序向栈中某个变量写入的字节数超过了变量本身所申请的字节数,导致与其相邻栈中变量的值被改变,允许攻击者控制程序执行的流程.

图2展示了APR的工作流程,具体而言,APR工具在接收到软件缺陷的相关信息(如包含缺陷的代码、缺陷的具体位置及类型等)后,借助对代码和缺



Fig. 2 General process of APR

图 2 APR 总体流程

陷的理解由补丁生成技术输出缺陷代码的 1 个甚至多个补丁. 已有的 APR 方案依据补丁生成技术的不同可以分为传统的 APR 方案、基于深度学习的 APR 方案和基于 LLM 的 APR 方案 3 类[5]. 本节将主要介绍传统的 APR 方案和基于深度学习的 APR 方案, 对基于 LLM 的 APR 方案将进行不同修复模式的讨论, 并将在第 2~3 节展开详细的分析.

1.2.1 传统的 APR 方案

传统的 APR 方案根据方案设计的不同可以继续被划分成基于搜索、基于模板和基于约束 3 种方案[5].

1)基于搜索(search-based). 基于搜索的 APR 方案首先借助启发式算法, 如遗传编程, 生成一些候选的补丁组成搜索空间. 然后通过在搜索空间中利用代码搜索方法, 如寻找相似度高的代码, 以生成修复代码[21-24]. 例如 TransplantFix[21] 使用了一种能感知继承关系的代码搜索方法, 其在代码搜索的过程中, 考虑了方法的内部特征(方法名和方法参数的类型)和外部特征(继承关系), 提出了一种考虑类和对象的继承关系来更准确有效地在搜索空间中搜索代码的方法. TransplantFix 借此搜索方法寻找具有相似功能的候选代码, 以实现缺陷代码的修复.

2)基于模板(template-based). 在基于模板的 APR 方案中, 补丁是借助修复模板生成的. 修复模板记录了缺陷修复的规律, 能够指导 APR 工具按照模板的结构将缺陷代码转换成修复代码. PyTER[25], ATR[26] 使用预先编好的修复模板生成候选的补丁. iFixR[27] 基于 kPAR[28] 收集了一套修复模板, TBar[29] 审计了前人的模板并总结出了一套修复模板. PraPR[30] 则是根据预先指定的转换规则设计了多个变异器给输入的程序做变异, 从而生成修复代码.

3)基于约束(constraint-based). 基于约束的 APR 方案将代码修复视为约束求解问题, 其首先从缺陷代码的测试样例中提取出代码应遵循的约束条件, 接着将约束应用于搜索空间的代码从而删去不满足测试样例要求的代码, 删除过程结束后即获得最终的修复代码. SOSRepair[31] 对此流程进行了进一步的优化, 当候选的补丁违背了测试样例的约束时, 其根据失败的信息总结出新的约束条件, 不断向约束条件中进行补充, 从而能更高效地获得修复代码.

1.2.2 基于深度学习的 APR 方案

受益于深度学习的发展, 学者们开始尝试将深度学习模型结合进 APR 领域, 例如卷积神经网络、循环神经网络、长短期记忆网络、生成对抗网络[32] 等. 受传统方案的启发, 基于深度学习的方案吸收了传统方案中的经验, 认识到了传统方案中的局限. 通过从大量开源软件系统中源码的自动化学习, 基于深度学习的方案具备了充足的代码知识, 能够生成更加丰富、更加复杂的修复代码. 因此, 基于深度学习的方案进一步提升了代码修复能力, 提升了 APR 领域的自动化程度, 使得 APR 工具能够更方便地适用不同语言、扩展不同缺陷类型, 总体上表现出比传统的方案更优秀的修复能力.

Phoenix[33] 提出了一种策略用于取代过于依赖测试样例来生成补丁的传统方案, 通过借助深度学习对挖掘到的公开补丁进行学习, 让机器总结出修复的策略. Dear[34] 结合了长短期记忆网络、神经网络和基于频谱的故障定位, 形成了对多块多语句代码的修复能力. 使用卷积神经网络模型的 CoCoNuT[35] 将 APR 看作一个将缺陷代码“翻译”成正确代码的过程, 设计出一个感知上下文的神经机器翻译模型. SNRepair[36] 借助自注意力机制缓解了循环神经网络在处理代码时遇到的长距离依赖问题, 更好地捕捉了代码中的结构信息, 提升了代码修复的效率.

1.2.3 基于 LLM 的 APR 方案

LLM 在自然语言处理领域能够处理多种任务, 如完形填空(cloze)、神经机器翻译(neural machine

translation, NMT)、情感分类等. 而随着代码大语言模型的提出, LLM 也逐渐被应用至编程语言的处理中. 在 APR 领域, LLM 相较于传统方案和基于深度学习的方案展现出更强大的理解和生成能力[37].

但由于 APR 技术需要在理解代码和缺陷的基础上推理出修复缺陷的补丁, 部分自然语言处理领域的任务, 如情感分类即无法适用于代码修复. 于是现有基于 LLM 的 APR 方案将自然语言领域的 2 类任务迁移至了编程语言领域以完成缺陷修复的目的, 即完形填空任务和神经机器翻译任务. 因此本文将基于 LLM 的 APR 分为完形填空模式[20]和神经机器翻译模式[38]2 类. 完形填空模式的 APR 将代码修复任务类比成语言类测试中的完形填空题型, 在完整的代码中将缺陷代码挖去, 交给 LLM 预测出补丁重新填回代码块中, 以此实现代码修复的目的. 神经机器翻译模式的 APR 将代码修复任务视为机器翻译任务, 大模型学习待修复代码向修复代码转化的规则, 以实现缺陷修复. 针对上述 2 类基于 LLM 的 APR 技术的详细分析将分别在第 2~3 节展开.

1.2.4 对比分析

表 1 总结了现有 APR 方案的详情, 从补丁质量、支持修复缺陷的难度、开源代码资源利用、代码句法理解能力和代码语义理解能力角度对比了现有的 APR 方案.

具体而言, 传统的 APR 方案为实现 APR 开辟了许多新的道路, 但面临着一些难以解决的问题. 比如基于搜索的 APR 方案面临的一大局限是补丁质量低, 由于补丁生成时使用的变异算法本身存在着随机性, 补丁的质量难以保证, 导致搜索空间庞大. 相比之下, 基于模板的 APR 方案一定程度上缓解了基于搜索的 APR 方案搜索空间过大的问题. 通过总结修复之间的相似性, 提炼出来的修复模板的搜索空间远小于候选代码的搜索空间. 但基于模板的 APR 方案也同样面临着局限, 比如学习能力欠佳, 无法利用开源代码平台如 GitHub 的海量代码. 这导致了其仅能依靠修复模板捕获编程语言的部分句法规则, 无法感知代码语义, 生成的补丁被局限在模板之中. 基于约束的 APR 方案面临着类似的问题, 尽管其通过约束条件缩小了搜索空间, 但其也过于依赖缩小后的搜索空间, 学习能力的不足使其生成的补丁也具有局限性, 无法处理复杂或非常规的补丁.

Table 1 Detailed Information on APR Schemes

表 1 自动代码修复方案详情

方案类型	补丁质量	支持修复缺陷的难度	开源代码资源利用	代码句法理解能力	代码语义理解能力
传统的 APR 方案	低	低	否	低	无
基于深度学习的 APR 方案	中	中	是	中	低
基于 LLM 的 APR 方案	高	高	是	高	高

基于深度学习的 APR 方案借助深度学习技术将大量开源代码利用起来, 但所使用的深度神经网络对于代码理解能力和代码生成能力上仍然存在不足, 部分复杂的语义关联很难被循环神经网络这类深度神经网络完全捕获, 并且编程语言不同于自然语言, 存在着更为严格的句法, 因此在理解代码找出缺陷的基础上还需要保证生成的修复代码依然符合句法约束, 这对此类深度神经网络来说仍然是一个挑战.

基于 LLM 的 APR 方案借助大模型技术将大量开源代码充分地利用起来, 大模型神经网络的深度能够支持模型充分捕获编程语言中复杂的句法规则, 且大模型的自注意力机制进一步提高了模型对于输入内容的理解能力, 使得模型拥有了更强的代码语义感知能力. 相较于传统的 APR 方案和基于深度学习的 APR 方案, 基于 LLM 的 APR 方案能够提供质量更高的补丁和修复难度更高的缺陷.

2 完形填空模式的 APR

完形填空模式的 APR 能够充分利用 LLM 在预训练阶段所学的代码知识和待修复代码中隐含的缺陷知识, 将 LLM 技术更直接、快速地应用于 APR 领域. 因此, 完形填空模式的 APR 成为了 APR 领域重要的研究方向之一. 本节将首先介绍完形填空任务的定义, 接着深入分析在完形填空任务下 APR 的工作模式与优势, 之后详尽梳理和对比近年完形填空模式下的 APR 方案及其优缺点, 最终归纳了完形填空模式的 APR 技术仍然面临的局限和未来改进方向.

2.1 完形填空任务下的 APR 技术概述

“完形填空”本身是语言类测试中的一种题型, 出题人将原先完整的一段文本挖去部分内容, 答题人根据空缺的上下文重新推测出空缺处正确的内容. 类似的, 在 LLM 领域, 完形填空任务指模型根据某处空缺文本的上下文推理空缺内容的任务. 此任务重点考察大模型对上下文文本的理解能力, 并且能够用于解决多种具体问题, 如缺陷修复、代码生成、代码编辑[12].

