博士学位论文

面向编译器测试与调试的程序构造方法研究

Research on Test Program Construction Approaches for Compiler Testing and Debugging

作 者 姓 名: 涂浩新

学 号: 11917009

指 导 教 师: 江贺 教授

学科、 专业: 软件工程

答 辩 日 期: 2023 年 12 月

大连理工大学

Dalian University of Technology

学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经指明引用的内容外，学位论文不含任何其他个人、法人或者非法人组织已经发表或尚未发表的作品，且学位论文中已经指明作者姓名或者名称、作品名称的内容，不影响该作品的正常使用，也不存在不合理地损害相关权利人的合法权益的任何情形。对学位论文研究做出重要贡献的个人和法人或者非法人组织，均已在论文中以明确方式标明，且不存在任何著作权纠纷。

若因声明不实，本人愿意为此承当相应的法律责任。

学位论文题目：面向编译器测试与调试的程序构造方法研究

作 者 签 名 ：_____ 日期：_____ 年 ____ 月 ____ 日

大连理工大学学位论文版权使用授权书

本人完全了解大连理工大学有关学位论文知识产权的规定，在校攻读学位期间论文工作的知识产权属于大连理工大学，允许论文被查阅和借阅。学校有权保留论文并向国家有关部门或机构送交论文的复印件和电子版，可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印、或扫描等复制手段保存和汇编本学位论文。

学位论文题目：面向编译器测试与调试的程序构造方法研究

作 者 签 名 ：_____ 日期：_____ 年 ____ 月 ____ 日

导 师 签 名 ：_____ 日期：_____ 年 ____ 月 ____ 日

摘要

编译器是重要的系统软件之一，大部分计算机软件依赖编译器将程序源代码编译成目标机器代码，以完成各类计算任务。然而，由于编译器源代码规模的庞杂性，编译器中难免存在一些缺陷。研究表明，编译器缺陷不仅会降低编译器的可靠性，甚至造成灾难性的影响。因此，保障编译器的质量尤为重要。编译器测试及调试是编译器质量保障的两种重要技术手段：前者旨在检测编译器中的缺陷，而后者旨在定位缺陷在编译器源码中的位置。目前，研究人员通常采用基于程序构造的方法实现编译器的缺陷检测及缺陷定位。然而，现有程序构造方法构造出的测试程序在有效性方面仍然存在诸多不足，如用于检测深层次编译器前端缺陷的测试程序语法多样性不足、用于检测深层次编译器中后端缺陷的测试程序语义多样性不足以及用于编译器缺陷定位的证人测试程序语义有效性不足等，阻碍了编译器质量的保障。

为了进一步保障编译器质量，本文以程序构造方法为切入点，面向编译器测试及调试场景，针对现有程序构造方法存在的不足，提出构造满足特定需求的测试程序方法，旨在提高编译器缺陷检测与缺陷定位能力。具体而言，本文主要研究内容如下：

（1）*面向编译器前端测试的结构感知程序构造方法*。由于编译器前端缺陷的特殊性和复杂性，亟需构造语法多样化的测试程序用于检测深层次的编译器前端缺陷。为了满足以上需求，该方法解决以下两个挑战：1）如何设计灵活化的结构以实现多粒度结合的变异操作；2）如何选择代表性的结构以构造多样化的测试程序。首先，该方法利用语法规约的完整性及多样性等特性构造灵活化的生成结构，以实施多粒度结合的变异操作，如细粒度的位反转变异和粗粒度的结构化交叉变异等。其次，该方法采用高效的等概率选择策略，均匀地选择最具多样化的语法结构，实现代表性生成结构的选择。最终，该方法将所选的多样化生成结构转化为语法多样化的测试程序。此外，为了有效识别各种类型的编译器前端缺陷，该方法还提出能够有效检测多种前端缺陷的编译器输出信息对齐算法。实验结果表明，该方法能够有效地提高现有方法的缺陷检测能力，在缺陷数量上的提高比例达 111% ～ 135%。该方法在最新版本的编译器中检测出 136 个编译器前端缺陷，其中 67 个已经被开发者确认或者修复。

（2）*面向编译器中后端测试的再造生成器程序构造方法*。程序生成器是目前编译器中后端测试技术的基石。然而，研究表明，随着编译器对现有程序生成器鲁棒性的增强，程序生成器难以生成语义多样化的测试程序，限制了其检测深层次编译器中后端缺陷的能力。为了提高现有程序生成器缺陷检测的能力，该构造方法解决以下两个挑战：1）如何轻量合成多样化的程序片段；2）如何有效选择揭示缺陷的程序片段以构造语义多样化的测试程序。首先，该方法构建了多样化程序片段合成组件。该组件利用程序生成器中代码可复用性高的特性实现轻量化合成，并采用语法覆盖信息指导

构造多样化的程序片段。其次，该方法构建了揭示缺陷程序片段选择组件。该组件根据已定义的语法覆盖率，采用语法覆盖率较高的标准选择最优的程序片段。实验结果表明，再造的程序生成器能够有效地提高现有程序生成器的缺陷检测能力，在检测到的缺陷数量上的提高比例达 10% ～ 16%。该方法在最新版本的编译器中检测出 56 个编译器中后端缺陷，其中 37 个缺陷已被开发者修复。

（3）面向编译器缺陷定位的大模型赋能程序构造方法。传统编译器缺陷定位方法中证人测试程序的构造过程通常需要消耗大量资源且现有方法较少考虑程序的语义有效性，引入先进的大语言模型技术可以有效缓解以上难题。为了充分挖掘大语言模型程序构造的潜力，该构造方法解决以下两个挑战：1）如何制定精确的提示；2）如何选择针对每个缺陷的特定提示。为了制定精确提示，该方法设计了用于程序构造的特定提示模版，并结合数据流和控制流等静态程序分析技术提取程序的复杂度信息填充模版。为了选择特定提示，该方法首先采用深度强化学习策略，通过学习历史的经验知识并结合未来的探索知识，在缺陷定位过程中不断更新每个提示的价值权重，从而有效地选择针对不同缺陷的特定提示。其次，该方法采用轻量级测试程序验证策略，针对不同缺陷产生的不同验证结果产生相应的反馈提示作为大模型附加输入。实验结果表明，与现有方法相比，该方法能够有效地提高缺陷定位效率，在不同的实验配置下提高比例达 13.6% ～ 90.9%。此外，实验结果表明该方法具有较好的扩展性，能够适配不同的大语言模型以进一步辅助编译器缺陷定位。

本文提出的一系列新的程序构造方法对编译器测试及调试均取得了较好的效果，能够帮助开发者更好地对编译器缺陷进行检测和定位，进一步保障了编译器质量。

关键词：编译器质量保障；编译器测试；编译器调试；程序构造

ABSTRACT

Compilers are one of the key system software that is used for the translation of high-level source code into low-level machine code to accomplish various computing tasks. However, due to the intricate implementation of compilers, bugs in them are inevitable. Compiler bugs not only significantly affect the reliability of compilers but may also have catastrophic consequences. Therefore, assuring the quality of compilers is of critical importance. Compiler testing and debugging are two fundamental technique designed for compiler quality assurance: the former aims to detect compiler bugs, while the latter seeks to pinpoint the location of these bugs. Currently, researchers primarily employ program construction approaches to achieve effective compiler bug detection and isolation. Although many studies are devoted to testing and debugging compilers, existing program construction approaches exhibit several limitations in terms of effectiveness, such as the lack of syntactic diversity in test programs for detecting deep compiler front-end bugs, insufficient semantic diversity in test programs for detecting deep compiler middle-back-end bugs, and the ineffectiveness witness test programs used for compiler bug isolation, obstructing the process of assuring the quality of compilers.

To further enhance compiler quality assurance, this thesis proposes three novel program construction approaches to boost compiler testing and debugging techniques. To be specific, this thesis includes the following major studies:

(1) *Structure-aware test program construction for compiler front-end testing*. Given the complexities of detecting front-end compiler bugs, this thesis proposes CCOFT, aiming to address two main challenges: designing flexible generation structures and selecting representative generation structures. First, CCOFT utilizes the characteristics of syntactic definitions to construct diverse generation structures, enabling various granularity levels of mutation operations such as fine-grained bit-flip mutations and coarse-grained structural crossover mutations. Second, CCOFT adopts an efficient equal-probability selection strategy to uniformly select diverse representative syntactic structures. Moreover, to identify complex front-end compiler bugs effectively, CCOFT proposes a compiler output information alignment algorithm that can efficiently identify various types of front-end bugs. Experimental results demonstrate that CCOFT can significantly increase existing approaches in terms of the number of detected bugs, with an improvement ratio ranging from 111% to 135%. Furthermore, CCOFT detected 136 (67 confirmed or fixed) new bugs in two mature compilers.

(2) *Generator-remanufactured test program construction for compiler middle-back-end testing*. Program generators serve as the foundation of current compiler middle-back-end

testing techniques, but existing generators can hardly trigger new compiler middle-back-end bugs. To enhance the compiler bug detection capabilities of generators, this thesis proposes RemGen, targeting to address two main challenges: how to synthesize lightweight and diverse program fragments, and how to effectively select the optimal program fragments. First, RemGen constructs a component for synthesizing diverse program fragments using lightweight synthesis, leveraging the high code reusability characteristics in program generators. Second, RemGen establishes a component for selecting the optimal program fragments based on grammar coverage metrics collected during the synthesis process. Specifically, such a component employs high syntax coverage standards to select the optimal program fragments. By regenerating an existing program generator (i.e., CCG) to RemCCG, experimental results demonstrate that RemCCG can effectively improve the bug detection capability of CCG, achieving an improvement from 10% to 16% in terms of the number of detected bugs. Additionally, RemCCG detects 56 (37 confirmed or fixed) new compiler bugs.

(3) *LLMs-empowered witness test program construction for compiler bug isolation.* Traditional witness test program construction often consumes significant resources and does not sufficiently consider the semantic validity of the generated programs. Leveraging potentials in LLMs can mitigate this predicament. However, obtaining accurate prompts to effectively control LLMs' performance remains challenging. To harness the program construction capabilities of large language models, this thesis proposes LLM4CBI to tackle two main challenges: formulation of accurate prompts and selection of specific prompts tailored to each bug. To devise precise prompts, LLM4CBI designs specific prompt templates for program construction, combined with static program analysis techniques such as data flow and control flow analysis to extract crucial information for populating the templates. To select specific prompts, LLM4CBI combines a deep reinforcement learning strategy that continuously updates the value weights of each prompt based on historical experiential knowledge and exploration knowledge and a lightweight program validation component to effectively select specific prompts for each bug. Experimental results demonstrate that LLM4CBI can significantly enhance bug isolation efficiency compared to existing methods, achieving an improvement ratio ranging from 13.6% to 90.9%. Moreover, experimental results confirm the extendibility of LLM4CBI for adapting more LLMs to assist compiler bug isolation.

The three new program construction approaches proposed in this thesis have achieved promising results in compiler testing and debugging, assuring the quality of compilers.

Key Words: Compiler Quality Assurance; Compler Testing; Compiler Debugging; Compiler Bug Isolation; Test Program Construction

目　　录

TABLE OF CONTENTS

图 目 录

表 目 录

主要符号表

符号	代表意义
英文字母	
C++99/11	C++ 语言 1999/2011 标准
c_i/C_i	某个待测编译器
e_i	编译器输出的错误诊断信息记录
Ochiai	用于对潜在缺陷位置进行排序的相似系数
名词缩写	
GCC	**G**NU **C**ompiler **C**ollection
LLVM	**L**ow **L**evel **V**irtual **M**achine
RDT	**R**andomized **D**ifferential **T**esting
DOL	**D**ifferent **O**ptimization **L**evels
EMI	**E**quivalence **M**odulo **I**nputs
CCG	**C** **C**ode **G**enerator
YARPGen	**Y**et **A**nother **R**andom **P**rogram **G**enerator
SBFL	**S**pectrum-**B**ased **F**ault **L**ocalization
CCOPT	**C**++ **CO**mpiler **F**ront-end **T**ester
ECS	**E**qual-**C**hance **S**election
CTD	**C**rash or **T**ime-out **D**etecting
CVS	**C**ross-**V**ersion **S**trategy
CSS	**C**ross-**S**tandard **S**trategy
RQ	**R**esearch **Q**uestion
RemGen	**Rem**anufacturing **Gen**erators
LLM4CBI	**L**arge **L**anguage **M**odels for **C**ompiler **B**ug **I**solation
LLM	**L**arge **L**anguage **M**odel
ANN	**A**ctor **N**eural **N**etwork
CNN	**C**ritic **N**eural **N**etwork
MFR	**M**ean **F**irst **R**anking
MAR	**M**ean **A**verage **R**anking

注：如文中对符号另有说明，以文中对应位置说明为准。

1 绪论

编译器是重要的基础系统软件，也是软件开发工具链中必不可少的一环。在软件开发过程中，无论是编写简易的应用界面程序还是复杂的操作系统程序，其背后均离不开编译器的支持。与其他软件系统一样，编译器由于其实现逻辑的复杂性和代码数量的庞大性，难免存在缺陷。然而，不同于普通软件的缺陷，编译器缺陷不仅会影响编译器的可靠性，也会影响基于其编译软件的正确运行。当使用有缺陷的编译器对软件进行编译时，可能会导致软件错误执行或者出现异常行为，甚至带来严重的经济损失。总之，编译器缺陷对构建安全攸关的关键系统正确性和可靠性构成了巨大威胁。因此，设计有效的方法保障编译器质量是至关重要的。

随着软件开发技术的不断发展，编译器质量保障问题受到了学术界和工业界的广泛关注[1]。研究者们提出了保障编译器正确性和可靠性的系列方案[2–6]。其中，基于程序构造的编译器测试及调试是编译器质量保障的主流方法之一。具体而言，编译器测试旨在通过构造多样化的测试程序检测编译器缺陷；编译器调试旨在通过构造权衡多样化和相似化的证人测试程序定位编译器缺陷。然而，由于编译器测试及调试场景中对输入测试程序要求的特殊性，现有的程序构造技术仍然不能很好地生成满足特定需求的有效程序。因此，如何构造有效的测试程序以更好地辅助编译器测试及调试仍然是一个值得研究的问题。为此，本文从程序构造的角度出发，针对编译器测试及调试中不同场景下对测试程序不同需求的问题，提出构造满足特定需求的测试程序方法，以提高编译器测试及调试效率，从而进一步保障编译器质量。

本章首先介绍本文的研究背景和意义，包括编译器质量保障、编译器测试及编译器调试，然后引出本文的主要工作内容，最后阐述本文的组织结构。

1.1 研究背景与意义

1.1.1 编译器质量保障

编译器作为重要的基础软件之一，其核心任务是将高级编程语言（如 C/C++）编写的源代码转化为机器代码，以供计算机直接执行[7,8]。编译器在计算机科学与软件开发中扮演着至关重要的角色[9]。其重要性体现在以下几个方面：1）连接高级语言与机器语言：编译器允许开发者采用高级编程语言（如 C++、Java 等）编写程序，然后将其转换为计算机可以直接理解的机器语言，从而大大简化了软件开发过程；2）代码优化：现代编译器不仅仅是进行代码转换，它们还可以优化代码，使得生成的机器代码运行更快、规模更小；3）跨平台开发：通过为不同的目标架构或平台提供不同的编译器，开发者只需要编写一次代码，然后进行重新编译即可在多个平台上运行；4）

错误检测：编译器在编译过程中能够检测到许多编程错误，从而在代码实际运行之前帮助开发者找到并修复相关编程错误；5）安全性：编译器可以为生成的代码增加某些安全特性，如堆栈保护、地址空间布局随机化等，以减少潜在的安全威胁；6）软件生态系统发展：各式各样的编程语言和工具链的存在，均离不开对应的编译器技术支持。编译器推动了编程语言的快速发展。综上所述，编译器不仅促进了高效、跨平台的软件开发，还为代码的优化、安全性以及错误检测等需求提供了支持。编译器的以上优点使其成为了推动计算机科学进步的关键要素之一。

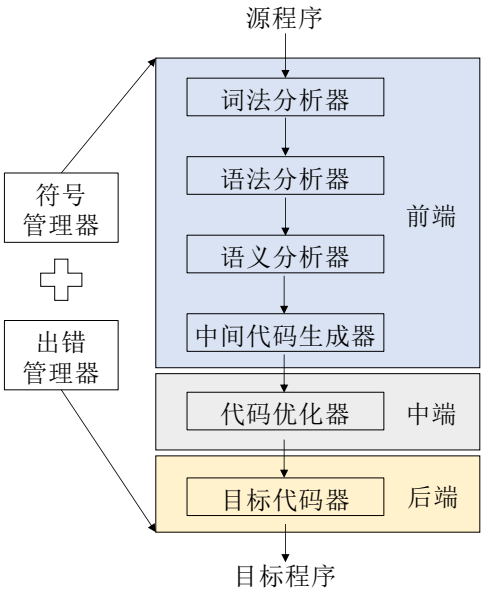


图 1.1　编译器基本组成

Fig. 1.1　The main components in a compiler

根据编译器的流程，编译器的基本组成如图1.1所示，主要的编译功能由三个部分完成[7]，即前端、中端和后端。编译器前端分析主要验证程序词法、语法及语义的正确性，并生成机器无关的中间代码。具体而言，为了确保源代码在语法结构上的正确性，前端通过词法分析将源代码拆分为基本词汇单元或记号并采用语法分析将该单元或记号构建为抽象语法树或其他中间表示。编译器中后端对中间代码进行优化，最后生成目标代码，即特定平台的机器语言代码。就编译器的功能而言，编译器主要用于解析、转化和优化源代码，生成可供机器直接执行的代码，同时在此过程中报告和处理可能出现的编程错误。在编译过程中，编译器还辅助采用全局定义的符号管理器和出错管理器对整个编译器流程进行管理。符号管理器负责管理编译过程中用到的各种符号，通常通过符号表（Symbol Table）来实现。符号表存储了源程序中变量、函数、类、对象等的信息，如类型、作用域以及内存地址等。出错管理器负责在编译过程中检测、报告和处理错误。编译过程中可能会遇到各种类型的错误，例如语法错误、类型不匹配错误以及未声明的符号错误等。所有的组件在完成自身任务的同时与其他组

件互相合作，为编译器的正常工作提供了有力的保障。

然而，和普通软件一样，编译器实各个部分的实现难免存在缺陷。在前端实现中，词法分析可能未能正确识别出某些字符或字符串作为有效的记号，导致“未识别记号”缺陷；或者在处理转义字符或字符串字面量时可能出错，导致“扫描”缺陷；或者在处理非标准或多种编码的源文件时出现问题，导致“编码”缺陷。在语法分析中，编译器对于某些输入无法构建有效的解析树，会导致“解析”缺陷；或者在处理复杂的算术表达式时出错，导致“优先级和结合性”缺陷；或者输入程序中存在某些文法使得编译器不能确定如何解析，导致“模糊文法解析”缺陷。在前端语义分析中存在的缺陷种类更是多种多样。例如，编译器试图访问未声明或已超出作用域的变量而没有给出任何警告信息，意味着编译器前端中的警告系统存在缺陷。其次，由于高级程序语言程序语义的复杂性，编译器前端可能对某段有效的程序发出错误警告信息，也意味着编译器前端在处理语义分析时存在缺陷。另外，编译器可能将前端产生的中间表示转化为不正确的中间代码，从而导致编译器中后端在中间代码优化上出现缺陷。在代码优化过程中，编译器采用的相关优化可能无法真正提高代码的性能，或者在某些情况下甚至会降低性能，导致编译器优化的性能缺陷。其次，编译器可能会由于无法识别可显著提高性能的优化机会，从而导致性能无法优化，最终导致编译器的性能缺陷。另外，编译器的某些优化可能会改变程序的预期行为，导致编译器生成错误的二进制代码。在编译器后端目标代码生成中，编译器可能为特定操作选择了非最佳的机器指令，导致指令选择错误。其次，在分配寄存器时可能导致频繁的数据移动或不必要的内存访问，导致编译器寄存器分配出现问题。另外，目标代码生成处理函数调用或返回时处理参数或返回值的方式不符合目标平台的应用程序二进制接口（ABI），导致错误的 ABI 实现。以上仅仅是编译器前中后端可能出现的基本缺陷类型。在实际应用中，特定编程语言的编译器可能存在其他更为复杂的缺陷类型。

与普通软件的缺陷不同，编译器缺陷可能带来灾难性的后果[2,4,5,10]。对于编译器前端缺陷，Apache Struts 2 2.3.x 在 2.3.32 版本之前和 2.5.x 在 2.5.10.1 版本之前的 Jakarta 解析器在文件上传尝试期间存在处理异常和生成不正确的错误消息等问题，使得远程攻击者可以通过精心构造的 Content-Type、Content-Disposition 或 Content-Length HTTP 头部来执行任意命令，造成了恶意攻击者以一个包含“#cmd=string”的 Content-Type 头部成功发起攻击的后果[11]。对于编译器中后端缺陷，著名的 Unix 工具 sudo 因为被错误版本的编译器（即 LLVM-3.3）编译，导致提权安全漏洞[1]，使得用户可以使用超级用户权限访问任何数据。此外，编译器的漏洞甚至会被恶意注入，以危害基于该编译器编译的应用程序的安全性。以著名的 XcodeGhost 事件为例，苹果 Xcode[2]编译器的一个恶意变种被引入了后门，对所有采用它编译的应用程序的安全性产生了严重

[1]http://blog.thetaphi.de/2011/07/real-story-behind-java-7-ga-bugs.html

[2]https://en.wikipedia.org/wiki/XcodeGhost

威胁。在 Xcode 事件中，超过 4000 款手机应用程序受到了影响，其中包括微信和银行应用等安全攸关的程序。综上所述，编译器缺陷，无论是前端还是中后端缺陷，均可能对软件的质量、性能和安全性产生严重影响。前端的缺陷可能导致代码的逻辑错误、性能下降或被攻击者利用的潜在安全隐患，而中后端的缺陷可能引起生成代码的不稳定、效率低下、平台兼容性问题或安全漏洞。因此，如何提高编译器的可靠性对保障编译器的整体质量至关重要。

然而，有效保障编译器的质量并不容易。在软件开发过程中出现错误或在软件运行后出现不符预期的行为时，软件开发人员通常难以确定错误的源头是自身编写的软件错误还是编译器本身存在缺陷[8]。通常情况下，开发人员倾向于自我推定在软件开发过程中引入了缺陷。为了确保软件的顺利运行，他们通常需要耗费大量人力和物力进行程序调试，但最终可能会发现问题的根本原因在于编译器中存在的缺陷，严重降低了软件开发效率。更为严重的是，当编译器开发人员发现某个缺陷属于编译器本身时，如何在数量庞大的编译器源文件中定位该缺陷的出错位置也十分困难[12–14]。以上种种原因导致编译器开发者修复缺陷的进程缓慢，加深了对软件开发效率的影响。

为了提高编译器的可靠性，研究者们通常采用编译器测试及调试方法对编译器的质量进行保障。后续章节介绍编译器测试及调试的相关背景。

1.1.2 编译器测试

编译器测试的必要性在于通过测试的方式尽可能确保编译器能够正确、高效地转换源代码为目标机器代码。编译器是软件开发中的核心工具，其正确性直接影响到最终应用的性能和可靠性。任何编译器中的缺陷或错误均可能导致生成的代码中出现难以检测的问题，导致软件开发成本的增加、产品发布的延误，并可能导致严重的经济损失。因此，对编译器进行全面的测试可以确保其按照预期工作，可以为开发者提供一个稳定、可靠的开发环境，并同时确保编译后的应用具有预期的性能和功能。

与其他软件测试相似，编译器测试通常涵盖三个主要阶段：测试程序构造、测试预言构建以及测试程序约简。具体流程如图1.2所示。其中，测试程序构造旨在构造有效的编译器测试输入来测试编译器，测试预言构建旨在判定编译器行为在给定的测试程序下是否符合预期，测试程序约简旨在对触发编译器缺陷的程序进行精简化以帮助开发者复现、定位以及修复缺陷。

（1）测试程序构造

测试程序构造是编译器测试流程中的关键步骤，也是最重要的步骤。然而，由于编译器处理的输入为高度结构化且复杂多变的程序，因此编写编译器输入测试程序必须深入理解各种编程语言的语义特点，给测试程序的自动化构造带来诸多挑战。如果向编译器提供了不合规范的输入程序，会导致该测试程序在编译器前端分析阶段被过滤。以强类型语言如 C、C++ 及 Java 为例，程序中引用的用户定义标识符必须先经过

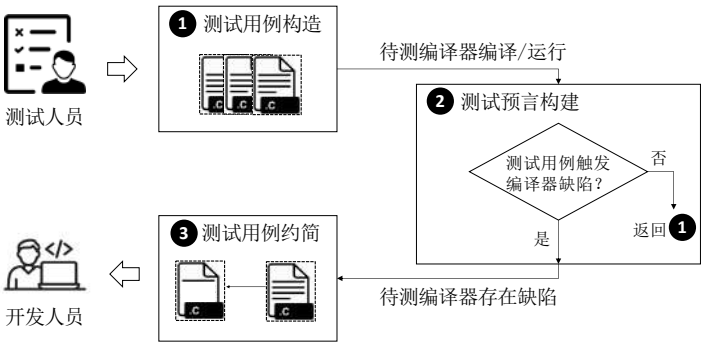


图 1.2　编译器测试基本流程

Fig. 1.2　Process of compiler testing

类型声明与初始化，否则将会生成不合格的测试程序。不合格的程序会在编译器前端语法检查时被拒绝，不会进入优化阶段，从而导致中后端测试效率受到影响。因此，与其他软件测试类似，为编译器测试准备的测试程序应具备丰富性。具备丰富语法特点的测试程序可以触及编译器的各个模块，有助于揭露其潜在问题。再者，由于不同的语法结构通常具有复杂的数据流或者控制流，因此采用涵盖多种语法结构的测试程序更有可能揭示编译器源代码实现中的不足或尚未被开发者关注的特殊情况。综上所述，测试程序质量直接决定了编译器测试的效果。

给定一个测试程序，编译器的各个阶段（即前端和中后端）共同参与了将源程序转换为目标程序的过程。在整个过程中，只有各个编译器组件被正确实现了才能有效保证整个编译流程的正确性。由于不同的组件所侧重任务的重点不同，不同的编译器组件对所需的测试程序有着不同的需求。

① 面向编译器前端测试的程序构造。 面向编译器前端测试的程序同样具有多个显著特征。首先，它们涵盖了广泛的测试领域，其中包括语法测试，以验证编译器对编程语言语法结构的正确解析；语义测试，以验证编译器对源代码的正确语义解释；错误处理测试，以确保编译器能够适当地检测和报告各种错误；性能测试，以评估编译器前端在速度和资源利用方面的性能；兼容性测试，以确保编译器前端能够处理不同标准版本的编程语言。以上特征共同确保编译器前端能够高效、准确、且稳定地解析和处理源代码，从而满足编译器的质量和性能要求。因此，用于有效测试编译器前端的测试程序应当具有词法及语法多样化的特点，以此充分测试编译器前端。

② 面向编译器中后端测试的程序构造。 面向编译器中后端测试的程序具有多个显著特征。首先，为了确保生成的目标代码在不同目标平台上能够正确运行，用于测试中后端的测试程序主要集中在编译器的代码优化及生成阶段。该阶段包括对生成的汇编代码、机器代码或中间代码的正确性进行验证。其次，性能测试用于评估编译器生成的代码在不同情况下的性能表现，包括执行速度和资源利用，以确保生成的代码在运行时具有卓越的性能。优化测试的目的是验证编译器循环展开、常量折叠和死代

码消除等各种代码优化技术是否正确应用，以提高生成代码的效率和质量。寄存器分配测试则旨在检查编译器是否能够有效分配寄存器，以优化目标代码的寄存器分配使用。最后，内存管理测试关注编译器的内存管理策略，以确保生成的代码对内存资源的使用是高效而可靠的。以上特点共同确保编译器后端在生成代码时能够满足性能、效率和质量方面的要求。因此，用于有效测试编译器中后端的测试程序应当具有语法正确且语义多样化的特点，从而尽可能检测出深层次编译器中后端优化缺陷。

（2）测试预言构建

测试预言是指在给定测试程序的情况下，评估编译器的执行是否符合预期的过程。总体而言，编译器的输入是由高级编程语言编写的源代码，其输出是能在目标计算机上执行的可执行程序或者编程错误。一方面，测试预言可以在测试程序经过前端编译阶段出现问题时，用来验证编译器的前端编译结果是否与正常的前端编译器输出一致。另一方面，测试预言可以在测试程序经过编译器中后端复杂的优化后，用来验证可执行程序的运行结果是否与原始测试程序的语义一致。具体而言，首先将相同的测试程序输入到多个不同的编译器，然后通过采用差分测试来比较它们的输出结果，最后如果发现输出结果不一致，则表明某个编译器可能存在缺陷[8,15]。预言构建的方法有助于判断潜在的编译器缺陷是否存在，确保编译器在不同情况下能够正确编译源代码和优化中间代码并生成正确的机器目标代码。

（3）测试程序约简

测试程序约简是对触发编译器缺陷的测试程序进行最小化精简的一项操作，该操作同时确保被约简的程序仍然能够触发与未精简源测试程序相同的编译器缺陷。一般情况下，能够引发编译器缺陷的测试程序通常具有庞大且复杂的结构，使得编译器开发人员难以直接确定缺陷出现在编译器的哪个部分。经过约简后的测试程序通常包含较少的代码行数（一般少于 20 行[15,16]）。例如，C-Reduce[17] 工具通过一系列的迭代变换，能够将测试程序的规模缩减到 10 行左右。约简后的测试程序不仅能够复现编译器缺陷，还有助于编译器开发人员快速而准确地理解导致编译器缺陷的根本原因[15]。因此，在编译器测试的过程中，触发缺陷的源程序通常需要进行约简（通常少于 15 行[18]），经过约简的测试程序可以直接提交到相应的编译器缺陷管理仓库中，供编译器开发人员用于复现并及时修复缺陷。

1.1.3 编译器缺陷定位

编译器缺陷定位的必要性体现在对于确保编译缺陷修复的效率至关重要。当编译器出现问题，仅仅知道存在缺陷是不够的，还需要精确地知道缺陷出现的位置（文件级，即出错的文件），以便进行快速修复。如果不能迅速和准确地定位编译器的缺陷，可能会导致更多的编译错误、性能问题或运行时异常。延迟的修复过程不仅会浪费开发者的时间和资源，还可能对最终的软件产品造成不可预测的影响。因此，为了高效

地诊断和修复缺陷，有必要提升缺陷定位的效率，确保最终生成代码的质量和性能。

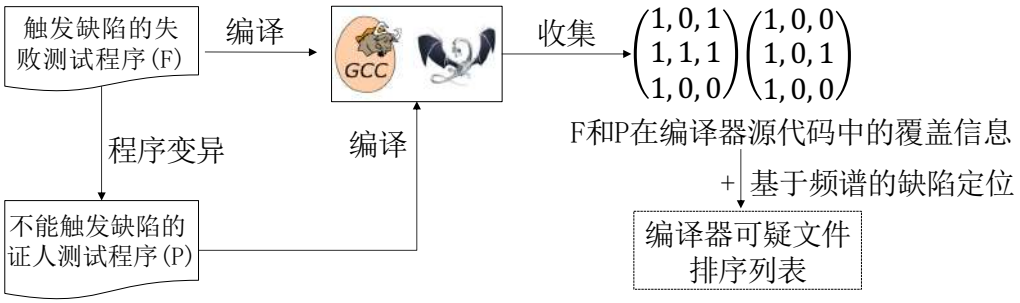


图 1.3　典型的基于测试用例的软件缺陷定位流程

Fig. 1.3　Process of traditional compiler bug isolation

编译器缺陷定位是编译器调试的重要一环[12,13,19–22]。现有的编译器缺陷定位方法，其工作流程如图1.3所示。给定可以触发编译器缺陷的测试程序 F，现有方法首先采用不同的变异策略生成一个不会触发该缺陷的证人测试程序 P。随后，两个测试程序均会被目标编译器（如 GCC 或者 LLVM）编译，以收集编译器源文件的代码覆盖信息。值得注意的是，在编译过程中，任何由失败测试程序覆盖的编译器源文件均被视为可疑的。相反，不能触发缺陷的测试程序用于降低可疑文件存在缺陷的可疑度。为了最终定位出存在缺陷的文件，现有方法依据基于频谱的缺陷定位（SBFL）策略并结合 Ochiai 公式[23]，比较可触发缺陷测试程序和不可触发缺陷测试程序之间编译器源代码的覆盖率。最终计算出每个文件的可疑率，得出可疑文件的排序列表。

1.2　本文主要工作

为了进一步保障编译器的质量，本文围绕面向编译器测试和调试的程序构造方法中尚未完全解决的问题展开研究，研究框架如图1.4所示。它们是：1）如何构造语法多样化的测试程序更好地检测出深层次的编译器前端缺陷；2）如何构造语义多样化的测试程序更好地检测出深层次的编译器中后端缺陷；3）如何构造语义完全有效且权衡多样化和相似化的证人测试程序更好地辅助编译器缺陷定位。特别地，针对不同的场景，三个工作分别构造词法、语法和语义有效的测试程序，呈递进关系（详见表1.1关于本文构造测试程序的联系和区别）。接下来将针对每个具体场景，概述研究目标、现存问题、本文提出解决方法的主要内容以及本文的主要贡献。

1.2.1　面向编译器前端测试的结构感知程序构造方法

（1）研究目标

本研究旨在构造语法多样化的测试程序检测深层次的编译器前端缺陷。

（2）现存问题

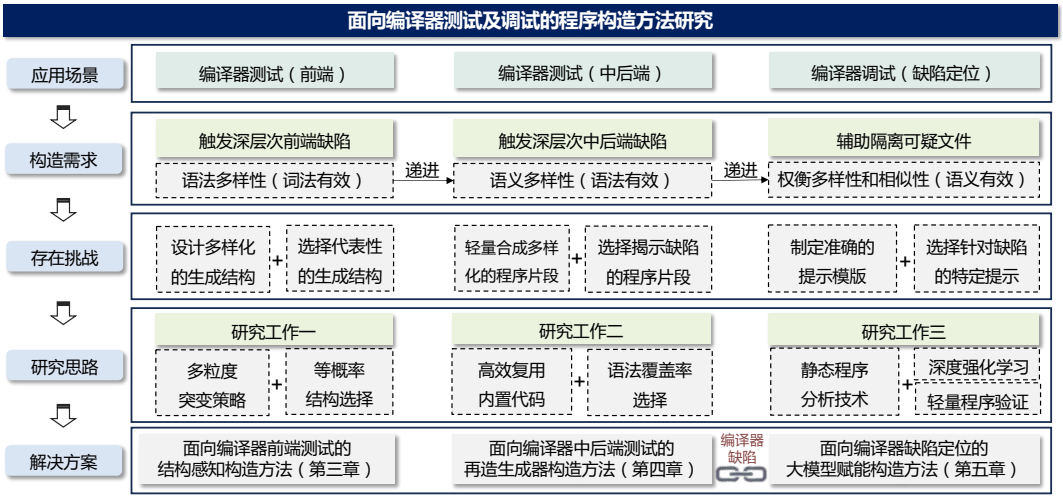


图 1.4　本文研究框架

Fig. 1.4　The research framework of this thesis

尽管已经有不少文献已经对编译器测试进行研究，但很少有研究专注于测试 C++ 编译器前端。现有的程序构造方法如 Csmith[2] 和 YARPGen[3] 主要生成完全语义有效的 C++ 测试程序，即满足所有语法和类型检查规则。由于该类程序会很快通过编译器前端，导致现有程序构造方法构造出的 C++ 程序很难检测出编译器前端中产生潜在的缺陷。其他程序生成器，如 Dharma 和 Grammarinator，可以生成词法无效的测试程序测试编译器前端。然而，该类测试程序只能检测到浅层次的前端缺陷。由于 C++ 语法的复杂性以及测试手写 C++ 编译器前端的难度，需要解决两项技术挑战才能有效地检测出深层次的 C++ 编译器前端缺陷。第一个挑战是如何设计灵活化的结构以实现多粒度结构化变异，第二个挑战是如何选择代表性的结构以生成多样化的测试程序。此外，如何从复杂的编译器输出中识别缺陷也具有一定的挑战性。

（3）主要内容

为了检测深层次的编译器前端缺陷，本文设计了新的程序构造方法 CCOFT。该方法旨在生成语法多样化的测试程序检测深层次的 C++ 编译器前端缺陷。首先，为了解决第一个挑战，CCOFT 实现了通用的 C++ 程序生成器。具体而言，生成器先将 C++ 语法转换为灵活的结构化格式，以支持多粒度结合的变异操作，如细粒度的位反转变异和粗粒度的结构化交叉变异等。为了解决第二个挑战，CCOFT 采用高效的平等机会选择（ECS）策略进行结构感知语法突变，均匀地选择多样化为代表的语法结构，最终生成多样化的 C++ 测试程序。此外，CCOFT 采用一套基于差分测试的策略，通过比较不一致的编译器输出来识别编译器前端中不同类型的缺陷。

（4）主要贡献

① 本文提出了旨在检测深层次 C++ 编译器前端缺陷的测试框架 CCOFT。

② 设计了基于结构化变异（配备 ECS）的程序构造方法，并利用一组基于差分

测试的策略和信息对齐算法来识别各种类型的前端缺陷。

③ 实现了 CCOFT，并将 CCOFT 与两种最先进的方法进行实证评估。实验结果表明在检测到的缺陷数量方面优于现有的两种最先进方法（Dharma 和 Grammarinator），提高分别达 135% 和 111%。

④ 向 GCC 和 LLVM 开发者报告了共 136 个编译器前端缺陷，其中 67 个已被开发者确认、已分配或修复。

1.2.2 面向编译器中后端测试的再造生成器程序构造方法

（1）研究目标

本研究旨在构造语义多样化的测试程序检测深层次编译器中后端缺陷。

（2）现存问题

高质量的程序生成器是现有检测编译器中后端缺陷方法的重要组成部分。然而，现有程序生成器难以直接检测出新的缺陷。例如，Csmith[2] 作为最著名的程序生成器之一，已有研究[6] 表明编译器已经对 Csmith 生成的测试程序具有了很强的抵抗力。不仅如此，许多活跃的研究员/开发者深刻地抱怨了该问题。本章旨在探究以下研究问题：是否有可能提高现有程序生成器的缺陷检测能力以从根本上提高编译器测试的效率，从而达到检测深层次编译器中后端缺陷的目的？再制造是一项把旧的失效产品恢复到重新有效产品的技术，然而，将再制造的方案应用到程序生成器上需要解决两个特定的挑战。第一个挑战是如何轻量化地合成多样化的代码片段。第二个挑战是如何选择揭示缺陷的代码片段以构造尽可能触发缺陷的测试程序。

（3）主要内容

为了检测深层次的编译器中后端缺陷，本文将再制造旧产品（如机器部件）的想法迁移到程序生成器中，并提出了基于再造生成器的程序构造方法 RemGen。具体而言，本文分别设计了两个新组件以解决两个挑战。第一个多样化代码片段合成组件采用语法辅助代码片段合成来生成各种代码片段。特别地，该组件首先在程序生成器中保留在生成过程中产生的上下文，然后通过调用生成器中的内置函数来利用上下文合成新的代码片段，从而解决多样化代码片段合成的挑战。接下来，本文提出了揭示缺陷的代码片段选择组件解决第二个挑战。该组件采用语法覆盖率指标来记录合成过程中所选代码片段语法规则的使用频率。在后续阶段中，该记录的覆盖率用于衡量每个合成代码片段的多样性。在覆盖率指标的指导下，RemGen 选择具有最多样化语法覆盖的代码片段来构建新的缺陷揭示测试程序。最终，生成的测试程序用来测试编译器中后端，并将检测到的缺陷提交给 GCC 和 LLVM 编译器开发者。

（4）主要贡献

① 本文提出将再制造的想法迁移至程序生成器，并设计了程序构造方法 RemGen 以对现有的程序生成器进行再制造。

② 在 RemGen 中设计了多样化代码片段合成和揭示缺陷的代码片段选择组件有效地对程序生成器进行再制造。

③ 通过 RemGen 框架对现有的程序生成器 CCG 进行了再制造，实现了工具 RemCCG。实验证明了 RemCCG 优于现有的基于生成和突变的程序构造方法。

④ 向 GCC 和 LLVM 开发者报告了共 56 个编译器中后端缺陷，其中 37 个已经被开发者修复。

1.2.3 面向编译器缺陷定位的大模型赋能程序构造方法

（1）研究目标

本研究旨在构造语义完全有效且权衡多样化和相似化的证人测试程序以辅助编译器缺陷定位。

（2）现存问题

当编译器缺陷被检测出来后，如何及时有效地定位缺陷是保障编译器质量的一项关键任务。然而，现有的编译器缺陷定位方法（如 DiWi[12] 和 RecBi[13]）不仅在突变策略的有效性方面存在一定的局限性，而且程序构造的过程也需要大量的人力干预，以上局限性严重影响了证人测试程序构造方法的有效性。随着大语言模型的快速发展以及其在代码生成方面的优异性能，采用基于大模型的方法生成满足语义完全有效的证人测试程序可能是一个较好的选择。然而，需要解决以下两个重要挑战才能有效激发大语言模型中的潜能。由于提示的质量对挖掘大模型的潜能十分重要，第一个挑战是如何制定精确的提示。因为不同的编译器缺陷对证人测试程序的要求不同，因此有必要针对每个缺陷选择专用的提示，即第二个挑战是如何选择专用提示。

（3）主要内容

为了更好地定位编译器缺陷，本文提出新的证人测试程序构造方法 LLM4CBI，旨在解决以上两个挑战。为了充分挖掘大模型技术潜能，LLM4CBI 设计了三个新组件用于有效地生成证人测试程序以辅助编译器缺陷定位。首先，设计精确提示生成组件组件来解决第一个挑战。该组件首先引入了精确的提示模版，该模版可以准确表示所需的突变操作。然后，利用通过数据和控制流分析测量的程序复杂度指标来识别最相关的变量和最佳插入位置以填充模版。其次，提出了两个新组件，即记忆提示选择组件和轻量级测试程序验证组件，用于选择针对每个缺陷的专用提示。在提示选择组件中，LLM4CBI 通过强化学习结合记忆搜索，根据大模型的表现跟踪和积累奖励，指导 LLM4CBI 不断选择专门的提示来改变特定的测试程序。在测试程序验证组件中，LLM4CBI 利用静态分析来检测和过滤掉可能包含未定义行为的无效测试程序，从而降低与无效程序相关的风险。最终，构造出的测试程序与 SBFL 技术相结合并对可疑文件进行排序，输出的排序列表可以帮助开发者快速定位存在缺陷的文件。

（4）主要贡献

① 本文提出的 LLM4CBI 是首个旨在挖掘大语言模型潜能以进行编译器缺陷定位任务的工作。

② 在 LLM4CBI 中提出了三个新组件，即精确提示生成、记忆提示选择和轻量级测试程序验证组件，以指导大模型构造出有效的证人测试程序辅助编译器缺陷定位。

③ 通过实证评估证明了 LLM4CBI 的有效性。实验结果表明，LLM4CBI 对编译器缺陷定位有效，并且可以扩展到其他大语言模型。

④ LLM4CBI 为编译器缺陷定位的未来研究拓展了新思路，同时为进一步探索和挖掘大语言模型的潜在功能提供了切实可行的途径。

1.2.4 三个程序构造方法的联系和区别

虽然本文的三个主要研究内容（即 CCOFT，RemGen 以及 LLM4CBI）均旨在构造有效的测试程序，但是三个方法构造出的测试程序之间是有联系和区别的（具体如下表1.1所示）。换而言之，三个研究内容在构造结果上是独立的（即三个工作构造出来的测试程序不能相互使用），而在构造目标上是有连续性的（即三个工作根据编译器基本结构和编译器缺陷的生命周期进行联系）。表中的✓ 表示在某个阶段有效，✗ 表示在某个阶段无效，***Na*** 表示可能有效也可能无效。值得一提的是，***Na*** 类的测试程序可以增加程序多样性，提高不同方法检测编译器不同阶段缺陷的能力。

表 1.1 本文构造的程序在有效性/场景/需求方面的总结

Tab. 1.1 Validity/scenario/requirement of newly constructed test program in this thesis

本文方法	词法有效	语法有效	语义有效		面向场景	测试程序需求
			类型匹配	定义行为		
CCOFT	✓	***Na***	✗	✗	前端测试	语法多样化
RemGen	✓	✓	✓	***Na***	中后端测试	语义多样化
LLM4CBI	✓	✓	✓	✓	缺陷定位	权衡多样/相似化

（1）联系

根据图1.1所示编译器框架图，表中将程序的有效性分为三个阶段，分别为词法分析阶段、语法分析阶段以及语义分析阶段。因此，本文提出的三个方法遵循编译器工作的基本流程，即均可能经过编译器的词法、语法与语义检查阶段。

（2）区别

面向不同的场景，本文构造的测试程序满足了针对不同场景的特定需求。具体来说，为了检测出深层次的编译器前端缺陷，该阶段对测试程序的需求为语法多样化，即语法可能有效但因为不能通过类型检查而语义无效。为了检测出深层次的编译器中后端缺陷，该阶段对测试程序的需求为语义多样化，即语法完全有效但可能含有未定义行为而语义无效。为了辅助编译器缺陷定位，该阶段对证人测试程序的需求为语义

完全有效且权衡多样化和相似化。值得注意的是，在缺陷定位场景中，本文提出的方法对任何编译器缺陷均可实现定位。但是，由于前端部分的缺陷的定位较易而中后端缺陷的定位极具挑战性，本文默认考虑定位编译器中后端缺陷。

总而言之，面向编译器测试及调试的不同场景，本文提出构造满足特定需求的测试程序方法达到进一步保障编译器质量的目的。

1.3 本文组织结构

本文共分为六个章节，每个章节的内容如下：

第一章为绪论。首先介绍编译器缺陷检测及定位的研究背景和意义，然后阐述本文主要研究工作，包括各个工作的目标、针对的问题、主要研究内容及主要贡献。

第二章为相关工作。总结分析编译器缺陷检测与定位的国内外研究进展与现状。

第三章研究面向编译器前端测试的结构感知程序构造方法。针对现有程序构造方法中缺乏能够有效检测深层次编译器前端缺陷的测试程序问题，本章详细描述并充分评估了提出的 CCOFT 方法。

第四章研究面向编译器中后端测试的再造生成器程序构造方法。针对现有程序构造方法中缺乏能够有效检测深层次编译器中后端缺陷的测试程序问题，本章详细描述并充分评估了提出的 RemGen 方法。

第五章研究面向编译器缺陷定位的大模型赋能程序构造方法。针对现有程序构造方法中缺乏能够有效辅助编译器缺陷定位的证人测试程序及构造开销较大问题，本章详细描述并充分评估了提出的 LLM4CBI 方法。

