软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(7):2871−2885 [doi: 10.13328/j.cnki.jos.007588] [CSTR: 32375.14.jos.007588]

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

软件供应链安全中 LLM 生成代码逻辑性缺陷检测*

赵祖威[1,2], 汤恩义[1,3], 李薛成[1,3], 戴新宇[1,2], 陈 鑫[1,4], 李宣东[1,4]

[1](计算机软件新技术全国重点实验室 (南京大学), 江苏 南京 210023)

[2](南京大学 人工智能学院, 江苏 南京 210023)

[3](南京大学 软件学院, 江苏 南京 210093)

[4](南京大学 计算机学院, 江苏 南京 210023)

通信作者: 汤恩义, E-mail: eytang@nju.edu.cn

摘 要: 随着大语言模型 (large language model, LLM) 在代码生成领域的快速发展, 其生成的代码在智能化基础软件供应链中的应用日益广泛. 基础软件供应链中集成了大量基于 LLM 生成代码开发的第三方模块与组件. 然而, 由于 LLM 主要基于开源代码进行训练, 训练代码中的缺陷与安全漏洞可能会导致生成代码存在潜在错误与供应链安全问题. 为此, 学术界有针对性地提出了 EvalPlus 等测试技术, 但这些技术主要依赖基于概率的测试用例生成机制, 难以实现对供应链关键路径的全面覆盖, 导致深层次逻辑性缺陷难以被有效发现. 为了解决上述问题, 提出一种融合符号执行的供应链 LLM 生成代码的缺陷检测方法. 该方法通过符号执行挂载机制自动识别 LLM 生成代码的输入参数并进行适配和符号挂载, 制导符号执行引擎对程序的关键路径进行精确的约束分析, 生成高效的边界测试用例, 从而发现现有方法难以检测到的深层逻辑性程序缺陷. 在现有主流基准数据集上, 对 LMSYS Chatbot Arena 中排名前 11 的主流 LLM 进行了实验评估. 实验结果表明, 该方法能够更有效地检测出 LLM 生成代码中的逻辑性缺陷, 使代码的平均测试通过率降低了 3.99%–18.98%, 平均测试覆盖率提高了 3.31%–8.19%, 有效提升了 LLM 生成代码的正确性和智能化基础软件供应链的安全性.

关键词: 软件缺陷检测; 供应链安全; 大语言模型; 代码生成; 符号执行

中图法分类号: TP311

中文引用格式: 赵祖威, 汤恩义, 李薛成, 戴新宇, 陈鑫, 李宣东. 软件供应链安全中LLM生成代码逻辑性缺陷检测. 软件学报, 2026, 37(7): 2871–2885. http://www.jos.org.cn/1000-9825/7588.htm

英文引用格式: Zhao ZW, Tang EY, Li XC, Dai XY, Chen X, Li XD. Logical Defect Detection for LLM-generated Code in Software Supply Chain Security. Ruan Jian Xue Bao/Journal of Software, 2026, 37(7): 2871–2885 (in Chinese). http://www.jos.org.cn/1000-9825/7588.htm

Logical Defect Detection for LLM-generated Code in Software Supply Chain Security

ZHAO Zu-Wei[1,2], TANG En-Yi[1,3], LI Xue-Cheng[1,3], DAI Xin-Yu[1,2], CHEN Xin[1,4], LI Xuan-Dong[1,4]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](School of Artificial Intelligence, Nanjing University, Nanjing 210023, China)

[3](Software Institute, Nanjing University, Nanjing 210093, China)

[4](School of Computer Science, Nanjing University, Nanjing 210023, China)

Abstract: As the large language models (LLMs) rapidly advance in code generation, their generated code gains increasingly widespread applications in intelligent foundational software supply chains. The foundational software supply chains integrate a large number of third-

* 基金项目: 国家自然科学基金 (62172210, 62172211)
本文由“智能化基础软件可信构造和供应链安全”专题特约编辑王林章教授、司徒凌云研究员、钟浩研究员、 向剑文教授、张路教授推荐.
收稿时间: 2025-09-08; 修改时间: 2025-10-20; 采用时间: 2025-12-08; jos 在线出版时间: 2025-12-26
CNKI 网络首发时间: 2026-06-02

party modules and components developed by employing LLM-generated code. However, since LLMs are primarily trained based on open-source code, defects and security vulnerabilities in the training code may cause potential errors in the generated code and security problems in the software supply chain. To this end, targeted testing techniques such as EvalPlus have been proposed, but it is difficult for these techniques to achieve the full coverage of critical paths in the supply chain due to their reliance on probability-based test case generation, which makes it hard to uncover deep-seated logical software defects. To solve the above-mentioned problems, this study proposes a defect detection method for LLM-generated code in software supply chains that integrates symbolic execution. This method employs a symbolic execution mounting mechanism to automatically identify input parameters in LLM-generated code and perform adaptation and symbolic mounting. It then guides the symbolic execution engine to conduct precise constraint analysis on the program's critical paths and generate efficient boundary test cases, thus detecting deep-seated logical software defects that the existing methods struggle to detect. This study conducts experimental evaluation on the top 11 mainstream LLMs from the LMSYS Chatbot Arena by adopting existing mainstream benchmark datasets. Experimental results show that the proposed method can more effectively detect logical defects in LLM-generated code, reducing the average test pass rate by 3.99% to 18.98% and increasing the average test coverage by 3.31% to 8.19%. Finally, the correctness of LLM-generated code and the security of intelligent foundational software supply chains are effectively improved.

Key words: software defect detection; supply chain security; large language model (LLM); code generation; symbolic execution

近年来, 随着大语言模型 (large language model, LLM) 在代码生成领域的快速发展[1−6], 以 Codex[7]和 CodeGen[8]等为代表的 LLM 已被广泛应用于智能化基础软件供应链中软件开发、测试、运维等各个环节, 以及系统软件、人工智能框架软件、嵌入式领域软件等各个层次智能化基础软件的构造过程中. 研究表明, 约 40.5% 的软件开发团队在项目中使用 LLM 生成的代码来构建模块与组件, 部分项目中由 LLM 生成的代码比例甚至超过 75%[9]. 目前, 软件供应链中大量第三方模块与组件均基于 LLM 生成的代码进行开发, 显著提升了基础软件的开发效率[10,11].

然而, 由于 LLM 主要基于大规模开源代码进行训练, 而已有研究表明, 约 84% 的开源项目中存在缺陷与安全漏洞[12]. 因此基于这些开源项目训练出来的 LLM 可能会无意中生成带有缺陷的代码, 导致软件供应链中许多第三方模块与组件频繁出现功能性错误与安全问题[13], 进一步加剧了供应链的代码安全风险. 已有研究指出, 在大模型生成的程序中, 约 40% 的程序存在安全漏洞[10], 对软件供应链安全构成严重威胁. 因此, 如何有效检测 LLM 生成代码中的潜在缺陷, 并提升基础软件供应链的安全性, 已成为当前亟需解决的关键技术难题.

为了提升对 LLM 生成代码的缺陷检测能力, 保障智能化基础软件供应链的安全性, 学术界提出了多种基于测试用例的缺陷检测方法. HumanEval[7]通过结合 LLM 与传统软件测试方法, 构建了一个用于评估 LLM 生成代码功能正确性的测试基准. 在此基础上, HumanEval-X[14]提出了一种面向多编程语言场景的代码生成评估基准, 以更全面地评测 LLM 在多语言环境下的代码生成能力, 从而更有效地发现生成代码中的潜在缺陷. EvalPlus[15]则通过结合 ChatGPT 与模糊测试 (fuzz testing) 方法, 自动批量生成测试用例, 进一步提升了对生成代码的缺陷检测能力. 然而, 现有方法大多依赖概率机制生成测试用例, 缺乏系统性的逻辑推理过程, 尤其在输入参数范围较窄或路径条件复杂的情况下存在局限性, 难以全面覆盖 LLM 生成程序的关键路径, 从而导致深层次程序缺陷难以被有效发现, 难以满足智能化基础软件对高可靠性和高安全性的要求.