完形填空任务下的 APR 技术首先根据缺陷定位的结果决定在代码中挖空的大致方位; 接着总结出已

有缺陷修复代码的变更规律，决定挖空的具体位置和数量；最后大模型根据挖空后的待修复代码片段推理出空缺处的修复代码，以此完成代码修复的任务. 完形填空模式的 APR 技术主要具备 2 个优点：

1）完形填空模式的 APR 技术能够在提示工程的范式下无需微调模型直接进行缺陷修复任务. 部分大模型，如 CodeBERT 经过掩码语言建模（mask language modeling, MLM）的预训练，而 MLM 正是受完形填空启发设计的训练任务，要求模型从经过随机挖空的输入中预测出空缺处的原始内容. 因此此类大模型无需微调就已经能直接胜任完形填空模式的任务，能够充分发挥模型在预训练阶段学习到的完形填空技巧，更快速、更精准地理解完形填空模式的代码修复任务.

2）完形填空模式的 APR 技术能够充分利用待修复代码的缺陷上下文信息. 完形填空任务要求模型首先理解空缺内容的临近信息，再推理出空缺的具体内容. 在 APR 领域，缺陷代码的临近信息包含宝贵的缺陷修复信息，直接影响缺陷修复的效果[39]. 因此以此模式进行代码修复能够使得模型更加关注缺陷上下文，提升 APR 工具对缺陷代码功能和缺陷成因的理解，从而提升缺陷修复的效果.

2.2 完形填空模式的 APR 方案现状

在 2022 年，AlphaRepair[20] 提出了以完形填空模式进行自动代码修复的方法. AlphaRepair 借助人工总结归纳出缺陷修复的规律，设计了一套用于挖空的修复模板. 当接收到待修复代码时，AlphaRepair 首先套用修复模板将缺陷代码挖去，接着借助 CodeBERT 大模型在挖空后的缺陷代码中填空，以此修复代码中的缺陷. AlphaRepair 发现在代码修复过程中，bug 代码本身包含一些代码修复需要使用的重要信息（如特定的变量、修改行的类型等）. 因此将 bug 代码行以自然语言注释的形式输入给大模型能帮助模型生成更多有效的补丁，这为后来的工作[40-41] 提供了一定的指导.

在 AlphaRepair 之后，学者们在 4 个方面进一步完善了完形填空模式的 APR 技术：修复模板设计、修复模板选择、模型填空指导、更多大模型.

2.2.1 修复模板设计

修复模板由缺陷修复的代码变更规律总结而来，用于在待修复的代码中挖去缺陷代码. AlphaRepair 所采用的修复模板依靠人工设计而成，种类、个数有限，这意味着 AlphaRepair 仅支持修复当前修复模板覆盖范围之内的缺陷. 于是，Gamma[40] 和 TYPEFIX[42] 在此基础之上分别设计了能够自动归纳修复模板的方案. 在 2023 年，Gamma[40] 不仅采用人工的方式归纳修复模板，还借助了现有的静态分析[43-44] 和机器学习[45-46] 的方案自动生成模板. 在 2024 年，TYPEFIX[42] 提出了修复模板设计的新方案，其设计了一个自动化模板挖掘方法，完全取代了人工编写修复模板的步骤. TYPEFIX 通过从 GitHub 提取出涉及修复 Python 类型错误（type error）的代码，把代码变更抽象和总结成修复模板. 相比于人工编写的修复模板，由此生成的修复模板具有修复较新出现、较复杂缺陷的能力，因为 TYPEFIX 可以自动化地从挖掘的代码中总结出修复规律，使得这些模板具有更新、更丰富的缺陷修复知识.

2.2.2 修复模板选择

AlphaRepair 在借助修复模板进行代码修复的过程中，会套用全部模板用于修复任务，然而随着修复模板数量不断增加，会导致其开销不断增大. 于是，Gamma[40]，TYPEFIX[42] 在此基础之上分别设计了修复模板的选择方案. Gamma，TYPEFIX 都受 TBar[29] 启发，借助了抽象语法树（abstract syntax tree, AST）来实现各自的方案. Gamma 首先生成待修复代码的 AST，接着判断 AST 的每个节点是否被预定义的修复模板所包含，当修复模板中包含待修复代码 AST 中任一节点时，修复模板即被选用. TYPEFIX 不仅为待修复代码生成 AST，对从 GitHub 公开项目提取的代码也进行 AST 生成，且将 GitHub 的 AST 进行聚类，通过把待修复代码的 AST 与经过聚类的代码 AST 进行比较决定选用何种修复模板. 以上 2 种方案能够在大量的修复模板中选择出较为合适的修复模板以供 LLM 使用，提高修复的效率和成功率.

2.2.3 模型填空指导

在选定了修复模板之后，AlphaRepair，Gamma，TYPEFIX 把补丁生成的过程全部交由代码大语言模型完成，但并没有进一步根据待修复代码的语义指导模型进行缺陷修复的推理. Repilot[1] 发现即使代码大语言模型经过代码有关任务的预训练，在预测补丁时所生成的部分 token（模型可以理解和生成的最小意义单位）仍会丢失代码结构或代码语义的理解，导致降低模型生成补丁的质量和修复能力. 因此，在 2023 年，Repilot[1] 为 APR 领域开辟了一个新的途径，其吸收了 IDE 中代码补全的思想，借助 Eclipse JDT Language Server 工具作为补全引擎为搜索空间进行修剪. 通过消除补全引擎所不认可的 token，能够使得生成的补丁更加符合代码的语义. 这类似开发人员

能够借助 IDE 自动补全更轻松地编程, LLM 在进行代码修复任务时有了补全引擎的指导, 不仅能够使得工具更好地判断出大模型生成 token 的可行性, 甚至可以主动地为当前程序进行补全. 经过 Repilot 实验表明, 结合补全引擎进行搜索空间删减和主动代码补全, 能够进一步提高所生成的补丁能被成功编译的个数和能被成功修复缺陷的个数, 提高模型的代码修复能力.

2.2.4 更多大模型

除 CodeBERT 外, 现实中存在其他经过不同任务预训练、拥有不同参数规模的大模型. 这些大模型具有各自擅长的能力, 但代码修复效果尚待探索. 于是, Gamma 和 Repilot 对比了更多大模型在完形填空模式下的代码修复效果. Gamma 不仅类似 AlphaRepair 利用 CodeBERT 进行实验, 也在完形填空模式中使用 UniXCoder[47] 和 ChatGPT 进行代码修复的实验. 由于 UniXCoder 经过了 MLM 任务的预训练, 其也可以直接被用于完形填空模式的代码修复任务. 并且从修复能力上看, CodeBERT 和 UniXCoder 差距也较小. 但 Gamma 发现, 在生成正确补丁的耗时上, CodeBERT 需要花费远远更多的时间. 这是因为正确补丁的长度是未知的, 对同一个缺陷的同一个修复模板需要提供给 CodeBERT 不同推理长度的设置多次运行. 而对于 UniXCoder 来说, 其可以循环预测补丁的下一个 token, 直至遇到文件结尾(end of file, EOF)token 终止, 因此只需提供给 UniXCoder 待修复代码的位置即可完成补丁的推理. Gamma 也借助了自然语言的对话与 ChatGPT 进行交互, 通过输入带有完形填空任务提示的指令使得 ChatGPT 理解接下来要进行的代码修复任务. 但从结果上来看, ChatGPT 的表现要弱于 CodeBERT 和 UniXCoder, 原因可能在于 CodeBERT 和 UniXCoder 经过 MLM 目标的预训练, 而 ChatGPT 由于闭源的缘故无法确定其预训练的过程. Repilot 将规模较大的 InCoder 大模型与规模较小的 CodeT5[48] 进行了对比, 发现使用更大的模型能够提升修复能力.

2.3 小 结

表 2 总结了现有完形填空模式的 APR 方案, 从模型选择、模型规模、缺陷类型、编程语言等角度对比了现有方案. 总体而言, 目前的完形填空模式的 APR 技术主要从如何设计修复模板、如何筛选修复模板、如何指导大模型“填空”、探索更多大模型的修复效果 4 个方面展开, 但现有的工作较少, 这是因为在完形填空模式的 APR 下大模型仅能借助已有知识生成补丁, 无法从开源代码中学习更多代码缺陷修复的深层规律. 现有的完形填空模式的 APR 也仍面临着一些挑战, 如仅开展了提示工程范式下的工作, 方案无法泛化至多种缺陷类型、不同编程语言. 未来可以在完形填空模式下尝试 MLM 任务微调[49], 学习更多缺陷修复的规律; 或者研究不同缺陷类型、不同编程语言之间缺陷的 AST 是否有相关性, 聚类算法是否继续有效, 以提升修复方案的泛化能力.

Table 2 Cloze-Style APR Schemes

表 2 完形填空模式的自动代码修复方案

方案	大语言模型	模型规模	缺陷类型	语言	特点	不足
AlphaRepair[20]	CodeBERT	1.25 亿	bug	Java Python	首个完形填空模式的代码修复方案.	修复模板仅由人为手动归纳, 成本高、扩展性低; 修复时套用所有的修复模板, 成本高.
Gamma[40]	CodeBERT UniXCoder ChatGPT	1.25 亿 1.25 亿	bug	Java	修复模板由人为手动和机器自动归纳; 修复时根据 AST 挑选模板套用.	修复代码长度固定, 灵活性较低; 基于提示工程的工作较为基础.
Repilot[1]	InCoder CodeT5	67 亿 2.2 亿	bug	Java	借助代码补全引擎指导大语言模型更高效生成补丁.	代码补全引擎会对搜索空间进行删减, 补全引擎的可靠性无法被完全保证.
TYPEFIX[42]	CodeT5	2.2 亿	error	Python	自动挖掘缺陷修复代码, 并基于挖掘代码自动总结修复模板.	有待进一步研究能否推广至不同缺陷类型、不同编程语言的缺陷修复.

3 神经机器翻译模式的 APR

神经机器翻译模式的 APR 能够充分利用海量开源项目中的缺陷修复知识, 学习到多类型缺陷和多编程语言缺陷的修复技巧, 直接提升了 LLM 技术的补丁生成能力, 因此神经机器翻译模式的 APR 成为了 APR 领域的一个重要研究方向. 本节将首先介绍“神经机器翻译”任务的定义, 接着深入分析在神经机器翻译任务下 APR 的工作模式与优势, 之后详尽梳理和对比近年神经机器翻译模式下 APR 方案及其优缺点, 最终归纳了神经机器翻译模式的 APR 技术仍然面临的局限、原因和未来改进方向.

