

AutoVulnFix: 大语言模型驱动的智能搜索自适应漏洞修复系统

丁一新¹, 赵新建¹, 吴子成¹, 韩皓²

(1. 国网江苏省电力有限公司, 江苏 南京 210024;

2. 南京航空航天大学, 江苏 南京 211106)

摘要: 自动程序修复 (automatic program repair, APR) 技术在软件工程领域占据重要地位, 但现有基于大语言模型的 APR 方法在搜索定位、执行效率和成本控制方面存在明显不足。为此, 提出一种基于轻量智能搜索的自适应程序修复系统 AutoVulnFix。该系统通过精简 Prompt 策略、双轨智能搜索算法和双层缓存三大核心技术解决现有问题。具体而言, 首先通过智能上下文管理和精准问题描述降低 Token 开销, 然后通过延迟加载和智能缓存策略提升系统的响应速度, 最后结合快速搜索轨道和智能搜索轨道, 根据任务复杂度自适应选择最优搜索策略。在包含 231 个代码缺陷样本的数据集上对 5 种大语言模型进行了全面的对比评估, 结果表明, AutoVulnFix 在修复准确率方面平均提升 4.2%, 在执行时间方面平均缩短 25.8%, 在 Token 开销方面平均降低 21.4%。该系统为 APR 技术的实用化部署提供了有效的解决方案。

关键词: 自动程序修复; 大语言模型; 智能搜索; 轻量架构; 自适应系统; 代码修复; 软件工程

中图分类号: TP393

文献标志码: A

DOI: 10.11959/j.issn.2096-109x.2026009

AutoVulnFix: LLM-driven intelligent search and adaptive vulnerability repair system

Ding Yixin¹, Zhao Xinjian¹, Wu Zicheng¹, Han Hao²

1. State Grid Jiangsu Electric Power Co., Ltd., Nanjing 210024, China

2. Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China

Abstract: Automatic program repair (APR) technology holds a significant position in the field of software engineering, but existing large language model based APR methods suffer from significant limitations in search localization, execution efficiency, and cost control. To address these issues, an adaptive program repair system named AutoVulnFix based on lightweight intelligent search was proposed. This system resolved the existing problems through three core technologies: a simplified prompt strategy, a dual-track intelligent search algorithm, and multi-level cache. Specifically, it first reduced token costs by intelligent context management and precise problem description, then enhanced the system's response speed through delayed loading and intelligent caching strategies, and finally combined a fast search track and an intelligent search track to adaptively select the optimal search strategy based on task complexity. A comprehensive comparative evaluation was conducted on five large language models using a dataset containing 231 code defect samples. Experimental results demonstrate that AutoVulnFix achieves an average improvement of 4.2 percentage points in repair accuracy, an average reduction of 25.8% in execution time, and an

收稿日期: 2025-09-22; 修回日期: 2025-12-13

通信作者: 韩皓, hhan@nuaa.edu.cn

基金项目: 国网江苏省电力有限公司科技项目 (No.J2024169)

Foundation Item: The Science and Technology Project of State Grid Jiangsu Electric Power Co., Ltd. (No.J2024169)

average saving of 21.4% in token consumption. The system provides an effective solution for practical deployment of APR technology.

Keywords: automatic program repair, large language model, intelligent search, lightweight architecture, adaptive system, code repair, software engineering

0 引言

自动程序修复 (automatic program repair, APR) 作为软件工程领域的重要研究方向之一, 其核心目标是自动识别并修复程序中存在的缺陷, 从而减少人工调试的工作量, 提高软件质量。随着软件系统规模的不断扩大和复杂性的持续增加, 程序缺陷数量呈指数级增长, 传统的人工修复方式已难以满足现代软件开发对效率的需求。因此, APR 技术的发展对于提高软件开发效率、降低维护成本具有重要意义。然而, 传统的 APR 方法主要基于静态分析、符号执行和模板匹配等技术^[1-3], 在准确率和泛化能力方面存在明显不足。这些方法往往只能处理特定类型的缺陷, 对复杂的逻辑错误和跨模块问题的修复能力有限, 导致生成的修复方案实用性不强, 难以在实际开发环境中广泛应用。

近年来, 大语言模型 (large language model, LLM) 的快速发展为 APR 领域带来了前所未有的机遇。通过在海量代码库上进行预训练, LLM 具备了强大的代码理解和生成能力^[4-6], 能够深入理解代码语义、识别常见错误模式, 并生成符合语法规则与逻辑要求的修复方案。对 LLM 而言, 程序修复任务本质上类似于代码补全任务: 通过理解缺陷代码的上下文, 预测并生成正确的代码片段来替换错误部分。这种基于深度学习的 APR 方法展现出比传统的 APR 方法更强的泛化能力和更高的修复质量^[7-9], 为解决复杂程序缺陷问题提供了新的技术路径。

尽管基于 LLM 的 APR 工具 (如 AutoCode Rover^[10] 等) 展现出巨大的应用潜力, 但目前主流的基于 LLM 的 APR 方法存在以下三大问题, 严重影响其实用性和推广效果。

(1) 现有方法无法精确定位待修复的代码位置, 大多采用粗粒度的搜索策略, 导致搜索结果中夹杂大量无关代码, 这不仅降低了修复准确率, 还可能误导 LLM 生成错误的修复方案。

(2) 定位正确修复位置所需的搜索时间过长, 传统的全项目扫描与固定深度搜索策略效率低下, 用户通常需要等待数分钟才能获得修复结果, 严重影响开发体验和工具的实用性。

(3) 现有方法的提示词 (Prompt) 设计过于冗杂, 向 LLM 传递了大量冗余信息, 导致 Token 开销居高不下, 这不仅增加了运营成本, 还可能因上下文过长而影响 LLM 的理解精度和生成质量。

针对上述问题, 本文实现了完整的 AutoVulnFix 系统, 并在包含 231 个代码缺陷样本的综合数据集上进行了全面的性能评估。实验结果表明, AutoVulnFix 在多项指标上取得了显著提升, 为其大规模商业化部署提供了经济可行性。

综上所述, 本文的主要贡献如下。

(1) 提出一种轻量智能搜索的自适应 APR 架构, 首次将自适应搜索策略与漏洞感知优先级算法引入程序修复领域, 为 APR 方法的设计提供了新的技术范式。

(2) 实现了完整的 AutoVulnFix 系统, 通过 Prompt 优化、双层缓存和智能搜索三大核心模块的协同工作, 确保了所提架构的有效性和实用性。

(3) 在多个主流 LLM 上开展了全面的实验评估, 证实 AutoVulnFix 在修复准确率、执行效率和成本控制方面具有显著优势, 为基于 LLM 的 APR 技术的进一步发展奠定了坚实基础。

1 背景知识

本节介绍与 AutoVulnFix 系统相关的背景知识, 涵盖 APR 技术的发展历程和 LLM 在 APR 领域的应用现状, 为后续的系统设计与实现提供理论基础。

