

覆盖率引导的 Python JIT 编译器缺陷检测^{*}

任志磊, 张子龙, 周志德, 李微微, 江贺

(大连理工大学 软件学院, 辽宁 大连 116024)

通信作者: 任志磊, E-mail: zren@dlut.edu.cn



摘要: Python 作为一种广泛应用的解释型语言, 在执行效率方面存在性能瓶颈. 即时 (JIT) 编译器被引入 Python 生态, 它通过将字节码动态编译为机器码, 显著提升了程序的运行速度. 然而, JIT 编译器复杂的优化策略可能导致程序缺陷, 影响程序的稳定性和可靠性. 现有的 Python 解释器模糊测试方法难以有效检测 JIT 编译器的深层优化缺陷和非崩溃缺陷. 为此, 提出了一种基于覆盖率引导的 Python JIT 编译器缺陷检测方法 PjitFuzz. 首先, 为了能够生成触发 Python JIT 编译器优化策略的程序变体, PjitFuzz 提出了 5 种基于 JIT 优化策略的程序变异规则. 其次, 为了聚合不同变异规则的优势并生成多样化的程序变体, PjitFuzz 设计了一种基于覆盖率引导的变异规则动态选择方法. 然后, 为了有效记录程序执行过程中变量值的变化情况, 从而检测输出不一致的缺陷, PjitFuzz 提出了一种基于计算校验和的代码块插入策略. 最后, 结合不同的 JIT 编译选项进行差分测试, 从而有效检测 Python JIT 编译器缺陷. 将 PjitFuzz 与目前最先进的两种 Python 解释器模糊测试方法 FcFuzzer 和 IFuzzer 进行比较, 实验结果表明, PjitFuzz 在缺陷检测能力上, 分别高出 150% 和 66.7%; 在代码覆盖率方面, 分别比现有方法高出 28.23% 和 15.68%; 在生成测试程序有效率方面, 分别高出 42.42% 和 62.74%. 在为期 8 个月的实验中, PjitFuzz 发现并报告了 16 个缺陷, 其中 12 个已得到开发人员的确认.

关键词: Python JIT 编译器; 缺陷检测; 优化策略; 校验和; 差分测试

中图法分类号: TP311

中文引用格式: 任志磊, 张子龙, 周志德, 李微微, 江贺. 覆盖率引导的 Python JIT 编译器缺陷检测. 软件学报. <http://www.jos.org.cn/1000-9825/7575.htm>

英文引用格式: Ren ZL, Zhang ZL, Zhou ZD, Li WW, Jiang H. Coverage-guided Defect Detection for Python JIT Compilers. Ruan Jian Xue Bao/Journal of Software (in Chinese). <http://www.jos.org.cn/1000-9825/7575.htm>

Coverage-guided Defect Detection for Python JIT Compilers

REN Zhi-Lei, ZHANG Zi-Long, ZHOU Zhi-De, LI Wei-Wei, JIANG He

(School of Software, Dalian University of Technology, Dalian 116024, China)

Abstract: As a widely employed interpreted language, Python faces performance challenges in execution efficiency. Just-in-time (JIT) compilers have been introduced to the Python ecosystem to dynamically compile bytecode into machine code, significantly improving program operation speed. However, the complex optimization strategies of JIT compilers may introduce program defects, thereby affecting program stability and reliability. Existing fuzz testing methods for Python interpreters struggle to effectively detect deep optimization defects and non-crashing defects in JIT compilers. To this end, this study proposes PjitFuzz, a coverage-guided defect detection method for Python JIT compilers. First, PjitFuzz proposes five mutation rules based on JIT optimization strategies to generate program variants that trigger the optimization strategies of Python JIT compilers. Second, a coverage-guided dynamic mutation rule selection method is designed to integrate the advantages of different mutation rules and generate diverse program variants. Third, a checksum-based code block insertion strategy is developed to effectively record changes in variable values during program execution and detect inconsistency in the output.

* 基金项目: 国家自然科学基金 (62132020, 62032004); 国家自然科学基金青年科学基金 (62302077); 中央高校基本科研业务费专项资金 (DUT24LAB126)

收稿时间: 2025-05-13; 修改时间: 2025-09-23, 2025-10-09; 采用时间: 2025-11-14; jos 在线出版时间: 2026-02-04

Finally, differential testing is performed by combining different JIT compilation options to effectively detect defects in Python JIT compilers. This study compares PjitFuzz with two state-of-the-art Python interpreter fuzzing methods, FcFuzzer and IFuzzer. The experimental results show that PjitFuzz improves defect detection capability by 150% and 66.7% respectively, and outperforms existing methods in terms of code coverage by 28.23% and 15.68% respectively. For the validity rate of generated test programs, PjitFuzz outperforms the comparative methods by 42.42% and 62.74% respectively. In an eight-month experiment, PjitFuzz has discovered and reported 16 defects, 12 of which have been confirmed by developers.

Key words: Python JIT Compiler; defect detection; optimization strategies; checksum; differential testing

Python 自 1991 年首次发布以来, 凭借其简洁的语法、强大的库支持和跨平台特性, 迅速成为全球开发者广泛使用的编程语言之一^[1-3]. CPython 是 Python 的官方实现, 通常作为默认的 Python 解释器在开发和生产环境中使用^[4]. 然而, 由于 CPython 是解释型执行的, 逐行解析代码的方式导致其在计算密集型任务中的运行效率相对较低, 尤其是在处理大规模数据和复杂运算时, 表现出明显的性能瓶颈.

为了提升 Python 程序的执行速度, JIT (just-in-time) 编译器的引入成为一种有效的解决方案^[5]. Python JIT 编译器一般工作流程如图 1 所示. 首先, Python 代码会被解析成抽象语法树 (AST), 然后转化为字节码. 在传统的 Python 执行模型中, 字节码由解释器逐行执行; 而 JIT 编译器则会根据程序的执行情况, 动态分析程序的热点部分, 将字节码编译成机器码, 从而避免了解释器逐行执行字节码的开销. 在分析和优化阶段, Python JIT 编译器应用多种策略对字节码进行优化, 以显著提升程序执行效率. 这种 JIT 编译与传统的解释执行机制相辅相成, 共同为程序提供了灵活性和高效性. 需要注意的是, Python 的 JIT 编译器不仅是一个 JIT 编译器, 它通常还集了解释器、垃圾回收器、内存管理器等多个组件. 然而, 随着其广泛应用, JIT 编译器的正确性和可靠性变得尤为关键. 如果存在缺陷或错误, 可能会导致程序运行时崩溃、输出结果不准确或者行为异常, 进而引发安全隐患. 因此, 确保 Python JIT 编译器的稳定性, 对于提升 Python 在高性能应用中的竞争力、保障其安全性和可靠性至关重要.

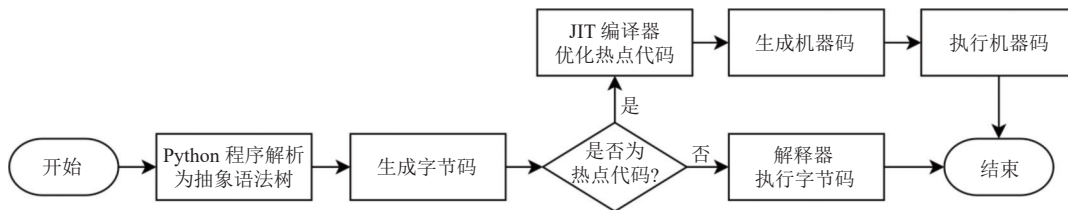


图 1 Python JIT 编译器的一般工作流程

模糊测试^[6]是一种自动化软件测试方法, 通过输入大量意外或随机生成的数据来触发系统的异常路径, 以此揭示潜在的错误或漏洞. 一方面, 基于生成的方法^[7]通过深度理解目标程序的语法结构, 生成符合规则的输入, 覆盖性高但过程复杂, 难以快速适应多样化需求; 另一方面, 基于变异的方法^[8,9]则通过随机变异现有输入生成新的测试数据, 生成速度更快, 但随机变异可能导致无效输入, 影响测试准确性. 针对 Python 解释器设计的 FcFuzzer^[10]和 IFuzzer^[11]是两种典型的基于变异的模糊测试方法. FcFuzzer 使用“骨架程序枚举”框架生成多样化的函数调用程序, 结合差分测试, 能够有效检测解释器在复杂调用场景中的潜在缺陷. IFuzzer 则通过树结构的骨架程序枚举直接生成非等价程序, 避免生成无效等价程序, 并利用控制流信息减少无效程序, 从而提升测试效率. 尽管这些模糊测试方法对 Python 解释器的稳定性检测提供了有力支持, 但在测试 JIT 编译器时, 现有方法仍然面临两大挑战.

挑战 1. 如何生成能够有效激活 Python JIT 编译器优化策略的测试程序. JIT 编译器需要对代码进行复杂的优化处理, 其缺陷通常涉及公共子表达式消除^[12]、常量折叠^[13]、函数内联^[14]、简化^[15]和循环优化^[16]等策略. 然而, 这些优化策略仅在特定条件下触发, 而现有的测试输入无法充分覆盖这些场景. 相比之下, Python 解释器通常不进行深度优化或仅进行有限优化, 因此传统的模糊测试方法难以有效激活 JIT 编译器的特定优化策略. 由于缺乏对 JIT 优化过程的深入测试, 传统模糊测试工具在 Python JIT 编译器上的测试覆盖率不足, 难以检测优化策略导致的缺陷. 尽管现有方法偶尔会暴露 JIT 缺陷, 但其实际效果有限, 亟需开发针对 JIT 编译器的缺陷检测方法.

挑战 2. 如何有效检测 Python JIT 编译器中的非崩溃缺陷. 现有方法通常将程序崩溃作为主要的缺陷判定标

准,重点关注触发解释器崩溃的输入.然而,JIT编译器中的许多缺陷并不会导致程序崩溃,而是表现为输出不一致或行为不一致.传统模糊测试往往忽视这些非崩溃类缺陷,未能有效检测到这些难以察觉的问题.例如,为了避免耗费大量人力检查误报的不一致缺陷,IFuzzer的研究主要集中在触发Python解释器崩溃的缺陷,明确将非崩溃缺陷排除在检测范围之外.这些非崩溃缺陷通常更加隐蔽,可能会对后续的科学研究或其他数据处理任务的结果和准确性产生严重影响.

为应对上述两个挑战,本文设计并开发了一种基于覆盖率引导的Python JIT编译器缺陷检测框架PjitFuzz.首先,为了解决挑战1,本文基于已有模糊测试工作的思想框架,针对Python JIT编译器的5种优化策略设计了适配其特性的专用变异规则.具体而言,本文根据公共子表达式消除、常量折叠、函数内联、简化和循环优化这些JIT优化策略,提出了5种对应的变异规则,以生成能够触发JIT优化策略的程序变体;在此基础上,为了聚合不同变异规则的优势并生成多样化的程序变体,本文结合汤普森采样(Thompson sampling)算法,提出了一种基于覆盖率引导的变异规则动态选择方法.其次,为了解决挑战2,本文提出了两种非崩溃的测试预言检测机制.首先是输出不一致缺陷检测:当同一测试程序在不同JIT编译选项下执行时,若其输出的校验和不一致,则判定存在缺陷.具体而言,为了有效记录程序执行过程中变量值的变化情况,本文提出了一种计算程序校验和的代码块插入策略,以检测输出不一致的缺陷.其次是行为不一致缺陷检测:若程序在一种JIT编译选项下正常执行,而在另一种编译选项下异常终止(如抛出未捕获异常),则判定存在缺陷.本文通过结合不同的JIT编译选项进行差分测试,比较操作系统返回的不同进程状态码的差异,用于检测Python JIT编译器的行为不一致缺陷.