为解决上述问题, 本文提出一种融合符号执行 (symbolic execution) 的 LLM 生成代码缺陷检测方法, 旨在系统提升基础软件供应链中 LLM 生成代码的安全性. 该方法通过符号执行挂载算法, 自动识别并适配 LLM 生成代码的输入参数, 实现符号变量的自动挂载. 随后, 利用符号执行技术[16−19], 系统推导生成程序的路径约束, 生成更具针对性的边界测试用例. 具体而言, 本文方法首先从 LLM 生成的功能代码中提取路径约束条件, 并借助可满足性模理论 (satisfiability modulo theory, SMT) 求解器进行求解, 进而生成能覆盖更多关键路径的边界测试用例, 从而发现现有方法难以检测到的深层次程序缺陷, 提升 LLM 生成的软件供应链各环节代码的正确性.

以图 1 中的函数 log_message 为例, 该函数是由 LLM 生成的一段程序代码, 用于在软件运行过程中记录不同类型的日志信息. 日志系统作为基础软件供应链中的关键组成部分, 对于软件运行状态的监控与问题定位具有重要意义. 一个设计合理的日志系统不仅能够帮助开发人员及时掌握软件的运行状况, 还能通过日志信息快速识别并定位潜在程序缺陷, 从而提高系统的可维护性与安全性. 在图 1 中, log_message 函数可能在软件运行期间被多

次调用, 并根据不同的用户消息内容决定记录普通日志还是错误日志. 当用户消息不等于错误码时, 系统将该消息视为普通信息并记录 (第 3、4 行). 反之, 当用户消息等于错误码时, 程序会进一步判断其内容以确定是否记录错误日志: 若用户消息的第 2 个字符 (下标为 1) 为“E”, 表明该消息为“程序错误”信息, 此时程序会记录错误日志, 并注明该错误是由程序错误导致的 (第 6、7 行). 若用户消息的第 3 个 (下标为 2) 与第 4 个 (下标为 3) 字符的 ASCII 码之和等于 65, 则表示系统在运行过程中已累计产生 65 条警告信息, 此时程序也会记录错误日志, 并注明该错误是由“过多警告”导致的 (第 8、9 行). 按照设计意图, 在每次日志记录时, 程序应当仅记录一种错误原因, 即在“程序错误”与“过多警告”这两种情形中选择其一: 若存在程序错误, 则优先记录“程序错误”作为错误原因; 仅在不存在程序错误的前提下, 且警告计数达到特定阈值时, 记录“过多警告”作为错误原因. 然而, 经分析发现, LLM 生成的该函数未能正确实现上述业务规则, 导致程序在特定输入条件下可能同时记录两种错误原因, 从而造成错误日志信息混乱, 增加了软件开发人员识别和定位软件漏洞的难度, 最终影响缺陷的高效修复, 降低基础软件的可靠性. 因此, 本文关注的核心问题在于: 如何有效检测软件供应链 LLM 生成代码中此类仅在特定输入条件下才会触发的深层逻辑性程序缺陷, 从而提升生成代码的正确性, 保障基于 LLM 构造的智能化基础软件的可靠性与安全性?

```
1.  void log_message(char* user_message,
2.                   char* error_code) {
3.    if (strcmp(user_message, error_code) != 0)
4.      log_info(user_message);
5.    else {
6.      if (user_message[1] == 'E')
7.        log_error("Program error");
8.      if (user_message[2] + user_message[3] == 65)
9.        log_error("Too many warnings");
10.   }
11.   return;
12. }
```

图 1　LLM 生成的用于日志记录的程序片段

由于上述缺陷仅在满足特定输入条件时才会被触发, 传统模糊测试方法在生成满足这些特定条件的测试用例时面临巨大挑战. 以图 1 为例, 输入字符串 user_message 必须首先等于字符串 error_code, 且同时满足 user_message[1] == 'E'和 user_message[2] + user_message[3] == 65 这两个条件, 才能触发该程序错误. 而随机测试或模糊测试方法生成满足上述复杂条件的测试用例的概率极低. 此外, 直接使用 LLM 推理出能够触发该错误的测试用例也存在较大困难, 因为当前主流 LLM 在处理此类深度逻辑推理任务时存在明显不足.

符号执行技术可以有效弥补 LLM 在深度逻辑推理能力方面的不足, 它能够从程序中提取严格的约束条件, 并通过求解这些约束条件生成触发程序问题的特定输入. 例如, 在本例中, SMT 求解器可以通过求解符号执行约束, 生成测试输入 user_message == error_code == "AE !", 从而准确触发目标程序缺陷, 有效解决传统缺陷检测方法难以生成满足特定输入条件测试输入的问题.

本文提出一种软件供应链安全中融合符号执行的 LLM 生成代码缺陷检测方法, 该方法充分利用符号执行技术, 提升测试的覆盖率 (coverage) 和有效性, 从而能够检测出更多软件供应链中 LLM 生成代码的缺陷. 我们基于多个主流测试基准中的 166 个代码生成任务, 对 LMSYS Chatbot Arena 中排名前 11 的 LLM 进行了实验评估. 实验结果表明, 本文方法在缺陷检测效果方面显著优于传统方法, 能更有效地发现 LLM 生成代码中的功能性和逻辑性缺陷, 进而提升代码的正确性. 具体而言, 相较于现有方法, 本文方法使代码的平均测试通过率 (pass rate) 降低了 3.99%–18.98%, 平均测试覆盖率提升了 3.31%–8.19%.

本文的主要贡献如下.

(1) 验证了符号执行技术可有效弥补现有 LLM 代码生成缺陷检测方法在深度逻辑推理方面的不足, 并提出将其集成至基础软件供应链的 LLM 生成代码缺陷检测框架中, 以提升缺陷检测能力.

(2) 设计并实现了一种软件供应链安全中融合符号执行的 LLM 生成代码缺陷检测方法, 该方法能够自动识

别 LLM 生成代码中的输入参数并进行符号挂载, 从而制导符号执行生成针对性更强的边界测试用例, 解决传统随机测试或模糊测试方法在处理复杂逻辑路径时覆盖率不足的问题.

(3) 在 LMSYS Chatbot Arena 中排名前 11 的主流 LLM 上进行了实验评估, 实验结果表明本文方法可以有效提升 LLM 生成代码的正确性, 增强基础软件供应链的安全性.

1 相关工作

1.1 LLM 代码生成

近年来, 利用 LLM 进行代码生成以帮助开发者进行智能化基础软件构造并提升编程效率已经变得愈加普遍[20−22]. 基于大规模开源代码数据集, 许多企业与研究机构利用 Transformer 架构训练出包含数百亿甚至上千亿参数的 LLM. OpenAI 推出的 Codex 模型是该领域的先行者, 其使用 GPT 架构预训练了一个专为代码生成而设计的大模型. 随后, 其他预训练模型也取得了显著进展, 包括 OpenAI 的 GPT-4[23]、Anthropic 的 Claude 3 系列模型[24]、Meta 的 Code Llama[25]、Google 的 Gemini 系列模型[26,27]、DeepSeek 的 DeepSeek-V2 系列模型[28]以及清华大学的 GLM 系列模型[29,30]. 这些 LLM 展现了通过自然语言描述生成代码的卓越能力, 极大地辅助了基础软件构造工作, 提升了软件开发人员的开发效率.

1.2 LLM 代码生成缺陷检测框架

为了检测 LLM 生成代码中的缺陷, 提升生成代码的正确性, 研究人员提出了多个基准测试框架. HumanEval[7]是评估 LLM 代码生成能力的开创性框架, 它使用 164 个手动创建的 Python 测试用例及其标准实现, 对大模型生成的代码进行缺陷检测. 在 HumanEval 的基础上, EvalPlus[15]通过 ChatGPT 生成种子测试输入, 并对其进行变异 (mutation) 操作以生成更多测试用例, 从而对 LLM 生成的代码进行更为严格的缺陷检测. 此外, HumanEval-X[14]、Spider[31]和 MultiPL-E[32]等框架则将测试用例扩展到多种编程语言, 以便对 LLM 进行更广泛的跨语言评测. 不同于这些框架, 我们将符号执行技术[16−19]引入对 LLM 生成代码的缺陷检测中, 实现了更全面和严格的评估. 此外, 我们的方法不依赖于传统方法中所需的程序标准实现, 而是可以完全根据自然语言描述和函数模板生成程序测试输入与预期输出, 显著提升了缺陷检测的可扩展性与通用性.