第六章为结论及展望。总结全文的主要工作内容和创新点，并展望未来的研究。

2 国内外研究现状

目前，国内外已有众多研究围绕编译器缺陷检测及定位展开[24–26]。针对编译器测试，一部分研究致力于解决测试程序构造中遇到的诸多问题，一部分研究则关注测试预言的构建技术，其他研究则是提出测试程序的约简策略。针对编译器调试，研究者提出了基于程序构造的方法及基于其他技术的定位方法。为了更好地理解本文内容，本章将从编译器测试与调试的角度出发，概述其在国内及国际的研究现状。

2.1 编译器测试

本节对编译器测试流程（如图1.2所示）中的三个关键步骤的研究现状进行综述，即测试程序构造、测试预言构建以及测试程序约简。

2.1.1 编译器测试程序构造

目前已有不同的技术方法（如符号执行[27]）用于自动化生成测试用例并对测试对象进行详尽测试。针对编译器测试对象，根据已有综述[8] 总结，用于编译器测试的测试程序主要通过基于规约的方法构造，主流的构造方法一般分为两类。一类是指从无到有的基于规约直接构造测试程序，另一类是指通过变异的方式，在已有的测试程序上进行变异操作从而构造出新的测试程序。根据编译器的不同组件，本小节分别阐述面向编译器前端和中后端的测试程序构造。

（1）面向编译器前端测试的测试程序构造方法

编译器前端测试需要词法及语法多样化的测试程序。由于该类测试程序的特殊性，目前较少有专门面向编译器前端缺陷检测的研究。

Sun 等人[28] 最早提出 Epiphron 工具用于检测编译器前端警告机制中的缺陷。理想情况下，编译器前端警告系统应当对潜在的不安全或有问题的代码片段给予准确而有意义的反馈信息，以辅助编程人员更好地完成编码任务。然而，编译器的警告系统不够可靠。具体而言，如果编译器给出了误导性的警告或未给出应有的警告，则意味着编译器前端警告机制存在缺陷。为了有效检测编译器中的警告缺陷，Epiphron 采用 C 语言语法为输入，并随机选取部分语法规范子集产生测试程序。在产生程序的过程中，该工具在条件语句和循环内随机插入有效的额外的语句：如在循环语句中插入 “break” 语句，或者在 “if” 语句中插入一条空语句。另外，Epiphron 维护了 C 语言中的未定义与未指定行为，以触发更多相关且有效的警告信息。实验表明，在六个月时间内，该工具已在 GCC 和 LLVM 编译器中检测到了共 99 个警告缺陷。

Wolf 等人[29] 采用手动的方法构造测试程序以验证 Arden2ByteCode 编译器是否可以被 Arden 的所有语法规约覆盖。该方法首先根据 Arden 的语法描述、特例程序及

编程实例构造预期的测试程序，然后利用 JUnit 框架进行形式化验证测试程序。最后，为了更好地完成编译器适配测试，该方法在框架上设计了相应的 Java 接口以供 Arden 处理器采用。实验结果表明，通过执行该方法构造的 161 个有效测试程序，检测出了 14 个测试程序的语法规约没有在 Arden 编译器中得到覆盖。

Hodovan 等人[30] 设计了 Grammarianor 工具。该工具是一种基于语法变异的测试程序生成工具。该方法基于种子测试程序的语法，通过变异其语法规则来构造新的 AST，从而产生新的测试程序。Grammarianor 能够产生含有语法错误的测试程序，适用于对编译器前端的测试。

Dharma[31] 是 Mozilla 社区维护的依赖于语法规约的测试程序构造工具。该工具采用特定的语法规约格式，并依此规约随机选择语法以产生测试程序。由于 Dharma 可以随机选择不依赖于复杂的上下文语法规约，因此该工具能够创建大量含有编程错误（如导致编译器词法分析阶段出错）的测试程序用于编译器前端的测试。然而，由于 Dharma 的语法规约并不具有广泛的适用性，如不能产生空规约和空语句，因此在采用该工具时需要调整部分语法规约以达到特定测试程序构造的目的。

Tang 等人[32] 提出了基于多样性引导的编译器前端警告机制测试方法 Epiphron。Epiphron 通过设计 63 种变异算子，对原始程序的抽象语法树（AST）进行变异（如删除和插入）操作，从而生成大量警告敏感结构的测试程序。在程序变异的过程中，Epiphron 采用了马尔可夫链蒙特卡罗方法（Markov Chain Monte Carlo，简称 MCMC）来引导变异算子的选择，以生成多样化的程序变体。为了构建测试预言，不同的程序变体将被输入到不同的编译器中，通过差分测试来检测编译器的前端警告缺陷。最后，采用 C-Reduce 工具对触发警告缺陷的程序变体进行约简，最后将该缺陷提交到编译器缺陷仓库中。该工具在实践中检测出了 8 个编译器缺陷。

尽管有少数工作关注于编译器前端测试，但是现有方法很难检测到深层次的编译器前端缺陷，即顺利通过词法分析阶段但可能在语法分析阶段出错的缺陷。具体来说，现有程序生成器生成语法完全正确的程序，该类程序快速通过了编译器前端，较难检测出编译器前端缺陷。另一类基于模糊测试的方法可以根据语法规则生成词法多样化的测试程序。然而，这些程序只能停留在编译器的词法分析阶段，无法进入语法分析阶段检测出深层次的前端缺陷。基于以上现有方法的局限性，亟待构造语法多样化的测试程序检测深层次的编译器前端缺陷。

（2）面向编译器中后端测试的测试程序构造方法

不同于编译器前端，测试编译器中后端需要构造语义多样化（语法完全正确但语义不一定正确）的测试程序。现有用于测试编译器中后端的测试程序构造方法大致可以分为四类，分别是基于语法规约的测试程序构造、基于深度学习的测试程序构造、基于变异的测试程序构造以及其他构造方法。

① 基于语法规约的测试程序构造

在编译器测试的初期阶段，测试工程师通常根据编程语言应用程序接口描述（如API 规范手册）手动构建与编程语言规范相一致的测试程序。该类测试程序有助于检测早期版本编译器中存在的缺陷。例如，Plum Hall[1] 提供了一套商业级的 C/C++ 测试集，其中的 C 语言测试集由于覆盖了该标准的所有功能，所以被视为 ANSI/ISO C99 标准的权威测试工具。测试套件中的 CppTest 测试集能够对 C/C++ 编译器源代码进行单元测试，分别检查编译器在语法解析、代码生成与代码优化等方面存在的问题。然而，以上手动构建的程序具有一定的局限性，无法全面覆盖编译器的各个组成部分。因此，通过给定编程语言的上下文无关文法遍历其规范的自动化方法被相继提出。

Yang 等人[2] 开发了基于 C99 语法规范测试程序的自动化生成工具—Csmith。该工具已广泛用于编译器、代码覆盖率工具以及静态分析工具的验证过程。为了生成符合规范的 C 测试程序，Csmith 在设计上满足了两个核心目标。其一，生成的代码具有明确的结构和单一的语义。因为在 C99 规范中存在 191 种未明确定义的行为（如空指针引用、有符号整数溢出等）以及 52 种未明确的行为（如函数参数求值顺序等），因此此类行为的存在会破坏代码的语义。为避免以上问题，Csmith 采用静态分析技术在运行时插入检查语句。其二，在满足第一个目标的基础上，Csmith 构造出的测试程序还具有多样化的表现力，即支持多种编程语言特性。基于以上特性，Csmith 生成的测试程序涵盖函数、全局与局部变量、控制流语句、数据结构、数组以及大多数 C 语言表达式。实验表明，Csmith 已经在主流编译器上检测到了 325 个编译器缺陷，超过一半已经被开发者修复，其中大部分是编译器中后端的严重缺陷。

Lidbury 等人[33] 将 Cmsith 的设计思路扩展到测试 OpenCL 编译器上，提出测试工具 CLsmith。CLsmith 提供了六种模式，分别生成不同类型的 OpenCL 测试程序，包括基础模式、矢量模式、障碍模式、原子性段模式、原子性规约模式及全模式。在应用于多种支持 OpenCL 的设备和编译器（如 CPU、GPU、FPGA、加速器和仿真器）的测试中，CLsmith 检测到了 50 多个 OpenCL 编译器缺陷。

Eide 等人[34] 根据 C 程序中采用 volatile 声明变量转换的准确性，研发了一种自动代码生成工具 Randprog。该工具生成的整数变量会被声明为 const、volatile 或 const volatile，并在有符号和无符号整数变量间进行类型转换。此外，Randprog 还会随机生成包括赋值语句、条件语句、循环语句及函数调用在内的函数体。由于该工具生成的所有语句均严格符合语法规范，因此该测试程序可以用来检测编译器中后端缺陷。

Purdom 等人[35] 提出了基于语法指导的代码生成方法，其目的是验证编译器在语法分析阶段的准确性。该方法生成的测试程序可用于检查和调试语法分析器的不可规约性。实验结果表明，虽然基于 Purdom 算法生成的测试程序能够涵盖语法分析器的

[1] http://www.plumhall.com/suites.html

多种状态，但对语法转换的测试效果尚不明确。由于缺乏测试效果的评估，该策略后来被应用于测试其他目标（如 IBM 的语言处理器[36]）。

Lindig 等人[37] 设计了自动化程序构建工具 Quest。该工具对 C 编译器中关于函数参数传递的均匀性进行验证。传递均匀性表示传输至函数的数据与原始数据应该保持一致，否则表明编译器在实施函数调用时存在缺陷。具体而言，Quest 制定了满足 ANSI-C 规范的函数声明和相应的函数实现，并在函数体内部插入断言，以监测传递过程中参数的均匀性。若参数传递不一致，则断言在执行时被触发。在 GCC 和 ICC 等编译器的实验中，Quest 已检测出编译器因错误处理函数调用而产生的 13 个缺陷。

Sirer 等人[38] 提出了基于 Lava 语法定义的测试程序构建策略。给定语法生成式和核心程序模板，该策略通过随机迭代语法定义以产出用于校验 Java 虚拟机完整性的测试程序。为了克服编译器测试中的预言问题，论文作者扩充了 Lava 语法定义，确保生成的测试程序有确定的预期行为。因此，在校验 Java 虚拟机时，无需对照其他 Java 编译器的输出。在对 Sun JDK 1.0.2 和 Microsoft Java 虚拟机的测试中，该策略所产出的测试程序提高了对虚拟机的代码覆盖率。

Vsevolod 等人[3] 设计了用于编译器测试的随机测试程序生成器 YARPGen。该工具专门针对 C 和 C++ 编译器测试，并侧重于揭示在 GCC、LLVM 以及 Intel® C++ 编译器中潜在的缺陷。通过 YARPGen，该工具已经向开发者报告了超过 220 个编译器的缺陷。YARPGen 的主要贡献在于提出了一种能够生成具有表达力的程序，同时在不利用动态检查的情况下规避未定义行为的创新方法。此外，YARPGen 引入了增强生成代码多样性和激发更多编译器优化机制的生成策略，该策略在检测难以触发的编译器缺陷时缩短了测试时间，并显著提升了编译器对标量优化的应用频次，对 LLVM 平均提升 20%，对 GCC 平均提升 40%。除此之外，YARPGen 还支持了一系列自动化工具，以便于实施大多数与编译器模糊测试相关的常规任务。为了进一步检测因循环优化导致的缺陷，Vsevolod 等人[39] 实现了 YARPGen 的一个变种。该变种的第一个主要贡献是在生成循环时采用静态分析的方法避免未定义行为。第二个主要贡献是一系列增加生成循环代码多样性的机制。实验结果表明，该工具已经在多种编译器中检测到了 122 个缺陷，包括数据并行语言的编译器，如 Intel® Implicit SPMD Program Compiler、Intel® oneAPI DPC++，GCC 以及 LLVM 等。

Karine 等人[40] 提出了 CsmithEdge，用于提升 Csmith 的缺陷检测能力。CsmithEdge 主要关注测试程序中未定义行为（Undefined Behavior，简称 UB）的管理和检测。现有的程序生成器（如 Csmith[2]）通过严格限制生成程序的方法，达到避免 UB 的目的。但由于生成的程序过于规范和局限，其能够提供的测试覆盖率以及能够检测到的编译器缺陷也相对有限。为了缓解以上问题，CsmithEdge 选择性地减弱用于确保 UB 自由的约束条件，意味着生成的程序不再保证完全无 UB。随后，它运用多种现有的 UB

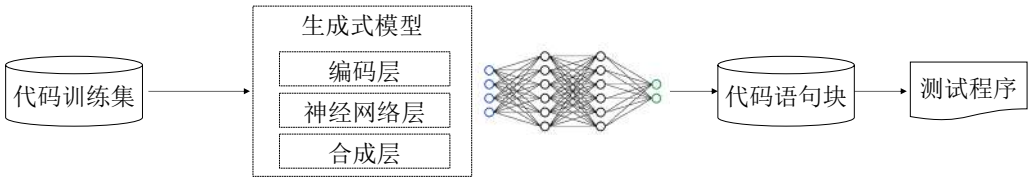


图 2.1　DeepSmith 框架图

Fig. 2.1　High-level overview of DeepSmith

检测工具和一种新颖的动态分析，检测生成程序中出现的 UB 以及确定并移除 Csmith 在采用保证算术运算 UB 自由的安全数学包装器。通过以上方法，CsmithEdge 得到的无 UB 程序可通过差分测试来测试误编译缺陷。而非无 UB 程序则用于检查测试中的编译器是否会崩溃或超时。实验评估显示，CsmithEdge 在 GCC、LLVM 和 Microsoft Visual Studio Compiler 的最新版本上检测到了 7 个编译错误缺陷（其中 5 个已经被开发者修复）和 2 个编译器超时的缺陷。

② 基于深度学习的测试程序构造

Cummins 等人[41] 采用了基于 LSTM（即 Long Short-Term Memory）神经网络的方法，开发了测试程序自动生成工具 DeepSmith，其框架如图2.1所示。该工具对 Github 仓库中大量的 OpenCL 核心进行训练，利用 LSTM 的长时间记忆功能，D 能够学习并理解代码的语法结构和 token 的上下文关系，进而创造出符合规范的测试程序。实验结果表明，此工具 OpenCL 编译器中检测出了 67 个缺陷。

Liu 等人[42] 为 C 编译器测试开发了基于 Seq2Seq（即 Sequence-to-Sequence）神经网络的测试程序生成工具 DeepFuzz。该工具主要从 GCC 和 LLVM 的测试套件中提取训练数据。为了确保生成的测试程序的多样性，DeepFuzz 设计了三种字符推理策略。实验结果显示，该工具检测出了 8 个编译器中的缺陷。然而，以上深度学习方法在处理程序时，通常把测试程序当作自然语言，而忽视了测试程序内部结构的重要性。为解决以上问题，Lee 等人[43] 提出了基于 AST 变异的方法 Montage，以专门测试 JavaScript 解释器。实验结果表明该方法在 JavaScript 引擎中检测出了 37 个缺陷。

Xu 等人[44] 提出了基于树的程序生成策略 TSmith。该策略首先将程序转化为 AST 形式，并采用基于树的 LSTM 网络（即 Long Short-Term Memory Network）进行编码。然后，通过树形解码器，TSmith 从上到下地展开非终结符以生成完整的 AST。实验数据显示，TSmith 提高了编译器的代码覆盖率，并检测到了 14 个 GCC 缺陷。

Chen 等人[45] 等人提出专注于语言处理器（例如编译器和解释器）的自动化测试框架 PolyGlot。具体而言，PloyGlot 是通用模糊测试框架，能够为不同编程语言的处理器生成高质量的测试程序。为实现其通用适用性，PloyGlot 通过统一的中间表示（Intermediate Representation, 简称 IR）来缓解不同编程语言在语法和语义上的差异。为了提高语言的有效性，PloyGlot 执行受限的突变和语义验证，以保持语法的正确性并

修复语义错误。PloyGlot 已经应用于 9 种编程语言的 21 个流行语言解释器上，实验中检测到了 173 个缺陷，其中 113 个已经被开发者修复。更重要的是，报告的缺陷中包含 18 个 CVE。另外，实验结果还表明，PloyGlot 能够支持广泛的编程语言，并且在代码覆盖率上比现有的模糊测试器有高达 30 倍的提升效果。

③ 基于变异的测试程序构造

程序变异的核心观点在于对现有测试程序进行部分修改，而不是从零开始创建一个完整的程序。程序变异用于生成测试程序的方法主要分为两大类：不保留语义的变异和保留语义的变异。

1）不保留语义的程序变异

此类变异大多基于一组预定的变异算子，如算术运算间的转换、数值和布尔值间的交换等，以更改程序的数据路径和控制结构，从而产生不同的测试程序变体。

Chen 等人[46] 提出了覆盖率引导的 JVM 测试方法 classfuzz。classfuzz 采用了 129 种不同的变异算子来改变程序的类别、接口、函数、参数和变量等。在变异过程中，classfuzz 对每一种变异算子所能生成的代表性测试程序的效率进行评估，并运用马氏链蒙特卡洛技术（Markov Chain Monto Carlo，简称 MCMC）来指导变异算子的选择。变异后的程序经过差分测试后会输入到多种 JVM 中以检测其潜在的编译器缺陷。由于 classfuzz 的产出多为语法错误，其主要被用于测试 JVM 的初始化阶段。研究结果显示，classfuzz 在 J9 编译器中检测到了 64 个缺陷。在后续工作中，为了测试 JVM 的运行时缺陷，Chen 等人[47] 在程序中添加了五种跳转指令以生成程序版本，并推出了 classming 工具。简言之，classming 初步确定要添加的跳转指令及其位置，进行变异操作后再生成程序版本。接着，该工具采用 MCMC 方法并基于是否与初始程序有不同的覆盖率来决定是否采纳该版本。实验表明，此方法在 J9 编译器中检测出了 30 个缺陷，其中 14 个已经被开发者修复。

Garoche 等人[48] 提出了以 Lustre 测试集程序为基准的四种新程序变异技术。第一种是对算术、关系和布尔操作进行更换；第二种是在 Lustre 语言中添加 pre 操作；第三种是对布尔表达式或操作符进行否定；最后一种则是对常量进行更换。该类变异后的程序结构被纳入 Lustre 的测试集中。经实验验证，大约有 25% 的变异版本能够触发 Lustre 到 C 语言的编译器缺陷。

Holler 等人[49] 提出了基于变异的策略来检测 JavaScript 解释器的方法 LangFuzz。LangFuzz 包含两个主要阶段：学习和变异。在学习阶段，该方法通过处理一组样本输入文件，学习大量代码段，并对输入的代码段进行语法上的扩展。在变异阶段，程序中的片段被随机选择一些非终端符进行替换。在某些情况下，LangFuzz 还会利用语法来生成新的非终端符替换原程序中的代码。如果被替换的代码段与原程序不匹配，该方法则会采用启发式的方法处理该问题，从而达到生成有效测试程序的目的。实验表

明，该方法在 JavaScript 解释器中检测出了 105 个重要缺陷。

Lin 等人[50] 为了深入探查 JavaScript 编译器的潜在缺陷，提出了基于 AST 子树替换的方法 Deity。对于 AST 中需进行变异的子树，Deity 会记录其类型信息和其内部变量的详情。在变异过程中，Deity 会根据已存储的数据对替换的子树进行筛选并对其变量进行重命名。最终，Deity 将变异后的 AST 转化为测试程序，并输入到不同的 JavaScript 编译器中以进行缺陷检测。在六个不同的 JavaScript 引擎上，实验数据显示 Deity 能显著提高覆盖率，并检测到了 35 个编译器缺陷。

Zhao 等人[51] 提出了基于历史驱动的测试程序合成技术 JavaTailor。该方法将从 JVM 历史上能揭示缺陷的测试程序中提取的成分编织到种子程序中，以覆盖更多的 JVM 行为/路径，从而合成多样的测试程序。具体而言，JavaTailor 首先从历史上能揭示缺陷的测试程序中提取五种类型的代码成分。然后，为了合成多样的测试程序，它迭代地将提取的成分插入到种子程序中，并通过在它们之间引入额外的数据依赖关系来加强它们的交互。最后，JavaTailor 利用合成的测试程序对 JVM 进行差分测试。在流行的 JVM 实现（例如 HotSpot 和 OpenJ9）上的实验结果表明，JavaTailor 在测试程序的多样化和有效性方面比现有方法要好。这是因为由 JavaTailor 生成的测试程序可以实现更高的 JVM 代码覆盖率，所以该工具能够检测到比最先进的技术更多的独特的不一致性。此外，JavaTailor 已经检测到 10 个（6 个被确认）编译器缺陷。

Chen 等人[52] 提出了记忆化的测试程序生成方法 MCS，旨在进一步提高编译器测试的性能。MCS 通过多代理强化学习策略对可能触发缺陷的程序特征进行记忆化搜索，并采用基于对编译器测试过程中探索的测试配置指导构建有效的测试配置。在此过程中，多代理模型学习配置选项之间的精细的协调最终辅助该方法构造出了尽可能触发缺陷的测试程序。具体而言，MCS 考虑到测试配置之间的多样性，以高效地探索输入空间，并学习在每个探索配置下的测试结果，以便探究何种输入空间更容易触发缺陷。通过在 GCC 和 LLVM 上进行的大量实验表明，MCS 在缺陷检测方面显著优于最先进的测试程序生成方法。此外，MCS 在 GCC 和 LLVM 编译器上检测到了 16 个缺陷，它们全部已被开发者确认或修复。MCS 已被全球 IT 公司（即华为）部署，用于测试其内部编译器，并检测到 10 个缺陷，所有报告的缺陷均已得到确认。

Liu 等人[53] 提出了模糊测试工具 NNSmith。该方法旨在检测深度学习（Deep-learning, DL）编译器中的缺陷。该工具的核心方法包括采用轻量级运算符规格生成多样化且有效的 DNN 测试模型以便测试编译器的大部分转换逻辑，执行基于梯度的搜索以找到在模型执行过程中避免任何浮点异常值的模型输入。为了减少漏掉的错误或假警报的可能性，采用差分测试来识别编译器缺陷。实验结果表明，NNSmith 已经在 TVM、TensorRT、ONNXRuntime 和 PyTorch 等编译器中检测到了 72 个缺陷，其中 58 个已经得到开发者确认，51 个已经由开发者修复。

Wu 等人[54] 提出了基于覆盖率引导的模糊测试框架 JITfuzz，用于自动检测 JIT 编译器中的缺陷。JITfuzz 采用了一组优化激活变异体（如函数内联和简化），用以触发典型 JIT 优化缺陷。同时，由于 JIT 优化与程序控制流紧密耦合，JITfuzz 也采用了变异体来丰富目标程序的控制流。此外，JITfuzz 还提出了变异体调度器，它根据覆盖率更新迭代地调度变异体，以最大化 JIT 的代码覆盖率。实验结果表明，就平均边缘覆盖率而言，JITfuzz 分别比基于变异的和基于生成的 JVM 模糊测试方法提高了 27.9% 和 18.6%。此外，JITfuzz 还在成熟的 JIT 编译器中检测到了 36 个真实缺陷，其中 26 个已经被开发者确认。

Karine 等人[55] 提出了针对 C 编译器和代码分析器的基于覆盖率的变异方法 GrayC。GrayC 设计了新型 C 编译器和解释器的灰盒模糊器，开发了一套针对常见 C 结构的新变异方法，并将模糊程序进行转换以产生有意义的输出，从而允许采用差分测试作为测试准则。该方法为模糊器生成的程序整合到编译器和代码解释器回归测试套件中提供了新思路。实验表明，与其他基于变异的方法相比，GrayC 覆盖了更多编译器和解释器的中间和后端阶段的代码，提高了代码覆盖率。在实践中，GrayC 检测到了 30 个缺陷，其中的 22 个已经被开发者修复。

2）保留语义的变异策略

本方法旨在在维持程序行为的前提下，对原始（或称为种子）程序执行变异，具体的变异操作包括删除、插入或修改原始程序的代码片段。大部分语义一致的变异均根据等价模输入（Equivalence Modulo Input，简称 EMI）进行。EMI 的定义指出，如果两个用相同程序语言编写的程序在一组特定输入下具有相同的语义，则表明两个程序被视为等价的。当 EMI 变体程序经过不同的编译器编译后表现出不同的行为，通常意味着该编译器存在问题。基于 EMI 定义的原则，Le 等人[4] 开发了 Orion 工具，它通过删除原始程序中未执行的语句来产生与源程序等价的变体。在 GCC 和 LLVM 编译器运行的实验结果表明，Orion 检测到了 147 个重要的编译器中后端缺陷，其中超过 100 个已经被开发者修复。

尽管 Orion 在检测编译器缺陷方面表现出色，但它也存在局限性。首先，若原始程序中的死代码量较少，则 Orion 在删除死代码以产生新的程序版本上会受到限制。其次，Orion 产生的程序版本在控制流和数据流方面的多样性相对有限。为了更高效地产生 EMI 版本，Le 等人[6] 提出了 Athena 方法，其框架如图2.2所示。该方法引入了在死代码区内插入代码片段。在插入代码的过程中，采用了 MCMC 方法进行指导。实验结果表明，Athena 在 19 个月内检测到了 72 个编译器缺陷。

值得注意的是，Orion 和 Athena 的变异策略主要集中在程序的死代码区，该策略在应用范围上存在局限性。例如，若死代码区域内的函数无法被调用，则会忽略任何导致编译缺陷的代码，从而无法触发潜在的编译器问题。为了克服以上局限性，Sun

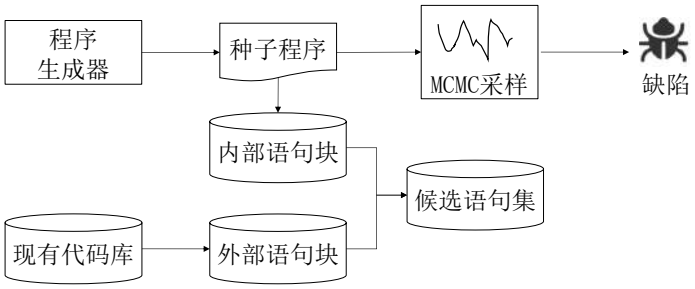


图 2.2　Athena 框架图

Fig. 2.2　High-level overview of Athena

等人[5] 提出在原始程序的活代码区进行变异的方法，并实现了 Hermes 工具。此工具可随机选择三种类型的代码段进行插入，包括 Always False Conditional Block（FCB）、Always True Guard（TG）和 Always True Conditional Block（TCB）。在以上过程中，工具会监控是否更改了代码片段的变量值，并确保插入后的程序与原始程序在语义上保持一致。当将原始和变异后的程序提交给同一个编译器并对其输出进行比较时，输出不一致通常意味着编译器存在问题。经过测试，与 Athena 相比，Hermes 在 GCC 和 LLVM 编译器上的代码行覆盖率提高了 34% 和 21.1%。此外，在 13 个月的时间里，Hermes 检测到了 168 个编译器缺陷，其中 132 个已经得到了修复。

④ 其他方法

Mandl 等人[56] 介绍了旨在防止程序内部生成重复元素的方法。该方法采用了正交拉丁方阵的方法构建针对 Ada 编译器的测试应用程序。在该方阵中，每一行和每一列只有一个元素，并且所有的元素均为互不相同。Mandl 等人将测试模型描述为正交拉丁方阵，并支持用特定值替换模型方阵中的行元素，从而产生测试应用程序。利用该正交拉丁方阵能够设计出有不同配置特点的测试模型。每种配置均可导出多个独特的测试程序，从而增加测试程序的多样性。

Austin 等人[57] 通过配置模板文档控制测试程序的构造，开发了用于检验 Ada 编译器的应用程序生成器。该作者通过随机选择语法约定，并采用递归下降法，生成了复杂的 Ada 表达式。此外，该方法采用新型构造方法定制应用程序生成器。例如，测试人员可以自定义整数范围，以产生所需的测试程序。

Ching 等人[58] 进一步探索了基于模板产生测试程序的方法。该方法主要用于 APL-to-C 编译器的单元测试。由于该方法中的模板程序从实际应用中提取，因此其产生的程序涵盖了多种 APL 内建函数和数据类型。在程序构造过程中，模板中的函数和数据类型被替换成具体的函数实现和待测试的数据类型。

Kalinov 等人[59,60] 通过模板制定表达式声明为主体的测试程序，以检测 MPC 编译器（一种并行语言编译器）中的潜在缺陷。该模板集成了一组 MPC 操作符，而表达式则由测试人员提供的合法种子表达式构成。从该种子表达式出发，通过采用种子

表达式的操作数和模板的操作符，该方法能产生多个表达式版本。

Zhang 等人[61] 提出了骨架程序枚举（Skeletal Program Enumeration，简称 SPE）策略用于构造测试程序。给定一个程序框架，即一个带有待填充变量名称的占位符源代码，以上策略会全面地枚举所有可能的变量采用模式来实例化该框架程序，并淘汰等效的测试程序。实验结果表明，该方法在 GCC 和 Clang 编译器上检测到了 217 个编译器缺陷，其中 119 个已经被开发者修复。同时，SPE 方法还在 CompCert 编译器[62,63] 中检测到了 29 个崩溃类型的缺陷。

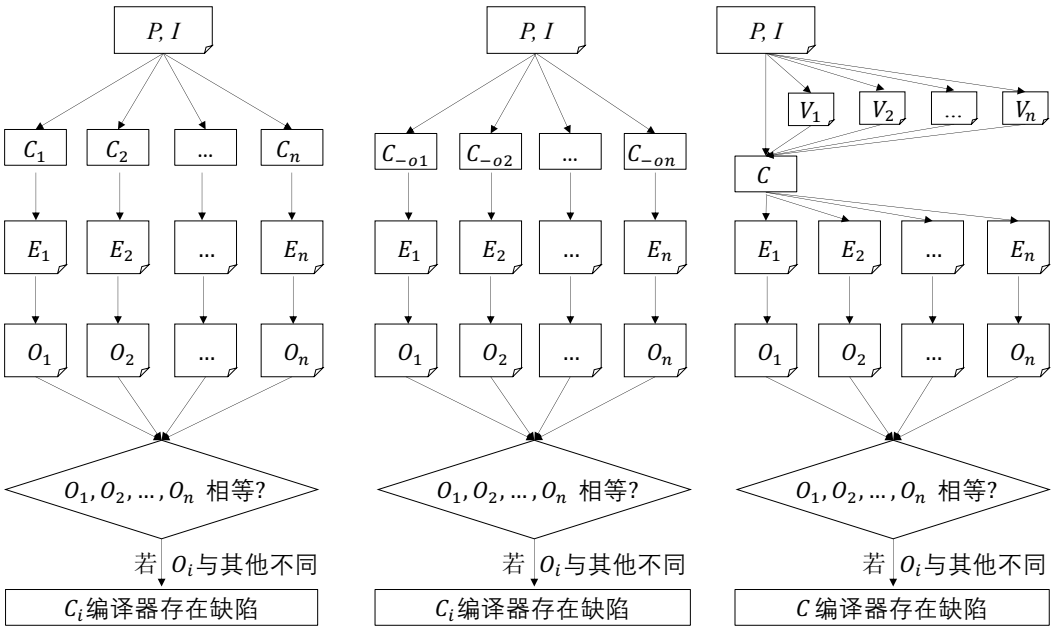
尽管存在以上大量研究专注于编译器中后端测试，但是现有方法的有效性仍然受到一定限制。首先，现有的两类程序构造方法（即基于生成及变异的构造方法）均依靠程序生成器的质量完成编译器测试。然而，现有研究表明现代编译器已经对程序生成器产生免疫，即直接使用程序生成器构造的测试程序已经很难再直接检测到新缺陷。此外，现有工作过于关注未定义行为，即要求测试程序不包含任何未定义行为。然而，该要求会降低测试程序的语义多样性。具体来说，用户可能不经意得在程序中引入了一个未定义行为，编译器应该对该类程序表现良好，即不能编译时崩溃或者超时。鉴于以上现有工作的两个主要局限性，亟待提出新的方法检测深层次编译器中后端缺陷，即由语义多样化的程序检测出来的缺陷。

2.1.2 编译器测试预言构建

编译器的独特的功能属性增加了测试预言构建的复杂度。目前的研究主要提及三种方法用于构建编译器的测试预言，分别为随机差异测试 (Randomized Differential Testing，简称 RDT)、多种优化层级分析 (Different Optimization Levels，简称 DOL) 以及等效模输入法 (Equivalence Modulo Inputs，简称 EMI)[4,10,15]。图2.3展示了以上三种测试预言构建方法的结构概览。在图中，P 代表测试程序；I 代表其输入数据；C 为编译器；E 指代生成的可执行文件；O 是当 P 接受输入 I 时产生的输出。后续小节将分别探讨以上三种用于编译器测试预言构建的相关工作。

(1) 随机性差异分析（RDT）

RDT 作为一种被广泛采用的软件验证技术，具有对众多软件结构，如编译器，进行深入评估的能力。RDT 基本原理是通过比较由相同规格但不同实现的软件结构在同样输入情况下的输出，从而识别潜在的系统缺陷。预期中，基于同等规范的不同系统应具有一致的输出。不同的输出则意味着其中的某个系统可能出现缺陷。对编译器的评估，设有一组编译器 $C_1, C_2, ..., C_n$，通常有 $n \geq 3$，它们均基于同一语言标准如 C 语言所构建。RDT 在此背景下的工作流程如图2.3(a) 所示。首先，对于给定的测试程序 P 和其输入 I，每个 C_i 会将其编译为相应的可执行文件 $E_1, E_2, ..., E_n$。然后，运行每个 E_i，从而获得输出 $O_1, O_2, ..., O_n$。若 O_i 与其他输出有所差异，则可能表明 C_i 编译器存在缺陷。



(a) RDT (b) DOL (c) EMI

图 2.3 编译器预言构建方法概览

Fig. 2.3 Summary of test oracle construction approaches

然而，尽管 RDT 的实施相对直接并具有高效性，它也有明显的局限性。首先，采用 RDT 的前提是存在多个基于相同规范的待测试系统，即需要多种可以解析同一编程语言的编译器作为测试基础条件。此外，如果所有系统的输出均为错误，则 RDT 存在进一步识别缺陷的挑战。

(2) 优化级别差异分析（DOL）

由于编译器的独特性，研究者引入了 DOL 方法。DOL 可被视为 RDT 的一个分支，其独特性在于不依赖多个编译器，而是基于单一编译器在不同优化级别上的输出进行分析。DOL 的工作流程如图2.3(b) 所示。给定测试程序 P 和输入 I，DOL 利用不同优化级别 $C_{-O1}, C_{-O2}, ..., C_{-On}$ 的编译器 C 对其进行处理，生成相应的可执行文件 $E_1, E_2, ..., E_i$。如果优化是无误的，执行所有 E_i 的输出 $O_1, O_2, ..., O_n$ 应该是一致的。若 O_i 与其他结果不一致，表明编译器 C_{-On} 存在缺陷。

尽管 DOL 具有简洁性的优点，但需要注意的是，编译器中的不同优化级别通常是层叠式且相互影响错综复杂的，因此，每个粗粒度优化级别（如 “-O3”）中的单功能优化序列（如 “-loop-vectorize”）缺陷难以被该方法检测到。

(3) 等效模输入法（EMI）

EMI，与 RDT 和 DOL 相比，是一种在近年来展现出高效性的编译器测试策略，其代表性应用包括 Orion[4]、Athena[6] 和 Hermes[5]，在众多开源编译器（如 GCC 和 LLVM）上检测出了了大量潜在缺陷。对于待检测的编译器 C，EMI 的策略是生成

测试程序的语义等效变种，并观察其输出是否一致，从而判断出错的编译器。具体流程如图2.3(c) 所示。对于特定的测试程序 P 和输入 I，EMI 采用变种技术生成等效程序 $V_1, V_2, ..., V_n$，随后用 C 对 P 和 $V_1, V_2, ..., V_n$ 进行编译，得到对应的可执行程序 $E_p, E_1, E_2..., E_n$。最后，执行 $E_p, E_1, E_2..., E_n$ 得到输出结果 $O_p, O_1, O_2, ..., O_n$。由于程序 P 和变体 $V_1, V_2, ..., V_n$ 是等价的，所以所有的输出 $O_p, O_1, O_2, ..., O_n$ 应该是一致的。如果存在不一致，则表明 C 存在缺陷。

相对于 RDT 和 DOL，EMI 的实施更为复杂，需要为测试程序构建等效变种，而该变种构建方法的耗时过程可能会影响测试的效率。由于 EMI 不需要其他对比的编译器，对于新开发的编译器，因此 EMI 可以克服 RDT 和 DOL 的局限性。

2.1.3 编译器测试程序约简

Zeller 等人[64] 提出了 dd （delta debugging）和 ddmin 两种通用的 delta 调试算法。dd 算法旨在减少触发编译器缺陷的测试程序与指定模板之间的差异，而当模板为空时，ddmin 是 dd 的特例，其目的是尽量减小触发编译器缺陷测试程序的文件大小。ddmin 算法以启发性方式去除测试程序中的连续块，并产生多种程序变体。若变体未能激发原始程序的相同缺陷，则将其丢弃。反之，算法则进一步以贪心策略对其进行精简。由于 ddmin 具备迅速精简能力，它可以快速将大程序简化为小型测试程序。

Misherghi 等人[65] 提出了分层增量调试（Hierarchical Delta Debugging, 简称 HDD）算法。该方法首先找出触发缺陷的测试程序的最小层次配置，然后细化该配置的输入输出流程，最后通过修剪程序的抽象语法树移除不必要的部分。与 ddmin 相比，实验表明在约简相同的程序时 HDD 所需的约简时间更少。

McPeak 等人[66] 提出了 ddmin 的变体方法 Berkeley delta。该方法主要针对程序的行语句进行约简，其中所有的程序变体均由删除一行或多行生成。此外，该方法还可以通过特殊的函数 topformflat 实现分层增量调试。该方法的简易特性使其在编译器开发过程中得到了广泛应用和推广。

Regehr 等人[17] 基于增量调试技术，提出了三种独特的测试程序约简工具，即 Seq-Reduce、Fast-Reduce 以及 C-Reduce。具体而言，Seq-Reduce 与 Fast-Reduce 仅适用于由 Csmith 生成的程序，而 C-Reduce 被设计为广泛适用的 C/C++ 程序约简工具。Seq-Reduce 的设计理念是将驱动程序与 Csmith 模块进行分离，使得 Csmith 主要负责生成各种程序变种，驱动程序则评估生成的变种是否触发了原始程序的相同缺陷。Fast-Reduce 则采用静态和动态方法来快速精简测试程序。C-Reduce 的设计是基于一系列的迭代转换方法，其中涉及到了三个关键函数，分别负责新的转换状态的创建、状态转换以及跟踪状态进展。实验结果表明，C-Reduce 在多个测试程序约简任务中表现出强大的约简能力，其加速效果可达其他工具的 25 倍。

Pflanzer 等人[67] 对 C-Reduce 进行了扩展开发了 CL-Reduce 工具，以适应 OpenCL

测试程序的约简。CL-Reduce 针对 C-Reduce 两大局限性进行了优化。首先，由于 C-Reduce 中的某些工具并不完全适用于 OpenCL，因此 CL-Reduce 替代采用了其他开源工具。其次，OpenCL 语言中存在某些在 C 语言中是没有的特定未定义行为。为了应对以上挑战，CL-Reduce 采用了硬编码检查和利用 Clang 静态分析方法识别 OpenCL 程序中的未定义行为。实验结果显示，CL-Reduce 在约简 OpenCL 测试程序方面具有出色的表现，能够有效对大型测试程序进行约简。

2.2 编译器调试

缺陷定位及修复是编译器调试的主要步骤。由于编译器缺陷定位是本文关注的重点内容且尚未有编译器缺陷自动修复的工作，本节主要对基于程序构造及基于其他技术的编译器缺陷定位方法进行调研。

2.2.1 基于程序构造的编译器缺陷定位

Chen 等人[12] 最早提出了编译器缺陷定位方法 DiWi。DiWi 的核心思想是将编译器缺陷定位问题转化为证人测试程序构造问题。在本文中，证人测试程序指的是不能触发编译器缺陷的测试程序变体。DiWi 通过比较编译器编译原始失败测试程序和证人测试程序时的覆盖信息差异来定位编译器缺陷所在的文件。为构造证人测试程序，DiWi 包含 6 类共计 132 种不同的变异算子，该类算子对原始测试程序进行微小的局部变异，例如用加法替换乘法，以使得编译器编译证人测试程序时的覆盖信息尽可能接近编译原始测试程序的覆盖信息。然而，对于触发编译器缺陷的测试程序，该 132 种变异算子的效果各不相同，因此，DiWi 采用基于 MCMC 的方法来选择变异算子，以生成更多样化的证人测试程序。为了验证 DiWi 的有效性，研究者构建了一个包含 90 个真实编译器缺陷（GCC 和 LLVM 编译器各 45 个）的数据集。结果显示，DiWi 能够将 66.67% 和 78.89% 的缺陷分别定位到 Top-10 和 Top-20 的文件中。

然而，尽管 DiWi 被证实有效，但其变异算子仅关注局部变异，限制了其在编译器优化缺陷定位方面的效果。因此，Chen 等人[13] 提出了基于强化学习编译器缺陷定位方法 RecBi。RecBi 与 DiWi 的基本原理相似，同样将编译器定位问题转化为证人测试程序构造问题。不同于 DiWi 的是，RecBi 增强了变异算子的多样性，除了 DiWi 的 132 种局部变异算子外，RecBi 还引入了 4 种结构性变异算子，例如在测试程序中添加一个 if 语句。此外，RecBi 采用强化学习的方式智能选择变异算子，以提高生成证人测试程序的效率和多样性。在实验中，RecBi 扩展了 DiWi 的数据集至 120 个真实编译器缺陷。实验结果表明，RecBi 能够将数据集中 23% 的缺陷定位在 Top-1 的文件中，在定位至 Top-1 的缺陷数量上比 DiWi 提升了 92.86%。

2.2.2 基于其他技术的编译器缺陷定位

Zeller 等人[68] 提出了基于因果链定位的调试方法 Delta，用于分析导致编译器缺陷的根本原因。该方法的核心思想是通过比较编译器在正常执行和缺陷出现时的状态差异，以定位导致缺陷的变量和相应的值。该方法首先对引发编译器缺陷的测试程序进行 Delta 调试，逐步缩减测试程序，同时获取约简过程中编译器执行的内存图。最后，通过对比编译器在正常执行和缺陷时的内存图，来逐渐缩小编译器缺陷的原因范围。然而，该方法存在缺陷，即获取和比较编译器的内存图是一项较耗时的操作，而且 Delta 调试的效率也对该方法的整体效率产生严重影响。

Holmes 等人[69] 提出了辅助编译器缺陷定位的变异方法。该方法的核心思想是，对于给定的编译器缺陷和相应的测试程序，通过对编译器本身进行变异操作，如果变异后的编译器不再出现缺陷，则变异操作的位置被认定为编译器缺陷发生的位置。为了实现该方法，研究者设计了一系列编译器变异算子，例如条件表达式取反等。如果经过多个不同的变异操作后的新测试程序不能复现编译器缺陷，则相关的变异操作将形成变异算子向量。当存在多个变异算子向量时，该方法通过对比算子向量来准确定位编译器缺陷的具体位置。然而，由于编译器的复杂性和庞大的代码量，该方法在实践中存在问题，如方法实施时面临着极大的时间开销。

Wang 等人[70] 提出了 ProbDD 算法。该算法的出发点在于，虽然许多现有方法所基于的基本算法 ddmin，且遵循预定义从序列中删除元素的序列，但该方法并未能利用现有测试结果中的信息。为了解决以上问题，ProbDD 构建了概率模型来估算保留在产生结果中的元素的概率，基于该模型选择一组元素以最大化下一个测试的收益，并基于测试结果改进模型。实验证明了 ProbDD 的正确性，并在最坏情况下分析了其结果的最小性和渐近测试次数。ProbDD 在最坏情况下的渐近测试次数为 O(n)，比 ddmin 的 O(n2) 的渐近测试次数要小。通过在 HDD 和 CHISEL 的 40 个主题上实验比较 ProbDD 和 ddmin 的结果显示，在用 ProbDD 替换 ddmin 之后，HDD 和 CHISEL 分别在用时上减少了 63.22% 和 45.27%。

Zhou 等人[71] 提出了基于搜索的编译器优化缺陷定位方法 LocSeq，用于专门定位 LLVM 编译器中的优化缺陷。该方法通过对比缺陷编译器优化序列与无缺陷编译器优化序列生成的执行路径，来排除与缺陷无关的文件，以确定缺陷发生的文件，从而将编译器优化缺陷定位问题转化为无缺陷编译器优化序列构造问题。为了解决以上一问题，该方法引入了一种约束遗传算法，以便根据给定的缺陷编译器优化序列构造一组无缺陷编译器优化序列。通过在 60 个真实编译器优化缺陷上的验证，该方法可将 65.00% 的缺陷定位在 Top-5 的文件中，提升幅度达 77.27%。

与 LocSeq 类似，Yang 等人[72] 提出了 ODFL 定位方法，用于专门定位 GCC 编译器中的优化缺陷。该方法通过细化编译器选项来定位编译器优化缺陷。具体而言，

ODFL 首先独立禁用了在触发缺陷的优化级别下默认启用的细粒度选项，以获取不包含缺陷和与缺陷相关的细粒度选项。然后，基于细粒度选项，配置了若干有效的通过和失败的优化序列，以获得多个失败和通过的编译器覆盖。最后，通过基于频谱的缺陷定位公式，可以利用生成的覆盖信息来排名可疑的编译器文件。在现有基准测试的 60 个有缺陷的 GCC 编译器上的实验结果表明，ODFL 在所有评估指标上明显优于最先进的编译器缺陷定位方法 RecBi，证明了 ODFL 的有效性。此外，ODFL 比 RecBi 更加高效，实验结果表明该方法在平均定位缺陷时能够节省超过 88% 的时间。

尽管现有方法将传统的缺陷定位问题转化为了证人测试程序构造问题，但是现有方法仍然存在一定的局限性。首先，现有方法变异的有效性仍然存在缺点，即 1）结构化的变异只支持条件语句插入而不支持语句块插入；2）变异时选择的变量和位置是随机的，导致生成的证人测试程序多样化受限。其次，现有方法构造程序的开销太大，通常需要手动编写代码完成变异的整个流程，包括提取程序中的语义信息和找出合适的位置执行插入变异操作。最后，现有工作没有关注语义无效的证人测试程序，该类程序因为无法帮助隔离出可能存在缺陷的编译器源文件，从而导致无法对真正存在缺陷的文件进行有效排序，严重影响缺陷定位的有效性。鉴于以上局限性，亟待提出新方法进一步提高编译器缺陷定位的效率。