3.1 神经机器翻译任务下的 APR 技术概述

神经机器翻译指借助神经网络实现某种语言向另一种语言翻译的过程. 类似的, 在 LLM 领域, NMT

任务旨在使模型学习输入文本和输出文本的转化关系，且不局限于自然语言之间的翻译，例如在编程语言中，NMT 可以将 Python 代码翻译为 Java 代码和将缺陷代码翻译为修复代码. 此任务的重点在于模型学习到输入序列和输出序列之间的映射关系，能够用于解决代码修复、代码翻译、代码总结等具体问题.

目前，学者们也将神经机器翻译技术应用于 APR 领域. 神经机器翻译任务下的 APR 首先借助缺陷修复数据集，如由缺陷代码和修复代码组成的缺陷-修复对（bug-fix pair, BFP）[50] 给 LLM 进行微调，学习缺陷代码向修复代码转化的规律. 经过微调后的大模型即可根据输入的缺陷代码，结合缺陷的类别、位置等额外信息并根据已学到的缺陷修复技巧推理出缺陷代码的补丁.

神经机器翻译模式的 APR 技术的优点在于其能够在预训练-微调范式下充分利用开源项目中的海量代码. 特别是在微调阶段，大模型能够从海量涉及缺陷修复的代码变更中学习到多种缺陷的修复规则，直接提升大模型的补丁生成能力. 相比之下，尽管完形填空模式的 APR 技术[42] 也能利用开源项目中的海量代码，但完形填空模式的 APR 技术仅能学习到缺陷修复的不同位置，而缺陷修复的更深层规律尚未得以学习，如补丁中需要借助缺陷代码的哪些上下文、如何借助上下文内容构造补丁.

表 3 总结了完形填空模式和神经机器翻译模式的优势和缺陷. 对比可知，完形填空模式和神经机器翻译模式在 APR 领域各有所长，因此存在各自适用的修复场景. 1）完形填空模式的 APR 技术对于大模型的适应能力强，在新款大模型发布后无需过多调整即可将新模型应用于代码修复任务，能够更快速地使用版本更新更快、能力更强的大模型. 同时完形填空模式的 APR 技术对数据集的依赖程度较低，由于如今 LLM 和 APR 技术发展迅速，数据集的质量仍在不断地提升. 在现有数据集缺陷种类缺失、缺陷种类分布不均衡、缺陷信息残缺等情况下，完形填空模式的 APR 技术依然能够保持较为稳定的修复水平. 2）神经机器翻译模式的 APR 技术拥有更强的学习能力，能够借助丰富的开源代码资源学习到更深层的缺陷修复技巧. 在拥有的模型、训练集、算力等的支持下，LLM 的缺陷修复能力能够在神经机器翻译模式下得到更显著的提升.

Table 3 Comparison of Different Program Repair Modes

表 3 不同代码修复模式的对比

修复模式	大模型适应能力	数据集依赖程度	学习能力
完形填空	强	低	弱
神经机器翻译	弱	高	强

3.2 神经机器翻译模式的 APR 方案现状

在 2021 年，CURE[38] 首次将大模型技术加入 NMT 模式的 APR 方案. CURE 对 GPT 进行了约 404 万开源方法的训练，学习了丰富的句法知识，具备了生成类似开发者所编写源码的能力. 并且 CURE 将预训练得到的 GPT 模型添加进 CoNuT[35] 的 NMT 模型，利用 CoCoNuT[35] 的 APR 数据集进行进一步微调，使得 CURE 不仅学习到了正确的代码句法，也学习到了充足的代码修复知识. 缓解了当时现有 NMT 方案所生成的补丁不仅包含句法错误，而且包含能通过编译但无法修复缺陷的代码的问题. 此外，被当时 APR 方案所广泛使用的词级标记（word-level tokenization）将代码中每个新出现的单词视为 1 个单独的 token 记进词表，这将导致词表规模变得巨大，对于 NMT 模型来说此种计算成本不可接受. CURE 借助字节对编码（byte pair encoding, BPE）技术，将编程语言中的单词进一步拆分成子词，借助子词的不同排列即可表示出多种单词，从而减小了词表的规模，提高了代码修复的效率.

此后，学者们从 5 个方面进一步完善了 NMT 模式的 APR 技术：更多缺陷类型、更多编程语言、代码语义指导、项目代码感知、丰富缺陷信息.

3.2.1 更多缺陷类型

软件缺陷可以具体分为软件 bug、编程错误和安全漏洞 3 类[2]，现有的基于大模型的 APR 方案对软件 bug[38] 和编程错误[42] 的修复已表现出不错效果，但由于安全漏洞的修复不仅需要填补漏洞，还需要保证修复前后程序功能性的一致，难度相较修复软件 bug 和编程错误更大，目前少有方案涉及. 因此 VulRepair[9] 和 VulMaster[51] 将 NMT 模式的代码修复技术应用于安全漏洞的修复上.

在 2022 年，VulRepair[9] 将 BPE 技术与大型代码语料结合进 T5 大模型，以实现 NMT 模式漏洞代码修复的方案. 其发现使用经过预训练的 LLM 相比先前的安全漏洞代码修复方案能够提升 30%~38% 的漏洞修复能力. CURE 指出在 APR 领域使用 BPE 能够减小词表规模并覆盖更多的正确补丁. VulRepair 在此之上发现，不论大模型架构如何，使用 BPE 都能够提升 9%~14% 的漏洞修复能力，并将 CURE 的这部分工作在安全漏洞修复上进行了验证. 此外，VulRepair 将代码视作无结构的自然语言文本输入给 LLM，而

VulMaster 则认为这难以让模型聚焦编程语言所特有的严格结构, 因此在 VulRepair 之上, 于 2024 年, 在 NMT 模式的漏洞代码修复中将树形结构的代码提供给大模型进行修复[51]. 由于当前的 CodeT5 等大模型无法直接处理 AST 等树形结构的输入, VulMaster 采用深度优先算法将漏洞代码的 AST 转化为树节点的序列提供给大模型, 提升了大模型对于代码复杂结构的理解, 进一步提高了漏洞修复效果.

3.2.2 更多编程语言

大部分 NMT 模式的 APR 技术将代码修复的范围限定在单种编程语言之内. 因为多种编程语言的编程规律和缺陷信息难以在 1 次微调中由 LLM 学习, 而 LLM 在多次微调后会显著遗忘先前学习到的知识. 于是, 为了将代码缺陷修复推广至多种编程语言, CIRCLE[52] 首次在 APR 领域应用终身学习实现了跨语言代码修复. 在 2022 年, CIRCLE 通过利用 T5 模型持续学习跨语言的 bug 修复规则, 借助样本重放技术和基于可塑权重巩固的正则化技术来解决持续学习中的灾难性遗忘问题, 实现了一个能够同时修复多种语言(C, Java, JavaScript, Python)的代码修复方案.

3.2.3 代码语义指导

在 CURE 的微调模式中, 模型会在缺陷代码和数据集中对应的修复代码之间进行学习, 以达到模型预测补丁和数据集中的修复代码完全一致的目的. 但事实上, 对于一个缺陷来说, 正确的修复补丁存在无穷多种写法, 强制大模型生成与数据集中补丁完全一致的修复代码会导致模型过矫正. 在 2022 年, RewardRepair[53] 基于此问题, 将以往使用 BFP 进行训练的过程称为句法训练, 在句法训练之上引入了新的判别模型(discriminative model). 判别模型根据大模型所生成补丁执行结果的好坏放缩模型训练的损失, 能够实现代码语义层面的训练. RewardRepair 是第一个引入补丁判别模型对大模型进行代码语义指导的方案, 以此鼓励大模型生成更多可编译、语义上等价的补丁, 缓解了先前方案使用 BFP 训练的过矫正的问题.

3.2.4 项目代码感知

先前 NMT 模式的 APR 方案在微调过程中考虑到模型过拟合的问题, 普遍将用于训练的代码和实际中待修复的代码进行分离, 即先使模型在其他数据集上学习缺陷修复的通用规则, 再用于修复某一特定数据集. 但 SelfAPR[54] 认为此类修复模式会让模型失去对待修复代码所在项目的完整感知, 从而限制大模型修复的能力. 因此, 在 2022 年, SelfAPR[54] 探索了自监督学习在 APR 领域的应用, 在待修复代码项目的旧版本代码中加入扰动以形成大量与待修复代码项目相关联的样本供模型学习. 通过此种自监督学习, SelfAPR 提高了 NMT 模型对于待修复代码所在整体项目的综合理解, 使得即使部分重要 token 在待修复代码中少有出现, 模型也能够成功生成出此类 token. 而由于 SelfAPR 将待修复代码在经过扰动模型的改变后才加入训练, 这也一定程度上缓解了模型过拟合的问题.

3.2.5 丰富缺陷信息

CURE 在微调阶段采用的数据集为 BFP 的形式, 即由 1 条待修复的缺陷代码和 1 条对应的修复代码组成. 随着 APR 技术的发展, 学者们发现不仅可以将每条缺陷代码提供给 NMT 模型, 而且可将更丰富的缺陷信息提供给模型以进一步提升模型的修复效果.

