面向API交互的Python语义化模糊测试方案

李政浩[1,2,3],闫新成[4,5],王继刚[4],贾相堃[1,2,3],苏璞睿[1,2,3*]
(1. 中国科学院大学,北京 100049;2. 中国科学院软件研究所,北京 100190;
3. 基础软件与系统重点实验室(中国科学院),北京 100049;4. 中兴通讯股份有限公司,广东深圳 518057;
5. 东南大学网络空间安全学院,江苏南京 211189)

摘 要： Python第三方库在现代软件生态中被广泛应用,其面临的安全威胁也变得日益严重,如Python包索引平台(Python Package Index,PyPI)中漏洞数量持续快速上升、依赖网络高度复杂,导致单一漏洞易通过依赖链波及大量下游项目。现有模糊测试工具在测试Python第三方库时,复杂的应用程序编程接口(Application Programming Interface,API)交互场景下接口间的隐式数据流关系会导致探索能力不足;Python语言自身的动态特性,例如鸭子类型及反射机制进一步降低了静态分析的准确性,导致对多重条件保护的复杂约束探索能力低下,使得漏洞发现能力受限。此外,传统的模糊测试引擎往往采用随机变异的策略,因而无法针对测试目标进行定制化变异,使得测试资源大量消耗在无状态依赖的低价值浅层路径上,难以探索深层代码漏洞。为此,本文提出PyBoros——一种针对Python第三方库的高效模糊测试框架。该框架通过构建包级API依赖图,并结合团渗流方法(Clique Percolation Method,CPM)进行子图分割,以精准捕获隐式依赖;在此基础上,使用大语言模型生成语义丰富的初始模糊测试驱动以捕捉隐式依赖;采用停滞触发式非阻塞动态分析,即以覆盖率增长停滞作为信号,按需捕获运行时代码状态与变量快照等约束上下文信息,并利用大语言模型进行智能约束推理以产生突破性种子;在模糊测试的过程中,通过引入API n-gram覆盖指导与分支价值评分相结合的资源调度策略,引导测试资源优先向高价值路径探索。我们在平均Github Star数超过2 000的10个真实Python第三方库中进行了测试,PyBoros发现20个真实漏洞(其中10个为0-day),漏洞检出数量较Atheris提高100%;边覆盖率较Atheris提高8.57%;初始生成的模糊测试驱动的API覆盖数达到Fuzz4All的1.8倍;对四种大语言模型(包含两种开源、两种闭源)生成的模糊测试驱动平均接受率为72.6%;在额外的扰动实验中,模糊测试驱动的接受率即使在10%静态分析扰动攻击场景也仅下降了3.8%,仍保持较高鲁棒性。总体而言,PyBoros模糊测试框架为Python第三方库的安全分析提供了一种高效实用的方法。

关键词： Python安全;Python第三方库;模糊测试;LLM;程序分析;漏洞挖掘
基金项目： 国家自然科学基金(No.62472414,No.62232016)
中图分类号： TN915.08;TP311.5 **文献标识码：** A **文章编号：** 0372-2112(2026)03-1280-16
电子学报URL: http://www.ejournal.org.cn **DOI:** 10.12263/DZXB.20250614

Python Semantic Fuzzing Solution for API Interaction

LI Zhenghao[1,2,3], YAN Xincheng[4,5], WANG Jigang[4], JIA Xiangkun[1,2,3], SU Purui[1,2,3*]
(1. *University of Chinese Academy of Sciences*, *Beijing* 100049, *China*;
2. *Institute of Software*, *Chinese Academy of Sciences*, *Beijing* 100190, *China*;
3. *Key Laboratory of System Software* (*Chinese Academy of Sciences*), *Beijing* 100049, *China*;
4. *ZTE Corporation*, *Shenzhen*, *Guangdong* 518057, *China*;
5. *School of Cyber Science and Engineering*, *Southeast University*, *Nanjing*, *Jiangsu* 211189, *China*)

Abstract: Third-party Python libraries are extensively utilized in modern software ecosystems, and the security threats they face have become increasingly severe. As the number of vulnerabilities in Python package index (PyPI) continues to surge and dependency networks grow highly complex, a single vulnerability can easily propagate through dependency chains and affect a large number of downstream projects. When testing third-party Python libraries, existing fuzzing tools suffer from insufficient exploration capability due to the implicit data flow relationships between interfaces under complex application programming interface (API) interaction scenarios. Meanwhile, Python's inherent dynamic features—such as duck typing and reflection—further reduce the accuracy of static analysis. This results in weak exploration of complex constraints protected by multiple conditional checks and ultimately limits vulnerability discovery capabilities. Traditional fuzzing engines often adopt random mutation strategies, making them incapable of performing targeted customized muta-

收稿日期:2025-07-10;录用日期:2026-03-16;责任编辑:覃怀银
*通信作者:苏璞睿

tions. As a result, testing resources are heavily wasted on stateless, low-value shallow paths, which severely hinders the exploration of deep code vulnerabilities. To address these challenges, this paper presents PyBoros, an efficient fuzzing framework tailored for third-party Python libraries. The framework constructs an package-level API dependency graph and employs the clique percolation method (CPM) for subgraph partitioning. On this basis, it leverages large language models to generate semantically rich initial harnesses that capture implicit dependencies; adopts a stagnation-triggered non-blocking dynamic analysis, which takes the stagnation of coverage growth as a signal to capture constraint context information such as runtime code states and variable snapshots on demand, and utilizes large language models for intelligent constraint reasoning to generate breakthrough seeds; and introduces a resource scheduling strategy that integrates API n-gram coverage guidance with branch value scoring to prioritize the exploration of high-value paths during the fuzzing process. We evaluated PyBoros on 10 real-world third-party Python libraries with an average GitHub star count exceeding 2 000. The results demonstrate that PyBoros discovered 20 real vulnerabilities (10 of which are 0-day), representing a 100% increase in vulnerability detection over Atheris. It also achievs an 8.57% improvement in edge coverage compared to Atheris. The API coverage of the initially generated harnesses reaches 1.8 × that of Fuzz4All. Across four large language models(two open-source and two closed-source), the average acceptance rate of the generated harnesses is 72.6%. In additional robustness experiments, even under 10% static-analysis perturbation attacks, the acceptance rate of the harnesses drops by only 3.8%, maintaining strong robustness. Overall, PyBoros provides an effective approach for the security analysis of third-party Python libraries.

Keywords: Python security; third-party Python libraries; fuzzing; LLM; program analysis; vulnerability mining

Foundation Item(s): National Natural Science Foundation of China (No.62472414, No.62232016)

0 引言

近年来，随着微服务架构和云原生开发的深度普及、人工智能尤其是大语言模型（Large Language Models，LLM）技术的快速发展，以及LLM高效微调、私有化部署和企业级应用需求的持续增长，Python已成为人工智能、数据科学、Web服务等领域最广泛采用的编程语言之一[1]，其庞大的第三方库生态在显著提升开发效率的同时，也带来了日益严峻的软件供应链安全威胁。第三方库依赖漏洞、恶意包注入、依赖混淆等攻击手段频发，已导致多个高影响力开源项目遭受供应链攻击[2-3]，严重影响系统可信性与软件关键基础设施安全，成为当前网络空间安全领域亟待解决的核心挑战之一。

与传统编译型语言（如C/C++）不同，Python的显著特点在于其庞大的第三方库生态（Python包索引）以及语言本身的动态特性，这给安全分析带来了独特而严峻的挑战。Alfadel等人[4]的研究显示，Python包索引平台（Python Package Index，PyPI）中698个库存在1 396个安全漏洞，且漏洞数量持续快速上升；同时，GitHub中超过一半的Python项目依赖了至少一个含已知漏洞的库。Python第三方库生态安全威胁分析之所以困难，主要源于库规模极其庞大和依赖关系高度复杂两个方面。Decan等人[5]和Kikas等人[6]的研究均指出，PyPI依赖网络在深度与广度上持续增长，拓扑结构复杂、调用链隐蔽，导致单一漏洞极易通过依赖链波及大量下游项目。美国国家漏洞数据库（National Vulnerability Database，NVD）最新数据进一步证实，Python第三方库已成为整个生态中最薄弱的环节之一[7]。

1 相关研究与挑战

1.1 基于模糊测试的漏洞挖掘

模糊测试（Fuzzing）作为当前最有效的漏洞挖掘技术之一，已在学术界与工业界得到广泛研究与应用。其核心是通过随机输入探索程序状态空间，试图触发错误状态。经典工具如Google的美利坚模糊测试器（American Fuzzy Lop，AFL）[8]及其改进版AFL++[9]以代码覆盖率作为反馈信号指导变异，大幅提升探索效率；此外，许航等人[10]提出基于分布三度的优化方法显著提高模糊测试性能，肖天等人[11]基于深度强化学习实现定向模糊测试，进一步提高了测试效率；侍言等人[12]与徐恪等人[13]的研究在内核及可信执行环境端到端模糊测试领域做出重要贡献。然而，这些经典方案依赖编译期插桩，难以直接支持如Python的动态语言。

针对Python，Google进一步于2020年推出Atheris[14]，通过import hook与字节码级插桩实现对Python代码的精确覆盖追踪。随后，PyRT-Fuzz[15]在此基础上引入SLang机制，对标准库应用程序编程接口（Application Programming Interface，API）进行自动识别与结构化输入生成，进一步提升了对Python状态空间的探索能力。

近年来，研究者开始尝试利用LLM的代码理解与生成能力改进模糊测试[16-24]，显著提升变异策略的

语义感知和路径探索效率[25-27]。Semantic-Aware Fuzzing[25]通过LLM生成针对性输入加速收敛；ChatAFL[26]从文档提取协议规范并构建状态机，实现有状态深度测试；SyzAgen[27]将LLM用于定向引导，驱动探索高价值代码区域。这些方案在语义导向和测试效率上展现出明显优势。在模糊测试驱动（Harness）自动生成方面，Fuzz4All[28]支持通过轻量配置为项目中的单个文件生成；Orion[29]进一步实现从代码分析、目标选择到Harness生成的端到端自动化方案。