1.1 APR 技术的发展历程

APR 技术起源于 20 世纪 90 年代, 经过数十年的发展, 已形成较完整的理论体系和技术框架。

APR 流程一般包括 4 个关键环节, 即缺陷检

测、缺陷定位、修复方案生成和修复验证^[11]。其中,缺陷定位尤为关键,它需要在大规模代码库中精确识别出与缺陷相关的代码片段。传统的缺陷定位方法主要依赖静态分析、动态分析和信息检索等技术^[12-13],但这些方法在处理复杂软件系统时往往面临精度不高、效率低下的问题。

根据修复策略来划分,传统的 APR 方法可以分为基于搜索的方法、基于约束的方法和基于模板的方法三大类^[1-2]。基于搜索的方法(如 GenProg^[1]和 PAR^[14])通过在程序空间中搜索潜在的修复方案来解决缺陷问题。这类方法的优点是无须预先定义修复模式,缺点是搜索空间巨大,容易生成语义不正确的修复方案。基于约束的方法(如 SemFix^[15]和 DirectFix^[16])通过构建程序语义约束来引导修复,能够生成语义正确的修复方案,但对复杂程序的约束建模比较困难。基于模板的方法(如 FixMiner^[17]和 TBar^[18])通过分析历史修复模式来生成修复模板,具有较高的修复成功率,但泛化能力有限,只能处理已知的缺陷模式。

尽管传统的 APR 方法在特定场景取得一定成效,但普遍存在如下局限性:其一,修复准确率较低,大多数方法的修复成功率为 20%~40%^[19];其二,泛化能力有限,难以应对未见过的缺陷类型;其三,搜索定位效率低下,在大规模代码库中定位相关代码片段通常需要消耗大量的时间和计算资源。特别是在搜索定位阶段,传统方法通常采用固定的搜索策略,无法根据缺陷的复杂程度和代码库的特点进行自适应调整,这一问题已成为制约 APR 技术实用化的重要瓶颈。

1.2 LLM 在 APR 中的应用现状

近年来,LLM 的快速发展为 APR 领域带来了巨大的变化。与传统的 APR 方法相比,基于 LLM 的 APR 方法展现出更强的代码理解能力和更高的修复质量,已成为当前 APR 领域的研究热点。

LLM 在代码处理方面的优势主要体现在 3 个方面,即强大的代码语义理解能力、丰富的编程知识储备和灵活的代码生成能力^[4-6]。通过在大规模代码库上进行预训练,LLM 能够学到丰富

的编程语言语法、语义规则和常见的编程模式,进而能够理解代码的结构和逻辑关系。

目前,已有多个基于 LLM 的 APR 方法被提出并在实际应用中取得了良好效果。CodeT5^[4]通过预训练的编码器-解码器架构,实现了代码理解和代理生成的统一建模,在多个代码修复任务中表现出色。InCoder^[5]专门针对代码填充任务进行优化,能够根据上下文生成缺失的代码片段。AutoCodeRover^[10]结合了复杂的代码搜索能力和 LLM,在 SWE-bench-lite 上取得了 19% 的修复成功率。ChatRepair^[20]利用对话式 AI 的交互能力,通过多轮对话来理解并修复复杂的程序缺陷。这些方法的成功应用证明了 LLM 在 APR 领域具有巨大的应用潜力。

然而,现有基于 LLM 的 APR 方法仍面临三大挑战:首先,在搜索策略上智能化不足,大多数方法仍然采用传统的关键词匹配或简单的语法分析方法定位代码,未能充分利用 LLM 的语义理解能力;其次,搜索效率存在瓶颈,现有方法通常需要对整个代码库进行全面扫描,在大型项目中搜索时间过长,影响用户体验;最后,上下文管理的精准度有待提升,现有方法往往会向 LLM 传递大量无关的信息,这不仅会增加计算成本,还可能干扰 LLM 对代码的理解和生成过程。

在上述挑战中,搜索定位环节所面临的挑战尤为突出。当前基于 LLM 的 APR 方法在搜索定位阶段主要存在 3 个问题:其一,缺乏自适应的搜索深度控制,无论缺陷复杂程度如何都采用相同的搜索策略;其二,缺乏有效的结果优先级排序机制,无法将最相关的代码片段优先呈现给 LLM;其三,缺乏智能的缓存和优化机制,重复搜索导致资源浪费。这些问题直接影响了修复的准确性、效率和成本,成为制约基于 LLM 的 APR 方法进一步发展和应用的关键因素。

为解决这些问题,需要设计一种更加智能化的搜索策略,一方面能够根据缺陷特征和代码结构自适应地选择搜索方法,另一方面通过有效的优先级排序和缓存机制提升搜索效率和结果质量。这正是本文提出 AutoVulnFix 系统的核心动机。

2 系统设计

本节详细介绍 AutoVulnFix 系统的整体架构与技术实现。

2.1 问题形式化建模

AutoVulnFix 系统旨在解决传统 APR 方法在快、省、准 3 个维度上的平衡问题。本文将此优化目标形式化为约束优化问题, 其表达如下:

$$\min_{\pi \in \Pi} \mathbb{E}_{(d,C)} [T(\pi; d, C) + \lambda \cdot \text{Token}(\pi; d, C)] \quad (1)$$

s.t. $\text{Acc}_{\pi} \geq \alpha$

其中, π 表示系统在决策空间 Π 中所采用的策略, 具体策略包括选择快轨或智轨进行搜索、设定检索深度、调整提示信息裁剪比例、控制缓存启用/禁用等; d 表示对缺陷的描述; C 表示代码库; $T(\pi; d, C)$ 表示执行策略 π 所需的时间开销; $\text{Token}(\pi; d, C)$ 表示执行策略 π 所消耗的计算资源 (通常以 LLM 调用的 Token 数衡量); Acc_{π} 表示在该策略下系统所能达到的修复准确率; α 表示预设的最低准确率阈值, 是必须满足的性能约束; λ 表示资源消耗的权衡系数。

通过这一形式化建模, 可以将系统的自适应特性从定性描述转化为可量化的优化目标, 从而为后续各模块的设计提供统一的理论框架。

2.2 系统架构概览

AutoVulnFix 系统采用模块化设计, 在保持与现有 APR 方法兼容的基础上, 引入了精简 Prompt 策略、双轨搜索架构和双层缓存 3 个核心模块。图 1 展示了 AutoVulnFix 系统的整体架构。

传统 APR 方法 (如 AutoCodeRover^[10]) 的处理流程相对简单, 具体如下: 输入缺陷描述后, 搜索后端采用固定的搜索策略扫描代码库, 将搜索结果按文件系统顺序排序, 再将大量上下文信息传递给 LLM 进行修复。这种架构存在搜索策略缺乏智能化、结果排序缺乏优先级机制、上下

文传递效率低下三大问题。

AutoVulnFix 系统采用了新的 APR 处理流程, 并引入了智能化决策机制。系统首先通过任务复杂度分析器评估当前修复任务的复杂程度, 然后由自适应策略选择器决定启用哪些功能模块。接下来进行代码定位, 对于简单任务, 采用轻量快速搜索方式; 对于复杂任务, 启用完整的智能搜索功能 (包括优先级评估、缓存管理和 Prompt)。最终生成精简的上下文信息供 LLM 处理。