为了评估PjitFuzz的有效性,本文在PyPy^[17]和Numba^[18]两种流行的Python JIT编译器上进行了为期8个月的实验.PjitFuzz共发现并报告了16个缺陷,其中12个已得到开发人员的确认.同时,本文将PjitFuzz与目前最先进的两种Python解释器模糊测试方法FcFuzzer和IFuzzer进行了比较.实验结果表明,PjitFuzz的缺陷检测能力分别比这两种基线方法提高了150%和66.7%,代码覆盖率高出了28.23%和15.68%,生成测试程序有效率分别高出42.42%和62.74%.此外,本文为PjitFuzz设计了3种变体方法,通过消融实验验证了基于JIT优化策略的程序变异、覆盖率引导的变异规则选择以及校验和代码块插入这3个关键组件的有效性.

综上所述,本文主要贡献如下.

- 1) 系统性地开展了Python JIT编译器缺陷检测研究工作,并针对其中5种优化策略设计了适配其特性的专用变异规则.

- 2) 提出一种覆盖率引导的Python JIT编译器缺陷检测方法PjitFuzz.

- 3) 对2种流行的Python JIT编译器进行了广泛实验,结果表明,PjitFuzz在缺陷检测能力上分别比现有的两种Python解释器测试方法高出150%和66.7%,在代码覆盖率方面分别比现有方法高出28.23%和15.68%,在生成测试程序有效率方面,分别高出42.42%和62.74%.此外,PjitFuzz在最新版本Python JIT编译器上发现并报告了16个缺陷,其中12个已被确认.

本文第1节介绍研究背景,并通过两个简单示例引出研究动机.第2节详细阐述提出的方法框架PjitFuzz.第3节通过设计对比实验,从多个角度验证所提方法的有效性.第4节介绍相关工作.第5节总结本文工作.

1 研究背景与动机

Python的编译是一种在程序运行时将字节码编译为机器码的技术.然而,JIT编译器的实现较为复杂.在分析和优化阶段,Python的JIT编译器需要应用多种技术对字节码进行优化,这种复杂性使得JIT编译器在开发和维护过程中容易引入缺陷.JIT编译器的缺陷分为两大类:崩溃缺陷和非崩溃缺陷.根据缺陷症状的不同,非崩溃缺陷可以进一步细分为输出不一致缺陷和行为不一致缺陷^[19,20].接下来,本文将通过两个Python JIT编译器缺陷案例,深入分析这两种非崩溃缺陷的表现及其影响.

输出不一致缺陷是指程序在不同的JIT编译选项下执行时,会产生不同的输出结果,但不会导致程序崩溃.此类缺陷通常发生在JIT编译器的优化过程中,尤其是在涉及循环和常量折叠时.尽管程序能够继续执行,但输出结

果可能会受到 JIT 优化策略缺陷的影响,从而导致不可预测的输出.例如,图 2 展示的代码片段来自 Python JIT 编译器 PyPy 的缺陷仓库中的#2904 号报告(详见 <https://github.com/pypy/pypy/issues/2904>).当禁用 JIT 编译器时,程序正确输出 4;然而,当使用“pypy --jit threshold=15”启动 JIT 编译器时,程序会输出 0 和 4.这是因为 JIT 编译器在循环展开优化过程中错误地推断了 height 的值,进而导致输出结果的不一致.

```

1 def eva(height):
2     for c in range(10):
3         print(height-1)
4         h = height
5         while h > 0:
6             h -= 1
7 while True:
8     eva(5)
9 #正确输出: 4
10 #错误输出: 0和4

```

图 2 PyPy 输出不一致缺陷示例 (#2904)

行为不一致缺陷是指在不同 JIT 编译选项下,编译器执行程序时表现出不符合预期的行为或异常错误,这类缺陷可能不会直接导致程序崩溃,通常难以通过常规测试方法发现,因此成为本文重点研究的对象.图 3 中展示的代码片段来自 Python JIT 编译器 Numba 的缺陷仓库中的#2518 号报告(详见 <https://github.com/numba/numba/issues/2518>).正常情况下,程序执行结束并输出 [5, 4, 2] 的数组,然而,编译器在处理 a.shape[:b.ndim] 时未能正确折叠常量,最终引发了类型推断失败,导致程序无法执行结束,并在终端输出堆栈错误信息.

```

1 import numpy as np
2 def f(a, b):
3     new_shape = a.shape[:b.ndim] + a.shape[b.ndim + 1:]
4     return np.ones(new_shape)
5 print(f(np.zeros((5, 4, 3, 2)), np.zeros((1, 1))).shape)
6 #正确行为: 程序正常执行结束, 输出[5, 4, 2]
7 #异常行为: 程序执行失败, 产生以下异常信息
8 #Failed at nopython (nopython frontend)
9 #Invalid usage of getitem with parameters ((int64 x 3), slice<a:b>)
10 # * parameterized

```

图 3 Numba 行为不一致缺陷示例 (#2518)

首先,以上两类缺陷均发生在 JIT 编译器优化过程中, JIT 编译器可能会对某些代码路径进行不恰当的处理,从而导致程序产生不一致结果.其次,这两类缺陷通常不会导致程序崩溃,且难以通过单次测试发现,需要开发人员在不同的 JIT 编译选项下进行多次测试和对比,这不仅增加了调试和排错的复杂性,还可能影响业务或科研结果的准确性.综上所述,现有的 Python 解释器模糊测试方法在检测 JIT 编译器深层优化导致的缺陷和非崩溃缺陷方面存在关键挑战,开发有效的缺陷检测方法对于确保 Python JIT 编译器的质量至关重要.

2 方法介绍

本文提出的 PjitFuzz 方法框架如图 4 所示. PjitFuzz 由程序收集与预处理、基于 JIT 优化策略的程序变异、覆盖率引导的变异规则选择、校验和代码块插入和缺陷检测这 5 个部分组成.具体来说,在程序收集与预处理组件中, PjitFuzz 从 GitHub 收集一组 Python 程序,并通过解释器执行剔除其中存在异常的程序,以保证所输入程序的可靠性,保留下来的程序作为后续实验的种子程序.然后,在基于 JIT 优化策略的程序变异组件中, PjitFuzz 采用针对 JIT 优化的变异规则(如公共子表达式消除、常量折叠、函数内联、简化和循环优化)对筛选出的种子程序进行变异,生成 JIT 程序变体.同时,在覆盖率引导的变异规则选择组件中, PjitFuzz 依据 JIT 编译器执行时获得的覆盖率信息,动态调整 JIT 变异规则的选择,以增强程序变体的多样性.随后,在校验和代码块插入组件中, PjitFuzz

对 JIT 程序变体插入计算校验和的代码块,生成校验增强的程序变体.校验和用于记录程序执行后变量值的变化,从而验证程序的输出一致性.最终,在缺陷检测组件中,PjitFuzz 通过设置不同的 JIT 编译选项进行差分测试,观察程序的输出校验和与异常行为,以检测可能存在的输出不一致和行为不一致缺陷.为了便于开发人员复现和确认问题,本文会去除无效和重复的缺陷,并对可疑程序进行约简,最终将处理后的结果打包并提交给开发人员.接下来,本节将介绍 PjitFuzz 各个组件的具体实现细节.

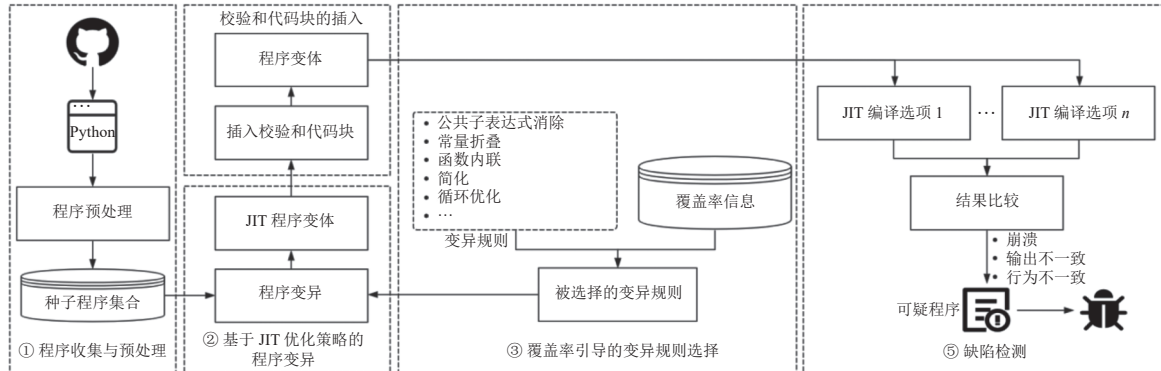


图4 PjitFuzz 方法框架

2.1 程序收集与预处理

本文的初始 Python 程序主要来源于两部分.首先,第 1 部分来自 GitHub 上与“Python”相关的开源存储库.本文根据这些存储库的受欢迎程度进行排名,下载前 2000 个热门存储库,并通过文件后缀筛选出其中的 Python 文件.这些存储库涵盖广泛的应用场景,包括数据科学、人工智能、Web 开发、系统工具等领域,从而确保了种子程序的多样性和广泛性.第 2 部分程序来自 4 个热门 GitHub 项目的 Issues 页面,包括 CPython、PyPy、Numba 和 NumPy^[21],这些项目与本文所测试的 JIT 编译器密切相关.例如,Numba 编译器和 NumPy 第三方库广泛应用于数值计算领域,且二者通常结合使用.具体而言,本文通过爬虫抓取这些存储库的 Issues 页面中开发人员和用户提交的 Python 代码片段.这些代码片段通常用于复现特定问题或异常行为,经过变异后,若能够生成新的执行路径,则有助于提高触发缺陷的可能性,从而为 PjitFuzz 提供种子程序的来源.

在获取初始程序集后,本文使用 Python 标准库 ast^[22]中的 ast.parse() 函数对每个 Python 源程序进行静态解析,确保程序的语法符合 Python 语言规范.具体而言,该函数接受 Python 源代码作为输入,并返回对应的抽象语法树对象.如果解析过程中抛出 SyntaxError 异常,则说明程序中存在语法错误,此类程序将被剔除.保留下来的程序将作为语法有效程序集,并为后续分析提供基础.对于第 1 部分的程序(即 GitHub 存储库中的代码),本文进一步使用 Python 标准解释器 CPython 执行这些程序,并在执行过程中剔除存在执行错误(如运行时异常)或长时间执行而不终止的程序.这样,PjitFuzz 能够有效隔离潜在问题,减少无效程序对变异和差分测试阶段的干扰.对于第 2 部分的程序(GitHub Issues 中的代码),由于这些代码片段的主要目的是复现特定问题或异常行为,因此只进行语法有效性筛选,不进行 CPython 执行验证.这种方法可以保留更多有价值的测试程序,避免因执行验证导致有潜力的缺陷复现代码被误剔除,从而提高缺陷检测的全面性和有效性.最终,经过上述筛选保留下来的程序将作为 PjitFuzz 的种子程序,为 JIT 编译器缺陷检测提供高质量的测试输入.