1.2 研究动机与挑战

尽管现有模糊测试方案[15,25-26,28]在API驱动生成、标准库支持以及项目级Harness构造等方面取得了显著进展，但它们对Python第三方库中深层语义错误与安全漏洞的感知与挖掘能力仍显不足。这些方法大多局限于语法层面的调用序列生成或浅层覆盖率优化，难以有效捕捉API之间复杂的交互语义、运行时隐式约束以及高价值路径，导致测试资源大量消耗在低语义价值的路径上，无法高效探索深层漏洞。

这些不足体现在以下三个核心挑战。

（1）API交互语义缺失：第三方库通常涉及多步调用序列、状态依赖、资源生命周期管理以及数据流约束，现有方案普遍缺乏对这些深层语义关系的系统性建模；同时，由于Duck Typing机制会引发大量仅在运行时显现的类型与状态检查，产生静态分析难以捕获的隐式依赖。

（2）动态特性与运行时约束难以建模：Python的动态类型系统、反射机制（如getattr、setattr）、元编程等，使得程序路径高度依赖具体执行上下文，传统约束求解器和静态分析难以准确预测可行路径与合法输入。

（3）覆盖率导向的语义局限：传统以基本块、语句或分支执行计数为主的反馈机制，忽略了API交互序列的语义价值与漏洞触发潜力，即使某些路径可达，模糊测试引擎仍倾向于在浅层、无状态依赖的低价值路径上过度探索与资源消耗，而对需要特定依赖顺序才能触发的高风险深层路径关注不足，显著限制了整体漏洞发现效率与深度。

1.3 解决方案与贡献

为了应对上述三大核心挑战，本文提出PyBoros框架。该框架旨在实现Python第三方库高质量Harness的自动化生成，并将模糊测试的反馈信号从传统的单一代码覆盖率，扩展为同时融合代码覆盖信息与API交互语义信息的综合度量。PyBoros的核心设计基于一项关键观察：传统代码覆盖率主要反映分支的执行情况，而API覆盖能够更充分地挖掘库的行为，从而有效地提升模糊测试的深度，以发现更多深层语义错误及安全漏洞。通过有机结合静态分析以及LLM驱动的语义理解，加以新的覆盖引导策略，系统性地解决Harness自动生成、运行时约束满足以及高价值路径探索三大难题。

本文的主要贡献与创新点总结如下。

（1）提出了基于API依赖图的Harness构建方法。该方法首先通过静态分析提取API间的调用依赖和数据流关系，再利用LLM的代码理解与推理能力，对隐式依赖进行语义补全，从而自动生成语义合理的复杂Harness。这一创新有效克服了现有方案的两大局限：一是过度依赖LLM有限上下文长度导致生成不完整或不准确；二是仅捕捉显式调用而忽略深层数据流与状态依赖。

（2）提出了基于LLM的智能约束推理机制。该机制在模糊测试过程中动态捕获Python运行时产生的隐式约束，并利用LLM进行约束建模与输入补全。这一设计显著缓解了传统约束求解器对动态特性的不足，以及现有基于的LLM方法因高延迟导致的整体吞吐量下降问题。

（3）设计了基于n-gram的API覆盖资源分配策略。通过引入新的反馈度量，来量化不同API交互序列的语义价值，优先引导测试资源向高语义价值路径倾斜；同时结合动态时间片分配机制，自适应地将计算资源集中于高潜力、高质量的Harness执行路径。相较于现有方案，PyBoros显著提升了现有针对Python生态的模糊测试效率与漏洞发现能力。

2 研究背景

本节通过一个真实工业场景中的崩溃案例，剖析现有Python模糊测试方案在挖掘深层缺陷时面临的三大核心难题：API调用序列的语义依赖难以捕捉（图1左）、运行时动态约束难以突破（图1中）、有效路径探索方向难以精准引导（图1右）。这些难题导致模糊测试难以触及由隐式状态依赖与联合数据约束的深层错误。

现有的Python模糊测试方案在从Harness生成到路径探索的整个流程中，存在多个相互关联的典型局限。本节以此为基础，通过具体案例进一步剖析这些局限的成因与影响。

2.1 案例描述

以一个嵌入式集群日志系统中的漏洞片段为例。如图2所示，日志记录器记录每个心跳包的延迟信息，并使用正则表达式“value=(\d+)”从文件中提取延迟值。设备定时调用_calculate_average函数计算过去一段时间的平均延迟，并在计算前去除一个最高值和一个最低值。

在极少数情况下，若过去一段时间仅记录到两个

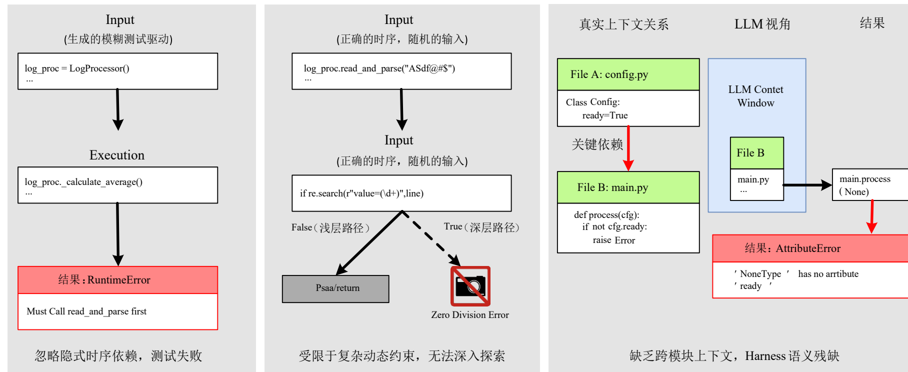


图1　现有Python模糊测试方案在深层漏洞挖掘过程中面临的三个典型环节示意图

Figure 1　Schematic of three typical stages in Python fuzzing for deep vulnerability mining

心跳包,则read_and_parse函数构建的列表仅包含两个值。此时去除最高值和最低值后,有效数据长度变为0,引发第25行除零错误,导致进程崩溃。该故障进一步导致其他设备对其心跳检测失败,并逐渐扩散,最终可能引发集群中多个设备相继停止响应。

现有模糊测试工具大多针对单一API独立建模,因此在测试时无论是单独调用get_statistic(“average”)或read_and_parse均无法触发崩溃。只有当两者按特定顺序组合且输入文件恰好满足边界条件时,才暴露深层错误。

```
 1 class LogProcessor:
 2     def __init__(self, filename):
 3         self.filename = filename
 4         self._values = None
 5
 6     def read_and_parse(self):
 7         try:
 8             with open(self.filename, "r") as f:
 9                 lines = f.readlines()
10             self._values = []
11             for line in lines:
12                 match = re.search(r"value=(\d+)", line)
13                 if match:
14                     self._values.append(int(match.group(1)))
15         except FileNotFoundError:
16             self._values = None
17             raise
18
19     def _calculate_average(self):
20         if self._values is None:
21             raise RuntimeError("Must call read_and_parse() first")
22         if len(self._values < 2):
23             raise RuntimeError("Length not enough")
24         trimmed = sorted(self._values)[1:-1]
25         return sum(trimmed) / len(trimmed)  # <- ZeroDivisionError
26
27     def _calculate_min(self):
28         if self._values is None:
29             raise RuntimeError("Must call read_and_parse() first")
30         return min(self._values)
31
32     def get_statistic(self, name):
33         method = getattr(self, f"_calculate_{name}")
34         return method()
```

图2　日志系统中的漏洞示意代码

Figure 2　Schematic code for vulnerabilities in logging systems

2.2　问题剖析

2.2.1　API组合与语义依赖盲点

模糊测试需生成有效的API调用序列,但现有方法普遍忽略隐式时序与状态依赖。以本案例为例,_calculate_average只有在read_and_parse成功执行后才有意义;若未先加载,self._values为None,直接抛浅层异常,无法进入除零路径。

Python的动态特性使这一依赖更难捕捉:get_statistic通过getattr运行时拼接方法名,静态分析难以准确提取调用链;现有的LLM方案[16-27]在分析时往往需要将整个项目代码拼接到上下文中;新的研究发现[30-32],LLM在上下文过长时会提幻觉率,从而在面对真实的项目分析时导致准确率降低。幻觉率的提高使得模型更容易忽略深层状态变化(如self._values从None到列表的转变),导致生成的序列语法正确却语义失效。现有基于静态签名的候选序列构建方式,在动态语言中准确率显著下降,难以覆盖真实项目中的复杂API交互逻辑。

2.2.2　运行时动态约束的处理难题

本案例崩溃路径受多重运行时联合约束严格保护:文件必须存在且可读、文件内容至少包含三行(保证去除极值后仍有数据)以及每行的内容需匹配特定正则格式。若任意一个条件不满足,self._values长度不足3,除零不可达。

传统模糊测试器擅长字节级变异,却难以感知外部文件依赖,更无法高效生成满足复杂正则的多行结构化内容。基于LLM的方法虽能理解约束描述,但难以实时高效指导输入生成,导致深层路径覆盖率极低。该路径由文件存在、正则匹配、列表长度三者共同约束,现有方案大多止步浅层逻辑,难以突破这类复杂约束的深层缺陷。

2.2.3　模糊测试路径探索低效

即使生成了部分包含正确序列的Harness,现有覆盖引导机制仍难以优先探索高价值路径。本案例中,高价值语义为文件恰好产生两条满足正则约束记录时的状态,但这个状态却因传统边覆盖指标对API交互序列的语义潜力缺乏量化,而被大量低价值组合

