LLMCfuzz:基于大语言模型的航空发动机编译器模糊测试方法

陆 炜[1),2)] 李伟湋[1),2),3)] 施彬彬[4)] 曾俊伟[1),2)] 黄志球[1),2),3)]

[1)](南京航空航天大学计算机科学与技术学院 南京 211106)

[2)](高安全系统的软件开发与验证技术工业与信息化部重点实验室 南京 211106)

[3)](软件新技术与产业化协同创新中心 南京 210023)

[4)](中国航发控制系统研究所 江苏 无锡 214000)

摘 要 航空发动机嵌入式系统对编译器的安全性和稳定性有极高要求。模糊测试是发现其缺陷的关键技术,但现有方法生成的测试程序缺乏多样性,难以触发中后端缺陷,且难以检测危害最大的静默误编译错误。本文提出了一种基于大语言模型的编译器模糊测试方法LLMCfuzz。该方法包括变异提示生成、测试生成和差分测试三个阶段。在变异提示生成阶段,构建包含嵌入式程序和测试套件的种子程序库,通过多样性引导策略选择变异算子并生成变异提示。在测试生成阶段,利用大语言模型生成具有复杂数据流和控制流的变体程序,同时设计变量追踪机制监测静默误编译错误,并通过前端错误反馈优化提示模板。在差分测试阶段,结合随机差异测试与不同优化级别测试以检测中后端缺陷。实验结果表明,LLMCfuzz在行覆盖率上较现有方法提高2.78%至21.08%,并成功发现5种误编译错误,其中包括3种静默误编译错误。

关键词 航空发动机嵌入式系统;编译器缺陷;大语言模型;编译器模糊测试;程序变异

中图法分类号 TP311 **DOI号** 10.11897/SP.J.1016.2025.02875

LLMCfuzz: A Fuzz Testing Method for Aero Engine Compilers Based on Large Language Models

LU Wei[1),2)] LI Wei-Wei[1),2),3)] SHI Bin-Bin[4)] ZENG Jun-Wei[1),2)] HUANG Zhi-Qiu[1),2),3)]

[1)](Collegeof Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106)

[2)](Key Laboratory of Safety-Critical Software Development and Verification, Ministry of Industry and Information Technology, Nanjing 211106)

[3)](Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 210023)

[4)](AECC Aero Engine Control System Institute, Wuxi, Jiangsu 214000)

Abstract The correctness of compilers is a non-negotiable cornerstone of safety for embedded systems within aero-engines, where even minuscule software flaws can precipitate catastrophic failures. In the domain of aerospace software verification, ensuring compiler reliability presents a unique set of formidable challenges. First and foremost, aero-engine compilers must adhere to stringent safety standards like DO-178C, which enforce prohibitions on high-risk language features and demand specialized optimizations for target-specific hardware (e.g., PowerPC).

收稿日期:2025-01-20;在线发布日期:2025-07-25。本课题得到国家科技重大专项(Y2022-V-0001-0027)、国家自然科学基金(6257070451)、上海市中央引导地方科技发展资金(YDZX20233100004008)资助。**陆 炜**,硕士研究生,中国计算机学会(CCF)会员,主要研究领域为智能化软件工程、编译器测试。E-mail:luwei6@nuaa.edu.cn。**李伟湋**(通信作者),博士,副研究员,中国计算机学会(CCF)专业会员,主要研究领域为机器学习、智能化软件工程、高安全软件开发方法研究、软件可靠性。E-mail:liweiwei@nuaa.edu.cn。**施彬彬**,硕士研究生,高级工程师,主要研究领域为高安全嵌入式实时操作系统、编译器验证、航空发动机控制软件。**曾俊伟**,硕士研究生,中国计算机学会(CCF)会员,主要研究领域为智能化软件工程、编译器测试。**黄志球**,博士,教授,中国计算机学会(CCF)杰出会员,主要研究领域为软件工程、软件安全性、形式化方法。

This high degree of customization often introduces subtle defects. Secondly, a critical problem known as silent miscompilation arises, where the compiler generates functionally incorrect machine code without issuing any warnings, posing an extreme threat to system integrity. Thirdly, the entire development process operates within a cross-compilation environment, where testing necessitates a complex workflow of host-side compilation, target-side execution, and verification via peripheral interfaces like serial ports. Existing fuzzing methodologies, both traditional and Large Language Model (LLM) based, exhibit significant deficiencies when applied to this safety-critical context. Conventional fuzzers, such asCsmith, generate programs that often lack the structural complexity to trigger deep optimization bugs and may violate aviation-specific coding standards. Moreover, they are inherently incapable of detecting the most dangerous silent miscompilation errors. While LLMs offer a promising new direction for test case generation, current approaches are generic and have failed to demonstrate efficacy for C compilers, let alone adapt to the unique constraints of the aero-engine cross-compilation workflow. They struggle to produce compliant, semantically rich programs that can effectively stress the compiler's backend. To address these multifaceted challenges, this paper introduces LLMCfuzz, a novel and systematic fuzz testing framework specifically engineered for C cross-compilers and driven by Large Language Models. The methodology of LLMCfuzz is architected into three distinct, synergistic stages. Specifically, in the initial mutation prompt generation stage, we establish a foundational seed library from real-world embedded programs and employ a diversity-guided strategy to systematically select mutation operators, which generate sophisticated prompts to instruct the LLM. Subsequently, during the core test generation stage, LLMCfuzz harnesses LLMs to produce variant programs with intricate data and control flows designed to stress optimization routines while adhering to aviation constraints. Critically, to solve the intractable problem of detecting silent miscompilations, we introduce an innovative variable tracking mechanism. This mechanism directs the LLM to inject non-intrusive monitoring statements, tracking runtime variable values which are then outputted via the target's serial port for cross-environment verification. Furthermore, a self-optimizing feedback loop uses frontend errors to dynamically refine prompt templates. Finally, the differential testing stage employs a two-pronged strategy, combining testing across different compiler versions with testing across various optimization levels to comprehensively uncover backend defects. Experiments conducted on a production-grade aero-engine compiler demonstrate the superior performance of our approach. The results show that LLMCfuzz improves line coverage by 2.78% to 21.08% compared to state-of-the-art methods. Crucially, it successfully identified five previously unknown miscompilation errors, including three critical silent miscompilation errors and two explicit ones, with four being unique discoveries of our method. The proposed framework represents a pioneering application of LLMs to the rigorous domain of safety-critical compiler testing, offering a significant advancement in ensuring the reliability of embedded aerospace software.

Keywords aeroengine embedded systems; compiler defects; large language model; compiler fuzzing; program mutation

1 引 言

编译器是航空发动机嵌入式系统的核心组件之一，其正确性和稳定性直接关系到系统的整体安全。任何编译器错误都可能导致严重的系统故障，甚至引发灾难性事故[1]。例如，F-16战斗机的控制系统曾因编译器生成的错误代码导致飞控指令失效，从而引发了飞行事故。三菱重工在其航空发动机的控制软件中，曾因编译器误编译导致发动机控

制系统出现异常，虽然没有导致直接事故，但该问题的发现促使了进一步的编译器验证工作。

这些真实案例表明，航空发动机编译器的微小错误都可能会导致系统出现致命缺陷，影响飞行器的正常运行，甚至引发灾难性后果。因此，确保航空发动机嵌入式系统中编译器的可靠性与安全性，是保障系统稳定和飞行安全的关键。

不同于传统的C语言编译器，航空发动机编译器具有许多独特之处。首先是严格的安全性与可靠性约束。航空发动机编译器需要遵循DO-178C等航空电子标准，禁止动态内存分配、递归调用等高风险操作，并强制类型安全与边界检查。其次是硬件适配与定制优化需求。航空发动机编译器通常针对PowerPC等嵌入式芯片定制，需要支持低延迟中断处理、代码体积压缩以及硬件加速指令集的优化功能，此类优化可能引入传统测试难以覆盖的静默误编译错误。并且，航空发动机系统需要长时间无故障运行，静默误编译错误可能会导致控制逻辑偏差（如油门指令计算错误），但编译时无显式报错，传统方法对此类错误的检测能力有限。

模糊测试是当前最成熟且广泛使用的编译器测试方法。模糊测试的一般过程是生成大量语法正确的测试程序并将它们输入到不同的编译器中，若编译时发生任何意外或不一致的行为，都表明有编译器出现了错误[2]。最著名的模糊测试方法Csmith[3]和YARPGen[4-5]等，在C语言编译器中已经发现数百个错误。

随着大语言模型的发展，有研究者开始尝试将大语言模型运用于软件测试领域中[6-8]。大语言模型，尤其是代码生成大模型[9-10]，经过大量自然语言和代码数据的训练，具有强大且智能的代码生成能力。通过提示工程[11-13]，大语言模型能够根据给定的任务描述生成特定的测试用例。目前已经有研究将大语言模型应用于编译器模糊测试中[14-16]并且在各种编程语言编译器和深度学习编译器中发现了大量错误。

尽管传统的方法和基于大语言模型的方法都被证实能够发现编译器错误，但它们依然存在缺陷。首先，传统的方法生成的测试程序多样性不足。基于生成的方法Csmith和YARPGen，它们生成符合语法规范的测试程序，但测试程序的多样性受限于手动设计的生成规则；最先进的基于变异的方法GrayC[17]，对种子程序积累微小的变异，从而生成大量不具备触发编译器中后端错误能力的测试程序。其次，几乎所有模糊测试方法都难以检测到危害性最大的静默误编译错误[18]，而只能检测到危害性较小的编译器崩溃错误和显式误编译错误；而且，目前还没有专门针对C编译器设计的基于大语言模型的方法。尽管Fuzz4all[14]是一种通用的基于大语言模型的方法，但是目前Fuzz4all尚未发现任何C编译器的错误。