2.2 基于 JIT 优化策略的程序变异

为了提高 Python 程序的执行性能,JIT 编译器采用了多种优化策略,如常量折叠、函数内联等.如第 1 节中的缺陷示例可知,JIT 编译器的优化策略常涉及复杂的代码转换和执行流程,可能引入潜在的缺陷.直观来看,如果测试程序能够激活 JIT 编译器中的这些优化策略,那么 JIT 缺陷被检测的可能性也会相应提高.因此,本文针对这些优化策略,设计了一系列变异规则,旨在触发特定的优化路径,创造能够暴露 JIT 编译器缺陷的场景.尽管可以为每种优化策略都设计相应的变异规则,但全面且细致地构建这些变异规则可能会导致成本较高且效率低下.本文

选择公共子表达式消除、常量折叠、函数内联、简化和循环优化这 5 种优化策略作为变异目标,主要是对 Python JIT 编译器的适配性以及最小规则集覆盖最大优化场景的考量. 首先,公共子表达式消除和常量折叠是编译器优化中常见的优化规则,在不改变程序所计算的函数的情况下改进程序^[23]. 其次,动态语言通常需要进行大量的类型检查. 对于计算密集的循环,大量的执行时间可能会花在这样的任务上,而不是实际的计算上,这也是 Python JIT 编译器优化的重点方向^[24]. 由于公共子表达式消除、常量折叠、循环优化在 Python JIT 编译器的广泛运用,针对这些规则设计变异策略,能够精准触达最关键的优化逻辑,提升潜在缺陷的挖掘效率,能以更低的变异成本获得更高的缺陷发现收益. 此外,借鉴 JITfuzz^[25]的实践经验,本文同样将函数内联与代码简化列为变异目标,以放大覆盖并挖掘深层缺陷. 对于 JITfuzz 中提出的标量替换与逃逸分析,受限于 Python JIT 编译器动态类型不确定性,优化影响收益有限,故未被选择. 最后,本文基于选择的 5 种策略提出了对应的程序变异规则. 表 1 列出了这些变异规则的详细信息.

表 1 基于 JIT 优化策略的变异规则

变异规则类型	示例	
	变异前	变异后
公共子表达式消除	$res=a*b+c*d$	$cse1=(a*b)/(c*d)$ $cse2=a*b-c*d$ $res=a*b+c*d$
常量折叠	$num=n$ $str=s$	$num=n1*n2$ $str=s1+s2$
函数内联	$res=a+b$	<code>def inline(x, y):</code> <code>return x+y</code> $res=inline(a, b)$
简化	$res=a+b$	$expr=rand(0, 100)$ $res=(a+expr)+(b-expr)$
循环优化	$a=b+c$ <code>for i in range(n):</code> <code>do_something</code>	<code>for i in range(n):</code> $a=b+c$ <code>do_something</code>

注: *表示Python语法中的乘法操作

- 公共子表达式消除^[12]: 这是一种优化策略,用于查找和消除代码中的冗余表达式,从而减少重复计算. 本文设计的公共子表达式消除变异规则通过提取表达式中的公共子表达式,并组合不同的运算生成等价表达式,以测试优化器对公共子表达式消除的处理效果. 具体来说,对于一个原表达式, PjitFuzz 首先提取其中的子表达式,将其存储为一个新的临时变量,然后使用不同的运算符重新组合子表达式生成新表达式. 例如,本文从 $result=a*b+c*d$ 中提取出子表达式 $(a*b)$ 和 $(c*d)$,并组合生成新的表达式,如 $(a*b)/(c*d)$ 和 $a*b-c*d$ 等. 通过这种方法, PjitFuzz 可以生成公共子表达式消除的优化场景,从而评估编译器的公共子表达式消除能力.

- 常量折叠^[13]: 指在编译阶段,编译器会识别并计算表达式中的常量部分,将其替换为计算结果,以减少运行时的计算负担. 本文设计的常量折叠变异规则通过在表达式中插入和替换常量值,以触发编译器的常量折叠优化行为. 对于数值常量, PjitFuzz 会将其替换为计算结果相同的复杂表达式. 对于字符串常量, PjitFuzz 会将其替换为组合后结果相同的子字符串表达式. 例如,对于数值表达式 $num=n$, PjitFuzz 会将其替换为 $num=n1*n2$,其中 $n1$ 和 $n2$ 是其他数值常量,且最终结果等于原始常量值. 同样,对于字符串表达式 $str=s$, PjitFuzz 会将其替换为 $str=s1+s2$,其中 $s1$ 和 $s2$ 是子字符串,且组合后等于原始字符串值. 通过创建等价的常量表达式组合, PjitFuzz 可以测试编译器对数值和字符串常量折叠的处理能力.

- 函数内联^[14]: 是指编译器将函数的代码直接嵌入到调用该函数的位置,从而避免了函数调用时的栈帧创建、参数传递和返回值处理等开销,因此特别适用于频繁调用的小型函数. 本文的函数内联变异规则通过将表达式替换为函数调用,以测试编译器的内联优化能力. 例如,对于原始代码中的表达式 $res=a+b$, PjitFuzz 会构建一个新函数 $inline(x, y)$,其功能是返回 $x+y$,并将 $res=a+b$ 替换为函数调用 $res=inline(a, b)$. 通过将简单表达式封装为函数调用, PjitFuzz 模拟了实际应用中频繁调用函数的场景,从而评估编译器在内联优化方面的处理能力.

• 简化^[15]: 需要注意的是, 这里的简化特指对算术表达式的简化. 因此, 本文的简化变异规则是指将简单算术表达式替换为语义不变但结构更复杂的表达式. 例如, 对于原始表达式 $res=a+b$, 简化变异规则会将其改写为 $res=a+b+expr$, 其中 $expr$ 是一个等于 0 的冗余表达式. 这种改写保持了表达式的原始结果不变, 但人为增加了计算复杂性, 从而测试编译器能否识别并消除这些冗余部分, 有效评估编译器在简化算术表达式方面的处理能力.

• 循环优化^[16]是一种编译器优化策略, 用于提高循环的执行效率, 减少不必要的计算和内存访问. 本文的循环优化变异规则通过在循环中引入冗余的计算负担, 以测试编译器对循环优化的处理能力. 具体来说, PjitFuzz 会将循环外部的不变量 (即在每次循环迭代中结果不变的表达式) 移入循环内部, 增加不必要的计算负担, 从而观察优化器是否能够识别并优化这些冗余表达式. 例如, 原始代码中的表达式 $a=b+c$ 应该在循环外执行, 但在变异过程中, 它被移入循环内部, 导致每次迭代中都重复计算. 通过生成多种等价的循环变体, PjitFuzz 可以评估编译器在循环优化方面的处理能力.

首先, 如第 2.1 节所述, 经过筛选得到的 PjitFuzz 种子程序在语法和语义上都是有效的. 其次, 本文基于 JIT 优化策略的程序变异过程是在抽象语法树级别上进行的, 能够确保生成程序的语法正确性; 同时这些变异规则并不会引入新的语义变化, 程序在变异前后的语义通常是等价的. 最后, 考虑到 Python 的动态类型特性与语义验证的固有矛盾, Python 变量类型、函数行为需在运行时确定, 静态语义验证不仅需覆盖类型兼容性、作用域合法性等复杂维度, 还需处理动态绑定带来的不确定性, 这会导致验证过程的复杂度呈指数级增长^[26]. 为平衡测试完整性与效率, 本文在方法设计中未引入语义验证步骤, 通过语法和语义正确性前置保障与运行时异常捕获的方式, 在确保测试效率的同时, 间接筛选出有效的测试程序.

2.3 覆盖率引导的变异规则选择

本文第 2.2 节已对基于 JIT 优化策略的变异规则进行了介绍, 那么如何将它们的优势进行聚合, 提升 PjitFuzz 的整体覆盖率, 也是前文提到的挑战之一, 即如何生成能够激活 Python JIT 编译器内部优化策略的测试程序. 为了解决这一问题, 本文提出了一种变异规则的选择策略. PjitFuzz 根据程序变体的执行覆盖率, 实时评估每个变异规则的效果, 动态选择下一次变异时要使用的变异规则. 随后, 这些规则会被应用于第 2.2 节所述的变异流程.

JIT 编译器的优化过程涉及复杂的路径和触发条件, 虽然每种变异规则都有其独特的优势, 但单一规则难以覆盖所有潜在路径. 因此, PjitFuzz 需要一个策略来指导多种变异规则的使用. 通过设计变异规则选择策略, PjitFuzz 可以根据运行时的反馈动态调整变异规则的选择, 使其在不同场景下发挥最大效用. 这种动态调整不仅能够更好地利用多种变异规则的组合优势, 提升整体测试效果, 还能确保有限的计算资源被高效利用, 避免不必要的开销. 直观来说, 覆盖率信息可以作为评估变异规则效果的度量标准. 基于此, PjitFuzz 将变异规则的选择问题形式化为经典的多臂老虎机问题 (multi-armed bandit problem)^[27,28]. 多臂老虎机问题是一种在多个选项中进行反复选择, 以最大化累积奖励的决策问题. 在本文中, 每个变异规则代表一个选项, 而覆盖率的提升则作为选择的回报. 面对这一决策优化问题, 汤普森采样^[29]是解决多臂老虎机问题的经典算法. 它是一种基于概率的决策算法, 能够动态调整选择策略, 以最大化长期回报, 且具有高效和易于实现的特点. 因此, PjitFuzz 采用汤普森采样算法作为变异规则的选择策略, 确保在不同测试情境下充分发挥各变异规则的优势. 该算法通过平衡探索 (尝试新规则) 与利用 (选择已知表现好的规则), 能够在测试过程中逐步优化变异规则的选择, 从而最大化覆盖率的提升.

如公式 (1) 所示, PjitFuzz 为每种变异规则维护一个 Beta 分布 $Beta(\alpha_j, \beta_j)$, 其中 j 表示不同变异规则序号, α_j 和 β_j 分别表示第 j 个变异规则成功 (提升覆盖率) 和失败 (未提升覆盖率) 的次数. 在每次迭代中, 从每个 Beta 分布中采样一个 θ_j 值, 并选择采样值最高的变异规则应用于当前程序变体, 具体如公式 (2) 所示.

$$\theta_j \sim Beta(\alpha_j, \beta_j), \forall j \in \{1, 2, \dots, N\} \quad (1)$$

$$indexMut = \arg \max_j \theta_j \quad (2)$$

在应用选定的变异规则后, 记录新的代码覆盖率, 并根据覆盖率的变化更新对应的 Beta 分布. 如果覆盖率提升, 则 α_j 加 1; 否则, β_j 加 1. PjitFuzz 在测试初期广泛尝试各种变异规则, 充分挖掘其潜力, 而在测试后期, 则更倾向于选择效果最优的规则. 通过这种方式, PjitFuzz 不仅能够聚合不同变异规则的优势, 还能生成多样化的测试程

序, 增强对 JIT 编译器不同优化路径的覆盖能力.

2.4 校验和代码块插入的策略

在编译器测试中, 比较不同编译器或编译选项下输出结果的一致性, 对验证编译器的正确性十分必要. 在 Java 虚拟机的测试工作中, Chen 等人^[30]引入了校验和机制, 用于检测内存中静默数据损坏的情况. 在 GCC 测试中, Csmith^[31]在生成的程序中加入了校验和机制, 帮助比较不同编译器或编译选项下的执行结果, 从而发现输出不一致的缺陷. 然而, 目前针对 Python JIT 编译器, 缺乏类似的工具, 且少有研究关注 Python 程序在变异测试中的输出一致性. 本文进一步认识到, 通过在 JIT 程序变体中插入计算校验和的代码块, 可以有效追踪变量值的变化, 从而增强检测的有效性. 首先, 插入计算校验和的代码块不会破坏程序的语义正确性, 也不会影响原始代码的逻辑; 其次, 这种方法增加了额外的验证层, 能够检测变量值的细微变化, 甚至是执行顺序的变化, 从而提升了 PjitFuzz 在细粒度缺陷检测方面的能力. 因此, 本文设计了一种校验和代码块插入策略. PjitFuzz 收集程序不同位置的变量信息, 并在程序的关键位置插入计算校验和的代码块. 在程序执行过程中, 变量的哈希值会被依次累加, 最终程序输出一个字符串形式的哈希校验信息.