在 2022 年, SelfAPR[54] 将测试执行的报错信息, 如编译器的报错信息、测试样例执行的报错信息, 提取总结成诊断信息加入 BFP 的训练当中, 使得模型不仅从 BFP 中学习修复代码, 同时能够从诊断信息中提取丰富的关于 bug 类型的描述, 能够让模型更加对症下药. 在 2023 年, RAP-Gen[55] 受到了开发者 debug 时会搜索类似代码修复案例并从相关代码的修复例子中学习修复 bug 的启发, 设计出了一个补丁检索器, 负责搜索与待修复代码有关的 BFP 并提供给大模型生成候选补丁. RAP-Gen 是第 1 个提出借助检索增强生成思想提供给大模型多条修复模板的方案, 以此令模型学习到与待修复代码相关的多条 BFP, 提高了 NMT 模型生成正确补丁的数量. 在 2024 年, VulMaster[51] 在 APR 领域借助融合解码器(fusion-in-decoder, FiD)将更丰富的缺陷信息提供给 NMT 模型. 由于 NMT 模型的编码器在接收大量输入时存在着编码长度限制, VulMaster 借助 FiD 将大量输入组合后提供给大模型, 以此解决了编码器对输入长度的限制问题, 在漏洞代码修复时将更丰富的漏洞知识, 如漏洞类型、漏洞的典型代码、漏洞的详细分析交给大模型进行修复, 提高了漏洞修复的效果.

3.3 小 结

表 4 总结了现有 NMT 模式的 APR 方案, 从不同大模型、不同模型规模、不同缺陷类型、不同编程语言等角度对比了现有方案. 总体而言, 目前 NMT 模式的 APR 技术主要从如何修复更多类别的缺陷、如何修复更多编程语言的缺陷、如何提供给 LLM 更专业的代码语义指导、如何使 NMT 模型完整地感知缺陷项目的代码、如何提供给 LLM 更丰富的缺陷信息

Table 4 NMT-Style APR Schemes

表 4 神经机器翻译模式的自动代码修复方案

方案	大语言模型	模型规模	缺陷类型	语言	特点	不足
CURE[38]	GPT		bug	Java	提出编号有效性检查技术和长度控制技术以提高修复率.	对 GPT 进行预训练的成本高；使用的 NMT 模型为卷积神经网络，能力弱于基于 Transformer 的 NMT 大模型.
VulRepair[9]	CodeT5	2.2 亿	vul	C	在漏洞代码修复中引入 BPE 提高了修复效率和修复效果.	漏洞数据集缺少测试样例，无法对漏洞代码的语义进行判断；将漏洞代码视为文本提供给 LLM，丢失了部分代码结构信息.
VulMaster[51]	CodeT5	2.2 亿	vul	C/C++	利用融合解码器将更丰富的漏洞信息和更高级的代码表示提供给 LLM 进行修复.	漏洞数据集缺少测试样例，无法对漏洞代码的语义进行判断.
CIRCLE[52]	T5	2.2 亿	bug	Java Python JavaScript C	借助终身学习实现了跨语言的代码修复.	不同编程语言之间仍然存在部分特性无法被跨语言学习，会导致 OOV 问题.
RewardRepair[53]	T5	2.2 亿	bug	Java	引入补丁辨别模型实现了 LLM 语义训练.	语义训练采用的数学放缩模型较为简单，分类依据可以继续细化.
SelfAPR[54]	T5	2.2 亿	bug error	Java	将测试执行诊断信息添加进修复流程，提升了修复能力；利用自监督学习提升了项目代码的完整感知能力.	自监督学习设计的代码扰动引擎仅针对 Java 语言部分缺陷设计，有待进一步推广至更多语言.
RAP-Gen[55]	CodeT5	2.2 亿	bug error	Java JavaScript	在代码修复中引入检索增强生成思想提高了代码的语义和词法的感知.	补丁评价指标采用完全一致匹配，无法对补丁的语义进行评估.

共 5 个方面展开. 现有的神经机器翻译模式的 APR 也仍面临着一些挑战, 如较为依赖高质量的 APR 数据集、部分跨语言的缺陷修复规律难以学习. 未来可以尝试在安全漏洞的缺陷修复中借助漏洞证明(proof-of-vulnerability, PoV)从语义上判断补丁是否有效, 以更加精准地提升模型的漏洞修复能力; 或者将跨语言修复中难以学习的特性提前归纳成统一表示以减轻模型学习的压力.

4 数据集与评价指标

数据集直接参与模型的训练和评估, 数据集的质量深刻地影响着大模型的代码修复效果[19]. 评价指标是评价 APR 方案效果好坏的标准, 量化了不同 APR 方案在多维度上效果的差异. 数据集和评价指标是 APR 领域的重要组成部分, 因此本节将分析和归纳现有数据集、评价指标之间的差异, 供未来 APR 学者们对比并选择.

4.1 数据集

完形填空模式的 APR 和 NMT 模式的 APR 都需要数据集的参与. 数据集可以作为基准评价模型的代码修复能力, 也可以作为模型的训练集帮助模型学习代码修复的相关知识. 得益于 APR 社区的大量研究, 当前存在许多优秀的数据集.

表 5 中展示了本篇综述中涉及的基于 LLM 的 APR 方案所使用的数据集.

从编程语言和缺陷类型上来看, 表 5 中大多数都是 Java 语言的软件 bug 数据集, 其中最广为使用的是 Defects4J[56] 数据集. 在 Defects4J v1.2 中收集了 6 个开源 Java 项目共 395 个可复现的 bug, 每个 bug 都包含了完整的项目 bug 代码、项目修复代码和测试样例, 且在大量方案中都被使用[1-2,20,37,40,52-53,55]. 而在 Defects4J v2.0 中, 共收集了来自 17 个开源 Java 项目共 835 个 bug. 作为一个更新、更丰富的数据集, 部分 APR 方案[1-2,20,37,40,53-55] 直面该挑战, 开始使用 Defects4J v2.0 来评估其对于代码修复的综合能力.

从缺陷数量上来看, 数据集的规模各有不同, 从 40 条数据到 2 400 万条数据不等, 这和数据集的构建难度有关. 例如规模较小的 Defects4J 数据集中包含了 bug 代码更为全面的信息, 其包括完整的项目代码、足够的测试样例等, 这在现实中是较难被获取的. 而在规模较大的数据集中, 数据主要仅以 BFP 的形式出现. 大规模数据集依靠缺陷数量上的优势, 尽管牺牲了部分完整项目的信息和测试样例, 仍然能满足代码大语言模型学习代码修复模式的需求. 并且得益于代码开源平台的存在, 大规模 BFP 数据集的构建更为简单. 以表 5 缺陷数量最多的 CoCoNuT 数据集[35] 为例, 其获取 GitHub 平台中开源项目的 commit 信息, 根据一些代码修复的英文关键词(fix, bug, patch)筛选出涉及到代码修复的代码变更, 从而构建出了一个包含大量缺陷、不同语言、不同缺陷类型的数据集.

此外, 数据集还可以从是否包含测试集合、是否被作为训练集和是否被作为基准进一步讨论. 测试集合是一系列能够触发缺陷的测试样例的集合. 在 APR 领域, 带有测试集合的数据集通常被作为基准

Table 5 Detailed Information of Datasets

表 5 数据集详情

数据集	语言	缺陷类型	缺陷数量	测试集合	训练集	基准
VulRepair[9]	C/C++	vul	8 482	无	是	是
VJBench[19]	Java	vul	42	有	否	是
VJBench-trans[19]	Java	vul	150	有	否	是
CPatMiner[34]	Java	bug	44 000	无	是	否
CoCoNuT[35]	C, Java, JavaScript, Python	bug	24×10^6	无	是	否
Prenner 等人[39]	Java, Python	bug	708 215	无	是	是
LMDefects[41]	Java	bug, error	113	有	是	是
HumanEval-Java[50]	Java	bug	173	有	否	是
SelfAPR[54]	Java	bug, error	1 039 873	有	是	否
Defects4J v1.2[56]	Java	bug	395	有	否	是
Defects4J v1.4.0[56]	Java	bug	438	有	否	是
Defects4J v2.0[56]	Java	bug	835	有	是	是
Recoder[57]	Java	bug	103 585	无	是	否
BFP S[58]	Java	bug	58 350	无	是	是
BFP M[58]	Java	bug	65 454	无	是	是
QuixBugs[59]	Java, Python	bug	40	有	否	是
SequenceR[60]	Java	bug	40 289	无	是	是
Tfix[61]	JavaScript	bug	100 000	无	是	是
Big Vul[62]	C/C++	vul	3 754	无	是	是
CVEFixes v1.0.0[63]	C/C++	vul	5 495	无	是	是
ManyBugs[64]	C	bug	185	有	否	是
Vul4J[65]	Java	vul	79	有	否	是
BugAID[66]	JavaScript	bug	105 000	无	否	是
Bugs.Jar[67]	Java	bug	1 158	有	否	是
MegaDiff[68]	Java	bug	663 029	无	是	是
BEARS[69]	Java	bug	251	有	是	否
CodRep[70]	Java	bug	58 069	无	是	否
TSSB-3M[71]	Python	bug	3×10^6	无	是	是
ManySStuBs4J[72]	Java	bug	153 652	无	否	是

注：bug 代表软件 bug，vul 代表安全漏洞，error 代表编程错误.

用来评价生成补丁的正确性. 而带有测试集合的数据集较少作为训练集出现, 原因有 2 点: 1)受数据集规模制约, 由于测试样例的公开程度远低于代码的开源程度, 此类数据集构建的难度大, 现有的数据集规模较小; 2)当数据集同时作为训练集和基准时会导致模型过拟合, 错误地量化模型的代码修复能力. 不带测试集合的数据集则既可以作为训练集, 也可以作为基准. 这是由于 BFP 中包含着代码修复的丰富知识, 大模型可以从中学习代码修复模式, 并且此类数据集规模足够支撑划分训练集和测试集, 通过对比大模型所生成的补丁与现实中修复代码的差异可以判断补丁生成的对错.