2.3 关键科学问题

综上所述，虽然已有众多研究采用基于程序构造的方法辅助编译器测试与调试，但针对编译器前中后端缺陷检测和缺陷定位场景下构造满足特定需求的测试程序仍然存在诸多不足。本文聚焦基于程序构造的编译器测试与调试方法，致力于解决以下三个科学问题，从而有效构造出满足特定需求的测试程序实现高效的编译器缺陷检测和缺陷定位，进一步推动编译器测试与调试技术的发展。

（1）如何构造语法多样化的测试程序检测深层次的编译器前端缺陷？

编译器前端缺陷会严重地影响编译器的可用性和可靠性，成为开发者调试和修复程序缺陷的重大障碍。现有大多数程序构造方法只关注中后端测试，仅有少部分关注编译器前端测试。即使存在一些编译器前端测试工作，它们仅能检测到浅层次（如在编译开始的词法分析阶段出错）的测试程序，大大降低了产生有效的测试程序检测深层次前端缺陷的可能性。然而，构造语法多样化的测试程序十分困难，需要解决如何设计灵活化结构以实现多粒度结构化变异以及选择代表性结构以生成有效程序两个关键挑战。为了解决以上两个关键挑战，本文拟提出一种基于结构感知的程序构造方法，采用灵活化的结构定义以支持多粒度结合的变异操作，并结合基于差分策略和编译器输出信息对齐算法，有效地检测深层次（测试程序可以通过词法分析，但可能因为其他错误不能通过语义分析）的编译器前端缺陷。

（2）如何构造语义多样化的测试程序检测深层次的编译器中后端缺陷？

编译器中后端缺陷通常比较隐蔽而且很难被检测到，且会直接影响被编译软件的正确性。现有致力于编译器中后端的测试方法严重依赖于程序生成器的质量：无论是基于生成的程序构造方法还是基于变异的程序构造方法，它们均依靠程序生成器生成种子程序，然后进行后续的测试流程。然而，主流的编译器已经对现有的程序生成器产生免疫，很难再检测出深层次（测试程序语法有效但可能包含未定义行为）的中后端缺陷。此外，现有方法过于关注构造未定义行为的测试程序，而忽略了含有未定义行为的测试程序也可以检测到深层次的编译器中后端缺陷。为此，本文拟提出一种再造程序生成器的方法，提高现有程序生成器的缺陷检测能力，以构造语义多样化的测试程序。但是，有效地对现有程序生成器进行再制造并非易事，需要解决如何轻量化地合成多样化代码片段以及如何选择更有可能构造出揭示缺陷测试程序的代码片段。为了解决以上两个关键挑战，本文拟有效利用现有程序生成器的良好特性（如高效复用性）并在此基础上新增两个关键组件（即多样化代码合成组件和代表性代码片段选择组件）从而构造出能够检测到编译器中后端深层次缺陷的测试程序。

（3）如何构造语义完全有效且权衡多样化和相似化的证人测试程序辅助编译器缺陷定位？

当编译器缺陷被检测出来后，及时定位缺陷的出错位置是编译器调试的重要环节之一。现有基于程序构造的编译器缺陷定位在有效性上存在一些局限性，如构造出来的证人测试程序多样性不足等，并且构造的过程需要消耗大量的人力，严重影响了缺陷定位的效率。受最近预训练大型语言模型（LLMs）的启发，本文拟提出一种基于大语言模型赋能的程序构造方法，以轻量化的方法构造语义完全正确且权衡多样化和相似化的证人测试程序。然而，如何有效地采用大模型使其产生期待的结果并不容易，需要解决如何制定精确的提示以及如何针对每个缺陷选择特定的提示。为了解决以上两个关键挑战，本文拟提出基于大模型赋能的程序构造方法，采用基于静态程序分析技术（如数据和控制流分析）的提示制定以及基于强化学习和轻量化程序验证的特定提示选择，构造能够有效辅助编译器缺陷定位的证人测试程序。

2.4 本章小结

本章首先介绍了编译器测试的基本流程，包括编译器测试程序构造、测试预言构建以及测试程序约简三个方面的相关工作。然后，阐述了编译器调试的相关技术，包括基于程序构造的编译器缺陷定位及基于其他技术（如基于优化序列及搜索方法等）的编译器缺陷定位技术。最后，在分析了现有工作的基础上，本章总结归纳了基于程序构造方法中编译器测试与调试相关的三个科学问题。后续的三个章节将描述每个关键科学问题的解决方法，包括方法概述，方法详细描述以及实验评估等。

3 面向编译器前端测试的结构感知程序构造方法

3.1 概述

随着计算机科学技术的发展，许多编程语言（如 C/C++、Java、Python、R、PHP 与 Kotlin 等）相继被提出并致力于构建满足用户各种需求的软件系统。在众多类型的编程语言中，C++ 是一门历史悠久的编程语言，自其 1979 年被发明以来至今已有超过 40 年的历史，目前已经成为一门深受欢迎且被广泛使用的编程语言[1]。具体而言，C++ 不仅仅是一门普通的编程语言，更包含了一整套丰富的工具集[73]。权威调查（即来自 JetBrain 的调研[2]）显示，截止 2015 年，C++ 用户群体已经超过 450 万，并且每年以约 10 万的数量稳定增长。就编译器基本工作流程而言，编译程序第一步通常需要对该程序进行前端（包括词法、句法和语义）分析[74–76]。因此，编译器前端在整个编译器框架中占据着重要的地位。特别地，由于 C++ 语法的复杂性和现代编译器中手写的 C++ 编译器前端[3]的不可靠性，C++ 相关组件在两个成熟且广泛使用 C++ 编译器（即 GCC 和 Clang）中是最容易出错的组件之一[16,18]。通常，编译器前端的任务是输出程序的中间代码，以将该中间代码用于后续的中端处理[7,77]。如果输入程序存在编程错误，编译器应该以友好并准确的方式报告输入程序中的任何错误[7]。此外，质量可靠的编译器前端可以更好地保护软件系统，防止攻击者利用编译器相关缺陷进一步攻击存在安全漏洞的软件系统[11,78,79]。因此，保障 C++ 编译器的正确性和可靠性，有效检测 C++ 编译器前端缺陷至关重要。

尽管已有众多研究致力于编译器测试[2,6,9,10,28,80,81]，但很少有研究专注于测试编译器前端。通常，编译器测试方法首先使用程序生成器构造测试程序，然后构建测试预言（如差分策略）对编译器进行测试。具体而言，差分测试方法将不同编译器的输出或已编译程序的执行结果进行比较，以检测输出中的不一致性行为，从而检测出潜在的编译器缺陷。Csmith[2] 和 YARPGen[3] 是两个著名的 C++ 程序生成器，但它们很难检测出编译器前端缺陷。首先，它们主要用于构造完全符合 C++ 语义且满足所有的语法和类型检查规则[41] 的测试程序。因为此类测试程序均会很快通过编译器前端，所以此类 C++ 程序难以触发潜在的编译器前端缺陷。其次，它们构造的测试程序包含的 C++ 语言特性也是有限的，例如，缺少 “template” 或 “class” 等关键词的支持。除上述两个工具外，基于语法的方法（如 Dharma[31] 和 Grammarinator[30]）也可以在 C++ 语法的帮助下构造出包含更多特性的 C++ 测试程序。然而，Dharma 和 Grammarinator

[1] https://www.stroustrup.com/TechRepublic-interview-Bjarne-Stroustrup.pdf

[2] https://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion

[3] In GCC: https://gcc.gnu.org/wiki/New_C_Parser; In Clang: http://clang.llvm.org/features.html#unifiedparser

构造的测试程序在多样性上仍然存在不足，使得它们很难检测出深层次的编译器前端缺陷。具体而言，Dharma 由于可扩展的局限性无法涵盖所有语法规则。当测试程序变得复杂时，Grammarinator 难以操作复杂的 AST（Abstract Syntax Tree，即抽象语法树）（详见第3.3.3节）。综上所述，上述限制可能会大大降低编译器前端测试的有效性。因此，有必要设计一个新方法构造有效的测试程序以检测深层次的编译器前端缺陷。

然而，构造有效的测试程序以对 C++ 编译器前端进行详尽测试并不容易。由于 C++ 语法的复杂性以及测试手写 C++ 编译器前端的难度，需要解决两个挑战。首先，构造多样化的测试程序对任何与软件测试相关的活动均至关重要[8,82,83]。尤其在编译器测试领域，有效的测试输入尤为重要。因为对编译器而言，测试输入是高度结构化的测试程序。因此，需要解决如何获取尽可能触发 C++ 编译器前端缺陷的测试程序的挑战。该挑战可拆解为两个子挑战，即如何设计灵活化的结构以实现多粒度结合的变异操作及选择代表性的结构以构造多样化的测试程序。其次，编译器输出信息通常不准确。例如，由于 GCC 和 Clang 在诊断系统中具有不同的机制，它们在编译同一程序时的编译器输出可能会有所不同。此外，现有方法很难处理复杂的编译器输出。以上不足使得识别潜在 C++ 编译器前端缺陷的过程变得困难。因此，需要解决如何从复杂编译器输出中有效识别潜在缺陷的挑战。

为了解决以上挑战，本章提出了面向 C++ 编译器前端测试的框架 CCOFT(**C**++ **CO**mpiler **F**ront-end **T**ester，即 C++ 前端测试器），旨在有效检测深层次的 C++ 编译器前端缺陷。为了解决第一个挑战，CCOFT 设计一个实用的 C++ 程序生成器。具体而言，生成器首先将 C++ 语法转化为更为灵活的结构化格式，该格式可以轻松实现多粒度结合的变异操作，如基于位的反转变异和基于结构化的交叉变异，以获得语法多样化的测试程序结构。然后，使用一个称为 ECS（**E**qual-**C**hance **S**election，即平等机会选择）的策略进行结构感知的结构选择，最终将多样化的生成结构转化为编译器测试的输入程序，从而构造多样化的 C++ 测试程序。为了解决第二个挑战，CCOFT 采用了一系列差分测试策略（如跨编译器版本、跨编译器种类及跨语言标准等）和错误诊断信息对齐算法，通过比较不一致的编译器输出识别编译器前端中各种类型的缺陷。

为了评估 CCOFT 的有效性，本章对两个主流编译器（即 GCC 和 Clang）进行了充足的实验评估。首先，实验将 CCOFT 与两种先进的方法（即 Dharma[31] 和 Grammarinator[30]）进行比较，以评估 CCOFT 的缺陷检测能力。结果表明，CCOFT 可以检测到 40 个有效缺陷，而 Dharma 和 Grammarinator 在同一测试周期内只能分别检测到 17 个和 19 个缺陷，表明 CCOFT 和现有方法相比在缺陷检测数量上的提高了 135% 和 111%。其次，实验结果还证实了 CCOFT 中的 ECS 策略对 CCOFT 的贡献。具体而言，通过采用 ECS 策略，CCOFT 能够比没有 ECS 的变体多检测出 18 个缺陷，提高达 82%。最后，实验还在编译器的开发主干版本中评估了 CCOFT 在实践中检测缺陷

的能力。在三个月的测试时间内，CCOFT 报告了 136 个 C++ 编译器前端缺陷，其中在 GCC 上检测出 78 个（其中 57 个已被开发者确认、分配或修复），Clang 上 58 个（其中 10 个已被开发者确认或修复）缺陷。同时，开发者对 CCOFT 报告的缺陷给予了积极的肯定，突出说明了 CCOFT 方法在实践中的有效性。

3.2 背景和动机

（1）典型 C++ 前端缺陷类型示例

图3.1列举了由 CCOFT 检测到的五个代表性的 C++ 编译器前端的缺陷。为了避免歧义，在后续的描述中，本章使用 “有效” 来指代语义上有效的测试程序，使用 “无效” 来表示在语法或语义上有误的测试程序。根据编译器前端任务的具体描述[7]，本章将编译器前端缺陷分为以下五个代表类型：

① 拒绝有效程序缺陷（Reject-valid）。C++ 编译器前端可能会拒绝一个有效的测试程序，即给定一个有效的测试程序输入，编译器可能给出错误诊断信息。图3.1(a) 描述了一个 GCC 缺陷，其中 GCC 的 C++ 编译器前端拒绝编译了该有效测试程序并给出了一个错误诊断消息，而 Clang 的 C++ 编译器前端正确地接受（成功编译）了该测试程序。值得一提的是，该类缺陷具有较高的优先级，与导致程序运行时错误的缺陷（编译器中最重要的缺陷之一）一样重要[4]。

② 接受无效程序缺陷（Accept-invalid）。与拒绝有效程序缺陷相反，C++ 编译器前端可能会接受一个无效的测试程序，即给定一个无效的测试程序输入，编译器可能不会给出任何错误诊断信息。图3.1(b) 展示了一个 Clang 缺陷，其中 Clang 的 C++ 编译器前端接受了模板声明中的一个空声明（该代码违反了 C++ 语言标准），而 GCC 的 C++ 编译器前端正确地拒绝了测试程序并给出了相应的错误诊断信息。

③ 诊断消息缺陷（Dignoistic）。准确的错误诊断消息可以帮助开发者发现和修复程序中的编程错误，而 C++ 编译器前端可能会发出模棱两可或重复的错误诊断消息，甚至可能漏报或者误报错误的准确位置（如行号）。图3.1(c) 描述了一个缺陷，其中 GCC 的 C++ 编译器前端对 C++ 中的推断返回类型的测试程序输出了一个模棱两可的错误诊断消息，而 Clang 的 C++ 编译器前端则准确地报告了该测试程序出错的真正原因。若采用 GCC 编译此测试程序，用户可能很难根据带有歧义的编译输出结果对源代码进行调试和修复，从而导致软件开发进度的延迟。

④ 崩溃缺陷（Crash）。对于一个有效或无效的测试程序，C++ 编译器前端在编译过程中可能会出现崩溃。崩溃缺陷可以分为两个子类，即在有效程序上崩溃和在无效程序上崩溃。图3.1(d) 描述了一个包含不完整模板扩展的无效程序，该无效的测试程序使 GCC 的 C++ 编译器前端崩溃。值得一提的是，此类崩溃会给开发者造成严重影

[4]https://www.gnu.org/software/gcc/bugs/management.html

```
1 //s1.cc
2 typedef int T;
3 using typename :: T;
4 /* GCC-trunk output:
5 s1.cc:2:18: error: expected nested-name-specifier before 'T' */
```

(a) 拒绝有效代码缺陷 (GCC #95597)

```
1 //s2.cc
2 template <class> ;
3 /* GCC-trunk output:
4 s2.cc:1:18: error: expected unqualified-id before ';' token
5 Clang-trunk output:
6 //no error */
```

(b) 接受无效代码缺陷 (Clang #46231)

```
1 //s3.cc
2 decltype(auto) foo () {};
3 /*GCC-trunk output:
4 s3.cc:1:10: error: expected primary-expression before 'auto'
5 Clang-trunk output:
6 s3.cc:1:1: error: deduced return types are a C++14 extension */
```

(c) 诊断信息缺陷 (GCC #96103)

```
1 //s4.cc
2 struct g_class : decltype (auto) ... { };
3 /* GCC-trunk output:
4 s4.cc:1:35: internal compiler error: in cxx_incomplete_type_diagnostic,
5 at cp/typeck2.c:584 */
```

(d) 崩溃缺陷 (GCC #95672)

```
1 //s5.cc
2 void a () { .operator b }
3 /* GCC-trunk compile-time-hog */
```

(e) 超时缺陷 (GCC #96137)

图 3.1　五种典型的 C++ 编译器前端缺陷

Fig. 3.1　Five bugs in C++ compiler front-ends

响，延缓软件的调试和修复过程。在 CCOFT 报告该缺陷之前，大部分 GCC 版本受到了该缺陷的影响。

⑤ 超时缺陷（Time-out）。C++ 编译器前端可能会花费大量或者无休止的编译时间分析一个测试程序。图3.1(e) 展示了一个此类缺陷，导致 GCC 的 C++ 编译器前端陷入死循环并进行无休止的分析。相反，Clang 的 C++ 编译器前端迅速完成了分析。

以上列举的五类典型 C++ 编译器前端缺陷可能会深刻地影响编译器的可用性

和可靠性，成为开发者调试和修复程序缺陷的重大障碍[84][85]。更糟糕的是，如前文第3.1节中所述，此类缺陷还可能使安全攸关的关键软件系统受到更加严重的安全性威胁（如系统被恶意攻击者攻击）[11,78,79]。

（2）C++ 编译器历史缺陷的定量研究

为了深入理解 C++ 编译器前端缺陷的分布情况，本小节对编译器历史缺陷进行定量研究。已有研究表明，在众多编译器实现组件中，C++ 组件是 GCC 和 Clang 编译器中存在缺陷数量最多的组件[16,18]。由于现有研究缺乏对 C++ 缺陷的深入研究，本节采用定量研究进一步对 C++ 组件中的缺陷进行分类并总结。由于 GCC 具有较长的开发历史和明确显示缺陷类型的关键词机制，该研究仅收集 GCC 的缺陷报告。具体而言，该研究从 GCC 缺陷仓库中收集序号从 1 至 93,000 的缺陷报告。最终，在所有收集到的 86,222 份缺陷报告中，20,441（23.7%）份属于与 C++ 相关的组件。接下来，本文根据各个缺陷报告中的关键词对 C++ 相关组件的缺陷进行分类，图3.2显示了 GCC 中和 C++ 组件相关的所有缺陷中前 5 种类型的缺陷数量分布统计情况。

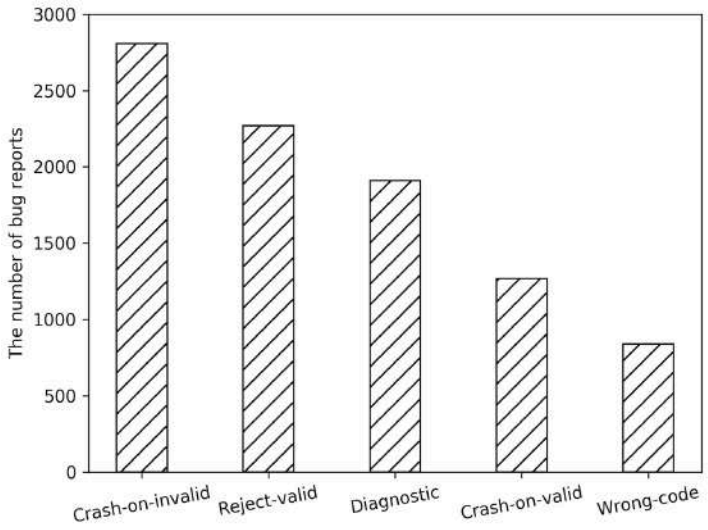


图 3.2　和 C++ 相关排名前五的缺陷类型统计

Fig. 3.2　Top-5 bug types of all the bugs of C++-related components in GCC

结合上述小节中提到的编译器前端缺陷类别，定量研究发现图3.2中的前四种缺陷类型均可能与 C++ 编译器前端有关。为进一步确认前四种缺陷是否真正属于编译器前端而不是属于其他组件，本文对前四种缺陷类型中的缺陷进行了小规模分析。具体而言，针对每种类型的缺陷，首先随机选择了 100 个已修复的缺陷。然后，手动检查所选的 400 个缺陷的修复文件位置以确认各个缺陷是否真正属于前端。结果显示，每种类型的 100 个缺陷中分别有 69、63、52 和 72 个缺陷是编译器前端的缺陷，即平均至少有 64% 的缺陷属于 GCC 的 C++ 编译器前端。鉴于 C++ 编译器前端仍然存在大量的缺陷，且现有工作较少关注 C++ 编译器测试，设计有效的方法辅助测试 C++

编译器前端并提高其质量是十分必要的。

3.3 CCOFT 框架描述

本章首先概述 CCOFT 整体框架，然后描述 C++ 程序构造器，包括构造过程中采用的语法感知模版的构建和语法感知的变异技术。最后描述基于差分测试与信息对齐算法的前端缺陷识别过程。

3.3.1 CCOFT 框架概述

图3.3展示了 CCOFT 的整体工作流程，其中包括两个主要部分，即 C++ 程序生成与 C++ 编译器前端缺陷识别。第一部分旨在构造更有可能触发 C++ 编译器前端缺陷的测试程序，该部分为构造有效测试程序的主体部分，第二部分采用差分测试策略及输出信息对齐算法识别潜在的编译器前端缺陷。

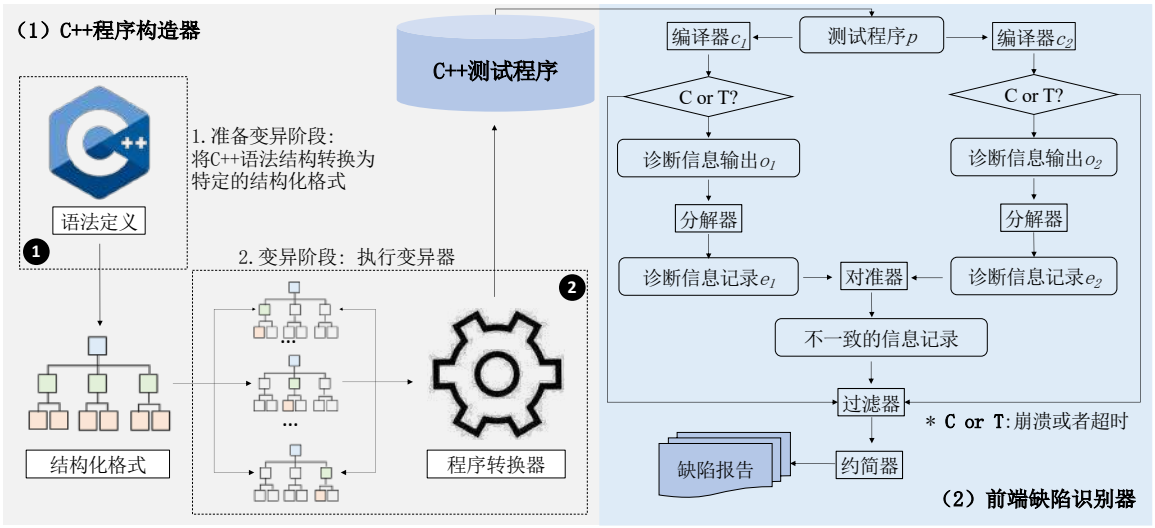


图 3.3 CCOFT 框架

Fig. 3.3 Framework of CCOFT

如第3.1节所述，编译器前端测试中最具挑战性的部分之一是如何构造语法多样化的测试程序。不同于优化阶段的编译器缺陷，深层次的编译器前端缺陷通常难以被检测到。值得一提的是，虽然词法或句法无效测试程序也能在一定程度上检测到编译器前端的一些缺陷，但是此类测试程序很难检测出较深层次的编译器前端缺陷。本章研究的目标是构造语法多样化的测试程序，从而检测出 C++ 编译器中较深层次的前端缺陷。为此，CCOFT 采用基于突变的测试程序构造方法构造语法多样化的测试程序。具体而言，CCOFT 设计了一个语法突变的策略，该突变直接突变语法而非测试程序。采用在语法结构上突变的主要好处是，与现有直接对测试程序突变的方法相比，CCOFT 采用的突变策略保证了测试程序语法的有效性。为了解决第二个挑战，当获

取了有效的 C++ 测试程序后，CCOFT 对编译器前端进行差分测试并结合编译器输出信息对齐算法来识别缺陷。特别地，CCOFT 是第一个利用差异性测试技术来检测前端的各种编译器缺陷的研究工作。后续小节将详细介绍各个部分的设计。

3.3.2 结构感知的模版构建

由于 C++ 测试程序高度结构化，采用随机的程序生成器（如 Dharma[31] 和 Grammarinator[30]）或基于突变的模糊器（如 AFL[5]或 LibFuzzer[6]）构造 C++ 测试程序对检测编译器前端缺陷的效果不佳。具体而言，Dharma 在处理未定义标识符的使用时存在一定的局限性。对于 XML 或 JSON 等语言包含的小规模语法，Dharma 调整相关语法结构以构造出所需的测试程序理论上是可行的。但是，当遇到相对复杂的编程语言（如 C++）时，Dharma 难以处理此类包含大规模语法定义的编程语言。值得注意的是，处理未定义标识符的能力是在编译器前端有效检测缺陷的关键。由于未定义标识符的缺陷通常在编译器前端分析靠前的阶段（即词法分析）产生。一旦存在此类的缺陷，后续的编译阶段（如句法或语义分析）的代码将无法覆盖。为了克服以上缺点并达到本文目的，CCOFT 的目标是构造语法多样化（即更有可能通过语法分析但可能在语义分析中出错）的测试程序。其次，虽然另一种方法 Grammarinator 为语法规则提供了可配置的操作，但它仍然存在以下两个限制。第一，因为它只支持一个简单的符号表来维护普通的变量，所以未定义的标识符问题并没有得到完全解决。然而，在复杂的 C++ 语法中有许多类型的标识符。第二，由于基于 AST 构造测试程序时操作的复杂性，Grammarinator 很难处理更深层次的递归（先前的研究[30] 表明，如果递归深度大于 50，该方法则会处理超时），从而阻碍了复杂测试程序的构造。下文将详细介绍 CCOFT 的工作流程并讨论其如何克服现有方法存在的缺点。

总体而言，CCOFT 采用一种简单而有效的方法完成 C++ 程序构造的过程，该过程包括两个关键阶段，即准备阶段和突变阶段。

准备阶段将标准的 C++ 语法定义转化为结构化格式文件，突变阶段执行结构感知的语法突变，然后经过程序转换器后构造 C++ 测试程序。为了说明 C++ 程序构造器的工作原理，下文将以简化的 C++ 模板声明语法（如图3.4所示）为例，描述上述两个阶段的详细信息。在准备阶段中，为了成功构造测试程序，CCOFT 采用 C++ 语法并将其转化为结构化格式文件。尽管 C++ 语法通常是公开可用（如发布在 ANTLR 社区[86] 中的）且结构化格式通常容易获得（如 JSON[7]、Cpn's Proto[8]和 Protobuf[9]等），但并不是每种格式均可以满足本方法的要求，即它应该具有灵活且易操作的特性，以

[5]http://lcamtuf.coredump.cx/afl

[6]https://llvm.org/docs/LibFuzzer.html

[7]https://www.json.org/json-en.html

[8]https://capnproto.org

[9]https://developers.google.com/protocol-buffers.html

```
1  Templatedeclaration
2      : Template ‘‘<’’ Templateparameter ‘‘>’’ Declaration
3  ;
4  Templateparameter
5      : Class  Identifier ?
6      | Typename  Identifier ?
7  ;
```

图 3.4　一个简化的 C++ 模版声明语法结构
Fig. 3.4　A C++ grammar of a simplified template declaration

便可以与现有的突变引擎结合。因此，在本文中，最终将 C++ 语法转化为 Protobuf 格式[10]，该格式是一种更小更快、语言中立、平台中立以及可扩展的序列化结构。CCOFT 选择 Protobuf 的原因有两个。首先，它与普通的语法定义具有对应的字段关系。其次，Protobuf 格式可以轻松地与现有的结构感知突变器（如 libprotobuf-mutator）结合。与传统基于源码的突变方式不同，结构感知的突变可以实施多粒度的变异：既支持细粒度的变异如位反转又支持粗粒度的结构化交叉等变异操作。

图3.4描述了关于模板声明的 C++ 语法部分，其中包括五个元素。根据语法与结构化格式之间的对应关系，可以获取如图3.5所示的结构化格式文件。更具体而言，在图3.4中，“Template”、“<” 和 “>” 是三个固定的元素，而 “Templateparameter” 和 “Declaration” 是两个可以被其他备选字段替代的字段。下一小节介绍如何采用定义好的结构化文件实施具体的变异操作。

3.3.3　结构感知的变异操作

如图3.3左侧所示，执行变异时 CCOFT 首先将准备好的结构化格式文件作为输入，然后将其突变为各种突变体。其中各个突变体将对应于一个测试程序。由于结构化格式文件中有三个关系字段，因此如何选择合适的可选字段（即 “oneof” 和 “optional”）对于确保结构感知突变的有效性十分重要。为此，CCOFT 采用一种等概率机会选择策略 ECS（Equal Chance Selection），即采用相等概率对 “optional” 和 “oneof” 的关系中的备选字段进行选择。最终，对于 “required” 字段，CCOFT 全部选择所包含的变量；对于 “optional” 字段，有 1/2 的概率选择该变量；对于 “oneof”’ 字段，以 1/n 的概率选择该变量（n 是其中一个 “oneof” 字段中的元素总数）。CCOFT 选择该策略选择该策略的原因是该策略采用相等的可能性对不同的语法元素进行选择，因而可以达到提高构造测试程序的语法多样性的目的。值得注意的是，CCOFT 中选择概率的灵活配置可以很容易地调整以满足其他要求，有效克服了 Dharma[31] 中的主要局限性。在执行变异时，Protobuf 内置的多粒度变异操作可以自动地执行，从而构造出多样化的中间

[10]https://developers.google.com/protocol-buffers

```
message TemplateDeclaration  {
      required  Tempalate template_name = 1;
      required  TemplateParameter template_param = 2;
      required   Declaration   declaration   = 3;
}
message TemplateParameter {
     oneof _ {
      TemplateParameter1 template_parameter1  = 1;
      TemplateParameter2 template_parameter2  = 2;
      }
}
message TemplateParameter1 {
      required  Class  class_name = 1;
      optional    Identifier    identifier_name   = 2;
}
message TemplateParameter2 {
      required  Typename typename_name = 1;
      optional    Identifier    identifier_name   = 2;
}
```

图 3.5　一个简化的结构化格式定义

Fig. 3.5　A simplified example of the structured format

突变文件。同时，在突变过程中，CCOFT 设置了一个上限，以避免无限递归而导致测试程序过大。

接下来，CCOFT 将各个得到突变的文件传递给程序转换器从而构造出真实 C++ 测试程序。具体而言，C++ 程序转换器遵循以下准则进行程序转换：

1）各个元素均可以转化为一个真实的 C++ 代码片段，例如，“Class” 语法元素转化为 C++ 关键词 “class”。

2）对于变量标识符，以不同的基本类型（如 char、int 与 long）提供固定的名称，并在算术表达式中反复使用。对于其他类型的标识符，例如类名或模板名称，CCOFT 维护不同的记录列表，以捕获它们并在需要时从记录列表中获取不同的标识符。由于 CCOFT 中提供了各种类型的标识符的管理，并保证了无未定义的变量，而语法符号[30] 只支持变量标识符的子集，CCOFT 在构造所需的测试程序方面表现得更好。

3）由于 CCOFT 已经设置突变的递归上限，CCOFT 最终根据突变后多样化的结构转换成有效且数量有限的测试程序。

具体而言，以图3.5所示的结构为例，突变器将消息 “TemplateDeclaration” 作为输入。在采用 ECS 策略后，突变的文件传递给程序转换器最终构造出一个 C++ 程序。根据转换准则，‘TemplateParameter” 语法中的各个部分均可以由以下基本元素构造：Class – “class”, Typename – “typename”, Identifier – “T”, 以及 Declaration – “;”, “class A {};”, 或 “void foo(){}”。在经过转换过程后，CCOFT 可以产生数量有限的 C++ 测试程序。对于图3.5中的示例，可以构造出如图3.6所示的 12 个唯一且真实的 C++ 代码片段。通过以上方式，代码片段 3、6 和 9 将触发第3.2节图3.1(b) 所示 LLVM 编译器前

端的接受无效代码缺陷。经过后续的重复缺陷过程以及测试程序约简步骤，最终将能够触发缺陷的测试程序提交给编译器缺陷仓库。

```
1  code 1: template <class> class A {};
2  code 2: template <class> void foo() { }
3  code 3: template <class> ;
4  code 4: template <typename> class A { };
5  code 5: template <typename> void foo() { }
6  code 6: template <typename> ;
7  code 7: template <class T> class A { };
8  code 8: template <class T> void foo () { }
9  code 9: template <class T> ;
10 code 10: template <typename T> class A { };
11 code 11: template <typename T> void foo() { }
12 code 12: template <typename T> ;
```

图 3.6　由 C++ 构造器构造的 12 个不同的 C++ 代码片段
Fig. 3.6　Twelve code snippets generated by C++ program generator

简而言之，通过上述变异准备和变异执行阶段，CCOFT 可以构造出大量可能触发 C++ 编译器前端缺陷的测试程序。尽管此类测试程序严格遵循语法规则，但由于缺少类型检查或包含无效语义，它们可能是无效的。值得注意的是，编译器无法精确地判断一个测试程序在语法上是否有效，它只在语义有效（假设没有缺陷）时才能成功编译程序。因此，构造出的大多数测试程序均无法成功编译。但是，在采用有效缺陷识别策略（详见第3.3.4节）之后，可以编译遵循语法规则的经过过滤和简化的小代码片段，并将其用于检测编译器中的缺陷。总而言之，与最先进的方法相比，CCOFT 有诸多优势。首先，CCOFT 配备了一套完整的变量记录，以避免未定义的标识符问题。其次，结合现有突变器（如 libprotobuf-mutator[11]）支持的多种内在变异算子，CCOFT 提供了一个可配置的选项来控制语法规则的选择。上述功能使 CCOFT 在构造测试程序时更加有效，即更有可能构造出触发深层次编译器前端缺陷的测试程序。

3.3.4　基于差分测试与信息对齐算法的前端缺陷识别

缺陷识别旨在通过差分测试策略识别潜在的 C++ 编译器前端缺陷。具体而言，缺陷识别包括五个组成部分，即 1）差分测试策略，其基于两个不同的编译器 $c1$ 和 $c2$ 及输出 $o1$ 和 $o2$，如果两个编译器在崩溃或超时行为上有差异，则表明其中一个编译器存在缺陷；2）编译器输出信息分解器（Decomposer），用于将 $o1$ 和 $o2$ 分解为缺陷记录 $e1$ 和 $e2$；3）信息对准器（EAligner），用于基于上述两个缺陷记录查找不一致的记录；4）缺陷过滤器（Filter），用于过滤潜在的重复缺陷；5）程序约简器（Reducer），用于将程序约简为可以触发相同缺陷行为的小代码段。CCOFT 在缺陷识别时基于以

[11] https://github.com/google/libprotobuf-mutator

下假设，在理想情况下，两个编译器 $c1$ 和 $c2$ 应为同一输入 p 发出相同或相似的编译器输出记录（即 $e1$ 和 $e2$）。因为在两个编译器之间检测到的任何不一致的编译器输出均会被视为 $c1$ 或 $c2$ 中的一个缺陷（或者两者均存在缺陷），所以以上假设对于差分测试的有效性至关重要。接下来，本节将详细介绍各个组成部分的设计。

（1）差分测试策略

CCOFT 主要采用以下差分策略识别 C++ 编译器前端潜在缺陷。

① 崩溃或超时缺陷检测（Crash or Time-out Detecting，即 CTD）。此策略用于检测当编译器 $c1$ 或 $c2$ 在编译程序 P 时发生崩溃或超时的相关缺陷。对于超时缺陷，如果一个编译器对某段代码编译超过 10 分钟，该段代码将被记录并等待进一步确认该编译器是否存在缺陷。

② 跨版本策略（Cross-Version Strategy，即 CVS）与跨编译器策略（Cross-Compiler Strategy，即 CCS）。上述两种策略在编译器测试中被广泛应用[8,28,46]。CVS 针对不同版本的编译器进行差分测试。而 CCS 选择独立维护的不同编译器进行测试。鉴于本章关注于编译器前端缺陷的检测，故没有采用优化选项相关的差分策略。

③ 跨标准策略（Cross-Standard Strategy，即 CSS）。此策略通过单一编译器在不同的 ISO C++ 标准下编译程序 P。例如，可以使用 GCC 在 $c1$ 下启用 C++11 标准 (即 “-std=c++11”) 进行编译，并在 $c2$ 下启用 C++14 标准（即 “-std=c++14”）进行编译。此处，假定编译器的不一致输出要么由 ISO C++ 标准的升级引起，要么 C++ 编译器前端应发出升级提示的诊断信息。否则，至少其中一个编译器存在缺陷。例如，图3.1(c) 中的诊断信息缺陷是由编译器输出不清晰的标准升级指示导致的。

设计好差分策略后，CCOFT 将对编译器输出的错误诊断信息进行处理，以识别潜在前端缺陷。下文将介绍完成缺陷识别的各个算法设计。

（2）编译器输出信息分解器

编译器输出信息分解器（即 EDecomposer）旨在分解复杂的编译器输出信息。具体而言，信息分解器将原始编译器输出消息作为输入，并输出便于在错误诊断信息记录对准器（即 EAligner）中处理的信息记录。值得一提的是，因为编译器的错误诊断信息均为自然语言描述，且不同编译器以不同的方式发出错误诊断消息，所以直接计算 $e1$ 和 $e2$ 的对称差异是具有挑战性的。为解决以上问题，CCOFT 设计了特定的分解器对各个编译器编译源程序后产生的消息进行分解。算法3.1描述了给定编译器的诊断输出信息分解的一般工作流程。函数 EDecomposer() 获取包含所有错误诊断消息的字符串（在行 4-6 之间），然后将其拆分为列表（在行 7-10 之间）。因此，在文本格式中的各个元素，例如，行号、列号和错误诊断信息描述，均可以被表示为一个单独的错误诊断信息记录。具体而言，该函数通过提取行号、列号和错误诊断信息描述，将输出字符串分解为一个类似于字典的记录。例如，图3.1(b) 中的 GCC 编译缺陷消息或

Algorithm 3.1 编译器输出错误信息分解算法（EDecomposer）

Input: $text$: 编译器输出信息文本
Output: $result$: 一组分解后的编译器错误诊断信息记录

```
1:  function EDecomposer(String text)
2:      error_lines ⟵ ∅ // 含有错误诊断信息的行号
3:      dict ⟵ {} // 一个编译器诊断信息的记录表（由字典结构组成）
4:      result ⟵ ∅ // 所有的输出信息记录
5:      for each line in text do
6:          if 'error' in line then
7:              error_lines.append(line)
8:          end if
9:      end for
10:     for each line in error_lines do
11:         dict = line.split(':')
12:         result += dict
13:     end for
14:     return result
15: end function
```

第 3.3.3节中的收集阶段的情况 3、6 或 9 可以分解为如下记录：{"行号 (row-number)" : "1", "列号 (column-number)": "18", "错误诊断消息描述 (Description)": "error: expected unqualified-id before ';' token "}。

（3）错误诊断信息记录对准器

在获取错误诊断信息记录后，信息记录对准器的目的是提取不一致的信息记录。其中，不一致的信息记录意味着潜在的前端缺陷。总体而言，对准器的输入为两组已分解的错误诊断信息记录，输出不一致的错误诊断信息记录。具体而言，通过在 $e1$ 和 $e2$ 中对齐编译器输出的错误诊断信息，可以得到编译器、编译器版本或 ISO C++ 标准之间的不一致性行为。对准器的输出是一系列错误诊断信息记录对，其中第一个元素要么是 $e1$ 中的一个潜在缺陷，要么是 $\perp$（即无缺陷），另一个要么是 $e2$ 中的一个缺陷，要么是 $\perp$。特别地，对准器在工作时将产生以下两种记录对 (a,b)：

① 等价记录：$a \in e1 \wedge b \in e2$，并且两者均有相同的位置（即行号和列号），并且两个错误诊断信息的描述中不包含 C++ 标准升级的信息（如 "C++99 standard" 字样）。由于此类别的记录指示编译器前端缺陷的概率较小，CCOFT 将其忽略。

② 缺失记录：$(a \in e1 \ \wedge b = \perp) \vee (a = \perp \wedge b \in e2)$。此类别包括需要用户进一步确定的不一致信息。该类记录是 CCOFT 感兴趣的信息。

算法3.2展示了对准器（EAligner）的具体工作流程。它首先从 $e1$ 和 $e2$ 中剔除所有等效的信息记录对（在第 5 行和第 7 行之间），然后从 $e1 \backslash rm_1$ 和 $e2 \backslash rm_2$ 中的剩余记录构建不一致的信息记录对（在第 9 行和第 13 行之间）。以第3.3.4节图3.1(b) 列举的代码片段为例，由于 Clang 在程序下没有输出，即出现了不一致的输出，CCOFT 将

Algorithm 3.2 错误诊断信息记录对准算法（EAligner）

Input: $e1$ 和 $e2$: 经分解器处理后的两组错误诊断信息记录
Output: $missing$: $e1$ 和 $e2$ 的缺失记录

```
1:  function EAligner(String text)
2:      rm1 ⟵ ∅ // 一组从 e1 移除的成员
3:      rm2 ⟵ ∅ // 一组从 e2 移除的成员
4:      步骤 1: 移除对等的信息记录
5:      for each (a, b) ∈ (e1 × e2) do
6:          if (a, b) is an equivalent pair then
7:              rm1 ⟵ rm1 ∪ {a}
8:              rm2 ⟵ rm2 ∪ {b}
9:          end if
10:     end for
11:     // 步骤 2: 记录缺失信息记录
12:     missing ⟵ ∅ // 一组缺失的信息记录对
13:     for each a ∈ (e1\rm1) do
14:         missing = missing ∪{(a, ⊥)}
15:     end for
16:     for each b ∈ (e2\rm2) do
17:         missing = missing ∪{(⊥, b)}
18:     end for
19:     return missing
20: end function
```

把该记录保存为缺失的记录 “$(e1, \perp)$”。

（4）崩溃缺陷与不一致信息过滤器

在从 EAligner 获得崩溃或超时程序和不一致的记录后，列表中的记录可能会重复。为了提高缺陷识别效率，有必要进一步对重复的记录进行过滤。过滤器的目标即为删除重复的测试程序或错误诊断信息记录。算法3.3描述了过滤重复程序或者记录的整体过程。对于导致编译器崩溃的测试程序，CCOFT 执行每一个导致编译器崩溃的测试程序并记录编译器崩溃的堆栈信息。如果崩溃的源头（即崩溃或断言失败的具体位置）不在 crash_set 中，CCOFT 将其添加到集合中（在第 4 行和第 6 行之间）。例如，“*internal compiler error: in cxx_incomplete_type_diagnostic, at cp/typeck2.c:584*” 和 “*TextDiagnostic.cpp:1026 Assertion ‘StartColNo <= EndColNo “Invalid range!” ’ failed*” 是 GCC 和 Clang 的 crash_set 中的两个不同记录。由于超时程序数量较少（仅两个测试程序），CCOFT 不对其进行单独过滤。对于不一致的缺陷诊断程序，CCOFT 首先根据 $e1$ 或 $e2$ 记录中的消息部分删除重复记录（在第 7 行和第 9 行之间）。具体而言，由于 GCC 和 Clang 之间的不兼容性，GCC 中的一个缺陷可能对应 Clang 中的两个或更多缺陷，反之亦然。在手动分析各个不一致的记录后，Filter 采用已有记录自动过滤相同的诊断信息记录。在增量分析过程的帮助下，Filter 只需要分析新的不一致的

记录，从而提高识别真实缺陷的效率。在实际操作中，作者发现记录数量通常为 100 以内，所以以上手动分析的处理是可以接受的。例如，在第3.3.3节中收集的实例 3、6 和 9 将在对准器中产生三个重复的缺失记录。最终，过滤算法将上述三种情况过滤并输出一个唯一的记录，并将该记录对应的测试程序传递给约简器进行约简操作。

Algorithm 3.3 崩溃缺陷与不一致信息过滤算法（Filter）

Input: $crash\_source$: 使编译器崩溃的源代码;
$re\_missing$: 经对准算法处理后的缺失信息记录
Output: $records$: 一组唯一的错误诊断信息记录

```
1:  function Filter(File crashed_sourced, String re_missing)
2:      crash_set ⟵ ∅ //一组唯一的崩溃缺陷记录
3:      missing_set ⟵ ∅ // 一组唯一的缺失信息记录
4:      // 步骤 1：过滤崩溃缺陷
5:      for each s.cc in crashed_source do
6:          if s.cc not in crash_set then
7:              crash_set.append(s.cc)
8:          end if
9:      end for
10:     // 步骤 2：过滤不一致的信息记录
11:     for each miss in missing do
12:         if miss not in missing_set then
13:             missing_set.append(miss)
14:         end if
15:     end for
16:     return [crash_set, missing_set]
17: end function
```

（5）测试程序约简过程

现有研究[18] 表明，揭示缺陷的测试程序代码行数通常较少，80%的用例代码行数少于 45 行。因为精简后的测试程序不仅可以帮助测试人员避免提交重复的缺陷报告，还可以协助开发人员及时对该缺陷进行分类或修复。因此，当测试程序触发编译器的缺陷且在缺陷提交到 GCC 或 Clang 缺陷库之前，通过移除无关程序将程序减少到较少的代码行数是至关重要的。因此，和已有研究类似，CCOFT 同样采取对程序约简的方法保留较小的代码片段，以促进缺陷的报告和修复进程。对于崩溃和超时测试程序，本文采用现有工具 C-Reduce[17] 进行自动化精简。对于触发不一致诊断信息缺陷的测试程序，由于 C-Reduce 不能很好地对程序进行约简，本文采用手动的方法对相关测试程序进行约简。具体而言，当只需要为测试程序保留特定的错误诊断消息时，经过 C-Reduce 约简的测试程序总是触发其他的错误诊断消息，该约简结果对开发人员找出让编译器产生唯一错误诊断信息的帮助较小。在本章研究中，由于 CCOFT 构造的各个测试程序均相对较小，此手动约简过程不需要花费太多时间（最多 10 分钟），因

此是可以接受的。此外，在手动约简的过程中，作者试图在编译代码片段时保留唯一的错误诊断消息，并根据作者的经验调整代码。在实际操作中，调整的目标是使 GCC 产生一个特定诊断消息而 Clang 则不产生任何错误诊断信息，或者使 Clang 产生一个特定诊断消息而 GCC 则不产生任何错误诊断信息。因此，通过以上设计，CCOFT 可以检测到拒绝有效代码、接受无效代码或诊断信息缺陷。需要说明的是，虽然采用了上面的人工分析过程，但是本文的实践结果表明该简化过程在实践中是有效的：一个测试程序最终可以减少到几行（通常在 5 行内）。作者将在未来工作中对此类代码的自动约简问题进行深入研究。

3.4 实验设计

本章详细介绍用于评估 CCOFT 有效性的实验设计方案，包括数据集与实验平台、研究问题、方法实现与实验设置、实验结果分析以及相关讨论。

3.4.1 数据集与实验平台

（1）数据集与研究对象

关于数据集，本章研究所用的数据集为 CCOFT 及相关对比方法构造出来的测试程序。关于研究对象，和现有编译器测试研究[4–6,28,42] 类似，在本章中选择 GCC 和 Clang 两个主流的编译器作为研究对象。GCC（GNU Compiler Collection）是由 GNU 项目开发的编译器套件，最初作为 GNU 操作系统的一部分。它支持多种编程语言，包括 C、C++、Objective-C、Fortran、Ada、Go 和 D 等，已被广泛应用于多种操作系统平台和架构之上，是开源软件中最为广泛应用的编译器之一。LLVM（Low Level Virtual Machine）是一种编译器及工具链的集合，旨在优化编译时、链接时和运行时的程序性能。LLVM 提供了一种中间代码表示（Intermediate Representation，简称 IR），允许先进的优化和转换。它支持多种前端语言，如 C、C++ 和 Objective-C，通过 Clang 前端，还可以实现与 GCC 高度兼容的编译效果。