算法 1 描述了对 JIT 程序变体插入校验和代码块的流程. 首先, PjitFuzz 初始化两个空集合 *globalVars* 和 *localVars* (第 1, 2 行), 分别用于存储全局变量和局部变量信息. 接着, PjitFuzz 将输入的 JIT 程序变体解析为抽象语法树 (第 3 行). 然后, PjitFuzz 遍历抽象语法树的每一个节点 (第 4 行), 根据节点的类型采取不同的处理方式. 对于赋值语句节点, PjitFuzz 通过 *CollectGlobalVars* 函数收集全局变量信息 (第 5–7 行). 当遇到函数定义节点时, PjitFuzz 通过 *CollectLocalVars* 函数收集该函数节点中的局部变量信息 (第 8, 9 行). 在函数内部, PjitFuzz 会在每个 *return* 语句之前插入计算校验和的代码块, 用于累加函数返回时的局部变量的校验和 (第 10–12 行). 这一步骤的目的是确保在函数通过 *return* 语句显式结束时, 能够捕获局部变量的最终值并计算其校验和. 然而, 函数结束的位置不一定总是在 *return* 语句处, 某些函数可能顺序执行到其结尾处而隐式结束. 因此, 为了尽可能覆盖所有可能的函数结束情况, PjitFuzz 还会在函数结尾插入类似的代码块, 以对所有局部变量进行校验和计算 (第 13 行). 在处理完所有节点后, PjitFuzz 会在文件结尾插入计算全局变量校验和的代码块 (第 16 行). 最后, PjitFuzz 插入打印校验和的语句块 (第 17 行), 并返回插入代码块后的程序变体 (第 18 行). 在 JIT 程序变体中插入校验和代码块, 可以增强 PjitFuzz 在细粒度上检测输出不一致缺陷的能力.

算法 1: 校验和代码块插入.

输入: JIT 程序变体 *JITvariant*, 全局变量集合 *globalVars*, 局部变量集合 *localVars*;

输出: 程序变体 *variant*.

```

1. globalVars ← ∅
2. localVars ← ∅
3. tree ← ParseToAST(JITvariant)
4. for each node in tree do
5.   if node is Assignment then //判断节点是否为赋值语句
6.     globalVars ← CollectGlobalVars(node)
7.   end if
8.   if node is FunctionDef then
9.     localVars ← CollectLocalVars(node)
10.    for each returnStmt in node do //遍历函数节点中所有的 return 语句
11.      InsertChecksumCode(returnStmt, localVars)
12.    end for
13.    InsertChecksumCode(node.end, localVars)

```

14. **end if**
15. **end for**
16. *InsertChecksumCode(file.end, globalVars)*
17. *InsertPrintCode(checksumValue)*
18. **return variant**

为了直观展示算法 1 对 JIT 程序的修改效果, 图 5 提供了插入代码块后的程序示例. 其中, 未着色的部分表示插入之前的原始代码. 具体修改如下: PjitFuzz 首先导入了第三方库 `xxhash` 并初始化了一个哈希对象 (第 1 行). 接着, PjitFuzz 定义了一个递归更新函数, 该函数能够根据传入变量的类型 (例如整数、字符串、列表、字典等) 将其转换为哈希对象, 并累加到校验和中 (第 2–5 行). 在函数内部, PjitFuzz 在 `return` 语句之前插入了计算校验和的代码块, 以确保在函数返回之前更新校验和 (第 13, 14 行). 此外, PjitFuzz 在函数结尾插入了计算校验和的代码块 (第 18, 19 行). 对于全局变量, PjitFuzz 在整个 Python 文件的结尾插入了计算校验和的代码块 (第 21, 22 行). 最后, PjitFuzz 插入了打印校验和的语句块 (第 23 行).

```

1 checksum = xxhash.xxh64(b'') # 初始化校验和为一个哈希对象
2 def update(value): # 定义更新校验和的函数
3     if isinstance(value, int): # 根据不同类型, 递归更新校验和
4         checksum.update(value)
5     ...
6 v0 = [1, "aaa", (0, 92), ...] # 全局变量
7 v1 = {"key1": "value1", "key2": "value2"} # 全局变量
8 ...
9 def function():
10     v2 = 2 # 局部变量
11     ...
12     if param:
13         for value in [v2]:
14             update(value) # 在return语句之前更新校验和
15         return True
16     ...
17     v3 = "bbb" # 局部变量
18     for value in [v2, v3]:
19         update(value) # 在函数结尾更新校验和
20 function()
21 for value in [v0, v1]: # 全局变量
22     update(value) # 在文件结尾更新校验和
23 print(checksum.hexdigest()) # 打印最终的校验和

```

图 5 插入校验和代码块后的程序示例

2.5 缺陷检测

PjitFuzz 采用了差分测试技术^[32], 这是一种广泛应用于软件测试领域的技术. 它通过比较同一程序在不同 JIT 编译选项下的执行结果来检测潜在的缺陷. 图 6 展示了差分测试的具体流程.

在不同的编译选项下, JIT 编译器执行程序后的结果差异主要体现在两个方面: 操作系统返回的状态码和程序输出的校验和. 以 Linux 操作系统为例^[33], 当 JIT 编译器执行程序结束后, 操作系统会返回相应的状态码. 例如, 状态码 0 表示程序正常执行并结束; 状态码 1 通常表示程序发生了非特定错误; 而状态码 139 则表示段错误. 接下来, 本文将根据图 6 所示的差分测试具体流程介绍崩溃、行为不一致和输出不一致的缺陷检测方法.

(1) 崩溃缺陷检测: 由于崩溃缺陷会导致程序异常终止并触发操作系统返回预定义的状态码, 而系统定义的状态码集合是有限且确定的, 因此可以通过监测和匹配这些特定状态码来实现快速识别.

(2) 行为不一致缺陷检测: 当操作系统返回的状态码不属于崩溃类的状态码时, 需进一步对比不同编译选项下的状态码一致性. 若状态码不一致, 则表明至少有一个编译选项下的 JIT 编译器存在缺陷, 此类缺陷被归类为行为不一致缺陷. 例如, S_1 编译选项下的状态码为 0, 而 S_n 选项下的状态码为 1. 该差异表明程序在 S_n 编译选项下无法正常执行结束, JIT 编译器可能存在缺陷.

(3) 输出不一致缺陷检测: 如果状态码相同且为 0 (即程序正常执行结束), 但程序输出的校验和不一致, 则说明至少有一个编译选项下的 JIT 编译器存在缺陷, 本文将此类缺陷归类为输出不一致缺陷. 例如 S_1 编译选项下程序输出的校验和为字符串“ef46db3751d8e999”, 而 S_n 选项下的校验和为“6399602edfe1ade4”. 该差异表明程序在执行过程中某一变量值出现了异常, JIT 编译器可能存在缺陷.

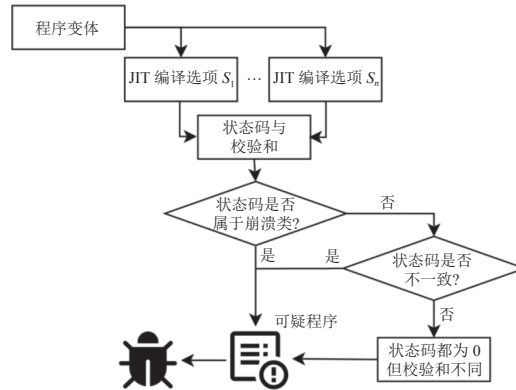


图 6 差分测试具体流程

PjitFuzz 会自动保存检测出的可疑程序及其对应的状态码与校验和. 需要注意的是, 某些细微的结果差异可能是由优化方式或运行机制本身的良性差异引起的, 这通常属于预期行为. 因此, 本文还对检测结果进行了进一步处理, 包括去除重复和无效的缺陷以及对测试程序进行约简, 以支持开发人员的缺陷定位和修复工作.

在讨论了每个组件之后, 接下来介绍算法 2 中 PjitFuzz 的一般工作流程. 首先, PjitFuzz 对收集的 Python 程序进行预处理, 得到种子程序集合 (第 2 行). 接着, PjitFuzz 选择一个种子程序 s , 获取其覆盖率, 并初始化汤普森采样的 $Beta$ 分布参数 α 和 β (第 3–5 行). 随后, PjitFuzz 在内层循环中选取最优的变异规则对种子程序 s 进行变异, 得到 JIT 程序变体, 然后插入计算校验和的代码块, 生成程序变体 (第 6–10 行). 在差分测试阶段, PjitFuzz 使用不同编译选项的 JIT 编译器, 依次执行程序变体, 并记录得到的状态码与校验和 (第 12–15 行). PjitFuzz 根据差分测试得到的状态码与校验和来判断是否产生了崩溃或不一致缺陷, 记录相应的结果信息和可疑程序 (第 16–18 行). 最后, PjitFuzz 根据程序变体的执行情况更新 $Beta$ 分布参数 α 和 β (第 19–24 行), 用于引导下一次迭代选择变异规则.

算法 2. PjitFuzz.

输入: Python 程序集合 set , 最大变异次数限制 $limit$, JIT 编译选项集合 $configs$;

输出: 差分测试结果 $results$.

1. $results \leftarrow \emptyset$
2. $seeds \leftarrow PreProcess(set)$
3. **for each** s **in** $seeds$ **do**
4. $\alpha, \beta \leftarrow [1,1,1,1,1], [1,1,1,1,1]$ //初始化 $Beta$ 分布的参数 α 和 β
5. $maxCov \leftarrow GetCoverage(s)$
6. **for** $num = 1$ **to** $limit$ **do**
7. $\theta \leftarrow SampleBeta(\alpha, \beta)$

```

8.   indexMut ← argmax( $\theta$ )
9.   JITvariant ← Mutate(s, indexMut)
10.  variant ← InjectChecksum(JITvariant)
11.  res ←  $\emptyset$ 
12.  for each config in configs do //使用不同的 JIT 编译选项进行差分测试
13.      (status, checksum) ← Execute(variant, config)
14.      res ← res  $\cup$  (status, checksum) //得到状态码与校验和
15.  end for
16.  if HasCrash(res) or HasDifference(res) then
17.      results ← results  $\cup$  (res, variant)
18.  end if
19.  currCov ← GetCoverage(variant)
20.  if currCov > maxCov then
21.       $\alpha$ [indexMut] ←  $\alpha$ [indexMut] + 1 //更新参数  $\alpha$ 
22.      maxCov ← currCov
23.  else
24.       $\beta$ [indexMut] ←  $\beta$ [indexMut] + 1 //更新参数  $\beta$ 
25.  end if
26.  end for
27. end for
28. return results

```

3 实验分析

3.1 实验设置与研究问题

(1) 实验设置

本文选择两个主流的 Python JIT 编译器 PyPy 和 Numba 作为测试对象. 为了构建 Python 种子数据集, 本文共收集了来自 GitHub 上与“Python”相关的 2000 个开源存储库, 以及 4 个热门存储库 Issues 页面中的 Python 代码片段, 具体的收集流程详见第 2.1 节. 经过语法检查和 CPython 执行验证后的程序预处理, 最终得到的数据集共包含 58521 个种子程序. 在覆盖率统计方面, 虽然 PyPy 的源码由 RPython 与 C 混合编写, 但是其 JIT 核心部分大部分由 RPython 编写, 这导致传统覆盖率工具, 如 llvm-cov、gcov 等, 无法准确收集编译器 JIT 部分的覆盖率信息, 因此本文选择在编译器核心 JIT 源码部分进行插桩统计. 具体而言, 在 JIT 编译器源码的函数定义中插入 RPython 调试代码, 用以记录函数的调用情况. 通过统计生成的日志信息, 本文能够分析模糊测试过程中哪些函数被执行. 对于 Numba 编译器的插桩设计也是同样的思路. 此外, 本文设置每个种子程序的最大变异次数为 30 次.