淹没，导致目标除零路径长期不可达。在真实项目中，类似的由探索方向盲目性导致的潜在安全风险尤为突出，显著降低深层漏洞的发现效率。

2.3 问题总结

现有主流模糊测试方法在此问题中表现出明显局限性：它们既难以感知代码对外部条件依赖，也缺乏同时生成多行满足复杂正则约束的结构化输入内容的能力。因此，此类技术通常只能覆盖程序的浅层逻辑，而难以深入到达那些由特定控制流路径的深层错误。

3 PyBoros框架设计

针对第2节中揭示的问题。本节提出基于API依赖感知的Python第三方库中漏洞挖掘的全新解决框架PyBoros。

3.1 总体架构

PyBoros的整体架构可以分为三个阶段来设计，如图3所示。

第一阶段聚焦于API依赖感知的Harness生成。该阶段通过静态分析构建库的API依赖图，并结合LLM的代码生成能力，生成初始Harness以应对传统模糊测试工具在处理复杂API协议时的语义鸿沟，从而提升测试用例的有效性。第二阶段实现LLM驱动的智能约束推理。当模糊测试引擎因复杂程序约束而导致探索停滞时，该阶段可自动识别相关约束条件，并利用LLM生成针对性的种子，以突破路径探索瓶颈。第三阶段构建以API交互为导向的智能反馈回路。通过引入新型停滞点评分函数，优化测试资源分配，确保有限的计算资源优先投入最具潜力的探索方向，从而提升整体的模糊测试效率。

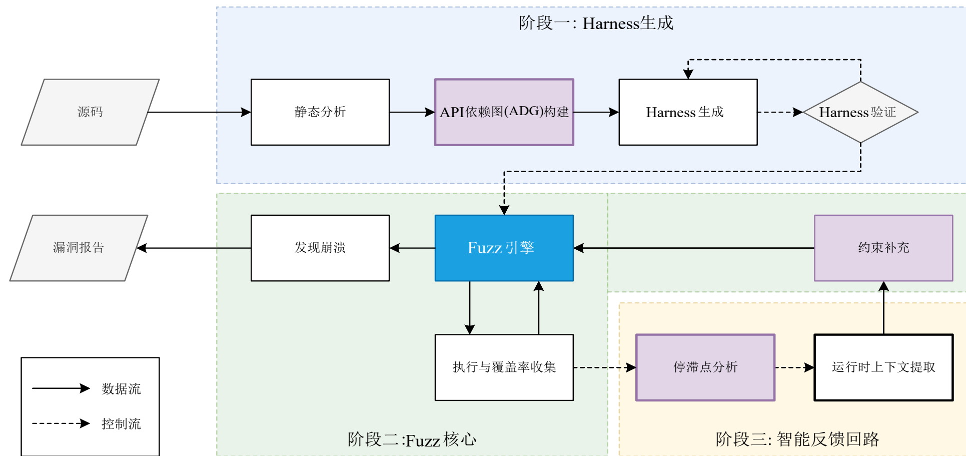


图3 PyBoros架构概览

Figure 3 PyBoros architecture overview

3.2 阶段一：API依赖感知的Harness生成

该阶段的目标是通过对库进行初步的静态分析，提取出API之间的依赖关系，并据此生成高语义关联度Harness。

3.2.1 API依赖图

API依赖图（API Dependency Graph，ADG）用来表示库API之间关系紧密程度，它能够从包含成百上千个API的复杂Python库中提取其中的依赖关系，并为后续的基于LLM的Harness生成提供高质量的上下文信息。

定义1

API依赖图。给定一个Python库L，其API依赖图是一个有向图$G_L=(V,E)$，其中，节点集合$V=V_{\text{api}}$，表示L中公开的函数或方法：

$$V_{\text{api}}=\left\{f \mid f\text{是}L\text{中公开的函数或方法}\right\} \tag{1}$$

边集合$E=E_{\text{call}}\cup E_{\text{state}}$由两类依赖关系构成。

（1）调用依赖边$E_{\text{call}}\subseteq V_{\text{api}}\times V_{\text{api}}$：表示API之间的直接调用关系，即

$$E_{\text{call}}=\left\{\left(f_1,f_2\right) \mid f_1\text{直接调用}f_2\right\} \tag{2}$$

（2）状态依赖边$E_{\text{state}}\subseteq V_{\text{api}}\times V_{\text{api}}$：表示API之间通过共享变量发生“写后读”模式的隐式数据流依赖关系，即

$$E_{\text{state}}=\left\{\left(f_1,f_2\right) \mid \exists v\in V_{\text{var}}\text{ s.t.}\left(\text{writes}\left(f_1,v\right)\wedge\text{reads}\left(f_2,v\right)\right)\right\} \tag{3}$$

其中，$\text{writes}\left(f_1,v\right)$和$\text{reads}\left(f_2,v\right)$为判定谓词，表示API函数$f$写入或读取了同一个库变量$v$。

由于Python的高度动态性，ADG在实际分析过程中几乎不可能完全精确地描述API间的依赖关系。与追求高精确性的程序依赖图（PDG）不同，ADG的目

标主要是找到一组存在潜在“关联”的API集合,因此有一定的容错能力。ADG的构建采用两阶段构建策略,首先,通过静态代码分析,识别库中所有API节点与共享变量节点,然后,通过抽象语法树(AST)遍历和数据流分析技术构建两类关键的依赖边。

ADG的构建算法如算法1所示。在节点识别阶段,通过遍历目标库的所有源文件,将所有公开的(非下划线开头的)函数和方法识别为API节点,同时将全局变量与成员变量作为共享变量节点;在ADG构建时,通过对AST遍历以及数据流依赖分析,补全调用依赖边与状态依赖边;最后,通过对共享变量节点进行节点收缩计算,得到只包含 V_{api} 节点的ADG。

算法1　API依赖图构建算法

输入: 目标 Python 库源码路径 library_path

输出: API 依赖图 $G_L=(V_{api}, E_{call}\cup E_{state})$

{阶段1:构建中间图}

$V_{api}, V_{var}, E_{call}, E_{write}, E_{read}\leftarrow\varnothing$

V_{api}, V_{var}= AST_analysis(library_path)

FOR $f\in V_{api}$; THEN

$E_{call}\leftarrow E_{call}\cup\left\{(f_1,f_2)\mid f_1\ \text{call}\ f_2\wedge f_2\in V_{api}\right\}$

$E_{write}\leftarrow E_{write}\cup\left\{(f,v)\mid f\ \text{write}\ v\wedge v\in V_{var}\right\}$

$E_{read}\leftarrow E_{read}\cup\left\{(f,v)\mid f\ \text{write}\ v\wedge v\in V_{var}\right\}$

END

{阶段2:收缩变量节点}

$E_{state}\leftarrow\varnothing$

FOR $v\in V_{var}$; THEN

$\text{writers}(v)\leftarrow\left\{f_w\mid(f_w,v)\in E_{write}\right\}$

$\text{readers}(v)\leftarrow\left\{f_r\mid(f_r,v)\in E_{read}\right\}$

$E_{state}\leftarrow E_{state}\cup\left(\text{writers}(v)\times\text{readers}(v)\right)$

END

RETURN $G_L=(V_{api}, E_{call}\cup E_{state})$

3.2.2　ADG引导的Harness生成

ADG中包含了整个库的API之间关联信息,PyBoros会对其中的子图进行分割,然后利用LLM的代码生成能力生成高质量的Harness。这一过程的核心在于将静态分析过程中得到的全局依赖关系转化为局部的、LLM能够理解的语义信息,从而确保生成的Harness的语义相关性。

为了全面且准确地从ADG中获取高关联API子集,PyBoros采用团渗流方法(Clique Percolation Method,CPM)进行子图分割与社区发现。CPM算法能够识别重叠社区,将库中的大量API分配到与其功能紧密关联的簇中,更加贴近现实的库API功能定位。

在Prompt工程设计方面,PyBoros提供了详细的角色定义、任务描述、目标API信息、相关API列表以及可能的出错样例,并明确输出要求。对于生成的Harness,还需要经过验证步骤(dry-run)才能被采纳。PyBoros会要求LLM生成至少一个可运行的输入,并通过一次动态运行确保合法性。对于验证失败的Harness,会将错误的代码与错误报告作为负样例进行反馈,并进入重试流程。

3.3　阶段二:LLM驱动的智能约束推理

模糊测试在探索深层程序逻辑时,常常受限于复杂的路径约束,尤其是涉及数据完整性验证、校验和计算以及结构化格式检查的分支条件,这类约束通常被称为Roadblock。传统依赖随机变异的模糊测试技术在遇到此类强约束时,变异效率急剧下降,测试进度容易陷入长期停滞。

本阶段的主要目标即针对上述问题提出解决方案。利用Python语言运行时的动态性和可观测性,在执行过程中实时捕获导致Roadblock的约束条件,并引入LLM进行在线、上下文感知的约束求解,从而显著提升对复杂约束的突破能力。

3.3.1　运行时约束捕获

运行时约束捕获依赖昂贵的动态分析,动态分析会严重影响模糊测试的效率,导致分析吞吐量降低。为了解决此问题,PyBoros设计了一套非阻塞的运行时分析架构(见算法2);在主Atheris模糊测试器进程内,还额外维护一个分析线程,该线程只有在检测到路径探索停滞时才触发动态分析。

算法2　运行时约束捕获算法

输入: 种子语料库 corps

输出: 约束上下文 context

```
启动子进程并开启 sys.settrace
执行计数器 exec_map={};last=ϕ;context={}
FOR 动态执行的代码行 line; THEN
    exec_map[line].append(last)
    last=line
END
sbranch = FindSingleBranch(exec_map)
FOR b in sbranch; THEN
    Breakpoint(b);
    Run(seed);
    context.append(Inspect(b))
END
RETURN context
```