此外，现有方法均是为通用C语言编译器设计的（Linux或Windows系统下），在航空发动机编译器测试中面临严峻挑战。首先，基于生成的方法生成的测试程序可能违反航空规范（如使用动态内存分配），导致生成大量无效测试程序。其次，航空发动机嵌入式系统使用交叉编译器，测试程序需要在主机中被编译，然后在芯片中上电运行，程序需要依赖串口模块进行输出，现有的所有方法都难以适配。并且，航空发动机编译器针对芯片系统的定制优化逻辑对代码变动高度敏感，需要生成兼具复杂控制流、数据流并且合规的测试程序，然而现有方法生成测试程序的结构较为简单，缺乏对编译器优化缺陷的敏感的数据流与控制流结构。

针对上述问题，本文提出了一种基于大语言模型的航空发动机编译器模糊测试方法LLMCfuzz（**L**arge **L**anguage **M**odel Based C **C**ross-compiler **fuzz**er）。本方法能够生成更加多样化、数据流和控制流更为复杂的测试程序，并且通过变量追踪机制，能够有效检测危害最大的静默误编译错误。首先，LLMCfuzz设计了多种变异算子与相应的变异提示模板，针对当前的种子程序，LLMCfuzz通过多样性引导策略来选择变异算子，并生成相应的变异提示。然后利用大模型生成数据流与控制流更为复杂的变体程序。并且，LLMCfuzz设计了一种变量追踪机制，使用大模型为变体程序插入多个输出语句，并通过串口输出来实现航空发动机编译器需要的跨环境验证，这使得程序能够检测危害最大的静默误编译错误。此外，LLMCfuzz还收集具有前端错误的变体程序及其错误信息，并用它们作为反馈来更新变异提示模板，从而提高大语言模型生成有效测试程序的能力。

本文的主要贡献如下：（1）提出LLMCfuzz方法，首次将大语言模型应用于航空发动机嵌入式编译器的模糊测试，突破了现有方法中规则驱动变异的局限，更高效地生成具有复杂的数据流和控制流结构的测试程序。LLMCfuzz在行覆盖率上较现有方法提高2.78%至21.08%。（2）设计了一种创新的

变量追踪机制，同时追踪全局和局部变量的值，使本方法能够更精确地检测危害最大的静默误编译错误。(3)LLMCfuzz在实际航空发动机编译器中成功发现了5种误编译错误，其中包括3种静默误编译错误、2种显式误编译错误。其中有4种错误为LLMCfuzz的独有发现。

本文第1节介绍编译器模糊测试的相关方法和研究现状；第2节介绍相关工作；第3节介绍本文提出的基于大语言模型的航空发动机编译器模糊测试方法LLMCfuzz；第4节介绍本文的实验设计并对实验结果进行讨论；第5节总结全文。

2 相关工作

编译器模糊测试方法主要分为基于生成的方法[3-5,19-21]和基于变异的方法[17,22-26]。此外，近年大语言模型在软件测试领域的应用[6-8,27-29]也取得了一定进展，包括基于生成与变异结合的模糊测试方法。

基于生成的模糊测试方法一般通过语法规约生成测试程序。Csmith[3]作为开创性的C程序生成器，通过设计C语言规约子集随机生成测试程序，发现了大量C编译器错误。YARPGen[4-5]进一步引入标量优化和循环优化组件，提高了针对性测试能力，弥补了Csmith的部分不足。然而，基于语法规约的方法生成程序的多样性和复杂度受限，难以覆盖编译器的深层特性[30]。为了克服以上局限，基于深度学习的方法如DeepFuzz[20]利用编译器测试套件训练模型生成程序，但生成程序的语法正确率较低，且仅能检测到编译器崩溃错误，而无法有效检测到航空发动机编译器中危害最大的静默误编译错误。

相比传统方法，本文提出的LLMCfuzz利用大语言模型生成测试程序，突破了手工规则的限制。基于精心设计的提示指令，LLMCfuzz生成数据流与控制流更复杂的测试程序，同时通过变量追踪机制来检测静默误编译错误。

基于变异的模糊测试方法通过修改现有程序生成测试程序，主要分为语义保留变异[22-24]和非语义保留变异[17,25-26]。语义保留变异方法如Orion[22]通过随机删除非执行部分代码生成测试。Athena[23]和Hermes[24]进一步拓展了变异操作的范围。然而，这类方法生成的程序语义一致性虽有保障，但测试程序多样性有限，难以触发深层次编译器缺陷。非语义保留变异如GrayC[17]，基于抽象语法树，使用语义感知的变异算子，但仍然需要对程序积累大量变异才有可能触发缺陷。此外，非语义保留变异中有一些研究关注优化交互问题，并设计了针对优化阶段的模糊测试方法。MopFuzzer[31]提出了一种最大化优化交互的测试方法，通过调整编译器的优化顺序来触发潜在错误，并结合差分测试进行验证。MopFuzzer的方法在JVM编译器上取得了成功。然而，其方法主要适用于动态优化(如JIT编译)，而在静态C语言编译器，特别是嵌入式编译器上，其适用性仍有待探索。

LLMCfuzz结合语义保留和非语义保留变异，设计了多样性引导的变异策略来选择变异算子，并通过增强程序数据流和控制流复杂度的提示指令来进行更为激进的变异，生成对编译器优化缺陷敏感的测试程序。

近年来，随着自然语言技术的发展，大语言模型涌现并开始应用于自然语言和代码任务[32]中。最先进的大语言模型基于Transformer[33]的解码器模型。基于指令的大语言模型能够理解人类提供的复杂指令并给出回答。大语言模型近年来在软件测试中逐渐展现优势。TitanFuzz[27]利用Codex生成种子程序并执行基于模板的变异，用于深度学习库测试。CodaMosa[28]为了突破传统搜索测试覆盖率瓶颈问题，利用Codex生成新的单元测试。

针对编程语言编译器，Fuzz4All[14]结合生成和变异策略，利用大语言模型提取特性文档生成测试提示[34]，在C++、Go等编译器中发现了缺陷。然而，Fuzz4All未能在以稳定性和安全性为主的C语言编译器中发现错误。并且，Fuzz4All缺少对测试程序中变量的数值监控，这导致该方法难以发现静默误编译错误。

综上所述，当前编译器模糊测试技术仍存在以下几个关键技术难题：(1)现有基于生成的方法生成的测试程序多样性不足，难以有效触发编译器的深层次优化缺陷；(2)已有变异方法大多需要积累大量微小变异或难以兼顾语义有效性与测试程序多样性，测试效率较低；(3)目前的基于大语言模型的方法尚未有效解决静默误编译错误检测问题，特别是缺乏针对航空发动机嵌入式系统等特定领域的适配机制。

本文的LLMCfuzz专为航空发动机嵌入式C编译器设计，通过变量追踪机制有效检测静默误编译错误，成功在一种航空发动机嵌入式C编译器中发现了5种误编译错误，其中包括3种静默误编译错误。

3　方　法

为了实现航空发动机嵌入式系统C编译器中后端错误检测，本文提出一种基于大语言模型的C编译器模糊测试方法LLMCfuzz，其结构如图1所示。LLMCfuzz的框架包括变异提示生成，测试、生成和差分测试三部分。在变异提示生成阶段，变异调度器会基于多样性引导策略，为种子程序选择一个变异算子；然后将变异算子的相关任务、指令和种子程序填入变异提示模板，生成变异提示。在测试生成阶段，将变异提示输入代码生成大模型（即DeepSeek-Coder[35]）中，由大模型生成变体程序，其中无效的变体程序及其前端错误信息将作为示例，加入当前的变异提示模板；然后，基于本文提出的变量追踪机制，LLMCfuzz使用大模型为变体程序插入多个输出语句来监测全局与局部变量，从而得到测试程序。最后，测试程序将用于编译器差分测试，从而检测编译器错误。

选择DeepSeek-Coder作为代码生成大模型的依据是，DeepSeek-Coder是一种开源的大语言模型，该模型经过专门的代码数据预训练，能够深刻理解程序语义与结构，具备强大的代码生成能力，擅长生成复杂结构、精确数据流和控制流的程序变体。在多个代码生成任务基准上，DeepSeek[34]系列模型均表现出明显的性能优势，已被广泛验证具备领先的代码智能能力。

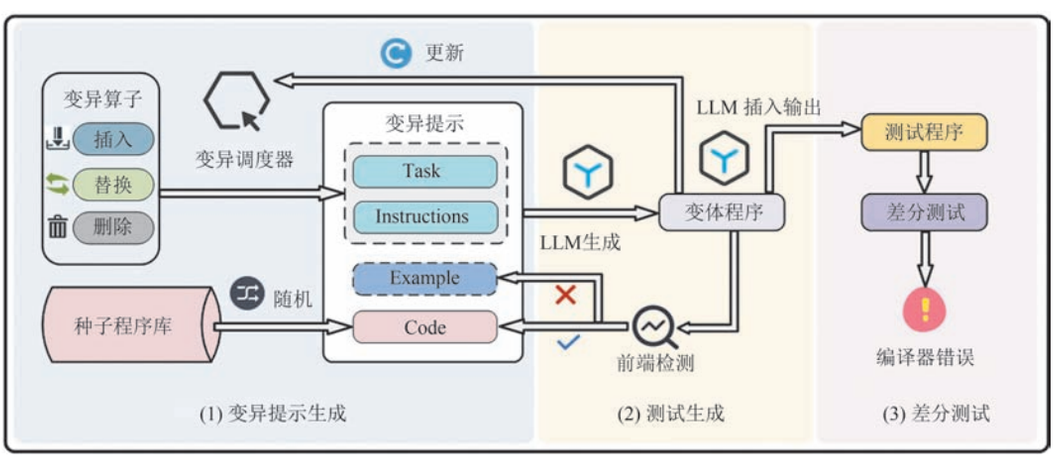

图1　LLMCfuzz框架示意

3.1　变异提示生成

进行程序变异的目标是生成能够触发航空发动机编译器中后端错误的测试程序。已有研究表明，大多数编译器中后端错误属于优化错误[36]，而编译器的优化组件一般依赖于程序的不同结构和数据流而发挥作用[37]，航空发动机编译器同样具有该特点，需要兼具复杂控制流、数据流并且合规的测试程序来对其进行测试。而基于变异的方法GrayC设计的变异算子几乎都是对程序结构影响较小的微小变异，如改变数据类型、替换运算符等，这导致GrayC需要对种子程序积累一定量的变异才能改变程序的结构。因此，LLMCfuzz结合了结构化变异（插入）和微小变异（替换和删除），设计了如表1所示的13种变异算子。并且，为了累积多个变异算子的效果，对于每一个种子程序，LLMCfuzz对其进行迭代变异，在迭代变异结束前，每一个变体程序都将作为新的种子程序再次进行变异。