本文所有实验均在一台高性能 Linux 服务器上完成. 该服务器搭载 32 核、64 线程的英特尔金牌至强处理器, 主频 2.70 GHz, 配备 256 GB 内存, 运行 Ubuntu 20.04 x86_64 操作系统.

(2) 研究问题

为了评估基于覆盖率引导的 Python JIT 编译器缺陷检测方法 PjitFuzz 的有效性, 本文设计了以下 3 个研究问题 (RQ).

- RQ1: PjitFuzz 能否有效地检测出真实的 Python JIT 编译器缺陷?
- RQ2: PjitFuzz 与现有的 Python 解释器模糊测试方法相比, 能否更有效地检测 Python JIT 编译器缺陷?

• RQ3: PjitFuzz 的 3 个关键组件对缺陷检测的有效性有何影响?

RQ1 旨在评估 PjitFuzz 在两个流行的 Python JIT 编译器 PyPy 和 Numba 中发现缺陷的能力. RQ2 的重点是评估 PjitFuzz 与现有的 Python 解释器模糊测试方法 FcFuzzer 和 IFuzzer 相比, 在检测 Python JIT 编译器缺陷方面的有效性. RQ3 则通过消融实验, 分析了基于 JIT 优化策略的程序变异、覆盖率引导的变异规则选择以及校验和代码块插入这 3 个关键组件对 PjitFuzz 缺陷检测能力的贡献, 以验证每个组件的必要性. 接下来, 本文将依次详细介绍这 3 个研究问题的实验结果.

3.2 实验结果与分析

RQ1: PjitFuzz 能否有效地检测出真实的 Python JIT 编译器缺陷?

为了评估 PjitFuzz 检测真实 Python JIT 编译器缺陷的能力, 本文于 2024 年 6 月–2025 年 2 月进行了为期 8 个月的实验, 每轮测试持续约 7 天. 在每轮测试结束后, 如果 JIT 编译器有新版本发布, 本文将更新实验所用版本并继续测试, 以确保测试结果的时效性, 避免向开发人员提交重复的缺陷.

在给开发人员提交缺陷报告之前, 本文将对能够触发 JIT 编译器缺陷的可疑程序进行一系列处理. 首先, 通过精确提取错误信息和程序代码中的关键字, 并结合历史缺陷数据进行匹配分析, 以有效识别并去除重复缺陷. 接着, 进一步分析程序代码, 排除由随机行为、并发行为或 JIT 编译器与 CPython 标准解释器之间实现差异所导致的无效缺陷. 最后, 对测试程序进行自动化约简, 通过逐步删除无关代码, 生成最小化的测试用例, 从而帮助开发人员更高效地修复缺陷.

在为期 8 个月的实验中, PjitFuzz 发现并报告了 16 个问题, 其中 12 个缺陷得到确认, 其余 4 个因重复、无效或其他原因未获确认, 已确认 12 个缺陷中有 2 个缺陷已被修复. 这些已确认的缺陷详情见表 2. 在 12 个已确认的缺陷中, 崩溃、输出不一致和行为不一致缺陷各有 4 个. 为了更清晰地展现 Python JIT 编译器缺陷特征以辅助开发者优化, 我们对这些缺陷进行了分析. 从缺陷表现来看, 输出不一致的缺陷主要出现在 Numba 中, 集中在“信号处理、向量化、动态模块 pickle”等场景. 而行为不一致的缺陷则主要集中在 PyPy 中, 多发生在 GC 测试、递归挂起、深度递归等复杂控制流与内存管理场景. 进一步从缺陷根源剖析, 这种差异可能源于 PyPy 和 Numba 在设计目标、实现方式和应用场景上的显著不同. Numba 专注于数值计算和科学计算领域, 通过将 Python 代码编译为机器码, 优化数值计算中的循环、矩阵运算等高性能操作, 这类优化策略在提升计算效率的同时, 容易在数值精度处理、运算顺序调整等方面出现偏差, 从而引发输出不一致问题. PyPy 作为一个通用的 Python 解释器, 旨在通过 JIT 编译技术提升 Python 程序的整体执行效率, 并兼容所有 Python 语言特性. PyPy 的优化重点不仅局限于数值计算, 而是广泛涉及 Python 程序的各个方面, 包括控制流、内存管理、垃圾回收等, 这使得 PyPy 在优化过程中可能会对不同类型的操作产生不同的执行行为. 这种差异体现了两种 JIT 编译器在不同优化目标和应用场景下的特点.

表 2 已确认的 Python JIT 编译器缺陷

编译器	缺陷编号	缺陷类型	缺陷描述
PyPy	4989	崩溃	The <code>_curses</code> module's exception crash on PyPy
	5011	输出不一致	<code>Pickle.dumps()</code> 's results are different
	5019	行为不一致	JIT causes <code>test_cyclic_gc</code> to fail with <code>function_threshold=1</code> in PyPy
	5021 (已修复)	行为不一致	Program's recursive call causes terminal to hang indefinitely
	5026	崩溃	RPython traceback and crash with custom bytecode modification
	5029 (已修复)	崩溃	Segmentation fault when calling <code>__setstate__</code> on <code>reversed()</code> iterator in PyPy
Numba	5103	行为不一致	Fault in PyPy when running deep recursion with JIT disabled
	9824	输出不一致	Signal handling issue when importing Numba
	9857	崩溃	Segmentation fault when using recursive generator with Numba JIT (<code>NUMBA_OPT=0</code>)
	9877	行为不一致	Numba with <code>@JIT</code> silently handles float-to-integer conversion in slice indices
	9884	输出不一致	Numba JIT function returns incorrect result when vectorized
	9974	输出不一致	<code>@JIT</code> causes unexpected success in pickling functions from unregistered dynamic modules

接下来将讨论 PjitFuzz 检测的两个缺陷案例, 并展示基于 JIT 优化策略的程序变异是如何发现这两个缺陷的.

• 缺陷案例 1: 图 7 展示了 PjitFuzz 发现的 PyPy 的一个输出不一致缺陷 (#5011, 详见 <https://github.com/pypy/pypy/issues/5011>). 需要注意的是, 为直观展示变异操作带来的效果, 图 7 未展示插入校验和的代码块. 在第 6 和 7 行, PjitFuzz 基于“常量折叠”的变异规则, 将字符串 i 替换为“空字符串+ i ”的拼接形式. 该变异操作会导致 JIT 编译器在循环中将“空字符串+ ab ”进行常量折叠, 将其优化为一个常量字符串. 这导致 pickle 在序列化时会重新生成该字符串的引用, 使得禁用 JIT 编译和启用 JIT 编译两种不同编译模式下的输出不一致. 本文成功检测到该问题并报告给开发人员, 开发人员将问题标记为“milestone”, 计划在未来的版本中修复.

```

1 import pickle
2 options = ['ab', 'ab', 'ab', 'ab', 'ab', 'ab', 'ab', 'ab']
3 temp = []
4 for i in options:
5     args = []
-6     args.append(i)
+7     args.append(' ' + i)
8     temp += args
9 data_bytes = pickle.dumps(temp)
10 print(data_bytes)

```

图 7 PyPy 的输出不一致缺陷 (#5011)

• 缺陷案例 2: 图 8 展示了 PjitFuzz 发现的 Numba 的一个行为不一致缺陷 (#9877, 详见 <https://github.com/numba/numba/issues/9877>). 本文通过基于“函数内联”的变异规则, 将第 4 行的简单赋值语句替换为第 5–9 行的函数调用. 变异后的程序在禁用 JIT 编译的选项下运行时会产生错误信息“TypeError: slice indices must be integers or none or have an __index__ method”, 而在使用 Numba 编译执行时却能够执行结束且没有异常提示. 该缺陷本质上源于 Numba 在优化过程中对切片操作的隐式处理, Numba 会自动将浮点数转换为整数. 然而, 这种隐式转换可能导致不一致的行为, 尤其是在浮点类型与整数类型的运算交互时. 开发人员通常会依赖于 Python 本身的错误提示, 但 JIT 编译器却隐藏了这一问题, 从而给开发人员带来了潜在的安全隐患. 目前, 该问题已被开发人员确认, 并标记为“bug - incorrect behavior”, 正在等待后续修复.

```

1 import numpy as np
2 from matplotlib import pylab
3 Fp = np.fft.fft(pylab.hamming(10), n=2**10)
-4 result = np.hstack((Fp[:len(Fp)/2], np.zeros(2**12), Fp[len(Fp)/2:]))
+5 from numba import jit
+6 @jit
+7 def inline(x, n):
+8     return np.hstack((x[:len(x)/2], np.zeros(n), x[len(x)/2:]))
+9 result = inline(Fp, 2**12)
10 p2 = np.fft.ifft(result)
11 print(p2)

```

图 8 Numba 的行为不一致缺陷 (#9877)

回答 RQ1: PjitFuzz 在开源 Python JIT 编译器中共检测到 16 个缺陷, 其中 12 个已获确认. 这些结果表明, PjitFuzz 能够有效地检测真实的 Python JIT 编译器缺陷.

RQ2: PjitFuzz 与现有的 Python 解释器模糊测试方法相比, 能否更有效地检测 Python JIT 编译器缺陷?

本文从缺陷检测能力、执行覆盖率以及程序生成有效性这 3 个维度进行评估. 根据本文的调研, 目前尚未有专门用于检测 Python JIT 编译器缺陷的工具. 因此, 本文选择最接近的解释器测试工具 FcFuzzer 和 IFuzzer 作为基线方法. FcFuzzer 基于骨架程序枚举框架, 通过生成多样化的函数调用程序并结合差分测试, 来检测 Python 解释器中的潜在缺陷. IFuzzer 引入了树形结构和支配关系来优化程序生成过程, 确保生成的非等价程序具备唯一性和有效性. 通过动态剪枝技术和差分测试策略, 进一步提高测试程序的多样性和缺陷检测能力. 需要注意的是, 本研究问题中的实验是在 PyPy 7.3.16 和 Numba 0.57.0 的旧版本上进行的, 这是因为旧版本的 JIT 编译器存在更多缺陷, 这有助于对比不同方法的缺陷检测能力. 为了确保对比的公平性, 本文为每种缺陷检测方法设置了统一的测

试时间 48 h. 对于检测到的缺陷, 本文首先在最新版的 JIT 编译器上尝试复现. 如果缺陷在新版本中已经不存在, 则视为该缺陷已被修复; 如果缺陷仍然存在, 本文将对其进行重复缺陷验证、无效缺陷检查以及程序约简, 最后将结果打包并提交给开发团队.