近年也有学者在论文中贡献了全新的数据集, 以解决现有数据集在 APR 任务上存在的一些问题. Wu 等人[19] 创建了一个全新的 Java 安全漏洞修复数据集. 为了消除数据集的部分代码已经在大语言模型训练时被学习过, 从而对基准评价能力产生的影响, Wu 等人利用代码变形技术, 将漏洞代码变形成语义等价的代码, 创建了评价漏洞修复能力的 VJBench-trans 数据集. SelfAPR[54] 则基于 Defects4J v2.0 构造了大规模的带有测试集合的数据集, 其使用代码扰动技术对 Defects4J 的项目代码进行扰动, 构造出了超过 100 万条数据的数据集. 其利用扰动生成的一系列代码依然适用于原先 Defects4J 的测试集合, 在没有扩大测试集合的前提下扩大了测试集合所适用的代码数量, 应对了前文提及的带有测试集合数据集数据量小的难题.

4.2 评价指标

在 APR 领域, 评价指标的作用在于量化地评价 APR 方案效果的好坏, 目前主要存在 2 种不同的指标: 基于执行的指标和基于匹配的指标[5], 如表 6 所示.

1)基于执行的指标. 基于执行的指标涉及代码能否编译运行和能否通过测试样例, 这需要代码运行环境和测试集合的参与. 编译通过率体现了 LLM

Table 6 Detailed Information of Evaluation Metrics

表 6 评价指标详情

指标类型	具体指标	指标特征			
		人为介入	运行环境	测试集合	耗时
基于执行[53]	编译通过率	不需要	需要	不需要	中
	测试样例通过率	不需要	需要	需要	中
基于匹配[9]	内容一致性	不需要	不需要	不需要	低
	语义等价性	需要	不需要	不需要	高

对于编程语言句法掌握的好坏. 测试样例通过率体现所生成的补丁能否从功能上修复原有的缺陷并且是否改变原来代码的功能性.

2)基于匹配的指标. 基于匹配的指标涉及到缺陷代码和修复代码之间的相似性. 匹配大模型所生成补丁和开发者所编写补丁的内容是否一致可以判断补丁的正确性. 通过人工判断补丁语义上是否等价判断补丁正确性.

总的来说, 每个评价指标各有侧重. 基于执行的指标更加注重自动化程度, 在运行环境或测试集合的支撑下, 可以自动化评估补丁的准确性, 但由于需要经过编译、运行测试样例, 消耗的时间相对较高. 基于匹配的指标则解决了许多开源代码缺少运行环境、缺少公开测试集合的困难. 其通过完全一致匹配牺牲部分语义上等价的补丁, 以更快的速度完成评判. 也可以引入人工判断补丁的语义来较为准确地判断此类难评判代码.

5 实证研究

APR 领域中, 除了研究新的代码修复方案之外, 也有学者进行了实证研究, 将 APR 流程中不同实验设置对于修复能力的影响进行对比, 分析了实验结果中隐含的逻辑, 同样推进了 APR 领域的发展. 本节将近年的实证研究按照研究内容的不同分为 4 个类别, 并分别总结了实验中的关键发现, 归纳出现有实证研究对未来 APR 工作的指导.

5.1 大模型生成范围

Xia 等人[37] 直接利用 LLM 进行 APR 的实证研究. 其设计了 3 种实验设置来比较大模型的 bug 修复能力: 利用 LLM 生成完整的函数、根据上下文生成多行代码、根据上下文生成单行代码. 通过对比发现, LLM 在后 2 种设置中能表现出优于第 1 种设置的代码修复能力. 原因在于对于第 1 种设置, 模型需要依靠自己的能力成功定位到缺陷生成完整的函数, 再给出合适的补丁, 这对于大模型来说不仅需要强大的代码修复能力, 还需要对缺陷定位任务熟练. 对于后 2 种设置, 缺陷的位置已经事先定位完毕, 因此模型只需根据缺陷代码的上下文生成修复代码. 这一研究结果揭示了现有 LLM 在 APR 领域的出色潜力, 未来的工作可以借助 LLM 更丰富的缺陷相关信息, 如缺陷所在项目的代码, 使 LLM 熟悉缺陷项目的编程风格, 进一步提升 LLM 的代码修复效果, 或探索其他大模型, 如 ChatGPT 以对话形式进行代码修复的效果.

5.2 大模型微调设置

Huang 等人[2] 对比了不同大模型在 NMT 模式下对软件 bug、安全漏洞、编程错误的修复能力. 经过实验发现, 全部 3 种缺陷大语言模型都相较于现有基于深度学习的方案表现更佳, 且对更难修复的代码缺陷也具有潜力. 此外, Huang 等人[2] 发现, 在微调过程中, 不同的设计, 如不同的代码抽象、代码表示和根据不同评价指标得到的大模型的选择会影响修复的效率. 也揭示了微调范式下模型修复能力受限的一些问题, 如算力的限制和修复成分的不足. 这一研究为未来微调 LLM 进行缺陷修复提供了指导, 例如可以进行参数高效微调(parameter efficient fine-tuning, PEFT), 或将高级代码表示, 如树或图形式的代码进行微调, 以面对算力限制和修复成分不足的问题.

5.3 安全漏洞类型的缺陷修复

Wu 等人[19] 进行了 LLM 修复 Java 安全漏洞的实证研究, 对比了前沿的 APR 技术、未经微调的大模型和经过微调的大模型修复 Java 安全漏洞的效果, 发现前沿的 APR 技术和未经微调的大模型对于漏洞代码的修复欠佳, 表现最好的 Codex 仅能修复 20.4% 的漏洞代码. 而经过微调的大模型能够修复更多的安全漏洞, 这表明了 LLM 对于漏洞修复具有一定的潜力. 但经过微调后的模型仍然存在着一些问题, 比如能够修复的漏洞种类较少, 修复能力局限于简单修改的漏洞, 对于代码句法知识学习的能力不足, 所生成候选补丁的编译率也较低. 未来的工作可以构造包含更多漏洞类别、更大漏洞规模的修复数据集, 或在漏洞修复领域探索少样本学习的效果, 提升模型对安全漏洞修复的理解.

5.4 缺陷代码的上下文窗口

Prenner 等人[39] 对缺陷代码的上下文展开了实证研究, 将 6 种不同的上下文窗口位置和 4 种不同的上下文窗口大小进行了对比. 上下文窗口位置指缺陷代码上下文行数的比例, 他们在上下文窗口位置上制定了如下实验设置: 仅上文、80% 上文、60% 上文、50% 上文、40% 上文、20% 上文和仅下文, 通过对比发现, 最优的窗口位置设置在不同数据集和不同的窗口大小上不一致, 总的来说, 60% 上文的设置在各个数据集上有较优的表现. 上下文窗口大小指缺陷代码上下文的行数, Prenner 等人[39] 在上下文窗口大小上进行了上下文各 1 行至上下文各 28 行的实验, 是在大窗口下进行微调的工作. 通过对比发现, 丰富的上下文内容能够提高模型成功修复缺陷的个数, 某

些在修复时需要使用上下文出现过的变量名、方法名等特定信息的缺陷能够通过增加上下文窗口显著提升被修复率. 这一研究不仅为未来的 APR 方案提供了上下文实验设置的指导, 也为 APR 数据集的发展提供了指引. 在实验中发现 TSSB-3M 和 ManySStuBs4J 数据集无法提供足够的上下文内容, 因此未来需要为 APR 数据集补充更丰富的上下文信息以提高模型修复能力.

5.5 小 结

目前在 APR 领域不仅有大量工作基于 LLM 设计代码修复的方案, 而且有大量对基于 LLM 的 APR 技术展开的实证研究. 现有的实证研究从大模型生成范围、大模型微调设置、安全漏洞类型的缺陷修复和缺陷代码的上下文窗口等角度分别对 LLM 在 APR 中的应用展开了分析, 本节在总结出实证研究的关键发现之上, 归纳出了实证研究作为未来工作方向的指引.

6 未来展望

随着 LLM 的迅速发展, APR 技术进入了新的高速发展阶段. 通过回顾近年基于 LLM 的 APR 技术结合对实证研究的总结, 我们将在本节指出在 APR 领域使用 LLM 的一些挑战并给出未来工作方向的展望.

6.1 提升 APR 修复效率

APR 技术的修复效率表示缺陷被修复时所花费的时间和算力成本, 更高的修复效率不仅能够节省缺陷修复的花费、扩大 APR 应用的覆盖范围, 还能在更短的时间窗口期内修复缺陷, 减小软件缺陷对系统和数据造成的危害. 在 APR 领域, 目前影响代码修复效率的主要挑战是“未登录词”(out-of-vocabulary, OOV)问题, 即当模型在推理的时候遇到了词表之外的词, 难以根据现有的词表表示这类新词. 特别是, 在 APR 领域, 编程语言中变量名、方法名、类名等单词的书写自由度极高, 传统的自然语言处理中的词级标记方法会导致词表规模显著增大以及词与词之间、token 与 token 之间的语义被稀释, 最终使得运算成本增加, 修复效率降低.

目前针对 OOV 问题, 主流的方法是使用子词级标记来提高词表的质量, 例如部分方案[2,9,38,44]使用 BPE, AlphaRepair[20]使用的字节级字节对编码(byte-level BPE, BBPE)和部分方案[52-54]使用句子片段(sentence piece), 但此类方案的泛化能力没有得到验证. 此外, CIRCLE[52]发现将 APR 技术推广到修复多种编程语言缺陷的时候, 子词级标记也依然面临着 OOV 问题, 无法理解与生成部分重要 token.

未来在面临更大规模的代码语料时, 为了提升代码修复效率, 可以从 3 方面展开研究: 1)词表压缩的方法. 在不影响模型性能的前提下, 缩小词表的规模, 进而降低运算的成本. 2)人机结合的词表设计方法. 在自动化分词的基础上, 借助领域专家经验提升词表的质量. 3)词表特征与修复效率间的实证研究. 对不同代码大模型使用的分词方案展开详细的实证研究, 分析不同词表规模、不同分词策略对 OOV 和修复效率的影响.