本方法对删除变异算子的设计基于以下原则：首先是前端正确性。删除变异算子被严格限定为微小变异，仅采用移除限定符、修饰符或一元运算符这三种算子，是为了让变体程序能够通过编译器前端的语法检查，这样变体程序才能到达编译器中后端，从而触发中后端错误。如果删除变异算子采用删除部分程序语句（如删除if分支语句或者某行代码），可能破坏控制流或导致变量缺少定义，使得测试程序无法通过编译器前端。其次考虑变体程序触发编译器中后端错误的潜力。移除限定符（如volatile）可能会改变变量的内存访问语义，影响编译器的优化逻辑；移除修饰符（如const）可能导致未定义行为，而该类错误仅在中后端优化时暴露；移除一元运算符可能改变表达式语义，有助于检测编译器对表达式计算的中后端错误。

表1 LLMCfuzz的变异算子

序号	类型	名称	[Task]
1	插入变异算子	插入分支结构	Insert one branching statement such as ‘if’, ‘if-else’, ‘switch’, or other conditional constructs into the provided C code.
2		插入循环结构	Insert one loop statement such as ‘for’, ‘while’, ‘do-while’, or other loop constructs into the provided C code.
3		插入控制语句	Insert one control statement such as ‘return’, ‘break’and ‘continue’ if there are loop statements, or other control statements into the provided C code.
4		插入结构体	Insert one structure definition and usage into the provided C code.
5		插入表达式	Insert one complex arithmetic, logical, bitwise, or ternary expression into the provided C code.
6		插入死代码	Insert one piece of dead code into the provided C code.
7	替换变异算子	替换某些常量	Replace some constants with another in the provided C code.
8		替换某些变量	Replace the type of some variable with another in the provided C code.
9		替换某些运算符	Replace some operator with another in the provided C code.
10		替换某些赋值语句的右值	Replace some right-hand value of an assignment statement with another variable or expression in the provided C code.
11	删除变异算子	移除某些限定符	Remove some qualifier (i. e. , volatile, const, and restrict) in the provided C code.
12		移除某些修饰符	Remove some modifier (i. e. , long, short, signed, and unsigned) in the provided C code.
13		移除某些一元运算符	Remove some unary operator in the provided C code.

LLMCfuzz利用代码生成大模型对种子程序进行变异。将变异提示输入大模型中，大模型会根据提示要求输出变体程序。为了能够根据不同的变异算子和种子程序生成需要的变异提示，我们设计了一种变异提示模板，初始的模板包含三部分：[Task]，[Instructions]，[Code]。[Task]是变异的任务描述，[Instructions]包含变异操作的多个具体指令，[Code]是即将进行变异的种子程序。

图2以插入分支结构变异为例，展示了模板三个部分的具体内容：[Task]部分表明了插入分支结构变异的总体需求。[Instructions]描述了4项具体指令，即重用程序中已有的变量或表达式；在复杂的控制流处进行插入；如果有必要可以声明并初始化新的变量；大模型的回答中应仅包含变体程序。[Code]部分用于填入种子程序，该程序来自GCC编译器的测试套件。表1中展示了每个变异算子对应的变异提示模板中[Task]的具体设置。

除了插入控制语句的提示模板的[Instructions]中仅包含指令[2]和指令[4]以外，每一种插入变异的提示模板中，[Instructions]的内容都是相同的，而在替换变异算子和删除变异算子中，考虑到替换与删除操作在当前种子程序上可能不具备执行条件，因此在替换变异的提示模板中，[Instructions]被设置为如下2条指令：[1] If there are no＜被替换内容＞to replace, respond me with the original code I provided. [2] The response should contain only code. 而在删除变异的变异提示中，[Instructions]被设置为：[1] If there are no＜被删除内容＞to remove, respond me with the original code I provided.

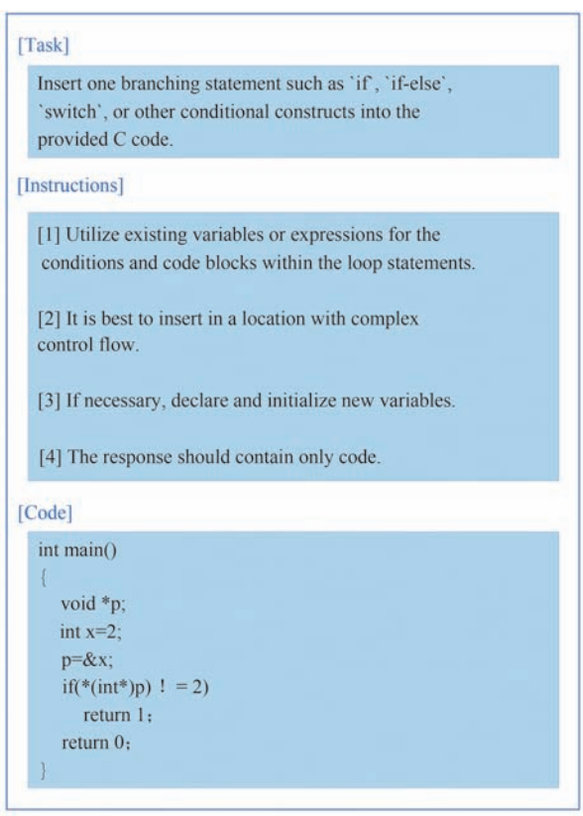

图2 插入分支结构的变异提示示例

[2] The response should contain only code.

值得注意的是，[Instructions]中的前三项指令中，指令[1]要求构建分支语句时，在分支条件和代码块中重用程序中已有的变量或表达式，这会将原程序中的数据流链接到分支语句中，从而达到数据流增强的效果。指令[2]要求在控制流复杂度高的位置执行插入，同样能增强数据流，并且当插入具有新的控制流的结构（如循环语句和分支语句）时，程序的控制流复杂度也将得到增强。指令[3]保证了在种子程序中已有变量不足时能够定义新的变量用于分支语句代码块中。

本文第4节中的实验表明，使用[Instructions]的前三项指令来提示大模型生成变体程序，可以提高编译器测试的覆盖率。

3.2　测试生成

3.2.1　前端错误反馈

尽管当前流行的代码生成大模型经过了大量语法正确的程序数据集的训练，但仍然有可能生成语法错误的程序。语法错误的程序在编译器前端的词法、语法分析等阶段将直接被丢弃，无法进入编译器中后端，因此它们无法检测编译器中后端缺陷。受到当前大语言模型提示工程中的少样本提示技术[38]（Few-shot Learning）启发，LLMCfuzz将具有语法错误的变体程序及其前端错误信息作为反馈，从而提高大语言模型生成有效程序的能力。

LLMCfuzz通过更新变异算子的提示模板来实现前端错误反馈机制。当使用某种变异算子产生了具有语法错误的变体程序，LLMCfuzz将为该变异算子的提示模板添加[Example to avoid syntax error]部分，并将该变体程序与编译时产生的前端错误信息填入该部分。更新后的变异提示模板将包含四部分：[Task]，[Instructions]，[Example to avoid syntax error]和[Code]. 当存在多个具有语法错误的变体程序时，可以添加多个样本，格式为[Example 1 to avoid syntax error]，[Example 2 to avoid syntax error]，…，[Example N to avoid syntax error]。其中N是最大样本数，若具有语法错误的变体程序数量大于N，LLMCfuzz将从中随机选择N个程序作为few-shot样本。

3.2.2　变量追踪机制

LLMCfuzz使用大语言模型生成的变体程序可以直接用于编译器测试，但是和大多数现有的模糊测试方法相同，变体程序仅能检测编译器崩溃错误和显式误编译错误（例如，段错误）。然而，发生崩溃错误时，编译器会显式地产生错误编译信息，并且不会生成有效的可执行文件；而显式误编译错误发生时，程序会出现提示错误执行信息并终止，因此该类错误容易被及时修复。因此，以上两种编译器错误的危害性相对较小。而静默误编译错误会导致编译器编译产生错误的可执行文件，而且在编译和执行时均不会出现显式错误信息。该类错误可能导致软件系统出现难以调试重大故障，严重威胁航空发动机运行时安全。

Csmith是少数能够检测编译器静默误编译错误的模糊测试工具之一。Csmith生成的测试程序在运行时会输出程序中所有全局变量哈希处理后的结果，在进行编译器差分测试时，如果程序在不同编译环境下的输出结果不同，并且没有显式错误信息出现，则说明触发了静默误编译错误。

LLMCfuzz设计了一种变量追踪机制来实现类似的功能。LLMCfuzz不会将变体程序直接用于编译器测试，而是利用大语言模型为变体程序插入多个输出语句。具体来说，LLMCfuzz在变体程序所有被调用函数和主函数的所有return语句之前或者程序执行的最后一行处插入printf语句，输出所有全局变量以及插入位置的作用域内可见的所有局部变量。与Csmith的方法相比，变量追踪机制具有两个优势：首先，除了全局变量，局部变量的值也被输出，这样能更全面地监测程序中变量的值，符合航空发动机程序对高可靠性的要求；其次，逐个输出变量的值，而不是输出对变量哈希处理后的结果，这样能够直接定位到出现错误的变量。