1.3 自动化测试用例生成

自动化测试用例生成技术已广泛应用于软件缺陷检测等领域. 传统的黑盒测试方法, 如模糊测试[33,34], 不考虑被测程序的内部实现, 导致生成的测试用例具有较高的随机性. 与之相反, 白盒测试方法则通过分析程序源代码生成更高质量的测试用例. 符号执行[16,35]作为白盒测试的代表性技术之一, 目前已被广泛应用于软件供应链安全中的漏洞分析与缺陷检测. 例如: 在网络服务器、文件系统与设备驱动程序等关键基础软件中, 符号执行技术被用于系统性探索程序路径, 从而识别潜在软件缺陷[18]. 此外, 最新研究表明, 符号执行在恶意软件分类、数值缺陷检测等供应链安全场景中也具有良好的检测性能[36−38]. 这些研究充分体现了符号执行在提升软件供应链的安全性与可靠性方面发挥了重要作用[39−41]. 本文方法利用符号执行技术在执行过程中为每条路径收集路径约束, 通过求解路径约束, 生成针对深度执行路径的测试用例, 提升了对 LLM 生成代码的测试覆盖率和缺陷检测能力.

2 软件供应链安全中的 LLM 生成代码逻辑性缺陷检测方法

为了提升软件供应链中 LLM 生成代码的正确性与安全性, 本文提出一种融合符号执行的 LLM 生成代码缺陷检测方法. 该方法引入符号执行挂载机制, 使符号执行能够自动适配 LLM 生成代码的输入参数, 从而生成针对性更强的边界测试用例. 此外, 方法还结合 EvalPlus 生成的测试输入与 GPT 生成的程序预期输出, 实现对生成代码的高覆盖率测试和缺陷检测, 有效提升基础软件供应链中 LLM 生成代码的功能正确性与安全性. 该方法综合利用现有 LLM 代码生成缺陷检测框架与符号执行技术, 增强对复杂程序路径的覆盖能力. 图 2 展示了该方法的整体流程, 主要包括 4 个阶段: LLM 代码生成、EvalPlus 测试输入生成、基于符号执行的过程、缺陷检测.

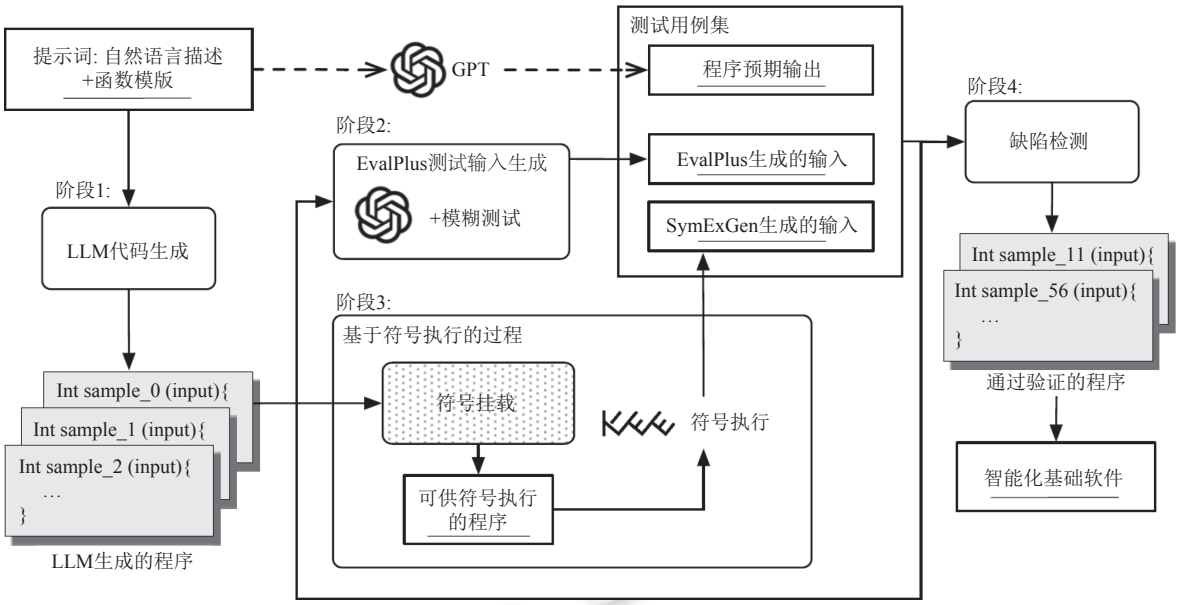

图 2　软件供应链安全中的 LLM 生成代码逻辑性缺陷检测方法

在阶段 1, LLM 会根据用户提供的自然语言描述和函数模板生成代码, 以构建软件供应链中的不同组件和模块. 此阶段输出的代码为后续测试流程的输入. 之后, 方法进入阶段 2, 使用 EvalPlus 方法生成大量测试用例输入. 接下来, 流程并行化进入阶段 3, 即基于符号执行的过程, 这是本文方法的核心部分. 在此阶段, 我们设计并实现了一个符号执行挂载流程, 使符号执行引擎能够自动适配 LLM 生成的程序, 收集程序中的符号约束. 这些约束条件由符号执行引擎中的 SMT 求解器进行求解, 生成符号执行测试用例输入. 最终, 阶段 2 和阶段 3 生成的所有测试输入, 与由 GPT 模型推理得到的程序预期输出一同进入阶段 4, 对 LLM 生成的程序进行全面的缺陷检测.

2.1　阶段 1: LLM 代码生成

当用户使用本文框架检测 LLM 生成的程序时, 首先需要由 LLM 生成代码. 在该阶段, LLM 根据用户提供的提示词生成具体实现代码, 这些代码可被进一步用于构造软件供应链中的各类基础软件, 例如: 日志系统、机器学习训练框架和推理引擎等. 在本文方法中, 为了保证 LLM 代码生成的全面性, 我们为所有 LLM 设置了统一的代码生成提示词模板, 如图 3 所示, 使其更准确地描述所有必要的代码逻辑.

```
请根据注释中的任务描述, 用适当的C++代码补全省略号部分.
/*
你的任务是编写一个函数, 当整数x是n的幂时返回true, 否则返回false.
当存在某个整数k, 使得n的k次幂等于x, 即n^k = x时,称x是n的幂.
例如:
is_simple_power(1, 4) => true
is_simple_power(2, 2) => true
is_simple_power(8, 2) => true
is_simple_power(3, 2) => false
is_simple_power(3, 1) => false
is_simple_power(5, 3) => false
*/

#include <stdio.h>
#include <math.h>
using namespace std;
bool is_simple_power (int x, int n) { ... }
```

图 3　LLM 代码生成提示词模板

按照该模板, 每个提示词被划分为 3 个部分: 任务说明、需求注释以及代码接口. 任务说明在模板中固定为: “请根据注释中的任务描述, 用适当的 C++代码补全省略号部分.” 这段文本明确了我们期望 LLM 如何完成代码生成任务. 需求注释部分以 C++注释的形式, 提供了函数的目标、功能需求、逻辑提示以及输入输出示例, 用以确

保 LLM 能充分理解生成代码的语义. 代码接口部分则提供了所生成代码的外围结构信息, 包括函数名称、参数列表以及依赖库等. 这种提示词组织方式使 LLM 能专注于生成核心逻辑代码, 保证所生成的代码能够被直接编译. 生成的程序代码将在后续检测流程中作为待测对象, 进入 EvalPlus 测试输入生成、符号执行与缺陷检测等阶段, 以系统评估其正确性与可靠性.