图 9 展示了 PjitFuzz 与基线方法在 48 h 内发现的缺陷数量. 可以看到, 在相同时间内 PjitFuzz 发现的缺陷数量多于两种基线方法. PjitFuzz 共发现了 5 个缺陷, 而 FcFuzzer 和 IFuzzer 分别只发现了 2 个和 3 个缺陷, PjitFuzz 发现的缺陷数量与两种基线方法发现的缺陷数量之和相当. 此外, PjitFuzz 发现的 5 个缺陷中包括 1 个崩溃缺陷、2 个输出不一致缺陷和 2 个行为不一致缺陷. 相比之下, FcFuzzer 仅发现了 1 个崩溃缺陷和 1 个行为不一致缺陷, 而 IFuzzer 则仅检测到 3 个崩溃缺陷, 未发现其他类型的缺陷. 图 10 探究 PjitFuzz 与基线方法在缺陷发现上的重叠关系. 具体而言, 3 种方法共同发现 1 个崩溃缺陷; PjitFuzz 与 FcFuzzer 共同发现了 1 个行为不一致缺陷. 此外, IFuzzer 检测到 2 个崩溃缺陷, 而 PjitFuzz 发现了 3 个唯一缺陷.

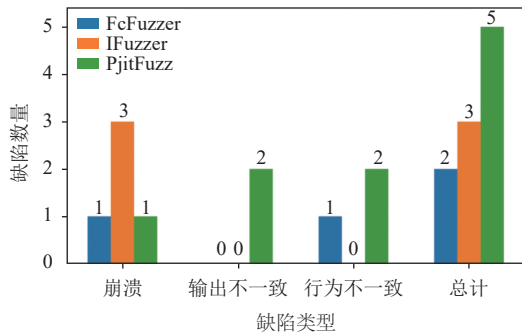


图 9 PjitFuzz 与基线方法在 48 h 内发现的缺陷数量

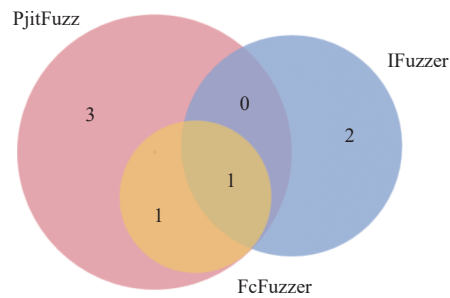


图 10 PjitFuzz 与基线方法在 48 h 内发现的缺陷重叠关系

本文对这些结果进行了详细分析. 首先, FcFuzzer 和 IFuzzer 主要集中于解释器层面的缺陷检测, 未涉及 JIT 编译器的深层优化逻辑, 因此无法生成触发相关优化策略的测试程序, 导致难以检测到 JIT 编译器缺陷. 本文方法 PjitFuzz 设计了 5 种基于 JIT 优化策略的变异规则, 并结合覆盖率信息对这些规则进行动态选择, 从而能够生成多样化的测试程序, 全面探索 JIT 编译器的优化空间, 从而更好地检测潜在缺陷. 其次, 两种基线方法难以检测非崩溃类缺陷. 为了避免耗费大量人力检查误报的不一致缺陷, IFuzzer 主要聚焦于检测导致程序崩溃的缺陷, 而忽略了非崩溃类缺陷. 针对 IFuzzer 发现而 PjitFuzz 没有发现的 2 个崩溃缺陷的原因本文也进行了分析. IFuzzer 基于树结构的骨架程序枚举特性, 能保留控制流依赖关系, 可触达 PyPy 中 socket 库协议处理、ctypes 跨模块调用等 JIT 优化无关的部分. 而 PjitFuzz 聚焦于 Python JIT 编译器的特定优化规则, 测试范围受限于与 JIT 优化相关的路径, 因此未能检测到这 2 个崩溃缺陷. 尽管 FcFuzzer 偶尔能够检测出行为不一致缺陷, 但其差分测试仅通过比较两种不同解释器的执行结果来实现, 局限于解释器层面. 相比之下, PjitFuzz 针对 JIT 编译器设置了 3 种甚至更多的编译选项进行差分测试, 从而能够更有效地检测出 Python JIT 编译器的行为不一致缺陷. 此外, PjitFuzz 还能够检测出两种基线方法无法发现的输出不一致缺陷, 这得益于其设计的校验和算法. 通过在测试程序中插入计算校验和的代码块, PjitFuzz 能够有效检测细粒度的输出不一致缺陷. 以上这些设计使得 PjitFuzz 能够克服两种基线方法的局限性.

此外, 本文利用 Mann-Whitney U 检验^[34]对方法 PjitFuzz 与两种基线方法在缺陷检测数量上的表现差异进行了统计分析. Mann-Whitney U 检验是一种非参数统计检验, 用于比较两组独立样本的分布是否存在显著差异. 本文的零假设 (H_0) 认为方法 PjitFuzz 与基线方法的缺陷数量没有显著差异, 而备择假设 (H_1) 认为存在显著差异. 根据实验统计的数据计算 p 值, 如果 p 值越小, 意味着实验结果与零假设之间的差异越大, 进而本文有更强的证据拒绝零假设, 接受备择假设. p 值是衡量统计显著性的关键指标, 它反映了在零假设成立的前提下, 观察到的数据或更极端情况出现的概率. 实验结果表明, PjitFuzz 与 FcFuzzer、IFuzzer 的 p 值均小于 0.025, 因此有足够的证据拒

绝零假设, 接受备择假设. 这表明至少两组独立样本之间存在统计学上的显著差异, 可以认为 PjitFuzz 在缺陷检测方面显著优于 FcFuzzer 和 IFuzzer, 能够更有效地发现潜在缺陷.

除了评估 PjitFuzz 的缺陷检测能力外, 本文还比较了 PjitFuzz 与两个基线方法的执行覆盖率. 由于目前缺乏针对 PyPy 和 Numba 编译器的覆盖率测试工具, 通过对 JIT 编译器的部分核心源代码进行插桩来实现覆盖率分析. 需要注意的是, 本文仅关注 JIT 编译器优化相关的核心源码, 而不包括解释执行的源码. 具体而言, 针对 PyPy 源码^[35]中的 rpython/jit 目录以及 Numba 源码中^[36]的 numba/core 目录下的代码进行了插桩. 在源码的函数入口处插入调试语句, 用以记录函数的调用情况, 并重新编译了 JIT 编译器. 通过统计模糊测试过程中生成的日志信息, PjitFuzz 能够分析哪些函数被覆盖. 为避免实验复杂性, 仅关注函数覆盖率信息, 函数覆盖率的实验结果如图 11 所示. 本文的主要关注点是测试的最初 12 h 内收集的覆盖率数据, 因为这一时期覆盖率信息通常会发生最大变化, 之后进入稳定阶段. 从图 11 中可以看出, 无论是在 PyPy 还是 Numba 的测试中, PjitFuzz 始终维持着最高的覆盖率, 明显优于 FcFuzzer 和 IFuzzer. 具体来说, PjitFuzz 的函数覆盖率比 FcFuzzer 提升了 28.23%, 比 IFuzzer 提升了 15.68%. 这种提升表明 PjitFuzz 在探索 JIT 编译器代码空间时更为有效, 能够触及更广泛的代码路径, 从而实现更全面的测试覆盖, 体现了 PjitFuzz 的优越性.

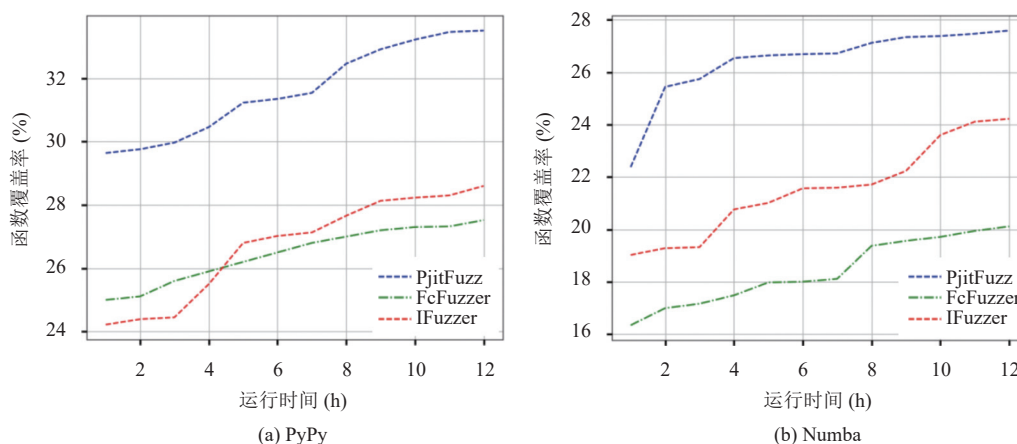


图 11 PjitFuzz 和基线方法函数覆盖率的对比

最后, 本文评估了 PjitFuzz 与两个基线方法生成有效测试程序的能力. 为量化程序生成的有效性, 本文进行 5 次时长 24 h 的对比实验, 统计 PjitFuzz 与两个基线方法在该时长内的总测试程序生成数量、有效测试程序数量及有效测试程序的代码行数, 有效测试程序指可通过语法解析且能在目标编译器上运行的程序. 如表 3 所示, PjitFuzz、FcFuzzer 和 IFuzzer 在 24 h 内平均分别生成总测试程序 253 551、231 690、223 706 个, 有效测试程序占比分别为 86.79%、44.37%、24.05%. 一方面, PjitFuzz 与两个基线方法的有效测试程序数量均随总测试程序数量同步增长, 且 PjitFuzz 的有效测试程序占比比基线方法 FcFuzzer 和 IFuzzer 分别高 42.42%、62.74%; 另一方面, 可以看到 24 h 内 PjitFuzz 生成的有效测试程序的代码行数显著高于基线方法, 体现出语法和语义正确性前置保障的重要性, 即使未引入语义验证, PjitFuzz 仍能通过运行时异常捕获机制筛选出有效程序, 保证实验有效性.

表 3 生成测试程序、有效测试程序及运行代码行数对比

方法	生成测试程序	有效测试程序	有效测试程序占比 (%)	有效测试程序的代码行数
FcFuzzer	231 690	102 798	44.37	1 000 139
IFuzzer	223 706	53 795	24.05	536 535
PjitFuzz	253 551	220 063	86.79	12 768 253

回答 RQ2: 在 48 h 的测试时间内, PjitFuzz 共发现 5 个缺陷, 在缺陷检测能力方面比 FcFuzzer 和 IFuzzer 分别高出 150% 和 66.7%, PjitFuzz 还能发现基线方法未能检测出的非崩溃缺陷. 同时, 在代码覆盖率方面, PjitFuzz 比

FcFuzzer 和 IFuzzer 分别提升了 28.23% 和 15.68%。此外,在程序生成有效性方面,PjitFuzz 比两种基线方法分别提升了 42.42% 和 62.74%。因此,PjitFuzz 在 JIT 编译器缺陷检测方面明显优于现有的 Python 解释器模糊测试方法。

RQ3: PjitFuzz 的 3 个关键组件对缺陷检测的有效性有何影响?