图3展示了在Linux编译并执行的测试环境下，插入输出语句的提示模板。模板包含四部分：[Task]部分是插入printf语句的总体需求；[Instructions]部分是多个具体指令；[Example]作为单样本提示，使用了一个示例来引导大语言模型进行printf语句的插入，该示例中共有3个printf语句，并且在printf语句前添加了注释；[Code]是待填入的变体程序。具有输出语句的变体程序即可作为测试程序，测试程序编译运行后的输出结果将被收集从而进行差分测试。

在差分测试阶段，测试程序需要经过交叉编译，并在航空发动机嵌入式系统的开发板上进行验证。然而，由于Linux环境下的测试程序无法直接在嵌入式环境中运行，LLMCfuzz对其进行了调整。具体而言，LLMCfuzz使用基于UART通信协议实现的Send()函数替换测试程序中的所有printf语句，

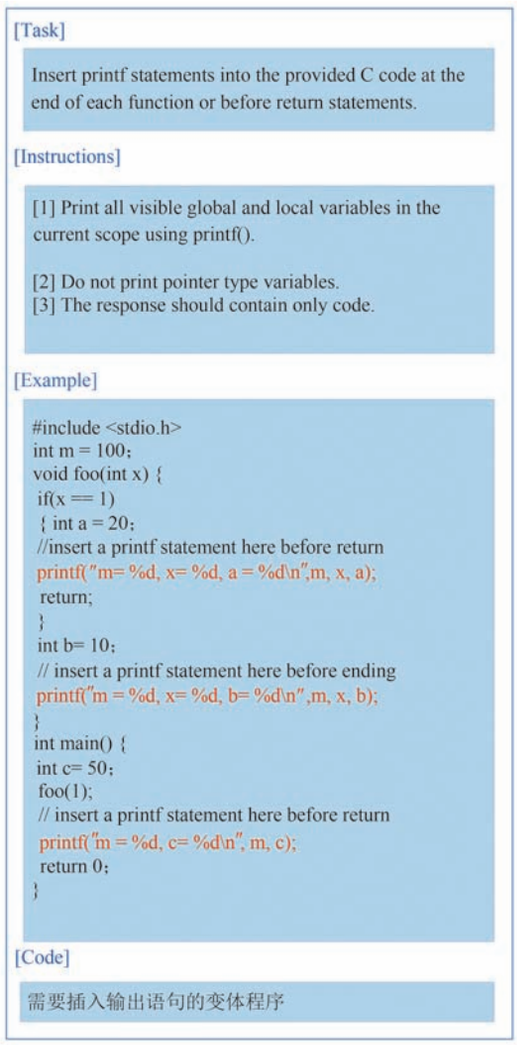

图3 插入输出语句的提示示例

该函数具有与printf()相同的参数和输出格式。为此，程序删除了<stdio.h>头文件，将其替换为UART相关的头文件及其他运行时必要的头文件。此外，为确保数据发送的稳定性，LLMCfuzz在main函数结束前插入延时函数，以避免中断发送过程。

3.3 多样性引导的变异算子选择策略

3.3.1 变异算子排序

程序变异操作会引发组合爆炸问题，这使得LLMCfuzz事实上不可能为单个种子程序枚举并生成其所有的潜在变体。换言之，多样化的测试程序能够探索编译器中后端更大的空间，从而更有可能发现编译器中后端缺陷。为此，一种核心策略在于，确保每次生成的变体程序与当前作为变异基础的种子程序（该种子程序可能源自前一轮迭代）具有显著的差异性。在LLMCfuzz中，这种程序间的相异度是通过计算其Jaccard距离来量化的。两个程序之间的距离计算公式如下：

$$Dist(P1,P2)=1-\frac{Stmt_{P1}\cap Stmt_{P2}}{Stmt_{P1}\cup Stmt_{P2}} \tag{1}$$

其中，$Stmt_{P1}$和$Stmt_{P2}$分别表示程序$P1$和$P2$的行级代码集合，即将程序按行分割后形成一个集合，集合中的每一个元素都是一行代码。

针对当前种子程序，LLMCfuzz选择一种变异算子对其进行变异。鉴于并非所有变异算子在构建“错误敏感结构”时都同等高效，一个动态的择优机制显得至关重要。LLMCfuzz采用了一种基于排名的选择策略来调度变异算子。该策略首先为每个算子量化一个优先级得分；随后，所有算子依据此得分进行降序排列，构成一个优先队列，LLMCfuzz将据此队列进行选择。各变异算子的优先级得分由下式给出：

$$Score(Mut)=\frac{1}{n}\sum_{i=1}^{n}Dist(P_i,P_{i-1})*Rate(Mut) \tag{2}$$

其中，Mut表示变异算子，n是由Mut变异得到的程序变体数量，$Dist(P_i,P_{i-1})$表示第i次变异产生的程序间Jaccard距离。这里的P_{i-1}可能是原始种子程序，也可能是变体程序，这是因为LLMCfuzz对一个种子程序迭代多次变异。$Rate(Mut)$表示Mut变异产生有效的变体程序的成功率，计算公式如下：

$$Rate(Mut)=\frac{\sharp FrontPass_{Mut}}{\sharp All_{Mut}} \tag{3}$$

其中，$\sharp FrontPass_{Mut}$表示大语言模型基于Mut生成的能够通过编译器前端的变体程序数量，$\sharp All_{Mut}$表示基于Mut生成的变体程序总数。

3.3.2 变异算子选择

LLMCfuzz的变异算子选择机制旨在规避朴素贪心策略的局限性。具体而言，尽管系统会在每次变异后根据历史效果对所有算子进行优先级评分和排序，但直接选择排名最高的算子并非最优解，因为历史效果无法保证其在未来变异中的表现。为此，我们构建了一个概率性的选择框架，其核心思想是将算子选择问题转化为一个从特定概率分布中进行抽样的过程。该框架确保了所有算子均有被选中的机会，同时赋予高优先级算子更高的选择概率。此选择过程具有马尔可夫性，即第i次选择的变异算子Mut_b仅与第$i-1$次选择的变异算子Mut_a有关。为了解决该抽样问题，LLMCfuzz采用了Metropolis-Hastings（MH）算法[39]，这是一种常见的马尔可夫链蒙特卡罗方法。MH算法通过一个提议分布（proposal distribution）从当前状态（当前

算子）生成一个候选状态（下一个算子），并依据一个接受概率p来决定是否采纳该候选状态作为下一个转移目标。与已有工作类似[40]，LLMCfuzz使用几何分布来建模伯努利试验的成功次数。一个对称的几何分布被用于定义从当前变异算子转移至候选变异算子的接受概率，具体如下：

$$Pr(X=k)=(1-p)^{k-1}p \tag{4}$$

在每次变异时，随机选择变异算子并判断是否使用该变异算子，因此LLMCfuzz采用对称的几何分布。给定上一次使用的变异算子Mut_a，当前随机选择的变异算子Mut_b被接受的概率为：

$$P_b(Mut_b|Mut_a)=\min\left(1,\frac{P_r(Mut_b)}{P_r(Mut_a)}\right)=\min(1,(1-p)^{k_a-k_b}) \tag{5}$$

其中，k_a和k_b分别是变异算子Mut_a和Mut_b的排名，若$k_b<k_a$，则Mut_b一定被接受；否则，Mut_b将以概率$(1-p)^{k_a-k_b}$被接受。

为了定义单次伯努利试验的成功概率p，LLMCfuzz施加了如下三项约束条件：

$$0.95\leqslant\sum_{k=1}^{13}Pr(X=k)\leqslant 1 \tag{6}$$

$$p>\frac{1}{13} \tag{7}$$

$$0.001<(1-p)^{13-1}p \tag{8}$$

公式（6）旨在确保整体概率分布的归一性，使其累积和趋近于1，公式（7）为排序最高的算子设定了一个不低于0.08的选择概率下限，以强化对优质变异算子的利用，公式（8）保证优先级排序最低的变异算子被选中的概率不为0，所以p的取值范围是$0.22<p\leqslant 0.39$。LLMCfuzz设置p的值为0.3。

算法1描述了在基于多样性引导的变异算子选择策略下，进行测试生成的流程。第1行随机选择一个变异算子Mut_a，第2行初始化测试程序集合P为空；第3到第28行循环进行测试生成。其中，第4行随机选择一个种子程序，第5行初始化变异算子排序，第6行初始化变异迭代次数为0；第7到第27行针对当前种子程序执行迭代变异，其中第8到第14行进行本节所述的基于马尔可夫链蒙特卡洛算法的变异算子采样方法来选择变异算子Mut_b；第15、16行生成变体提示并利用代码生成大模型生成变体程序VP；17到22行判断变体程序VP无语法错误时，生成插入输出语句的提示并利用大语言模型生成测试程序TP，然后迭代当前种子程序为VP，最后按照3.3.1节所述方法更新变异算子Mut_b的优先级分数；第23到25行判断当变体程序VP存在语法错误时，更新变异算子Mut_b的优先级分数，并且中断本次迭代变异，从种子程序库中随机选择下一个种子程序。

3.4　差分测试

LLMCfuzz使用测试程序对编译器进行差分测试，差分测试的主要思想是测试程序在不同配置下进行编译运行，若编译或运行结果出现任何意外或不一致的情况，则认为有编译器出现了错误，最后再进行人工错误分析。差分测试包括随机差异测试和不同优化级别下测试。一般地，随机差异测试进行跨编译器版本的测试。

具体来说，将一个测试程序使用两个以上版本的编译器分别进行编译运行，然后根据编译和运行结果判断编译器是否出现错误；不同优化级别下测试时进行跨优化级别的测试。具体来说，将一个测试程序使用同一个编译器但在不同优化级别下编译运行，然后根据编译和运行结果判断编译器是否出现错误。