（2）实验平台

本章所有的实验均在 Intel(R) CoreTM i7-7700 CPU @3.60GHZ × 8 处理器和 16GB RAM 的 Linux PC 上进行，该机器运行的是 Ubuntu 18.04 操作系统。

3.4.2 研究问题

为了充分评估 CCOFT 的有效性，本章实验将探索以下三个研究问题（Research Questions，简称 RQs）：

① RQ1：与最先进的方法相比，CCOFT 是否能在编译器前端检测更多的缺陷？

RQ1 旨在将 CCOFT 与两种先进的方法（即 Dharma 和 Grammarinator）进行对比，评估 CCOFT 是否能够在缺陷检测能力上优于现有方法。具体而言，在相同的测试期

间运行 CCOFT、Dharma 和 Grammarinator，并从两个方面进行对比，即检测到的缺陷总数以及独特的缺陷数量（某一方法可以检测到但其他方法无法检测到的）。

② RQ2：提出的 ECS 策略是否助于 CCOFT 在编译器前端检测到更多的缺陷？

RQ2 旨在评估本章所提出的 ECS 策略是否对 CCOFT 的缺陷检测能力有积极的贡献。具体而言，本实验对比了 CCOFT 及其变种 CCOFT(¬ECS)（采用默认选择策略的版本）以评估 ECS 策略对 CCOFT 的贡献。

③ RQ3：CCOFT 在实践中的缺陷检测能力如何？

RQ3 旨在评估 CCOFT 在实践中检测 C++ 编译器前端缺陷的能力，即实用价值。具体而言，本实验将 CCOFT 运行在主干开发版本的编译器上，并从检测到的缺陷数量、已确认缺陷的类型、缺陷的重要性及编译器开发者对报告缺陷的反馈等几个方面评估 CCOFT 在实践中的缺陷检测能力。

3.4.3 方法实现与实验设置

（1）CCOFT 的实现

CCOFT 的实现主要分为两部分：C++ 程序构造器和前端缺陷识别器。对于 C++ 程序构造器的实现，采用 Grammar-v4 中的 C++ 语法文件[12]作为输入，该文件是多种 ANTLR[13]语法的集合，由全球的开发者共同开发并积极维护。CCOFT 中结构感知变异策略 ECS 是通过 Google 公司研发的 libprotobuf-mutator 实现的，该库支持随机变异结构化格式（如 Protobuf）文件，并在支持用户自定义变异策略方面具有良好的扩展性。具体而言，该库提供一个标准的结构化格式文件（即 protobuf-specification 文件）来描述输入（即 C++ 语法定义）的结构。此结构化格式文件随后被编译成 C++ 类的相关代码。一个程序输入对应于一个类似 C++ 对象的结构；通过 libprotobuf-mutator 变异器生成的结构化语法定义结构进行操作：它将给定对象修改为突变对象。此外，本文还实现了一个程序转换函数以实现程序转换器，将类 C++ 对象转化为 C++ 测试程序。关于缺陷识别器的实现，各个部分，即 EDecomposer、EAligner、Filter 和 Reducer，均由 Python 或 Shell 脚本语言编写。整个工具的实现代码和测试脚本代码共同完成了 CCOFT 对编译器前端的自动化测试。

（2）RQ1 实验设置

① RQ1 的对比方法。为了评估 CCOFT 的缺陷检测能力，将 CCOFT 与两种最先进的方法（即 Dharma[31] 和 Grammarinator[30]）进行了比较。Dharma 是由 Mozilla 开发的一个基于语法的模糊测试器，它允许用户定义一个语法文件，然后在给定的语法下构造测试程序。Grammarinator 是一个随机测试程序生成器，它根据 grammar-v4 中的语法创建测试程序。在本实验中选择 Dharma 和 Grammarinator 的原因有两点。其

[12]https://github.com/antlr/grammars-v4

[13]https://www.antlr.org

一，它们均采用基于语法的方法构造测试程序，所以它们与 CCOFT 是最直接相关。其二，它们是本文进行本项研究时较先进的方法，在近年来被多项研究采用（如已有研究[87,88] 在实验评估中均采用了上述两个工具作为对比方法）。

② 实验运行设置。为了实现公平比较，测试周期均设置为 10 天（与先前研究[89] 相同）。值得注意的是，Dharma 和 Grammarinator 均不能直接采用 grammar-v4 仓库中定义的 C++ 语法。为了配置运行 Dharma 工具，本文根据在线的教程[14]并将 C++ 语法转化为 “.dg” 格式以构造测试程序。Grammarinator 的运行设置同样根据在线的教程[15]完成。此外，由于 Grammarinator 不能处理较大的递归深度（例如 50），为了进一步实现公平比较，本文将三种工具的递归深度均设置为 30。此外，为了通过分析 GCC 和 Clang 的缺陷仓库来验证检测到的缺陷是否为真实缺陷，实验中选择了两个开发主干版本的编译器，分别是 GCC[16]和 Clang[17]。

（3）RQ2 实验设置

① RQ2 的对比方法。为了评估新提出 ECS 策略的有效性，本实验中将 CCOFT 与 CCOFT(¬ECS) 进行对比。CCOFT(¬ECS) 是不配备 ECS 的 CCOFT 的一个变种，以检查 ECS 策略对 CCOFT 的贡献。在本章中，本文采用 libprotobuf-mutator 提供的默认变异策略来在 CCOFT(¬ECS) 中进行结构感知变异。该默认策略在结构化格式文件中以高概率（99%）选择 “required” 和 “oneof” 字段，而在结构化格式文件中以低概率（1%）应用 “optional” 字段。由于 libprotobuf-mutator 的默认变异策略也是有效的，实验中选择该策略作为 ECS 的对比策略。

② 实验运行设置。为了评估所提出 ECS 策略的影响，本实验运行与 RQ1 中相同的测试期间（即 10 天），并采用相同的编译器版本运行 CCOFT 与 CCOFT(¬ECS)。然后，通过两种方法检测到的缺陷数量对比 CCOFT 与 CCOFT(¬ECS) 的缺陷检测能力。

（4）RQ3 实验设置

为了评估 CCOFT 在实践的缺陷检测能力（即实用价值），因为编译器开发人员在开发版本中修复缺陷的速度总是比在稳定版本中快[4–6]，所以本文在三个月的非连续时期（从 2020 年 6 月初到 8 月中旬）内对每日更新的 GCC 和 Clang 的开发版本进行测试。具体而言，本章主要从三个方面评估 CCOFT 在实践中的缺陷检测能力，即检测到的缺陷数量、已经被编译器开发者确认或者修复的缺陷类型以及缺陷的重要性（包括开发者对 CCOFT 报告缺陷的积极反馈）。

（5）差分测试实施过程

在上述三个实验的缺陷识别过程中，本文采用在第3.3.4介绍的四种策略对 C++

[14]https://github.com/MozillaSecurity/dharma

[15]https://github.com/renatahodovan/grammarinator/issues/21

[16]GCC commit by 05430b9b6a7c4aeaab595787ac1fbf6f3e0196a0

[17]Clang commit by f4b0ebb89b3086a2bdd8c7dd1f5d142fa09ca728

编译器前端进行差分测试，即崩溃或超时检测（CTD）、跨版本策略（CVS）、交叉编译器策略（CCS）和跨标准策略（CSS）。对于 CVS 和 CCS，采用了两个 GCC（GCC-10.1 和 GCC 的开发版本）和 Clang（Clang-10.0 和 Clang 的开发版本）比较。在 CSS 场景中，采用一组著名的 ISO C++ 标准版本（即 C++11，C++14 和 C++17）检测 C++ 编译器前端中因不同 C++ 标准而触发的缺陷。对于 CTD，在编译测试程序时，如果上述三种策略被终止，即编译器崩溃或者超时，CCOFT 将收集该测试程序然后继续测试过程。CCS、CVS 和 CSS 策略的目标是检测由不一致的编译器输出导致的缺陷，即拒绝有效代码、接受无效代码和诊断信息缺陷，而 CTD 的目标是检测崩溃或超时缺陷。

3.4.4 评价指标

本节主要采用检测到的缺陷数量评估 CCOFT 的有效性。在 RQ1 中，本章从两个方面分析了 CCOFT 与两种先进方法在缺陷检测能力上的对比，即检测到的缺陷总数和唯一缺陷的数量。在 RQ2 中，本章主要采用检测到的缺陷总数对不同的对比方法进行评估。对于 RQ3，除了阐述检测到的缺陷数量，本章还从不同的角度进一步分析了各个由 CCOFT 提交缺陷的具体情况，包括其优先级、报告状态、缺陷类型、采用策略以及影响的编译器版本。

3.5 实验结果分析

3.5.1 和现有方法对比分析

表3.1列出了 CCOFT 与两种对比方法在缺陷检测数量上的比较结果。在表3.1中，第一列是 RQ1 中检测到的缺陷类型，第二到四列是 Dharma、Grammarinator 和 CCOFT 检测到的所有缺陷的数量。具体而言，“n(x+y)” 表示相应的方法完全找到了 “n” 个缺陷，“x” 和 “y” 分别表示在 GCC 和 Clang 中检测到一定数量的缺陷。从表中可以看出，CCOFT 检测到的缺陷总数为 40 个，远远大于 Dharma（即 17 个）和 Grammarinator（即 19 个）检测到的缺陷总数，即 CCOFT 在缺陷检测数量上分别提高了 135% 和 111%，充分证实了 CCOFT 在编译器前端缺陷检测能力上优于现有方法。

为了进一步比较三种对比方法在检测到的唯一缺陷之间的关系（即有多少缺陷是只有 CCOFT 可以检测出来的缺陷），图3.7中的六幅维恩图进一步展示了各个方法检测到的唯一缺陷的数量。特别地，图3.7(a) 到图3.7(e) 显示了按缺陷类型分类的每种方法检测到的缺陷的维恩图，图3.7(f) 是每种方法检测到的总体缺陷的数量。从图3.7整体可以看出，CCOFT 总是能检测到最大数量的唯一缺陷。然而，Grammarinator 只检测到一个唯一的崩溃缺陷，而 Dharma 在测试期间没有检测到任何唯一的缺陷。特别是从图3.7(d) 中，CCOFT 检测到的唯一缺陷总数为 13，比 Dharma（即 0）和 Grammarinator（即 1）检测到的缺陷数量大一个数量级。此外，值得一提的是，在为期 10 天的测试

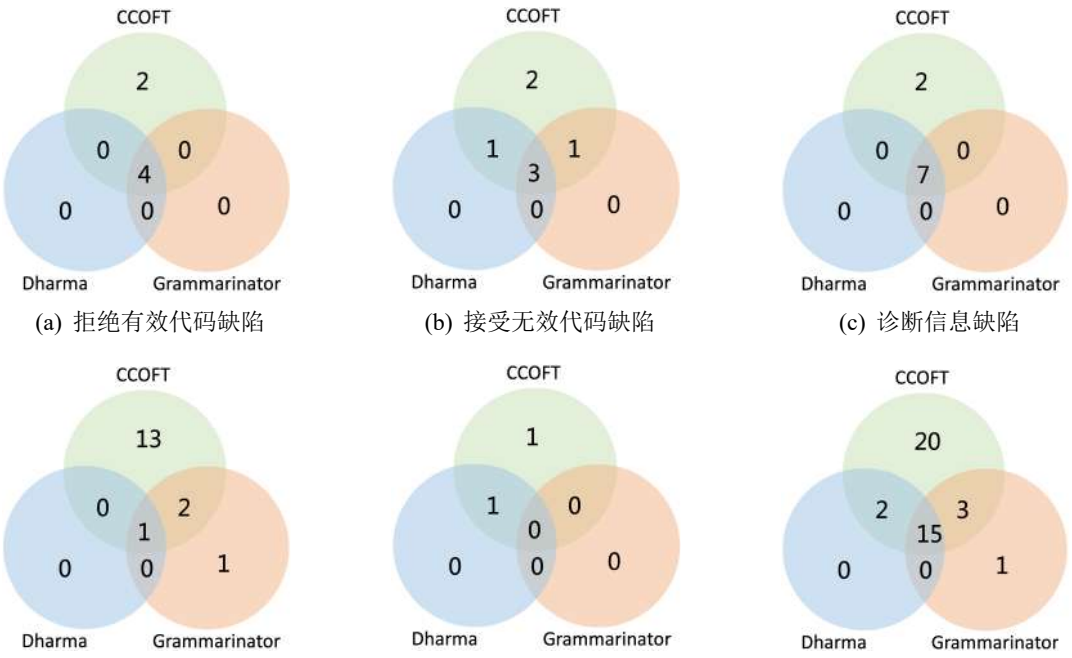
表 3.1　CCOFT 和比较方法所检测到的缺陷数量对比结果

Tab. 3.1　The number of bugs detected by CCOFTand the comparative approaches

缺陷类型	Dharma	Grammarinator	CCOFT
Reject-valid	4 (4+0)	4 (4+0)	6 (6+0)
Accept-invalid	4 (3+1)	4 (3+1)	7 (5+2)
Diagnostic	7 (6+1)	7 (6+1)	9 (7+2)
Crash	1 (1+0)	4 (4+0)	16 (13+3)
Time-out	1 (1+0)	0 (0+0)	2 (1+1)
Total	17	19	40

期间，CCOFT 可以检测到 98%（41 个缺陷中的 40 个）的缺陷，充分证实了 CCOFT 在编译器前端缺陷检测方面优于现有方法。

为了进一步验证 CCOFT 在统计学上的有效性，对实验数据进行了可信度为 0.05 的曼-惠特尼 U 检验[90]（即 Mann-Whitney U-test）其中零假设为 “CCOFT 和比较方法之间没有显著差异”，备择假设为 “CCOFT 和比较方法之间存在显著差异”。计算结果 P-值（P-value）均小于或等于 0.05，表明备择假设成立，即 CCOFT 性能优于其他对比方法的实验结论是具有统计显著性的。



(a) 拒绝有效代码缺陷　(b) 接受无效代码缺陷　(c) 诊断信息缺陷

(d) 崩溃缺陷　(e) 超时缺陷　(f) 总体统计

图 3.7　CCOFT 和现有方法在检测到的唯一缺陷数量对比结果

Fig. 3.7　The number of unique bugs found by CCOFT and comparative approaches

对 RQ1 的回答：与现有最好方法（即 Dharma 和 Grammarinator）相比，实验结果表明 CCOFT 具有更好的缺陷检测能力，在检测到的缺陷数量上比 Dharma 和 Grammarinator 分别提高了 135% 和 111%。

3.5.2 新设计组件的有效性分析

结果表3.2显示了 CCOFT 与 CCOFT(¬ECS) 在缺陷检测数量上的比较结果。从表中可以看出，采用 ECS 策略的 CCOFT 总是能检测到比默认选择策略更多的缺陷。特别地，CCOFT 可以检测到 16 个崩溃缺陷，而 CCOFT(¬ECS) 只能检测到 4 个崩溃缺陷，即 CCOFT 在检测崩溃缺陷方面实现了 300% 的提升。总体而言，CCOFT 总共可以检测 40 个缺陷，而 CCOFT(¬ECS) 只能检测到 22 个缺陷，即 CCOFT 在检测到的缺陷数量方面比 CCOFT(¬ECS) 提高了 82%。由于在 CCOFT(¬ECS) 中不支持更多的机会选择不同种类的语法元素，CCOFT(¬ECS) 构造的测试程序只能包含较少的语法元素，所以此类测试程序触发编译器缺陷的可能性较低，因此该结果是合理的。此外，通过支持更多的机会选择不同种类的语法元素也辅助 CCOFT 构造出语法多样化的测试程序，从而大大提高 CCOFT 检测深层次前端缺陷的能力。

表 3.2　CCOFT 和 CCOFT(¬ECS) 检测到的缺陷数量对比结果
Tab. 3.2　The number of bugs detected by CCOFT and CCOFT(¬ECS)

缺陷类型	CCOFT(¬ECS)	CCOFT
Reject-valid	3 (3+0)	6 (6+0)
Accept-invalid	6 (4+2)	7 (5+2)
Diagnostic	8 (7+1)	9 (7+2)
Crash	4 (2+2)	16 (13+3)
Time-out	1 (1+0)	2 (1+1)
Total	22	40

对 RQ2 的回答： CCOFT 中的等概率选择策略（即 ECS）对 CCOFT 在 C++ 编译器前端缺陷检测上具有积极的贡献。具体而言，在相同的测试期间内，CCOFT 比 CCOFT(¬ECS) 多检测出了 82% 的编译器缺陷。

3.5.3 实践中缺陷检测能力分析

本小节首先对 CCOFT 报告的缺陷进行定量和定性分析（包括缺陷数量和缺陷类型），然后分类列举由 CCOFT 检测到的一些有代表性的缺陷，以进一步说明 CCOFT 检测到的缺陷在实践中的重要影响。

表 3.3　由 CCOFT 报告的已确认/已分配/已修复的缺陷细节统计（第一部分）
Tab. 3.3　Details of confirmed/assigned/fixed bugs reported by CCOFT (Part 1)

序号	缺陷报告编号	优先级	报告状态	缺陷类型	采用策略	影响的编译器版本
1	GCC-95597	P3	Confirmed	*Reject-valid*	CCS	10.1-11.0 (trunk)
2	GCC-95610	P3	Confirmed	*Reject-valid*	CCS	10.1-11.0 (trunk)
3	GCC-95641	P3	Confirmed	*Diagnostic*	CCS	10.1-11.0 (trunk)
4	GCC-95657	P3	Confirmed	*Diagnostic*	CSS	10.1-11.0 (trunk)
5	GCC-95672	P3	Fixed	*Crash*	CTD	10.1-11.0 (trunk)
6	GCC-95742	P5	Confirmed	*Diagnostic*	CCS	10.1-11.0 (trunk)
7	GCC-95744	P3	Confirmed	*Diagnostic*	CCS	10.1-11.0 (trunk)
8	GCC-95807	P3	Confirmed	*Accept-invalid*	CCS	10.1-11.0 (trunk)
9	GCC-95820	P3	Fixed	*Crash*	CTD	10.1-11.0 (trunk)
10	GCC-95872	P5	Confirmed	*Diagnostic*	CCS	6.1-11.0 (trunk)
11	GCC-95925	P5	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
12	GCC-95927	P5	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
13	GCC-95932	P5	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
14	GCC-95935	P3	Assigned	*Crash*	CTD	10.1-11.0 (trunk)
15	GCC-95937	P3	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
16	GCC-95938	P3	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
17	GCC-95945	P5	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
18	GCC-95954	P5	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
19	GCC-95956	P5	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
20	GCC-95972	P3	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
21	GCC-95999	P3	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
22	GCC-96045	P1	Fixed	*Diagnostic*	CVS	11.0 (trunk)
23	GCC-96048	P5	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
24	GCC-96068	P3	Fixed	*Reject-valid*	CCS	10.1-11.0 (trunk)
25	GCC-96077	P3	Fixed	*Reject-valid*	CCS	10.1-11.0 (trunk)
26	GCC-96082	P3	Fixed	*Reject-valid*	CCS	10.1-11.0 (trunk)
27	GCC-96103	P3	Fixed	*Diagnostic*	CCS	10.1-11.0 (trunk)
28	GCC-96116	P3	Confirmed	*Accept-invalid*	CCS	10.1-11.0 (trunk)
29	GCC-96119	P3	Confirmed	*Accept-invalid*	CCS	10.1-11.0 (trunk)
30	GCC-96137	P1	Fixed	*Time-out*	CTD	10.1-11.0 (trunk)

（1）检测缺陷的基本统计

表3.5列举了截止论文发表时所有报告缺陷的详细情况。总体而言，CCOFT 已经向两个编译器 GCC 和 Clang 报告了共 136 个缺陷，其中 67 个已经被开发者确认（Confirmed）、分配（Assigned）或者修复（Fixed）。由于在测试期间，主干版本发生的变化可能会影响缺陷的处理，所以开发人员可能需要处理完其他更新版本带来的问题才能考虑报告的缺陷。开发人员将受版本更新影响的缺陷报告标记为 “WorksForMe”，结果有三个 Clang 缺陷属于该类别。因为 GCC 开发人员对于报告的缺陷反馈比在 Clang 快，所以此类缺陷在 GCC 中不存在。值得注意的是，由于活跃的 Clang 开发人员忙于 Swift 项目[28]，导致人力资源有限，因此，Clang 中的 58 个缺陷只有 10 个被确认或修复。此外，本文还报告了一些重复的无效报告，该类报告被开发人员标记为 “Invalid”

表 3.4　由 CCOFT 报告的已确认/已分配/已修复的缺陷细节统计（第二部分）
Tab. 3.4　Details of confirmed/assigned/fixed bugs reported by CCOFT (Part 2)

序号	缺陷报告编号	优先级	报告状态	缺陷类型	采用策略	影响的编译器版本
31	GCC-96162	P4	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
32	GCC-96182	P3	Confirmed	*Diagnostic*	CSS	10.1-11.0 (trunk)
33	GCC-96183	P3	Confirmed	*Accept-invalid*	CSS	10.1-11.0 (trunk)
34	GCC-96184	P2	Confirmed	*Accept-invalid*	CSS	10.1-11.0 (trunk)
35	GCC-96209	P3	Confirmed	*Diagnostic*	CSS	10.1-11.0 (trunk)
36	GCC-96328	P4	Fixed	*Crash*	CTD	10.1-11.0 (trunk)
37	GCC-96329	P4	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
38	GCC-96359	P4	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
39	GCC-96360	P3	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
40	GCC-96364	P3	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
41	GCC-96380	P2	Fixed	*Crash*	CTD	10.1-11.0 (trunk)
42	GCC-96437	P4	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
43	GCC-96438	P5	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
44	GCC-96440	P4	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
45	GCC-96441	P3	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
46	GCC-96442	P4	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
47	GCC-96462	P2	Fixed	*Crash*	CTD	10.1-11.0 (trunk)
48	GCC-96464	P3	Confirmed	*Accept-invalid*	CCS	10.1-11.0 (trunk)
49	GCC-96465	P4	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
50	GCC-96467	P4	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
51	GCC-96478	P3	Confirmed	*Accept-invalid*	CCS	10.1-11.0 (trunk)
52	GCC-96552	P3	Confirmed	*Accept-invalid*	CCS	10.1-11.0 (trunk)
53	GCC-96553	P3	Confirmed	*Crash*	CCS	10.1-11.0 (trunk)
54	GCC-96623	P1	Fixed	*Crash*	CTD	10.1-11.0 (trunk)
55	GCC-96636	P3	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
56	GCC-96637	P5	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
57	GCC-96638	P4	Confirmed	*Crash*	CTD	10.1-11.0 (trunk)
58	Clang-46231	P3	Fixed	*Accept-invalid*	CCS	10.0-12.0 (trunk)
59	Clang-46417	P3	Fixed	*Diagnostic*	CCS	10.0-12.0 (trunk)
60	Clang-46425	P3	Confirmed	*Diagnostic*	CCS	10.0-12.0 (trunk)
61	Clang-46428	P3	Confirmed	*Diagnostic*	CCS	10.0-12.0 (trunk)
62	Clang-46484	P3	Fixed	*Crash*	CTD	10.0-12.0 (trunk)
63	Clang-46487	P3	Fixed	*Crash*	CTD	10.0-12.0 (trunk)
64	Clang-46540	P3	Confirmed	*Crash*	CTD	10.0-12.0 (trunk)
65	Clang-46682	P3	Fixed	*Crash*	CTD	10.0-12.0 (trunk)
66	Clang-46729	P3	Fixed	*Accept-invalid*	CCS	10.0-12.0 (trunk)
67	Clang-46859	P3	Fixed	*Crash*	CTD	10.0-12.0 (trunk)

（详见后文关于误报率的更多计算和讨论）。

表3.3和3.4进一步列出了所有已确认或修复的缺陷的详细信息，包括它们的缺陷报告编号、优先级、提交论文时缺陷报告的状态、缺陷类型、检测缺陷所采用策略和受影响的编译器版本（其中的 trunk 表示主干开发版本）。其中的缺陷状态包括 Fixed/Confirmed/Assigned，Fixed 表示该缺陷已经被开发者修复，Confirmed 表示该缺

陷已经确认是一个真实有效的缺陷，Assigned 表示开发者已经在编写相应的补丁修复该缺陷。因为在所有已确认的缺陷中，只有一个缺陷被标记为 “Minor”，还有两个缺陷被标记为 “Enhancement”，所以表中没有列出其严重性状态。值得注意的是，表中只列出了已测试的编译器中受影响的版本范围，在实际中可能会有大量版本的编译器受到提交的缺陷报告的影响。例如，表3.3中序号为 10 的缺陷影响了从 GCC-6.1 到当前主干版本的所有版本（成千上万个小版本）。长时间潜伏的缺陷也证实了 CCOFT 所报告的缺陷处于编译器中较深的位置且尚未被其他工具检测到，充分说明了语法正确但包含未定义行为的测试程序有助于检测出深层次的编译器中后端缺陷。

（2）误报率讨论

和已有研究[6]类似,本章采用以下计算方式来计算误报率: $[\frac{rejected}{reported}, \frac{rejected+pending}{reported}]$。在实验评估中，误报率的范围是 $[\frac{4}{136}, \frac{4+49}{136}] = [3\%, 39\%]$。值得注意的是，由于相对大量的等待确认的 “Pending” 缺陷（特别是对于 Clang 编译器），所以 39% 只是假阳性率的一个上限。对于各个潜在缺陷，由于在报告之前均对其有效性进行了仔细检查，所以作者相信大多数 “Pending” 的缺陷均将被开发者确认并修复。

表 3.5　提交给 GCC 和 Clang 编译器的缺陷数量统计

Tab. 3.5　The number of all the reported bugs for GCC and Clang

缺陷报告状态	GCC	Clang	总计
Fixed	13	7	20
Confirmed	43	3	46
Assigned	1	0	1
Worksforme	0	3	3
Pending	10	39	49
Duplicate	10	3	13
Invalid	1	3	4
Total	78	58	136

表 3.6　在 GCC 和 Clang 中已确认缺陷的类型统计

Tab. 3.6　The number of bug types of confirmed bugs for GCC and Clang

缺陷类型	GCC	Clang	总计
Reject-valid	5	0	5
Accept-invalid	8	2	10
Diagnostic	9	3	12
Crash	34	5	39
Time-out	1	0	1
Total	57	10	67

（3）缺陷类型统计

根据第3.2节中提到的五类典型的编译器前端缺陷，即拒绝有效代码、接受无效代码、诊断信息、崩溃和超时缺陷，表4.5显示了 CCOFT 检测到的每种已确认的缺陷数量。从表4.5中可以看出崩溃缺陷的数量比其他类型的缺陷要多。在 67 个已被开发者确认/分配/修复的缺陷中，有 39 个崩溃缺陷，表明崩溃缺陷是目前降低 C++ 编译器前端质量的最主要原因。关于此类崩溃缺陷的重要影响，详见第3.6节中的讨论。

（4）缺陷重要性说明

在 GCC 和 Clang 的缺陷仓库中，缺陷的重要性被描述为优先级和严重性的组合。优先级是指修复缺陷的优先级，严重性度量缺陷的影响，从最严重的 “Release Blocker” 到最不严重的 “Enhancement”。以上两个字段在缺陷生命周期中均会由开发人员在具体实施调试缺陷时进行调整。如表3.3和表3.4所示，大多数确认的缺陷均具有默认的优先级 P3，即其中 42 个（69%）被标记为 P3 及以上。只有一个报告的缺陷被标记为 “Minor”，还有两个被开发人员标记为 “Enhancement”，其余的均被默认标记为 “Normal” 严重程度。值得注意的是，编译器开发人员对本文报告的编译器前端缺陷十分重视，CCOFT 报告的 19 个缺陷已经在 GCC 和 Clang 的最新发布版本中得到了修复。另外，有一个缺陷被确认并标记为 “Assigned”，意味着开发者们正在积极修复该缺陷。此外，开发者对报告缺陷的反馈对于评估 CCOFT 在实践中的缺陷检测能力十分重要。有力证据表明，6 个缺陷被标记为 “P1”（即 GCC#96045、GCC#96137 以及 GCC#96623）或 “P2”（即 GCC#96184、GCC#96380 以及 GCC#96462），它们被视为 GCC 社区中最严重和最紧急的缺陷。同样值得一提的是，CCOFT 提交的漏洞获得了较高的评价，正如一个 GCC 开发人员所反馈的，“*This case is useful and it shows that the change in somewhere has a corner case that I didn't consider.*”[18]，一位 Clang 开发人员也有同样的评价 — “*These are useful bug reports. Thank you for filing them!*”[19]. 此外，可以注意到，大量报告的缺陷是由语法或者语义无效的测试程序引起的崩溃缺陷，此类缺陷在实践中可能具有重要的现实影响（详见第3.6节讨论部分）。上述缺陷的统计分析及开发者的积极反馈证实了 CCOFT 提交缺陷报告确实是十分重要的，进一步确认了 CCOFT 在实践中具有较强的缺陷检测能力。

（5）各类已确认的缺陷示例分析

此小节挑选了 CCOFT 所检测到的部分缺陷，以分类展示其在 C++ 编译器前端中检测不同类型缺陷的能力。值得注意的是，挑选的缺陷均对保障编译器质量产生了实质性的影响，其中一些甚至被标记为最高严重性的 “P1” 或 “P2”，例如在第3.2节中已进行讨论的 GCC 缺陷 #96137。下面将详述各个由 CCOFT 报告的缺陷示例。

GCC 拒绝有效代码缺陷 #96068。以下测试程序被 GCC 的编译器前端拒绝编译，但被 Clang 的编译器前端接受。GCC 编译器前端的问题在于，函数外的额外分号在

[18]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=96137#c2

[19]https://bugs.llvm.org/show_bug.cgi?id=46487#c2

C++11 之后应该被允许，但 GCC 在几乎所有版本中均拒绝该有效测试程序。

```
1 void foo() { };
```

GCC 拒绝有效代码缺陷 #95610。在以下精简后的测试程序中，存在缺陷的 GCC 编译器前端无法处理类定义中的全局变量（即 GCC 编译该段代码时报错）。同样地，替换类与其他 “class” 说明符（即 “struct” 或者 “union”）时，也会出现同样的情况。而正确的编译器可以接受并正常地编译该段代码。

```
1 class s;
2 class :: s { } ss;
```

GCC 接受无效代码缺陷 #96116。下述无效测试程序被 GCC 的 C++ 编译器前端接受。值得注意的是，“enum struct/class” 声明只能在定义枚举或作为独立前向声明的一部分时使用，其余的用法均为无效。但 GCC 允许将 “enum struct/class” 作为 “using” 声明的一部分，这违反了 C++ 语言标准，从而引发了该缺陷。

```
1 using alias1 = enum struct E1;
2 using alias2 = enum class E2;
```

Clang 接受无效代码缺陷 #46729。Clang 的 C++ 编译器前端接受了以下无效程序。在该段测试程序中，模板显式专用化和类模板部分专用化不应插入 C 语言声明的外部链接。然而，Clang 的 C++ 编译器前端将其视为有效程序，从而错误地接受了该测试程序，导致 Clang 编译器触发了该缺陷。

```
1 template <class> void F( ){ }
2 extern ”C” {
3     template < > void F<int>();
4 }
```

GCC 诊断信息缺陷 #96045。以下测试程序在第 2 行缺少 “;”，但 GCC 的 C++ 编译器前端在错误诊断消息中遗漏了列号。该缺陷被标记为 “P1”，在 GCC 缺陷库中是最紧急的级别，GCC 的开发者很快修复了它。相反，Clang 编译器给出了该段代码的正确诊断信息，可以帮助开发者及时修复程序中的编程错误。

```
1 template <class> class A {};
2 struct A <int>
```

Clang 崩溃缺陷 #46682。以下测试程序使 Clang 的 C++ 编译器前端在编译无效的明确声明时崩溃。该缺陷意味着在某些情况下，Clang 的 C++ 编译器前端在编译该的测试程序时，不能很好地从编译时错误处理的状态完成正常退出。

```
int b = 0;
int foo () { explicit ( && b );}
```

GCC 崩溃缺陷 #95820。在以下代码中，GCC 的 C++ 编译器前端在编译时崩溃。根据开发者的经验，尽管上述程序是一个语法无效的测试程序，但它仍然是十分重要的，且是一个经常会出现的缺陷。因此，增强 C++ 编译器前端以输出正确的缺陷消息而不是直接崩溃是很重要的。

```
constexpr (*a)()->bool,
```

GCC 超时缺陷 #96137。下述测试程序使 GCC 的 C++ 编译器前端陷入对程序的无休止分析。值得一提的是，因为超时编译会浪费开发者在编写运行程序时的时间，并且很难在长时间的编译中找到根本原因，所以超时缺陷对编译器的影响较大。

```
void a () { . operator b }
```

对 RQ3 的回答： CCOFT 在实践中能够有效地检测 C++ 编译器前端的缺陷。在三个月的测试时间内，CCOFT 总共报告了 GCC 和 Clang 的 5 种类型的 136 个编译器前端缺陷。其中，有 67 个缺陷已经被编译器开发人员确认、分配或者修复。

3.6 讨论

本节讨论实验评估中所采用的各种差分测试策略（即 CTD，CVS，CSS 和 CCS）之间的关系、无效代码触发的编译器崩溃缺陷的重要影响、与现有覆盖率引导的模糊测试工具的比较、报告的缺陷位置以及 CCOFT 方法的局限性。

（1）差分策略之间的关系

总体而言，每种差分测试策略在编译器前端检测缺陷时均有独特的能力。具体而言，因为 CTD、CVS 和 CSS 很容易检测到真实的崩溃或超时的缺陷，以及不同编译器版本和不同 C++ 标准中的缺陷，所以通过此类策略检测到缺陷的假阳性率较低。CCS 可以在编译器前端检测到更多类型的缺陷。但是，由于两种不同的编译器之间的差异，CCS 可能报告一些误报。例如，以 GCC 和 Clang 为例，虽然 Clang 的设计是 GCC 的替代品，但它与 GCC 并不完全兼容。因此，如果编译器 $c1$ 和 $c2$ 支持不同的缺陷消息集，则可能出现误报。例如，Clang 允许对变量进行从 “int” 到 “bool” 的转换，而 GCC 没有类似的实现规则。此外，因为 CTD、CVS 和 CSS 策略从不同的角度测试编译器，并且只需要一个编译器就能达到测试的目的，所以可以作为对 CCS 的补充。

（2）由无效代码触发的编译器崩溃缺陷的重要影响

在所有提交的缺陷报告中，可以发现很大一部分属于崩溃缺陷，表明编译器前端的错误处理或错误恢复能力相当有限。值得注意的是，编译器前端中存在的缺陷（特别是在 web 应用程序[20]中采用的编译器）会导致严重的安全漏洞，甚至被攻击者利用[79]。比如，XML 编译器前端[11] 中的一个崩溃缺陷存在一定的安全威胁。具体而言，当 XML 发出崩溃时会显示编译器内部详细的实现信息，如堆栈跟踪、数据库转储和缺陷代码。由于此类信息揭示了不应该透露给用户的实现细节，黑客可以根据此类信息发起对该网站的攻击。简而言之，由无效测试程序引发的崩溃可能导致信息泄漏进而引发各种安全隐患[21]。例如，在 CVE-2017-563816[22]中，由无效的测试输入而导致的编译器前端缺陷已被证明是可利用的。虽然没有此类公开揭露 C++ 编译器前端崩溃引起的安全影响，但是有经验的攻击者可以通过利用两个编译器发出的详细堆栈信息（如在缺陷 GCC#96359 和 Clang#46560）推测一些可行的方式对所编译的软件进行攻击。作者考虑把此类对编译器前端攻击的实现作为未来的工作方向之一。

（3）与覆盖率引导的模糊测试工具的比较

感兴趣的读者也可能关心 CCOFT 和基于覆盖引导的模糊方法之间的性能比较。接下来将讨论本文在采用 CCOFT 和此类工具进行比较的经验和发现。AFL 是一个传统的覆盖率引导模糊测试工具，其在学术界和工业界均被广泛应用。它在某种程度上适用于测试编译器。然而，当在 GCC 编译器上运行 AFL 时，在 7 天的测试期间内没有检测出任何崩溃或性能问题。本文还使用差分测试策略运行测试程序来识别其他类型的缺陷。同样的，在实验期间没有找到任何的编译器前端缺陷。因为 AFL 内置位/字节级突变操作对于构造用于测试编译器前端的缺陷揭示测试程序的帮助较小，大部分由变异产生的测试程序无法通过最开始的词法分析阶段，使得该方法很难检测出较深层次的编译器前端缺陷，所以该结果是合理的。另外本文将 CCOFT 和 Prog-fuzz[23] 进行了对比，其中 Prog-fuzz 的目的是检测编译器崩溃。在 10 天的运行时间内，Prog-fuzz 只检测到了一个 GCC 崩溃，该缺陷也可以被 CCOFT 检测到。相反，Prog-fuzz 没有检测到任何 CCOFT 检测到的其他缺陷。

（4）缺陷触发的位置分析

如第3.2节所述，准确地判断拒绝有效代码、接受无效代码、诊断信息、崩溃或超时缺陷确切地属于编译器前端的哪一部分（即词法分析、语法分析（解析）或语义分析）是具有挑战性的。一般认为，只有在提交的缺陷得到修复后才能知道缺陷的确切位置。例如，对于缺陷 GCC#96077，由于开发人员在源文件 “parser.c” 上修复了它。此外，从代码修复的记录（即修复初步解析时枚举的指定符）中，可以知道该缺陷确实

[20]https://owasp.org/www-community/Improper_Error_Handling

[21]https://sucuri.net/guides/owasp-top-10-security-vulnerabilities-2021/

[22]https://nvd.nist.gov/vuln/detail/cve-2017-5638

[23]https://github.com/vegard/prog-fuzz

存在于 GCC 编译器前端的解析器（语法分析器）中。因为前端的不同阶段总是相互交错的，所以另一个原因来自 GCC 和 Clang 的实现。例如，尽管在语义分析中发生了一个崩溃，但根本原因可能来自以前的解析缺陷。简而言之，除了查看缺陷修复的位置，目前没有可行的方法来精准确定缺陷发生在前端的哪个部分。因此，为了确定缺陷的具体位置，在研究中，作者进一步检查了 20 个已经修复的缺陷，在修复的文件名或开发人员的描述的帮助下，发现其中有 19 个缺陷位于语法分析器中，1 个缺陷位于语义分析器中。上述发现表明 CCOFT 检测出的缺陷确实属于较深层次的编译器前端缺陷，进一步证明了 CCOFT 对编译器前端测试的有效性。

（5）CCOFT 的局限性

CCOFT 的一个局限来自构造程序的方式。CCOFT 首先采用随机的方式构造 C++ 测试程序，然后基于差分测试的策略测试编译器前端。虽然有研究表明并不是所有基于覆盖率的方法均有利于缺陷检测的[91]，但是在 CCOFT 中仍然没有采用任何覆盖率反馈信息。由于以上原因，CCOFT 可能难以在 C++ 编译器前端中找到更深层次的语义缺陷。目前，研究者已经提出了各种基于自动覆盖的模糊方法[92,93] 用于测试不同的软件系统。基于以上的解决方案，作者计划将各种技术集成到 CCOFT 中以测试更广泛的软件系统。CCOFT 中的另一个局限性可能是由 CCOFT 构造出的测试程序有效性受限造成的。因为由 CCOFT 构造出的测试程序在语义上不完全有效，因此可能很难在编译器中触发其他中后端优化缺陷。具体而言，因为构造的测试程序在语法上是正确的但仍然可能是语义无效的（如在 C++ 编程语言中不满足一些类型检查机制），所以在 CCOFT 构造的测试程序可以检测到较深层次的编译器前端缺陷。具体而言，第3.5节的结果证实了 CCOFT 构造的测试程序确实更有可能触发较深层次的 C++ 编译器前端缺陷。换而言之，CCOFT 与现有的基于随机程序构造器方法（如 Csmith[2] 或 YARPGen[3] 等）互补，共同用于保障编译器质量。

（6）有效性威胁

CCOFT 方案主要存在内部、外部和构建有效性方面的威胁。

① 内部有效性威胁。内部有效性威胁主要源自 CCOFT 的实现过程。在本章研究中，所提出的突变策略的有效实现是成功地采用 CCOFT 检测 C++ 编译器前端缺陷的关键。因此，所提出的突变策略的实施可能会影响 CCOFT 的检测效率。为了缓解以上威胁，本文采用了谷歌开发的 Protobuff 结构以及广泛用于随机变异协议缓冲区的库 libprotobuf-mutator 来实现所提出的突变策略。除此之外，参与本工作的两位作者对工具的实现进行了严格的测试和检查，以降低由实现问题带来的威胁。

② 外部有效性威胁。外部有效性威胁主要在于测试程序的约简过程。因为 C-Reduce 不能很好地处理存在不一致的编译器输出行为，因此，本文采取了手动的方法对该类测试程序进行约简。具体而言，在 C-Reduce 的约简过程中，如果只想在简化的

测试程序中保留一个特定的错误诊断信息，在简化的过程中，除了目标诊断信息之外，还会触发一些其他的诊断信息。因为约简的过程可能会很耗时，导致难以对较大的测试程序进行有效约简，所以采用 C-Reduce 约简方法的效率取决于研究人员对 C++ 编程语言的熟练程度。为了减少以上威胁，参与本工作的前两位作者手动对相应的测试程序进行约简，并对约简后的测试程序进行了仔细检查。另一个威胁来自 CCOFT 框架的通用性。一般而言，CCOFT 具有较强的通用性，可以用于测试其他语言编译器前端，理由如下。首先，CCOFT 对 grammar-v4 仓库中 100 多种程序语言的语法均可以进行调优，以构造针对其他程序语言的结构化测试程序。其次，缺陷识别策略可以轻易迁移以识别其他编译器前端（如 Javascript[94,95]）中的缺陷。由于 Javascript 被广泛采用，在使用存在缺陷的该语言系统时更可能导致严重的安全漏洞。

③ 构建有效性威胁。该威胁可能涉及实验中的随机性、采用的评估指标以及评估过程中的参数设置等方面的潜在威胁。关于随机性，实验中将各个方法至少运行了 10 天，以减少因为随机性带来的威胁。对于评估指标，实验采用了广泛采用的缺陷检测总数和唯一缺陷数到来评估 CCOFT 的有效性，以确保评估结果的可靠性和可比性。对于实验中用到的参数，如构造测试程序时对递归深度的设置，作者计划在未来工作中增加对比实验对其影响进行深入研究讨论。

3.7 本章小结

本章提出了一种面向编译器前端测试的结构感知程序构造方法 CCOFT，构造出了语法多样化的测试程序，有效检测出了深层次的编译器前端缺陷。CCOFT 首先根据现有的语法定义将其转换为灵活的可供突变的结构化格式，该格式可以自动化地进行多粒度变异操作，如细粒度的位反转变异和粗粒度的结构化交叉变异等。然后，CCOFT 采用了等概率机会选择（ECS）策略进行结构感知语法突变，从而构造多样化的 C++ 测试程序。为了有效识别编译器前端缺陷，CCOFT 采用了一套基于差分测试的比较策略，通过比较不一致的编译器输出来识别编译器前端中的不同类型的缺陷。本章在两种广泛采用且成熟的编译器（即 GCC 和 Clang）上对 CCOFT 的有效性进行了评估。实验结果表明，在检测到的前端缺陷数量上，CCOFT 显著优于两种基准方法（即 Dharma 和 Grammarinator），提升比例达 135% 和 111%。同时，CCOFT 在检测编译器新缺陷上也表现出较好的性能。在三个月的运行时间内，CCOFT 向 GCC 和 Clang 开发者报告了共 136 个缺陷，其中 67 个已被编译器开发者确认、分配或修复，充分证明了 CCOFT 在实践中具有较强的缺陷检测能力。

4 面向编译器中后端测试的再制造生成器程序构造方法

4.1 概述

当编译器前端分析正确执行后，有效检测编译器中后端缺陷可以进一步保障编译器质量，减轻相关由编译器中后端缺陷带来的危害[96]。由于测试程序必须完全通过前端检查才能进入中后端继续优化和代码生成，构造能够有效测试编译器中后端的语义多样化的测试程序并非易事。为了提高编译器中后端的可靠性，研究者已经提出了大量的方法[2–6,10,51,61,81,89,97]用于构造能够有效测试编译器中后端的测试程序。

如图4.1所示，广泛用于编译器中后端测试的测试程序构造方法可分为两类：基于生成的构造方法（程序生成器，如 CCG[1]）和基于突变的构造方法（程序突变器，如 Hermes[5]）。值得注意的是，基于以上两类测试程序构造方法的测试流程通常从程序生成器开始。在前者中，基于生成的方法旨在设计有效的程序生成器，以直接生成揭示缺陷的测试程序（即图4.1中的技术路线❶）。在后者中，基于突变的方法通常采取两个步骤来获得测试程序。第一步是收集种子程序，该程序通常来自程序生成器。在第二步中，通过对种子程序执行不同突变（如代码片段插入/删除）操作以构造出更可能触发缺陷的测试程序。以上两个步骤遵循图4.1中的技术路线❷。基于上述程序生成器被广泛使用的事实，高质量的生成器显然是基于生成和基于突变方法的关键，即它们可以对检测编译器中的新缺陷产生重大影响。现有典型程序生成器包括 CCG、Csmith[2] 和 YARPGen[3] 等。CCG 被设计用于构造语法上有效的 C 程序，而 Csmith 和 YARPGen 的目标是构造不包含未定义和未指定行为的测试程序。上述所有程序生成器总计检测到了上千个编译器中后端缺陷，研究者们和编译器开发者期望它们能够继续构造出触发缺陷的测试程序，以持续且有效地测试编译器。

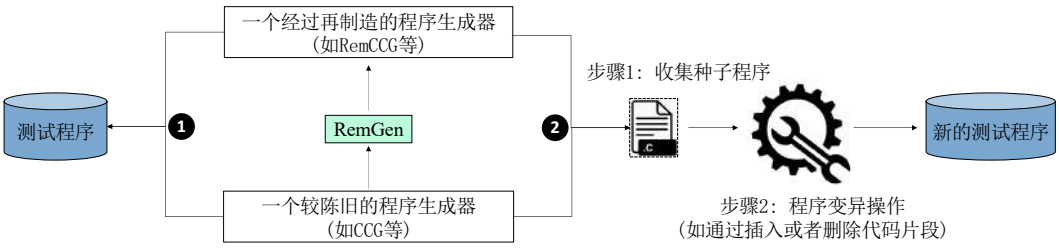


图 4.1 RemGen 的总体设计思路以及测试程序构造的两种主流方法

Fig. 4.1 The general idea of RemGen and the two prevalent test program construction approaches