PjitFuzz 的 3 个关键组件分别是基于 JIT 优化策略的程序变异、覆盖率引导的变异规则选择以及校验和代码块插入。为了评估这 3 个组件对有效性影响,本文设计了一组消融实验,通过逐一移除每个组成部分,来确定每部分对 PjitFuzz 整体缺陷检测能力的贡献。具体来说,本文根据控制变量的思想为 PjitFuzz 设计了 3 种变体,即 PjitFuzz_{NJ}, PjitFuzz_{NC} 和 PjitFuzz_{NI}。其中, PjitFuzz_{NJ} 是指不使用基于 JIT 优化策略的变异规则,而是采用朴素的变异规则进行程序变异,这些变异规则包括随机位翻转、随机替换运算符以及随机替换字符等操作; PjitFuzz_{NC} 则去除了覆盖率信息的引导,随机选取基于 JIT 优化策略的变异规则进行变异;而 PjitFuzz_{NI} 则指不在 JIT 程序变体中插入计算校验和的代码块。为更容易区分不同变体的缺陷检测能力,PjitFuzz 及其 3 个变体仍然在 PyPy 7.3.16 和 Numba 0.57.0 旧版本上进行测试。为了确保实验的公平性,设定每轮实验时长为 72 h,进行 5 次实验,并收集和统计测试过程中检测到的缺陷。

表 4 展示了实验结果。第 1 列为不同变体,第 2、3 列分别为 PjitFuzz 及其 3 种变体在 5 次实验中发现的最小缺陷数和最大缺陷数。第 4 列表示 5 次实验中发现的缺陷数量的平均值。实验结果表明,PjitFuzz 在缺陷数量的平均值、最小缺陷数和最大缺陷数上均优于其 3 种变体。本文还利用 Mann-Whitney U 检验对方法 PjitFuzz 与其 3 种变体在缺陷检测数量上的表现差异进行了统计分析,表格第 5 列展示了 Mann-Whitney U 检验的 p 值。可以看到,PjitFuzz 与其 3 种变体的 p 值均小于 0.05,表明 PjitFuzz 在缺陷检测能力上显著优于其 3 种变体。图 12 以最大缺陷数为例展示了 PjitFuzz 各变体在 72 h 内发现的缺陷重叠关系。具体而言,PjitFuzz_{NI} 与 PjitFuzz_{NC} 共同发现 2 个缺陷,PjitFuzz_{NJ} 与 PjitFuzz_{NC} 共同发现 1 个缺陷。此外,PjitFuzz_{NI} 发现 5 个唯一缺陷,PjitFuzz_{NC} 发现 3 个唯一缺陷。值得注意的是,本文发现 PjitFuzz_{NJ} 变体检测出的缺陷数量显著少于 PjitFuzz_{NC}、PjitFuzz_{NI},且并未发现唯一缺陷。这是因为 PjitFuzz_{NJ} 变体没有使用基于 JIT 优化策略的变异规则,而是采用相对简单的基本变异规则,这导致生成的测试程序难以激活 JIT 编译器的深层优化策略。因此,在检测 JIT 编译器缺陷方面,PjitFuzz_{NJ} 的效果不如其他变体。

表 4 PjitFuzz 及其变体检测到的缺陷数量

变体	最小缺陷数	最大缺陷数	平均缺陷数	p值
PjitFuzz _{NJ}	0	1	0.5	0.004
PjitFuzz _{NC}	3	6	5	0.022
PjitFuzz _{NI}	3	7	5.3	0.042
PjitFuzz	5	9	7.3	—

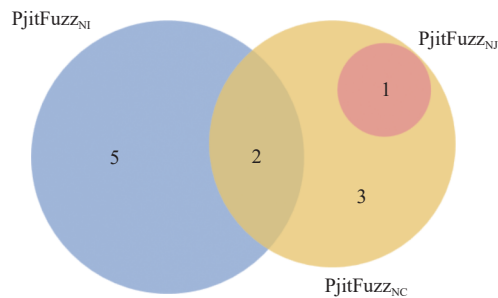


图 12 PjitFuzz 各变体在 72 h 内发现的缺陷重叠关系

回答 RQ3: PjitFuzz 在缺陷检测效果上优于其 3 种变体 PjitFuzz_{NJ}、PjitFuzz_{NI} 和 PjitFuzz_{NC}。基于 JIT 优化策略的程序变异、覆盖率引导的变异规则选择以及校验和代码块插入这 3 个组件都可以不同程度地提高 PjitFuzz 缺陷检测的有效性。

3.3 讨论

尽管 PjitFuzz 在检测 Python JIT 编译器缺陷方面表现出色,但仍然存在一些局限性,这些局限可能影响实验结果的有效性。

首先,种子程序的选择可能存在偏差。本文的种子程序主要来源于 GitHub 上的开源软件仓库以及热门项目的 Issues。尽管通过程序预处理确保了种子的有效性,但 GitHub 上的项目未必能完全涵盖所有 Python 程序的使用场景和特性。例如,热门项目的代码风格和复杂度可能与一些小众或特定领域的程序存在较大差异,从而影响 PjitFuzz 在这些场景下的表现。为了减轻这种偏差带来的威胁,本文可以考虑引入大语言模型,针对特定场景生成具有特定结构的测试程序,从而增强测试的覆盖率和多样性。

其次,JIT 编译器的通用性。本文实验主要针对 PyPy 和 Numba 两种 Python JIT 编译器,它们分别使用 C/C++ 和 Python 实现。尽管本文提出的方法在 PyPy 和 Numba 上取得了显著效果,但其是否能够推广到其他类型的 Python JIT 编译器上仍需进一步验证与改进。根据本文的调研,Python 标准解释器 CPython 3.13 版本新增了一个实验性质的 JIT 编译器,该编译器基于 copy-and-patch 技术实现。因此,未来工作可以考虑将 PjitFuzz 方法扩展到 CPython 3.13 的 JIT 编译器上,以验证其通用性和适用性。

此外,JIT 编译选项的局限性。JIT 编译器的行为受配置参数的影响较大,例如优化级别、内存管理策略等。本文实验采用了默认配置和部分调整过的 JIT 编译选项,但不同的运行环境可能会导致不同的优化触发模式,从而影响测试结果。例如,有些 JIT 编译器的缺陷可能只会特定的 JIT 编译选项下重现,这使得不同的配置选项可能会显著影响缺陷的检测与修复。为了缓解这一问题,未来可以考虑扩展 PjitFuzz 的测试范围,覆盖更多 JIT 编译选项的组合,以全面评估其在不同配置下的缺陷检测能力。

最后,PjitFuzz 的性能开销存在优化空间。PjitFuzz 采用多 JIT 编译选项下的差分测试策略,每个程序变体需在多种配置下重复执行,并收集状态码与校验和信息。这一过程虽然有助于发现非崩溃类缺陷,但也成倍增加了测试执行时间。特别是在覆盖率引导的变异过程中,随着迭代次数的增加,累积的执行开销会显著上升,限制了其在资源受限环境下的可扩展性。未来可以考虑引入分布式测试框架,将多配置执行任务拆分到多节点并行处理,以降低整体执行开销。

4 相关工作

4.1 Python 解释器的缺陷检测

近年来,随着 Python 语言的广泛流行,Python 解释器的自动化测试逐渐成为研究热点。Xia 等人^[10]提出了一个名为 FcFuzzer 的工具,专门用于检测 Python 解释器中的缺陷。传统的骨架程序枚举 (SPE) 方法主要关注变量的使用,而忽视了函数调用的复杂性,导致其在检测与函数调用相关的缺陷时存在局限性。为了解决这一问题,研究人员在 SPE 方法中集成了函数调用的枚举。他们通过定义函数骨架,并采用分阶段的变量和参数枚举策略,从而生成多样化的测试程序。该方法确保了生成的每个测试程序在函数调用和变量使用上的独特性,避免了冗余的等价程序。此外,FcFuzzer 还采用了差分测试技术,通过比较不同解释器的执行结果,检测解释器中的崩溃、行为异常等缺陷。与传统的 SPE 方法相比,FcFuzzer 在检测缺陷的多样性和检测效率上均有显著提升。IFuzzer^[11]则进一步引入树形 SPE 方法,并利用控制流依赖关系来确保程序生成的唯一性和有效性。具体来说,研究人员在 IFuzzer 中引入了树形结构和支配关系来优化程序生成过程。IFuzzer 通过定义程序骨架并利用控制流和数据依赖的关系,确保生成的测试程序具备唯一性和有效性,避免了生成冗余的 α -等价程序。为了进一步提高程序的多样性和质量,IFuzzer 在生成每个测试程序时会进行分阶段的变量和参数枚举,这样不仅增强了测试覆盖范围,还确保了生成的程序能够有效触发潜在缺陷。IFuzzer 还采用了动态剪枝技术,在生成过程中实时过滤那些不满足控制流依赖关系或者存在逻辑错误的程序,从而大幅减少无效测试程序的数量。最后,IFuzzer 通过差分测试策略,比较不同 Python 解释器在执行相同测试程序时的输出结果,有效检测了崩溃类型的缺陷。PyRTFuzz^[37]专门用于检测 Python 运行时系统中的缺陷,采用两级协作的测试方法,结合生成式和变异式测试策略,全面覆盖 Python 解释器和运行时库

的交互. 这种方法能够有效发现运行时系统中的潜在缺陷. 尽管这些方法在 Python 解释器的缺陷检测中取得了显著成果, 但针对 Python JIT 编译器的测试方法尚未得到充分研究. 本文提出的 PjitFuzz 是 Python 领域中专门针对 JIT 编译器的缺陷检测方法.

4.2 JIT 编译器的缺陷检测

目前, 国内外学者针对 JIT 编译器的测试研究主要集中在 JavaScript 引擎和 Java 虚拟机方面. 例如, Fuzzilli^[38]是专注于 JIT 编译器漏洞的测试工具. 它通过设计新的中间表示 (IR) 生成特殊结构的 JavaScript 代码, 以触发 JIT 编译中的潜在漏洞. Jit-Picking^[39]通过比较 JavaScript 引擎的解释器与 JIT 编译器在执行相同代码时的行为差异, 识别优化过程中的逻辑错误并揭示潜在漏洞. FuzzJIT^[40]通过设计输入包装模板, 确保每个测试样本在每次执行时都能激活 JIT 编译器, 并借助增强的测试预言机制, 在执行过程中比较解释器与 JIT 编译路径的差异, 从而自动捕捉执行中的不一致, 揭示因 JIT 优化引起的潜在漏洞. JOpFuzzer^[41]通过探索二维输入空间, 其中一维是种子输入的变化, 另一维是 JIT 编译器的优化选项配置, 结合这两者扩展测试场景, 从而全面挖掘 JIT 编译器的潜在漏洞, 深入测试不同优化路径下的编译行为. JITfuzz^[25]通过采用覆盖率引导的测试框架, 设计优化激活变异器和控制流变异器, 针对特定的优化路径和控制流路径进行变异, 确保 JIT 编译器的代码空间能被有效覆盖, 进而提高漏洞检测效率. Artemis^[42]通过引入编译空间的概念, 模拟所有可能的 JIT 编译路径, 探索不同优化路径下的执行行为, 确保测试覆盖 JIT 编译器的所有优化状态, 并在执行过程中验证每个优化路径的正确性, 从而提供了一种有效的验证方法. 这些研究工作不仅说明了 JIT 编译器测试的重要性, 也为本文的测试工作提供了参考.

5 总 结

本文设计并实现了一种基于覆盖率引导的自动测试框架 PjitFuzz, 用于检测 Python JIT 编译器中的缺陷. PjitFuzz 设计了 5 种基于 JIT 优化策略的变异规则, 并结合覆盖率信息选择这些规则, 生成多样化的测试程序, 有效激活 JIT 优化策略. 此外, PjitFuzz 还针对 JIT 程序变体设计了校验和代码块插入算法, 并通过差分测试方法有效检测非崩溃缺陷. 实验结果表明, PjitFuzz 共检测到 16 个缺陷, 其中 12 个已得到开发人员的确认. 与现有两种 Python 解释器测试方法 FcFuzzer 和 IFuzzer 相比, PjitFuzz 的缺陷检测能力分别高出 150% 和 66.7%, 代码覆盖率分别高出 28.23% 和 15.68%, 生成测试程序有效率分别高出 42.42% 和 62.74%. 此外, 本文还验证了基于 JIT 优化策略的程序变异、覆盖率引导的变异规则选择以及校验和代码块插入这 3 个关键组件都能在不同程度上提高 PjitFuzz 缺陷检测的有效性.