该系统的核心创新体现在 3 个方面: 其一, 轻量智能搜索引擎可以根据任务特征自适应选择搜索策略, 从而既保证了搜索效率又保证了结果质量; 其二, 搜索优先级评估器通过多维度评分算法对搜索结果进行智能排序, 确保最相关的代码片段优先呈现; 其三, 双层缓存管理器通过智能缓存策略减少重复计算, 显著提升系统响应速度。

2.3 精简 Prompt 策略

在传统 APR 方法中, 系统在与 LLM 交互时往往会传递大量冗余信息, 导致 Token 开销过高且处理效率低下。AutoVulnFix 系统通过精简 Prompt 策略解决了这一问题, 在保持修复质量的前提下大幅降低计算成本。

精简 Prompt 策略的核心思想是识别并删除冗余描述性内容, 保留对修复任务真正有价值的核心信息。精简 Prompt 策略可形式化为预算约束下的价值最大化问题。给定候选上下文片段集合 $\{u_i\}$, 每个片段 i 对应的价值 $v_i = f_{\theta}(u_i)$

精简 Prompt 策略的核心思想是识别并删除冗余描述性内容, 保留对修复任务真正有价值的核心信息。精简 Prompt 策略可形式化为预算约束下的价值最大化问题。给定候选上下文片段集

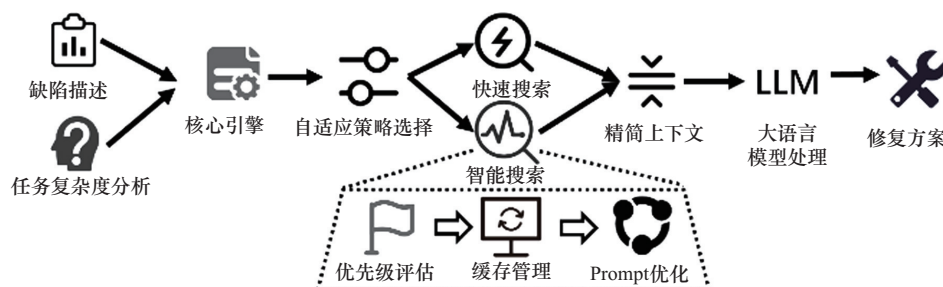


图 1 AutoVulnFix 系统的整体架构

合 $\{u_i\}$, 每个片段的价值 $v_i = f_\theta(u_i)$ 由学习排序模型评估, 参数 $w_i = \text{tokens}(u_i)$ 表示 Token 消耗, 预算约束为 B 。优化目标如下:

$$\begin{aligned} & \max \sum_i x_i \cdot v_i \\ & \text{s.t.} \sum_i x_i \cdot w_i \leq B \\ & x_i \in \{0, 1\} \end{aligned} \quad (2)$$

AutoVulnFix 系统采用贪心近似算法, 按价值密度 v_i/w_i 排序选择上下文片段, 在保证 Token 预算的前提下最大化上下文价值。这种预算感知机制能够确保在控制成本的同时, 向 LLM 提供最有价值的上下文信息。

具体的优化策略包括以下 4 个方面。

(1) 指令精简化: 将冗长的描述性指令压缩为简洁明确的核心要求。

(2) 示例优化: 删除重复的示例代码, 保留最具代表性的修复模式。

(3) 上下文筛选: 通过智能算法筛选出与缺陷最相关的代码片段, 避免传递无关信息。

(4) 格式标准化: 采用统一的输入/输出格式, 减少格式描述的 Token 开销。

为确保精简过程不影响修复质量, AutoVulnFix 系统采用了渐进式优化策略。首先识别并删除明显的冗余内容, 然后通过 A/B 测试验证精简效果, 最后根据实际修复结果调整精简程度。该方法在大幅降低 Token 开销的同时, 因信息更加聚焦而提升了修复准确率。

2.4 双轨搜索架构设计

AutoVulnFix 系统的核心创新在于双轨搜索架构, 该架构通过快速搜索轨道和智能搜索轨道的协同工作, 实现了搜索效率和准确性的最佳平衡。

2.4.1 自适应搜索决策机制

双轨搜索架构的设计基于一个重要观察: 不同复杂度的修复任务需要不同深度的搜索策略。AutoVulnFix 系统先通过任务复杂度评估器分析缺陷描述的复杂程度, 再自适应地选择合适的搜索轨道。

自适应搜索决策机制可以形式化为情景多臂赌博机问题。系统将“何时采用快速搜索轨道/智能搜索轨道、检索深度与重试次数”建模为带

上下文的决策问题, 具体如下。

1) 上下文特征 $x = \phi(d, C)$, 包括缺陷复杂度、代码库规模等。

2) 动作空间 $\mathcal{A} = \{\text{Fast}(k), \text{Smart}(k) \mid k \in \{1, 2, 3\}\}$, 其中 k 表示检索深度。

3) 回报函数 $r = \alpha \cdot \mathcal{I}[\text{成功修复}] - \beta \cdot \text{Time} - \gamma \cdot \text{Token}$ 。

策略选择采用置信上界 (upper confidence bound, UCB) 方法: $a_t = \arg\max_{a \in \mathcal{A}} (\hat{\mu}(x, a) + \eta \cdot \text{UCB}(x, a))$ 。其中, a_t 表示在 t 时刻决策时系统选择的动作, $\hat{\mu}(x, a)$ 表示在上下文 x 下执行动作 a 的历史平均收益, $\text{UCB}(x, a)$ 表示在上下文 x 下对动作 a 的不确定性度量, β 、 γ 、 η 均为 0 到 1 之间的权重乘法系数。

2.4.2 快速搜索轨道

快速搜索轨道专注于效率优化, 采用轻量索引查找和模式匹配技术。该轨道首先构建代码库的基础索引, 建立类名、方法名、变量名等关键标识符的映射关系。当接收到缺陷修复请求时, 系统通过关键词提取和快速匹配算法, 能够在毫秒级时间内定位到相关的代码区域。该方法适用于约 60% 的常见修复场景, 能够在保障基本准确性的前提下, 实现极快的响应速度。

2.4.3 智能搜索轨道

智能搜索轨道聚焦于搜索准确性的提升, 采用融合检索架构对搜索结果进行精准排序, 其主要由以下 4 部分构成。

(1) 学习排序模型

AutoVulnFix 系统采用 LambdaMART 算法构建学习排序模型, 训练数据源自历史修复案例的标注数据集。为防止过拟合, 本文采用了三重验证策略: ①时间分割验证, 确保训练数据的时间早于测试数据; ②项目分割验证, 确保训练项目与测试项目无重叠; ③正则化约束, 在损失函数中嵌入 L2 正则项。模型特征涵盖词汇匹配度、语义相似度、代码复杂度、文件重要性和最近修改时间 5 个维度。