然而，现有程序生成器已经很难直接检测到新的缺陷。例如，Csmith[2] 作为最著名的程序生成器之一，已有研究表明最近版本的编译器已经对 Csmith 直接构造出的

[1] https://github.com/Mrktn/ccg

测试程序产生了很强的抵抗力[5,6]。此外，许多活跃的研究员/开发者强烈地抱怨了该问题。例如，CCG（一个存在了超过十年的程序生成器）的开发者提到："*Compilers have now caught up with CCG (since it's been pretty hard to spot crashes last time I tried)*"[2]。Csmith[2] 的开发者之一 John Regehr 表示："*I hadn't run Csmith for a while and it turns out LLVM is now amazingly resistant to it, ran a million tests overnight without finding a crash or miscompilation*"[3]。此外，YARPGen 的贡献者之一 Dmitry Babokin 在评论中写道："*Same with YARPGen*"[4]。产生以上现象是合理的，因为现代编译器（如 GCC 和 LLVM）的开发者正在积极并迅速地修复此类工具报告的缺陷。换而言之，程序生成器检测出的大部分缺陷已经及时地被开发者直接或者间接修复，使编译器对此类程序生成器中的测试程序构造机制具有鲁棒性[2,4–6,10]。基于程序生成器质量下降的问题，本章旨在探究以下研究问题：能否通过提高现有程序生成器的缺陷检测能力以从根本上提高编译器中后端测试的效率？

为了探索以上研究问题，本章研究将再制造旧产品的想法（使旧产品更新的过程，详见第4.2节）迁移并应用到程序生成器中。为了有效地对程序生成器进行再制造，需要解决两个主要技术挑战，以增强现有程序生成器再次构造出尽可能触发编译器缺陷的测试程序的能力。第一个挑战是如何轻量化地合成多样化的代码片段。因为揭示缺陷（即更有可能触发缺陷）的代码片段特征不但难以获取[8,98]，而且随机程序生成器可以在几秒钟内生成大量代码片段，所以如何合成多样化的代码片段以揭示编译器缺陷并非易事。此外，现有的基于突变的方法（如 Hermes[5]）成本相对较高，因为开发者需要编写额外代码分析种子程序以收集所需的上下文信息（例如，全局或局部变量的名称和类型），并根据收集的信息维护突变后测试程序的有效性。第二个挑战是如何有效选择揭示缺陷的代码片段以构造揭示缺陷的测试程序。已有研究表明[2,4,8,99] 只有少数构造的测试程序可以触发缺陷，为了节省计算或人力资源，有必要选择更有可能触发编译器缺陷的测试程序片段。

为了解决以上挑战，本章提出面向编译器中后端的缺陷检测框架 RemGen（即 **Rem**anufacturing **Gen**erator）。总体而言，RemGen 的设计出发点有两个：1）现有程序生成器中的某些功能（即新的且有价值代码片段生成和轻量级上下文保留）可以帮助构造新的代码片段，该代码片段可能使编译器执行更深层次的代码；2）有效地利用现有程序生成器的功能并引入 "语法覆盖率" 指标来指导测试程序的构造（而不是随机构造）可以提高检测到编译器深层次代码区域缺陷的可能性。具体而言，在 RemGen 框架中，利用现有生成器中的内置能力（详见4.2节的程序生成器结构），RemGen 设计了两个新组件以分别克服两个关键挑战并达到有效再制造目的。第一个组件称为多

[2]https://github.com/Mrktn/ccg/blob/master/README

[3]https://twitter.com/johnregehr/status/1134866965028196352

[4]https://twitter.com/DmitryBabokin/status/1134907976085516290

样化代码片段合成，该组件使用语法辅助的方法合成多样化的代码片段。特别地，该组件首先在程序生成器中保留在生成过程中产生的上下文（如函数中的局部和全局变量），然后通过调用生成器中的内置函数来利用上下文合成新的代码片段，以解决第一个挑战。为了解决第二个挑战，RemGen 提出了揭示缺陷的代码片段选择组件。该组件采用语法覆盖率指标来记录合成过程中所选代码片段语法规则的使用频率。记录的覆盖率将用于量化各个合成代码片段的多样性。在该覆盖率指标的指导下，RemGen 选择揭示缺陷的代码片段来构造揭示缺陷的测试程序。

为了评估 RemGen 的有效性，本章使用 RemGen 框架将旧程序生成器 CCG 再制造为 RemCCG（具体过程详见图4.1），并充分评估了 RemCCG 对两个主流编译器（即 GCC 和 LLVM）中后端缺陷检测的能力。实验首先在较旧版本的编译器上评估了 RemCCG 在两种促进主流程序构造方法方面的能力。结果显示，与基于生成的方法（即 CCG）相比，RemCCG 在 GCC 和 LLVM 中分别比 CCG 多检测出了 16% 和 11% 的缺陷。为了与基于突变的方法（即 Hermes[5]）进行比较，使用 CCG 和 RemCCG 生成种子程序，并执行相同的突变操作来构造用于编译器测试的测试程序。结果表明，由 RemCCG 构造的种子程序可以帮助 Hermes 分别在 GCC 和 LLVM 中比 CCG 多检测出 14% 和 11% 的缺陷。最后，在 GCC 和 LLVM 开发版本上运行 RemCCG 的结果显示，RemCCG 共检测到了 56 个编译器缺陷（其中 37 个已经被开发者修复）。值得注意的是，大部分由 RemCCG 报告的缺陷均为严重且长期潜伏的，其中 5 个缺陷被标记为最高严重等级，2 个缺陷分别潜伏达 1 年和 3 年之久。

4.2 背景和动机

本节首先介绍程序生成器和再制造的相关背景，然后介绍 RemGen 的概念，最后使用一个缺陷示例来说明和突出 RemGen 再制造后新程序生成器的优势。

（1）程序生成器的基本结构

现有程序生成器如 CCG、Csmith[2] 和 YARPGen[3] 等允许用户构造全新的测试程序对进行编译器测试。如图4.2所示，程序生成器遵循类似的工作流程。通常，程序生成器从初始化种子数（❶）开始。在给定的种子随机数下，确定性地构造测试程序，即在执行具有相同种子号的程序生成器时，构造出的测试程序是相同的。从技术角度而言，为了构造出一个测试程序，程序生成器首先创建一个全局上下文（❷），其中包含了特定作用域中的基本属性（如全局作用域中的变量名）。接下来，程序生成器根据当前上下文构造函数。在函数构造内部，使用局部上下文（❸）和局部定义的变量构造函数块。在此期间，一些语句也在模块（❹）内生成。所有函数构造完后，程序生成器将输出新的测试程序（❺）。用户通常将程序生成器的标准输出重定向到源文件，并将此文件用于编译器测试。

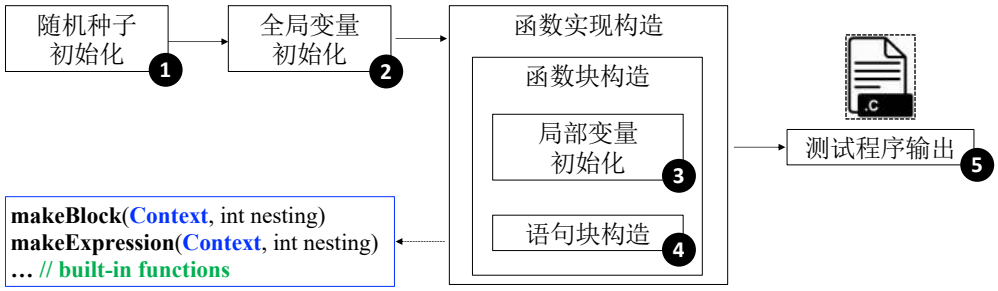


图 4.2　程序生成器的基本工作流程

Fig. 4.2　The basic workflow of a program generator

值得注意的是，程序生成器至少有两个重要且有用的功能尚未得到充分研究。首先，程序生成器支持各种内置函数（如 makeBlock 和 makeExpression 等）用于生成新的且有价值的代码片段。通常，通过多次调用该类函数，可以方便地生成各种代码片段。其次，程序生成器中保存了用于生成代码片段的上下文（即内置函数如 makeBlock 中使用的参数），调用该参数可以以轻量地方式生成代码。具体而言，用户可以首先直接保留上下文并调用不同的内置函数来生成所需的代码片段，然后使用它们来合成新的测试程序。由于重新开发此类工具的难度较大（如 Csmith[2] 的实现超过了两年），研究者应该有效激发在生成器中隐藏的各种功能，以便不断发挥其潜在的测试程序构造能力，从而检测出新的编译器缺陷。

（2）再制造的典型过程

再制造是将二手产品重制为“几乎全新”产品的过程，旨在使旧产品再次有效。因为再制造过程能够比传统制造更加有效且对环境的危害更小[100,101]，该过程被视为一种可持续的制造模式。再制造已成功应用于许多领域，例如汽车零件，航空航天和医疗设备[101–103]。从再制造功能的角度出发，再制造大致可以分为以下三个过程：

① 再制造准备。该过程的任务是勘察和评估旧产品进行再制造的可行性，包括检查、拆卸和预处理旧产品的子部件。

② 再制造。该过程侧重于首先设计新组件并将其引入旧产品，然后将新组件和拆卸后的零件重新组装在一起，最终制成一个“接近新”的产品。

③ 测试再制造产品。该过程测试经过再制造后产品的有效性。

（3）RemGen 的主要设计思路与缺陷示例

在本章研究中，RemGen 主要将再制造旧产品（如旧机械部件）的想法迁移至程序生成器，以提高编译器中后端测试效率。实现 RemGen 的具体思路是，通过有效利用现有程序生成器中隐藏的高效复用功能，能够增强现有程序生成器的中后端缺陷检测能力。具体而言，旧程序生成器经过 RemGen 的再制造过程后，新程序生成器至少能够具有以下三个优势。第一，通过直接从程序生成器保留上下文，不需要和现有方法（如 Hermes[5]）一样编写额外功能代码来分析和提取种子程序的上下文信息（详见

第4.3.3节）。第二，因为新旧代码片段的上下文语义相同，在进行代码合成时，不需要编写其他代码维护代码的语法有效性。第三，由于现有程序生成器被证明对编译器测试是有效的，内置函数生成的代码片段对构造可以揭示新编译器缺陷的测试程序十分有价值。得益于以上三个优势，RemGen 将第4.2节中提到的三个传统再制造过程应用于程序生成器的再制造中，具体实施过程如下：

① 准备过程。该过程首先检查生成器的源代码是否可用，以确定其是否能够在 RemGen 的框架中进行再制造。然后，如果通过了初步检查，RemGen 则根据第4.2节中提到的不同内置函数划分为子部分（如 makeBlock 等）来分解生成器。最后，RemGen 对分解后的子部件进行预处理，使它们易于与其他部件（如新设计的部件）集成。

② 再制造过程。RemGen 在再制造过程中为程序生成器设计了两个新组件，即多样化代码片段合成组件和揭示缺陷代码片段选择组件，从而能够将旧生成器再制造成新生成器。特别地，该过程为再制造的主要过程，后续将主要介绍该部分。

③ 测试过程。该过程测试经过再制造程序生成器的缺陷检测能力。

总而言之，给定一个旧程序生成器（如 CCG），RemGen 对其再制造，经过再制造后的程序生成器（如 RemCCG）可以再次生成揭示缺陷的测试程序。为了进一步说明本章的研究动机，下文使用图4.3中的示例来说明现有方法的局限性以及 RemCCG 在生成揭示编译器缺陷测试程序方面的优势。

（2）动机缺陷示例

图4.3展示了由 RemCCG 构造的揭示缺陷的测试程序，该程序触发了 LLVM 编译器的超时缺陷。该缺陷使 LLVM-13 编译器在采用 “-O3” 优化选项编译时超时，而由不包含该缺陷的编译器或编译选项编译时正常退出。为了便于读者理解，在本章中仅显示该测试程序的简化版本。特别地，由 RemCCG 合成的揭示缺陷代码片段位于第 4-15 行之间。值得注意的是，以灰色突出显示的第 9-14 行代码片段是通过调用程序生成器中的内置函数（即 makeBlock）生成的。

① 缺陷产生的根本原因。触发该缺陷的根本原因是 LLVM 编译器在执行 “-loop-unrolling” 优化（现代编译器中一种重要且广泛应用的循环优化技术[5]）时非法采用了未绑定的 “llvm.assume” 指令，而 “llvm.assume” 指令允许优化程序假定提供的条件是真实的。在此情况下，LLVM 在第 11 行的 if 分支上错误地进行了无休止的 “llvm.assume” 指令操作，从而导致该编译器编译时间爆炸问题。在作者提交该缺陷后，LLVM 开发者在 48 小时内及时修复了该缺陷。

② 现有方法的局限性。由于该缺陷在删除合成的代码片段（第 4-6 行之间）后消失，该揭示缺陷的测试程序很难通过基于生成的测试程序构造方法（如 CCG）生成。此外，现有基于突变的测试程序构造方法（如 Orion[4]、Athena[6] 或 Hermes[5]）也存

[5]https://en.wikipedia.org/wiki/Loop_unrolling

```
1  int a, b, c, d;
2  void e() {
3        ...// code snippets generated by CCG
4       a = 7;
5       for (; a <= 78; a++) {
6            d = 3;
7            for (; d <= 73; d++) {
8                 // code produced by makeBlock() highlighted in gray
9                 int f = 0;
10                b += c;
11                if (b) {
12                     int g = 0;
13                     for (f = 5; f; g);
14                }
15           }
16      }
17 } /* Grammar Coverage : G={0,0,0,2,0,0,0,0,0,0,0} */
```

图 4.3　LLVM 13.0 在采用 “-O3” 优化选项编译时超时示例 (#49171)

Fig. 4.3　LLVM trunk 13.0 hangs at compile-time with -O3 (LLVM#49171)

在构造此类代码片段的局限性，具体原因如下。首先，Orion 和 Athena 只能修改位于死代码区域的代码，而示例中揭示缺陷的代码片段位于活代码区。其次，尽管 Hermes 能够在活代码区域中插入代码片段，但 Hermes 合成的此类代码片段受到多样性的限制。例如，第 9 行中的局部变量 f 无法在 Hermes 中合成。更糟糕的是，Hermes 完成合成代码片段的时间成本相对较高，进一步降低了构造揭示缺陷测试程序的效率。

③ RemCCG 方法的优势。与现有方法相比，RemCCG 可以有效地构造出以上揭示缺陷的代码片段。RemCCG 是 CCG 经过 RemGen 再制造后的新程序生成器。RemGen 是本章提出的再制造程序生成器的框架，其通过将两个新组件（即多样化的代码片段合成和揭示缺陷的代码片段选择）附加到程序生成器的现有工作流程中以完成再制造程序生成器过程。具体而言，在代码片段合成过程中，RemGen 首先保留（而不是从种子程序中提取）全局和局部上下文，包括程序生成器中所有可能变量，如图4.3中的变量 a、b、c、d 和 f。然后，RemGen 调用代码片段合成组件以生成两个 for 循环语句，以及通过采用保留上下文执行 makeBlock 生成的代码片段。最后，整个代码片段被整合到第 3 行后，从而构造出了该揭示缺陷的测试程序。值得注意的是，在构造过程中，合成的代码片段数量可能较大。为了提高构造有效测试程序的效率，RemGen 利用揭示缺陷代码片段选择组件来选择最具多样化的代码片段来构造新测试程序。具体而言，RemGen 引入了语法覆盖率来区分不同代码片段并为后续评估其多样性做准备。例如，在图4.3中，RemGen 为该段代码维护了一个记录列表（即 {0,0,0,2,0,0,0,0,0,0,0}），其中各个维度代表一个特定的语法规则，各个维度值表示语法规则在代码片段中出现的频率。其中，记录列表中的数值 2 表示该代码片段中出现了两个 for 循环语法语句（排除内置函数合成的代码段后）。此外，RemGen 会在合成过程中不断更新该列表，并将

其用来指导选择揭示缺陷的代码片段。

需要指出的是，现有程序生成器在理论上仍然可以构造出与 RemCCG 合成的代码片段类似的代码片段。但是，只有当该程序生成器作一定修改后才有可能生成类似代码。具体而言，Csmith[2] 和 YARPGen[3] 旨在构造没有未定义行为的测试程序。因为以上需求是一个相对严格的要求，该策略会减少测试程序的多样性。CCG 可能可以构造出类似于 RemGen 构造出的测试程序，但由于构造其测试程序的随机性太大，在实践中同样很难构造出与图4.3一样能够触发编译器缺陷的测试程序。相比之下，经过 RemGen 再制造后的 RemCCG 旨在构造出语义多样化（即语法有效但可能包含未定义行为）的测试程序，可以作为 Csmith[2] 和 YARPGen[3] 较好的补充。

4.3 RemGen 框架描述

本节首先介绍 RemGen 的设计框架概述，接下来详细介绍该框架具体的实施步骤（即准备、再制造和测试过程）。

4.3.1 RemGen 框架概述

RemGen 的整体框架如图4.4所示。RemGen 将一个旧程序生成器作为输入，输出一个新程序生成器，最终采用新程序生成器构造的测试程序检测潜在的编译器缺陷。具体而言，RemGen 的主要工作流程可以分为三个过程：

1）准备过程。该过程研究输入程序生成器用于再制造的可行性的准备过程，包括检查、拆卸和预处理拆卸后的程序生成器子组件。

2）再制造过程。该过程首先将两个新组件（即❻代码片段合成组件和❼代码片段选择组件）整合到输入程序生成器的原始工作流程中（如图4.2所示），然后将所有组件重新组装为新再制造生成器。

3）测试过程。该过程用于评估经再制造后新程序生成器的有效性，即采用基于生成或基于突变的方法对编译器进行测试，并查看是否能检测出新编译器缺陷。

接下来的小节将详细描述以上各个过程。

4.3.2 再制造之预处理过程

准备工作的任务是评估输入程序生成器在再制造方面的可行性。通常，RemGen 的用户通过手动构建和运行旧程序生成器以评估其能够再制造的可能性。此外，用户执行生成器主要功能的代码审查，以确认该生成器是否具有与图4.2中所示相同的工作流程。值得注意的是，与现有程序生成器（如 CCG、Csmith[2] 和 YARPGen[3] 等）长时期和劳动力密集的开发过程相比，RemGen 中的评估不需要消耗太多的人力。此外，大多数程序生成器均维护相关文档，用户可以方便地查阅文档以完成评估工作。具体而言，如图4.4所示，RemGen 的用户需要执行以下步骤完成准备工作。

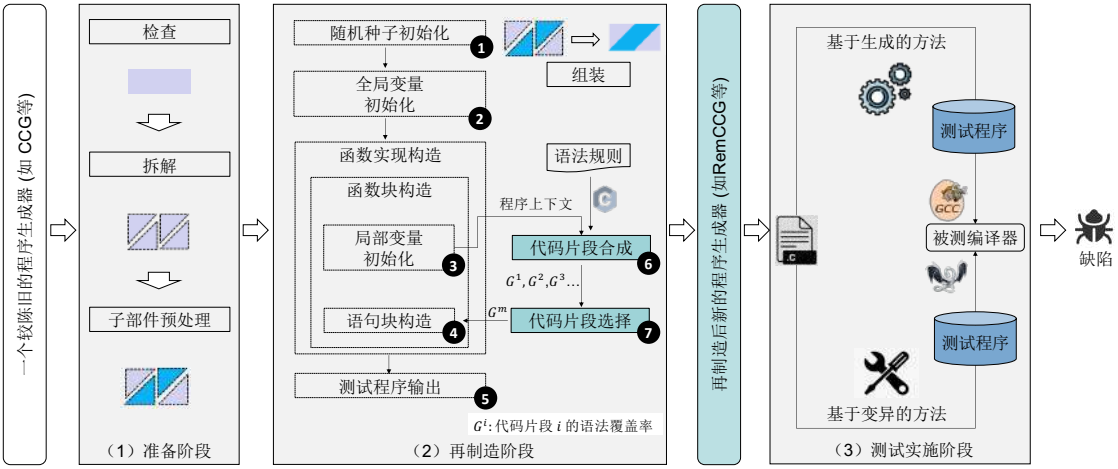


图 4.4　RemGen 总体设计框架

Fig. 4.4　The high-level architecture of RemGen

1）检查。从程序生成器的“外观”检查其功能，例如检查程序生成器的代码是否公开以及是否能够正常编译和运行。

2）拆解。将程序生成器的生成组件进行解构，例如分离程序生成器实现中不同表达式/语句/块构建的逻辑（如图4.2所示）。

3）预处理子部分。重建部分拆解后的所需组件（如程序生成器中的内置函数），以便与其他组件集成。例如，收集上下文（全局或局部）并将其作为外部变量，以供其在新组件中被顺利调用。

在预处理过程之后，两个内置函数 makeBlock 和 makeExpression 已经为后续的再制造过程做好准备。因为该类内置函数可以在旧生成器中生成不同类型的支持语句/表达式，所以用户可以通过调用内置函数来合成不同的代码片段。

4.3.3　再制造之制造过程

本小节首先介绍再制造过程的基本工作流程，然后详细介绍 RemGen 的两个主要组件，即多样化代码片段合成组件和揭示缺陷代码片段选择组件。

如图4.4所示，模块❶、❷、❸、❹ 和 ❺与图4.2程序生成器中的基本模块相同，即均为从程序生成器中拆解的子组件。除了程序生成器中已有的组件外，RemGen 还设计了两个新组件（即代码片段合成组件❻和代码片段选择组件❼）用于合成不同的代码片段并选择最多样化的代码片段来构建新缺陷揭示测试程序，以分别解决第4.1节提到的两个挑战。完成到两个新组件的设计后，所有拆卸的子组件均重新组装到再制造的程序生成器中，以对编译器进行测试。下节将详述两个新组件的设计。

（1）多样化代码片段组件

该组件旨在通过轻量化的方式合成多样化的代码片段。该组件将所需的上下文和一组语法作为输入，并输出一组不同的代码片段。本小节首先给出了“多样性”的度

量定义，即代码片段的语法覆盖率，然后说明用于减少合成开销的上下文保留策略。后续小节将详细介绍合成过程。

① 语法覆盖率的定义。RemGen 的核心实现思路是第一步从现有程序生成器中调用内置函数生成的代码片段中合成新代码片段，第二步将新代码片段整合到完整的测试程序中。由于第一步合成的代码片段数量可能较大，因此有必要采用量化指标来区分不同的代码片段，以便选择最具多样化的代码片段来构建新测试程序。为了定量和有效地衡量新代码片段的多样性，RemGen 采用了已有研究中相关度量的指标，即语法覆盖率。换而言之，RemGen 主要利用语法覆盖率指标来量化生成程序片段的复杂性/表现力。由于该指标遵循了现有研究[2,3] 的基本假设，即复杂或者语言特征较多的测试程序更有可能在编译器中执行深层代码区域，从而可能触发更多更深层次的编译器缺陷，所以在 RemGen 中采用该指标是合理的。具体而言，RemGen 定义了如下用于区分各种代码片段的语法覆盖率计算公式：

$$G(\mathtt{CS}) = (G_1, G_2, \ldots, G_i, \ldots) \tag{4.1}$$

在上述公式中，G 是一个记录列表，表示代码合成过程中语法规则出现的频率。具体而言，该记录列表包含一组语法规则 G_i，其中 i 代表特定的语法规则。例如，图4.3中的 G 是 “{0,0,0,2,0,0,0,0,0,0,0}”，表明该合成的代码片段中只使用了两个 for 循环语句。总体而言，该度量方法可用于定量地表示各种代码片段，其中具有更大语法覆盖率的代码片段意味着其具有更复杂的结构，更可能检测到深层次编译器缺陷。

② 上下文保留策略。在合成新代码片段之前，需要收集合成程序时所需的代码上下文（即 context）。通常，根据程序语义的不同范围，需要考虑两种上下文。一个是局部区域的上下文（即在函数实现内部）；另一个是全局区域中的上下文（即函数实现之外）。通常，全局上下文（如全局变量）可以同时在全局和局部区域合法使用。但是，因为在定义范围之外使用某个局部变量可能会引入 “未定义的变量” 编译错误，所以局部上下文只能在局部区域中使用。现有策略[5,6] 通常采用编写额外代码的方式完成提取/分析源代码以收集代码片段的上下文。然而，该策略比较耗时，严重影响了编译器测试的效率。例如，先前的研究[5] 表明，基于突变的方法合成新测试程序平均需要 1.7s（最多 6s）的分析，而生成种子程序通常不会超过 1s。

为了降低收集上下文的开销，与现有方法不同，RemGen 以更轻量的方式从输入程序生成器中保留所需的上下文。具体而言，如图4.4所示，由于已将输入的程序生成器分解为不同的子组件，因此 RemGen 可以轻松地从输入程序生成器获取和保留全局或局部上下文。在此过程中，需要考虑的一个小问题是需要根据不同的范围（即全局或局部）区分上下文的不同用途。例如，如果将使用全局上下文生成代码片段集成到其他函数中，则不会出现未定义变量的编程错误。但是，由于本地上下文只能在局部

区域使用，RemGen 维护了一个局部上下文列表并确保只在局部区域使用该保留的上下文。总而言之，在合成新代码片段之前，RemGen 通过保留全局和本地上下文并调用程序生成器的内置函数构造出多样化的测试程序。特别地，RemGen 的上下文保留策略有两个优势。首先，因为该代码片段是在同一语义上下文下合成的，该策略可以自然地保证构造出的代码片段的有效性，其次，因为不需要分析原始测试程序并编写额外的代码来提取所需的上下文，所以收集的过程是轻量级的。在准备好原始代码片段的上下文信息之后，RemGen 开始合成不同的代码片段。

Algorithm 4.4 代码片段合成算法

Input: C: 程序上下文; $depth$: 合成深度; T: 一组语法定义; N: 代码片段数量界限值
Output: 一组代码片段 CS (每段片段包含语法覆盖率信息 G)

1: $i = 0$; // 初始化计数变量
2: **function** `synForStmt`(C, T, N)
3: $G[11] \leftarrow \emptyset$; //初始化一个覆盖率向量
4: $depth = 0$; // 初始化深度变量
5: **while** $i < N$ **do**
6: $CS_i(G^i) =$ `synStmtSeq`$(C, depth, T)$;
7: **end while**
8: **end function**
9: **function** `synStmtSeq`($C, depth, T$)
10: $i = i + 1$;
11: **if** random(0,1) **then**
12: $CS(G^i) =$ `synStmt`$(cc, depth, T)$;
13: **else**
14: $CS(G^i) =$ `synStmtSeq`$(cc, depth, T)$;
15: **end if**
16: **return** $CS(G^i)$;
17: **end function**
18: **function** `synStmt`($C, depth, T$)
19: **if** $depth > max\_depth$ **then**
20: **return** $CS(G^i)$;
21: **end if**
22: **switch** $T.type$ **do**
23: **case** AssignStmt
24: `synAssignStmt`(cc); $G^i[0]$++;
25: **case** WhileStmt
26: `synWhileStmt`(cc); $G^i[1]$++;
27: ... // 处理其他类型的语句
28: **case** JumpStmt
29: `synJumpStmt`(cc); $G^i[1]$++;
30: **return** $CS(G^i)$;
31: **end function**

③ 代码片段合成。为了有效合成代码片段，一个简单的解决方案是通过调用程序生成器中的内置函数并将代码片段整合（即以原有生成器的工作方式）到测试程序中来直接合成新代码片段。但是，该方案对构造揭示缺陷的测试程序存在一定的局限性（详见第4.6节讨论部分）。因此，本章提出了一种语法辅助合成策略来缓解相关局限性。具体而言，RemGen 的代码片段合成中采用了一组语法（本章研究为 C 语法）对合成代码的过程进行了记录，并通过保留的上下文自上而下构造新代码片段。算法4.4给出了代码片段合成的详细过程。该算法采用上下文 C、合成深度 $depth$、一组语法 T 和边界值 N 作为输入，输出一组代码片段 CS 以及各个代码片段的更新语法覆盖率 G。边界值 N 用于限制代码段的数量。具体而言，该过程由函数 synCode 实现。synCode 首先初始化语法覆盖 G 和深度计数器深度 $depth$（第3-4行）。然后，该函数通过采用上下文、深度计数器和语法的参数调用函数 synStmtSeq 来合成新语句序列（第6行）。当达到预设的边界值时，代码合成部分完成。通过以上合成操作，一组代码片段 CS 将作为新揭示测试程序的候选代码片段。

在具体的合成过程中，算法中的 synStmtSeq 函数要么通过递归调用函数自身来合成一组语句，要么通过调用函数 synStmt 来合成单个语句（算法4.4中第9-17行）。在具体的调用过程中，RemGen 随机决定所构造语句的顺序，该随机策略也是现有研究中建议的策略[5,6]。函数 synStmt 的实现逻辑主要分为两部分（第18-31行）。首先，该函数检查当前深度是否大于定义的 max_depth（第19-20行）。其次，如果检查失败，则在 T 的帮助下合成一个随机语句并更新相应的语法覆盖率（第22-29行）。值得注意的是，现有的合成策略构造出的代码片段多样性具有一定的局限性。具体而言，现有方法[5]只能合成一小部分语法结构受限的语句（如 Always False Conditional Block（FCB）、Always True Guard（TG）和 Always True Conditional Block（TCB））。接下来，该算法应用 C 中所有定义的语句级语法规则来帮助合成多样化的代码片段，并维护相应的语法覆盖率列表以备后续的代码片段选择组件采用。特别地，由于 CCG 不支持一些语句（即 switch、while 和 do-while），本文对此进行了额外评估，结果发现 RemGen 改进的缺陷检测能力主要来自新提出的组件，而不是新支持的语句（详见第4.6节）。由于合成不同类型语句的过程相似，为了精简表达起见，下文只详细展示合成 for 循环语句，其他类型语句的合成采用类似的操作。

算法4.5展示了 for-loop 语句的合成算法。值得注意的是，RemGen 并没有完全遵循 C 语法中预定义语句的语义，而是根据作者经验并基于以下两个指导原则手动构建语句结构：1）因为活/死代码的合成[4–6] 已被证明在编译器中有效地检测深层次的缺陷，合成时试图合成更大的代码块来作为活代码或死代码（即分别在执行代码时可以或不可以被覆盖）区域；2）尽量重用收集到的上下文中尽可能多的信息（如 for 条件中的变量），以增加代码片段的多样性。基于以上两个原则，算法4.5中的 for-loop

Algorithm 4.5 合成 for-loop 语句算法

Input: C: 程序上下文, $depth$: 合成深度, T: 一组语法定义
Output: 一个 for-loop 代码片段

```
1:  function synForStmt(C, depth)
2:      cmp ← {<, >, =, >=, <=}; // 大于或等于比较运算符
3:      op ← {++, + =, −−, − =}; // 自加/自减运算符
4:      depth + = 1;
5:      var = randomVar (C); // 从程序上下文 C 中选择一个变量
6:      buildPredicate (var, cmp, op, randomNumber)
7:      out « "{" // 开始合成 for-loop 语句块
8:      makeBlock (C, nesting)
9:      CS_for = synStmtSeq (C, depth, T)
10:     out « "}" // 结束合成 for-loop 语句块
11: end function
```

语句的合成主要包括两个部分，即谓词构建（第6行）和主体构造（第8-9行）。

对于谓词构建，RemGen 采用不同的方法分别合成真和假的谓词。根据构建谓词条件的真/假，可以合成一个语句作为活动代码或死代码。为了使 for-loop 语句位于不同代码区域，RemGen 首先预先初始化两种运算符，即比较运算符和增量/减量运算符。然后，通过 randomVar 函数从现有上下文 C 中随机选择一个变量。接下来，RemGen 为 for-loop 条件合成一个条件为真的谓词。具体而言，按照以下两个指导原则构造一个预测为真的谓词：1）选择一个适当的 randomNumber 和 cmp，使得 “var cmp randomNumber” 为真，以及 2）选择一个适当的 op；例如，如果选择比较运算符 “<” 或 “<=”，稍后选择的相应运算符应该是 “++” 或 “+=”。对于合成假谓词，直接产生一个假条件（如 0）作为谓词。对于合成真或假谓词的概率，RemGen 设定了相同的数值（各占 50%）。在合成主体时（第7-10行），首先调用输入程序生成器中的内置函数（即 makeBlock）来构建一个函数块，然后调用 synStmtSeq 函数在主体中合成语句序列。在合成过程中，RemGen 采用输入程序生成器中的默认参数 nesting 和边界值（由算法4.4中的参数 N 控制）来控制程序大小和递归次数。

综述所述，上述合成过程可以产生大量的多样化代码片段，其中任何一个均可能被选择并整合到揭示缺陷的测试程序中。为了提高揭示缺陷的测试程序构造的效率，需要考虑一个可行的方法来选择揭示缺陷的代码片段，以构造出新的测试程序。下文将介绍 RemGen 设计的代码片段选择组件。

（2）揭示缺陷代码片段选择组件

本小节描述 RemGen 的针对上述选择问题的具体解决方案，即利用第4.3.3节中引入的语法覆盖率来选择揭示缺陷的代码片段。考虑到大量的代码片段候选及其各个候选的独特语法覆盖率，RemGen 计算各个候选的语法覆盖率的平方和，其中更大的值代表代码片段中的更高多样性。选择这个指标的原因是已有研究[104] 表明，测量候选

语法覆盖率和坐标原点之间的欧几里得距离可以有效地区分测试程序，所以 RemGen 中的选择策略是合理的。接下来，RemGen 将上述计算结果应用于对各种代码片段进行排序。在具体选择的过程中，RemGen 选择具有最大数值的多样化代码片段作为揭示缺陷的代码片段。在选择代码片段后，RemGen 下一步则将其整合到测试程序中的合适位置。理论上，代码片段可以整合到图4.4中所示的本地上下文初始化（即❸）后任何位置。在本章研究中，RemGen 考虑将新的代码片段插入在函数体中构造语句的位置，即构造函数体内最后一个语句（即❹）之后的位置。由于新设计的合成代码片段组件与现有程序生成器的现有代码片段之间有更清晰的界限，评估 RemGen 各部分的有效性是可行的。综上所述，输入一个旧程序生成器，RemGen 采取以上设计再制造该程序生成器并输出新程序生成器，新程序生成器可以采用基于生成或者变异的程序构造方法构造新测试程序以有效测试编译器。

4.3.4 再制造之测试过程

如图4.4所示，经过 RemGen 再制造给定的程序生成器（如 CCG）后，会产生新程序生成器（如 RemCCG），用户期望新程序生成器能够再次构造出揭示缺陷的测试程序。为此，后续章节设计了详细的实验对新程序生成器在缺陷检测能力方面的有效性进行评估。具体而言，给定新程序生成器，基于生成或基于突变的方法均可以被重新应用于测试编译器，并依次评估新程序生成器对以上两种方法的增强效果。下一节将详细介绍和分析评估的具体过程和结果。

4.4 实验设计

本章详细介绍用于评估 RemGen 有效性的实验设计方案，包括数据集与实验平台、研究问题、方法实现与实验设置、实验结果分析以及相关讨论。

4.4.1 数据集与实验平台

① 数据集及研究对象。本实验所用的数据集是由 RemCCG 及对比方法构造出的测试程序。关于研究对象，和现有的编译器测试研究[4,5,28,42] 类似，在本章中选择 GCC 和 Clang 两个主流的编译器作为研究对象。具体而言，对于测试对象和运行选项，本章选择了两个编译器的五个标准优化选项进行差分测试：“-O0”、“-O1”、“-Os”、“-O2”和 “-O3”。选择以上设置是因为现有的编译器测试研究工作[2,4–6,61] 均采用该实验配置检测编译器中后端缺陷，所以本实验同样选择以上常用选项。

② 实验平台。本章所有的实验均在 Ubuntu 18.04 服务器上运行，该服务器配备了 Intel(R) Core(TM) i7-6900K CPU @ 3.20GHz × 16 处理器和 64GB RAM。

4.4.2 研究问题

为了充分评估 RemGen 的有效性，本章实验部分将探讨以下三个研究问题（Research Questions，简称 RQs）：

① RQ1：RemCCG 是否能够提升基于生成和基于变异的编译器测试方法的效果？

RQ1 采用旧版本编译器来评估 RemCCG 在增强基于生成和基于的测试程序构造方法的效果，通过比较不同方法检测到的缺陷数量来比较不同方法的缺陷检测能力。

② RQ2：RemCCG 中新增加的组件是否有效？

RQ2 采用旧版本编译器来评估两个新组件是否能够提高 RemCCG 的缺陷检测能力，通过比较不同变体方法检测到的缺陷数量来比较不同方法的缺陷检测能力。

③ RQ3：RemCCG 是否能够在实践中检测到新的编译器缺陷？

RQ3 将 RemCCG 运行在主干开发版本的编译器上以评估 RemCCG 在实践中缺陷检测能力，即实用价值。

4.4.3 方法实现与实验设置

（1）RemCCG 实现

为了评估 RemGen 的有效性，本章通过一个案例研究来说明 RemGen 的有效性。具体而言，本章采用 RemGen 对 CCG 进行了再制造，最终得到一个新程序生成器 RemCCG。实验中通过评估 RemCCG 的有效性来说明 RemGen 的有效性。实验中选择 CCG 的主要原因是 CCG 存在的时间已经足够长，且在很长一段时间内没有更新（在 2016 年之后没有更新）。本文的主要出发点是，如果一个老旧程序生成器经过再制造后可以找到新的编译器缺陷，说明被 RemGen 再制造后的 RemCCG 在实践中具有很好的缺陷检测能力，并且很大概率能够提升最先进的基于生成和基于突变的方法，以上事实将作为说明 RemGen 有效性的有利证据。然而，由于 CCG 自身功能的有限性（详见第4.6节相关讨论），本章目标是在本研究中检测编译器中的崩溃或性能缺陷。为了实现代码合成策略，RemCCG 结合了 grammar-v4 中的一部分 C 语法（其中包括各种 ANTLR-v4[86] 语法。另外，RemCCG 还采用一种在转换过程中可以灵活操作的实用格式 ProtoBuffer，该格式可以将 C 语法顺利转换为实际 C 程序的中间表示。对于合成阶段中代码片段界限值设置，用户可以根据不同的需求配置不同的值，本实验中将界限设置为 10（关于边界值的选择讨论详见第4.6节）。

（2）RQ1 的实验配置

① RQ1 的对比方法。为了评估 RemCCG 在增强基于生成方法的编译器测试方面的有效性，实验中将 RemCCG 与 CCG 进行了比较。为了评估 RemCCG 在增强基于突变方法的编译器测试方面的有效性，实验中选择一种基于变异的方法作为研究对象。因为 Hermes[5] 已被证明是优于其他现有方法（如 Orion[4] 和 Athena[6]）的最先进方

法，所以实验中选择 Hermes 作为基准对比方法。参照现有工作[81,89]，所有对比方法在两个旧版本的主流编译器上（即 GCC-4.4.3 和 LLVM-2.6）运行。

② RQ1 中的方法运行策略。为了实现公平比较，在相同的测试周期内（即 90 小时）首先对每种对比方法重复运行各 10 次（与现有工作[10,15] 的设置相同），然后计算各个方法检测到的缺陷数量平均数。

（3）RQ2 的实验配置

① RQ2 的对比方法。为了评估两个提出的组件的有效性，实验将 RemCCG 与几个变体进行了比较，包括 RemCCG(¬G)（采用程序生成器中的内置函数进行随机合成，而不是采用基于语法的合成）、RemCCG(¬S)（没有选择策略）和 RemCCG(S_R)（采用随机选择而不是基于语法覆盖引导的选择）。通过将 RemCCG 与 RemCCG(¬G)对比，可以评估多样化代码片段合成组件的有效性。通过将 RemCCG 与 RemCCG(¬S)和 RemCCG(S_R) 进行对比，可以评估揭示缺陷代码选择组件的有效性。

② RQ2 中的方法运行策略。在本研究问题中以 CCG 作为基准方法，和 RQ1 一样在两个编译器（即 GCC-4.4.3 和 LLVM-2.6）上运行各种变体方法，运行时间为 24 小时。为了进一步减小误差，实验重复进行了 5 次，最后统计检测到的缺陷平均数量。

（4）RQ3 的实验配置。

为评估 RemCCG 在实践中的缺陷检测能力，本文在 2021 年中旬至 9 月底的非连续时期内持续测试了每日更新的 GCC 和 LLVM 的开发主干版本。因为编译器开发者通常比已发布版本更迅速地修复开发版本中的缺陷[4–6,10]，所以在 RQ3 中检测编译器主干版本上的缺陷。具体地，RQ3 从三个方面评估 RemCCG 的缺陷检测能力，即检测到的缺陷数量、修复的缺陷类型及其重要性分析。

4.4.4 评价指标

本节主要采用检测到的缺陷数量衡量 RemCCG 的有效性。具体而言，RQ1 从两个方面分析了 RemCCG 与两种先进方法在缺陷检测能力上的对比，即检测到的缺陷总数和唯一缺陷的数量。RQ2 主要采用检测到的缺陷总数对不同的对比方法进行评估。RQ3 除了对比检测到的缺陷数量，还从不同的角度进一步分析了各个缺陷的具体情况，包括其优先级、报告状态、缺陷类型、测试策略及影响的编译器版本。

4.5 实验结果分析

4.5.1 与现有方法对比分析

（1）与现有基于生成方法对比结果分析

表4.1列举了 RemCCG 与基于生成程序构造方法的比较结果。表中的第一列表示四个对比方法。接下来的四列是每种方法检测到的缺陷的平均统计，包括计算崩溃缺

陷（Crash）、性能缺陷（Performance）、检测到的缺陷总数（Sum）和检测到的缺陷改进（Improvement）。此处，Improvement 是 RemCCG 相对于比较方法的相对改进。如表4.1最后一列所示，可以观察到，在两个编译器 GCC 和 LLVM 中，RemCCG 检测到的缺陷数量比 CCG 分别多 16% 和 11%。

表 4.1　促进基于生成的构造方法的结果
Tab. 4.1　Results of boosting in generation-based approach

编译器	对比方法	Average Statistics			
		Crash	*Performance*	*Sum*	*Improvement*
GCC	CCG	2.9	0.3	3.2	16%
	RemCCG	3.1	0.6	3.7	-
LLVM	CCG	9.2	2.7	11.9	11%
	RemCCG	9.7	3.5	13.2	-

（2）与现有基于变异方法对比结果分析

为了评估 RemCCG 在促进基于突变的测试程序构造方法（即 Hermes[5]）方面的有效性，本实验中采用基准方法 CCG 和再制造的 RemCCG 作为程序生成器来构造种子程序，然后采用 Hermes 中的变异策略构造新测试程序。遵循 Hermes 中的突变策略，对种子程序进行突变（即插入 FCB，TG 和 TCB），以构造用于编译器测试的测试程序。表4.2显示了 RemCCG 与以上两种方法的实验对比结果。每行每列的含义和表4.1相同。表格中的数据结果清楚地表明，在编译器 GCC 和 LLVM 中，采用 RemCCG 的 Hermes 能够检测到的缺陷数量比采用 CCG 的 Hermes 分别多 14% 和 11%。

表 4.2　促进基于变异的构造方法的结果
Tab. 4.2　Results of boosting in mutation-based approach

编译器	对比方法	Average Statistics			
		Crash	*Performance*	*Sum*	*Improvement*
GCC	Hermes(CCG)	3.0	0.5	3.5	14%
	Hermes(RemCCG)	3.2	0.8	4.0	-
LLVM	Hermes(CCG)	9.8	3.6	13.4	11%
	Hermes(RemCCG)	10.6	4.3	14.9	-

为了进一步确认 RemCCG 方法在统计学上的有效性，对于表4.1和表4.2中的所有结果，本文进行了显著性水平为 0.05 的 Mann-Whitney U 检验[90]，其中零假设为“RemCCG 和比较方法之间没有显著差异”，备择假设为 “RemCCG 和比较方法之间存在显著差异”。由于计算出的 p 值小于 0.05，表明接受备择假设而拒绝零假设，即

RemCCG 的对比实验具有统计显著性。值得注意的是，在 LLVM 中检测到的缺陷数量大于 GCC 中的缺陷（如表4.1中 GCC 的 3.7 和 LLVM 的 13.2）。虽然两个编译器开发版本发布的时间比较接近，但两个编译器的开发历史不一样：GCC 的历史比 LLVM 更长（GCC 的第一个版本于 1987 年发布，而 LLVM 于 2003 年发布）。因此，LLVM 存在更多的缺陷是合理的。综述所述，RemCCG 中未被挖掘出的功能有助于提高现有程序生成器的缺陷检测能力。RemCCG 在实践中的缺陷检测能力评估结果将具体在后续的 RQ3 中进行总结和分析。

对 RQ1 的回答：RemCCG 不仅能够促进基于生成的方法（在 GCC 和 LLVM 编译器上的缺陷检测数量分别比 CCG 提高 16% 和 11%），而且可以促进基于突变的方法（在 GCC 和 LLVM 编译器上的缺陷检测数量分别比 Hermes 提高 14% 和 11%）。

4.5.2 新设计组件的有效性分析

（1）多样化的代码片段组件的有效性

表4.3列举了对新组件有效性的评估结果。每行每列的含义和表4.1相同。结果显示，在 GCC 和 LLVM 编译器上，RemCCG 能够分别检测到 2.6 个和 6.8 个缺陷，而 RemCCG(¬G) 只能分别检测到 2.4 和 5.4 个缺陷，在检测到的缺陷数量上分别实现了 8% 和 26% 的提升。为了进一步理解 RemCCG 中引入的新语言特性（如 while-loop 语句）对结果的影响，在 RQ2 中，作者还仔细检查了报告的揭示缺陷的测试程序。结果显示，在 56 个测试程序中，只有 5 个包含了原始 CCG 中不存在的新语言特性，表明 91% 的揭示缺陷测试程序是通过 RemGen 有效再制造过程构造出来的，进一步说明了程序生成器内置函数生成代码片段的有效性。

表 4.3　新组件有效性评估结果

Tab. 4.3　Results of the effectiveness of two newly designed components

编译器	对比方法	Average Statistics			
		Crash	*Performance*	*Sum*	*Improvement*
GCC	RemCCG(¬G)	2.3	0.1	2.4	8%
	RemCCG(¬S)	1.8	0.2	2.0	30%
	RemCCG(S_R)	2.1	0.1	2.2	18%
	RemCCG	2.4	0.2	2.6	-
LLVM	RemCCG(¬G)	5.2	1.2	5.4	26%
	RemCCG(¬S)	4.0	1.0	5.0	36%
	RemCCG(S_R)	4.0	1.2	5.2	31%
	RemCCG	5.4	1.4	6.8	-