动态分析线程有两个核心设计。(1)停滞信号检测:PyBoros维护一个用于监测边覆盖率变化的滑动时间窗口,当该窗口内未观测到任何边覆盖率增量时,则判定探索过程已进入停滞状态,并触发停滞信号;(2)运行时约束捕获:当停滞事件发生时,PyBoros

会使用昂贵的 sys.settrace 对语料库中的种子进行动态运行，并记录分支的跳转情况；对于所有的停滞时只有单边跳转的分支（简称“停滞分支”），将其栈帧作为运行时信息缓存，在下一次停滞分析时，仅对新未采样单边命中分支进行运行时捕获。

3.3.2 基于LLM的约束推理

PyBoros借助LLM在代码语义理解与逻辑推理上的优势，将程序路径求解的复杂问题转化为推理任务，为动态约束求解带来了全新视角。动态约束推理需全面纳入代码位置信息、约束条件以及当前状态，PyBoros通过将程序约束转换为结构化自然语言描述来实现这一目标，该描述整合了代码上下文片段及其行号以定位问题、布尔表达式及其条件以定义求解焦点，以及运行时变量快照以说明停滞时程序的状态。

由此生成的推理结果包括可绕过约束的输入字典，这些字典作为新语料补充到语料库中，成为后续种子的来源。该方法既简化了求解流程，也增强了系统的在不同复杂约束场景下的语义理解能力。

3.4 阶段三：API交互导向的智能探索策略

在实际模糊测试过程中，出于计算成本与效率的考虑，无法对所有停滞分支进行约束推理。为此，第三阶段聚焦于模糊测试中的分支选择问题，其核心目标是引导有限且昂贵的计算资源优先投入到潜在价值更高的分支。为实现这一目标，PyBoros从多维度评估分支，并基于此提出了一套新的探索策略。

3.4.1 API n-gram覆盖

随着API排列组合种类的增加，隐藏在复杂交互中的漏洞往往更容易被触发。然而，传统的基于代码覆盖率的模糊测试策略忽略了API调用序列的语义价值，为此，PyBoros引入了基于API n-gram 覆盖的指标，将API调用序列视为连续的n元组，并将测试导向从单纯的代码行覆盖转向对API组合的优先探索。

定义2

API n-gram。给定一次运行过程中产生的API调用序列 $T=\langle c_1,c_2,\cdots,c_m\rangle$，其中，$c_i$是第$i$个被调用的API。一个长度为$n$的API n-gram是从轨迹$T$中提取的、由$n$个连续API调用组成的元组：$(c_i,c_{i+1},\cdots,c_{j+n-1})$，其中，$1\le i\le m-n+1$。

定义3

API n-gram覆盖。在模糊测试的整个生命周期中，记录一个全局已观测的API n-gram集合，记为S_n，有

$$S_n=\bigcup_{T\in\tau_{\text{all}}}\left\{(c_i,c_{i+1},\cdots,c_{i+n-1})|(c_i,c_{i+1},\cdots,c_{i+n-1})\ \text{in}\ T\right\} \tag{4}$$

其中，τ_{all}表示所有已经执行过的测试用例的API轨迹集合。

PyBoros以发现新API n-gram覆盖为探索目标的关键度量，将模糊测试的推理优先级更多地分配向能够给全局集合S_n中添加新的元素的输入。与传统的代码覆盖率相比，API n-gram覆盖考虑了API之间的时序关系，具有更强的语义表达能力：如果一个测试用例产生了至少一个当前S_n中不存在的n-gram，那么它就被认为有价值的，并优先选择。

当n越大时能够考虑更多的API序列，但是分析成本也会指数增加；参考Peng等人[33]的相关研究与经验，在$n=4$时能够表现出最佳的测试效率，因此我们沿用了$n=4$（即4-gram），以平衡API序列语义以及分析效率。

3.4.2 智能探索策略

PyBoros通过智能探索策略指导模糊测试资源的分配。该策略通过量化所有停滞分支的探索价值，为约束推理目标的选择决策提供客观依据。约束推理会优先选择评分高的停滞分支，评分函数为启发式规则，其设计参考多篇经典论文的结论[15,27,33]，综合考虑API 4-gram的覆盖以及代码复杂度。对于未命中分支b，评分函数如下：

$$\text{Score}(b)=W_{\text{NG}}\cdot \text{NG}(b)+W_C\cdot\log\big(1+C(b)\big)+W_D\cdot\frac{1}{D(b)}-W_F\cdot N_F(b) \tag{5}$$

其中，NG(b)项为分支b预期产生的新4-gram数；$C(b)$项评估了分支b的代码片段复杂度，用于引导模糊测试优先探索逻辑复杂的代码，实现上使用分支b的AST节点数进行近似估计，并进行对数处理，以避免数值波动过大导致的剧烈影响；$D(b)$表示分支b的调用栈深度，PyBoros采用了广度优先搜索策略，优先确保对库API的广泛覆盖，以建立一个更多样化的停滞点语料库；惩罚项$N_F(b)$用于避免系统在困难分支上过度投入资源，更新方式如下：

$$N_F(b)\leftarrow N_F(b)+1\ \big(\text{当分支}b\text{推理失败}\big) \tag{6}$$

权重参数W_{NG}、W_C、W_D以及W_F为超参数，在默认情况下全部配置为1。为了测量不同权重参数对探索阶段的影响，我们对超参数进行了参数敏感性分析，通过以下三个关键指标评估权重的敏感性。

（1）唯一求解次数：指在测试过程中，被选为唯一约束推理目标的分支数。该指标用于衡量目标选择倾向的集中度——数值较小表示策略偏向于反复尝试少数分支，可能导致资源在困难路径上的过度投入；数值较大表示策略更分散到多个分支，有助于广泛探索但可能增加求解开销。

（2）新种子数：指通过约束推理生成并成功添加到种子语料库的新输入数量。该指标反映目标选择的效果——数值较大表示策略更有效地突破约束，产

生更多有效输入，从而提升整体探索深度和漏洞发现潜力；数值较小则可能表明策略在低价值路径上浪费资源。

（3）边覆盖数：指测试过程中覆盖的分支跳转的数量。该指标衡量路径探索的全面性——数值较大表示更好的代码覆盖，意味着策略成功揭示更多潜在漏洞路径；数值较小则可能反映探索效率低下。

敏感性验证以pyyaml样本的safe_load函数为测试入口进行测试；设置单次求解时间上限为5 min，其他参数与后续第4节的默认配置相同。依次将W_{NG}、W_C、W_D以及W_F四个参数分别独立上调至10，其他参数保持为1不变。每种权重配置均进行60 min的模糊测试。表1展示了不同权重下的参数敏感性验证结果。

表1 不同权重下safe_load的参数敏感性验证结果

Table 1 Results of parameter sensitivity validation for safe_load under different weights

指标	唯一求解次数	新种子数	边覆盖数
$W_{NG}=10$	2	15	871
$W_C=10$	4	27	903
$W_D=10$	4	33	898
$W_F=10$	10	42	1 102
默认权重	7	57	1 074

验证结果显示，当W_{NG}、W_C以及W_D过高时，唯一求解次数显著降低（如$W_{NG}=10$时仅为2），表明策略偏移向集中于少数分支，导致新种子数和边覆盖数较低（如新种子数仅15），可能反复求解困难路径造成资源浪费；相反，当W_F过高时，唯一求解次数增加10，展现出更强的发散倾向，但新种子数（42）和边覆盖数（1 102）虽高于部分配置，却不如默认权重（新种子数57，边覆盖数1 074），且求解次数多出42%，在面对复杂约束时收敛性下降。在敏感性分析中，默认权重实现了最佳平衡，能够兼顾探索能力与效率。

综上，在本文中，Score(b)的所有权重参数被设置为默认值1，即

$$\mathrm{Score}(b)=\mathrm{NG}(b)+\log\left(1+C(b)\right)+\frac{1}{D(b)}-N_F(b) \quad (7)$$

4 实验评估

为验证PyBoros在Python第三方库模糊测试中的有效性，本实验在统一软硬件环境中进行评估。硬件配置为Intel Core i9-13900K处理器、128 GB DDR4内存和1 TB NVMe SSD；软件环境采用NixOS 25.05（Linux Kernel 6.12.32）和Python 3.9.22。所有阶段的超参数均采用统一配置：路径覆盖率5 min滑动窗口停滞判定（最多3次重试）、评分函数权重均设置为1；LLM相关实验的采样温度（Temperature）配置为1、top-p为1、上下文窗口为4 096 Tokens、超时时间120 s，模型选用Claude Sonnet 4.0。

数据集方面，尽管DyPyBench[34]是当前Python动态分析的代表性基准，但直接将其应用于模糊测试场景面临严峻挑战：其多数项目严重依赖网络I/O或系统环境，导致在隔离测试环境中极易产生误报或执行失败。针对这一局限，本实验按照可测试性与场景多样性的标准进行清洗与扩充：首先剔除DyPyBench中不适合模糊测试的项目，保留5个核心样本；随后，为了提升数据集的工业代表性与场景覆盖度，额外引入了5个下载量极高（GitHub上超过2 000 Star）的样本，构建了包含10个真实项目的数据集DyPyBench-Fuzz（见表2），包括了从图像处理到数据序列化等多种复杂场景，有效弥补了原基准在模糊测试场景下的短板。

表2 DyPyBench-Fuzz数据集详情

Table 2 Details of the DyPyBench-Fuzz dataset

样本编号	样本名称	版本	#公开 API
1	ansi2html	a3a93a6	32
2	cbor2	e1b65f2	315
3	exifread	d60f18d	43
4	html2text	f49a1a7	40
5	pdoc	15.0.0	246
6	pillow	10.2.0	928
7	pypdf	5.0.1	675
8	pyyaml	957ae4d	442
9	thefuck	c7e7e1d	406
10	tomli	e2f8d2d	62