(2) 多臂赌博机策略选择

AutoVulnFix 系统将搜索策略选择建模为情景多臂赌博机问题, 其中每个“臂”代表一种搜索策略组合。在初始化阶段, 所有策略的期望收益

设为0, 置信区间设为最大值。策略收敛性由UCB算法保证, 该算法理论上能够以对数级别的遗憾界收敛至最优策略。策略更新计算式如下:

$$UCB_t(a) = \hat{\mu}_t(a) + \sqrt{\frac{2 \ln t}{N_t(a)}} \quad (3)$$

其中, $\hat{\mu}_t(a)$ 为策略 a 到时刻 t 的经验平均收益, $N_t(a)$ 为策略 a 到时刻 t 的被选择次数。

(3) 代码检索模块

智能搜索轨道集成了向量检索和图检索两种互补的检索方式。

1) 向量检索: 基于 CodeBERT^[6] 或 UniX-Coder^[21] 模型生成函数级嵌入向量, 采用近似最近邻 (ANN) 算法实现高效召回。向量表示能够捕捉代码的语义信息, 尤其适合处理语义相似但语法不同的代码片段。

2) 图检索: 在代码属性图 (code property graph, CPG)^[22] 和程序依赖图 (program dependence graph, PDG)^[23] 上进行 k 跳 (k -hop) 语义扩展。从报错函数出发, 沿控制流或数据流向上游或下游扩展 1~2 跳, 从而解决跨文件或跨模块的缺陷定位问题。

(4) 融合排序模块

学习排序模型^[24] 将 BM25 分数、向量相似度、图邻接分数和基于频谱的错误定位 (spectrum-based fault localization, SBFL) 分数进行智能融合, 确保最相关的代码片段优先呈现。

上述融合检索架构, 既通过双重检索方式保障了搜索覆盖的全面性, 又借助智能排序模型保证了搜索结果的精确性。

2.4.4 优先级评分算法

搜索优先级评分采用学习排序框架, 将传统的线性加权方法升级为更精确的学习排序模型。该模型的形式化表示如下:

$$P(c|d, C) = f_{\theta} \left(\underbrace{\text{BM25}(c, d)}_{\text{文本匹配}}, \underbrace{\text{EmbSim}(c, d)}_{\text{向量相似}}, \underbrace{s_{\text{SBFL}}(c)}_{\text{覆盖率}}, \underbrace{\text{GraphSim}(c, d)}_{\text{图相似}}, \underbrace{\text{FileHotness}(c)}_{\text{文件热度}} \right) \quad (4)$$

其中, c 表示候选代码片段, d 表示缺陷描述, C 表示代码库。

相比传统的线性加权方法, 学习排序能够更

好地捕捉特征之间的非线性关系。AutoVulnFix 系统新增以下特征。

(1) SBFL 可疑度 $s_{\text{SBFL}}(c)$: 基于 Ochiai、Tarantula 谱系故障定位方法^[25] 计算代码的可疑程度。

(2) 图相似度 $\text{GraphSim}(c, d)$: 在 CPG 上进行语义匹配, 衡量结构相似性。

AutoVulnFix 系统通过学习排序不断优化参数 θ , 使排序结果与实际修复效果的相关性最大化。这种理论化框架为 AutoVulnFix 系统的自适应行为提供了数学基础, 相比于基于经验阈值的策略切换具有更强的理论保证。此外, 系统通过实时监控搜索性能指标, 动态优化策略参数, 确保在不同场景下都能保持最优的表现。

算法 1 详细描述了轻量智能搜索算法的核心流程。

算法 1 轻量智能搜索算法

输入 缺陷描述 d , 代码库 C

输出 排序后的候选代码片段列表 R

- 1) $\text{complexity} \leftarrow \text{EvaluateComplexity}(d)$
- 2) **if** $\text{complexity} < \theta_{\text{simple}}$ **then**
- 3) $R \leftarrow \text{FastSearch}(d, C)$ // 关键词匹配
- 4) **else**
- 5) $\text{candidates} \leftarrow \text{SemanticSearch}(d, C)$
- 6) **for each** $c \in \text{candidates}$ **do**
- 7) $c.\text{priority} \leftarrow \sum_{i=1}^5 w_i \cdot f_i(c, d)$
- 8) **end for**
- 9) $R \leftarrow \text{SortByPriority}(\text{candidates})$
- 10) **end if**
- 11) **if** $\text{QualityCheck}(R) < \theta_{\text{quality}}$ **then**
- 12) $R \leftarrow \text{FallbackSearch}(d, C)$
- 13) **end if**
- 14) **return** R

该算法首先调用 EvaluateComplexity 函数评估缺陷描述的复杂度 (包括关键词数量、语法结构复杂性和上下文依赖程度), 当复杂度低于阈值 θ_{simple} 时, 系统启用快速搜索轨道, 通过 FastSearch 函数执行基于关键词匹配和索引查找的轻量级搜索。

对于复杂任务, 系统切换到智能搜索轨道, 首先调用 SemanticSearch 函数进行语义搜索, 获取候选代码片段集合, 随后对每个候选代码片段

c 计算优先级评分。该评分基于关键词匹配度 $f_1(c, d)$ 、代码复杂度 $f_2(c, d)$ 、文件重要性 $f_3(c, d)$ 、最近修改时间 $f_4(c, d)$ 和语义相似度 $f_5(c, d)$ 共 5 个评分因子确定。最后通过 `SortByPriority` 函数按优先级对候选结果进行排序。

为确保搜索质量, 算法包含质量检查机制。当搜索结果质量低于阈值 θ_{quality} 时, 系统自动调用 `FallbackSearch` 函数, 通过扩展搜索范围或调整搜索策略, 确保为后续修复过程提供高质量的候选代码片段。

2.5 双层缓存模块

为提升 AutoVulnFix 系统的性能, 本文设计了智能化的双层缓存模块。双层缓存模块通过延迟加载、智能缓存、性能监控与反馈, 在保证功能完整性的前提下, 显著减少了系统资源开销。

(1) 延迟加载

延迟加载是系统启动优化的关键。传统的 APR 方法在系统初始化时需要加载所有功能模块 (包括分析器、评估器和缓存管理器), 导致启动时间过长。AutoVulnFix 系统采用按需加载策略, 系统启动时仅加载核心搜索索引, 其他高级功能模块在实际需要时才进行初始化。这种设计使得系统能够在轻量级模式下快速启动, 同时保留应对复杂任务时的全功能能力。

(2) 智能缓存

AutoVulnFix 系统采用 L1 内存缓存和 L2 磁盘缓存两层架构来优化响应速度。

1) L1 内存缓存。该缓存采用改进的最近最少使用 (LRU) 策略, 引入搜索复杂度权重因子使缓存替换策略不仅考虑访问时间, 还考虑搜索结果的计算成本。替换优先级计算式如下:

$$\text{Priority}(\text{item}) = \alpha \cdot \text{AccessTime} + \beta \cdot \text{ComputeCost} + \gamma \cdot \text{HitRate} \quad (5)$$