2.2 阶段 2: EvalPlus 测试输入生成

在完成了 LLM 代码生成后, 框架进入阶段 2, 该阶段集成了 EvalPlus 中的基于 LLM 以及基于变异的测试输入生成技术, 旨在生成大规模且随机性较高的测试输入, 以保证测试的广度和多样性. 首先, EvalPlus 会利用 ChatGPT 根据用户提示词生成种子输入 I_{seed}, 然后使用模糊测试中的变异方法生成大量额外测试输入 I_{fuzz}, 最终产生该阶段的测试输入集, 如公式 (1) 所示:

$$I_{\text{EvalPlus}} = I_{\text{seed}} \cup I_{\text{fuzz}} \tag{1}$$

2.3 阶段 3: 基于符号执行的过程

● 为 LLM 生成的程序挂载符号. 阶段 3 与阶段 2 并行运行, 重点通过基于符号执行的技术对关键测试路径进行分析, 并利用约束求解生成高效的边界测试用例, 从而捕获在随机输入下难以触发的深层逻辑性程序缺陷. 与阶段 2 中 EvalPlus 生成的随机性较高、数量庞大的测试输入不同, 阶段 3 生成的测试输入具有更强的针对性与精确性, 能够有效发现仅在特定条件下触发的深层逻辑缺陷. 在阶段 3, 我们首先为每个 LLM 生成的程序挂载符号, 使程序满足符号执行的要求, 从而可以制导符号执行引擎对程序的关键路径进行精确的约束分析. 符号执行挂载过程主要用于验证并调整由 LLM 生成的程序, 确保其格式符合后续符号执行的要求. 算法 1 展示了该过程的具体步骤. 首先, 系统会判断 LLM 生成的程序 P 是否可编译 (第 2 行). 对于不可编译的程序 (如不完整的函数或无意义的字符串), 将直接忽略 (第 3 行), 不再进行后续处理. 对于可编译的程序, 本文方法会识别其输入参数的数量、类型及名称 (第 7 行). 随后, 根据这些参数, 应用符号挂载模板 (图 4) 对程序 P 进行适配并挂载符号, 从而生成一个新版本程序 P' (第 9 行), 该程序可被符号执行引擎成功编译与执行 (第 11、12 行).

算法 1. 符号执行挂载.

输入: LLM 生成的程序 P;
输出: 可供符号执行的程序 P'.

```
1.  //步骤 1: 验证程序是否可编译
2.  if Compilable(P) = False then
3.      Discard(P)
4.      return nil
5.  end if
6.  //步骤 2: 识别输入参数
7.  {(p_i, t_i) | i ∈ 1,…,n } ← ExtractParameters(P)
8.  //步骤 3: 挂载符号执行
9.  P′ ← Harness(P, {(p_i, t_i)})
10. //步骤 4: 验证程序 P′
11. if Compilable(P′) and Executable(P′, KLEE) then
12.     return P′
13. else
14.     return nil
15. end if
```

```
1.  #include "klee/klee.h"
2.  int main() {
3.      char user_message[SIZE];
4.      char error_code[SIZE];
5.      klee_make_symbolic(user_message, sizeof user_message, "user_message");
6.      klee_make_symbolic(error_code, sizeof error_code, "error_code");
7.      klee_assume(user_message[SIZE–1]=='\0');
8.      klee_assume(error_code[SIZE–1]=='\0');
9.      //调用待测函数
10.     log_message(user_message, error_code);
11. }
```

图 4　符号挂载模板

本文中, 我们使用 KLEE[17,19]作为符号执行引擎. KLEE 是当前最具代表性的符号执行工具之一, 能够遍历程序所有可能执行路径并自动生成高覆盖率的测试用例. 凭借其在约束收集、路径搜索及测试用例生成等方面的高效实现, KLEE 已被广泛应用于软件缺陷检测、漏洞分析以及程序测试等研究领域. 本文方法基于预定义模板为待测程序进行符号的挂载. 图 4 展示了一个符号挂载模板, 其中定义了待测程序输入的格式 (第 3 行和第 4 行), 推导符号约束所需的符号声明 (第 5 行和第 6 行), 以及这些约束的前置条件 (第 7 行和第 8 行), 同时还包含了供符号执行引擎使用的其他适配信息 (第 1 行). 最终, 挂载了符号的程序将作为输入传递给符号执行引擎 KLEE, 以便推导路径约束, 并生成相应测试用例.

● 通过符号执行生成测试输入. 符号执行是一种严格的、基于推导的程序分析技术, 用于生成满足路径约束的测试输入. 一旦符号被挂载, 待测程序就可以被当作 KLEE 引擎的输入进行分析. KLEE 中的符号执行过程包括两个主要部分: 符号约束收集和 SMT 约束求解. 首先, 符号约束收集模块会根据符号执行策略系统遍历程序的每条执行路径 (例如, 第 j 条路径为 $path_j$). 它会收集每条路径上遇到的分支条件, 对判定为 False 的条件进行取反, 并将判定为 True 的条件合并进合取范式 (conjunctive normal form, CNF) 中. 这个过程中会传播常量和变量, 以构造约束条件. 随后, SMT 求解器模块会使用可满足性模理论来求解这些约束, 并且生成当前路径对应的测试输入. SMT 是对布尔可满足性问题 (SAT) 的拓展, 其目标是在特定理论 (如整数算术、实数算术、数组、位向量等) 下判断一阶逻辑公式是否可满足. 目前主流的 SMT 求解器 (比如 Z3、STP 等) 支持多种理论的组合求解, 并在约束简化、冲突分析与分支搜索等方面进行了高度优化, 从而显著提升了约束求解的效率. 在本文中, 对于路径 $path_j$ 的约束集 $constraints(path_j)$, 使用 SMT 求解器求解得到的对应的测试输入 in_j 可表示为:

$$SMTSolve(constraints(path_j)) = in_j \tag{2}$$

需要注意的是, 并非所有的条件约束都是可解的. 阶段 3 所生成的测试输入集 I_{SymExGen} 包含了这个过程中生成的所有可解输入. 总体而言, 符号执行能够系统且高效地生成一组边界测试用例, 尽可能覆盖程序的关键执行路径, 从而显著提升测试覆盖率和缺陷检测能力. 在实际应用中, 符号执行可能因路径爆炸或非线性约束求解失败而受到限制. 对于此类情况, 本文方法将直接采用阶段 2 中 EvalPlus 生成的测试用例继续进行缺陷检测. 通过结合符号执行与 EvalPlus, 本文方法在保证较高覆盖率的同时, 有效增强了缺陷检测的鲁棒性与通用性.

2.4　阶段 4: 缺陷检测

在阶段 4, 我们使用阶段 2 和阶段 3 生成的测试用例, 检测 LLM 生成程序中的缺陷. 本文方法的测试输入集 I 由两个部分组成: I_{EvalPlus}、I_{SymExGen}, 最终形成完整的测试输入集 I:

$$I = I_{\text{EvalPlus}} \cup I_{\text{SymExGen}} \tag{3}$$

对于缺乏标准实现的待测程序, 我们的测试框架会利用 GPT 生成待测程序的预期输出. 通过将阶段 1 中的自然语言描述 $desc$ 和相应的待测程序输入 $in_i \in I$ 作为提示词提供给 GPT, 然后由 GPT 给出该输入对应的预期输出. 由于这一过程并不依赖于待测程序或其约束条件, 因此产生的输出结果会呈现出异质性. 图 5 展示了我们使用 GPT 生成图 1 的预期输出时的提示词示例. 在方法的实际运行中, 程序功能描述与测试用例输入将被替换为实际值.

请根据下列对程序的功能描述, 以及测试用例输入, 为程序生成预期输出.

程序功能描述:

该函数名为log_message, 包含两个字符串参数: char*user_message以及char*error_code, 用于在软件运行过程中记录不同类型的日志信息. 当用户消息 (user_message) 不等于错误码 (error_code) 时, 系统将该消息视为普通信息并记录. 当用户消息等于错误码时, 程序会进一步判断其内容以确定是否记录错误日志: 若用户消息的第2个字符 (下标为1) 为 'E',表明该消息为程序错误 (Program error) 信息, 此时程序会记录错误日志, 并注明该错误是由程序错误导致的.若用户消息的第3个与第4个字符的ASCII码之和等于65, 则表示系统在运行过程中已累计产生65条警告信息, 此时程序也会记录错误日志, 并注明该错误是由过多警告 (Too many warnings) 导致的. 在每次日志记录时, 程序应当仅记录一种错误原因, 即在 “程序错误” 与 “过多警告” 这两种情形中选择其一: 若存在程序错误,则优先记录 “程序错误” 作为错误原因; 仅在不存在程序错误的前提下, 且警告计数达到特定阈值时, 记录 “过多警告” 作为错误原因.