算法1.　基于多样性引导的变异算子选择策略下的测试生成算法

输入：种子程序集合S；最大迭代变异次数MAX；变异算子列表M；储存每个变异算子生成的无效变体程序及其错误信息的字典D

输出：测试程序集合P

BEGIN

1. $Mut_a \leftarrow M_{i\leftarrow \text{random}(1,\cdots,13)}$
2. $P \leftarrow \{\}$
3. WHILE 未达到最大测试程序数 DO
4. $seed \leftarrow$ RanSel(S) //随机选择种子程序
5. mutatorSort$\leftarrow$ Sort(M) //对变异算子列表排序
6. $i \leftarrow 0$　//初始化迭代次数
7. 　FOR i= 1 to MAX DO
8. $k_a \leftarrow$ Position($mutatorSort$, $ListMut_a$)
9. $Mut_b \leftarrow$ None
10. 　DO
11. $Mut_b \leftarrow M_{i\leftarrow \text{random}(1,\cdots,13)}$
12. $k_a \leftarrow$ Position($mutatorSort$, Mut_a)
13. $f \leftarrow$ random() //f在区间[0, 1)随机取值
14. 　WHILE $f \geqslant (1-p)^{k_a-k_b}$
15. $mutPrompt$=GenMutPrompt(Mut_b,$seed$,D[Mut_b])
16. $VP \leftarrow$ LLM($mutPrompt$)　//生成变体程序
17. 　IF VP无语法错误 THEN
18. $oraclePrompt \leftarrow$ GenOraclePrompt(UP)
19. $TP \leftarrow$ LLM($oraclePrompt$)
20. $P \leftarrow P \cup \{TP\}$

21. *seed* ← *VP* //迭代种子程序为新的变体程序
22. UpdateScore(Mut_b)
23. ELSE
24. UpdateScore(*M*)
25. BREAK
26. END IF
27. END FOR
28. END WHILE
29. RETURN *P*

END

为了测试的全面性，LLMCfuzz结合随机差异测试和不同优化级别下测试。选择两种编译器进行随机差异测试，即一种基于GCC的航空发动机嵌入式系统交叉编译器和Linux系统下的GCC-14.1编译器；选择5种优化级别（即[-O0，-O1，-O2，-Os，-O3]）进行不同优化级别下测试。

在编译器中后端错误中危害最大的错误是静默误编译错误。图4展示了在嵌入式环境下，LLMCfuzz使用测试程序对航空发动机交叉编译器进行差分测试并发现静默误编译错误的过程。回顾3.3.2节，LLMCfuzz为变体程序插入多个printf()语句从而得到测试程序，而这些测试程序在Linux系统下的GCC-14.1中能够顺利编译，并在相同系统下执行。然而，它们经过交叉编译器编译生成的可执行文件需要写入芯片后执行。因此，我们将测试程序中的输出语句修改为串口输出代码，当可执行文件在芯片中运行时，利用串口输出将变量值传回主机。最后，对比在Linux系统与芯片内测试程序的输出，若出现两种以上结果，则说明其中有编译器出现了错误，此时可进行人工分析具体情况。

算法2描述了LLMCfuzz的差分测试方法。第1～3行初始化优化级别列表、运行结果集合和两种编译器。第4～19行描述了测试流程。第6、7行判断若当前编译器为交叉编译器，则修改测试程序*tp*使其适用于交叉编译环境。第8～12行根据编译时的信息判断测试程序是否具有语法错误或出现编译器崩溃错误；第14～16行根据程序运行时信息判断是否出现显式误编译错误；第17～19行收集程序输出结果并判断，若程序出现两种以上结果，则判断出现静默误编译错误。

算法2. 差分测试算法

输入：Linux环境下测试程序*tp*，交叉编译器*CrossCompiler*，GCC-14.1编译器*GCC*

输出：差分测试结果

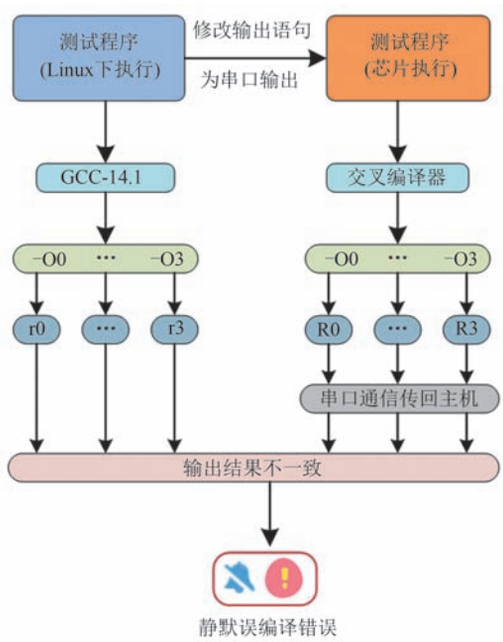

图4 静默误编译错误检测方法

BEGIN

1. *opts* ← [-O0，-O1，-O2，-Os，-O3]
2. *results* ← {}
3. *compilers* ← [*CrossCompiler*，*GCC*]
4. FOR *C* in *compilers* DO
5. FOR *opt* in *opts* DO
6. IF *C* == *CrossCompiler*THEN
7. *tp* ← Modify(*tp*)
8. END IF
9. *compileInfo*，*binary* ← *C*. compile(*tp*，*opt*)
10. IF *compileInfo* 为程序前端错误 THEN
11. RETURN FrontError(*C*，*tp*，*opt*)
12. ELSE IF *compileInfo* 为编译器崩溃错误 THEN
13. RETURN CrashBug(*C*，*tp*，*opt*)
14. ELSE
15. *executeInfo*，*res* ← *binary*. Execute()
16. IF *executeInfo* 为显式误编译错误 THEN
17. RETURN ExplicitMisBug(*C*，*tp*，*opt*)
18. END IF
19. *results* ← *results*∪{*res*}
20. END IF
21. END FOR
22. END FOR
23. IF len(*results*) > 1 THEN //为静默误编译错误
24. RETURN SilentMisBug(*C*，*tp*，*opt*)
25. END IF

END

4　实验分析

4.1　实验设置

(1)种子程序集。为了与其他方法公平比较，并且获取具有发现编译器错误潜力的种子程序，我们从GCC和LLVM编译器的测试套件中收集了1000个测试程序作为种子程序集的主要部分；此外，为了使种子程序更有可能发现与航空发动机领域相关度更高的编译器错误，从航空发动机嵌入式系统源码中收集了200个测试程序，总计1200个种子程序。这些种子程序都已被验证能够通过编译器前端，并且不能触发航空发动机编译器的任何错误。

为了确保种子集的全面性与多样性，我们对其进行了静态分析。该种子集包含1200个C程序，总代码量约16万行，程序规模从20行到超800行不等(平均约135行)，平均圈复杂度约为8.2，体现了其在规模与结构上的显著多样性。在语言特性上，种子集广泛覆盖了指针运算、嵌套数据结构、位域、复杂控制流(如goto)及宏定义等已知容易引发编译器错误的构造。特别地，来自航空发动机系统的200个种子程序还引入了volatile关键字、硬件位操作等领域特有的编程范式，为测试编译器在真实、复杂场景下的鲁棒性提供了坚实基础。

(2)测试环境与参数设置。实验选择一种航空发动机嵌入式系统常用的基于GCC的交叉编译器作为待测编译器，选择GCC-14.1作为Linux环境下进行差分测试的参照编译器。Linux环境为32 GB内存，Ubuntu-22.04操作系统。本文使用的大语言模型为DeepSeek-Coder-V2.5，该模型为开源的、具备代码生成能力的大型语言模型，采用Transformer架构，参数规模为236B，支持多种编程语言理解与生成任务。我们使用其官方提供的API接口，未进行额外微调或训练，直接通过提示词驱动完成变异生成任务。在实验中，模型的temperature参数设置为默认值1，以增加输出多样性。变异提示模板中的前端错误反馈最大样本数设置为3；最大迭代变异次数设置为8；伯努利试验成功率设置为0.3。

(3)基线方法。在进行对比实验时，将LLMCfuzz与四种现有的编译器模糊测试方法进行比较，分别是Fuzz4All，GrayC，Clang-Fuzzer，universalmutator，以及Csmith。Fuzz4All是一种通用的基于大语言模型的编译器模糊测试方法，利用大语言模型提取编程语言特性生成测试提示，再使用代码生成大模型根据测试提示生成测试用例，为了实验公平性，我们将Fuzz4All使用的代码生成大模型设置为DeepSeek-Coder-V2.5。GrayC是最新的基于变异的编译器模糊测试方法，该方法设计多种语义感知的变异算子对种子程序进行变异，生成无语法错误的测试程序，这些测试程序通过编译器前端从而检测编译器中后端错误。Clang-Fuzzer将种子程序视为自然语言文本，对其进行字节级突变从而生成测试程序用于编译器测试。universalmutator是一种通用的模糊测试工具，设计了一系列适用于多种语言的通用变异算子，因此适用于生成C测试程序。对比实验选择这些方法是因为它们都基于种子程序集，使用一系列变异算子生成丰富的测试程序用于编译器测试，这与LLMCfuzz生成测试程序的流程相同。Csmith是一种最广泛使用的基于语法规约的C程序生成器，支持C语言的大多数功能，因此我们将Csmith也作为基线方法的一种来进行对比实验。