实验旨在回答以下5个研究问题（Research Question，RQ）。

RQ1：PyBoros的漏洞挖掘能力如何？

RQ2：PyBoros在Harness生成方面的表现如何？

RQ3：PyBoros对不同LLM的泛化支持程度如何？

RQ4：PyBoros相对于现有工具的路径探索能力如何？

RQ5：PyBoros在分析过程中的鲁棒性与性能开销表现如何？

4.1 漏洞挖掘能力

PyBoros基于Atheris开发，在崩溃识别上直接沿用了其基于SEGV信号的机制。其漏洞处理与上报流程包含以下步骤。

（1）收集：将所有导致崩溃的测试语料存入专用集合。

（2）去重：使用PDB工具提取崩溃堆栈信息并进行去重。

（3）验证：对去重后的崩溃样本进行人工审查，确

认其为真实未知漏洞。

(4)查重与上报:在项目原始仓库中搜索确认该崩溃未被记录后,将其上报至CVE数据库。

(5)公开:在获得CVE编号且开发者完成修复后,该漏洞信息才会被公开。

在本实验中,PyBoros在DyPyBench-Fuzz中共发现了20个真实漏洞。我们将这些漏洞分类为已公开或已知的漏洞(1-day),以及未公开或新发现的漏洞(0-day)。如表3所示,包括10个1-day漏洞和10个0-day漏洞。0-day漏洞中的7个已在相应库的最新版本中得到修复。

针对所有10个0-day漏洞,我们遵循负责任披露原则,并已向相关开发者及NVD进行上报。截至目前,已获得4个全新CVE编号(其中2个为高危级别):CVE-2025-25555、CVE-2025-62707、CVE-2025-62708和CVE-2025-64078。另外3个漏洞(涉及内存泄露、越界读取及压缩炸弹类型)已获开发者确认,剩余3个正在审核中。

表3 PyBoros在真实场景中最终发现的安全漏洞

Table 3 Security vulnerabilities finally discovered by PyBoros in real-world scenarios

类别	编号	目标库	漏洞ID
1-day	1	html2text	Issue-96
	2	html2text	Issue-119
	3	ansi2html	Issue-127
	4	pdoc	Issue-797
	5	pypdf	Issue-3295
	6	pypdf	Issue-3340
	7	pypdf	Issue-3347
	8	pillow	Issue-8924
	9	pillow	Issue-8956
	10	pillow	Issue-8986
0-day	11	ansi2html	CVE-2025-25555*
	12	pillow	Issue-9272*
	13	cbor2	CVE-2025-64076*
	14	cbor2	PR-265*
	15	pypdf	Issue-3499*
	16	pypdf	CVE-2025-62707*
	17	pypdf	CVE-2025-62708*
	18	pyyaml	Issue-895
	19	pyyaml	Issue-896
	20	pyyaml	Issue-900

注:*表示已修复。

为验证PyBoros的漏洞挖掘能力,我们将其与两个基线工具Python-AFL和Atheris进行了比较。实验以已知的20个漏洞作为真实漏洞数据集(Ground Truth),并在每个漏洞的位置手动插入触发探针,进行12 h模糊测试,来观察探针能否被触发,结果如表4所示。

表4显示,基线工具Python-AFL和Atheris分别触发了4个和10个漏洞,而PyBoros则覆盖了全部20个漏洞,比Python-AFL提升了高达400%,比Atheris提升了100%。实验结果说明了本方案在漏洞挖掘能力上的优越性。

为进一步探究PyBoros中4-gram机制与漏洞发现能力之间的关联,我们以广泛在fuzzing覆盖度量中采用的、刻画执行过程中行为多样性[33]的"唯一n-gram"数量[35]作为API调用序列丰富度的量化指标,并将真实漏洞数据集中的20个首次成功触发漏洞的harness作为执行样本开展回测实验,分析其与漏洞触发路径之间统计相关性。

表4 不同工具对真实场景漏洞的挖掘能力对比

Table 4 Comparison of real-world vulnerability discovery capabilities among different tools

编号	Python-AFL	Atheris	PyBoros
1	✔	✔	✔
2	✔	✔	✔
3	✔	✔	✔
4	×	✔	✔
5	✔	✔	✔
6	×	×	✔
7	×	✔	✔
8	×	✔	✔
9	×	×	✔
10	×	×	✔
11	×	✔	✔
12	×	×	✔
13	×	×	✔
14	×	×	✔
15	×	×	✔
16	×	×	✔
17	×	✔	✔
18	×	×	✔
19	×	✔	✔
20	×	×	✔

注:✔表示对应工具能够发现对应漏洞,×表示对应工具未能发现对应漏洞。

执行样本被划分为三组:(1)1-day漏洞触发组(对应首次触发漏洞编号1~10的Harness);(2)0-day漏洞触发组(对应首次触发漏洞编号11~20的Harness);(3)normal组(即上述的20个Harness)。对于0-day组和1-day组,每个Harness独立采样20条能够触发漏洞的执行轨迹;对于normal组,每个Harness独立采样10条正常执行轨迹,共采样600条。对于每条

轨迹，本实验采用重叠滑动窗口的方法提取API调用序列中的4-gram数量。实验结果如图4所示。

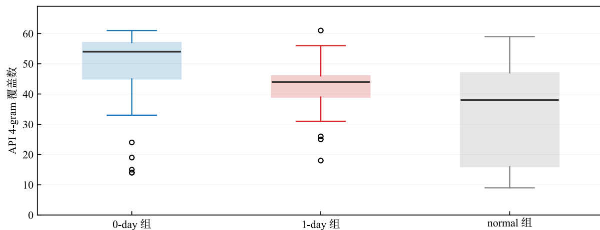


图4　真实漏洞回测实验中不同组别的API 4-gram覆盖数

Figure 4　API 4-gram coverage counts of different groups in real vulnerability backtesting experiments

统计结果表明：0-day组的唯一4-gram数均值为51.01（中位数53），1-day组均值为42.39（中位数44），normal组均值为35.07（中位数36）。相较于1-day组，0-day组提升19.11%；相较于normal组，0-day组提升44.99%。从分布形态看，0-day组整体位于更高区间。

为检验统计显著性，本文采用Mann-Whitney U单侧检验（$\alpha = 0.05$，备择假设为前者组优于后者组），并对三组两两比较采用Holm校正；结果显示0-day>1-day（$U = 31\,968.5$，$p = 1.12\times10^{-26}$）、0-day > normal（$U = 332\,861.0$，$p = 8.75\times10^{-34}$）以及1-day>normal（$U=25\,342.5$，$p = 1.87\times10^{-6}$）均达到显著水平；进一步以Vargha-Delaney $\hat{A}_{12}$评估效应量，分别得到$\hat{A}_{12}$为0.806 9、0.848 4和0.633 6，表明前两组比较为大效应、后一组比较为小效应，整体结果支持API 4-gram丰富度与漏洞触发能力之间存在显著统计相关性。

4.2　Harness生成质量评估

在基于LLM的程序分析相关研究中，分析方法的贡献归属度备受关注。为客观评估PyBoros中Harness生成方法的有效性，本节将其与两种代表性基线进行对比：（1）纯LLM生成方法（Naive LLM），即不引入任何外部结构化指导，直接依赖提示工程生成Harness；（2）Fuzz4All框架，作为当前最先进的通用LLM驱动的Harness生成工具，已在多语言场景中展现出较强的能力。

对于Naive LLM基线，我们针对每个被测库重复生成30次Harness，并统计累计覆盖的API数目、边覆盖数，以及计算平均初始接受率。提示词采用固定模板（如图5所示），包括角色定义、任务描述、目标API列表及输出格式要求。该方法模拟了LLM直接生成场景，便于隔离ADG引导的作用。

对于Fuzz4All基线，我们遵循其论文设置和开源代码的默认配置：即将目标项目中所有Markdown格式的文档文件作为独立输入，采用generate-new策略进行生成；使用GPT-4-0613模型进行自动提示，结合

```
You are a Python fuzzing expert.
Your task is to generate test code for APIs within the {} library.
You should aim to combine API sequences as complexly as possible.
Your output format should satisfy: ...
```

图5　Naive LLM方案对比过程中使用的提示词

Figure 5　Prompts for comparison with the naive LLM method

HuggingFace-StarCoder模型执行实际代码生成循环；采样参数设置为temperature = 1、top-p = 1、最大输出长度1 024 tokens。

为全面、客观地评估所生成Harness的质量，本文选用以下三个核心指标。

（1）初始API覆盖：指生成的有效Harness覆盖的公共API数量。该指标反映Harness对库API语义和依赖关系的理解深度——数值越高，表示生成的调用序列越完整、语义越相关。需要强调的是，当Harness中包含了相关的API调用即认为覆盖。

（2）初始接受率：指生成的结果中，可直接运行（无语法错误和未捕获崩溃）的Harness比例。该指标衡量生成结果的鲁棒性和实用性——高接受率意味着更少的无效样本和更高的测试启动效率。

（3）初始边覆盖：指Harness首次执行时对目标库代码的分支覆盖数量。该指标量化Harness触发的路径深度——高值表明生成的输入能有效探索库内部逻辑，而非停留在浅层。需要注意的是，由于默认计量工具中的基本块聚合机制，使得部分线性接口不会产生边覆盖计数，该指标实际为表示保守下界。

上述指标从语义理解能力、生成稳定性以及实际探索潜力三个维度，共同刻画了Harness的整体质量。

如表5所示，我们将目标库的总API数目（即#API，基于ADG节点估计）与总分支数（#Edge）作为参考基准，详细对比了Naive LLM（T1）、Fuzz4All（T2）以及PyBoros（PB）三种方案的Harness在DyPyBench-Fuzz上的生成质量。