测试用例输入:
user_message="ABCD", error_code="ABCE"
user_message="ABCD", error_code="ABCD"
user_message="AECD", error_code="AECD"
user_message="AE D", error_code="AE D"
user_message="AB !", error_code="AB !"
user_message="AE !", error_code="AE !"

程序预期输出:

图 5 GPT 生成图 1 预期输出的提示词示例

为了提高生成的预期输出的准确性, 我们使用 GPT 多次生成输出, 并选择出现频率最高的输出作为预期输出:

$$GPTCompute(desc, in_i) = out_{i1}, out_{i2}, \ldots \tag{4}$$

$$exp\_out_i = max\_freq(out_{i1}, out_{i2}, \ldots) \tag{5}$$

GPT 通过图 5 中的提示词, 为图 1 的 6 条测试用例输入生成的预期输出分别为: (1) Info: ABCD; (2) 无输出; (3) Error: Program error; (4) Error: Program error; (5) Error: Too many warnings; (6) Error: Program error.

接下来, 我们将 GPT 生成的预期输出提取出来, 每个测试输入 in_i 及其对应的预期输出 exp_out_i 都会被组合成一个测试用例对 (in_i, exp_out_i), 最终所有测试用例对会形成一个完整的测试用例集 T:

$$T = \{(in_1, exp\_out_1), (in_2, exp\_out_2), \ldots\} \tag{6}$$

从缺陷检测的角度来看, 如果程序 P 可以在所有测试用例对上都通过测试 (如公式 (7) 所示), 我们就认为它通过了缺陷检测. 通过检测的程序片段可被用于构造各类智能化基础软件. 本文方法使用公式 (7) 对 LLM 生成的程序进行功能性测试, 检测生成代码中的缺陷, 并根据公式 $p = c/n$ 计算程序集的测试通过率 (其中, c 为通过检测的程序数量, n 为 LLM 生成的程序总数):

$$\forall (in, exp\_out) \in T,\ P(in) = exp\_out \tag{7}$$

2.5 方法复杂性分析

我们对方法的时间复杂度与空间复杂度进行分析. 设 LLM 生成的程序为 P, 其函数参数数量为 n, 程序 P 中包含 k 个依赖于符号输入的条件分支, 变异方法生成的测试用例数量为 m.

- 时间复杂度. 本文方法的时间复杂度可表示为:

$$T = \mathrm{O}(m \cdot C_{\text{mut\_seed}} + 2^k \cdot C_{\text{sym}} + (m + 2^k) \cdot C_{\text{run\_test}}) \tag{8}$$

其中, $C_{\text{mut\_seed}}$ 表示阶段 2 通过变异生成单个测试输入的时间开销. C_{sym} 表示阶段 3 符号执行过程中收集并求解一组路径约束所需的时间开销, 越复杂的约束求解时间越长. 2^k 表示在 k 个条件分支下符号执行可能探索的路径数上界. $C_{\text{run\_test}}$ 表示阶段 4 执行单个测试用例的时间开销.

- 空间复杂度. 本文方法的空间复杂度可表示为:

$$S = \mathrm{O}(m \cdot S_{\text{mut\_seed}} + 2^k \cdot S_{\text{sym}} + (m + 2^k) \cdot S_{\text{run\_test}}) \tag{9}$$

其中, $S_{\text{mut\_seed}}$ 表示阶段 2 通过变异生成单个测试输入的空间开销. S_{sym} 表示阶段 3 符号执行过程中收集路径约束并进行求解所需的空间开销. $S_{\text{run\_test}}$ 表示阶段 4 执行单个测试用例的空间开销.

公式 (8)、(9) 中, 第 1 项对应阶段 2 生成测试输入的时间与空间开销, 第 2 项对应阶段 3 符号执行在最坏情况下搜索路径并求解路径约束生成测试用例的开销, 第 3 项对应阶段 4 执行所有测试用例的开销. 其中, 第 1 项和第 3 项的开销相对稳定, 主要由测试用例数量决定, 因此其对整体复杂度的影响有限. 在第 2 项中, 尽管符号执行路径数在最坏情况下呈指数级增长, 但由于目前在软件供应链项目中, LLM 生成的程序通常规模较小, 复杂度有限, 条件分支数 k 一般不超过 10, 因此符号执行所产生的开销仍可控制在合理范围内, 从而保证了本文方法的可行性与实用性.

3 实验分析

在本文中, 我们设计并实现了一个软件供应链安全中的 LLM 生成代码缺陷检测方法, 并进行实验评估以回答以下研究问题.

- RQ1: 将符号执行集成进传统 LLM 代码生成缺陷检测方法中, 能否提升对生成代码的缺陷检测效果?
- RQ2: 为什么集成了符号执行的方法能更有效地检测出 LLM 生成代码中的缺陷?

3.1 实验数据与实验设置

本文使用来自 HumanEval[7]、EvalPlus[15]和 KLEE[17,19]测试集中的基准程序对我们的缺陷检测框架进行评估. HumanEval 和 EvalPlus 是此前针对 LLM 生成代码的主流缺陷检测框架, 主要基于模糊测试技术. 从这些基准中, 我们选取 152 个与符号执行引擎兼容的程序作为主要测试任务. 此外, 我们还从符号执行引擎 KLEE 中选取了 14 个测试程序, 作为补充的代码生成测试任务. 本文中的所有实验均在 Ubuntu 22.04 平台上进行, 该平台配备了主频为 2.60 GHz 的 Intel Xeon Gold 6240 CPU 处理器和 64 GB 物理内存. 实验中 KLEE 符号执行引擎的运行时间上限设定为 20 min, 其余参数均保持默认配置.

表 1 展示了不同方法生成的测试用例分布情况. HumanEval 中的测试用例均来自基准测试集中. 对于 14 个补充的测试任务, 我们基于与原始 HumanEval 测试用例相同的分布生成了一个替代测试集, 以尽量减少偏差. 在该替代测试集中, 测试用例一半为随机生成, 另一半则由 GPT-4-Turbo 根据程序直接生成. EvalPlus 的测试用例则是使用 EvalPlus 测试方法生成.

表 1 不同缺陷检测方法中的测试用例数量分布

方法	平均值	中间值	最小值	最大值
HumanEval	8.9	7.0	1	105
EvalPlus	756.6	977.5	12	1 100
SymExGen	78.9	66.5	1	150

SymExGen 的测试用例是采用本文提出的方法生成的, 包含由 SMT 求解器成功求解生成的测试用例. 如表 1 所示, EvalPlus 生成了大量的测试用例. 相比之下, 本文基于符号执行的方法生成的测试用例数量较少. 这表明, 本文方法能够以更低的测试成本生成更准确的测试用例.

表 2 给出了在 166 个测试任务中, 本文方法在缺陷检测过程中的 CPU 时间与内存开销情况. 实验结果显示, 本文方法在绝大多数任务中均能在可接受的时间和内存开销范围内完成缺陷检测, 因此本文方法在真实软件供应链项目中具有较高的可行性与实用性.

表 2 单任务 CPU 时间与内存开销统计

开销类型	平均值	中间值	最小值	最大值
CPU时间 (s)	451.7	206.9	5.1	1 200.0
内存 (MB)	288.2	268.1	220.7	442.7

3.2 RQ1: 将符号执行集成进传统 LLM 代码生成缺陷检测方法中, 能否提升对生成代码的缺陷检测效果?

我们部署了 11 个 LLM 来评估我们的缺陷检测框架. 这些模型来自 LMSYS Chatbot Arena, 这是一个用于评

估 LLM 能力的重要平台, 提供了最新的 LLM 能力评估数据. 我们于 2024 年 8 月 1 日从该平台获取了模型能力排名, 并根据表现选取了排名前 11 的模型, 同时遵循每个机构最多选取 3 个模型的标准.

表 3 展示了针对不同的代码生成任务, 使用 3 种不同缺陷检测方法对每个 LLM 生成程序进行检测时的平均测试通过率. 其中, 模型的成功率 (success rate) 和参数规模 (size) 在一定程度上代表了不同模型的代码生成能力, 模型成功率指的是每个模型在响应指定提示时生成的可成功编译程序占所有生成程序的百分比. 参数规模指的是每个 LLM 的参数数量, 以 10 亿 (B) 为单位. 对于未正式公开参数数量的模型, 我们使用了非官方渠道的估计值, 并用“*”标注以表示其非官方性. 对于某些无法获得参数数量的私有模型, 我们用“—”表示该数据缺失.