(4)LLMCFuzz变体。在消融研究中，我们评估LLMCfuzz的多种变体。针对变异提示模板中[Instructions]，设置变体LF-No-DFCF(LLMCfuzzwithout Data Flow and Control Flow Enhancement)。该变体中去除了插入变异的提示模板中[Instructions]的[1]、[2]和[3]，这三条指令旨在增强生成变体程序的控制流和数据流。针对前端错误反馈，设置变体LF-No-Feedback。在该变体中，仅使用最初始的变体提示模板，即[Task]+[Instructions]+[Code]，而不再为模板加入[Example to avoid syntax error]部分。针对多样性引导的变异算子选择策略，设置变体LF-Random。在该变体中，变异算子将被随机选择用于当前的变异。值得注意的是，我们并没有为变量追踪机制设计变体，这是因为变量追踪机制是为了使测试程序能够发现编译器的静默误编译错误，其效果将在实验部分的4.4节中体现。

(5)评估指标。为了能够更细粒度地评估测试程序对编译器内部的探索程度，实验采用不同方法对待测编译器的覆盖行数，覆盖提升率以及生成测试程序的有效率作为评估指标。其中覆盖提升率的计算公式如下：

$$\text{覆盖提升率} = \frac{\textit{LLMCfuzz}\text{覆盖行数} - \text{其他方法覆盖行数}}{\text{其他方法覆盖行数}} \times 100\% \tag{9}$$

4.2 覆盖率对比实验

除了Csmith以外，每种方法使用4.1节所述种子程序集，分别生成10 000个测试程序，每种方法运行10次，取覆盖行数和有效程序数的平均值。与先前的工作[17]类似，我们在编译时开启-O3级别优化（开启最多的优化组件），超时时间（编译超时则认为程序无效）设置为20秒，并利用Gcov工具来收集覆盖行数结果。

表2展示了每种方法对编译器的覆盖行数，覆盖提升率和测试程序的有效性。从表中数据可以看出，LLMCfuzz对编译器的覆盖率相比于其他方法更高，相较于Fuzz4All和GrayC，在覆盖行数上分别多出了9475行和4483行，覆盖提升率分别为6.06%和2.78%。其中Clang-Fuzzer的覆盖行数最少，这是因为Clang-Fuzzer生成的测试程序绝大多数未能通过编译器前端，其有效率仅1.48%。

表2 每种方法的覆盖行数和测试程序有效率

方法	覆盖行数	覆盖提升率	有效率	有效程序数/程序总数
LLMCfuzz	165 843	—	97.73%	9773.3/10 000
Fuzz4All(2024)	156 368	↑ 6.06%	95.27%	9526.8/10 000
GrayC(2023)	161 360	↑ 2.78%	99.65%	9965.4/10 000
Clang-Fuzzer(2018)	136 973	↑ 21.08%	1.48%	148.1/10 000
Universalmutator(2018)	145 774	↑ 13.77%	98.54%	9853.6/10 000
Csmith(2011)	147 305	↑ 12.58%	99.65%	9965.3/10 000

与GrayC、Universalmutator和Csmith相比，LLMCfuzz生成测试程序的有效性略低一些，这不仅是因为大语言模型代码生成能力有限，还受到以下因素的影响：首先，LLMCfuzz在生成变体程序时引入了更复杂的提示模板和变异策略，这提高了程序的多样性，但也可能增加生成无效程序的概率；其次，前端错误反馈机制的样本容量有限，未能覆盖所有潜在错误模式，导致部分问题未能及时修正。尽管如此，LLMCfuzz对编译器的测试覆盖率依然显著高于其他方法。

4.3 消融实验

本节将LLMCfuzz与4.1节所述的三种变体进行了消融实验，与对比实验相同，LLMCfuzz的各变体分别运行10次，取覆盖行数和有效程序数的平均值。表3展示了消融实验的结果。

表3 消融实验结果

方法	覆盖行数	覆盖提升率	有效率	有效程序数/程序总数
LLMCfuzz	165 843	—	97.73%	9773.3/10 000
LF-No-DFCF	162 334	↑ 2.16%	98.24%	9823.9/10 000
LF-No-Feedback	165 269	↑ 0.35%	96.15%	9615.4/10 000
LF-Random	163 560	↑ 1.40%	97.94%	9794.1/10 000

与LF-No-DFCF相比，LLMCfuzz的覆盖率提升最为显著，达到了2.16%（3509行）。这表明在变异提示的[Instructions]部分中，添加用于增强数据流和控制流的指令，有助于更深入地探索编译器的内部路径。然而，LLMCfuzz生成测试程序的有效率略低于LF-No-DFCF。可能的原因在于，复杂的提示指令增加了大语言模型生成测试程序时出错的概率。

对于LF-No-Feedback变体，由于缺少前端错误反馈机制，测试程序的有效率相较LLMCfuzz有所下降（从97.73%下降到96.15%）。同时，该变体的覆盖行数也略低于LLMCfuzz。这可能是因为有效率的降低导致更多的测试程序在编译器前端被拒绝，从而影响了覆盖行数。

在与LF-Random的比较中，LLMCfuzz的覆盖行数提高了2283行，提升了1.40%。这种提升归因于LLMCfuzz采用了基于多样性引导的变异算子选择策略，生成多样化的测试程序有助于更广泛地探索编译器内部路径。值得注意的是，尽管LLMCfuzz在覆盖率上表现更优，但LF-Random的有效率略高一些（97.94%对比97.73%）。这可能是因为在计算变异算子优先级时，尽管考虑了生成有效程序的成功率，但变体程序之间的多样性对优先级的影响更大。LLMCfuzz优先选择能够生成多

样化程序的变异算子，而不是一味追求程序有效性，因此在生成有效程序时有效率略有下降。

4.4　编译器错误发现与分析

LLMCfuzz的设计旨在发现航空发动机嵌入式系统编译器内部的误编译错误，尤其是危害最大的静默误编译错误。静默误编译错误由于其缺少错误信息提示，直接生成错误的可执行文件，会对软件系统留下巨大隐患，甚至导致灾难级事故。

为了检测LLMCfuzz的错误发现能力，与先前的工作类似[17]，我们将LLMCfuzz与其他5种基线方法均运行24小时。除了Csmith以外，每种方法使用4.1节所述种子程序集，将每种方法在该时段内生成的所有测试程序用于交叉编译器的差分测试。值得注意的是，其他基线方法生成的测试程序都将采用3.4节中的差分测试方法，对于生成的测试程序具有输出语句的方法（GrayC，Csmith），其输出语句将被替换为串口输出语句。表4展示了所有方法发现的编译器中的后端错误数量和类型。

表4　每种方法发现的错误数量与类型

方法	静默误编译错误	显式误编译错误	崩溃错误	总计
LLMCfuzz	3	2	0	5
Fuzz4All(2024)	0	0	0	0
GrayC(2023)	0	1	0	1
Clang-Fuzzer(2018)	0	0	0	0
Universalmutator(2018)	0	0	1	1
Csmith(2011)	0	0	0	0

从表4可以看出，LLMCfuzz共发现了5种错误，明显高于其他方法（0～1种）。特别是静默误编译错误，仅有LLMCfuzz发现，这体现了LLMCfuzz在检测航空发动机交叉编译器中隐蔽且危害严重的缺陷方面的独特优势。此外，universalmutator是唯一检测出编译器崩溃错误的方法，而GrayC检测到的唯一显式误编译错误与LLMCfuzz中发现的一种错误（表5中序号4）重合，通过调试优化选项和汇编分析，发现2种错误均由-O3优化级别下的-funswitch-loops优化选项错误导致，该标志在对多层循环语句进行拆分时导致公用循环变量越过循环条件，从而引发了程序运行时的段错误。因此，仅有universalmutator发现了一种LLMCfuzz未能发现的崩溃错误，并且仅有LLMCfuzz发现了危害最大的静默误编译错误。

表5统计了5种错误的具体信息，优化级别由低到高分别为-O0（无优化），-01，-02，-Os，-O3。其中出错级别为-O0及以上的静默误编译错误（错误1和错误2）危害程度最大，这是因为即使在不开启编译器优化的保守编译下，仍可能触发该类错误。而在-O3优化级别下触发的静默误编译错误（错误3），只需要采取其他优化级别就能规避错误。而显式误编译错误（错误4和错误5）的危害程度低于静默误编译错误，这是由于程序运行时会提示0错误信息并终止，这有利于程序调试以及编译器错误发现。如表5所示，本文根据危害程度从高到低将5个编译器错误分为I类，II类和III类。其中I类错误为静默误编译错误并且触发错误的优化级别为-O0，II类错误为静默误编译错误且触发错误的优化级别为非-O0，III类错误为非静默误编译错误。目前LLMCfuzz尚未发现编译器崩溃错误，该类错误同样属于III类。表5中给出了5种错误的各自成因，其中错误1、2、5均是由于航空发动机编译器的特殊设计导致的。

表5　LLMCfuzz发现错误统计

序号	错误类型	出错优化级别	危害程度	出错原因
1	静默误编译	-O0	I类	隐式类型转换错误
2	静默误编译	-O0	I类	常量折叠错误
3	静默误编译	-O3	II类	非法代码移动错误
4	显式误编译	-O3	III类	循环语句拆分错误
5	显式误编译	-Os	III类	只读内存被写入

与其他大多数无法发现I类错误的模糊测试方法不同，LLMCfuzz能够发现静默误编译错误得益于变量追踪机制，并且在差分测试中结合了随机差异测试和不同优化级别下测试两种方式。我们展示了触发5种编译器错误的测试用例与差分测试结果，并且分析了每种错误的具体原因。为了便于展示，我们将测试用例进行了最小化[41]，并采用Linux环境下的用例格式。

图5展示了错误1的测试程序。经过随机差异测试发现程序在GCC-14.1与交叉编译器下编译后运行结果不同，即变量b的值不同。该错误是由于交叉编译器对混合类型（无符号和有符号）在条件运算符中进行隐式转换时处理不一致导致的。本应将有符号-10按16位转换为65 526，却错误地以8位转换为246，导致结果偏离预期。这是交叉编译器针对芯片环境适配设计时产生的缺陷，属于隐式类型转换错误。