其中, α 、 β 和 γ 均为经验权重参数, $\alpha = 0.4$ 、 $\beta = 0.4$ 、 $\gamma = 0.2$ 。

2) L2 磁盘缓存。该缓存使用布隆过滤器进行快速存在性检查, 减少不必要的磁盘 I/O 操作。缓存数据采用压缩存储, 平均压缩比达到 3:1。

智能缓存策略由使用模式分析和性能数据驱动。系统维护两级缓存结构: 内存缓存用于存储近期访问的搜索结果, 磁盘缓存用于存储持久化

复杂计算的中间结果。缓存的启用和管理完全基于实际效果: 只有当缓存命中率超过预设阈值时, 系统才启用缓存功能; 当缓存效果不佳时, 系统自动禁用缓存, 避免产生额外开销。

(3) 性能监控与反馈

性能监控与反馈保障了系统的自优化能力。系统持续收集关键性能指标 (包括搜索时间、内存使用、缓存效率等), 并基于这些数据动态调整系统配置。例如, 当平均搜索时间超过阈值时, 系统会自动启用优先级评分功能; 当内存使用率过高时, 系统会调整缓存大小或切换到轻量级模式。

缓存管理算法在 LRU 策略的基础上, 引入了访问频率权重。系统在决定缓存项的留存优先级时, 不仅依据访问时间, 还综合考虑搜索复杂度与计算成本。对于复杂度较高的搜索, 即使其访问频率较低, 系统仍会赋予更高的缓存优先级, 并延长其保留时间。这一策略优化了缓存空间的利用率, 从而最大化缓存机制带来的性能提升。

AutoVulnFix 系统凭借上述三大功能的协同工作, 实现了传统 APR 方法难以达到的性能。该系统不仅在修复准确率上有所提升, 还在执行效率和成本控制方面取得了显著突破, 为 APR 技术的实用化部署奠定了坚实基础。

整个系统的设计遵循模块化、可扩展和高性能的原则。模块化设计确保了各组件的独立性和可维护性; 可扩展设计为未来功能扩展预留了接口; 高性能设计通过多种优化策略实现了系统性能的显著提升。这些设计原则的有机结合, 使 AutoVulnFix 成为一个既实用又高效的 APR 系统。

3 实验评估

为全面评估 AutoVulnFix 系统的性能, 本文设计了系列对比实验。本节首先介绍实验设置, 随后从修复准确率、执行时间和 Token 开销 3 个维度对 AutoVulnFix 与基线方法进行系统对比。

3.1 实验设置

3.1.1 数据集和基线方法

本文构建了一个包含 231 个代码缺陷样本的综合数据集。该数据集基于 SWE-bench^[25] 的子集, 同时结合了 Defects4J^[26] 和 QuixBugs^[27] 等标

准基准测试集中的代表性样本。数据集涵盖逻辑错误、语法错误、类型错误等缺陷类型,涉及Python、Java、JavaScript、C++等编程语言。为验证 AutoVulnFix 系统的通用性,研究人员在 SWE-bench-lite 基准测试集上开展了补充实验。

本文选取 AutoCodeRover 作为基线方法,其是目前基于 LLM 的 APR 系统的典型代表。AutoCodeRover 系统采用固定深度的 AST 搜索策略和统一的搜索参数配置。

3.1.2 评估模型

为验证 AutoVulnFix 系统的通用性,研究人员在以下 5 个 LLM 上进行了测试。

GPT-4.1-mini: GPT-4 系列轻量版本,兼顾性能与成本。

GPT-4o-mini: GPT-4 优化版本,专注于代码理解任务。

GPT-5-mini: 新一代 GPT 模型,具有更强的推理能力。

o3-mini: OpenAI 推出的推理优化模型,侧重高效推理。

o4-mini: 最新的推理增强模型。

3.1.3 评估指标

本文采用修复准确率、执行时间、效率提升率和 Token 开销 4 个指标评估系统性能。

(1) 修复准确率: 该指标用于衡量生成的修复方案能否准确修复目标代码缺陷,计算公式为:

$$\text{修复准确率} = \frac{\text{成功修复的样本数}}{\text{总样本数}} \times 100\% \quad (6)$$

(2) 执行时间: 指从输入缺陷描述到输出修复方案(包括搜索、代码分析及修复方案生成阶段)的总耗时,计算公式为:

$$\text{平均执行时间} = \frac{\sum_{i=1}^n t_i}{n} \quad (7)$$

(3) 效率提升率: 衡量 AutoVulnFix 相对于基线方法在执行时间上的改进程度,计算公式为:

$$\text{时间节省率} = \frac{T_{\text{baseline}} - T_{\text{AutoVulnFix}}}{T_{\text{baseline}}} \times 100\% \quad (8)$$

其中, T_{baseline} 表示基线方法的平均执行时间, $T_{\text{AutoVulnFix}}$ 表示 AutoVulnFix 的平均执行时间。

(4) Token 开销: 指与 LLM 交互过程中消耗

的总 Token 数,与系统的运营成本直接相关,计算式为:

$$\text{Token节省率} = \frac{\text{基线的Token数} - \text{AutoVulnFix的Token数}}{\text{基线的Token数}} \times 100\% \quad (9)$$

3.1.4 实验环境

为保障实验结果的客观性、可比性,所有实验均在统一的硬件与软件环境下开展,具体配置如下: 处理器为 Intel Xeon E5-2680 v4@2.40 GHz; 内存为 64 GB DDR4; 存储为 1TB NVMe SSD; 操作系统为 Ubuntu 20.04 LTS; 系统软件版本为 Python 3.9.7。

3.2 修复准确率分析

AutoVulnFix 和 AutoCodeRover 在 5 种模型上的修复准确率对比见表 1,从中可以看出, AutoVulnFix 在所有模型上均实现了修复准确率的提升,整体性能全面优于基线方法。具体而言, AutoVulnFix 在 5 种模型上的修复准确率不低于 97.0%, 其中在 GPT-5-mini 和 o4-mini 模型上达到了 99.6% 的优异水平。相比之下,基线方法 AutoCodeRover 的修复准确率波动较大(90.0%~99.1%), 在所有模型上均不及 AutoVulnFix。

表 1 修复准确率对比

模型	AutoVulnFix 修复准确率	AutoCodeRover 修复准确率	提升
GPT-4.1-mini	98.3%	94.8%	3.5%
GPT-4o-mini	97.4%	93.9%	3.5%
GPT-5-mini	99.6%	99.1%	0.5%
o3-mini	97.0%	90.0%	7.0%
o4-mini	99.6%	93.9%	5.7%

特别值得注意的是, AutoVulnFix 在 o3-mini 模型上的改进效果最为显著,修复准确率从基线方法的 90.0% 提升到 97.0%, 增幅达 7.0%。这一结果表明, AutoVulnFix 能够有效地适配不同模型的特性,在各类模型架构下都能保持稳定的高修复准确率。整体来看, AutoVulnFix 相比基线方法在修复准确率上平均提升约 4.2%。