表 3 各 LLM 生成的可编译程序的平均测试通过率

模型	成功率 (%)	参数规模 (B)	平均测试通过率 (%)		
			HumanEval	EvalPlus	SymExGen
Claude-3.5-Sonnet	97.58	—	76.54	54.32	49.38
GPT-4o	96.83	—	74.39	58.54	53.66
GPT-4-Turbo	95.97	—	71.25	55.63	51.88
DeepSeek-Coder-V2	95.90	236	77.71	58.60	54.14
DeepSeek-Chat-V2	94.40	236	61.01	46.54	40.88
Yi-Large	94.40	34	69.43	54.14	50.96
Gemini-1.5-Pro	92.74	120*	50.32	38.22	35.03
Claude-3-Opus	91.87	100*	61.94	49.03	44.52
GLM-4	90.16	9	43.42	31.58	29.61
Gemini-1.5-Flash	83.20	32*	46.90	28.28	25.52
Claude-3-Sonnet	71.31	70*	32.06	25.19	20.61

本实验中包含每个代码生成任务的标准实现程序, 作为判断测试结果是否正确的参照. 测试通过率指标表示的是在测试中完全正确的 LLM 生成程序占所有生成程序的百分比. 因此, 对于用不同方法测试的同一批 LLM 生成程序而言, 较低的通过率表明相应的测试方法检测到的错误程序数量更多, 反映出测试方法的缺陷检测能力更强. 相反, 对于不同 LLM 生成的程序来说, 较低的通过率则表明该模型的代码生成能力较弱, 正确输出的比例更低.

表 3 中的测试通过率数据显示, 随着新测试方法的逐步引入 (从 HumanEval 到 EvalPlus, 再到 SymExGen), 各模型的测试通过率依次下降. 这一趋势表明 3 种方法的缺陷检测能力逐步增强. 本文提出的 SymExGen 方法有效增强了对软件供应链中 LLM 生成代码的缺陷检测能力, 进而提升了软件的可靠性. 值得注意的是, SymExGen 方法将 HumanEval 和 EvalPlus 的平均测试通过率分别从 60.45% 和 45.46% 降低至 41.47%, 这表明 SymExGen 方法在检测 LLM 生成程序缺陷方面更加有效.

图 6 展示了所有模型在 166 个代码生成任务上的平均测试通过率. 图中较长的灰色柱子表示集成符号执行的方法在降低测试通过率方面表现更为突出. 从图中可以看出, 本文方法在多个任务场景下均能有效检测生成代码中的缺陷, 具备较强的通用性和适应性.

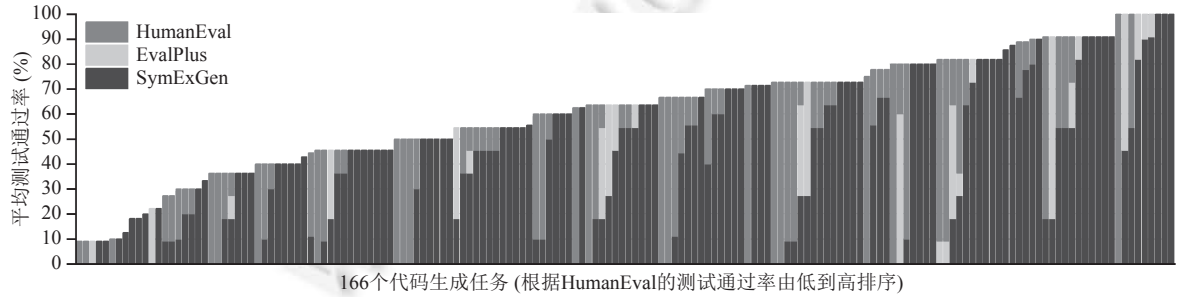

图 6 针对 LLM 在 166 个代码生成任务上生成的程序, 3 种方法的平均测试通过率

我们进一步在软件供应链场景中的 Juliet[42]和 MegaVul[43]数据集上随机选取了 20 个程序进行了实验评估. 这两个数据集在软件供应链安全研究中被广泛应用, 其中包含大量源自真实开源项目的典型缺陷与安全漏洞. 表 4 展示了在这 20 个测试任务上, 不同 LLM 生成程序的平均测试通过率. 实验结果表明, 在供应链数据集上 SymExGen 方法将 HumanEval 和 EvalPlus 方法的平均测试通过率分别从 75.69% 和 64.68% 进一步降低至 47.91%, 这一结果验证了本文方法在真实软件供应链项目中具有有效性.

表 4　各 LLM 在供应链数据集上生成程序的平均测试通过率 (%)

模型	HumanEval	EvalPlus	SymExGen
Claude-3.5-Sonnet	93.22	74.58	57.63
GPT-4o	93.22	79.66	66.10
GPT-4-Turbo	84.75	81.36	62.71
DeepSeek-Coder-V2	83.05	61.02	42.37
DeepSeek-Chat-V2	62.71	57.63	33.90
Yi-Large	64.41	52.54	47.46
Gemini-1.5-Pro	81.36	74.58	59.32
Claude-3-Opus	85.19	75.93	38.89
GLM-4	66.10	57.63	47.46
Gemini-1.5-Flash	61.02	47.46	33.90
Claude-3-Sonnet	57.63	49.15	37.29

对于未通过缺陷检测的程序, 我们进一步对其中的缺陷类型进行了分析, 实验结果如表 5 所示. 从表 5 中可以看出, 本文提出的 SymExGen 方法能够更有效地检测出 LLM 生成代码中的程序崩溃 (crash)、结果不一致 (incorrect output) 和运行超时 (timeout) 等缺陷, 相较于 EvalPlus 方法, SymExGen 方法检测出的缺陷总数提升了 9.61%. 在这些缺陷中, 结果不一致缺陷占比最高, 约占总数的 75%, 这体现了 LLM 生成程序在功能正确性方面的缺陷较为突出, 而程序崩溃缺陷和运行超时缺陷则分别占比 17% 和 8%.

表 5　未通过缺陷检测程序的缺陷类型分析

方法	检测出缺陷程序总数	缺陷数量		
		程序崩溃	结果不一致	运行超时
HumanEval	660	153	635	35
EvalPlus	917	185	862	77
SymExGen	985	209	925	98

3.3　RQ2: 为什么集成了符号执行的方法能更有效地检测出 LLM 生成代码中的缺陷?

为了进一步分析为什么 SymExGen 方法能更加有效地检测出 LLM 生成程序中的缺陷, 提升软件供应链的安全性, 本文对代码覆盖率 (code coverage) 这一指标进行了分析. 该指标定义为不同缺陷检测方法所执行的代码数占代码总数的百分比. 表 6 展示了针对不同的代码生成任务, 使用 3 种不同的缺陷检测方法对每个 LLM 生成程序进行测试时的平均代码覆盖率.

从表 6 中的数据可以看出, 缺陷检测能力更强的测试方法 (EvalPlus 和 SymExGen) 确实实现了更高的代码覆盖率. 其中, EvalPlus 方法为每个代码生成任务平均生成了 756.6 个测试用例, 而 SymExGen 方法仅生成了 78.9 个. 尽管如此, SymExGen 方法的整体平均测试覆盖率达到了 88.07%, 显著高于 HumanEval 方法的 79.88% 和 EvalPlus 方法的 84.76%.

图 7 展示了所有模型在 166 个代码生成任务上的平均代码覆盖率. 图中较长的黑色柱子表示集成符号执行的方法在提升代码覆盖率方面表现更为突出. 从图中可以看出, 本文方法在其中 164 个任务上均提升了测试覆盖率. 这一覆盖率的显著提升, 使得更多关键路径和边界情况能够被有效执行, 从而增加了缺陷暴露的可能性. 这一结果也进一步验证了集成符号执行的方法能够生成更多具有针对性的边界测试用例, 从而有效提升测试覆盖率, 并增

强对深层逻辑性程序缺陷的检测能力. 因此, 测试覆盖率的提升促进了缺陷检测能力的增强. 本文方法在提高代码缺陷检测能力方面表现出有效性与通用性, 为提升 LLM 生成程序的可靠性和软件供应链的安全性提供了有力支撑.