虽然 PjitFuzz 在 Python JIT 编译器的函数覆盖、缺陷检测能力等方面都比现有的工具有优势, 但仍然存在一些局限性, 我们希望在未来能够解决这些局限性. 结合 JIT 编译器测试领域的技术趋势及现有研究的拓展方向, 未来将围绕以下维度展开深化与延伸. 面对种子程序场景覆盖局限问题, 我们计划引入大语言模型生成特定结构的测试程序, 突破开源项目种子的场景边界. 此外, 我们将扩展 PjitFuzz 至 CPython 3.13 的 JIT 编译器作为关键性工作, 进一步验证与完善 PjitFuzz 在不同 Python JIT 编译器实现中的通用性设计. 针对 JIT 编译选项覆盖不足, 下一步工作可从扩展测试维度模拟真实环境中复杂的参数组合场景进行, 更全面评估 PjitFuzz 在复杂运行环境下的缺陷检测稳定性与有效性. 最后, 本文后续可尝试引入分布式测试框架进行 PjitFuzz 的性能开销优化.

References

- [1] van Rossum G, Drake Jr FL. An Introduction to Python. Bristol: Network Theory Ltd., 2011.
- [2] Rossum G. Python Reference Manual. Amsterdam: Centrum voor Wiskunde en Informatica, 1995.
- [3] Pérez F, Granger BE, Hunter JD. Python: An ecosystem for scientific computing. *Computing in Science & Engineering*, 2011, 13(2): 13–21. [doi: 10.1109/MCSE.2010.119]
- [4] Python. 2004. <https://www.python.org>
- [5] Thomassen EW. Trace-based just-in-time compiler for Haskell with RPython [MS. Thesis]. Norwegian: Norwegian University, 2013.
- [6] Klees G, Ruef A, Cooper B, Wei SY, Hicks M. Evaluating fuzz testing. In: Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. Toronto: ACM, 2018. 2123–2138. [doi: 10.1145/3243734.3243804]

- [7] Han H, Cha SK. IMF: Inferred model-based fuzzer. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. Dallas: ACM, 2017. 2345–2358. [doi: [10.1145/3133956.3134103](https://doi.org/10.1145/3133956.3134103)]
- [8] Cha SK, Woo M, Brumley D. Program-adaptive mutational fuzzing. In: Proc. of the 2015 IEEE Symp. on Security and Privacy. San Jose: IEEE, 2015. 725–741. [doi: [10.1109/SP.2015.50](https://doi.org/10.1109/SP.2015.50)]
- [9] Woo M, Cha SK, Gottlieb S, Brumley D. Scheduling black-box mutational fuzzing. In: Proc. of the 2013 ACM SIGSAC Conf. on Computer & Communications Security. Berlin: ACM, 2013. 511–522. [doi: [10.1145/2508859.2516736](https://doi.org/10.1145/2508859.2516736)]
- [10] Xia XM, Feng Y. Detecting interpreter bugs via filling function calls in skeletal program enumeration. In: Proc. of the 34th IEEE Int'l Symp. on Software Reliability Engineering. Florence: IEEE, 2023. 612–622. [doi: [10.1109/ISSRE59848.2023.00066](https://doi.org/10.1109/ISSRE59848.2023.00066)]
- [11] Xia XM, Feng Y, Shi QK, Jones JA, Zhang XY, Xu BW. Enumerating valid non-alpha-equivalent programs for interpreter testing. ACM Trans. on Software Engineering and Methodology, 2024, 33(5): 118. [doi: [10.1145/3647994](https://doi.org/10.1145/3647994)]
- [12] Wikipedia. Common subexpression elimination. 2004. https://en.wikipedia.org/wiki/Common_subexpression_elimination
- [13] Wikipedia. Constant folding. 2002. https://en.wikipedia.org/wiki/Constant_folding
- [14] Wikipedia. Inline expansion. 2003. https://en.wikipedia.org/wiki/Inline_expansion
- [15] Wikipedia. Computer algebra. 2004. https://en.wikipedia.org/wiki/Computer_algebra#%20Simplification
- [16] Wikipedia. Loop optimization. 2005. https://en.wikipedia.org/wiki/Loop_optimization
- [17] Bolz CF, Cuni A, Fijalkowski M, Rigo A. Tracing the meta-level: PyPy's tracing JIT compiler. In: Proc. of the 4th Workshop on the Implementation, Compilation, Optimization of Object-oriented Languages and Programming Systems. Genova: ACM, 2009. 18–25. [doi: [10.1145/1565824.1565827](https://doi.org/10.1145/1565824.1565827)]
- [18] Lam SK, Pitrou A, Seibert S. Numba: A LLVM-based Python JIT compiler. In: Proc. of the 2nd Workshop on the LLVM Compiler Infrastructure in HPC. Austin: ACM, 2015. 7. [doi: [10.1145/2833157.2833162](https://doi.org/10.1145/2833157.2833162)]
- [19] Liu D, Feng Y, Yan YY, Xu BW. Towards understanding bugs in Python interpreters. Empirical Software Engineering, 2023, 28(1): 19. [doi: [10.1007/s10664-022-10239-x](https://doi.org/10.1007/s10664-022-10239-x)]
- [20] Wang ZY, Bu DX, Sun AY, Gou SY, Wang Y, Chen L. An empirical study on bugs in Python interpreters. IEEE Trans. on Reliability, 2022, 71(2): 716–734. [doi: [10.1109/TR.2022.3159812](https://doi.org/10.1109/TR.2022.3159812)]
- [21] van der Walt S, Colbert SC, Varoquaux G. The NumPy array: A structure for efficient numerical computation. Computing in Science & Engineering, 2011, 13(2): 22–30. [doi: [10.1109/MCSE.2011.37](https://doi.org/10.1109/MCSE.2011.37)]
- [22] ast—Abstract syntax trees. 2008. <https://docs.python.org/3/library/ast.html>
- [23] Aho AV, Lam MS, Sethi R, Ullman JD. Compilers: Principles, Techniques, and Tools. 2nd ed., Boston: Addison-Wesley Longman Publishing Co., Inc., 2006.
- [24] Ardö H, Bolz CF, Fijalkowski M. Loop-aware optimizations in PyPy's tracing JIT. In: Proc. of the 8th Symp. on Dynamic Languages. Tucson: ACM, 2012. 63–72. [doi: [10.1145/2384577.2384586](https://doi.org/10.1145/2384577.2384586)]
- [25] Wu MY, Lu MH, Cui HM, Chen JJ, Zhang YQ, Zhang LM. JITfuzz: Coverage-guided fuzzing for JVM just-in-time compilers. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 56–68. [doi: [10.1109/ICSE48619.2023.00017](https://doi.org/10.1109/ICSE48619.2023.00017)]
- [26] Yang Y, Milanova A, Hirzel M. Complex Python features in the wild. In: Proc. of the 19th Int'l Conf. on Mining Software Repositories. Pittsburgh: ACM, 2022. 282–293. [doi: [10.1145/3524842.3528467](https://doi.org/10.1145/3524842.3528467)]
- [27] Wu RX, Zhang HY, Kim S, Cheung SC. ReLink: Recovering links between bugs and changes. In: Proc. of the 19th ACM SIGSOFT Symp. and the 13th European Conf. on Foundations of Software Engineering. Szeged: ACM, 2011. 15–25. [doi: [10.1145/2025113.2025120](https://doi.org/10.1145/2025113.2025120)]
- [28] Lee M, Cha S, Oh H. Learning seed-adaptive mutation strategies for greybox fuzzing. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 384–396. [doi: [10.1109/ICSE48619.2023.00043](https://doi.org/10.1109/ICSE48619.2023.00043)]
- [29] Karamcheti S, Mann G, Rosenberg D. Adaptive grey-box fuzz-testing with thompson sampling. In: Proc. of the 11th ACM Workshop on Artificial Intelligence and Security. Toronto: ACM, 2018. 37–47. [doi: [10.1145/3270101.3270108](https://doi.org/10.1145/3270101.3270108)]
- [30] Chen DQ, Messer A, Bernadat P, Fu GR, Dimitrijevic Z, Lie DJF, Mannaru D, Riska A, Milojicic DS. JVM susceptibility to memory errors. In: Proc. of the 1st Java Virtual Machine Research and Technology Symp. Monterey: USENIX, 2001. 67–78.
- [31] Yang XJ, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. In: Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation. San Jose: ACM, 2011. 283–294. [doi: [10.1145/1993498.1993532](https://doi.org/10.1145/1993498.1993532)]
- [32] McKeeman WM. Differential testing for software. Digital Technical Journal, 1998, 10(1): 100–107.
- [33] Errno(3)—Linux manual page. 2025. <https://www.man7.org/linux/man-pages/man3/errno.3.html>
- [34] Nachar N. The Mann-Whitney U: A test for assessing whether two independent samples come from the same distribution. Tutorials in Quantitative Methods for Psychology, 2008, 4(1): 13–20. [doi: [10.20982/tqmp.04.1.p013](https://doi.org/10.20982/tqmp.04.1.p013)]

- [35] GitHub. PyPy. 2005. <https://github.com/pypy/pypy>
- [36] GitHub. Numba. 2014. <https://github.com/numba/numba>
- [37] Li W, Yang HR, Luo XP, Cheng L, Cai HP. PyRTFuzz: Detecting bugs in Python runtimes via two-level collaborative fuzzing. In: Proc. of the 2023 ACM SIGSAC Conf. on Computer and Communications Security. Copenhagen: ACM, 2023. 1645–1659. [doi: [10.1145/3576915.3623166](https://doi.org/10.1145/3576915.3623166)]
- [38] Groß S, Koch S, Bernhard L, Holz T, Johns M. Fuzzilli: Fuzzing for JavaScript JIT compiler vulnerabilities. In: Proc. of the 30th Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2023.
- [39] Bernhard L, Scharnowski T, Schloegel M, Blazytko T, Holz T. JIT-picking: Differential fuzzing of JavaScript engines. In: Proc. of the 2022 ACM SIGSAC Conf. on Computer and Communications Security. Angeles: ACM, 2022. 351–364. [doi: [10.1145/3548606.3560624](https://doi.org/10.1145/3548606.3560624)]
- [40] Wang JJ, Zhang ZY, Liu S, Du XN, Chen JJ. FuzzJIT: Oracle-enhanced fuzzing for JavaScript engine JIT compiler. In: Proc. of the 32nd USENIX Security Symp. Anaheim: USENIX Association, 2023. 1865–1882.
- [41] Jia HX, Wen M, Xie ZF, Guo XC, Wu RX, Sun ML, Chen K, Hai J. Detecting JVM JIT compiler bugs via exploring two-dimensional input spaces. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 43–55. [doi: [10.1109/ICSE48619.2023.00016](https://doi.org/10.1109/ICSE48619.2023.00016)]
- [42] Li C, Jiang YY, Xu C, Su ZD. Validating JIT compilers via compilation space exploration. In: Proc. of the 29th Symp. on Operating Systems Principles. Koblenz: ACM, 2023. 66–79. [doi: [10.1145/3600006.3613140](https://doi.org/10.1145/3600006.3613140)]

作者简介

任志磊, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为智能软件工程, 程序动态追踪.

张子龙, 硕士生, 主要研究领域为软件测试, 缺陷定位.

周志德, 博士, CCF 专业会员, 主要研究领域为智能软件工程, 可信编译, 深度学习优化.

李微微, 硕士生, 主要研究领域为软件测试, 缺陷定位.

江贺, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为智能软件工程, 工业软件测试.