（2）揭示缺陷代码片段选择组件的有效性

表4.3中的结果显示，在 GCC 和 LLVM 编译器上，RemCCG(¬S) 能够分别检测到 2.0 个和 5.0 个缺陷，RemCCG(S_R) 能够分别检测到 2.2 个和 5.2 个缺陷。与 RemCCG 检测到的 2.6 个和 6.8 个缺陷相比，RemCCG 在缺陷检测平均数上对 RemCCG(¬S) 实现了 30% 和 36% 的提升，对 RemCCG(S_R) 实现了 18% 和 31% 的提升。以上 RemCCG 与 RemCCG(¬S) 和 RemCCG(S_R) 的对比结果清楚地证实了揭示缺陷的代码片段选择组件的有效性，即该组件对 RemCCG 有积极的贡献。

对 RQ2 的回答： RemGen 中设计的两个组件，即多样化代码片段组件和揭示缺陷代码片段选择组件，均对 RemCCG 的缺陷检测能力有积极的贡献，与各种变体方法相比，在缺陷检测数量上提升数值达 8% 至 36%。

4.5.3 实践中缺陷检测能力分析

（1）检测到的缺陷统计

表4.4列举了 RemCCG 在两个编译器报告的 56 个新编译器缺陷。第一列表示缺陷报告的状态，第二至四列分别表示 RemCCG 在 GCC 编译器、LLVM 编译器和两个编译器检测到的缺陷总数。从表中可以看出，在总共报告的 56 个缺陷中，61% 的缺陷（即 37 个）已经被开发者修复。值得注意的是，报告给 LLVM 的缺陷数量（46 个报告，其中 29 个已经被开发者修复）略多于报告给 GCC（10 个报告，其中 8 个已经被开发者修复）的缺陷数量。产生该结果的可能原因是 LLVM 中优化组件的独特和复杂特性可能导致更多的缺陷。例如，现有研究[10] 表明，LLVM 中的不同最优选项序列也可能导致严重的缺陷。

表 4.4 提交的缺陷报告列表

Tab. 4.4 Results of all the reported bugs

缺陷报告状态	GCC	LLVM	总计
Fixed	8	29	37
WorksForMe	0	2	2
Duplicate	2	3	5
Pending	0	12	15
Total	10	46	56

由于开发者需要一些时间来确认报告的缺陷，主干版本在此期间会频繁地更新。因此，某些更改可能会延迟对缺陷的修复进程。在更新期间，由于编译器版本存在更新问题而待确认的缺陷通常会被标记为 “WorksForMe”。实验中在 LLVM 编译器上检

测到了两个此类缺陷。因为 GCC 的开发者对新报告的缺陷反应迅速，所以在 GCC 编译器上没有出现该类缺陷报告。此外，因为开发者通常平均花费较长的时间（如超过一年[16,18]）来修复缺陷，仍有 12 个缺陷等待开发者的回应。RemCCG 还报告了 5 个重复的缺陷，其中 3 个缺陷是性能缺陷。因为两个缺陷的崩溃位置与已存在的缺陷报告（如 bug#100512）不同的断言信息，导致作者误解了该缺陷，所以有两个 GCC 崩溃缺陷（如bug#100578）也被标记为 “Duplicate”。表4.6进一步列出了所有 37 个已被开发者修复的缺陷，包括被测的编译器及其缺陷报告编号、优先级、类型、受影响的优化选项和受影响的版本。

（2）缺陷类型统计

实验评估时本文按以下标准对缺陷的类型进行区分。如果一个编译器在编译过程中崩溃（如 GCC 编译时期异常终止或者 LLVM 中出现断言失败），则检测到一个崩溃（Crash）缺陷。如果编译器花费了相当长的时间（根据之前的研究[2] 设置为 30 秒），来编译一个测试程序，则检测到一个性能（Performance）缺陷。基于上述分类，37 个已修复的缺陷可以分为两类（即崩溃和超时缺陷），结果如表4.5所示。

表 4.5　已修复的缺陷类型统计
Tab. 4.5　Results of bug types of fixed bugs

缺陷报告类型	GCC	LLVM	总计
Crash	6	16	22
Performance	2	13	15
Total	8	29	37

（3）缺陷的重要性说明

开发者积极处理了 RemCCG 提交的缺陷报告。截止论文提交时，编译器开发者已经修复了大多数（66%）缺陷。具体而言，GCC 开发者对提交的缺陷反馈速度更快，修复了报告的所有缺陷。特别地，衡量缺陷重要性的方法是查看开发者在缺陷报告中设置的 “Importance” 字段属性。从表4.6中可以看到，开发者将 8 个 GCC 缺陷中的 5 个标记为 “P1” 或 “P2”，即两个最高优先级（默认优先级为 “P3”）。对于 LLVM，因为开发者通常不对缺陷优先级进行分类，所以他们将所有报告缺陷均标记为默认值 “P Normal”。即便如此，许多缺陷均使得已发布的版本上进行了回退（如bug#49205、bug#49218、bug#49475、bug#49541至 LLVM-12.0.0，以及bug#50308至 LLVM-12.0.1）。因为开发者通常将最严重的缺陷回退并在已发布的版本上进行彻底修复，所以以上提交的缺陷报告清楚地证明了 RemCCG 提交缺陷的重要性。

（4）受影响的优化级别

RemCCG 报告的缺陷存在于编译器较深的代码区域。基于已有研究[16,18]，深层

表 4.6　已修复缺陷的细节统计
Tab. 4.6　Details of fixed bugs

序号	编译器-报告编号	优先级	缺陷类型	影响的优化级别	影响的编译器版本
1	GCC-99694	P2	*Performance*	-O1,2,3	9.3-11.0 (trunk)
2	GCC-99880	P2	*Crash*	-O3	10.2-11.0 (trunk)
3	GCC-99947	P1	*Crash*	-O3	11.0 (trunk)
4	GCC-100349	P2	*Crash*	-O2,3,s	11.0-12.0 (trunk)
5	GCC-100512	P3	*Crash*	-O2,3,s	12.0 (trunk)
6	GCC-100626	P2	*Crash*	-O1,2,3,s	11.0-12.0 (trunk)
7	GCC-102057	P3	*Crash*	-O1,2,3,s	12.0 (trunk)
8	GCC-102356	P3	*Performance*	-O3	11.0-12.0 (trunk)
9	LLVM-49171	P3	*Performance*	-O3	13.0 (trunk)
10	LLVM-49205	P3	*Performance*	-O1,2,3,s	11.0-13.0 (trunk)
11	LLVM-49218	P3	*Crash*	-O1	12.0-13.0 (trunk)
12	LLVM-49396	P3	*Crash*	-O2,3,s	12.0-13.0 (trunk)
13	LLVM-49451	P3	*Crash*	-Os	13.0 (trunk)
14	LLVM-49466	P3	*Crash*	-O2	13.0 (trunk)
15	LLVM-49475	P3	*Performance*	-O1	12.0-13.0 (trunk)
16	LLVM-49541	P3	*Performance*	-O2,s	7.0-13.0 (trunk)
17	LLVM-49697	P3	*Crash*	-O3	7.0-13.0 (trunk)
18	LLVM-49785	P3	*Performance*	-O3	13.0 (trunk)
19	LLVM-49786	P3	*Performance*	-O2	13.0 (trunk)
20	LLVM-49993	P3	*Crash*	-O3	13.0 (trunk)
21	LLVM-50009	P3	*Crash*	-Os	12.0-13.0 (trunk)
22	LLVM-50050	P3	*Crash*	-O2,3,s	13.0 (trunk)
23	LLVM-50191	P3	*Crash*	-O2	13.0 (trunk)
24	LLVM-50238	P3	*Crash*	-O1,2,3,s	13.0 (trunk)
25	LLVM-50254	P3	*Performance*	-O2,3	13.0 (trunk)
26	LLVM-50279	P3	*Performance*	-O3	13.0 (trunk)
27	LLVM-50302	P3	*Performance*	-O3	13.0 (trunk)
28	LLVM-50307	P3	*Crash*	-Os	13.0 (trunk)
29	LLVM-50308	P3	*Performance*	-O1,2,3,s	12.0-13.0 (trunk)
30	LLVM-51553	P3	*Crash*	-O3	14.0 (trunk)
31	LLVM-51584	P3	*Performance*	-O1,2,3,s	14.0 (trunk)
32	LLVM-51612	P3	*Crash*	-O2,3	14.0 (trunk)
33	LLVM-51656	P3	*Crash*	-O2,3	14.0 (trunk)
34	LLVM-51657	P3	*Performance*	-O2,3,s	12.0-14.0 (trunk)
35	LLVM-51762	P3	*Performance*	-O1	14.0 (trunk)
36	LLVM-52018	P3	*Crash*	-O3	14.0 (trunk)
37	LLVM-52024	P3	*Crash*	-O2	14.0 (trunk)

次优化缺陷通常是难以被检测到的。本章的研究加强了现有工作的中后端缺陷检测能力，即所有缺陷均与优化阶段有关。具体而言，在已修复的缺陷中，有 26 个是由“-O3”（该选项默认打开了大多数细粒度的优化选项）选项引起的，而 6 个缺陷出现在所有的 “-O1” 到 “-Os” 选项中。上述结果进一步证明了 RemCCG 具有检测深层次编译器中后端缺陷的能力。

（5）受影响的编译器版本

RemCCG 方法可以检测到许多长期潜伏的缺陷。尽管本实验的重点是测试 GCC 和 LLVM 的开发版本，但 RemCCG 检测到了 10 个相对较旧版本编译器中的缺陷（如表4.6所示）。值得注意的是，其中两个缺陷已经存在了很长时间。其中一个是 GCC 的 (bug#99694)，另一个是 LLVM 的 (bug#49541)，在 RemCCG 报告该缺陷之前，它们已经分别存在了超过 1 年和 3 年时间。

（6）各类已确认的缺陷示例分析

此小节挑选了 RemCCG 所检测到的部分缺陷，以展现其在编译器中后端测试中检测多种类型缺陷的能力。该类缺陷对开发者产生了实质性的影响，其中一些甚至被标记为最高严重性的 “P1” 或 “P2”，例如在表4.6中的 GCC 缺陷 #99947。

```
1  #include <stdint.h>
2  int a, b, d, e;
3  int16_t c;
4  void f() {
5    for (; e; e++) {
6      int g = 6;
7      for (; g > 2; g--) {
8        int i = -8;
9        while (i < 20) {
10         i += 5;
11         a += b;
12       }
13       c *= d;
14     }
15     b--;
16   }
17 }
```

(a) GCC 崩溃缺陷 #99947:

```
1  #include <stdint.h>
2  int a; int8_t b;
3  void c() {
4    for (b = 8; b <= 6;);
5    int d = -4;
6    while (d < 20) {
7      d += 5; int e = 7;
8      do {
9        int f = 7; a = b;
10       while (f--)
11       for (b = 20; b <= 30; b++) {
12         int8_t *g = &b;
13         (*g -= a) || 0 > g < g;
14       }}
15     while (e--);
16   }
17 }
```

(b) LLVM 崩溃缺陷 #49697

图 4.5　由 RemCCG 检测到的崩溃缺陷示例

Fig. 4.5　Bug examples of crash bugs in GCC and LLVM detected by RemCCG

GCC 崩溃缺陷 #99947。图4.5(a)所示的测试程序使开发主干版本的 GCC 编译器在采用 “-O3” 优化选项时产生崩溃。该缺陷产生的根本原因是 GCC 编译器在处理向量 push 操作时出现缺陷，导致编译器崩溃。值得一提的是，开发者将该缺陷标记为最严重的 “P1” 等级，并在作者提交报告的 24 小时内对其进行了及时修复。

LLVM 崩溃缺陷 #49697。图4.5(b)所示的测试程序使开发主干版本的 LLVM 编

译器在采用 “-O3” 优化选项时产生崩溃。该缺陷产生的根本原因是 LLVM 在采用 “LoopStrengthReduce” 优化处理代码中第 6 行开始的复杂循环时出现错误，导致 LLVM 编译器内部处理出错而崩溃，严重影响了用户体验并可能延迟开发进度。

```
1  #include <stdint.h>
2  int a, b, c;
3  void d() {
4    uint16_t e;
5    int32_t *f;
6    int32_t *g;
7    if (a) {
8      int32_t *k;
9      for (;; *k += 1) {
10       int32_t **i = &f;
11       int32_t **l = &g;
12       for (e = 6; e; e++) {
13         g = k = f;
14       j:
15         **l = 0;
16       }
17       *i = c;
18     }
19   }
20   uint16_t i = &e;
21   b = i / 0;
22   goto j;
23 }
```

(a) GCC 性能缺陷 #99694:

```
1  #include <stdint.h>
2  int a,b,c;
3  void d(int e) {
4    int8_t *f; int16_t g;
5    int32_t i = &a; uint16_t *j;
6    int8_t *k = &c; int16_t l = 246;
7    uint64_t m; int8_t **n = &k;
8    int64_t o; int16_t *p;
9    for (; p;) {
10     int64_t *q = o;
11     for (*q = 5; *q; *q += 1)
12       if (*k = b)
13         for (*j = 3; *j; j++) {
14           int8_t r; o = r;
15         }
16     for (; p <= 2; p++) s: l = 1;
17   }
18   g = m = e; uint64_t v;
19   int32_t **u = &i; uint64_t t = &v;
20   f = *u; *f = c = l;
21   v = (g ?: (**u = m << **n)) == *f;
22   for (; i <= 8; *f = t) ; goto s;
23 }
```

(b) LLVM 性能缺陷 #51657

图 4.6　由 RemCCG 检测到的性能缺陷示例

Fig. 4.6　Bug examples of performance bugs in GCC and LLVM detected by RemCCG

GCC 性能缺陷 #99694。图4.6(a)所示的测试程序使 GCC 开发主干版本及多个发布版本（如 GCC-10.2.0 和 GCC-9.3.0）在采用 “-O3” 优化选项时产生超时缺陷，受影响的编译器采用无休止的时间编译该测试程序。该缺陷产生的根本原因是 GCC 在遇到 for 循环时，在执行 VN（即 Value Numbering）操作时出现错误，导致编译器进行无休止的编译过程。值得一提的是，开发者将该缺陷标记为严重的 “P2” 等级，并在作者提交报告的 24 小时内对多个存在该缺陷的编译器版本进行了及时修复。

LLVM 性能缺陷 #51657。图4.6所示的测试程序使 LLVM-12.0 开始至开发主干版

本的多个版本在采用 “-O3” 优化选项时产生超时，受影响的编译器采用无休止的时间编译该测试程序。该缺陷产生的根本原因是 RemCCG 构造的测试程序在采用 “-O3” 优化选项时产生了少见的未优化 IR（即 LLVM Bitcode）指令序列，因此触发了 LLVM 编译器 Instcombine 优化中的错误。值得一提的是，开发者对该缺陷报告表示了积极的肯定[6]并鼓励作者多提交类似的重要缺陷报告，以帮助更好地检测更多的前端缺陷从而进一步保障编译器质量。

通过以上 RemCCG 检测到的缺陷示例，可以看出 RemGen 能够有效地再制造生成器从而构造出触发编译器深层次中后端的测试程序。

对 RQ3 的回答： RemCCG 在实际应用中能够有效地检测深层次编译器中后端缺陷。RemCCG 总共报告了 GCC 和 LLVM2 种类型 56 个重要的编译器中后端缺陷。其中，有 37 个缺陷已经被开发人员修复。

4.6 讨论

本节讨论算法4.4中边界值 N 的选择对 RemCCG 有效性的影响、与现有工具（如 Csmith 和 YARPGen）的比较结果、RemGen 的局限性以及实验中存在的有效性威胁。

（1）边界值选择对 RemCCG 的影响

在实现算法4.4时，本文将界限 N 设置为 10。然而，目前尚不清楚界限值的选择对方法有效性的影响。为此，本文设置了不同的界限值（即 3、5、10、20、40、60、80 和 100），以评估其缺陷检测的能力。具体而言，本文在 GCC-4.4.3 上运行不同的界限值，运行时间同样为 24 小时，并且重复 5 次然后取平均值。结果显示值边界值为 10 时 RemCCG 的缺陷检测能力最强。因为较小的界限值限制了构造出的测试程序的多样性，而较大的界限值虽然可以合成多样化的测试程序，但它也会增加代码片段选择和程序编译的时间成本。综上，选择一个适中的值是合理的。为了在选择代码有效性和检测缺陷效率之间取得更好的权衡，RemCCG 选择了一个适中的值（即 10）。

（2）与 Csmith 和 YARPGen 的比较

除了第4.4节的实验评估外，本文还进行了附加实验进一步比较了 RemCCG 与其他现有方法在缺陷检测能力的效率。具体而言，通过采用与 RQ1 中相同实验设置，本文将 RemCCG 与另外两种广泛应用的基于生成的测试程序构造方法（即 Csmith[2] 和 YARPGen[3]）进行比较。结果表明，RemCCG 的性能明显优于 Csmith 和 YARPGen，即 RemCCG 在 GCC/LLVM 中比 Csmith 和 YARPGen 分别多检测出 164%/363% 和 120%/595% 的缺陷。上述实验结果是合理的，因为 Csmith/YARPGen 和 RemCCG 具有不同的设计和实现，以及不同的测试目标。具体而言，Csmith 和 YARPGen 旨在检

[6]https://bugs.llvm.org/show_bug.cgi?id=51657#c7

测编译器中错误编译缺陷，该类缺陷的触发要求测试程序不含有任何未定义行为。由于 RemCCG 可以构造出 Csmith 难以构造的互补测试程序（即语义多样化但可能包含未定义的行为的测试程序），因此 RemCCG 的缺陷查找能力优于 Csmith 和 YARPGen 是合理的。值得注意的是，RemCCG 具有构造能够揭示深层次编译器缺陷测试程序的能力，开发人员也以积极的反馈[7]证实了以上结论。

（3）RemGen 的局限性

经过 RemGen 再制造处理后的 RemCCG 的局限性主要来源于再制造之前的程序生成器 CCG。具体而言，RemCCG 只能在编译器中找到两种缺陷（即崩溃和性能缺陷）。然而，本章提出的框架 RemGen 是通用的，可以应用于其他现有的程序生成器，例如 Csmith[2] 和 YARPGen[3]。为了再制造对测试程序有高较需求（即构造出的测试程序不含有未定义行为）的程序生成器，在再制造该类程序生成器的过程中，应该在合成的代码片段中考虑更有效的检查模块（即在图4.4中的❹之前）。该检查模块用于保证合成代码片段的有效性（即没有未定义的行为）。如果合成的代码片段在检查中失败，则应拒绝该代码段指导 RemGen 并重新合成。值得注意的是，实现高效的检查组件设计并非易事。例如，Csmith 采用复杂的启发式算法来确保测试程序没有未定义的行为，并且以上的检查过程被证明是费时的[3]。因此，在再制造过程中应考虑更实用的解决方案（如采用静态分析技术），使检查过程变得高效。作者考虑沿着该方向继续扩展 RemGen，使其更加通用化以检测更多类型的缺陷。

（4）有效性威胁

RemCCG 方案中主要存在的内部、外部和构建有效性方面的威胁。

① 内部有效性威胁。主要的内部威胁来自本研究中选择的旧程序生成器。本研究只选择了 CCG 而没有选择其他现有的程序生成器，再制造后的 RemCCG 只能检测两种类型的缺陷（即崩溃或性能缺陷）。然而，基于第4.4节的实验评估，结果表明 RemCCG 不仅能够提升两种流行的测试程序构造方法，而且在实践中具有较强的缺陷检测能力。作者将再制造其他程序生成器的研究方向作为将来的工作之一。另一个威胁在于 RemCCG 的实现。算法4.4中边界值 N 的选择可能会影响 RemCCG 的有效性。如第4.6节的讨论，本研究已经通过评估不同边界值的影响来确定最佳的边界值。因为在讨论中已经进行了大量的实验来评估不同边界值的有效性，由边界值选择带来的威胁可以有效得到缓解。另外，本文采用了较长时间评估 RemCCG 整体的有效性。例如，为了充分评估 RQ1 中每种方法的缺陷检测能力，RQ1 的整个实验持续了 30 多天（每种方法运行 10 次每次 90 小时）。以上充足的测试周期可以减轻该威胁。另外，实现的代码可能含有一定的局限性。为了减轻该威胁，作者认真检查了 RemCCG 的实现并对其进行了详细的测试。

[7]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99694#c15

② 外部有效性威胁。外部威胁主要在于测试对象的选择。实验中选择了两个版本的两个编译器作为测试对象，此对象可能不足以代表不同编译器（未测试的编译器如 Intel C++ Compiler 或 Microsoft Visual Studio Compiler）。为了减少该威胁，和现有研究的实验设置一样[2,3,5,6,28,61]，本章在实验中也选择了两个最受欢迎且被广泛研究的 C 编译器，即 GCC 和 LLVM。更具体而言，本文考虑了不同版本的编译器，包括旧版本（在 RQ1 和第4.6节中的附加实验）和被活跃开发者维护的开发版本（RQ3 的实验评估），以在多种不同版本的编译器测试对象上评估 RemCCG 的有效性，从而减小由不同测试对象带来的威胁。

③ 构建有效性威胁。该威胁可能涉及随机性、评估指标以及评估过程中的参数设置等方面带来的潜在威胁。实验中本文已经通过以下方法解决缓解问题：1）将实验重复 10 次并记录其平均数作为最终的实验结果，从而减少因随机性而带来的影响；2）采用总体检测到的缺陷数量和唯一检测到的数量来评估 RemCCG 的有效性，以确保评估结果的可靠性和可比性；3）在第4.6节讨论了不同边界值对 RemCCG 缺陷检测能力的影响，并根据实验结果选择了合适的边界值。经过以上步骤，降低了相关构建有效性威胁，增强了 RemCCG 实验评估结果的可信度。

4.7 本章小结

本章提出了一种面向编译器中后端测试的再制造生成器构造方法 RemGen，构造出了语义多样化的测试程序，有效检测出了深层次编译器中后端缺陷。RemGen 主要设计了两个新组件，即多样化代码片段合成与揭示缺陷代码片段选择组件，对现有程序生成器进行再制造。多样化代码片段合成组件采用语法辅助代码片段合成来生成多样化的代码片段。特别地，该组件首先在程序生成器中保留在生成过程中产生的上下文，然后利用该上下文中调用生成器中的内置函数来合成新的代码片段。揭示缺陷代码片段选择组件采用语法覆盖率指标，记录合成过程中所选代码片段的语法规则的使用频率。在覆盖率指标的指导下，RemGen 选择最具多样化语法覆盖的代码片段来构建新揭示缺陷的测试程序。在实验中，本章将旧生成器 CCG 经过 RemGen 构造为一个新程序生成器 RemCCG，并在两种广泛使用且成熟的编译器（即 GCC 和 LLVM）上对 RemCCG 的有效性进行了评估。实验结果表明，在缺陷检测数量上与 CCG 进行对比，RemCCG 显著由于基于生成和变异的测试程序构造方法。此外，RemCCG 向 GCC 和 LLVM 开发者报告了共 56 个缺陷，其中 37 个已被开发者及时修复。值得注意的是，许多提交的缺陷均为严重、深度和长期潜伏的缺陷（5 个缺陷被标记为最高严重性，2 个缺陷分别潜伏了 1 年和 3 年）。以上结果充分证明了 RemCCG 在实践中具有较强的缺陷检测能力。

5 面向编译器缺陷定位的大模型赋能程序构造方法

5.1 概述

编译器在构建可靠软件系统中起着重要作用，编译器缺陷会产生灾难性的后果[2,8]。当缺陷被检测出来后，开发者通常会及时对编译器缺陷进行调试。编译器调试中的一项关键任务是如何及时有效地定位缺陷，以保证缺陷能够被及时修复。然而，由于编译器缺陷调试信息有限且编译器源文件数量庞大（1500+ 个源文件），定位编译器缺陷面临重大挑战。目前解决该问题的一个常见方法是将缺陷定位问题转换为测试程序变异问题[12,13]。该类方法背后的核心思想是首先通过改变给定失败（即可以触发编译器缺陷）的测试程序构造一组通过（即不能触发编译器缺陷）的证人测试程序，然后收集通过和失败的测试程序在编译器源代码上的覆盖率（即程序频谱）。最后，结合基于频谱的缺陷定位[19,23]（SBFL）技术对可疑文件进行排名。

DiWi[12] 和 RecBi[13] 是现有工作中最先进的两个编译器缺陷定位方法。然而，DiWi 和 RecBi 仍然存在一些局限性，使得它们不能高效地完成缺陷定位任务。首先，现有方法中内置的变异策略在构造多样化的证人测试程序方面受到限制。为了统一表述，本章将能够从可疑文件集合中消除文件嫌疑（即不能触发缺陷）的测试程序叫做证人测试程序。具体而言，DiWi 仅支持局部变异，例如只能够通过更改变量的类型以完成对失败测试程序的变异操作。尽管 RecBi 支持更多的变异策略（如结构化的变异，即改变失败测试程序的控制流），但它只能合成语句条件结构而不能合成语句体（详见第5.2节动机示例代码）。其次，DiWi 和 RecBi 对变异变量和位置选择的随机性也使得构造的证人测试程序多样性受到一定限制。由于面向编译器缺陷定位的变异旨将触发缺陷的失败测试程序变异成通过的证人测试程序，因此，变异的位置和变异时使用的变量对变异的有效性至关重要。最后，DiWi 和 RecBi 的变异过程需要大量的人力消耗。在变异之前，DiWi 和 RecBi 需要手动编写功能代码来收集失败测试程序中必要的上下文信息，RecBi 同样需要手动从现有的失败测试程序中提取相关程序语义。此外，现有变异策略较少关注构造出的证人测试程序是否包含未定义行为，因而继续降低缺陷定位的有效性。总体而言，程序变异的限制以及较多的人力消耗凸显了对提高面向编译器缺陷定位程序变异效率的迫切需求。

由于预训练大型语言模型（即 Large Language Models，简称 LLMs，如 ChatGPT[105]）在代码生成方面的高效性和轻量化的构造过程，本章提出了面向编译器缺陷定位的大语言模型赋能程序构造方法 LLM4CBI（**L**arge **L**anguage **M**odels for **C**ompiler **B**ug **I**solation）。设计 LLM4CBI 主要出发点是：1）LLMs 由超大规模代码和文本训练而成，因此 LLMs 产生的测试程序通常是多样化的；2）LLMs 具有良好的学习和反思机制，

根据用户的反馈，能够按照快速“提示-回答”范式生成更好的输出；3）LLMs 采用的提示是自然语言表达，更方便用户使用，减少完成任务的劳动力消耗。因此，充分挖掘 LLMs 构造有效证人测试程序的潜能是合理的。然而，直接采用 LLMs 构造有效的证人测试程序辅助编译器缺陷定位存在一定困难，需要解决以下两个重要挑战。

第一个挑战是如何制定精确的提示。提示的质量在释放 LLMs 的程序变异能力方面起着至关重要的作用[106]，但采用现有的自然变异描述作为提示的有效性受限。例如，变异规则“插入 if 语句”是现有工作 RecBi[13] 中采用的变异描述。该描述缺乏采用的具体变量以及插入语句的具体位置的精确性，降低了将失败测试程序（即可以触发缺陷的测试程序）变异为通过（即不能触发缺陷的测试程序）测试程序的可能性。由于难以选择精确的变量和位置，现有方法 DiWi[12] 和 RecBi[13] 仅采用随机的方法选择变量和变异位置。然而，该随机策略被证明是无效的（详见第5.5.2节中的评估结果）。因此，有必要制定精确的变异提示，以协助 LLMs 构造有效的证人测试程序。第二个挑战是如何选择专用提示。当收集到多个提示时，选择用于改变特定失败测试程序的专用提示不但重要且具有挑战性。原因有两个方面。首先，各个编译器缺陷的特征往往不同，并且触发不同缺陷的失败测试程序具有不同的语言特性。具体而言，某个提示对于改变一个特定的失败测试程序可能有效，但可能对另一个失败测试程序没有帮助。随机选择提示来改变测试程序可能是无效的。此外，LLMs 对不同的失败测试程序进行变异时可能会产生不同的错误，为了提高效率，有必要采用不同的提示对失败测试程序进行变异。因此，有必要设计一种新的提示选择策略来选择针对每个编译器缺陷的专用提示以提高程序变异效率。

为了解决上述挑战，LLM4CBI 设计了三个新组件，以充分挖掘 LLMs 构造有效证人测试程序以辅助编译器缺陷定位的潜能。首先，设计了一个精确提示制定组件以解决第一个挑战。该组件首先引入了一种精确的提示模版，该模版可以准确表示 LLMs 所需的变异操作。然后，LLM4CBI 通过数据和控制流分析失败测试程序的复杂度指标来识别最相关的变量和最佳插入位置。其次，LLM4CBI 提出基于强化学习提示选择和轻量级测试程序验证等两个新组件，用于选择针对各个缺陷的专用提示。在提示选择组件中，LLM4CBI 通过强化学习结合记忆搜索，针对各个编译器缺陷，LLM4CBI 根据 LLMs 的表现跟踪和积累奖励，允许 LLM4CBI 不断选择专门的提示来改变特定的失败测试程序。在测试程序验证组件中，LLM4CBI 利用形式化分析技术来检测和过滤掉可能包含未定义行为的测试程序，从而降低与无效程序相关的风险。

为了评估 LLM4CBI 的有效性，本章在两个广泛应用的 C 开源编译器（即 GCC 和 LLVM）的 120 个真实缺陷上进行了实证评估。首先，为了评估 LLM4CBI 在编译器缺陷定位能力，本文将 LLM4CBI 与两种最先进的方法（即 DiWi[12] 和 RecBi[13] 进行了比较。结果表明，在 Top-1/Top-5 的结果中，LLM4CBI 可以分别比 DiWi 和 RecBi

多定位 90.91%/35.14% 和 50.00%/13.64% 的缺陷。其次，实验还评估了 LLM4CBI 的三个新组件（即精确提示制定、基于强化学习提示选择以及轻量化测试程序验证组件）的有效性，结果表明所有组件均对 LLM4CBI 的缺陷定位能力有积极的贡献。最后，为了评估 LLM4CBI 的可扩展性，本文将各种最新的大语言模型扩展到 LLM4CBI 中。实验结果表明 LLM4CBI 有较高的可扩展性，很容易与其他大语言模型结合以适配最新的大语言模型技术进一步增强缺陷定位的性能。

5.2 背景和动机

本节首先介绍关于面向编译器缺陷定位的测试程序构造和大型语言模型（LLMs）的相关背景知识。然后，通过一个真实的缺陷示例来说明现有方法的局限性，并凸显 LLM4CBI 方法的优势。

（1）基于程序变异的编译器缺陷定位

第1.1.3节中的图1.3展示了现有基于程序构造编译器缺陷定位方法（如 DiWi[12] 和 RecBi[13]）的典型工作流程。具体而言，给定一个可以触发编译器缺陷的失败测试程序，现有方法首先采用不同的变异策略来构造一个不会触发任何缺陷的证人测试程序。然后，对失败和通过测试程序进行编译，从而可以收集编译器源文件的代码覆盖率信息。在编译过程中失败测试程序执行过的所有编译器文件均被视为可疑文件。相反，通过测试程序有助于减轻对可能涉及的可疑文件的怀疑。为了最终定位存在缺陷的文件，根据基于频谱的缺陷定位（SBFL）[23,107] 的既定原则，采用 Ochiai[23] 等公式比较失败测试程序和证人测试程序对编译器源代码的执行覆盖率（简称程序频谱），最终生成可疑度排序的文件列表。

最近两个研究工作 DiWi[12] 和 RecBi[13] 均遵循以上策略，它们的目标是采用程序变异的方法构造出与失败测试程序的控制流和数据流信息稍有不同的测试程序集合，从而构造一组能够将编译器执行结果从失败（failing）测试程序翻转为通过（passing）的证人测试程序。本章采用基于 LLMs 的新方法来实现相同的目标，以达到进一步辅助编译器缺陷定位的目的。

（2）大型语言模型

自 2022 年 11 月以来，ChatGPT[105] 等预训练大型语言模型（LLMs）表现出强大的处理问题能力，相关应用无处不在，在许多任务中表现出了优异的性能，例如机器翻译[108]，文本摘要[109]，分类问题[110] 和代码生成[111] 等等。总体而言，LLMs 可以通过向模型提供任务描述（称为提示）来直接用于处理特定的下游任务，而无需对专用数据集进行微调。以上提示的制定通常是通过 “提示工程”[112–114] 的技术实现的。提示工程旨在找到在特定任务上产生最佳性能的提示。先前的研究[115,116] 表明，提示工程可以在各种下游任务上实现最先进的性能。受益于 LLMs 的巨大潜力，近期有越来越

多的研究工作表明 LLMs 可有效地用于解决软件工程中的各种不同任务[106,114,117–119]。与已有采用 LLMs 的任务不同，本章的研究目标是充分挖掘 LLMs 在证人测试程序构造任务领域的潜力。

从技术角度出发，Transformer 模型是 Vaswani 等人[120] 于 2017 年提出的一种深度学习模型，广泛应用于自然语言处理领域。该模型的核心由编码器和解码器组成，均由多层相似的神经网络结构构成。编码器包含自注意力机制和前向神经网络，通过该两个子层，模型能够在编码时考虑到输入序列中各个位置词的影响。解码器除了具有与编码器相同的结构外，还加入了编码器-解码器注意力层，使得解码器可以有效利用编码器的输出。同时，Transformer 模型还融入了位置编码，使模型能够理解序列中单词的顺序，从而实现对序列数据的高效并行处理。由于以上优良特性，使得 Transformer 模型成为许多现代自然语言处理模型的基础。现有的大多数 LLMs 采用了 Transformer 架构的解码器架构。具体而言，给定包含任务描述的提示，解码器按照公式5.1，逐个标记地生成测试程序 Y 的序列，

$$y_t = \arg\max_{y} P(y|p, y_{<t}) \tag{5.1}$$

其中 y_t 是要预测的当前标记，$y_{<t}$ 指的是先前预测的所有标记，p 是输入提示。该公式指出，在给定提示 p 和先前生成的标记 $y_{<t}$ 的情况下，要预测的当前标记 y_t 由选择使条件概率 $P(y|p, y_{<t})$ 最大化的标记 y 确定。由于构造出的测试程序 Y 取决于输入提示 p，并且输入提示 p 的搜索空间巨大，因此找到能够构造出有效证人测试程序 Y 的最佳提示 p 具有挑战性。本章提出一个新方法 LLM4CBI 来自动制定有效的提示 p，以此构造能够辅助编译器缺陷定位任务的有效证人测试程序 Y。

目前，大语言模型研究社区主要采用两种类别的 LLMs 解决代码生成任务：填充模型和通用模型[111,117,119,121]。填充模型（如 CodeGen[122]、Incoder[123] 和 PolyCoder[124]）用于在双向上下文中（即在代码片段中间）填充最自然的代码，而通用模型（如 LLaMA[125]、Alpaca[126]、ChatGPT[105]、Vicuna[127] 和 GPT4ALL[128]）旨在仅通过自然语言描述即可构造出满足特定提示需求的完整代码片段。由于通用模型遵循简单直接的 “提示-回答” 对话范式，涉及到最少的人力投入，符合本章研究的目标，因此，在本章研究中，作者主要考虑通用大语言模型对编译器缺陷进行定位。

（3）编译器缺陷定位动机示例

图5.1(a)展示了一个在 LLVM-3.4 编译器的 “-O3” 优化级别下触发缺陷的失败测试程序，该代码在 “-O3” 的优化等级的编译选项下使得编译后的二进制代码引发了一个 “division by zero” 错误。编译器产生该缺陷的根本原因是 LLVM 的 “induction variable elimination” 优化过程在处理该测试程序时引入了不合法的 “division by zero” 操作，从而导致编译器触发了该缺陷。值得注意的是，图5.1(b)表示本章提出的解决

```
1  short s; int a, b, c;
2  volatile int v;
3  static int u[] = {0, 0, 0, 0, 0, 1};
4
5  void foo() {
6    int i, j;
7    for (; b <= 0; ++b) {
8      int k, d = 0;
9      for (; d <= 5; d++) {
10       int *l = &c;
11       int e = 0;
12       for (; e <= 0; e++) {
13         int *m = &k;
14         unsigned int n = u[d];
15         i = !a ? n : n / a;
16         j = s ? 0 : (1 >> v);
17         *m = j;
18       }
19       *l = k < i;
20     }
21   }
22 }
23 int main() {
24   foo();
25   return 0;
26 }
27 # clang -O2 test.c ; ./a.out
28 # clang -O3 test.c ; ./a.out
29 Folating point exception (core dumped)
```

(a) 失败测试程序

```
1  short s; int a, b, c;
2  volatile int v;
3  static int u[] = {0, 0, 0, 0, 0, 1};
4
5  void foo() {
6    int i, j;
7    for (; b <= 0; ++b) {
8      int k, d = 0;
9      for (; d <= 5; d++) {
10       int *l = &c;
11       int e = 0;
12       for (; e <= 0; e++) {
13         int *m = &k;
14         unsigned int n = u[d];
15         i = !a ? n : n / a;
16         j = s ? 0 : (1 >> v);
17         *m = j;
18         if (a ==0) v = s;
19         else if ( a> 10) s = a + 1;
20         else s = 1;
21       }
22       *l = k < i;
23     }
24   }
25 }
26 int main() {
27     foo();
28     return 0;
29 }
```

(b) 通过的证人测试程序

图 5.1 LLVM 缺陷 #16041示例

Fig. 5.1 The example of LLVM bug #16041

方案（即 LLM4CBI）构造的一个通过（即不能揭示编译器缺陷）的证人测试程序，其中灰色部分代码由大模型构造。后续小节将通过以上示例展示现有方法的局限性以及 LLM4CBI 在构造证人测试程序方面的优势。

① 现有方法的局限性。现有方法 DiWi[12] 和 RecBi[13] 在构造上述通过的证人测试程序时存在以下几个方面的局限性。第一，DiWi 和 RecBi 采用的变异策略存在一定的局限性。具体而言，DiWi 仅支持不更改失败测试程序控制流的局部变异操作，如改变程序变量定义的类型。因此，该方法不能构造出其他条件控制语句，例如图5.1(b)中突出显示的灰色 if 语句 “if (a == 0)”。虽然 RecBi 允许插入结构条件语句，如 “if (a==0)”，但 RecBi 缺乏构造相应语句主体的能力，即无法构造出图5.1(b)中第 18 行的 “s=v;” 语句。第二，DiWi 和 RecBi 均依赖于随机选择策略来确定在变异过程中使用的具体变量以及将新的代码片段插入的具体位置，表明 DiWi 和 RecBi 能否将失败变成通过的结果在很大程度上取决于随机性。第三，现有方法中涉及的变异过程需要大量的人力

消耗。在变异之前，测试人员必须编写额外的功能代码收集失败测试程序中的上下文信息（如变量的名称 s, a, b, c 及其定义的类型），并从现有的代码仓库或者其他测试程序数据集中提取相关语句元素（如 if 语句）作为语句备选。此外，在变异过程中，还需要手动编写功能代码来随机确定在变异期间将新代码片段插入失败测试程序的合适位置（现有方法仅考虑随机选择插入的位置）。最后，现有方法较少关注证人测试程序的语义有效性。然而，语义无效的证人测试程序的出现会对编译器缺陷定位产生不良影响（详见第5.4节实验部分的示例说明）。总之，上述限制凸显了现有方法在构造高质量证人测试程序方面的局限性以及对人力消耗的较高需求等缺点，强调了克服以上缺点并能够有效构造出证人测试程序的新解决方案的迫切需求。下文将说明 LLM4CBI 如何克服以上缺点并辅助编译器缺陷定位任务。

②LLM4CBI 的优势。与 DiWi[12] 和 RecBi[13] 相比，LLM4CBI 擅长充分挖掘 LLMs 代码生成的潜能构造有效的证人测试程序。首先，LLM4CBI 通过新设计的组件制定了一个精确的提示，例如 “*根据失败测试程序 F，请构造一个变体程序 P，具体变异过程为：复用变量 {a, s, v} 并在第 11-27 行执行插入一个 if 语句操作*”，以指导 LLMs 按照用户特定的要求达到对失败测试程序进行变异的目的。值得注意的是，与采用模棱两可的提示信息不同，LLM4CBI 考虑了程序中更详细的语义信息（如数据流和控制流信息），从而构建精确的提示，以增加从失败到通过的翻转的可能性。通过采用制定出的精确提示，LLM4CBI 插入图5.1(b)中灰色显示的新 if 语句从而构造出新的证人测试程序。值得注意的是，LLM4CBI 支持在结构变异中插入语句体（如 “s = v;” 语句），以此增加证人测试程序的多样性。此外，得益于 LLMs 的快速响应对话范式，与已有研究相比，整个变异过程无需编写额外功能代码完成程序变异操作，仅涉及很少的人力消耗。另外，当 LLM4CBI 构造出一个新测试程序后，LLM4CBI 还会采用轻量化的验证方法对测试程序进行语义有效性检测，减小了语义无效证人测试程序对缺陷定位结果的影响，提高了基于频谱方法的有效性，进一步增强了 LLM4CBI 在编译器缺陷定位方面的能力。

5.3 LLM4CBI 框架描述

本节首先介绍 LLM4CBI 的总体设计框架概述。然后，在接下来的小节中详细介绍该框架中新设计三个组件，包括基于程序分析的提示制定、基于强化学习的提示选择以及轻量级的测试程序验证。

5.3.1 LLM4CBI 框架概述

图5.2展示了 LLM4CBI 的一般工作流程，LLM4CBI 利用 LLMs 代码生成的功能为编译器缺陷定位构造有效的证人测试程序，从而解决第5.1节提到的两个主要挑战，即如何制定精确的提示以及如何选择专用提示。从架构的角度出发，LLM4CBI 首先制

定精确的提示，并在精确提示制定组件中收集所有制定的提示（❶）。然后，LLM4CBI 通过基于强化学习的提示选择组件选择一个特定提示（❷），并将其用作为 LLMs 的输入（❸）。接下来，LLM4CBI 生成一个新测试程序（❹），该程序经过一个测试程序验证组件（❺）。如果构造的测试程序有效且为通过（即不能触发缺陷），则采用相关编译器对其进行编译（❻），并收集编译器源文件的覆盖信息。此外，在选择特定提示时，提示的奖励和损失计算（❼）由构造出的证人测试程序的质量，即其相似性和多样性指标来衡量，以帮助选择更好的提示。但是，如果程序由于语义缺陷而无效，LLM4CBI 向 LLMs 提供反馈（❽）提示，指导它们不再犯同样的错误。最终，在达到终止条件（如 1 小时）时，LLM4CBI 采用基于频谱的缺陷定位公式对编译器可疑源文件进行排名（❾）。具体而言，精确提示制定组件旨在解决精确提示制定的挑战。基于强化学习提示选择组件和轻量化测试程序验证组件用于解决选择针对不同缺陷的专用提示挑战。后续小节中将详细描述 LLM4CBI 中新设计的三个组件。

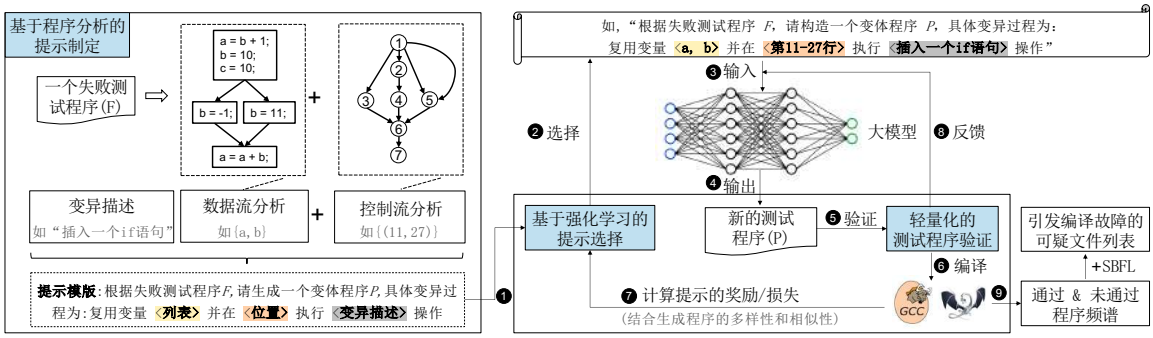


图 5.2　LLM4CBI 框架

Fig. 5.2　Overview design of LLM4CBI

5.3.2　基于程序分析的提示制定

本节介绍精确提示制定组件，该组件解决了精确提示制定的第一个挑战。本节首先概述 LLMs 的提示制定模版设计，然后描述如何利用静态程序分析技术并采用程序复杂性指标（如数据流和控制流复杂度）来填充该模版。

（1）程序变异的提示模式

为了充分挖掘 LLMs 代码生成的潜能，为 LLMs 提供精确的提示输入是获取其优异结果的保证。为此，LLM4CBI 设计了以下模版用于制定精确且有效的提示。

> 提示模版：“根据证人程序 F，请构造一个变体程序 P，具体变异过程为：复用变量＜列表＞并在＜位置＞执行＜变异规则＞操作”

在上述模版中，P 表示由大语言模型构造出的新测试程序，F 表示给定的输入失败

测试程序，即能够触发编译器缺陷的失败测试程序。至于其余部分，＜变异规则＞表示实际的变异操作（表5.1详细列出了模版中用到的变异规则）；＜变量＞和＜位置＞描述了执行变异时的特定要求。设计该模版的理由是因为编译器采用不同的策略来优化具有不同数据或控制流的测试程序[2]，且与在现有方法中随机改变程序相比，改变失败测试程序中最复杂的部分更有可能将失败翻转为通过（即不能触发编译器缺陷）。为此，LLM4CBI 通过两种方式测量最复杂的部分：保存包含复杂数据流的变量列表和涉及复杂控制流的位置（即行号范围）。下文将详细描述 LLM4CBI 中采用的数据流和控制流分析技术。

表 5.1　应用于提示模版中的变异规则列表
Tab. 5.1　Mutation rules applied in the prompt pattern

序号	变异规则的自然语言描述
1	插入 if 语句
2	插入循环语句（如 while 和 for 等）
3	插入函数调用
4	插入 goto 语句
5	插入修饰符（如 volatile、const 和 restrict 等）
6	移除修饰符（如 volatile、const 和 restrict 等）
7	插入修饰符（如 long、short、signed 和 unsigned 等）
8	移除修饰符（如 long、short、signed 和 unsigned 等）
9	用另一个有效的常量替换常量
10	用另一个有效的二进制运算符替换二进制运算符
11	移除变量上的一元运算符
12	用另一个有效的一元运算符替换一元运算符
13	用另一个有效的变量替换变量

（2）基于数据流分析的复杂变量选择

数据流分析旨在输出在给定的失败测试程序中定义和使用的最复杂的变量列表。由于 LLM4CBI 的目标是通过量化变量的复杂度以检查变量如何影响以及在多大程度上影响测试程序的执行过程。在本章的研究中，假设变量被定义（或使用）的越多，变量上的复杂依赖关系则越多。因此，改变最复杂的变量更可能将失败的程序变异为通过的测试程序。因此，本章依据现有的数据流复杂度定义[129]，并借鉴了现有工作[130]选择采用变量的 Def-Use 链[7] 来分析程序的数据流复杂度。具体而言，LLM4CBI 通过以下公式计算变量的复杂度 ($Comp_{var}$)：

$$Comp_{var} = N_{def} + N_{use} \tag{5.2}$$

其中 N_{def} 计算变量被定义的次数，包括重新定义或赋值。值得注意的是，由于 Def 跟踪变量的更改情况，因此数据依赖性分析被包括在内，保证了详尽数据流分析的执