表 6 各 LLM 生成的可编译程序的平均代码覆盖率 (%)

模型	HumanEval	EvalPlus	SymExGen
Claude-3.5-Sonnet	76.52	84.16	88.35
GPT-4o	72.30	76.67	79.74
GPT-4-Turbo	82.83	87.78	91.43
DeepSeek-Coder-V2	85.76	89.18	91.70
DeepSeek-Chat-V2	84.76	88.62	91.29
Yi-Large	90.96	92.88	95.19
Gemini-1.5-Pro	73.93	78.80	81.72
Claude-3-Opus	74.10	80.17	84.25
GLM-4	80.46	85.52	89.14
Gemini-1.5-Flash	74.72	81.97	86.44
Claude-3-Sonnet	82.30	86.64	89.50

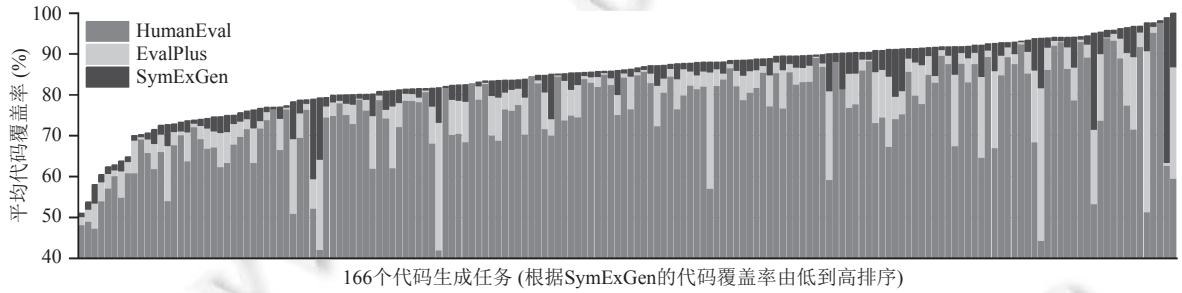

图 7 针对 LLM 在 166 个代码生成任务上生成的程序, 3 种方法的平均代码覆盖率

4 讨 论

本节讨论本文方法的可行性与适用性. 本文方法在检测软件供应链中 LLM 生成代码的缺陷方面具有较好的效果. 这主要得益于当前软件供应链开发中, LLM 生成的代码规模相对较小, 且函数的输入参数与输出返回值信息较为明确, 从而便于符号挂载适配器在符号执行过程中自动识别输入参数并进行符号化处理. 然而, 将本文方法直接应用于传统软件测试仍存在若干挑战: (1) 路径爆炸: 传统软件测试中的程序结构往往更加复杂, 当函数参数数量较多且路径嵌套较深时, 符号执行可能产生指数级增长的路径, 导致计算与内存开销急剧增加; (2) 符号约束过长: 复杂程序的路径约束可能十分冗长, 从而显著增加了约束求解难度和求解时间; (3) 自动化测试难度高: 对于复杂程序, 通常需要人工指定符号变量并设计符号执行策略, 这增加了人工干预成本并降低了测试效率. 上述问题在一定程度上限制了本文方法在传统软件测试场景中的可行性. 相比之下, 本文方法针对软件供应链场景中 LLM 生成代码进行了专门优化, 符号挂载适配器能够自动识别并符号化程序参数, 进行符号推导, 实现缺陷检测过程的高度自动化, 从而有效降低人工干预成本, 提高缺陷检测效率.

此外, 对于缺乏标准实现的待测程序, 本文方法采用 GPT 生成的程序预期输出作为测试用例的标准输出. 为验证该设计的合理性, 我们通过人工核对的方式对 GPT 生成的预期输出进行评估. 实验结果显示, 在测试任务中, GPT 基于程序功能描述与测试输入生成的预期输出的准确率 (*Accuracy*) 达到 92.8%. 这一结果表明, GPT 在生成预期输出方面具有较高的正确性与可靠性, 将 GPT 输出作为标准输出构造完整测试用例具有可行性. 即便在少数情况下 GPT 生成的预期输出存在偏差, 导致部分待测程序被误判为存在缺陷, 该类程序也不会被用于供应链软件

开发中, 仅会增加开发人员在缺陷验证与修复阶段对缺陷真实性进行人工确认的时间. 换言之, 本文方法仅会引入约 7.2% 的误报, 而该误报所带来的额外确认时间仍处于可接受范围内, 不会对软件供应链的可靠性与安全性造成实质影响.

5 总 结

本文提出一种融合符号执行的 LLM 生成代码缺陷检测方法, 用于检测基础软件供应链中 LLM 生成代码的功能正确性, 保障基于 LLM 构造的智能化基础软件的安全性. 该方法利用符号执行从程序中提取精确条件作为符号约束, 并结合 SMT 求解器对这些约束进行求解, 有效弥补了传统基于模糊测试的方法在深度逻辑推理方面的不足. 因此, 本文方法能够生成补充性的边界测试用例, 有效覆盖供应链中生成代码的关键路径, 进而提升对 LLM 生成代码的缺陷检测能力. 我们在 LMSYS Chatbot Arena 中排名前 11 的主流 LLM 上进行了实验评估, 实验结果表明, 与现有方法相比, 本文方法在测试通过率方面降低了 3.99%–18.98%, 在代码覆盖率方面提升了 3.31%–8.19%, 这些结果体现了本文方法在软件供应链缺陷检测方面的有效性.

References

[1] 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.

[2] Liu YH, Ott M, Goyal N, Du JF, Joshi M, Chen DQ, Levy O, Lewis M, Zettlemoyer L, Stoyanov V. RoBERTa: A robustly optimized BERT pretraining approach. arXiv:1907.11692, 2019.

[3] 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.

[4] 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. Journal of Machine Learning Research, 2020, 21(1): 140.

[5] Gulwani S, Polozov O, Singh R. Program synthesis. Foundations and Trends in Programming Languages, 2017, 4(1–2): 1–119. [doi: 10.1561/2500000010]

[6] 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.

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

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

[9] Rasnayaka S, Wang GL, Shariffdeen R, Iyer GN. An empirical study on usage and perceptions of LLMs in a software engineering project. In: Proc. of the 1st Int'l Workshop on Large Language Models for Code. Lisbon: ACM, 2024. 111–118. [doi: 10.1145/3643795.3648379]

[10] Williams L, Benedetti G, Hamer S, Paramitha R, Rahman I, Tamanna M, Tystahl G, Zahan N, Morrison P, Acar Y, Cukier M, Kästner C, Kapravelos A, Wermke D, Enck W. Research directions in software supply chain security. ACM Trans. on Software Engineering and Methodology, 2025, 34(5): 146. [doi: 10.1145/3714464]

[11] Feng ZY, Guo DY, Tang DY, Duan N, Feng XC, Gong M, Shou LJ, Qin B, Liu T, Jiang DX, Zhou M. CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. ACL, 2020. 1536–1547. [doi: 10.18653/v1/2020.findings-emnlp.139]

[12] Imtiaz N, Thorn S, Williams L. A comparative study of vulnerability reporting by software composition analysis tools. In: Proc. of the 15th ACM/IEEE Int'l Symp. on Empirical Software Engineering and Measurement. Bari: ACM, 2021. 5. [doi: 10.1145/3475716.3475769]

[13] Liu JQ, Tian X, Shu YQ, Zhu XX, Liu YL, Liu QX. A review of security research in the development stage of software supply chain enhanced by large models. Journal of Cyber Security, 2024, 9(5): 87–109 (in Chinese with English abstract). [doi: 10.19363/J.cnki.cn10-1380/tn.2024.09.11]

[14] Zheng QK, Xia X, Zou X, Dong YX, Wang S, Xue YF, Shen L, Wang ZH, Wang AD, Li Y, Su T, Yang ZL, Tang J. CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In: Proc. of the 29th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining. Long Beach: ACM, 2023. 5673–5684. [doi: 10.1145/3580305.3599790]

[15] Liu JW, Xia CS, Wang YY, Zhang LM. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In: Proc. of the 37th Int'l Conf. on Neural Information Processing Systems. New Orleans: Curran Associates Inc., 2023. 943.