实验结果表明，在初始API覆盖数方面，PyBoros展现出显著的优势：PyBoros生成的Harness平均覆盖约206个API，远超Naive LLM的24.1个以及Fuzz4All的95.9个。这一数据的提升（约为T1的8.5倍，T2的2.1倍）有力地证明了PyBoros通过ADG引导LLM能够有效突破单一API测试的局限，自动构建出包含复杂调用序列的高语义密度Harness。

在初始接受率方面，PyBoros表现出更强的鲁棒性。虽然Naive LLM在结构简单的库（如ansi2html）上表现出较高的接受率，但其严重受限于LLM的幻觉问题。例如在pdoc与thefuck两个库中，Naive LLM因频繁调用不存在或已弃用的API，导致生成的Harness初始接受率极低（分别为0%和17%）。相比之下，

表5 Harness生成质量对比

Table 5 Comparison of Harness generation quality

目标库	统计信息		初始API覆盖数			初始接受率/%			初始边覆盖数		
	# API	# Edge	T1	T2	PB	T1	T2	PB	T1	T2	PB
ansi2html	32	184	3	15	32	87	62	71	33	67	70
cbor2	315	1 031	26	77	270	46	71	89	67	116	127
exifread	43	575	12	34	39	67	79	86	11	19	19
html2text	40	86	35	40	40	90	80	87	30	42	46
pdoc	246	3 172	13	164	205	0	76	59	98	274	281
pillow	928	3 802	33	213	514	71	49	53	251	475	665
pypdf	675	4 834	29	199	441	32	55	62	225	397	728
pyyaml	442	3 684	49	82	344	37	73	92	112	213	277
thefuck	406	1 468	12	99	129	17	51	48	72	77	113
tomli	62	545	29	36	46	12	69	79	69	92	98
平均	318.9	1 938.1	24.1	95.9	206.0	45.9	66.5	72.6	96.8	177.2	242.4

注：T1表示Naive;T2表示Fuzz4All;PB表示本方法。

Fuzz4All虽然通过上下文优化提升了部分通过率，但在面对pdoc这种复杂库时仍不及PyBoros稳定。PyBoros通过静态分析对API存在性进行预校验，有效缓解了模型幻觉，使得在pdoc上仍能保持59%的接受率，确保了测试的顺利启动。

最后，在初始边覆盖数指标上，PyBoros取得了全面领先。边覆盖数直接反映了Harness探索代码深层逻辑的能力。数据表明，即使在接受率相近的情况下，PyBoros生成的Harness也能触发更多的程序状态。以pypdf为例，PyBoros的初始边覆盖数达到728个，分别是Naive LLM（225个）的3.2倍和Fuzz4All（397个）的1.8倍；在pillow中也呈现出类似的趋势（PB：665 vs T2：475 vs T1：251）。这充分说明PyBoros生成的Harness并非简单的API序列组合，而是构建了符合程序语义的数据流和控制流，从而在测试初期就能深入到核心代码逻辑中。

4.3 LLM泛化能力评估

为了评估PyBoros在不同LLM上的泛化能力，我们选取了4个有代表性LLM模型作为对比：分别是开源消费级的Qwen 2.5-Coder（32 B）、开源工业级的DeepSeek R1（671 B）、闭源商用模型GPT-3.5 Turbo以及默认的Claude Sonnet 4.0。表6显示了各模型的平均初始接受率Acc和平均边覆盖数Cov。

可以看出，Claude 4.0 Sonnet上综合表现最优：平均初始接受率达到72.6%，平均边覆盖数为242.4。符合模型缩放定律，即模型的参数量越大，生成的Harness质量越高。

值得注意的是，开源模型在实验中表现出极具竞争力的性能。DeepSeek R1在各项指标上均显著超越了商用闭源模型GPT-3.5 Turbo（平均边覆盖数218.9 vs. 141.5），展现了开源大模型在程序分析领域的巨大潜力。更有趣的是，即便是参数量较小的Qwen 2.5-Coder（32 B），其平均边覆盖数（190.2）也优于GPT-3.5 Turbo。这一现象表明，针对代码任务微调的模型比通用的早期LLM更适合PyBoros的应用场景。

综上，实验数据证实了PyBoros具有良好的泛化能力与鲁棒性。虽然更先进的模型能提升测试的上限，但PyBoros并不严格依赖于昂贵的闭源模型；在低成本的开源模型支持下，它依然能够保持有效的Harness生成与路径探索能力，这为PyBoros在计算资源受限环境下的实际部署提供了有力支撑。

表6 PyBoros在不同LLM上的泛化能力对照

Table 6 Comparison of PyBoros generalization ability across different LLMs

目标库	Qwen 2.5-Coder		DeepSeek R1		GPT-3.5 Turbo		Claude 4.0 Sonnet	
	Acc/%	Cov	Acc/%	Cov	Acc/%	Cov	Acc/%	Cov
ansi2html	51	54	65	60	35	37	71	70
cbor2	73	106	83	127	45	81	75	117
exifread	63	14	74	16	51	10	86	19
html2text	64	36	82	43	46	28	87	46
pdoc	47	204	55	262	34	148	59	281
pillow	38	556	47	608	28	369	53	665
pypdf	45	577	53	651	34	467	62	728
pyyaml	73	196	96	245	59	148	92	277
thefuck	37	89	41	96	29	69	48	113
tomli	61	70	73	91	39	58	79	98
平均	55.2	190.2	64.5	218.9	40.0	141.5	72.6	242.4

4.4 路径探索能力对比

作为模糊测试的核心指标之一，边覆盖数可有效

量化工具对程序状态空间的探索深度。本实验通过对比 PyBoros 与基线工具 Python-AFL 和 Atheris,从边覆盖增长视角验证 PyBoros 在动态约束处理上的优势(平均求解 103.6 次,平均消耗 4.78×10^6 Tokens,平均请求成本$79.21)。

实验选用 DyPyBench-Fuzz 作为基准,所有工具均使用4.2节中初始覆盖数最优的 Harness 及种子为起点,并各自在单个 CPU 核心上运行 12 h。为了确保公平性,PyBoros 的约束推理设计为并发运行,实验中以线程的方式绑定在同一个 CPU 核心上,确保所有工具用于测试的 CPU-Time 相同。为增强结果的稳健性和可重复性,减少随机变异策略带来的不确定性,本实验进行三次独立重复实验,并引入时间戳规避 LLM Prefix Caching 干扰。

图 6 展示了边覆盖数随时间的增长曲线,其中半透明边界为上下界,实线为平均值。

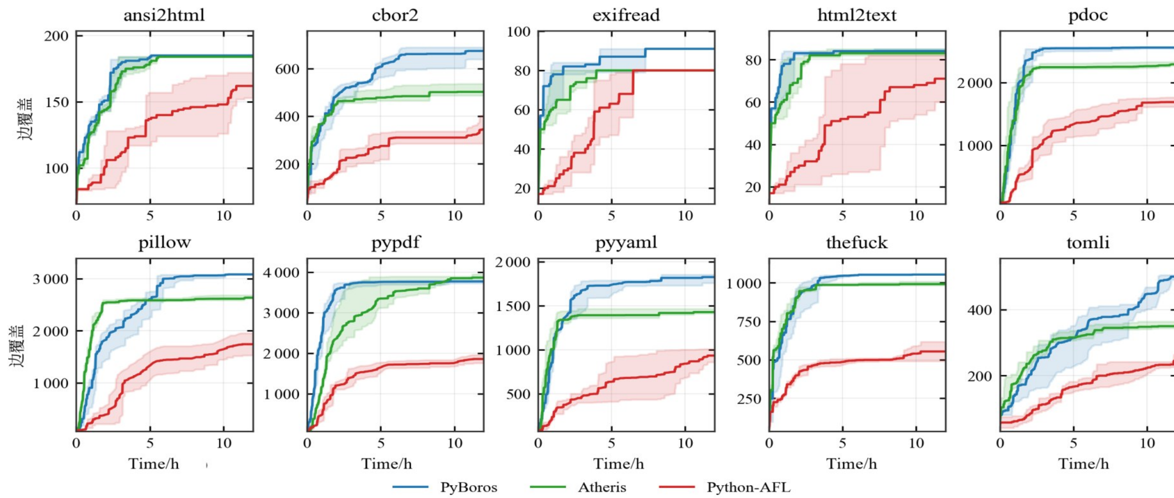


图6　边覆盖数随时间的变化曲线图

Figure 6　Curve of edge coverage over time

整体而言,Python-AFL 效率最低,受限于无语义指导的变异策略;在小规模库(如 ansi2html、html2text)中,Atheris 与 PyBoros 迅速趋于饱和;在大规模库(如 pillow、pyyaml)中,PyBoros 与 Atheris 初期相当,但 PyBoros 展现阶梯状增长,说明约束推理模块有助于探索深层路径。PyBoros 的平均覆盖数达到了 Python-AFL 的 2.13 倍,相较于 Atheris 提升了 8.57%,这证明了 PyBoros 路径探索策略的有效性。值得注意的是,尽管在 pypdf 样本上 PyBoros 的边覆盖数(3 773)略低于 Atheris(3 867),但是 PyBoros 依然发现了 3 个 0-day 漏洞,且其中两个 Atheris 未能发现;这与 TortoiseFuzz[36]的观察一致,即优秀的探索策略能够在更低的覆盖率中探索到更多的独特路径,避免资源浪费在低价值路径上。

为统计验证差异显著性,我们采用 Mann-Whitney U 单侧检验(显著性水平 $\alpha = 0.05$;原假设 H_0:对比工具之间无显著差异;备选假设 H_1:PyBoros 优于基线工具)。进一步地,采用 Vargha-Delaney $\hat{A}_{12}$ 效应量评估效应大小:

$$\hat{A}_{12} = \frac{\#\text{wins} + 0.5 \times \#\text{ties}}{m \times n} \tag{8}$$