行。N_{use} 计算变量被使用的次数。该次数是指变量值在一些计算中被使用而没有被修改的次数。根据公式5.2，LLM4CBI 对图5.1(a)中的失败测试程序进行数据流分析将产生一个变量列表 {a, s, v}，该列表即为程序中使用最多列表中排名前三复杂变量，并用于填充模版中的 “变量” 字段。

（3）基于控制流分析的复杂位置选择

控制流分析的目标是输出失败测试程序中最复杂语句（即最佳执行变异操作）的位置。LLM4CBI 利用输入失败测试程序的控制流图（即 CFG），其中各个节点表示语句，边表示执行流程。根据控制流图，基于圈复杂度[129]，LLM4CBI 采用公式5.3计算各个语句的复杂度：

$$Comp_{control} = N_{edge} - N_{node} + 2 \tag{5.3}$$

其中 N_{edge} 和 N_{node} 分别表示 CFG 中的边数和节点数。由于圈复杂度不是设计用于在语句级别上衡量复杂性的，LLM4CBI 通过在计算圈复杂度期间获取复杂性值来计算各个语句的复杂性。值得注意的是，由于包括测试预言的代码块，更有可能破坏测试预言，意味着可能会构造出虚假的证人测试程序。因此，LLM4CBI 选择性地忽略了包含测试预言（即包含 printf 或 abort 函数）的语句。再次以图5.1(a)所示的代码为例，LLM4CBI 中设计的控制流分析发现最复杂的控制流位于第 12-18 行之间的 for 循环中。该行号信息将用于填充提示模版中缺失的 “位置” 字段。

通过以上方式，在获取变量列表和要插入的所需位置后，LLM4CBI 基于设计的模式制定了一定数量的提示。例如，针对图5.1(a)的示例，制定的提示之一为：

> 提示示例：“根据证人程序 F，请构造一个变体程序 P，具体变异过程为：复用变量 <{a, s, v}> 并在 <第 12-18 行> 执行 <插入 if 语句> 操作”

如第5.1所述，针对不同的失败测试程序中，并不是各个提示均对其有相同的贡献。此外，LLMs 在变异程序时可能会产生相同的编程错误。例如，LLMs 在一个失败测试程序中可能会产生语法错误（如某个未定义的但被使用的变量），但在另一个程序中可能会产生同样的错误。因此，有必要将该类错误作为反馈输入给 LLMs，使其不再构造含有类似错误的测试程序。然而，由于该类错误可能存在重复，没有必要将所有错误信息均作为反馈提示提供。此外，由 LLMs 输出的测试程序可能含有未定义行为，该类程序将对缺陷定义的效果产生负面的影响。为了解决以上问题，LLM4CBI 制定了针对各个失败测试程序专用的提示选择策略（即基于强化学习的提示选择组件），并结合了轻量化测试程序验证组件来共同实现特定提示的选择。

5.3.3 基于强化学习的提示选择

本小节和下小节介绍基于强化学习的提示选择和轻量级测试程序验证组件的设计，以解决选择专门提示的第二个挑战。本小节首先介绍强化学习的简要背景，然后详细说明基于强化学习提示选择的具体设计。

（1）强化学习背景

强化学习（一种记忆性搜索算法）旨在指导智能体如何在环境中采取行动，以便在长期内最大化其累计奖励[131,132] 的过程。强化学习中的关键角色包括：1) 智能体：学习者和决策者的角色；2) 环境：由与智能体之外的事物构成并与其互动的所有事物；3) 行动：智能体采取的动作；4) 状态：智能体从环境中获得的信息；5) 奖励/惩罚：关于行动的来自环境的反馈。

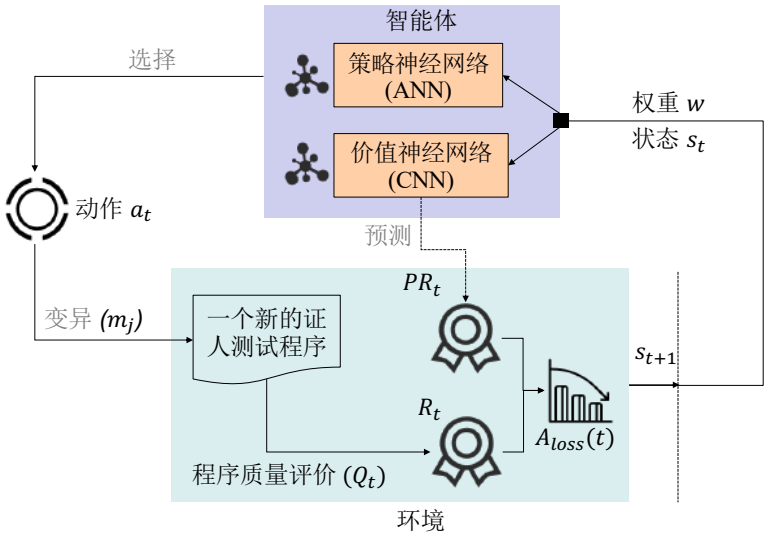
强化学习通常可被分类为基于价值的算法和基于策略的算法。基于价值的强化学习算法专注于优化价值函数，即通过估计状态或动作对未来回报的价值来间接学习策略（如 Deep Q-Learning 算法[133]）；而基于策略的算法直接优化策略，直接学习从状态到动作的映射以最大化预期回报（如 Policy Gradients 算法[134]）。两者在实践中各有优势：基于价值的方法在学习稳定性和处理离散动作空间上可能表现更佳，而基于策略的方法在处理高维、连续动作空间和学习随机策略上通常更为强大。在实际应用中，不同算法会根据问题的实际需求和特征进行适度调整。随着强化学习的进步，研究者已经提出了一类结合了基于价值和基于策略的策略元素的算法，如 Actor-Critic (AC) 系列算法[135]。因为 Advantage Actor-Critic（即 A2C[136]）算法与 AC[135] 相比既有效又适合单线程和多线程系统（与 Asynchronous Advantage Actor-Critic，即 A3C[136] 相比），所以，LLM4CBI 采用 A2C 框架来学习提示的效果以应对编译器缺陷定位的挑战，最终构造辅助编译器缺陷定位的的有效证人测试程序。总而言之，LLM4CBI 选择 A2C 框架是由于它在实际中的有效性、高效率和低方差的稳定性[137]。

（2）用于提示选择的强化学习

由于在给定的失败测试程序上随机应用变异的有效性可能有限[13]，因此，有必要采用更好的策略选择更加精确的提示辅助 LLMs 构造出有效的测试程序变种。换而言之，行之有效的方案需要在给定时间内针对特定的编译器缺陷有效地构造出数量较多且有效的证人测试程序。为了实现以上目标，LLM4CBI 采用强化学习策略，将其中构造的证人测试程序的质量作为提示的奖励指标，以制定对各个特定的缺陷选择专用且有效的提示。

图5.3描述了 LLM4CBI 中基于强化学习的提示选择策略的主要流程。总体而言，智能体利用 ANN（Actor Neural Network）来预测一个行动者 a_t（即用于变异程序的提示 m_j）。在从 LLMs 获取新测试程序后，环境根据新构造出的测试程序质量（Q_t）来计算实际回报 R_t。与此同时，智能体中的 CNN（Critic Neural Network）预测一个

潜在回报 PR_t。接下来，通过结合实际回报和潜在回报来测量优势损失 (A_{loss})。后续，ANN 和 CNN 的权重（w）及状态（s_t）将被更新至智能体，从而帮助智能体在 $t+1$ 时刻选择更精确的提示。



* PR_t: 时刻 t 的潜在回报
* R_t: 时刻 t 的实际回报
* A_{loss}: 时刻 t 的优势损失函数
* m_j: 被选中的变异提示 ($j \in \{1, 13\}$)

图 5.3　基于强化学习的提示选择过程

Fig. 5.3　Memorized prompt selection guided by reinforcement learning

具体而言，LLM4CBI 根据 A2C 框架，首先初始化两个神经网络：智能体中的策略神经网络 (ANN) 和价值神经网络 (CNN)。基于历史知识预测动作的概率分布，LLM4CBI 采用 ANN 选择一个最优的动作。CNN 用于预测执行所选动作后从当前状态到未来状态可以累积的潜在奖励。如此一来，LLM4CBI 通过一个动作 a_t 来选择一个提示（第一次随机选择）并测量新构造出的证人测试程序质量 (Q_t)。为了促进学习，LLM4CBI 采用了基于预测的潜在奖励（PR）和从所选提示获得的实际奖励（R）的优势损失函数（A_{loss}）。最后，LLM4CBI 更新所有的 ANN、CNN 和状态到智能体。LLM4CBI 重复选择提示构造新测试程序，直到终止条件（如达到 1 小时限制或构造出 10 个证人测试程序）。特别地，与现有的基于 A2C 的方法[135,136] 一致，LLM4CBI 中的 ANN 和 CNN 均配置单个隐藏层，以达到轻量级和快速收敛的效果。后续小节将详细说明该算法的关键部分，包括实际奖励和优势损失函数计算方法。

① 测量实际奖励（R）。衡量基于 A2C 方法的有效性的一个关键因素在于如何量化在应用某提示后产生的实际奖励。受到先前研究[12,13] 的启发，一组高质量的证人测试程序需要满足相似性和多样性的标准。相似性要求各个证人测试程序的编译器执行轨迹应与给定的失败测试程序相当。因此，遵循 SBFL 的原则，可以减少对大量与缺陷无关文件相关的怀疑。多样性要求不同的证人测试程序之间的编译器执行轨迹有

所不同，以减少对各种无缺陷文件的怀疑。通过以上方法，聚合一组经过变异的证人测试程序有助于通过规避偏差来有效定位真正存在缺陷的文件。和现有工作一样，相似性和多样性均依赖于所定义的 dist 度量，如下所示：

$$dist(a,b) = 1 - \frac{Cov_a \cap Cov_b}{Cov_a \cup Cov_b} \tag{5.4}$$

其中，$dist(a,b)$ 表示测试程序 a 和 b 之间的覆盖距离，该距离是根据计算 Jaccard 距离度量[12] 来确定的。Cov_a 和 Cov_b 分别表示测试程序 a 和 b 中编译器中覆盖的语句集合。此外，将证人测试程序集合的相似性和多样性度量形式化，具体如公式5.5和公式5.6所示：

$$sim = \frac{\sum_{i=1}^{N}(1 - dist(p_i, f))}{N} \tag{5.5}$$

$$div = \frac{\sum_{i=1}^{N-1}\sum_{j=i+1}^{N} dist(p_i, p_j)}{N(N-1)/2} \tag{5.6}$$

其中，公式中将 LLM4CBI 构造出的证人测试程序的集合表示为 $p = \{p_1, p_2, ..., p_n\}$，失败测试程序表示为 f。N 表示证人测试程序的数量。

在时间步 t，一旦构造出一个证人测试程序，LLM4CBI 通过在公式5.7中线性组合相似性和多样性的得分来量化当前证人测试程序集合的效果。

$$Q_t = n(\alpha \times div_t + (1-\alpha) \times sim_t) \tag{5.7}$$

其中，系数 α 表示在公式5.7中线性组合多样性和相似性的权重因子。此外，根据现有研究[13]，公式5.7还包括另一个系数 n，该系数表示证人测试程序集合的数量。

随后，LLM4CBI 根据与前一个时间步长（表示为 t−1）相比，它是否可以提高一组证人测试程序的整体质量来确定是否接受新构造出的证人测试程序。公式5.8计算了相对于前一个时间步骤质量增强的过程。

$$\triangle Q_t = Q_t - Q_{t-1} \tag{5.8}$$

然而，在各个状态中，仅允许选择一个提示来构造证人测试程序，而提示的性能可能因不同的缺陷而存在显著变化。为了平衡多样的提示性能影响，LLM4CBI 同时考虑了当前时间步骤的改进和与当前提示相关的历史累积改进，并将该合并值作为当前时间步骤获得的实际奖励，具体如下公式5.9所示：

$$R_t = \frac{\sum_{i=1}^{t} \triangle Q_i}{T(m_j)} \tag{5.9}$$

其中 $T(m_j)$ 表示选择采用提示 m_j（一个变异提示）变异失败测试程序的次数。如果第 i 个时间步骤选择的提示不是 m_j，则 $\triangle Q_i$ 被定义为零（$\triangle Q_i = 0$）；如果选择的提示是 m_j，则采用公式5.8计算 $\triangle Q_i$。关于是选择一个新的提示 m_j 还是以前选择的提示 m_j，该过程由采用智能体中的策略神经网络（ANN）作出决定。

② 计算优势损失（A_{loss}）。虽然实际奖励（R_t）是在当前时间步骤 t 获得的，但 LLM4CBI 还采用卷积神经网络（CNN）来预测潜在奖励（PR_{t+i}）。为了有效地考虑未来因素，A2C 引入了优势损失函数 A_{loss}。A_{loss} 函数旨在解决两个神经网络中高方差的问题，并防止收敛到局部最优解[136]。

优势损失函数由公式5.10进行计算：

$$A_{loss}(t) = \sum_{i=t}^{t+u}(\gamma^{i-t}R_i) + \gamma^{u+1}PR_{t+u} - PR_t \tag{5.10}$$

其中变量 u 表示 CNN 在预测潜在奖励时考虑未来的 u 个连续状态和动作。γ 代表分配给实际未来奖励的权重。PR_{t+u} 和 PR_t 表示由 CNN 确定的第 $(t+u)$ 和第 t 时间步骤的预测潜在奖励。值得注意的是，LLM4CBI 在一个时间步骤内重复该过程 u 次，以估计实际未来奖励。

利用优势函数在公式5.10中计算的损失，LLM4CBI 接着根据公式5.11更新 ANN 和 CNN 的权重。

$$w = w + \beta\frac{\partial(\log P_w(a_t|s_t)A_{loss}(t))}{\partial w} \tag{5.11}$$

其中，s_t 表示当前状态，a_t 表示相应的动作。$P_w(a_t|s_t)A_{loss}(t)$ 表示基于 ANN 和 CNN 参数 w 在状态 s_t 上执行动作 a_t 的概率。β 代表权重更新的学习率。

经过以上步骤，LLM4CBI 将不断构造新的证人测试程序。然而，某些证人测试程序并不是完全语义有效的，即可能存在未定义行为。因此，有必要设计一个新的组件验证新构造证人测试程序的有效性，并有效利用 LLMs 的反馈机制避免构造出含有重复错误的证人测试程序。下文将详细阐述 LLM4CBI 中的验证及反馈部分。

5.3.4 轻量化的测试程序验证

证人测试程序的语义有效性也会影响定位方法的效果，但现有的编译器缺陷定位研究很少关注变异后测试程序的语义有效性。证人测试程序有效性影响定位效果的主要原因有三个方面。首先，现有方法构造出的证人测试程序可能包含未定义的行为（如在第5.5.2节中的评估结果所示，此类测试程序会带来副作用从而降低编译器缺陷定位的有效性）。其次，现有方法可能会构造出不存在测试预言的测试程序，该类程序也会影响缺陷定位的有效性。第三，现有的研究对在变异过程中所犯的编程错误没有关注，因此它们在变异失败测试程序时经常犯同样的错误，如产生包含相同语法错

误或者相同未定义行为的证人测试程序，从而影响缺陷定位的效率。

为了解决现有方法的局限性，LLM4CBI 设计了一个基于形式化分析技术的验证组件来过滤掉语义无效的测试程序，并采用测试断言验证来修复违反测试断言的测试程序。此外，LLM4CBI 收集了 LLMs 构造的测试程序中的所有验证错误，并将该错误信息作为反馈提示作为 LLMs 的输入，以避免 LLMs 构造含有重复相同的编程错误的证人测试程序。接下来的小节将详细介绍验证过程。

（1）程序语义验证

语义验证的主要任务是对新构造出的可能包含任何未定义行为的测试程序进行静态形式化分析。与动态分析方法相比，基于形式化分析的静态检查被证明是轻量级的。如图5.2中的步骤 ❺ 所示，新构造出的测试程序被传递到此组件以进行语义验证。基于 Frama-C[138] 分析能力及给出的断言信息，LLM4CBI 检测测试程序中五种不同类别的未定义行为，分别为：

① memem_access 断言。无效的内存访问，如越界读取或越界写入。

② shift 断言。用于右移或左移操作的无效 RHS（Right-Hand Shift，即右手操作数）或 LHS（Left-Hand Shift，即左手操作数）操作数。

③ index_bound 断言。访问数组的越界索引操作。

④ initialization 断言。访问未初始化的左值，即使用未初始化的变量。

⑤ division_by_zero 断言。除 0 操作。

如果检测到上述任何一种未定义行为，语义错误信息将作为反馈提示更新到 LLMs，以避免 LLMs 产生相同的错误。例如，如果新测试程序包含未定义的 division_by_zero 行为，LLM4CBI 会在步骤 ❽ 中向 LLMs 提供以下额外的反馈提示：“以上程序包含未定义的行为 *division_by_zero*，请不要再次生成包含该错误的测试程序”。

（2）测试预言验证

除了以上未定义行为的验证，还需要检查构造出的测试程序测试预言是否被破坏，即是否通过或失败[12,13,71,72]。根据编译器缺陷的类型（即崩溃和错误代码缺陷）。与现有工作[12,13] 类似，LLM4CBI 考虑了两种类型的测试断言：其一，对于崩溃缺陷（即在采用某些编译选项编译测试程序时，编译器崩溃），测试预言是采用相同的编译选项编译构造的测试程序时，编译器是否仍然会崩溃。其二，对于错误代码缺陷（即编译器在没有任何失败消息的情况下错误地编译了测试程序，导致测试程序在不同的编译选项下具有不一致的执行结果），测试预言编译并执行新构造出的测试程序时，执行结果是否和失败测试程序一致。与 DiWi[12] 和 RecBi[13] 类似，LLMs 可能会构造出被修改或者删除测试预言的测试程序，因此 LLM4CBI 采用类似的验证来检查是否缺少测试预言。具体而言，LLM4CBI 检查测试程序是否包含与失败测试程序相同数量的 abort 或 printf 语句。

如果新构造的测试程序通过了上述验证，且不能触发编译器缺陷，则在步骤❻中采用包含某缺陷的编译器编译所构造出来的证人测试程序，并收集相应的编译器覆盖率信息。否则，LLM4CBI 收集语义错误或测试预言信息，并将其用作反馈提示，以指导 LLMs 不要构造出包含相同类型错误的测试程序。值得注意的是，由于不同的失败测试程序可能包含不同类型的缺陷，因此针对不同的缺陷其反馈提示有所不同。为了提高缺陷定位的有效性及效率，故在 LLM4CBI 中设计并支持针对不同缺陷的特定提示选择部件是有必要的。

基于程序频谱的可疑文件排序。和先前的研究[12,13]一样，LLM4CBI 采用基于谱缺陷定位（SBFL）的方法来通过比较失败和证人测试程序在❾中的覆盖情况，以识别潜在的存在缺陷的编译器文件。具体而言，由于 Ochiai[23] 性能表现较好，LLM4CBI 采用 Ochiai 公式5.12计算各个语句的可疑分数。

$$score(s) = \frac{ef_s}{\sqrt{(ef_s + nf_s)(ef_s + ep_s)}} \tag{5.12}$$

公式中的 ef_s 和 nf_s 分别表示执行和不执行语句 s 的失败测试数，ep_s 表示执行语句 s 的证人测试程序数量。在 LLM4CBI 中，由于只有一个给定的失败测试程序，执行语句 s 的失败测试数 ef_s 固定为 1。此外，LLM4CBI 仅关注由给定的失败测试程序执行的语句，表明不执行语句 s 的失败测试数 nf_s 为 0。因此，Ochiai 公式可简化为公式5.13。

$$score(s) = \frac{1}{\sqrt{(1 + ep_s)}} \tag{5.13}$$

一旦获得了各个语句的可疑分数，LLM4CBI 继续计算各个编译器文件的可疑分数。类似于之前的研究，LLM4CBI 将给定失败测试程序执行语句的可疑分数在编译器文件内进行聚合，以确定文件级别的可疑分数。

$$SCORE(f) = \frac{\sum_{i=1}^{n_f} score(s_i)}{n_f} \tag{5.14}$$

其中，失败测试程序在编译器文件 f 中执行的语句数表示为 n_f。LLM4CBI 根据该信息计算各个编译器文件的可疑分数。通过根据其可疑分数以降序排列编译器文件，LLM4CBI 最终得到一个排名列表，每个文件均对应一个可以度分数，排名越靠前分数越大，表明其存在缺陷的可能性越大。

5.4 实验设计

本章详细介绍用于评估 LLM4CBI 有效性的实验方案设计，包括数据集与实验平台、研究问题、方法实现与实验设置、实验结果分析以及相关讨论。

5.4.1 数据集与实验平台

① 数据集与研究对象。实验采用包含 120 个真实编译器缺陷的失败测试程序作为基准数据集，其中有 60 个是 GCC 缺陷，60 个是 LLVM 缺陷。该 120 个缺陷包含先前工作[12,13] 中研究的所有编译器缺陷。具体而言，各个缺陷数据集中附有缺陷的详细信息，包括存在缺陷的编译器版本、导致失败测试程序、缺陷的编译选项以及存在缺陷文件，该已标记的存在缺陷的文件作为实验中的基准（即 ground truth）。关于研究对象，和前两章的测试对象一样，本章选择对 GCC 和 LLVM 两个成熟编译器中存在的缺陷进行定位。由于两个编译器在现有文献中被广泛应用[10,12,13,71,72]，因此可以说明实验评估的全面性。从编译器的代码规模而言，GCC 编译器包含 1,758 个文件，1,447K 行源代码，而 LLVM 编译器包含 3,265 个文件，1,723K 行源代码。

② 实验平台。本章所有实验在一台配备 12 核 CPU 的台式机上进行，该机器配置为 Intel(R) Xeon(R) W-2133 CPU @ 3.60GHz，64G RAM，运行 Ubuntu 18.04 操作系统，不支持 GPU。

5.4.2 研究问题

本章的实验设计旨在评估以下主要研究问题（Research Questions，简称 RQs）：

① RQ1：LLM4CBI 是否能够在编译器缺陷定位效果和效率上优于现有的先进方法（即 DiWi[12] 和 RecBi[13]）?

RQ1 旨在评估 LLM4CBI 在编译器缺陷定位方面是否优于现有方法。具体而言，RQ1 评估 LLM4CBI 是否能在相同运行时间和构造相同数量证人测试程序的两种情况下均优于现有方法。

② RQ2：三个新设计的组件，即精确提示制定、基于强化学习的提示选择和轻量级测试程序验证，是否能够对 LLM4CBI 产生积极的贡献？

RQ2 旨在评估 LLM4CBI 中新提出的三个组件是否可以有效地促进 LLM4CBI 的编译器缺陷检测能力。

③ RQ3：是否可以将其他 LLMs 轻松扩展至 LLM4CBI 以辅助编译器缺陷定位？

RQ3 旨在评估 LLM4CBI 的可扩展性，具体评估将其他 LLMs 扩展至 LLM4CBI 所需要的人力消耗情况及其缺陷定位的效果。

5.4.3 方法实现与实验设置

（1）LLM4CBI 的实现

LLM4CBI 是由 OClint[1]（v22.02）、srcSlice[130]（v1.0）、Gcov[2]（v4.8.0）和 PyTorch[3]

[1]https://github.com/oclint

[2]https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[3]https://pytorch.org

（v1.10.1+cu113）实现的。其中，OClint 用于计算各个语句的圈复杂度；srcSlice 用于获取程序中定义和使用的变量的数据流复杂度；Gcov 用于收集编译器覆盖信息；PyTorch 用于支持 A2C 框架。对于 A2C 的具体实现，实验中的超参数设置和先前研究[13] 相同。对于测试程序验证组件的实现，LLM4CBI 采用 Frama-C[138]（Phosphorus-20170501）来检查测试程序的语义有效性，并编写 Python 脚本（Python 3.8.5）来检查是否存在违反测试预言的测试程序。LLM4CBI 默认采用 GPT-3.5 作为 LLM4CBI 的内置 LLMs，其中在 GPT-3.5 中设置参数 temperature 为 1.0（关于 temperature 设置的讨论详见第5.6节）。

（2）RQ1 实验设置

① 对比方法。在回答 RQ1 时将 DiWi[12] 和 RecBi[13] 作为 LLM4CBI 的对比算法。

② 实验运行设置为研究 RQ1，本实验采用以下两种实验设置：

1）设置-1：设置各个方法运行固定时长一个小时。该设置是现有编译器缺陷定位研究中常用的标准对比策略[12,13,71,72]。

2）设置-2：设置各个方法在构造固定数量（即 10 个）的证人测试程序后终止。此设置旨在进一步展示 LLM4CBI 的效率。基于现有的经验研究[23]，本实验选择了 10 作为确定的数量。此外，若某种方法在 2 小时内不能构造出目标数量的证人测试程序，则强制对其进行终止。增加此设置的原因是在设置-1 的实验中，发现各个对比方法构造出的证人测试程序数量存在一定差异，LLM4CBI 通常可以构造出更多的证人测试程序，所以增加该实验设置以进一步评估 LLM4CBI 的优越性能是来源于更多数量的证人测试程序还是新构造出的证人测试程序质量。

对于以上两种设置，本实验重复运行所有比较方法三次，并计算 Top-N、MFR 和 MAR 指标的中位数结果，以减少随机性的影响。此外，设置-2 还比较了每种方法的执行时间以及 LLM4CBI 所取得的速度提升效果。

（3）RQ2 设置

① 对比方法。RQ2 采用以下几个 LLM4CBI 的变种方法对 LLM4CBI 主要组成部分的有效性进行评估：

1）LLM4CBI(EP) 采用简单的提示（即 Existing Prompts），不进行数据流和控制流分析。换而言之，LLM4CBI(EP) 只保留 ＜变异规则＞，而不依赖＜变量＞ 和 ＜位置＞信息来填充提示模版。

2）LLM4CBI(SP) 将特殊提示（即 Special Prompts），即 “最复杂的数据和控制流”，来替代数据流和控制流分析。具体而言，LLM4CBI(SP) 将＜变量＞替换为 “最复杂的变量”，并将＜位置＞替换为 “在最复杂的语句中”，该提示采用 LLMs 的程序理解能力完成填充提示模版。

3）LLM4CBI(Rand) 在选择提示过程中采用随机的方式选择一个提示（即 Random Selection），而不采用新设计的基于强化学习的提示选择。

4）LLM4CBI(SelNoV) 表示在 LLM4CBI 的提示选择过程中删除了测试程序验证组件的变体（即 Selection with No Validation），即 LLM4CBI(SelNoV) 不关心编译器缺陷定位的测试程序有效性。

通过设计以上变种，LLM4CBI(EP) 和 LLM4CBI(SP) 旨在调查新的程序复杂度指导提示生产模式是否有效，而 LLM4CBI(Rand) 和 LLM4CBI(SelNoV) 则分别旨在了解基于强化学习的提示选择和测试程序验证是否能为 LLM4CBI 作出贡献。

② 实验运行设置。本评估实验按照 RQ1 的设置-1 的策略运行四种变种。然后，通过比较每种方法的 Top-N、MFR 和 MAR 指标，阐明该组件的贡献。

表 5.2　LLM4CBI 中评估的开源大语言模型列表
Tab. 5.2　Evaluated open-source LLMs in LLM4CBI

大语言模型	参数大小	发布日期	受欢迎程度 (GitHub)
Alpaca[126]	7B	March 2023	25.0K star
Vicuna[127]	7B	March 2023	22.8K star
GPT4ALL[128]	13B	March 2023	46K star

（4）RQ3 设置

① 对比方法。RQ3 评估了三个最具代表性和受欢迎的 LLMs。表5.2中展示了本研究中评估的开源 LLMs，其中 “参数大小” 列反映了模型的参数规模（以十亿为单位），“发布时期” 列显示了 LLMs 发布的时间，“受欢迎程度” 列则表示截至 2023 年 6 月 1 日的 GitHub 关注数。RQ3 选择了上述 LLMs 的理由如下：1）它们受欢迎程度高，star 数量在几个月内上涨至 20k+；2）它们在代码生成任务中被证明是有效的[111,125,126,128]。基于所选的 LLMs，通过将新的 LLMs 替换 LLM4CBI 中的 LLMs，RQ3 设计了三种新的变体方法，即 LLM4CBI(Alpaca)、LLM4CBI(Vicuna) 和 LLM4CBI(GPT4ALL)，来进一步回答该研究问题。

对于三种 LLMs 的实现，本文从 HuggingFace 网站下载了 GGML[4]格式的模型，并采用 llama.cpp[5]的 python 包 llama-cpp-python[6]将它们运行在只支持 CPU 的机器上。具体而言，本文采用了一个由 llama-cpp-python 支持的 Web 服务器，该服务器旨在作为 OpenAI API 的替代方案。该工具的支持使得用户能够将与 llama.cpp 兼容的模型与任何 OpenAI 支持的客户端（如语言库、服务等）一起运行而不用考虑模型的大小和较高的硬件需求（如 GPU 支持）。需要注意的是，在 LLM4CBI 中扩展其他更强大的模型较易，实验证明需要的人力投入较小。用户唯一的工作是需要为 llama-cpp-python

[4]https://github.com/ggerganov/ggml
[5]目标是在仅支持 CPU 的计算机上采用 4 位整数量化来运行 LLMs。
[6]https://github.com/abetlen/llama-cpp-python

Web 服务器提供 GGML 格式的模型，该模型可以直接从 HuggingFace 网站下载，也可以采用 GGML 主页上详细且得到积极维护的教程进行转换。

② 对比策略。RQ3 在一个小时的时间限制内运行以上三种变体方法，然后比较每种方法的 Top-N 指标。因为本实验的目标是说明 LLM4CBI 替换不同 LLMs 的可扩展性，所以各个模型在实验中只运行一次。

5.4.4 评价指标

每种用于编译器缺陷定位的方法均会生成一个可疑编译器文件的列表。为了评估每种方法的效果，实验中采用基准来测量各个错误文件在排序列表中的位置。根据先前研究工作[139,140]，当存在多个编译器文件具有相同可疑得分的情况时，说明该方法不能很好地区分存在缺陷的文件，故 LLM4CBI 将其排在列表最后。具体而言，实验中计算了在编译器缺陷定位研究中常用的以下指标。

① Top-N。该指标表示在前 N 个位置内有效定位和包含的缺陷数，其中 N 是研究中指定的 {1，5，10，或 20} 之一。较大的 Top-N 值表示方法的定位效果更好。

② 首次排名均值（Mean First Ranking，简称 MFR）。该指标表示存在缺陷的排名列表中第一个有问题文件的平均排名。MFR 的目标是用来评估定位缺陷的速度，以加速调试过程。较小的值更好，表示开发人员能够尽快定位相应的缺陷。

③ 平均排名均值（Mean Average Ranking，简称 MAR）。该指标测量存在缺陷的排名列表中各个有问题文件平均排名的平均值。MAR 指标旨在准确定位所有有问题的元素。与 MFR 类似，较小的 MAR 值表示方法性能更好。

5.5 实验结果分析

5.5.1 和现有方法对比分析

（1）设置-1 的实验结果分析

评估 RQ1 设置 1 的比较结果如表5.3所示。第一列表示测试编译器对象，第二列显示不同的方法（即 DiWi、RecBi 以及 LLM4CBI）。第 3-10 列记录了从每种方法的三次迭代中获得的中值得出的 Top-N 指标，包括 Top-N 的数量（即 Top-N 数）和 LLM4CBI 对比较方法的提高（即 $\Uparrow_{Top-N}$ (%)）。第 11-14 列表示 MFR 和 MAR 指标，以及 LLM4CBI 在 MFR 和 MAR 指标上实现的改进。

从表中可知，LLM4CBI 分别在 Top-1、Top-5、Top-10 和 Top-20 文件中成功定位 21、50、74 和 98 个编译器缺陷（GCC 和 LLVM 中总共 120 个编译器缺陷），展示了 LLM4CBI 缺陷定位的能力。特别地，在定位 Top-1、Top-5、Top-10 和 Top-20 文件中，LLM4CBI 比 DiWi 提高了 17.50%、41.67%、61.67% 和 81.67%，比 RecBi 提高了 17.50%、41.67%、61.67% 和 81.67%。本文进一步对不同主题编译器效果的进行了对

表 5.3　LLM4CBI 与两种先进方法的对比实验结果（RQ1, 设置-1）

Tab. 5.3　Compiler bug isolation effectiveness comparison with two state-of-the-art approaches (under Setting-1 in RQ1)

编译器	对比方法	Num. Top-1	⇑$_{Top-1}$ (%)	Num. Top-5	⇑$_{Top-5}$ (%)	Num. Top-10	⇑$_{Top-10}$ (%)	Num. Top-20	⇑$_{Top-20}$ (%)	MFR	⇑$_{MFR}$ (%)	MAR	⇑$_{MAR}$ (%)
GCC	DiWi[12]	7	57.14	19	36.84	32	28.13	40	25.00	23.44	36.11	23.88	33.88
	RecBi[13]	8	37.50	23	13.04	35	17.14	42	19.05	19.33	22.56	19.90	20.65
	LLM4CBI	11	-	26	-	41	-	50	-	14.97	-	15.79	-
LLVM	DiWi[12]	4	150.00	18	33.33	27	22.22	40	20.00	26.83	44.84	26.90	44.80
	RecBi[13]	6	66.67	21	14.29	29	13.79	44	9.09	25.32	41.55	25.63	42.06
	LLM4CBI	10	-	24	-	33	-	48	-	14.80	-	14.85	-
ALL	DiWi[12]	11	90.91	37	35.14	59	25.42	80	22.50	25.13	40.77	25.39	39.66
	RecBi[13]	14	50.00	44	13.64	64	15.63	86	13.95	22.33	33.33	22.77	32.70
	LLM4CBI	21	-	50	-	74	-	98	-	14.89	-	15.32	-

表 5.4　LLM4CBI 与两种先进方法的对比实验结果（RQ1, 设置-2）

Tab. 5.4　Compiler bug isolation effectiveness comparison with two state-of-the-art approaches (under Setting-2 in RQ1)

编译器	对比方法	Num. Top-1	⇑$_{Top-1}$ (%)	Num. Top-5	⇑$_{Top-5}$ (%)	Num. Top-10	⇑$_{Top-10}$ (%)	Num. Top-20	⇑$_{Top-20}$ (%)	MFR	⇑$_{MFR}$ (%)	MAR	⇑$_{MAR}$ (%)
GCC	DiWi[12]	6	33.33	20	20.00	33	6.06	41	7.32	22.15	28.13	22.18	26.38
	RecBi[13]	7	14.29	22	9.09	34	2.94	42	4.76	18.88	15.68	19.21	14.99
	LLM4CBI	8	-	24	-	35	-	44	-	15.92	-	16.33	-
LLVM	DiWi[12]	5	60.00	19	26.32	29	20.69	39	10.26	26.37	37.62	26.43	37.27
	RecBi[13]	7	14.29	21	14.29	34	2.94	40	7.50	19.43	15.34	19.76	7.43
	LLM4CBI	8	-	24	-	35	-	43	-	16.45	-	16.58	-
ALL	DiWi[12]	11	45.45	39	23.08	62	12.90	80	8.75	24.26	33.29	24.31	32.30
	RecBi[13]	14	14.29	43	11.63	68	2.94	82	6.10	19.16	15.51	19.49	15.55
	LLM4CBI	16	-	48	-	70	-	87	-	16.19	-	16.46	-

注：列 “⇑ ∗” 表示 LLM4CBI 在各种指标上相较于对比方法的提升率（%）。

比，结果发现，尽管与 GCC 相比，LLVM 中的编译器文件数量更多，但在 GCC 的情况下，LLM4CBI 表现出略好的效果。例如，LLM4CBI 对 GCC 的 MFR 和 MAR 值分别为 14.97 和 15.79，而 LLVM 的相应值为 14.89 和 15.32。

与 DiWi 和 RecBi 相比，在 GCC 和 LLVM 编译器上的评估结果显示 LLM4CBI 在所有指标均优于 DiWi 和 RecBi。值得注意的是，LLM4CBI 在前 1 名和前 5 名方面比 DiWi 有 90.9% 和 35.14% 的显著改善。对于 RecBi，就 Top-1 和 Top-5 指标而言，LLM4CBI 可以比 RecBi 定位 50.00% 和 13.64% 的缺陷。值得一提的是，先前的研究强调了 Top-5 指标的实际意义，表明如果将存在缺陷的文件无法定位在 Top-5 位置内，开发人员通常不愿意尝试采用该自动调试工具。此外，在 MFR 和 MAR 方面，LLM4CBI 分别比 DiWi 提高了 40.77% 和 39.66%，比 RecBi 提高了 33.33% 和 32.73%。MFR 和 MAR 的改进表明，LLM4CBI 可以及时准确地定位比 DiWi 和 RecBi 更多的编译器缺陷。总体而言，与 DiWi 和 RecBi 相比，LLM4CBI 能够大幅提高编译器缺陷定位的有效性，特别是与 Top-1/5 指标，表现出更好的编译器缺陷定位能力。

（2）设置-2 的实验结果分析

表5.4列举了 RQ1 设置-2 的实验结果。总体而言，对于所有 120 个 GCC 和 LLVM 编译器缺陷，可以看出 LLM4CBI 在 Top-N 上的性能也优于两种对比方法，且 LLM4CBI 可以定位更多的缺陷并保持最低的 MFR 和 MAR。

对于效率方面，图5.5展示了三种方法更为具体的性能比较结果。图5.5中的 x 轴显示了 GCC 和 LLVM 中的不同方法，y 轴表示定位各个缺陷消耗的时间。从图中的数字中可以观察到，LLM4CBI 定位大多数缺陷消耗的时间更少。此外，还可以看出定位 LLVM 编译器缺陷消耗的时间通常比定位 GCC 编译器缺陷多。产生以上结果的原因是 LLVM 的文件数量比 GCC 大的多，文件越多意味着计算覆盖范围所花费的时间越多，故该实验结果是合理的。

为了进一步评估 LLM4CBI 在性能方面的优势，实验还计算了 LLM4CBI 对现有方法实现的加速比。具体而言，按照下面的公式来计算加速比，加速比数值越高表明加速的效果越明显：

$$\frac{T_{baseline} - T_{\text{LLM4CBI}}}{T_{baseline}} \times 100 \tag{5.15}$$

其中 $T\_baseline$ 表示基准对比算法（即 DiWi 或者 RecBi）平均构造 10 个证人测试程序消耗的时间，T_LLM4CBI 描述研究者提出的 LLM4CBI 构造相同数量的证人测试程序消耗的时间。图5.4显示了 LLM4CBI 在性能方面和对比方法的比较结果。从图中可以发现，和 DiWi 和 RecBi 相比，LLM4CBI 能够在 GCC 和 LLVM 上分别实现 53.19% 和 64.54%，以及 47.31% 和 63.19% 的加速比，充分证实了 LLM4CBI 能够高效地构造出证人测试程序，能够进一步提高编译器缺陷定位效率。

值得注意的是，在 RQ1 的实验中，设置-1 与设置-2 相比，LLM4CBI 在设置-1 中

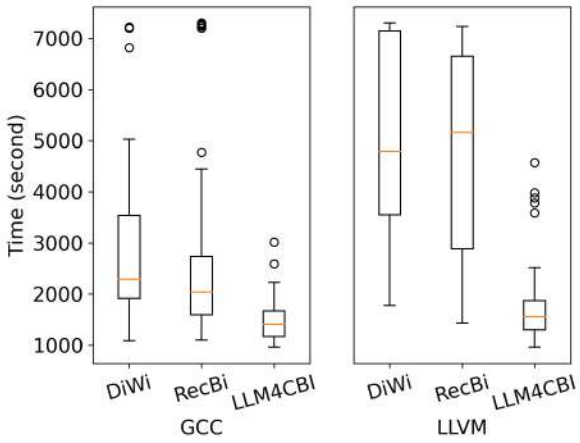


图 5.4　LLM4CBI 和 DiWi 及 RecBi 运行时间分布情况（RQ1 中的设置-2）

Fig. 5.4　Execution time comparison over DiWi, RecBi, and LLM4CBI (under Setting-2 in RQ1)

的总体结果更好，即能够将更多的缺陷定位在 Top-1/5 的文件中且保持较低的 MFR 和 MAR 指标。产生上述结果的原因是 LLM4CBI 在设置-1（即运行一个小时）中可以构造出更多的证人测试程序，从而提高了缺陷定位的有效性。此外，根据 RQ1 的实验结果，为了获得更好的编译器缺陷定位结果，减轻编译器调试难度，作者建议用户运行 LLM4CBI 足够长的时间。

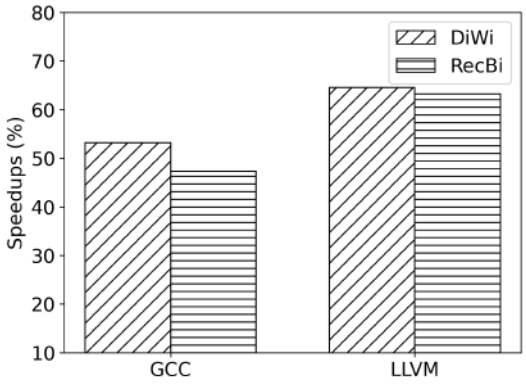


图 5.5　LLM4CBI 对 DiWi 及 RecBi 加速比统计（RQ1 中的设置-2）

Fig. 5.5　Speedup comparison over DiWi, RecBi, and LLM4CBI(under Setting-2 in RQ1)

为了进一步验证 LLM4CBI 在统计学上的有效性，对设置一和设置二实验数据进行了可信度为 0.05 的曼-惠特尼 U 检验[90]（即 Mann-Whitney U-test），其中零假设为“LLM4CBI 和比较方法之间没有显著差异”，备择假设为“LLM4CBI 和比较方法之间存在显著差异”。计算结果 P-值（P-value）均小于或等于 0.05，表明备择假设成立，即 LLM4CBI 的缺陷定位能力优于其他对比方法的实验结论是具有统计显著性的。

对 RQ1 的回答：在两种不同的实验配置中 LLM4CBI 均明显优于两种最先进的方法，即 LLM4CBI 比 DiWi 和 RecBi 更有效且高效地构造出有效的证人测试程序。

5.5.2 新设计组件的有效性分析

表5.5呈现了 LLM4CBI 与比较变种方法之间的比较结果，其中每列和每行的意义与表5.3或者表5.4相同。下面将分别分析各个组件对 LLM4CBI 的贡献。

① 提示制定组件的贡献。如表5.5所示，在 GCC 和 LLVM 中所有 120 个缺陷中，对所有指标（Top-N、MFR 和 MAR）而言，LLM4CBI 优于 LLM4CBI(EP)（即该方法采用没有数据流和控制流分析的简单提示）和 LLM4CBI(SP)（即该方法通过提供 “最复杂的数据和控制流” 来替换提示模版中的变量和位置）。值得注意的是，LLM4CBI 不仅在 Top-1 和 Top-5 指标上比 LLM4CBI(EP) 提升 40.00% 和 61.54%，并且在 MFR 和 MAR 方面也比 LLM4CBI(EP) 提高了 18.66% 和 17.99%。与 LLM4CBI(SP) 相比，LLM4CBI 在 Top-N 指标上同样显示出更好的结果。此外，LLM4CBI 还具有较小的 MFR 和 MAR 值，表明 LLM4CBI 具有更好的编译器缺陷定位能力。

上述结果清楚地表明，本章提出的程序复杂性指标有助于制定精确的提示，删除数据流和控制流分析或将其替换为对 LLMs 的显式描述是无效的。因为 LLMs 已被证明在语义理解方面受到限制[141]，所以该实验结果是合理的。因此，在提示描述有限的情况下，LLMs 在变异失败测试程序时可能会随机选择有效的变量和位置，从而难以将执行结果从失败翻转为通过，以构造出有效的证人测试程序辅助编译器缺陷定位。对比之下，基于静态程序分析的提示制定可以选择最佳的变量和位置进行变异操作，且能够提升缺陷定位的有效性。

② 提示选择组件的贡献。实验中运行 LLM4CBI(Rand)（即一个随机选择提示的方法）和 LLM4CBI，以研究基于强化学习提示选择组件的贡献。表5.5呈现了总体实验结果。结果表明 LLM4CBI 也优于 LLM4CBI(Rand)。具体而言，与 LLM4CBI(Rand) 相比，LLM4CBI 在 Top-1、Top-5、Top-10、Top-20、MFR、和 MAR 指标上分别提高了 50.00%、8.70%、15.63%、18.07%、14.03% 和 13.08%。以上结果不仅凸显了本章提出的基于强化学习提示选择相对于随机策略的优越性，而且证实了该组件对 LLM4CBI 编译器缺陷定位能力的积极贡献。

③ 程序验证组件的贡献。通过对表5.5第 4 和 5 行所示的方法进行对比，可以发现，在 GCC 和 LLVM 上，LLM4CBI 在所有的指标上均优于 LLM4CBI(SelNoV)（一个移除测试验证组件的变种方法）。具体而言，与 LLM4CBI(SelNoV) 相比，LLM4CBI 在 Top-1 和 Top-5 文件中分别能够定位比 LLM4CBI(SelNoV) 多 40.00% 和 13.64% 的缺陷。在 MFR 和 MAR 方面，LLM4CBI 也相对于 LLM4CBI(SelNoV) 分别提高了 17.14% 和 32.32%。以上结果有力地证明了引入测试程序验证组件提高了 LLM4CBI 编译器缺

表 5.5　LLM4CBI 与四种变体方法的对比实验结果

Tab. 5.5　Compiler bug isolation effectiveness comparison with four variants of LLM4CBI