6.2 提升 APR 补丁质量

补丁质量表示 APR 技术所能修复缺陷的种类和难度, 更高的补丁质量能够消除缺陷, 为软件系统和数据要素带来更多威胁. 随着 LLM 和 APR 技术的发展, 现有的代码大语言模型已经对处理编程语言有了超出以往的能力, 且部分代码逻辑简单、代码结构简单的缺陷已经能被妥善修复. 但仍存在部分复杂的代码 LLM 不仅无法被理解, 更无法生成其中缺陷的修复补丁. 此外, 代码修复任务是一个较为精细的任务, 大模型的预训练任务与代码修复任务仍有较大的差距, 这直接影响了大模型生成补丁的质量.

为了提升 APR 补丁质量, 现有的工作借助 AST 的形式将树状代码结构提供给了 CodeT5 模型[51], 证明了将代码的更高级结构展示给 LLM 能够提高复杂代码的理解能力. 但由于 CodeT5 不能直接处理树状结构的代码输入, 目前依然需要借助树的遍历策略将 AST 转换成序列的形式供大模型修复, 增大了模型理解代码的难度.

未来的学者们可以尝试以下 4 个方向以提高补丁质量: 1)优化微调设计. 在预训练-微调范式下进一步优化微调的设计, 除仅将缺陷代码提供给 LLM, 还可以在模型微调的过程中加入更多代码修复的专业元素, 比如缺陷类型、缺陷成因、缺陷触发信息等额外数据, 帮助模型进一步提高补丁的质量. 2)探索提示学习. 在提示工程的基础之上探索提示学习在 APR 领域的潜力, 缩小模型预训练任务与代码修复任务的差距. CodeT5+[18]已经证明由提示学习引出的指令微调(instruction fine-tuning)技巧相比普通微调在处理零样本任务有更突出的能力. 3)采用代码高级表示. 探索其余代码结构的高级表示对代码缺陷修复的影响, 比如程序的控制流图(control-flow graph, CFG)或数据流图(data-flow graph, DFG)等图结构和中间表示(intermediate representation, IR)的代码表示. 4)利用

感知代码结构的大模型. 探索能够直接处理树、图等输入的 LLM 对代码修复任务的处理能力，如 GraphCodeBERT[73], AST-T5[74] 等，提升 LLM 对复杂代码结构的感知从而提升补丁质量. 5)探索国产大模型. 探索国产大模型在 APR 领域的潜力，如 PanGu-Coder2[75], CodeFuse[76] 等. 国产大模型中使用的许多前沿学习技术有助于 LLM 生成更高质量的补丁，PanGu-Coder2 证明强化学习技术可以利用代码调试、执行、测试、验收中的反馈信息提升 LLM 的代码生成能力，CodeFuse 借助指令微调技术提升了 LLM 完成多种代码任务的能力.

6.3 强化 APR 泛化能力

泛化的 APR 指能够修复多种编程语言类型、多种缺陷类型、多个修复位置、多种代码长度的代码修复技术. 拥有更强的泛化性能够提升 APR 技术的扩展性和适用性，使得 APR 在更多系统中消除软件缺陷对数据安全造成的威胁. 现有 APR 方案重点解决特定缺陷种类或特定编程语言的代码修复，而忽视了代码修复工具跨缺陷、跨语言的泛化能力. 这是由于目前缺少对现有缺陷和编程语言的归纳分析，无法总结出共性的知识供 LLM 学习. 此外，当前被研究的缺陷被局限在单段代码的缺陷上，即仅限 1 处连续代码修改的缺陷上. 对于多段代码的自动修复领域少有探索. 其首要原因是多段代码的修复难度要显著高于单段代码的修复难度，导致鲜有方案能够泛化至此类任务. 并且由于当前 LLM 的输入长度存在限制，APR 方案往往需要将缺陷代码的上下文根据长度限制截断，导致丢失部分缺陷上下文信息，无法泛化至较长代码的缺陷修复中.

目前，已有部分方案尝试提升 APR 技术的泛化能力，但仍面临着一些挑战. CIRCLE[52] 借助终身学习得到了一个能够修复跨编程语言的 APR 模型，但缺陷类型限制在软件 bug 之中. Huang 等人[2] 将多段缺陷的位置标记给大模型进行了 NMT 模式的多段代码缺陷修复，但发现随着缺陷待修复位置的增加，其方案的修复效果会显著地下降. Ye 等人[77] 将多段缺陷的修复拆分为多次单段代码修复，每次仅对 1 个待修复位置的代码进行修复. 其方案根据编译错误数量的减少和测试样例通过数量的增大判断单段修复在缺陷修复中的有效性. 但现实中，不是所有多段缺陷的表现都会随着单段代码的正确修复而改变，这部分缺陷在此模式下无法被成功拆分.

未来，学者们可以尝试从 4 个方向构建泛化的 APR 技术：1)归纳共性知识. 总结不同缺陷种类、不同编程语言之间的共性特征，例如 XSS 漏洞修复时可以在变量前加入编码函数修复，而类型错误(type error)修复时可以类似地在变量前加入强制类型转换声明来修复. 将归纳的共性知识供 LLM 学习以提高泛化能力. 2)采用更多学习策略. 结合先进的泛化能力强的模型训练方法，例如多任务学习(multi-task learning, MTL)、集成学习(ensemble learning)等，进而强化模型的泛化能力. 3)扩充修复成分. 深入地探索单段代码修复在多段缺陷中的效果，例如组合多处单段修复联合判断缺陷修复的有效性，将更多单段修复的补丁利用至多段代码的修复中. 4)提高上下文价值. 将更有修复价值的上下文提供给 LLM，例如借助代码切片技术[78] 或程序分析技术[79] 将长代码问题转化成较短代码修复问题.

6.4 全流程缺陷修复

在现实中，完整地修复一个缺陷不仅依赖 APR 的补丁生成技术，也需要编程领域的其他方向配合完成，例如提供缺陷所需修复具体位置的缺陷定位技术和判断补丁是否修复缺陷、是否改变代码功能和是否带来额外缺陷的补丁评估技术等. 随着 LLM 技术的发展，越来越多基于 LLM 的 APR 技术被提出，大多数方案将重点放在了利用 LLM 进行缺陷补丁的生成上. 而 LLM 技术在缺陷定位技术和补丁评估技术的潜力尚待挖掘. 因此目前仍然需要从缺陷修复各个阶段进行强化，构建全流程的缺陷修复.

未来，可以尝试以下 2 个方向进一步强化 APR 的修复能力，实现全流程缺陷修复：1)提升缺陷定位效果. 进一步深入挖掘 LLM 的能力，将缺陷代码与运行报错等代码异常信息应用到缺陷定位领域，结合 LLM 实现更精准、多位置、跨缺陷类别的缺陷定位技术. 2)提升补丁评估能力. 在现有的补丁评估技术之上借助 LLM 生成高质量的测试样例，或借助其他程序测试技术如模糊测试或符号执行[80] 来判断补丁的有效性.

7 总　　结

本文从数据要素安全的角度分析了软件系统和 APR 技术的重要性，系统性地回顾了近年 APR 领域的先进工作，并且对基于 LLM 的 APR 技术进行了深入的归纳分析. 根据 APR 的不同设计模式，我们将现有的基于 LLM 的 APR 方法分为了完形填空模式和神经机器翻译模式 2 类，并且详细分析比较了每一类的技术特点、优势和发展方向. 此外，我们对现有

的数据集、评价指标以及现有的实证研究进行了归纳总结. 最终我们总结了近年的文献, 指出了继续借助 LLM 推动 APR 发展的挑战和未来工作方向.

作者贡献声明: 许鹏宇负责文献收集和整理, 以及撰写论文; 况博裕负责论文修订; 苏铠负责部分论文修订; 付安民负责论文整体架构规划和修订.

参 考 文 献

[1] Wei Yuxiang, Xia Chunqiu, Zhang Lingming. Copiloting the copilots: Fusing large language models with completion engines for automated program repair[C]//Proc of the 31st ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering. New York: ACM, 2023: 172−184

[2] Huang Kai, Meng Xiangxin, Zhang Jian, et al. An empirical study on fine-tuning large language models of code for automated program repair[C]//Proc of the 38th IEEE/ACM Int Conf on Automated Software Engineering. Piscataway, NJ: IEEE, 2023: 1162−1174

[3] Tian Xiao, Chang Jiyou, Zhang Chi, et al. Survey of open-source software defect prediction method[J]. Journal of Computer Research and Development, 2023, 60(7): 1467−1488 (in Chinese)
(田笑, 常继友, 张弛, 等. 开源软件缺陷预测方法综述[J]. 计算机研究与发展, 2023, 60(7): 1467−1488)

[4] Gazzola L, Micucci D, Mariani L. Automatic software repair: A survey[J]. IEEE Transactions on Software Engineering, 2019, 45(1): 34−67

[5] Zhang Quanjun, Fang Chunrong, Ma Yuxiang, et al. A survey of learning-based automated program repair[J]. ACM Transactions on Software Engineering and Methodology, 2024, 33(2): 1−69

[6] Jiang Jiajun, Chen Junjie, Xiong Yingfei. Survey of automatic program repair techniques[J]. Journal of Software, 2021, 32(9): 2665−2690 (in Chinese)
(姜佳君, 陈俊洁, 熊英飞. 软件缺陷自动修复技术综述[J]. 软件学报, 2021, 32(9): 2665−2690)

[7] Wang Zan, Gao Jian, Chen Xiang, et al. Automatic program repair techniques: A survey[J]. Chinese Journal of Computers, 2018, 41(3): 588−610 (in Chinese)
(王赞, 部健, 陈翔, 等. 自动程序修复方法研究述评[J]. 计算机学报, 2018, 41(3): 588−610)

[8] Xu Tongtong, Liu Kui, Xia Xin. Survey on automated vulnerability repair[J]. Journal of Software, 2024, 35(1): 136−158 (in Chinese)
(徐同同, 刘逵, 夏鑫. 漏洞自动修复研究综述[J]. 软件学报, 2024, 35(1): 136−158)