其中,m 和 n 分别表示 PyBoros 和基线工具的 12 h 采样数;#wins 表示满足条件 $\text{Cov}_{\text{PyBoros}}(t_i) > \text{Cov}_{\text{baseline}}(t_i)$ 的样本数目;#ties 表示相等的配对数目。

表 7 汇总了在 DyPyBench-Fuzz 上的 p 值与 $\hat{A}_{12}$ 效应量:相较于基线工具 Python-AFL,本方案在全部样本上表现出 $p < 0.001$,平均 $\hat{A}_{12} = 0.92$,呈现出大效应;相较于基线工具 Atheris,所有 $p < 0.001$,其中 5 个样本中表现为大效应(平均 $\hat{A}_{12} = 0.83$)。尽管在 pypdf 上表现出 N 效应(即接近无差异),但是整体的统计学结果依然显著支持 PyBoros 的优势。

综上所述,本节实验证实 PyBoros 的语义化路径探索策略在动态语言环境中显著优于传统工具,尤其在突破深层约束与高价值路径挖掘方面表现出色,为 Python 第三方库的安全测试提供了更高效、更有针对性的解决方案。

4.5　鲁棒性与性能开销分析

为了全面评估 PyBoros 在实际应用中的综合表现,特别是量化在非理想静态分析条件下的容错能力以及动态约束推理引入的运行时开销,本节选取具有高度动态特性的 pyyaml 库(7121 Python LOC)作为微基准测试对象,进行深入的定量分析。

表7　PyBoros 和基线方案在覆盖数上的 Vargha-Delaney 效应量对比

Table 7　Vargha-Delaney effect size comparison of coverage between PyBoros and baseline schemes

Benchmark	Vs. Atheris			Vs. Python-AFL		
	p-value	$\hat{A}_{12}$	Size	p-value	$\hat{A}_{12}$	Size
ansi2html	5.5×10^{-28}	0.60	S	<0.001	0.90	L
cbor2	1.4×10^{-278}	0.81	L	<0.001	0.95	L
exifread	1.0×10^{-240}	0.78	L	<0.001	0.86	L
html2text	2.8×10^{-83}	0.67	M	<0.001	0.95	L
pdoc	<0.001	0.86	L	<0.001	0.93	L
pillow	7.6×10^{-36}	0.61	S	<0.001	0.89	L
pypdf	3.9×10^{-5}	0.53	N	<0.001	0.95	L
pyyaml	8.3×10^{-296}	0.82	L	<0.001	0.94	L
thefuck	4.3×10^{-220}	0.78	L	<0.001	0.96	L
tomli	4.0×10^{-12}	0.56	S	<0.001	0.85	L

4.5.1　ADG构建鲁棒性分析

ADG的构建依赖于静态分析的精确性。然而，由于Python的动态特性，静态分析过程中不可避免地存在误差。为验证PyBoros在非理想状态下的有效性，我们设计了基于扰动注入的敏感度分析实验。

本实验将结构扰动阈值设定为10%。这一参数的选择参考了图结构鲁棒性领域的经典结论：Zügner等人[37]指出，图对抗性攻击中仅需改动边总数的5%，即可导致大部分下游分析任务失效。基于此，本实验将结构扰动阈值设定为10%，以验证系统在超越最坏情况下的鲁棒性。

我们构建了以下两个ADG变体进行对比。

(1)ADG^{+noise}：噪声注入组，即在原始ADG中随机添加10%的虚假依赖边，用于模拟动态语言在静态分析中的过近似错误，旨在测试PyBoros在高ADG噪声场景下的语义过滤能力。

(2)ADG^{-drop}：依赖缺失组，即在原始ADG中随机移除10%的真实依赖边，用于模拟动态语言在静态分析中的欠近似错误，旨在测试PyBoros在关键上下文缺失时的语义理解与补全能力。

为消除单次实验的随机性，我们在微基准测试上独立重复进行10次。实验结果如图7所示。

在ADG^{+noise}组中，Harness的初始接受率保持在88.9%±3.1%，与无扰动基准组相比无显著差异。这表明LLM能够凭借预训练语义知识，充当语义过滤器，有效识别并剔除上下文中的无关干扰项；在ADG^{-drop}组中，尽管部分显式依赖关系被移除，但初始接受率仍维持在85.1%，且初始边覆盖数仅由原始的276.1下降至261.0，降幅约为5.5%。这证明即使在静态分析缺失部分关键路径的情况下，LLM依然能够基于代码的语义理解能力，推断出潜在的API调用序列，自动补全缺失的依赖关系。

综上，本文提出的基于ADG引导的Harness生成方案在面对静态分析误差时具有显著鲁棒性，确保其在最差场景下依然能够进行有效测试。

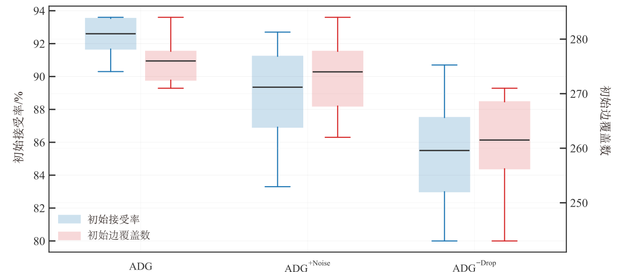


图7　在不同ADG扰动场景下微基准测试的鲁棒性

Figure 7　Robustness of microbenchmarks under different ADG perturbation scenarios

4.5.2　性能与扩展性限制

动态约束推理依赖sys.settrace提供的运行时信息采样，这往往是性能敏感的操作。为评估PyBoros在实际模糊测试中的性能损耗，我们对比了三种配置下微基准测试中60 min测试的平均吞吐量(execs/sec)。

(1)Native Atheris：仅使用Atheris的字节码插桩，代表理想情况下的性能上限。

(2)Full Tracing：在每次执行中强制开启全量sys.settrace，用于模拟传统的全量动态分析性能。

(3)PyBoros：即本文提出的基于停滞触发的按需分析策略，用于评估本方案在实际分析中的性能表现。

实验结果如表8所示，Native Atheris的平均吞吐量约为2 140 execs/sec。相比之下，Full Tracing方案的吞吐量急剧下降至437 execs/sec，性能损耗高达4.8倍，这使得全量追踪方法在实际生产环境中几乎不可用。而PyBoros的平均吞吐量稳定在1 948 execs/sec。与Native基线相比，其额外的分析开销占比仅为8.9%。

PyBoros的性能优势归功于基于停滞触发的按需分析方法带来的性能开销平摊：系统仅在覆盖率增长停滞的极少数时刻触发昂贵的动态分析线程，而在绝大多数时间维持原生执行速度。

值得注意的是，pyyaml包含大量复杂的纯Python逻辑控制流，代表了追踪开销的最坏情况。对于包含大量C扩展的库(如Pillow、cbor2)，由于sys.settrace不追踪C层面的指令，本方案的额外开销预计将进一步降低。根据Paramitha等人[38]对Python生态的大规模测量结果，PyPI中有约95.46%的库小于微基准测试的代码规模(7 121行)；对于超大规模包(>10×10^3 LOC)的拓展支持将是未来的工作重点。

表8　不同配置下微基准测试上的平均吞吐量

Table 8　Average throughput on microbenchmarks under different configurations

配置	平均吞吐量/(execs/sec)	相对吞吐量/(% of Native)	吞吐损失/%
Native Atheris	2 140	100.00	0.00
Full Tracing	437	20.42	79.58
PyBoros	1 948	91.03	8.97

综上所述，PyBoros在保持高吞吐量的同时实现了深度的语义分析，成功在性能与分析精度之间取得了平衡。

5　结论

Python第三方库生态面临日益严峻的软件供应链安全威胁，现有的模糊测试方法在处理动态特性、复杂API交互和运行时约束时效果受限，难以高效挖掘深层漏洞。本文提出PyBoros框架，通过融合静态分析、LLM语义理解和新型覆盖策略，系统解决API交互语义缺失、运行时约束建模困难以及路径探索低效三大挑战。该框架包括：基于ADG和LLM的语义丰富Harness自动生成；停滞触发式动态分析结合LLM的智能约束推理；以及API n-gram覆盖导向的智能资源分配策略。这些设计显著提升了测试的有效性和深度。

实验结果表明，PyBoros在真实项目数据集上优于现有工具，在漏洞发现能力、代码覆盖率和Harness质量均有明显提升，并在扰动场景下保持较高鲁棒性。该框架为Python第三方库的安全分析提供了一种实用、高效的方法，并为Python生态的安全保障提供新思路。

然而，PyBoros在实际分析时依然具有局限性。在停滞事件发生时，考虑到分析效率与LLM成本，优先对价值更高的分支进行约束推理。虽然通过参数敏感性分析实验验证在默认参数下能够平衡表现与开销，然而在面对不同的分析目标时，设置不同的权重可能产生更优的推理效果。如何为不同的分析目标自动化的计算出最优的权重配置，以及各个参数对漏洞发现的具体相关性验证仍有不足。此外PyBoros的停滞分析虽然有性能平摊机制，但在面对超大型库时依然可能产生性能瓶颈。如何提高对于超大型库的分析支持能力，将是未来工作需要解决的问题。

参考文献

[1] Github. Programming languages-GitHub innovation graph[EB/OL]. (2023-09-21)[2025-07-02]. https://innovation-graph.github.com/global-metrics/programming-languages.

[2] Birney R. Lessons from the recent PyTorch supply chain attack[EB/OL]. (2025-05-21)[2026-02-10]. https://www.get-safety.com/blog-posts/lessons-from-the-recent-pytorch-supply-chain-attack.

[3] REVERSINGLABS. The 2025 software supply chain security report[R/OL]. (2025-03-17)[2026-02-10]. https://ntsc.org/wp-content/uploads/2025/03/The-2025-Software-Sup -ply-Chain-Security-Report-RL-compressed.pdf.