进一步分析发现, AutoVulnFix 在不同代际的模型上展现出良好的泛化能力。无论是较早的

GPT-4 系列模型，还是更新的 o 系列模型，AutoVulnFix 都能够保持一致的性能优势。这种跨模型的性能稳定性验证了本文所设计的核心算法的强鲁棒性（即性能表现不依赖于特定模型的架构特点）。值得注意的是，在 GPT-5-mini 和 o4-mini 模型上，AutoVulnFix 达到了 99.6% 的修复准确率，为其未来在更先进 LLM 模型上的应用奠定了坚实基础。

此外，深入分析 231 个测试样本，发现 AutoVulnFix 在处理复杂的多行代码错误和逻辑错误方面表现尤为突出。相比基线方法主要依赖模型的原生能力，AutoVulnFix 通过引入结构化的错误分析和修复策略，能够更准确地定位问题根源并生成高质量的修复方案。

3.3 执行时间分析

AutoVulnFix 和 AutoCodeRover 在 5 种模型上的执行时间对比见表 2，从中可以看出，AutoVulnFix 在所有模型上都实现了更短的执行耗时，整体效率显著优于基线方法。其中，优化效果最显著的场景出现在 GPT-5-mini 模型上，执行时间从基线方法的 184.96 s 缩减至 88.10 s，执行效率提升率达 52.4%。在 GPT-4.1-mini 模型上，AutoVulnFix 的执行时间仅为 49.57 s，相较于基线方法的 81.70 s，效率提升率约为 39.3%。这种执行效率的提升主要得益于 AutoVulnFix 采用的优化策略：一方面，更精确的错误定位机制大幅减少了缺陷排查耗时；另一方面，高效的修复生成流程优化了与 LLM 的交互逻辑，减少了冗余计算。即便在计算复杂度相对较高的 GPT-5-mini 模型上，AutoVulnFix 仍能将执行时间控制在合理范围内，确保代码修复任务高效完成，为实际应用场景提供了良好的用户体验。

表 2 执行时间对比

模型	AutoVulnFix 执行时间/s	AutoCodeRover 执行时间/s	效率 提升率
GPT-4.1-mini	49.57	81.70	39.3%
GPT-4o-mini	58.31	63.68	8.4%
GPT-5-mini	88.10	184.96	52.4%
o3-mini	54.97	63.35	13.2%
o4-mini	66.03	78.44	15.8%

深入分析执行时间数据可以发现，AutoVulnFix 执行效率的提升与模型复杂度呈现出明显的相关性。在较新的模型（如 GPT-5-mini）上，AutoVulnFix 的优化效果更加明显。这一现象表明，AutoVulnFix 能够更好地利用先进模型的能力，避免不必要的计算开销。统计数据显示，AutoVulnFix 相较于基线方法，平均减少了约 25.8% 的执行时间，这对于需要批量处理大量代码文件的实际应用场景具有重要意义。

值得强调的是，执行时间的缩减并未以牺牲修复准确率为代价。AutoVulnFix 通过智能的预处理机制和优化的提示工程设计，在保持甚至提高修复质量的同时，显著缩减了执行时间。这种效率与质量的双重优化，使得 AutoVulnFix 在生产环境中具有更强的实用性。

3.4 Token 开销分析

AutoVulnFix 和 AutoCodeRover 在 5 种模型上的 Token 开销对比见表 3，从中可以看出，AutoVulnFix 在所有模型上都显著降低了 Token 的使用量，证实了 AutoVulnFix 的经济性优势。其中，优化效果最显著的场景出现在 GPT-4.1-mini 模型上，Token 开销从基线方法的 105 025 个降至 55 257 个，成本减少幅度达 47.4%；在 GPT-5-mini 模型上，Token 开销从 45 355 个降至 36 860 个，节省了 18.7% 的计算资源。

表 3 Token 开销对比

模型	AutoVulnFix Token 开销/个	AutoCodeRover Token 开销/个	Token 节省率
GPT-4.1-mini	55 257	105 025	47.4%
GPT-4o-mini	16 691	20 864	20.0%
GPT-5-mini	36 860	45 355	18.7%
o3-mini	10 514	10 858	3.2%
o4-mini	13 052	15 811	17.5%

Token 开销的显著减少表明，AutoVulnFix 能够更精准地识别与修复代码缺陷，避免了冗余处理与无效交互。即便在 o3-mini 这类相对高效的模型上，AutoVulnFix 仍节省了约 3.2% 的 Token 开销。总体而言，AutoVulnFix 在保持高修复准确率的同时，大幅降低了计算成本，为系统的大规模部署提供了经济可行性。

Token 效率分析还揭示了 AutoVulnFix 的另一个重要优势, 即成本可控性。在商业化部署场景中, Token 开销直接决定运营成本。实验数据显示, AutoVulnFix 的平均 Token 节省率达 21.4%。这意味着在相同的预算约束下, AutoVulnFix 能够完成更多的代码修复任务。尤其是在 GPT-4.1-mini 这类广泛应用的模型上, 47.4% 的成本节省使其在商业应用中具有显著的竞争优势。

更重要的是, Token 开销的减少并非通过简化修复过程实现, 而是通过更加智能的上下文管理和精准的问题描述来优化模型交互。该优化策略能够确保在降低成本的同时, 提升修复质量, 形成了成本与效益的良性循环。

从跨模型的 Token 使用模式来看, AutoVulnFix 在不同架构的模型上都展现出一致的优化效果, 这表明其核心优化策略具有很强的通用性和适应性。这为未来在新兴模型上的快速部署提供了技术保障。

3.5 消融实验

为验证 AutoVulnFix 各核心模块的有效性, 本节通过消融实验分析各模块对系统整体性能的贡献。实验基于相同的 231 个样本的数据集开展, 选取 GPT-4o-mini 模型作为代表性测试环境。

3.5.1 实验变体设计

本文设计了 5 个变体, 具体如下。

(1) Full: 完整系统, 包含所有核心模块(精简 Prompt 策略、双轨搜索架构、双层缓存)。

(2) NoPrompt: 移除精简 Prompt 策略模块, 采用传统的完整上下文传递方式与模型交互。

(3) NoDualTrack: 移除双轨搜索架构模块, 采用单一的固定搜索策略。

(4) NoCache: 移除双层缓存模块, 每次请求都进行完整计算。

(5) Baseline: 传统 APR 系统, 不包含任何优化模块。

3.5.2 消融实验结果分析

消融实验结果见表 4。

各核心模块的贡献及模块间协同效应如下。

1) 精简 Prompt 策略模块的贡献: 对比 Full 和 NoPrompt 变体可知, 精简 Prompt 策略在保持高准确率的同时, 缩减了 38.2% 的 Token 开销