[9] Fu M, Tantithamthavorn C, Le T, et al. VulRepair: A T5-based automated software vulnerability repair[C]//Proc of the 30th ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering. New York: ACM, 2022: 935−947

[10] Peng Changgen, He Xing, Tan Weijie, et al. Research status and countermeasures of artificial intelligence algorithm security[J]. Journal of Guizhou Normal University: Natural Sciences, 2022, 40(6): 1−16 (in Chinese)
(彭长根, 何兴, 谭伟杰, 等. 人工智能算法安全研究现状与对策[J]. 贵州师范大学学报: 自然科学版, 2022, 40(6): 1−16)

[11] Naveed H, Khan A U, Qiu Shi, et al. A comprehensive overview of large language models[J]. arXiv preprint, arXiv: 2307.06435, 2023

[12] Zheng Zibin, Ning Kaiwen, Wang Yanlin, et al. A survey of large language models for code: Evolution, benchmarking, and future trends[J]. arXiv preprint, arXiv: 2311.10372, 2023

[13] Iyer S, Konstas I, Cheung A, et al. Summarizing source code using a neural attention model[C]//Proc of the 54th Annual Meeting of the Association for Computational Linguistics. Stroudsburg, PA: ACL, 2016: 2073–2083

[14] Svyatkovskiy A, Deng Shaokun, Fu Shengyu, et al. Intellicode compose: Code generation using transformer[C]//Proc of the 28th ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering. New York: ACM, 2020: 1433–1443

[15] Raffel C, Shazeer N, Roberts A, et al. Exploring the limits of transfer learning with a unified text-to-text transformer[J]. Journal of Machine Learning Research, 2020, 21(1): 5485−5551

[16] Devlin J, Chang Mingwei , Lee K, et al. BERT: Pre-training of deep bidirectional transformers for language understanding[C]//Proc of the Conf of the North American Chapter of the Association for Computational Linguistics. Stroudsburg, PA: ACL, 2019: 4171–4186

[17] Feng Zhangyin, Guo Daya, Tang Duyu, et al. CodeBERT: A pre-trained model for programming and natural languages[C]//Proc of the Conf on Empirical Methods in Natural Language Processing. Stroudsburg, PA: ACL, 2020: 1536–1547

[18] Wang Yue, Le H, Gotmare A, et al. CodeT5+: Open code large language models for code understanding and generation[C]//Proc of the Conf on Empirical Methods in Natural Language Processing. Stroudsburg, PA: ACL, 2024: 1069–1088

[19] Wu Yi, Jiang Nan, Pham H V, et al. How effective are neural networks for fixing security vulnerabilities[C]//Proc of the 32nd ACM Int Symp on Software Testing and Analysis. New York: ACM, 2023: 1282–1294

[20] Xia Chunqiu, Zhang Lingming. Less training, more repairing please: Revisiting automated program repair via zero-shot learning[C]//Proc of the 30th ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering. New York: ACM, 2022: 959−971

[21] Yang Deheng, Mao Xiaoguang, Chen Liqian, et al. TransplantFix: Graph differencing-based code transplantation for automated program repair[C/OL]//Proc of the 37th IEEE/ACM Int Conf on Automated Software Engineering. Piscataway, NJ: IEEE, 2022[2024-07-12]. https://conf.researchr.org/details/ase-2022/ase-2022-research-papers/1/TransplantFix-Graph-Differencing-based-Code-Transplantation-for-Automated-Program-Re

[22] Wong C, Santiesteban P, Kästner C, et al. VarFix: Balancing edit expressiveness and search effectiveness in automated program repair[C]//Proc of the 29th ACM Joint Meeting on European

Software Engineering Conf and Symp on the Foundations of Software Engineering. New York: ACM, 2021: 354−366

[23] Lou Yiling, Chen Junjie, Zhang Lingming, et al. History-driven build failure fixing: How far are we[C]//Proc of the 28th ACM Int Symp on Software Testing and Analysis. New York: ACM, 2019: 43−54

[24] Abdessalem R B, Panichella A, Nejati S, et al. Automated repair of feature interaction failures in automated driving systems[C]//Proc of the 29th ACM Int Symp on Software Testing and Analysis. New York: ACM, 2020: 88−100

[25] Oh W, Oh H. PyTER: Effective program repair for Python type errors[C]//Proc of the 30th ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering. New York: ACM, 2022: 922−934

[26] Zheng Guolong, Nguyen T, Brida S G, et al. ATR: Template-based repair for alloy specifications[C]//Proc of the 31st ACM Int Symp on Software Testing and Analysis. New York: ACM, 2022: 666−677

[27] Koyuncu A, Liu Kui, Bissyandé T F, et al. iFixR: Bug report driven program repair[C]//Proc of the 27th ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering. New York: ACM, 2019: 314−325

[28] Liu Kui, Koyuncu A, Bissyandé T F, et al. You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems[C]//Proc of the 12th IEEE Int Conf on Software Testing, Verification and Validation. Piscataway, NJ: IEEE, 2019: 102−113

[29] Liu Kui, Koyuncu A, Kim D, et al. TBar: Revisiting template-based automated program repair[C]//Proc of the 28th ACM Int Symp on Software Testing and Analysis. New York: ACM, 2019: 31−42

[30] Ghanbari A, Benton S, Zhang Lingming. Practical program repair via bytecode mutation[C]//Proc of the 28th ACM Int Symp on Software Testing and Analysis. New York: ACM, 2019: 19−30

[31] Afzal A, Motwani M, Stolee K T, et al. SOSRepair: Expressive semantic search for real-world program repair[J]. IEEE Transactions on Software Engineering, 2021, 47(10): 2162−2181

[32] Niu Cuicui, Pan Zhengzhi, Liu Hai. Framework and approaches of differentially private generative adversarial networks: A survey[J]. Journal of Guizhou Normal University: Natural Sciences, 2022, 40(4): 84−99 (in Chinese)
(牛翠翠, 潘正芝, 刘海. 差分隐私生成式对抗网络的框架与方法综述[J]. 贵州师范大学学报: 自然科学版, 2022, 40(4): 84−99)

[33] Bavishi R, Yoshida H, Prasad M R. Phoenix: Automated data-driven synthesis of repairs for static analysis violations[C]//Proc of the 27th ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering. New York: ACM, 2019: 613−624

[34] Li Yi, Wang Shaohua, Nguyen T N. Dear: A novel deep learning-based approach for automated program repair[C]//Proc of the 44th ACM/IEEE Int Conf on Software Engineering. New York: ACM, 2022: 511−523

[35] Lutellier T, Pham H V, Pang L, et al. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair[C]//Proc of the 29th ACM Int Symp on Software Testing and Analysis. New York: ACM, 2020: 101−114

[36] Cao Heling, Liu Yu, Han Dong. Self-attention neural machine translation for automatic software repair[J]. Acta Electronica Sinica, 2024, 52(3): 945−956 (in Chinese)
(曹鹤玲, 刘昱, 韩栋. 基于自注意力机制神经机器翻译的软件缺陷自动修复方法[J]. 电子学报, 2024, 52(3): 945−956)

[37] Xia Chunqiu, Wei Yuxiang, Zhang Lingming. Automated program repair in the era of large pre-trained language models[C]//Proc of the 45th ACM/IEEE Int Conf on Software Engineering. New York: ACM, 2023: 1482–1494

[38] Jiang Nan, Lutellier T, Tan Lin. CURE: Code-aware neural machine translation for automatic program repair[C]//Proc of the 43rd ACM/IEEE Int Conf on Software Engineering. New York: ACM, 2021: 1161−1173

[39] Prenner J A, Robbes R. Out of context: How important is local context in neural program repair[C/OL]//Proc of the 46th ACM/IEEE Int Conf on Software Engineering. New York: ACM, 2024[2024-07-12]. https://conf.researchr.org/details/icse-2024/icse-2024-research-track/82/Out-of-Context-How-important-is-Local-Context-in-Neural-Program-Repair-

[40] Zhang Quanjun, Fang Chunrong, Zhang Tongke, et al. Gamma: Revisiting template-based automated program repair via mask prediction[C]//Proc of the 38th IEEE/ACM Int Conf on Automated Software Engineering. Piscataway, NJ: IEEE, 2023: 535−547

[41] Fan Zhiyu, Gao Xiang, Mirchev M, et al. Automated repair of programs from large language models[C]//Proc of the 45th ACM/IEEE Int Conf on Software Engineering. New York: ACM, 2023: 1469−1481

[42] Peng Yun, Gao Shuzheng, Gao Cuiyun, et al. Domain knowledge matters: Improving prompts with fix templates for repairing Python type errors[C/OL]//Proc of the 46th ACM/IEEE Int Conf on Software Engineering. New York: ACM, 2024[2024-07-12]. https://conf.researchr.org/details/icse-2024/icse-2024-research-track/8/Domain-Knowledge-Matters-Improving-Prompts-with-Fix-Templates-for-Repairing-Python-T

[43] Liu Kui, Koyuncu A, Kim D, et al. Avatar: Fixing semantic bugs with fix patterns of static analysis violations[C]//Proc of the 26th IEEE Int Conf on Software Analysis, Evolution and Reengineering. Piscataway, NJ: IEEE, 2019: 456–467

[44] Liu Xuliang, Zhong Hao. Mining stackoverflow for program repair[C]//Proc of the 25th IEEE Int Conf on Software Analysis, Evolution and Reengineering. Piscataway, NJ: IEEE, 2018: 118–129

[45] Koyuncu A, Liu Kui, Bissyande T F, et al. FixMiner: Mining relevant fix patterns for automated program repair[J]. Empirical Software Engineering, 2020, 25(3): 1980−2024

[46] Liu Kui, Kim D, Bissyande T F, et al. Mining fix patterns for findbugs violations[J]. IEEE Transactions on Software Engineering, 2018, 47(1): 165−188