[4] Alfadel M, Costa D E, Shihab E. Empirical analysis of security vulnerabilities in Python packages[J]. Empirical Software Engineering, 2023, 28(3): 59.

[5] Decan A, Mens T, Grosjean P. An empirical comparison of dependency network evolution in seven software packaging ecosystems[J]. Empirical Software Engineering, 2019, 24(1): 381-416.

[6] Kikas R, Gousios G, Dumas M, et al. Structure and evolution of package dependency networks[C]//2017 IEEE/ACM 14th International Conference on Mining Software Repositories. Piscataway: IEEE, 2017: 102-112.

[7] Pashchenko I, Plate H, Ponta S E, et al. Vulnerable open source dependencies: Counting those that matter[C]//Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. New York: ACM, 2018: 1-10.

[8] Zalewski M. American Fuzzy Lop[CP/OL]. (2017-11-24)[2025-07-08]. http://lcamtuf.coredump.cx/afl/.

[9] FioraldI A, Maier D, Fioraldi D, et al. AFL++: Combining Incremental Steps of Fuzzing Research[C]//Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 20). [S.l.]: USENIX Association, 2020: 1-16.

[10] 许航, 计江安, 马哲宇, 等. 基于分布散度的自适应模糊测试优化方法[J]. 网络与信息安全学报, 2024, 10(6): 37-58.
Xu Hang, Ji Jiang'an, Ma Zheyu, et al. Self-adaptive fuzzing optimization method based on distribution divergence[J]. Chinese Journal of Network and Information Security, 2024, 10(6): 37-58. (in Chinese)

[11] 肖天, 江智昊, 唐鹏, 等. 基于深度强化学习的高性能导向性模糊测试方案[J]. 网络与信息安全学报, 2023, 9(2): 132-142.
Xiao Tian, Jiang Zhihao, Tang Peng, et al. High-performance directional fuzzing scheme based on deep rein-

forcement learning[J]. Chinese Journal of Network and Information Security, 2023, 9(2): 132-142. (in Chinese)

[12] 侍言, 羌卫中, 邹德清, 等. 进化内核模糊测试研究综述[J]. 网络与信息安全学报, 2024, 10(1): 1-21.
Shi Yan, Qiang Weizhong, Zou Deqin, et al. Survey of evolutionary kernel fuzzing[J]. Chinese Journal of Network and Information Security, 2024, 10(1): 1-21. (in Chinese)

[13] 徐恪, 冯学伟, 李琦, 等. 安全可信的互联网体系结构与端到端传送关键技术[J]. 中兴通讯技术, 2022, 28(6): 17-22.
Xu Ke, Feng Xuewei, Li Qi, et al. Secure and trusted Internet architecture and key technologies of end-to-end transmission[J]. ZTE Technology Journal, 2022, 28(6): 17-22. (in Chinese)

[14] GOOGLE. Atheris: A Coverage-Guided, Native Python Fuzzer[CP/OL]. (2020-11-18)[2025-07-08]. https://github.com/google/atheris.

[15] Li W, Yang H R, Luo X P, et al. PyRTFuzz: Detecting bugs in Python runtimes via two-level collaborative fuzzing[C]//Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2023: 1645-1659.

[16] Deng Y L, Xia C S, Peng H R, et al. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models[C]//Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM, 2023: 423-435.

[17] Deng Y L, Xia C S, Yang C Y, et al. Large language models are edge-case fuzzers: Testing deep learning libraries via FuzzGPT[PP/OL]. V1. arXiv (2023-04-04) [2025-07-10]. https://doi.org/10.48550/arXiv.2304.02014.

[18] Yang C Y, Deng Y L, Lu R Y, et al. WhiteFox: White-box compiler fuzzing empowered by large language models[J]. Proceedings of the ACM on Programming Languages, 2024, 8(OOPSLA2): 709-735.

[19] Wang J C, Yu L, Luo X P. LLMIF: Augmented large language model for fuzzing IoT devices[C]//2024 IEEE Symposium on Security and Privacy. Piscataway: IEEE, 2024: 881-896.

[20] Yang L Q, Yang J, Wei C R, et al. FuzzCoder: Byte-level fuzzing test via large language model[PP/OL]. V1.arXiv (2024-09-03)[2025-07-10]. https://doi.org/10.48550/arXiv.2409.01944.

[21] Wang D W, Zhou G, Chen L, et al. ProphetFuzz: Fully automated prediction and fuzzing of high-risk option combinations with only documentation via large language model[C]//Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2024: 735-749.

[22] Zhang C, Zheng Y W, Bai M Q, et al. How effective are they? Exploring large language model based fuzz driver generation[C]//Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM, 2024: 1223-1235.

[23] 许婷, 肖桐, 张圣林, 等. 基于LLM的日志故障诊断[J]. 电子学报, 2025, 53(4): 1123-1141.
Xu Ting, Xiao Tong, Zhang Shenglin, et al. Log fault diagnosis based on large language models[J]. Acta Electronica Sinica, 2025, 53(4): 1123-1141. (in Chinese)

[24] 张奎元, 张启亮, 陈朋朋, 等. 面向大规模地下空间的多智能体端边协作全局SLAM方法[J]. 电子学报, 2025, 53(11): 3852-3864.
Zhang Kuiyuan, Zhang Qiliang, Chen Pengpeng, et al. Robots-edge collaborative absolute SLAM in large-scale underground environments[J]. Acta Electronica Sinica, 2025, 53(11): 3852-3864. (in Chinese)

[25] Lu Mengdi, Ding S, Alaca F, et al. Semantic-aware fuzzing: An empirical framework for LLM-guided, reasoning-driven input mutation[PP/OL]. V1.arXiv (2025-09-30)[2025-07-08]. https://arxiv.org/abs/2509.19533.

[26] Meng R J, Mirchev M, Böhme M, et al. Large language model guided protocol fuzzing[C]//Proceedings 2024 Network and Distributed System Security Symposium.[S.l.]: Internet Society, 2024: 1-16. DOI:10.14722/ndss.2024.24556.

[27] Li X, Yuan Z Y, Zhang Z D, et al. Towards large language model guided kernel direct fuzzing[M]//Fundamental Approaches to Software Engineering. ChamSpringer Nature Switzerland, 2025: 33-42.

[28] Xia C S, Paltenghi M, Jia L T, et al. Fuzz4All: Universal fuzzing with large language models[C]//Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. New York: ACM, 2024: 1-13.

[29] Bazalii M, Fleischer M. Orion: Fuzzing workflow automation[PP/OL]. V1. arXiv (2025-09-29) [2025-07-08]. https://arxiv.org/abs/2509.15195.

[30] Liu N F, Lin K, Hewitt J, et al. Lost in the middle: How language models use long contexts[J]. Transactions of the Association for Computational Linguistics, 2024, 12: 157-173.

[31] Laban P, Fabbri A, Xiong C M, et al. Summary of a haystack: A challenge to long-context LLMs and RAG systems[C]//Proceedings of the 2024 Conference on Empiri-

cal Methods in Natural Language Processing. Stroudsburg: ACL, 2024: 9885-9903.

[32] Kurfali M, Östling R. Conflicting Needles in a Haystack: How LLMs behave when faced with contradictory information[C]//Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing. Stroudsburg: ACL, 2025: 34349-34364.

[33] Peng X, Jia P, Fan X M, et al. ENZZ: Effective N-gram coverage assisted fuzzing with nearest neighboring branch estimation[J]. Information and Software Technology, 2025, 177: 107582.

[34] Bouzenia I, Krishan B P, Pradel M. DyPyBench: A benchmark of executable Python software[J]. Proceedings of the ACM on Software Engineering, 2024, 1: 338-358.

[35] Xiao J F, Jiang P, Zhao Z X, et al. Robust, efficient, and widely available greybox fuzzing for COTS binaries with system call pattern feedback[C]//Proceedings of the 34th USENIX Conference on Security Symposium. New York: ACM, 2025: 6239-6258.

[36] Wang Y H, Jia X K, Liu Y W, et al. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization[C]//Proceedings 2020 Network and Distributed System Security Symposium. [S.l.]: Internet Society, 2020: 1-15. DOI:10.14722/ndss.2020.24422.

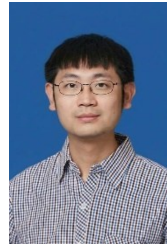
[37] Zügner D, Akbarnejad A, Günnemann S. Adversarial attacks on neural networks for graph data[C]//Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. New York: ACM, 2018: 2847-2856.

[38] Paramitha R, Massacci F. Technical leverage analysis in the Python ecosystem[J]. Empirical Software Engineering, 2023, 28(6): 139.

作者简介

李政浩　男,1997年12月出生于山西省太原市。现为中国科学院软件研究所博士研究生、CCF学生会员。主要研究领域为系统安全、漏洞挖掘、漏洞分析。

E-mail: zhenghao2021@iscas.ac.cn



贾相堃　男,1990年2月出生于河北省邯郸市。博士,现为中国科学院软件所副研究员,CCF专业会员。主要研究领域为系统安全、漏洞挖掘与分析。

E-mail: xiangkun@iscas.ac.cn

闫新成　男,1977年12月出生于甘肃省武威市。现为东南大学网络空间安全学院校外导师。主要研究领域为通信网络安全和AI安全。

E-mail: 230239737@seu.edu.cn

苏璞睿　男,1976年12月出生于湖北省宜昌市。博士,博士生导师,现为中国科学院软件研究所研究员、CF专业会员。主要研究领域为系统安全、恶意代码分析、漏洞挖掘。中国电子学会会员编号:E190029904M。

E-mail: purui@iscas.ac.cn

王继刚　男,1978年7月出生于安徽省马鞍山市。现为中兴通讯股份有限公司首席专家。主要研究领域网络安全、具身智能。

E-mail: wang.jigang@zte.com.cn