图6展示了错误2的测试程序。经过随机差异测试发现程序在GCC-14.1与交叉编译器下编译后

[Test Program]

```
#include <stdio.h>
signed char a = (signed char)-10;
unsigned short b = (unsigned short)37996;
int main() {
   b=2U>a? 2U: a;
   printf("b= %hu  ",b);
}
```

[Output]

```
GCC-14.1所有优化级别下结果：b=65 526
交叉编译器下所有优化级别下结果：b=246
```

图5 错误1的测试程序

运行结果不同，即变量c的值不同，在这个测试用例中，交叉编译器在优化表达式-10 *（32 739>t+4503599ULL? 32 739:t+4503599ULL)时(这里简化为-10 *（condition ? a:b)），将乘法外提为condition? -10 * a: -10 * b，并且在使用无符号条件移动指令(cmovb)时忽略了符号扩展和乘法的有符号语义，且采用移位加法替代乘法时未正确处理负数的低位补全，最终导致计算结果发生偏差。该问题实质上是常量传播与条件表达式优化中的符号处理失误与算术变换副作用的耦合错误。

[Test Program]

```
#include <stdio.h>
shortt= -32 166;
unsigned long long int a, b, c;
void foo(unsigned long long int *a_ptr, unsigned long
long int *b_ptr) {
    *a_ptr=32 739>t+4 503 599ULL ? 32 739: t+
4 503 599ULL;
   *b_ptr =-10 * (*a_ptr);
   printf("t=%hd, a=%llu, b=%llu\n", t, *a_ptr,__
*b_ptr);
}

int main() {
   foo(&a, &b);
   c=-10*(32 739>t+4 503 599ULL?32 739:t+
4503599ULL);
   printf("t=%hd, a=%llu, b=%llu, c=%llu\n", t, a, b,
c);
   return 0;
}
```

[Output]

```
GCC-14.1所有优化级别下结果：t=-32 166,
a=4 471 433,b=18 446 744 073664 837 286
t=-32 166, a=4 471 433,b=18 446 744 073 664 837 286,
c=18 446 744 073 709 224 286
交叉编译器下所有优化级别下结果：
t=-32 166, a=4 471 433,b=18 446 744 073 664 837 286
t=-32 166, a=4 471 433,b=18 446 744 073 664 837 286,
c=18 446 744 073 664 837 226
```

图6 错误2的测试程序

图7展示了错误3的测试程序。该程序通过不同优化级别下测试发现了编译器错误，在交叉编译器开启-O3优化下的编译运行结果中，a[1]. y=9，而其他编译环境下的运行结果中a[1]. y=0。该问题由编译器在-O3优化级别下启用-fpredictive-commoning优化选项时候，对循环体进行展开与重排所致。由于错误地判断a[0]=d与随后d=a[0]之间不存在存后读依赖，编译器将对a[0]. y的读取操作提前至写入前，从而读到了未更新的旧值(初始化为9)，并错误赋值给a[1]. y。该错误属于编译器在别名分析与循环展开优化中破坏值依赖关系所致，实质是非法代码移动问题。

[Test Program]

```
#include <stdio.h>
struct S
{
   int x;
   int y;
} a[2];
int i;
int main()
{
   struct S d= {3,9};
   for (i=0; i<10; i++)
   {
      a[i]= d;
      d=a[0];
   }
   printf("i= %d, d.x= %d, d.y= %d, a[0].x= %d,
a[0].y=%d, a[1].x= %d, a[1].y=%d\n", i, d.x, d.y,
a[0].x,a[0].y, a[1].x,a[1].y);
   return 0;
}
```

[Output]

```
交叉编译器下(-O3)的运行结果：
i=10,d.x=3,d.y=9,a[0].x=3,a[0].y=9,a[1].x=
3,a[1].y= 0
其他所有运行结果：
i=10,d.x=3,d.y=9,a[0].x=3,a[0].y=9,a[1].x=
3,a[1].y= 9
```

图7 错误3的测试程序

图8展示了错误4的测试程序。该程序通过不同优化级别下测试发现了编译器错误。程序中的内外层循环共用了同一个循环变量d，造成了逻辑冲突。当-funswitch-loops优化被开启(-O3默认开启)时，编译器会对循环内不变的条件(即if(a<0))做提前判断，并拆分重组循环结构。然而，这个提前的结构拆分导致变量d的控制逻辑被打乱，出现循环越界，使得数组f被越界访问，从而产生了段错误。

图9展示了错误5的测试程序。该程序通过不

[Test Program]

```
#include <stdio.h>
int a;
int main ()
{
  int b=-1, d, e=0, f[2]= {0 };
  unsigned short c = b;
  for (;  e<3; e++)
    for(d=0; d<2; d++)
      if(a<0)
        for(d=0; d<2; d++)
          if (f[c])
            break;
  printf("b= %d, c= %hu, d = %d, e= %d, f[0] = %d,
f[1]= %d\n ", b, c, d, e, f[0], f[1];
  return 0;
}
```

[Output]

```
GCC-14.1所有优化级别下结果：b=-1,c=65 535,d
=2,e=3,f[0]=0,f[1]=0
交叉编译器下-O3优化级别下程序执行失败，错误
信息：segmentation fault
```

图8　错误4的测试程序

[Test Program]

```
#include <stdio.h>
int a= 0;
charx= 'a';
char *b = &x;
static int c(){
  while (a) {
    b="";
    *b='\0';
  }
  printf("a= %d, x= %c\n", a, x);
  return 1;
}
int main(){
    c();
    printf("a = %d, x= %c\n ", a, x);
    return 0;
}
```

[Output]

```
GCC-14.1所有优化级别下结果：a=0,x=a
a=0,x=a
交叉编译器下-Os优化级别程序执行失败，错误信
息：SIGSEGV,return code 139
```

图9　错误5的测试程序

同优化级别下测试发现了编译器错误。在这段程序中，虽然变量a始终为0，但循环内部对字符串字面量的写操作属于未定义行为；在 -O2 下，编译器通过死代码消除完全移除了这部分代码，而 -Os 为了减小代码体积未能删除，从而保留了写只读内存的风险代码，最终导致段错误（SIGSEGV，返回值139）。该错误是由于交叉编译器为嵌入式程序设定的代码体积优化存在缺陷导致的。

LLMCfuzz发现的5种交叉编译器错误中，其中错误1、2、3为危害最大的静默误编译错误，该类错误的发现得益于变量追踪机制和结合了随机差异测试和不同优化级别下测试的差分测试方法。其中错误1、2、5均为交叉编译器针对航空发动机嵌入式系统的特定优化导致的，这得益于LLMCfuzz采用航空发动机嵌入式系统程序作为种子程序，并且通过增强程序控制流和数据流复杂度的变异策略生成错误敏感结构，生成触发航空发动机编译器错误能力更强的测试程序。

4.5　Few-shot提示的上下文长度评估

为了验证在3.2.1节中设置的N=3的few-shot提示是否存在上下文长度超过限制的问题，我们执行LLMCfuzz，并从N=3之后开始统计LLMCfuzz调用的10 000次大模型API中上下文总长度，以及当生成一个具有前端错误的测试程序之后，编译器为该程序生成的前端错误信息长度。

根据表6的统计可以发现，LLMCfuzz方法在few-shot提示数量N=3时，提示上下文的平均总长度约为1235个token，远低于所采用的DeepSeek-Coder模型的上下文长度限制（128K tokens，大语言模型中一般计为128 000 tokens）。因此不存在上下文超长问题。同时，前端错误信息长度较短，平均仅28个token，并不会显著增加上下文负担。

表6　上下文与前端错误长度（tokens）统计

统计项目	最小长度	最大长度	平均长度
前端错误信息长度	17	74	28
上下文总长度	745	1643	1235

5　总　结

本文提出了一种基于大语言模型的航空发动机编译器模糊测试方法LLMCfuzz。通过设计多样化的变异提示模板和变量追踪机制，LLMCfuzz能够生成具有复杂数据流和控制流的测试程序，有效检测编译器的静默误编译错误，从而将测试覆盖率较其他方法提高了2.78%至21.08%。

实验结果表明，LLMCfuzz在行覆盖率上较现有方法提升了2.78%至21.08%，并成功发现了5种误编译错误，其中包括3种静默误编译错误，这验证了方法的有效性。此外，LLMCfuzz在航空发动机嵌入式系统中的编译器测试具有显著的优势，

尤其是其设计的变量追踪机制，为确保高安全需求系统的稳定性提供了重要保障。

未来的研究可以进一步探索如何更有效地使用大语言模型提升模糊测试的效率和准确性。例如，可采用检索增强生成（Retrieval-Augmented Generation，RAG）技术来构建一个存储和检索编译器前端错误信息的知识库，以动态地引导大语言模型生成更精准、更有针对性的测试用例。此外，也可以探讨将大模型与其他智能技术（如强化学习或主动学习）相结合，建立一种自适应的测试框架，使测试过程能够自动识别和集中关注编译器中容易出现问题的部分，从而显著提高编译器测试的覆盖率和缺陷检测能力。

尽管本文提出的LLMCfuzz方法专门针对航空发动机嵌入式系统的特定需求设计，但其核心思路和框架在其他领域的编译器测试中也具备潜在应用价值。然而，不同领域的编译器存在差异明显的安全规范、优化特性和运行环境，例如汽车电子、医疗设备或航天器等嵌入式系统可能具有独特的编译约束或硬件接口。因此，当LLMCfuzz迁移至其他领域时，应根据目标领域的特定编译规则对变异算子、提示模板及变量追踪机制进行适配调整，例如修改或增加领域特定的变异策略、调整变量追踪的输出机制，以确保生成的测试程序满足领域特定的编译要求。这种策略上的调整将帮助LLMCfuzz更好地融入新的应用场景，充分发挥方法的通用性与有效性。