[47] Guo Daya, Lu Shuai, Duan Nan, et al. UniXCoder: Unified cross-modal pre-training for code representation[C]//Proc of the 60th Annual Meeting of the Association for Computational Linguistics. Stroudsburg, PA: ACL, 2022: 7212−7225

[48] Wang Yue, Wang Weishi, Joty S, et al. CodeT5: Identifier-aware

unified pre-trained encoder-decoder models for code understanding and generation[C]//Proc of the Conf on Empirical Methods in Natural Language Processing. Stroudsburg, PA: ACL, 2021: 8696–8708

[49] Meng Yu, Krishnan J, Wang Sinong, et al. Representation deficiency in masked language modeling[C/OL]//Proc of the 12th Int Conf on Learning Representations. 2024[2024-07-12]. https://iclr.cc/virtual/2024/poster/18345

[50] Jiang Nan, Liu K, Lutellier T, et al. Impact of code language models on automated program repair[C]//Proc of the 45th ACM/IEEE Int Conf on Software Engineering. New York: ACM, 2023: 1430−1442

[51] Zhou Xin, Kim K, Xu Bowen, et al. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources[C/OL]//Proc of the 46th ACM/IEEE Int Conf on Software Engineering. New York: ACM, 2024[2024-07-12]. https://conf.researchr.org/details/icse-2024/icse-2024-research-track/222/Out-of-Sight-Out-of-Mind-Better-Automatic-Vulnerability-Repair-by-Broadening-Input-

[52] Yuan Wei, Zhang Quanjun, He T, et al. CIRCLE: Continual repair across programming languages[C]//Proc of the 31st ACM Int Symp on Software Testing and Analysis. New York: ACM, 2022: 678−690

[53] Ye He, Martinez M, Monperrus M. Neural program repair with execution-based backpropagation[C]//Proc of the 44th ACM/IEEE Int Conf on Software Engineering. New York: ACM, 2022: 1506−1518

[54] Ye He, Martinez M, Luo Xiapu, et al. SelfAPR: Self-supervised program repair with test execution diagnostics[C/OL]//Proc of the 37th IEEE/ACM Int Conf on Automated Software Engineering. Piscataway, NJ: IEEE, 2022[2024-07-12]. https://conf.researchr.org/details/ase-2022/ase-2022-research-papers/45/SelfAPR-Self-supervised-Program-Repair-with-Test-Execution-Diagnostics

[55] Wang Weishi, Wang Yue, Joty S, et al. RAP-Gen: Retrieval-augmented patch generation with codet5 for automatic program repair[C]//Proc of the 31st ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering. New York: ACM, 2023: 146−158

[56] Just R, Jalali D, Ernst M D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs[C]//Proc of the 23rd ACM Int Symp on Software Testing and Analysis. New York: ACM, 2014: 437–440

[57] Zhu Qihao, Sun Zeyu, Xiao Yuanan, et al. A syntax-guided edit decoder for neural program repair[C]//Proc of the 29th ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering. New York: ACM, 2021: 341−353

[58] Tufano M, Watson C, Bavota G, et al. An empirical study on learning bug-fixing patches in the wild via neural machine translation[J]. ACM Transactions on Software Engineering and Methodology, 2019, 28(4): 1−29

[59] Lin D, Koppel J, Chen A, et al. QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge[C]//Proc Companion of the ACM SIGPLAN Int Conf on Systems, Programming, Languages, and Applications: Software for Humanity. New York: ACM, 2017: 55–56

[60] Chen Zimin, Kommrusch S, Tufano M, et al. SequenceR: Sequence-to-sequence learning for end-to-end program repair[J]. ACM Transactions on Software Engineering and Methodology, 2021, 47(9): 1943−1959

[61] Berabi B, He Jingxuan, Raychev V, et al. TFix: Learning to fix coding errors with a text-to-text transformer[C]//Proc of the 20th Int Conf on Machine Learning. New York: ACM, 2021: 780–791

[62] Fan Jiahao, Li Yi, Wang Shaohua, et al. A C/C++ code vulnerability dataset with code changes and CVE summaries[C]//Proc of the 17th IEEE/ACM Int Conf on Mining Software Repositories. Piscataway, NJ: IEEE, 2020: 508−512

[63] Bhandari G, Naseer A, Moonen L. CVEFixes: Automated collection of vulnerabilities and their fixes from open-source software[C]//Proc of the 17th Int Conf on Predictive Models and Data Analytics in Software Engineering. New York: ACM, 2021: 30−39

[64] Goues C L, Holtschulte N, Smith E K, et al. The manybugs and introclass benchmarks for automated repair of C programs[J]. IEEE Transactions on Software Engineering, 2015, 41(12): 1236−1256

[65] Bui Q, Scandariato R, Ferreyra N E D. Vul4J: A dataset of reproducible Java vulnerabilities geared towards the study of program repair techniques[C]//Proc of the 19th IEEE/ACM Int Conf on Mining Software Repositories. Piscataway, NJ: IEEE, 2022: 464–468

[66] Hanam Q, Brito F S M, Mesbah A. Discovering bug patterns in javascript[C]//Proc of the 24th ACM SIGSOFT Int Symp on Foundations of Software Engineering. New York: ACM, 2016: 144–156

[67] Saha R K, Lyu Y, Lam W, et al. Bugs.Jar: A large-scale, diverse dataset of real-world Java bugs[C]//Proc of the 15th IEEE/ACM Int Conf on Mining Software Repositories. Piscataway, NJ: IEEE, 2018: 10–13

[68] Monperrus M, Martinez M, Ye He, et al. MegaDiff: A dataset of 600 k Java source code changes categorized by diff size[J]. arXiv preprint, arXiv: 2108.04631, 2021

[69] Madeiral F, Urli S, Maia M, et al. BEARS: An extensible Java bug benchmark for automatic program repair studies[C]//Proc of the 26th IEEE Int Conf on Software Analysis, Evolution and Reengineering. Piscataway, NJ: IEEE, 2019: 468−478

[70] Chen Zimin, Monperrus M. The codrep machine learning on source code competition[J]. arXiv preprint, arXiv: 1807.03200, 2018

[71] Richter C, Wehrheim H. TSSB-3M: Mining single statement bugs at massive scale[C]//Proc of the 19th IEEE/ACM Int Conf on Mining Software Repositories. Piscataway, NJ: IEEE, 2022: 418–422

[72] Karampatsis R, Sutton C. How often do single statement bugs occur? The ManySStuBs4J dataset[C]//Proc of the 17th IEEE/ACM Int Conf on Mining Software Repositories. Piscataway, NJ: IEEE, 2020: 573–577

[73] Guo Daya, Ren Shuo, Lu Shuai, et al. GraphCodeBERT: Pre-training code representations with data flow[C/OL]//Proc of the 9th Int Conf on Learning Representations. 2021[2024-07-12]. https://iclr.cc/virtual/2021/poster/2598

[74] Gong Linyuan, Elhoushi M, Cheung A. AST-T5: Structure-aware pretraining for code generation and understanding[J]. arXiv preprint,

arXiv: 2401.03003, 2024

[75] Shen Bo, Zhang Jiaxin, Chen Taihong, et al. PanGu-Coder2: Boosting large language models for code with ranking feedback[J]. arXiv preprint, arXiv: 2307.14936, 2023

[76] Di Peng, Li Jianguo, Yu Hang, et al. CodeFuse-13b: A pretrained multi-lingual code large language model[C]//Proc of the 46th Int Conf on Software Engineering: Software Engineering in Practice. Piscataway, NJ: IEEE, 2024: 418-429

[77] Ye He, Monperrus M. ITER: Iterative neural repair for multi-location patches[C/OL]//Proc of the 46th ACM/IEEE Int Conf on Software Engineering. New York: ACM, 2024[2024-07-12]. https://conf.researchr.org/details/icse-2024/icse-2024-research-track/55/ITER-Iterative-Neural-Repair-for-Multi-Location-Patches

[78] Mei Rui, Yan Hanbing, Shen Yuan, et al. Application research of slicing technology of binary executables in malware detection[J]. Journal of Cyber Security, 2021, 6(3): 125-140 (in Chinese)
(梅瑞, 严寒冰, 沈元, 等. 二进制代码切片技术在恶意代码检测中的应用研究[J]. 信息安全学报, 2021, 6(3): 125-140)

[79] Zhang Jian, Zhang Chao, Xuan Jifeng, et al. Recent progress in program analysis[J]. Journal of Software, 2019, 30(1): 80-109 (in Chinese)
(张健, 张超, 玄跻峰, 等. 程序分析研究进展[J]. 软件学报, 2019, 30(1): 80-109)

[80] Xie Xiaofei, Li Xiaohong, Chen Xiang, et al. Hybrid testing based on symbolic execution and fuzzing[J]. Journal of Software, 2019, 30(10): 3071-3089 (in Chinese)
(谢肖飞, 李晓红, 陈翔, 等. 基于符号执行与模糊测试的混合测试方法[J]. 软件学报, 2019, 30(10): 3071-3089)



Xu Pengyu, born in 1999. Master candidate. His main research interests include large language model and automated program repair.

许鹏宇, 1999 年生. 硕士研究生. 主要研究方向为大语言模型、自动代码修复.



Kuang Boyu, born in 1994. PhD. His main research interests include IoT security and IoV security.

况博裕, 1994 年生. 博士. 主要研究方向为物联网安全、车联网安全.



Su Mang, born in 1987. PhD, associate professor. Her main research interests include cloud security, access control, and right management.

苏　铓, 1987 年生. 博士, 副教授. 主要研究方向为云安全、访问控制、权限管理.



Fu Anmin, born in 1981. PhD, professor, PhD supervisor. Senior member of CCF. His main research interests include cryptography and privacy preserving.

付安民, 1981 年生. 博士, 教授, 博士生导师. CCF 高级会员. 主要研究方向为密码学、隐私保护.