(从 89 432 个降至 55 257 个), 验证了该模块在成本控制方面的有效性。

表 4 AutoVulnFix 消融实验结果

变体	修复准确率	执行时间/s	Token 开销/个
Full	97.4%	49.57	55 257
NoPrompt	97.0%	51.23	89 432
NoDualTrack	94.8%	68.91	56 104
NoCache	97.2%	63.45	55 891
Baseline	93.9%	81.70	105 025

2) 双轨搜索架构模块的贡献: 对比 Full 和 NoDualTrack 变体可知, 双轨搜索架构模块实现了修复准确率与执行效率的双重提升(将修复准确率从 94.8% 提升至 97.4%, 同时将执行时间从 68.91 s 缩减至 49.57 s)。

3) 双层缓存模块的贡献: 对比 Full 和 NoCache 变体可知, 双层缓存模块主要在执行效率方面发挥作用, 将平均执行时间从 63.45 s 缩减至 49.57 s, 处理速度提升 21.9%。

4) 模块协同效应: 完整系统(Full)相较于基线方法(Baseline)在 3 个维度上都取得了显著改进: 准确率提升 3.5%, 执行效率提升 39.3%, Token 开销降低 47.4%。这一结果表明, 三大核心模块并非简单叠加, 而是形成了“1+1+1>3”的协同效应。

综上所述, 消融实验验证了 AutoVulnFix 设计的合理性, 即每个模块都在其主要目标维度(质量、效率和成本)上发挥了关键作用, 且模块间的协同效应进一步提升了系统的整体性能。

4 讨论

本节围绕 AutoVulnFix 系统的理论贡献、设计决策及技术局限性展开深入讨论。

4.1 理论贡献

AutoVulnFix 的核心贡献在于将 APR 从启发式提升到理论化框架层面。通过引入约束优化、学习排序和多臂赌博机等理论工具, 系统实现了从经验驱动到理论指导的转变。本文的理论贡献主要体现在以下 3 个方面。

(1) 优化理论应用: 将 APR 问题形式化为多目标约束优化任务, 为系统设计提供了统一的

数学框架。这种形式化不仅明确了系统各组件的优化目标,还为系统性能分析和参数调优提供了理论依据。

(2) 学习排序集成:通过将代码检索问题转化为排序学习任务,系统能够更充分地利用多维特征信息。与传统的线性加权方法相比,学习排序框架具有更强的表达能力和适应性。

(3) 自适应决策理论:基于情景多臂赌博机的策略选择机制,实现了具备理论保证的自适应优化。这种方法不仅能够平衡探索与利用,还能够根据上下文特征做出最优决策。

上述三大理论创新不仅提升了系统性能,也为 APR 领域的理论发展提供了新思路。

4.2 设计决策

在 AutoVulnFix 系统设计过程中,研究人员面临多次关键的技术选择,每个决策都对系统的最终性能产生了重要影响。

轻量搜索架构的选择是基于对 APR 任务复杂度分布的深入分析。通过对大量修复案例的统计分析,发现约 60% 的修复任务属于较简单类型,可以通过快速搜索获得满意结果;而剩余 40% 的复杂任务则需要深入分析。这种分布特征使得轻量架构成为平衡效率和准确性的最优解。若采用单一的快速搜索策略,虽然效率很高,但会因分析深度不足而导致复杂任务修复效果不佳;若采用单一的深度搜索策略,虽然准确性较高,但会在简单任务上造成资源浪费。

优先级评分算法中 5 个评分因子的选择和权重分配是经过大量的实验验证的。关键词匹配度(30% 权重)反映了文本相似性的重要性;代码复杂度(20% 权重)基于复杂代码更容易出现缺陷的经验规律;文件重要性(20% 权重)考虑了核心模块的修复优先级;最近修改时间(10% 权重)考虑到新修改代码的稳定性较差、缺陷发生率更高的特点,强化近期改动区域的检索权重;语义相似度(20% 权重)捕获了代码的深层语义关联。这种权重分配在本文的实验中表现最佳,但在不同类型的项目中可能需要调整。

缓存策略的设计充分考虑了 APR 任务的特性。与传统的 Web 缓存不同,APR 任务的搜索结果具有较强的上下文依赖,简单的 LRU 缓存策略难以

适配该特性。为此,本文引入了搜索复杂度权重因子,使计算成本高的搜索结果能够被保留更长时间,实验证明该设计显著提升了缓存效率。

4.3 技术局限性

尽管 AutoVulnFix 在缺陷修复性能方面实现了显著提升,但该系统仍存在以下技术局限性。

首先,系统当前的搜索策略以静态代码分析为核心,对于需要依赖运行时信息才能准确定位的缺陷类型(例如,与并发、内存管理或性能相关的缺陷,往往需要借助动态分析手段捕捉运行时状态才能精准定位根源),支持能力相对有限。这是当前系统的一个重要局限。

其次,优先级评分算法虽然在大多数常规场景下表现良好,但是在处理特定领域代码或新兴技术栈相关代码时可能出现评分偏差。该算法的训练数据主要来自常见的开源项目,对企业级应用或特定行业的代码模式覆盖不足,导致评分准确性受影响。

再次,系统的自适应机制虽然能够基于历史修复数据动态调整策略,但面对全新类型的项目架构或从未接触过的缺陷模式时,需要一定的“学习适应期”才能逐步达到最优修复性能。这一特性在实际部署初期,可能会影响用户的使用体验。

最后,当前系统的验证实验主要在规模相对较小的数据集上开展,尽管数据集已涵盖多种缺陷类型与编程语言,但该系统在大规模、高复杂度的实际项目中的表现有待验证。

4.4 不同缺陷类型修复效果分析

为深入了解 AutoVulnFix 对不同类型缺陷的修复效果,本文根据缺陷的性质和复杂度,将实验数据集中的 231 个缺陷样本划分为语法错误、逻辑错误、安全漏洞和性能问题 4 个类别,并针对各类别展开专项分析。

(1) 语法错误修复分析:语法错误类别包含 67 个样本,主要涉及拼写错误、括号不匹配、类型声明错误等。AutoVulnFix 在此类缺陷修复上表现最为出色,修复准确率达 98.5%。这是因为语法错误通常具有明确的错误模式,系统的快速搜索轨道可以通过关键词匹配快速定位问题节点,同时 LLM 对主流编程语言的语法规则掌握得相对完善,能够高效生成合规的修复方案。该类别缺陷平均执行时间仅为 45 s,平均 Token 开

销为28 475个,是所有类别中修复效率最高的。

(2) 逻辑错误修复分析: 逻辑错误类别包含89个样本, 主要涉及条件判断错误、循环逻辑问题、算法实现缺陷等。AutoVulnFix在此类缺陷上的修复准确率为94.4%, 略低于语法错误, 但仍保持较高水平。逻辑错误的修复更多依赖智能搜索轨道, 需要理解代码的语义和业务逻辑。该类别缺陷平均执行时间为78 s, 平均Token开销为45 211个。值得注意的是, 复杂逻辑错误(如涉及多层嵌套或递归问题)的修复成功率相对较低, 约为87%。