[16] King JC. Symbolic execution and program testing. Communications of the ACM, 1976, 19(7): 385–394. [doi: 10.1145/360248.360252]

[17] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation. San Diego: USENIX Association, 2008. 209–224.

[18] Cadar C, Sen K. Symbolic execution for software testing: Three decades later. Communications of the ACM, 2013, 56(2): 82–90. [doi: 10.1145/2408776.2408795]

[19] Cadar C, Nowack M. KLEE symbolic execution engine in 2019. Int'l Journal on Software Tools for Technology Transfer, 2021, 23(6): 867–870. [doi: 10.1007/s10009-020-00570-3]

[20] Iyer S, Konstas I, Cheung A, Zettlemoyer L. Mapping language to code in programmatic context. In: Proc. of the 2018 Conf. on Empirical Methods in Natural Language Processing. Brussels: ACL, 2018. 1643–1652. [doi: 10.18653/v1/D18-1192]

[21] Li YJ, Choi D, Chung J, *et al*. Competition-level code generation with AlphaCode. Science, 2022, 378(6624): 1092–1097. [doi: 10.1126/science.abq1158]

[22] Zhao WX, Zhou K, Li JY, Tang TY, Wang XL, Hou YP, Min YQ, Zhang BC, Zhang JJ, Dong ZC, Du YF, Yang C, Chen YS, Chen ZP, Jiang JH, Ren RY, Li YF, Tang XY, Liu ZK, Liu PY, Nie JY, Wen JR. A survey of large language models. arXiv:2303.18223, 2023.

[23] OpenAI, Achiam J, Adler S, *et al*. GPT-4 technical report. arXiv:2303.08774, 2023.

[24] Anthropic. The Claude 3 model family: Opus, Sonnet, Haiku. 2024. https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf

[25] Rozière B, Gehring J, Gloeckle F, *et al*. Code Llama: Open foundation models for code. arXiv:2308.12950, 2023.

[26] Team G, Anil R, Borgeaud S, *et al*. Gemini: A family of highly capable multimodal models. arXiv:2312.11805, 2023.

[27] Team G, Georgiev P, Lei VI, *et al*. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv: 2403.05530, 2024.

[28] DeepSeek-AI, Zhu QH, Guo DY, *et al*. DeepSeek-Coder-V2: Breaking the barrier of closed-source models in code intelligence. arXiv:2406.11931, 2024.

[29] Du ZX, Qian YJ, Liu X, Ding M, Qiu JZ, Yang ZL, Tang J. GLM: General language model pretraining with autoregressive blank infilling. In: Proc. of the 60th Annual Meeting of the Association for Computational Linguistics. Dublin: ACL, 2022. 320–335. [doi: 10.18653/v1/2022.acl-long.26]

[30] Zeng AH, Liu X, Du ZX, Wang ZH, Lai HY, Ding M, Yang ZY, Xu YF, Zheng WD, Xia X, Tam W, Ma ZX, Xue YF, Zhai JD, Chen WG, Liu ZY, Zhang P, Dong YX, Tang J. GLM-130B: An open bilingual pre-trained model. arXiv:2210.02414, 2022.

[31] Yu T, Zhang R, Yang K, Yasunaga M, Wang DX, Li ZF, Ma J, Li I, Yao QN, Roman S, Zhang ZL, Radev D. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In: Proc. of the 2018 Conf. on Empirical Methods in Natural Language Processing. Brussels: ACL, 2018. 3911–3921. [doi: 10.18653/v1/D18-1425]

[32] Cassano F, Gouwar J, Nguyen D, Nguyen S, Phipps-Costin L, Pinckney D, Yee MH, Zi YT, Anderson CJ, Feldman MQ, Guha A, Greenberg M, Jangda A. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. IEEE Trans. on Software Engineering, 2023, 49(7): 3675–3691. [doi: 10.1109/TSE.2023.3267446]

[33] Miller BP, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. Communications of the ACM, 1990, 33(12): 32–44. [doi: 10.1145/96267.96279]

[34] Holler C, Herzig K, Zeller A. Fuzzing with code fragments. In: Proc. of the 21st USENIX Conf. on Security Symp. Bellevue: USENIX Association, 2012. 38.

[35] Cadar C, Godefroid P, Khurshid S, Păsăreanu CS, Sen K, Tillmann N, Visser W. Symbolic execution for software testing in practice: Preliminary assessment. In: Proc. of the 33rd Int'l Conf. on Software Engineering. Honolulu: ACM, 2011. 1066–1071. [doi: 10.1145/1985793.1985995]

[36] Bailey J, Nicholas C. Symbolic execution in practice: A survey of applications in vulnerability, malware, firmware, and protocol analysis. arXiv:2508.06643, 2025.

[37] Vouvoutsis V, Casino F, Patsakis C. Beyond the sandbox: Leveraging symbolic execution for evasive malware classification. Computers & Security, 2025, 149: 104193. [doi: 10.1016/j.cose.2024.104193]

[38] Chen JC, Shao ZZ, Yang S, Shen YM, Wang YL, Chen T, Shan ZY, Zheng ZB. NumScout: Unveiling numerical defects in smart contracts using LLM-pruning symbolic execution. IEEE Trans. on Software Engineering, 2025, 51(5): 1538–1553. [doi: 10.1109/TSE.2025.3555622]

[39] Ji SL, Wang QY, Chen AY, Zhao BB, Ye T, Zhang XH, Wu JZ, Li Y, Yin JW, Wu YJ. Survey on open-source software supply chain security. Ruan Jian Xue Bao/Journal of Software, 2023, 34(3): 1330–1364 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6717.htm [doi: 10.13328/j.cnki.jos.006717]

[40] Luo CH, Meng W, Wang S. Strengthening supply chain security with fine-grained safe patch identification. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 89. [doi: 10.1145/3597503.3639104]

[41] Sun Z, Quan ZH, Yu SR, Zhang L, Mao DM. A knowledge-driven framework for software supply chain security analysis. In: Proc. of the 8th Int'l Conf. on Control Engineering and Artificial Intelligence. Shanghai: ACM, 2024. 267–272. [doi: 10.1145/3640824.3640866]

[42] Amankwah R, Chen JF, Amponsah AA, Kudjo PK, Ocran V, Anang CO. Fast bug detection algorithm for identifying potential vulnerabilities in Juliet test cases. In: Proc. of the 8th IEEE Int'l Conf. on Smart City and Informatization. Guangzhou: IEEE, 2020. 89–94. [doi: 10.1109/iSCI50694.2020.00021]

[43] Ni C, Shen LY, Yang XH, Zhu Y, Wang SH. MegaVul: A C/C++ vulnerability dataset with comprehensive code representations. In: Proc. of the 21st Int'l Conf. on Mining Software Repositories. Lisbon: ACM, 2024. 738–742. [doi: 10.1145/3643991.3644886]

附中文参考文献

[13] 刘井强, 田星, 舒钰淇, 朱小溪, 刘玉岭, 刘奇旭. 大模型赋能软件供应链开发环节安全研究综述. 信息安全学报, 2024, 9(5): 87–109. [doi: 10.19363/J.cnki.cn10-1380/tn.2024.09.11]

[39] 纪守领, 王琴应, 陈安莹, 赵彬彬, 叶童, 张旭鸿, 吴敬征, 李昀, 尹建伟, 武延军. 开源软件供应链安全研究综述. 软件学报, 2023, 34(3): 1330–1364. http://www.jos.org.cn/1000-9825/6717.htm [doi: 10.13328/j.cnki.jos.006717]

作者简介

赵祖威, 博士生, 主要研究领域为智能化软件工程, 程序分析, 软件测试.

汤恩义, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为智能化软件工程, 新型程序分析, 软件测试.

李薛成, 硕士生, 主要研究领域为智能化软件工程.

戴新宇, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为自然语言处理, 大语言模型.

陈鑫, 博士, 副教授, CCF 专业会员, 主要研究领域为形式化方法, 可信深度学习, 软件工程.

李宣东, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为软件工程, 系统软件, 可信软件, 形式化方法.