参 考 文 献

[1] Chen J, Patra J, Pradel M, et al. A survey of compiler testing. ACM Computing Surveys (CSUR), 2020, 53(01): 1-36

[2] Chen J, Hu W, Hao D, et al. An empirical comparison of compiler testing techniques//Proceedings of the 38th International Conference on Software Engineering. Austin, USA, 2016: 180-190

[3] Yang X, Chen Y, Eide E, et al. Finding and understanding bugs in C compilers//Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. San Jose, USA, 2011: 283-294

[4] Livinskii V, Babokin D, Regehr J. Random testing for C and C++ compilers with YARPGen. Proceedings of the ACM on Programming Languages, 2020, 4(OOPSLA): 1-25

[5] Livinskii V, Babokin D, Regehr J. Fuzzing loop optimizations in compilers for C++ and data-parallel languages. Proceedings of the ACM on Programming Languages, 2023, 7(PLDI): 1826-1847

[6] Wang J, Huang Y, Chen C, et al. Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering, 2024, 50(04): 911-936

[7] Schäfer M, Nadi S, Eghbali A, et al. An empirical evaluation of using large language models for automated unit test generation. IEEE Transactions on Software Engineering, 2023, 50(01):85-105

[8] Liu Z, Chen C, Wang J, et al. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions//Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. Lisbon, Portugal, 2024: 1-13

[9] Xu F F, Alon U, Neubig G, et al. A systematic evaluation of large language models of code//Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming. San Diego, USA, 2022: 1-10

[10] Vaithilingam P, Zhang T, Glassman E L. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models//CHI conference on human factors in computing systems extended abstracts. New Orleans, USA, 2022: 1-7

[11] Liu P, Yuan W, Fu J, et al. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM Computing Surveys (CSUR), 2023, 55(9): 1-35

[12] Reynolds L, McDonell K. Prompt programming for large language models: Beyond the few-shot paradigm//Extended abstracts of the 2021 CHI conference on human factors in computing systems. Yokohama, Japan, 2021: 1-7

[13] Jiang E, Olson K, Toh E, et al. Promptmaker: Prompt-based prototyping with large language models//CHI Conference on Human Factors in Computing Systems Extended Abstracts. New Orleans, USA, 2022: 1-8

[14] Xia C S, Paltenghi M, Le Tian J, et al. Fuzz4all: Universal fuzzing with large language models//Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. Lisbon, Portugal, 2024: 1-13

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

[16] Gu Q. Llm-based code generation method for golang compiler testing//Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. San Francisco, USA, 2023: 2201-2203

[17] Even-Mendoza K, Sharma A, Donaldson A F, et al. GrayC: Greybox fuzzing of compilers and analysers for C// Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. Seattle, USA, 2023: 1219-1231

[18] Yang C, Chen J, Fan X, et al. Silent compiler bug de-duplication via three-dimensional analysis//Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. Seattle, USA, 2023: 677-689

[19] Tang Y, Ren Z, Jiang H, et al. Detecting compiler bugs via a deep learning-based framework. International Journal of Software Engineering and Knowledge Engineering, 2022, 32(05): 661-691

[20] Liu X, Li X, Prajapati R, et al. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing// Proceedings of the AAAI Conference on Artificial Intelligence. Honolulu, USA, 2019: 1044-1051

[21] Ye G, Hu T, Tang Z, et al. A Generative and Mutational Approach for Synthesizing Bug-Exposing Test Cases to Guide Compiler Fuzzing//Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. San Francisco, USA, 2023: 1127-1139

[22] Le V, Afshari M, Su Z. Compiler validation via equivalence modulo inputs. ACM Sigplan Notices, 2014, 49(6): 216-226

[23] Le V, Sun C, Su Z. Finding deep compiler bugs via guided stochastic program mutation. ACM Sigplan Notices, 2015, 50(10): 386-399

[24] Sun C, Le V, Su Z. Finding compiler bugs via live code mutation// Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. Amsterdam, Netherlands, 2016: 849-863

[25] Holler C, Herzig K, Zeller A. Fuzzing with code fragments// 21st USENIX Security Symposium (USENIX Security 12). Bellevue, USA, 2012: 445-458

[26] Groce A, Holmes J, Marinov D, et al. An extensible, regular-expression-based tool for multi-language mutant generation// Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. Gothenburg, Sweden, 2018: 25-28

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

[28] Lemieux C, Inala J P, Lahiri S K, et al. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models//2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). Melbourne, Australia, 2023: 919-931

[29] Alshahwan N, Chheda J, Finogenova A, et al. Automated unit test improvement using large language models at Meta// Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, Porto de Galinhas, Brazil, 2024: 185-196

[30] Li S, Theodoridis T, Su Z. Boosting Compiler Testing by Injecting Real-World Code. Proceedings of the ACM on Programming Languages, 2024, 8(PLDI): 223-245

[31] Xie Z, Wen M, Qiu S, et al. Validating JVM Compilers via Maximizing Optimization Interactions//Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. San Diego, USA, 2024: 36-51

[32] Li R, Allal L B, Zi Y, et al. Starcoder: may the source be with you!.arXiv preprint arXiv:2305.06161, 2023

[33] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need//Proceedings of the 31st International Conference on Neural Information Processing Systems. Long Beach, USA, 2017: 6000-6010

[34] Liu J, Xia C S, Wang Y, et al. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation//Proceedings of the 37th Conference on Neural Information Processing Systems. New Orleans, USA, 2023: 21558-21572

[35] Guo D, Zhu Q, Yang D, et al. DeepSeek-Coder: When the Large Language Model Meets Programming--The Rise of Code Intelligence. arXiv preprint arXiv:2401.14196, 2024

[36] Sun C, Le V, Zhang Q, et al. Toward understanding compiler bugs in GCC and LLVM//Proceedings of the 25th International Symposium on Software Testing and Analysis. Saarbrücken, Germany, 2016: 294-305

[37] Chen J, Ma H, Zhang L. Enhanced compiler bug isolation via memoized search//Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. Melbourne, Australia, 2020: 78-89

[38] Brown T, Mann B, Ryder N, et al. Language models are few-shot learners//Proceedings of the 34th Conference on Neural Information Processing Systems. Vancouver, Canada, 2020: 1877-1901

[39] Kass R E, Carlin B P, Gelman A, et al. Markov chain Monte Carlo in practice: a roundtable discussion. The American Statistician, 1998, 52(2): 93-100

[40] Chen Y, Su T, Sun C, et al. Coverage-directed differential testing of JVM implementations//Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. Santa Barbara, USA, 2016: 85-99

[41] Regehr J, Chen Y, Cuoq P, et al. Test-case reduction for C compiler bugs//Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. Beijing, China, 2012: 335-346

LU Wei, M. S. candidate. His research interests include intelligent software development and compiler testing.

LI Wei-Wei, Ph. D. , associate professor. Her research interests include machine learning, intelligent software engineering, high-security software development methods research and software reliability.

SHI Bin-Bin, M. S. candidate. His research interests include high-security embedded real-time operating system, compiler verification and aero-engine control software.

ZENG Jun-Wei, M. S. candidate. His research interests include intelligent software development and compiler testing.

HUANG Zhi-Qiu, Ph. D. , professor. His research interests include software engineering, software security and formal method.

Background

The research problem addressed in this paper lies at the intersection of software engineering, software security, and formal methods, specifically focusing on the domain of embedded systems in aerospace applications, particularly those related to aircraft engine compilers. Compilers serve as a critical component of embedded systems, as their accuracy directly impacts the overall safety and reliability of these systems. Errors in compilers can lead to catastrophic failures, making rigorous testing essential for high-reliability systems like aircraft engines.

Currently, fuzz testing is the predominant method employed for compiler verification and is recognized for its maturity and widespread application. The essence of fuzz testing involves generating a large number of syntactically correct test programs and feeding them into compilers to uncover unexpected behaviors that could indicate errors. Prominent methods such as Csmith and YARPGen have effectively identified hundreds of errors in C language compilers, showcasing the effectiveness of these fuzz testing techniques. However, despite these advancements, the existing methods often struggle with generating diverse test cases, which limits their ability to trigger back-end compiler defects or detect the most harmful silent miscompilation errors.

The emergence of large language models (LLMs) has opened new avenues for software testing, as these models are capable of generating sophisticated test cases based on natural language prompts. Recent studies have attempted to integrate LLMs into fuzz testing for compilers, yielding promising results across various programming languages and compiler systems. However, these approaches still face challenges. Traditional fuzz testing techniques often lack the diversity needed to explore all facets of the compiler's behavior effectively. Additionally, most existing methods have difficulty detecting silent miscompilation errors, which pose significant risks but remain largely unaddressed.

This paper proposes a novel fuzz testing approach named LLMCfuzz (Large Language Model-Based C Compiler Fuzzer), aimed at overcoming the limitations of current methods. LLMCfuzz employs a three-phase strategy encompassing mutation prompt generation, test case generation, and differential testing to enhance the testing process. By leveraging a diverse seed program library and employing various mutation strategies, LLMCfuzz aims to produce a wider variety of test cases that feature complex data and control flows. Moreover, a variable tracking mechanism is introduced to continuously monitor variables throughout the execution, thus facilitating the detection of the most critical silent miscompilation errors.

Through experimental validation, LLMCfuzz has demonstrated significant improvements, increasing line coverage by 2.78% to 21.08% over existing methods and successfully identifying five miscompilation errors in a specific aircraft engine cross-compiler, including three silent miscompilation errors. This research contributes significantly to the field by offering a more effective testing framework that enhances compiler reliability, thereby contributing to the safety of embedded systems in aerospace applications.

This work was supported by the National Science and Technology Major Project of China (No. Y2022-V-0001-0027), the National Natural Science Foundation of China (No. 6257070451), and the Shanghai Central Guidance Science and Technology Development Fund (No. YDZX20233100004008).