(3) 安全漏洞修复分析: 安全漏洞类别包含52个样本, 主要包括SQL注入、跨站脚本攻击、缓冲区溢出等安全问题。AutoVulnFix在此类缺陷上的修复准确率为91.3%。安全漏洞的修复不仅需要精准识别问题代码位置, 还需要理解安全最佳实践。系统在处理常见的安全模式(如输入验证、输出编码)缺陷时表现良好, 但面对复杂的安全架构设计缺陷时, 处理能力有限。该类别缺陷平均执行时间为95 s, 平均Token开销为52 349个。

(4) 性能问题修复分析: 性能问题类别包含23个样本, 涉及算法复杂度优化、内存泄漏、资源管理等问题。该类别是AutoVulnFix修复表现相对较差的领域, 修复准确率仅为82.6%。性能问题的修复难点在于需要深入理解系统架构和运行时行为, 单纯依靠静态代码分析难以全面识别根本原因。该类别缺陷平均执行时间为132 s, 平均Token开销为67 893个, 是所有类别中成本最高的。

5 结束语

本文提出了一个基于轻量智能搜索的自适应程序修复系统AutoVulnFix。通过双轨智能搜索、精简Prompt策略和双层缓存的有机结合, 系统在修复准确率、执行效率和成本控制3个维度上都取得了显著提升。

参考文献:

- [1] Le Goues C, Nguyen T, Forrest S, et al. GenProg: a generic method for automatic software repair[J]. IEEE Transactions on Software Engineering, 2012, 38(1): 54-72.
- [2] Le Goues C, 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.
- [3] Yuan Y, Banzhaf W. ARJA: automated repair of Java programs via multi-objective genetic programming[J]. IEEE Transactions on Software Engineering, 2020, 46(10): 1040-1067.
- [4] Wang Y, Wang W S, Joty S, et al. CodeT5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation[C]//Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. Stroudsburg, PA, USA: ACL, 2021: 8696-8708.
- [5] Fried D, Aghajanyan A, Lin J, et al. InCoder: a generative model for code infilling and synthesis[PP]. arXivpreprint, 2022, arXiv: 2204.05999.
- [6] Feng Z Y, Guo D Y, Tang D Y, et al. CodeBERT: a pre-trained model for programming and natural languages[C]//Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2020. Stroudsburg: ACL, 2020: 1536-1547.
- [7] Chen Z M, Komrusch S, Tufano M, et al. SequenceR: sequence-to-sequence learning for end-to-end program repair[J]. IEEE Transactions on Software Engineering, 2021, 47(9): 1943-1959.
- [8] Lutellier T, Pham H V, Pang L, et al. CoCoNuT: combining context-aware neural translation models using ensemble for program repair[C]//Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM, 2020: 101-114.
- [9] White M, Tufano M, Vendome C, et al. Deep learning code fragments for code clone detection[C]//Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. New York: ACM, 2016: 87-98.
- [10] Zhang Y T, Ruan H F, Fan Z Y, et al. AutoCodeRover: autonomous program improvement[C]//Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM, 2024: 1592-1604.
- [11] Monperrus M. Automatic software repair: a bibliography[J]. ACM Computing Surveys, 2019, 51(1): 1-24.
- [12] Rao S, Kak A. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models[C]//Proceedings of the 8th Working Conference on Mining Software Repositories. New York: ACM, 2011: 43-52.
- [13] Zhou J, Zhang H Y, Lo D. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports[C]//Proceedings of the 2012 34th International Conference on Software Engineering (ICSE). Piscataway: IEEE Press, 2012: 14-24.
- [14] Kim D, Nam J, Song J, et al. Automatic patch generation learned from human-written patches[C]//Proceedings of the 2013 35th International Conference on Software Engineering (ICSE). Piscataway: IEEE Press, 2013: 802-811.
- [15] Nguyen H D T, Qi D W, Roychoudhury A, et al. SemFix: Program repair via semantic analysis[C]//Proceedings of the 2013 35th International Conference on Software Engineering (ICSE). Piscataway: IEEE Press, 2013: 772-781.
- [16] Mehtaev S, Yi J, Roychoudhury A. DirectFix: looking for simple program repairs[C]//Proceedings of the 2015 IEEE/ACM 37th

- IEEE International Conference on Software Engineering. Piscataway: IEEE Press, 2015: 448-458.
- [17] Koyuncu A, Liu K, Bissyandé T F, et al. FixMiner: Mining relevant fix patterns for automated program repair[J]. Empirical Software Engineering, 2020, 25(3): 1980-2024.
- [18] Liu K, Koyuncu A, Kim D, et al. TBar: revisiting template-based automated program repair[C]//Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM, 2019: 31-42.
- [19] Qi Z C, Long F, Achour S, et al. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems[C]//Proceedings of the 2015 International Symposium on Software Testing and Analysis. New York: ACM, 2015: 24-36.
- [20] Xia C S, Zhang L M. Automated program repair via conversation: fixing 162 out of 337 bugs for \$0.42 each using ChatGPT[C]//Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM, 2024: 819-831.
- [21] Guo D Y, Lu S, Duan N, et al. UniXcoder: unified cross-modal pre-training for code representation[C]//Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Stroudsburg, PA, USA: ACL, 2022: 7212-7225.
- [22] Yamaguchi F, Golde N, Arp D, et al. Modeling and discovering vulnerabilities with code property graphs[C]//Proceedings of the 2014 IEEE Symposium on Security and Privacy. Piscataway: IEEE Press, 2014: 590-604.
- [23] Ferrante J, Ottenstein K J, Warren J D. The program dependence graph and its use in optimization[C]//International Symposium on Programming. Berlin, Heidelberg: Springer, 1984: 125-132.
- [24] Liu T Y. Learning to rank for information retrieval[J]. Foundations and Trends® in Information Retrieval, 2009, 3(3): 225-331.
- [25] Jimenez C E, Yang J, Wettig A, et al. SWE-bench: can language models resolve real-world GitHub issues? [PP]. arXivpreprint, 2023, arXiv: 2310.06770.
- [26] Just R, Jalali D, Ernst M D. Defects4J: a database of existing faults to enable controlled testing studies for Java programs[C]//Proceedings of the 2014 International Symposium on Software Testing and Analysis. New York: ACM, 2014: 437-440.
- [27] Lin D, Koppel J, Chen A, et al. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge[C]//Proceed-

ings of the Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. New York: ACM, 2017: 55-56.

[作者简介]



丁一新 (1975-), 男, 江苏溧阳人, 国网江苏省电力有限公司高级工程师, 主要研究方向为网络运维与网络安全。



赵新建 (1987-), 男, 山东济宁人, 国网江苏省电力有限公司高级工程师, 主要研究方向为网络与信息安全。



吴子成 (1998-), 男, 江苏南京人, 国网江苏省电力有限公司助理工程师, 主要研究方向为软件安全、Web 安全 and 安全攻防技术等。



韩皓 (1982-), 男, 上海人, 博士, 南京航空航天大学教授, 主要研究方向为系统与软件安全、AI 安全等。