编译器	对比方法	Num. Top-1	⇑$_{Top-1}$ (%)	Num. Top-5	⇑$_{Top-5}$ (%)	Num. Top-10	⇑$_{Top-10}$ (%)	Num. Top-20	⇑$_{Top-20}$ (%)	MFR	⇑$_{MFR}$ (%)	MAR	⇑$_{MAR}$ (%)
GCC	LLM4CBI(EP)	7	57.14	24	8.33	34	20.59	42	19.05	17.62	15.04	18.28	13.62
	LLM4CBI(SP)	6	83.33	23	13.04	33	24.24	40	25.00	19.12	21.71	19.67	19.73
	LLM4CBI(Rand)	6	83.33	23	13.04	33	24.24	42	19.05	18.23	17.88	18.73	15.70
	LLM4CBI(SelNoV)	8	37.50	22	18.18	35	17.14	40	25.00	19.38	22.76	19.64	19.60
	LLM4CBI	11	-	26	-	41	-	50	-	14.97	-	15.79	-
LLVM	LLM4CBI(EP)	8	25.00	20	20.00	31	6.45	43	11.63	18.98	22.02	19.08	22.17
	LLM4CBI(SP)	7	42.86	23	4.35	32	3.13	39	23.08	17.75	16.62	17.88	16.95
	LLM4CBI(Rand)	8	25.00	23	4.35	31	6.45	41	17.07	16.40	9.76	16.52	10.11
	LLM4CBI(SelNoV)	7	42.86	22	9.09	31	6.45	40	20.00	16.55	10.57	25.63	31.92
	LLM4CBI	10	-	24	-	33	-	48	-	14.80	-	14.85	-
ALL	LLM4CBI(EP)	15	40.00	44	13.64	65	13.85	85	15.29	18.30	18.66	18.68	17.99
	LLM4CBI(SP)	13	61.54	46	8.70	65	13.85	79	24.05	18.44	19.26	18.78	18.40
	LLM4CBI(Rand)	14	50.00	46	8.70	64	15.63	83	18.07	17.32	14.03	17.63	13.08
	LLM4CBI(SelNoV)	15	40.00	44	13.64	66	12.12	80	22.50	17.97	17.14	22.64	32.32
	LLM4CBI	21	-	50	-	74	-	98	-	14.89	-	15.32	-

注：列 “⇑ ∗” 表示 LLM4CBI 在各种指标上相较于其他变体方法的提升率（%）。

```
1  int a,b,c,d; char e;
2  void foo () { a = 0; }
3
4  int main () {
5    unsigned char f;
6    for (; b < 1; b++)
7    {
8      for (e = 1; e >= 0; e--){
9      d = 0;
10     if (a) { break; }
11     f = 179 * e;
12     c = f « (-1);    // c = f << 1;
13     foo ();
14     }
15   }
16   printf ("%d\n", c);
17   return 0;
18 }
```

(a) LLVM Bug #18000: 由第 12 行变异操作导致在同一行 “invalid shift value” 的未定义行为

```
1  int a;
2  short int b = 1; // int b = 1;
3
4  int main (){
5    int i;
6    for (i = 0; i < 56; i++)
7    {
8      for (; a; a--)
9      {;}
10   }
11   int c = &b;
12   if (*c)
13   {
14     *c = 1 % (unsigned int) *c | 5;
15   }
16   printf ("%d\n", b);
17   return 0;
18 }
```

(b) GCC Bug #64682：由第 2 行变异操作导致在第 11 行 “invalid memory access” 的未定义行为

图 5.6　含有未定义行为的证人测试程序对缺陷定位的影响

Fig. 5.6　Side effects of the test programs with undefined behaviors

陷定位的效果，从而证实了该组件在 LLM4CBI 中的积极贡献。

为进一步理解过滤语义上无效的证人测试程序的重要性，本文通过图5.6中展示的两个带有未定义行为证人测试程序示例指出其对编译器缺陷定位的不良影响。图5.6(a)中的第一个代码示例包括非法 shift（第 12 行）的未定义行为，该行为是由替换常数值（从 “1” 到 “-1”）产生的。由于未定义行为的干扰，RecBi 将此文件排在第 23 位。第二个示例揭示了 memem_access 的未定义行为，该行为是由第 2 行改变变量的类型修饰符（从 int 到 short int）引发的。由于 short int（64 位系统中为 2 字节）的大小小于 int（64 位系统中为 4 字节），所以对变量 b 的读取会导致越界访问的未定义行为。该示例使得 RecBi 将此缺陷排在第 28 位。采用测试程序验证组件后，LLM4CBI 在对可疑文件进行最终排序之前过滤了该类证人测试程序，结果显示 LLM4CBI 将 LLVM 缺陷和 GCC 缺陷分别排在第 9 位和第 5 位。如先前的研究[2,18] 所述，如果程序在语义上是无效的，运行编译后的二进制代码可能会产生不可预估的后果。总之，实验证明语义无效的证人测试程序会对编译器缺陷定位的有效性产生不良影响，及时过滤该类证人测试程序可以帮助 LLM4CBI 提升编译器缺陷定位的有效性。

对 RQ2 的回答： LLM4CBI 中设计的精确提示制定、基于强化学习的提示选择和轻量化的测试程序验证等三个组件，均有助于提高 LLM4CBI 缺陷定位的有效性。

5.5.3 框架可扩展性分析

表5.6展示了三个新设计的分别采用不同LLMs的变体方法（即LLM4CBI(Alpaca)、LLM4CBI(Vicuna)和LLM4CBI(GPT4ALL)）对编译器缺陷定位的结果。总体而言，结果显示所有变体均对编译器缺陷定位任务做出了贡献。具体来说，不同的LLMs在构造用于编译器缺陷定位的证人测试程序方面占先出了不同的定位能力，而GPT-3.5（在LLM4CBI中用作默认模型）在与其他LLMs相比的性能方面表现最好。为了充分理解大语言模型的能力，本文还分析了LLMs产生不同结果的原因。具体而言，其他LLMs的表现不如LLM4CBI中采用的GPT-3.5的原因主要有三个，分别是：

① 仅给出建议性的回答。有些大语言模型的输出结果只是建议性的回答，告诉用户如何更改代码，而不是输出实际的代码。例如，在对LLVM#17388进行缺陷定位时，采用提示“请将变量列表{a，c}中的二元运算符替换为另一个有效的运算符”时，LLM4CBI(Alpaca)的回答是“您可以将二元运算符‘||’替换为逻辑运算符‘&&’”。显然，该输出只给出了具体的建议但并未直接包含完整的证人测试程序。虽然该信息没有提供准确的证人测试程序代码，但是信息中的有用提示仍然可以帮助用户手动编写有效的证人测试程序，从而增强缺陷定位的性能。

② 有限的代码生成能力。尽管其他LLMs的性能与GPT-3.5或GPT-4在性能上表现相当，但它们可能因训练数据规模限制而存在一定的局限性。例如，模型Alpaca采用OpenAI公式发布的“text-davinci-003”模型生成了52K条指令数据作为训练数据，该训练规模与GPT-3.5的训练数据规模相比仍然存在一定的差距（尽管OpenAI并没有透露实际数字，但研究者普遍认为GPT-3.5训练规模庞大）。

③ 性能问题。运行LLMs通常需要强大的GPU加持以获得最佳性能。由于本章的实验仅在不支持GPU的CPU上运行，实验中发现对比的LLMs获取回复的响应时间较长：LLMs获得答案的响应时间大约需要300秒。相比之下，GPT-3.5只需调用API进行代码生成，响应时间仅在10秒以内。

值得注意的是，尽管采用其他LLMs的LLM4CBI效果不如内置GPT-3.5，但实验证实了在LLM4CBI中采用新LLMs的难易程度，即用户可以轻松地替换LLM4CBI中的大语言模型以执行编译器缺陷定位任务。随着LLMs的快速发展（如Falcon[142,143]），采用更加先进的前沿技术可以有效缓解上述限制，并在不久的将来辅助编译器缺陷定位任务，以进一步保障编译器质量。

对RQ3的回答： LLM4CBI是高度可扩展的，用户可以轻松将其他LLMs集成到LLM4CBI中以更好地辅助编译器缺陷定位任务。

表 5.6　各种大语言模型的编译器缺陷定位效果对比结果
Tab. 5.6　Compiler bug isolation capability of different LLMs in LLM4CBI

编译器	对比方法	Top-1	Top-5	Top-10	Top-20
GCC	LLM4CBI(Alpaca)	1	10	16	22
	LLM4CBI(Vicuna)	5	15	19	27
	LLM4CBI(GPT4ALL)	2	7	15	21
	LLM4CBI	11	26	41	50
LLVM	LLM4CBI(Alpaca)	1	9	18	23
	LLM4CBI(Vicuna)	1	9	19	32
	LLM4CBI(GPT4ALL)	1	5	18	26
	LLM4CBI	10	24	33	48
ALL	LLM4CBI(Alpaca)	2	19	34	45
	LLM4CBI(Vicuna)	6	24	38	59
	LLM4CBI(GPT4ALL)	3	12	33	47
	LLM4CBI	21	48	74	98

注意: LLM4CBI 采用 GPT-3.5 作为内置的大语言模型。

5.6　讨论

本节首先讨论与 GPT-4 的比较结果，然后评估 LLM4CBI 中不同 temperature 值的影响，最后讨论 LLM4CBI 的一些局限性和本章实验的有效性威胁。

（1）与 GPT-4 大语言模型的比较

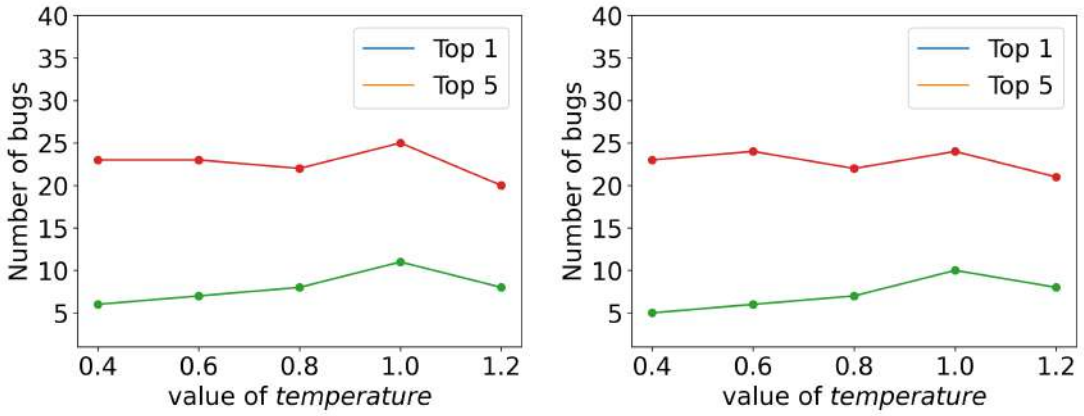
在本章研究中，由于 GPT-4 相对于 GPT-3.5 具有较高的 API 成本[7]，本文没有采用 GPT-4 赋能的 LLM4CBI 进行大规模实验。然而，为了进一步说明不同版本 GPT 对 LLM4CBI 的有效性差异，本文采用 LLM4CBI 的一个变体，称为 LLM4CBI(GPT-4)，对实验数据集中的两个缺陷进行了定位。结果表明 LLM4CBI(GPT-4) 的性能优于 LLM4CBI，其中两个排在 Top 20+ 的缺陷均锁定在 Top-5 的文件中。LLM4CBI(GPT-4) 性能的提升可以归因于两个主要因素。首先，GPT-4 构造出的包含语法错误的测试程序数量较少，表明输出的质量更高。其次，已有研究[111,127] 表明，GPT-4 表现出更好的提示理解能力，从而提高了结果。随着 LLMs 的快速发展，会有更具能力和用户友好的模型可以集成到 LLM4CBI 中，进一步增强 LLM4CBI 的编译器缺陷定位能力。

（2）LLMs 中不同参数设置对实验结果的影响

LLMs（如 GPT-3.5）中的 temperature 参数在控制生成的输出随机性方面起着至关重要的作用。较低的 temperature 值使输出更具确定性，而较高的 temperature 值增加了随机性。在 LLM4CBI 的背景下，研究不同 temperature 值对面向编译器缺陷定位的证人测试程序构造性能影响至关重要。为此，实验分别采用了 0.4、0.6、0.8、1.0 和 1.2

[7]https://openai.com/pricing

等不同 temperature 值进行了对比实验。图5.7显示了不同 temperature 设置下 LLM4CBI 的缺陷定位能力。结果表明，采用 temperature 的默认值 1.0 在 LLM4CBI 中表现最佳。因为 OpenAI 已经对此参数进行了微调，并将 1.0 设置为默认值，所以在 LLM4CBI 中采用默认值产生了最佳结果。



(a) GCC 编译器　　(b) LLVM 编译器

图 5.7　不同 temperature 参数设置的影响结果

Fig. 5.7　The results of the impact of different temperature settings

（3）LLM4CBI 的局限性

LLM4CBI 的局限性继承了 SBFL 技术和 LLMs 的问题。首先，SBFL 存在两个文件排名并列问题，即 tie 问题[144,145]。一个可行的解决方案是通过合并采用提交历史信息来缓解该问题[146]。其次，LLMs 可能会构造出语法无效的测试程序，从而会降低 LLM4CBI 的有效性。但是，LLM4CBI 中增加了相应的反馈机制。实验结果表明 LLM4CBI 产生的语法无效测试程序（即存在编程错误的测试程序）数量是可以接受的，并且对结果没有太大影响。作者计划将在未来工作中采用更有效的方法如利用程序修复技术[118] 自动修复 LLMs 生成的代码。

（4）大语言模型的使用效率

由于不同大语言模型的架构差异，不同的大语言模型对于运行需求是不一样的。本文使用的 GPT-3.5 作为最流行的大语言模型之一，使用的代价（即响应时间和反馈质量等）会受到其公司 OpenAI 的影响。作者相信，随着技术的不断创新，大语言模型将会被越来越低耗且有效地运行。

（5）有效性威胁

LLM4CBI 方案中主要存在的内部、外部和构建有效性方面的威胁。

① 内部有效性威胁。该威胁源自 LLM4CBI 和比较方法（DiWi[12] 和 RecBi[13]）的实现。为了缓解该威胁，LLM4CBI 采用了先前研究中提供的实现[12,13]。至于 LLM4CBI，本文利用已建立的库对其功能进行了仔细地实现（如第5.4.3节所述），并对代码进行

了彻底的检查。在测试程序验证工具方面，因为在程序中识别未定义行为通常是一个具有挑战性的问题[147,148]，即使本章采用了性能较佳的 Frama-C[138] 工具，但它仍然不能保证检测出测试程序内存在的所有未定义行为。作者计划利用更加先进的技术如 CompDiff[149] 来检测测试程序中的潜在未定义行为。

② 外部有效性威胁。该威胁可能受到两个关键因素的影响：编译器的选择和数据集的选择。为确保研究结果的可靠性，在选择编译器时，本文遵循了先前编译器缺陷定位研究中的常用设置[12,13]。因为两个广泛应用的开源 C 编译器（即 GCC 和 LLVM）在社区得到了广泛认可，所以在本章研究中选择了 GCC 和 LLVM。关于研究中采用的缺陷数据集，实验选择了一组包含 120 个真实编译器缺陷的完整数据集，涵盖了先前编译器缺陷定位工作所有的已知编译器缺陷。为了进一步增强研究的外部有效性并应对潜在威胁，作者考虑在未来工作中加入更多真实编译器缺陷以扩展实验中的缺陷数据集，以充分评估 LLM4CBI 的有效性。

③ 构建有效性威胁。该威胁可能涉及随机性、评估指标以及评估过程中的参数设置等方面的潜在威胁。通过以下方法解决以上问题：（1）将实验重复三次并计算中位数结果以考虑随机性，从而减少随机变化的影响；（2）采用广泛应用的缺陷定位指标来评估 LLM4CBI 的有效性，以确保评估结果的可靠性和可比性；（3）在第5.6节中提供明确的参数配置，并深入研究其影响。以上策略增强了 LLM4CBI 结果的可信度，帮助读者对不同参数设置的影响有了更深入的理解。

5.7 本章小结

本章提出了一种面向编译器缺陷定位的大语言模型赋能构造方法 LLM4CBI，构造出了语义完全有效且权衡多样化和相似化的证人测试程序，有效提高了编译器缺陷定位的效率。LLM4CBI 设计了三个新组件，以赋能大语言模型构造出有效的证人测试程序辅助编译器缺陷定位。首先，精确提示制定组件首先设计了一个精确的提示模版，该模版可以准确表示所需的变异操作。然后，利用通过数据和控制流分析测量的程序复杂度指标来识别最相关的变量和最佳变异位置。其次，基于强化学习的提示选择组件和轻量化测试程序验证组件被设计用来选择针对各个编译器缺陷的专用提示。在提示选择组件中，LLM4CBI 通过强化学习结合记忆搜索，根据大语言模型的表现跟踪和积累奖励，允许 LLM4CBI 不断选择专门的提示来改变特定的失败测试程序。在测试程序验证组件中，LLM4CBI 利用静态分析来检测和过滤掉可能包含未定义行为的潜在无效证人测试程序。本章在 120 个真实的 GCC 和 LLVM 编译器缺陷数据集上进行实验，实验结果表明，无论在有效性方面还是效率方面，LLM4CBI 均显著优于现有的最好方法，即 DiWi 和 RecBi。另外，实验还证明 LLM4CBI 能够与其他不断更新迭代的大语言模型结合，进一步提高了编译器缺陷定位的性能。

6 结论与展望

本章主要包括三个方面部分，即全文工作总结、创新点总结和未来工作的展望。

6.1 工作总结

编译器测试和调试是保障编译器质量的主流技术手段。现有工作大多采用基于程序构造的方法构造有效测试程序达到编译器缺陷检测和定位的目的。然而，现有方法仍然存在一定的局限性，降低了质量保障的有效性。为了解决现有方法中的不足，本文从程序构造的角度出发，面向编译器测试和调试的不同场景，展开三个方面的研究，分别是面向编译器前端测试的结构感知程序构造（第三章）、面向编译器中后端测试的再造生成器程序构造（第四章）以及面向编译器缺陷定位的大模型赋能程序构造（第五章）。充分的实验评估表明，本文提出的新程序构造方法有助于帮助开发者有效测试和定位编译器缺陷，从而进一步保障编译器质量。

第三章提出了面向编译器前端测试的结构感知程序构造方法 CCOFT，用于检测深层次的编译器前端缺陷。该方法解决了如何设计灵活化的结构以实现多粒度结合的变异操作及选择代表性的结构以构造多样化的测试程序。为了解决以上挑战，CCOFT 实现了一个通用 C++ 程序生成器。具体而言，CCOFT 首先将 C++ 语法转换为灵活的结构化格式便于实施多粒度结合的变异操作，然后采用平等机会选择（ECS）策略进行结构感知语法突变，从而选择选择多样化为代表的语法结构，最终构造出语法多样化的 C++ 测试程序检测深层次编译器前端缺陷。此外，为了有效识别前端缺陷，CCOFT 采用了一套差异测试策略和编译器输出诊断信息对齐算法，通过比较不一致的编译器输出来识别编译器前端中的不同类型的缺陷。在 GCC 和 Clang 编译器上的实验证明，该方法能显著提高编译器前端缺陷的检测效率，并在编译器开发版本中检测出 136 个缺陷，其中 67 个已被开发者确认、分配或修复。

第四章提出了面向编译器中后端测试的再造生成器程序构造方法 RemGen，用于检测深层次的编译器中后端缺陷。该方法解决了如何轻量化地合成多样化的代码片段和如何有效选择揭示缺陷（即更有可能触发缺陷）的代码片段以构造能够揭示缺陷的测试程序。为了轻量化合成代码片段，RemGen 设计了一个多样化代码片段合成组件。该组件使用语法辅助代码片段合成来生成各种代码片段。为了有效选择揭示缺陷的代码片段，RemGen 设计了揭示缺陷的代码片段选择组件。该组件采用语法覆盖率指标来记录合成过程中所选代码片段语法规则的使用频率。在覆盖率指标的指导下，RemGen 选择具有最多样化语法覆盖的代码片段来构建新揭示缺陷的测试程序。在 GCC 和 LLVM 编译器上的实验证明，该方法能显著提高编译器中后端缺陷的检测效率，并在编译器开发版本中检测出 56 个缺陷，其中 37 个已被开发者修复。

第五章提出了面向编译器缺陷定位的大模型赋能程序构造方法 LLM4CBI，用于有效辅助编译器缺陷定位。该方法解决了如何制定精确提示指导大语言模型的行为和如何选择针对每个缺陷的专用提示。为了制定精确的提示，LLM4CBI 设计了一个精确提示生成组件。该组件首先引入了一个精确的提示模版，该模版可以准确表示所需的突变操作，然后利用通过数据和控制流分析测量的程序复杂度指标来识别最相关的变量和最佳插入位置以填充模版。为了选择特定提示，LLM4CBI 提出了两个新组件，即基于强化学习的提示选择组件和轻量化测试程序验证组件。提示选择组件通过强化学习结合记忆搜索，根据大语言模型的表现跟踪和积累奖励，帮助 LLM4CBI 不断选择专门的提示来变异特定的失败测试程序。在测试程序验证组件中，LLM4CBI 利用静态程序分析技术检测和过滤掉可能包含未定义行为的语义无效测试程序，从而降低语义无效测试程序对定位效率的影响。对 120 个（GCC 和 LLVM 各 60 个）真实编译器缺陷定位的实验结果表明，LLM4CBI 能够轻量且高效地构造高质量的证人测试程序，提升了编译器缺陷定位的效率。

6.2 创新点

为了更好地保障编译器质量，本文从程序构造的角度出发，面向编译器测试和调试的场景展开了三个方面的研究，包括面向编译器前端测试的结构感知程序构造方法、面向编译器中后端测试的再造生成器程序构造方法以及面向编译器缺陷定位的大语言模型赋能程序构造方法。围绕以上三个方法，本文的创新点总结如下：

（1）面向编译器前端测试的结构感知程序构造方法

良好的编译器前端可以帮助编程人员及时调整和修复代码中存在的编程错误，而编译器前端缺陷的存在将大大降低编程人员的开发效率。本文提出了首个 C++ 编译器前端缺陷测试框架 CCOFT，该框架不仅能够提升编译器前端缺陷检测的效率，而且能够在实践中检测出新的编译器缺陷。

创新点 1：提出了面向编译器前端测试的结构感知程序构造方法 CCOFT，解决了灵活化结构设计以实现多粒度结合变异及代表性结构选择以构造多样化的程序等关键挑战，构造出了语法多样化的测试程序，有效提升了编译器前端缺陷检测的效率。

（2）面向编译器中后端测试的再造生成器程序构造方法

良好的编译器中后端可以正确地对代码进行优化，而编译器中后端缺陷可能对软件造成灾难性的后果。本文提出再制造程序生成器并致力于提高其编译器中后端缺陷检测能力的框架 RemGen，该框架不仅提升了编译器中后端缺陷检测的效率，而且能够在实践中检测出新的编译器缺陷。

创新点 2：提出了面向编译器中后端缺陷检测的再造生成器程序构造方法 RemGen，解决了轻量化合成多样化的代码片段和有效选择揭示缺陷的代码片段等关键挑

战，构造出了语义多样化的测试程序，有效提升了编译器中后端缺陷检测的效率。

（3）面向编译器缺陷定位的大模型赋能程序构造方法

当缺陷被检测出来后，及时定位产生缺陷的文件是开发者修复缺陷的关键步骤。本文提出挖掘大语言模型潜能的编译器缺陷定位方法，并提出生成高质量的证人测试程序的框架 LLM4CBI，从而有效地提升缺陷定位的效率。

创新点 3：提出了面向编译器缺陷定位的大模型赋能程序构造方法 LLM4CBI，解决了制定精确提示和选择针对每个缺陷的专用提示等关键挑战，构造了语义完全有效且权衡多样化和相似化的证人测试程序，有效提升了编译器缺陷定位的效率。

全文工作的实用价值。本文工作不仅在方法上具有一定的创新，而且在实践中也为编译器测试和调试产生了积极的现实影响。截止目前，CCOFT 和 RemCCG 已经向两个主流编译器 GCC 和 LLVM 中提交了近 200 个重要的编译器缺陷，超过一半的缺陷已经被开发者修复。同时，在和开发者的交流中，开发者也对本文提交的缺陷报告表示了积极的肯定，并鼓励作者继续提交高质量的编译器缺陷报告。对于编译器缺陷定位，通过充分挖掘大语言模型的潜能，本文提出的 LLM4CBI 方法部署仅涉及较少的人力，可以有效辅助开发者自动化地定位编译器缺陷，能够有效提升开发者修复缺陷的效率。此外，三个工作实现的工具均已开源并申请了软件著作权，后续可直接落地应用到现有或者新设计的编译器中，持续为编译器提供质量保障。

6.3 未来展望

本文使用基于程序构造的方法进一步保障编译器质量，尽管在编译器缺陷检测与缺陷定位的研究取得一定进展，但是本文的工作尚有改进空间，仍然存在一些问题值得进一步研究，本文未来研究方向主要包括以下几个方面：

（1）编译器缺陷检测

构造有效的测试程序是编译器测试的首要任务。虽然本文提出的方法能够有效地生成多样化的测试程序检测编译器缺陷，但构造满足更复杂需求的测试程序仍然需要继续探索。首先，尚未有研究对特定的优化进行测试。编译器优化缺陷通常比较难被检测到，现有的工作仅测试了粗粒度的优化选项（如 “-O1” 到 “-O3”）[2,3] 或者细粒度优化序列的组合[10]。然而，对单一优化序列（如只对 LLVM 中某个特定优化如 “-loop-vectorize”）的测试并没有很好地提出。其次，如何有效利用编译时的动态信息（如编译时使用到的优化策略）指导测试程序的生成同样值得研究。目前，大部分测试程序构造的方法均基于随机生成的模式，该模式虽然可以找到一些编译器缺陷，但是有些潜伏的深层次缺陷可能仅当涉及到的优化序列足够复杂才可能被触发。因此，设计有效的优化交互指导机制用于指导测试程序的构造可以进一步增加编译器测试的有效性。作者在未来工作中考虑研究每个优化的特点以设计专用的测试程序构造方

法以及深入结合优化操作指导的测试技术进一步加强编译器测试。

（2）编译器缺陷定位

缺陷被检测出后，准确定位存在缺陷的位置对加快缺陷的修复十分重要。虽然本文提出的方法能够优于现有的方法，在一定程度上提高编译器缺陷定位的效率，但是由于编译器缺陷的特殊性，目前的缺陷定位方法仍然存在诸多问题值得继续研究。其一，目前定位的缺陷位置级别仅停留在文件级别，由于某个文件可能包含上万或者更多行数的代码，准确找到具体的出错位置仍然是个具有挑战性的问题。因此，定位到更细粒度的位置（如函数或代码行）可以进一步帮助开发者快速修复缺陷。其二，现有的编译器定位工作较少挖掘开发人员的经验以辅助缺陷定位。具体而言，编译器开发人员通常对缺陷有较深的理解，在设计新方法时，如何自动化地将开发者对缺陷理解的反馈用于缺陷定位的过程是一个值得研究且具有挑战性的问题。在未来工作中，作者考虑有效利用编译器代码仓库中的历史信息（如历史修复记录和代码更新记录）并结合机器学习和人工智能技术解决上述问题。

（3）编译器缺陷自动修复

当缺陷被定位到某个函数或者某行后，接下来的重要一步是如何修复该缺陷。目前，已经有不少研究致力于自动化修复普通的软件缺陷。然而，由于编译器缺陷的特殊性（即编译器代码量庞大及各个优化之间逻辑关系复杂），难以将现有的缺陷自动修复工作直接迁移到编译器中。未来工作中，作者考虑从以下两个方面考虑实现编译器缺陷的自动修复。首先，探索并发展结合符号执行、静态分析、机器学习等多种技术的修复方法，以实现更精确和智能的编译器缺陷自动修复。其次，研究将编译器开发者的经验和直觉以及编译器缺陷仓库的历史信息结合到自动修复系统中，以实现智能化的人机协作修复系统，提升自动修复的成功率和效率。

（4）不同技术路线的结合

虽然本文提出的三个程序构造方法分别解决不同测试场景下的特殊问题，但是三个方法中用到的技术路线是通用的。例如，第三章在 CCOFT 中提出的多粒度结合变异也可以应用于面向编译器中后端测试及缺陷定位。此外，第五章提出的基于大语言模型的程序变异方法也可以用于编译器前端或者中后端测试。作者考虑将更多不同的先进技术结合，以进一步保障编译器质量。

参 考 文 献

[1] 张策, 孟凡超, 考永贵, et al. 软件可靠性增长模型研究综述 [J]. 软件学报, 2017, 28 (9): 2402–2430.

[2] Yang X, Chen Y, Eide E, et al. Finding and Understanding Bugs in C Compilers [C]. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2011: 283–294.

[3] Livinskii V, Babokin D, Regehr J. Random Testing for C and C++ Compilers with YARPGen [J]. ACM on Programming Languages, 2020, 4 (196): 1–25.

[4] Le V, Afshari M, Su Z. Compiler Validation via Equivalence Modulo Inputs [C]. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2014: 216–226.

[5] Sun C, Le V, Su Z. Finding Compiler Bugs via Live Code Mutation [C]. In Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2016: 849–863.

[6] Le V, Sun C, Su Z. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation [J]. ACM SIGPLAN Notices, 2015, 50 (10): 386–399.

[7] Aho A V, Lam M S, Sethi R, et al. Compilers: Principles, Techniques, and Tools [M]. 2nd Edition. USA: Addison-Wesley, 2006:3-125.

[8] Chen J, Patra J, Pradel M, et al. A Survey of Compiler Testing [J]. ACM Computer Survey, 2020, 53 (1): 1–36.

[9] Morisset R, Pawan P, Zappa Nardelli F. Compiler Testing via a Theory of Sound Optimisations in the C11/C++ 11 Memory Model [J]. ACM SIGPLAN Notices, 2013, 48 (6): 187–196.

[10] Jiang H, Zhou Z, Ren Z, et al. CTOS: Compiler Testing for Optimization Sequences of LLVM [J]. IEEE Transactions on Software Engineering, 2022, 48 (7): 2339–2358.

[11] Späth C, Mainka C, Mladenov V, et al. SoK: XML Parser Vulnerabilities [C]. In Proceedings of 10th USENIX Workshop on Offensive Technologies (WOOT), 2016: 141‑154.

[12] Chen J, Han J, Sun P, et al. Compiler Bug Isolation via Effective Witness Test Program Generation [C]. In Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2019: 223–234.

[13] Chen J, Ma H, Zhang L. Enhanced compiler bug isolation via memoized search [C]. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2020: 78–89.

[14] 姜文, 刘立康. 软件调试问题研究 [J]. 计算机技术与发展, 2017, 27 (11): 1–6.

[15] Chen J, Hu W, Hao D, et al. An Empirical Comparison of Compiler Testing Techniques [C]. In

Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE), 2016: 180–190.

[16] Zhou Z, Ren Z, Gao G, et al. An Empirical Study of Optimization Bugs in GCC and LLVM [J]. Journal of Systems and Software, 2021, 174 (1): 1–13.

[17] Regehr J, Chen Y, Cuoq P, et al. Test-Case Reduction for C Compiler Bugs [C]. In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2012: 335–346.

[18] Sun C, Le V, Zhang Q, et al. Toward Understanding Compiler Bugs in GCC and LLVM [C]. In Proceedings of the 25th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2016: 294–305.

[19] 陈翔, 鞠小林, 文万志, et al. 基于程序频谱的动态缺陷定位方法研究 [J]. 软件学报, 2015, 26 (2): 390–412.

[20] 张芸, 刘佳琨, 夏鑫, et al. 基于信息检索的软件缺陷定位技术研究进展 [J]. 软件学报, 2020, 31 (8): 2432–2452.

[21] 张文, 李自强, 杜宇航, et al. 方法级别的细粒度软件缺陷定位方法 [J]. 软件学报, 2019, 30 (2): 195–210.

[22] 邹卫琴, 张静宣, 张霄炜, et al. 缺陷报告质量研究综述 [J]. 软件学报, 2021, 34 (1): 171–196.

[23] Abreu R, Zoeteweij P, Van Gemund A J. On the Accuracy of Spectrum-based Fault Localization [C]. In Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques (TAICPART), 2007: 89–98.

[24] 陈俊洁. 数据驱动的编译器测试与调试若干技术研究 [D]. 北京大学. 2019.

[25] 周志德. 编译器优化故障的检测与定位 [D]. 大连理工大学. 2022.

[26] 王冠成. 基于机器学习的编译器测试优化方法研究 [D]. 吉林大学. 2018.

[27] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs [C]. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, 2008: 209–224.

[28] Sun C, Le V, Su Z. Finding and Analyzing Compiler Warning Defects [C]. In Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE), 2016: 203–213.

[29] Wolf K, Klimek M. A Conformance Test Suite for Arden Syntax Compilers and Interpreters. [J]. Studies in Health Technology and Informatics, 2016, 228 (1): 379–383.

[30] Hodován R, Kiss Á, Gyimóthy T. Grammarinator: a Grammar-based Open Source Fuzzer [C]. In Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (TEST), 2018: 45–48.

[31] Team M S. Dharma [EB/OL]. 2020. https://github.com/MozillaSecurity/dharma.

[32] Tang Y, Jiang H, Zhou Z, et al. Detecting Compiler Warning Defects Via Diversity-Guided Program Mutation [J]. IEEE Transactions on Software Engineering, 2022, 48 (11): 4411–4432.

[33] Lidbury C, Lascu A, Chong N, et al. Many-Core Compiler Fuzzing [C]. In Proceedings of the 36th ACM Conference on Programming Language Design and Implementation (PLDI), 2015: 65–76.

[34] Eide E, Regehr J. Volatiles Are Miscompiled, and What to Do about It [C]. In Proceedings of the 8th ACM International Conference on Embedded Software (ICES), 2008: 255–264.

[35] Purdom P. A Sentence Generator for Testing Parsers [J]. BIT Numerical Mathematics, 1972, 12 (1): 366–375.

[36] Bird D L, Munoz C U. Automatic Generation of Random Self-checking Test Cases [J]. IBM Systems Journal, 1983, 22 (3): 229–245.

[37] Lindig C. Random Testing of C Calling Conventions [C]. In Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging (ISAADD), 2005: 3–12.

[38] Sirer E G, Bershad B N. Using Production Grammars in Software Testing [J]. ACM SIGPLAN Notices, 1999, 35 (1): 1–13.

[39] Livinskii V, Babokin D, Regehr J. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages [J]. ACM on Programming Languages, 2023, 7 (181): 1826–1847.

[40] Even-Mendoza K, Cadar C, Donaldson A F. CsmithEdge: More Effective Compiler Testing by Handling Undefined Behaviour Less Conservatively [J]. Empirical Software Enggineering, 2022, 27 (6): 1–35.

[41] Cummins C, Petoumenos P, Murray A, et al. Compiler Fuzzing through Deep Learning [C]. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2018: 95–105.

[42] Liu X, Li X, Prajapati R, et al. Deepfuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing [C]. In Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI), 2019: 1044–1051.

[43] Lee S, Han H, Cha S K, et al. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer [C]. In Proceedings of 29th USENIX Security Symposium, 2020: 2613–2630.

[44] Xu H, Fan S, Wang Y, et al. Tree2tree Structural Language Modeling for Compiler Fuzzing [C]. In Proceedings of 20th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), 2020: 563–578.

[45] Chen Y, Zhong R, Hu H, et al. One Engine to Fuzz'em A ll: Generic Language Processor Testing with Semantic Validation [C]. In Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P), 2021: 642–658.

[46] Chen Y, Su T, Sun C, et al. Coverage-Directed Differential Testing of JVM Implementations [C]. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2016: 85–99.

[47] Chen Y, Su T, Su Z. Deep Differential Testing of JVM Implementations [C]. In Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE), 2019: 1257–1268.

[48] Garoche P-L, Howar F, Kahsai T, et al. Testing-based Compiler Validation for Synchronous Languages [C]. In Proceedings of 6th International Symposium on NASA Formal Methods (NFM), 2014: 246–251.

[49] Holler C, Herzig K, Zeller A. Fuzzing with Code Fragments [C]. In Proceedings of 21st USENIX Security Symposium, 2012: 445–458.

[50] Lin H, Zhu J, Peng J, et al. Deity: Finding Deep Rooted Bugs in JavaScript Engines [C]. In Proceedings of the 19th IEEE International Conference on Communication Technology (ICCT), 2019: 1585–1594.

[51] Zhao Y, Wang Z, Chen J, et al. History-Driven Test Program Synthesis for JVM Testing [C]. In Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE), 2022: 1133–1144.

[52] Chen J, Suo C, Jiang J, et al. Compiler Test-Program Generation via Memoized Configuration Search [C]. In Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE), 2023: 2035–2047.

[53] Liu J, Lin J, Ruffy F, et al. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers [C]. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2023: 530–543.

[54] Wu M, Lu M, Cui H, et al. JITfuzz: Coverage-Guided Fuzzing for JVM Just-in-Time Compilers [C]. In Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE), 2023: 56–68.

[55] Even-Mendoza K, Sharma A, Donaldson A F, et al. GrayC: Greybox Fuzzing of Compilers and Analysers for C [C]. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2023: 1219–1231.

[56] Mandl R. Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing [J]. Communications of the ACM, 1985, 28 (10): 1054–1058.

[57] Austin S, Wilkins D, Wichmann B A. An Ada Program Test Generator [C]. In Proceedings of the Conference on TRI-Ada'91, 1991: 320–325.

[58] Ching W-M, Katz A. The Testing of an APL Compiler [C]. In Proceedings of the International Conference on APL (APL), 1993: 55–62.

[59] Kalinov A, Kossatchev A, Petrenko A, et al. Coverage-driven Automated Compiler Test Suite Generation [J]. Electronic Notes in Theoretical Computer Science, 2003, 82 (3): 500–514.

[60] Kalinov A, Kossatchev A, Petrenko A, et al. Using ASM Specifications for Compiler Testing [C]. In Proceedings of the 10th International Workshop on Abstract State Machines: Advances in Theory and Practice (ASM), 2003: 415–415.

[61] Zhang Q, Sun C, Su Z. Skeletal Program Enumeration for Rigorous Compiler Testing [C]. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2017: 347–361.

[62] Leroy X. Formal Verification of a Realistic Compiler [J]. Communications of the ACM, 2009, 52 (7): 107–115.

[63] 杨萍, 王生原. CompCert 编译器目标代码生成机制分析 [J]. 计算机科学, 2020, 47 (9): 17–23.

[64] Zeller A, Hildebrandt R. Simplifying and Isolating Failure-inducing Input [J]. IEEE Transactions on Software Engineering, 2002, 28 (2): 183–200.

[65] Misherghi G, Su Z. HDD: Hierarchical Delta Debugging [C]. In Proceedings of the 28th IEEE/ACM International Conference on Software Engineering (ICSE), 2006: 142–151.

[66] McPeak S, Wilkerson D S. Berkeley Delta [EB/OL]. 2003. `http://delta.tigris.org/`.

[67] Pflanzer M, Donaldson A F, Lascu A. Automatic Test Case Reduction for OpenCL [C]. In Proceedings of the 4th International Workshop on OpenCL, 2016: 1–12.

[68] Zeller A. Isolating Cause-effect Chains From Computer Programs [J]. ACM SIGSOFT Software Engineering Notes, 2002, 27 (6): 1–10.

[69] Josie H, Alex G. Using Mutants to Help Developers Distinguish and Debug (Compiler) Faults [J]. Software Testing, Verification and Reliability, 2020, 30 (2): 1–34.

[70] Wang G, Shen R, Chen J, et al. Probabilistic Delta Debugging [C]. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2021: 881–892.

[71] Zhou Z, Jiang H, Ren Z, et al. LocSeq: Automated Localization for Compiler Optimization Sequence Bugs of LLVM [J]. IEEE Transactions on Reliability, 2022, 71 (2): 896–910.

[72] Yang J, Yang Y, Sun M, et al. Isolating Compiler Optimization Faults via Differentiating Finer-grained Options [C]. In Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022: 481–491.

[73] Stroustrup B. Thriving in a Crowded and Changing World: C++ 2006–2020 [J]. ACM on Programming Languages, 2020, 4 (70): 1–168.

[74] Padhye R, Lemieux C, Sen K, et al. Semantic Fuzzing with Zest [C]. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2019: 329–340.

[75] Pan W, Chen Z, Zhang G, et al. Grammar-agnostic Symbolic Execution by Token Symbolization [C]. In Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2021: 374–387.

[76] Salls C, Jindal C, Corina J, et al. Token-Level Fuzzing [C]. In Proceedings of 30th USENIX Security Symposium, 2021: 2795–2809.

[77] Waite W M, Goos G. Compiler Construction [M]. USA: Springer-Verlag, 2012:1-34.

[78] Howard M. A Process for Performing Security Code Reviews [J]. IEEE Security & Privacy, 2006, 4 (4): 74–79.

[79] Jana S, Kang Y J, Roth S, et al. Automatically Detecting Error Handling Bugs Using Error Speci-

fications [C]. In Proceedings of the 25th USENIX Security Symposium, 2016: 345–362.

[80] Marcozzi M, Tang Q, Donaldson A F, et al. Compiler Fuzzing: How Much Does it Matter? [J]. ACM on Programming Languages, 2019, 3 (155): 1–29.

[81] Chen J, Bai Y, Hao D, et al. Learning to Prioritize Test Programs for Compiler Testing [C]. In Proceedings of 39th IEEE/ACM International Conference on Software Engineering (ICSE), 2017: 700–711.

[82] Bueno P M, Wong W E, Jino M. Improving Random Test Sets Using the Diversity Oriented Test Data Generation [C]. In Proceedings of the 2nd International Workshop on Random Testing (RT), 2007: 10–17.

[83] Vincenzi A M R, Maldonado J C, Delamaro M E, et al. Component-based Software: An Overview of Testing [J]. Component-Based Software Quality, 2003, 1 (1): 99–127.

[84] Becker B A, Denny P, Pettit R, et al. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research [C]. In Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR), 2019: 177–210.

[85] McCall D, Kölling M. A New Look at Novice Programmer Errors [J]. ACM Transactions on Computing Education, 2019, 19 (4): 1–30.

[86] Parr T. The Definitive ANTLR 4 Reference [M]. USA: Pragmatic Bookshelf, 2013:1-45.

[87] Gopinath R, Zeller A. Building Fast Fuzzers [EB/OL]. 2019. https://arxiv.org/abs/1911.07707.

[88] Havrikov N, Zeller A. Systematically Covering Input Structure [C]. In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019: 189–199.

[89] Chen J, Wang G, Hao D, et al. History-Guided Configuration Diversification for Compiler Test-Program Generation [C]. In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ICSE), 2019: 305–316.

[90] Arcuri A, Briand L. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering [C]. In Proceedings of the 33rd IEEE/ACM International Conference on Software Engineering (ICSE), 2011: 1–10.

[91] Wang Y, Jia X, Liu Y, et al. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization [C]. In Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS), 2020: 1–17.

[92] Babić D, Bucur S, Chen Y, et al. FUDGE: Fuzz Driver Generation at Scale [C]. In Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2019: 975–985.

[93] Ispoglou K, Austin D, Mohan V, et al. FuzzGen: Automatic Fuzzer Generation [C]. In Proceedings of 29th USENIX Security Symposium, 2020: 2271–2287.

[94] Wang J, Chen B, Wei L, et al. Superion: Grammar-Aware Greybox Fuzzing [C]. In Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE), 2019: 724–735.

[95] Mathis B, Gopinath R, Mera M, et al. Parser-Directed Fuzzing [C]. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2019: 548–560.

[96] Bauer S, Cuoq P, Regehr J. Deniable Backdoors Using Compiler Bugs [J]. International Journal of PoC or GTFO, 2015, 1 (1): 7–9.

[97] 高国军, 任志磊, 张静宣, et al. 编译优化序列选择研究进展 [J]. 中国科学: 信息科学, 2019, 49 (10): 1267–1282.

[98] Chen Y, Groce A, Zhang C, et al. Taming Compiler Fuzzers [C]. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2013: 197–208.

[99] Le V, Sun C, Su Z. Randomized Stress-Testing of Link-Time Optimizers [C]. In Proceedings of 24th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2015: 327–337.

[100] Bernard S. Remanufacturing [J]. Journal of Environmental Economics and Management, 2011, 62 (3): 337–351.

[101] Matsumoto M, Yang S, Martinsen K, et al. Trends and Research Challenges in Remanufacturing [J]. International Journal of Precision Engineering and Manufacturing-Green Technology, 2016, 3 (1): 129–142.

[102] Steinhilper R. Remanufacturing- The Ultimate Form of Recycling [M]. France: Fraunhofer IRB Verlag, 1998:34-59.

[103] Commis U S I T. Remanufactured Goods: An Overview of the U.S. and Global Industries, Markets, and Trade [M]. USA: USITC Publication, 2013:101-136.

[104] Chen J, Bai Y, Hao D, et al. Test Case Prioritization for Compilers: A Text-Vector Based Approach [C]. In Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST), 2016: 266–277.

[105] OpenAI. ChatGPT: Optimizing Language Models for Dialogue [EB/OL]. 2022. https://openai.com/blog/chatgpt/.

[106] Liu Z, Chen C, Wang J, et al. Fill in the Blank: Context-Aware Automated Text Input Generation for Mobile GUI Testing [C]. In Proceedings of the 45th International Conference on Software Engineering (ICSE, 2023: 1355–1367.

[107] Wong W E, Gao R, Li Y, et al. A Survey on Software Fault Localization [J]. IEEE Transactions on Software Engineering, 2016, 42 (8): 707–740.

[108] Sutskever I, Vinyals O, Le Q V. Sequence to Sequence Learning with Neural Networks [C]. In Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS),

2014: 3104–3112.

[109] Liu Y. Fine-tune BERT for Extractive Summarization [EB/OL]. 2019. https://arxiv.org/abs/1903.10318.

[110] Yang Z, Dai Z, Yang Y, et al. XLNet: Generalized Autoregressive Pretraining for Language Understanding [C]. In Proceedings of the 33rd International Conference on Neural Information Processing Systems (NIPS), 2019: 5753–5763.

[111] 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 [C]. In Proceedings of the Thirty-seventh Conference on Neural Information Processing Systems (NeurIPS), 2023: 1–15.

[112] Liu P, Yuan W, Fu J, et al. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing [J]. ACM Computer Survey, 2023, 55 (9): 1–35.

[113] White J, Fu Q, Hays S, et al. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT [EB/OL]. 2023. https://arxiv.org/abs/2302.11382.

[114] Reynolds L, McDonell K. Prompt Programming for Large Language Models: Beyond the Few-shot Paradigm [C]. In Proceedings of CHI Conference on Human Factors in Computing Systems (CHI), 2021: 1–7.

[115] Brown T, Mann B, Ryder N, et al. Language Models Are Few-shot Learners [J]. Advances in Neural Information Processing Systems, 2020, 33 (159): 1877–1901.

[116] Niu C, Li C, Ng V, et al. An Empirical Comparison of Pre-Trained Models of Source Code [C]. In Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE), 2023: 2136–2148.

[117] Xia C S, Wei Y, Zhang L. Automated Program Repair in the Era of Large pre-trained Language Models [C]. In Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE), 2023: 1–13.

[118] Fan Z, Gao X, Mirchev M, et al. Automated Repair of Programs from Large Language Models [C]. In Proceedings of the 45th International Conference on Software Engineering, 2023: 1469–1481.

[119] Deng Y, Xia C S, Peng H, et al. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models [C]. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2023: 423–435.

[120] Vaswani A, Shazeer N, Parmar N, et al. Attention Is All You Need [C]. In Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS), 2017: 6000–6010.

[121] Cassano F, Gouwar J, Nguyen D, et al. MultiPL-E: a Scalable and Polyglot Approach to Benchmarking Neural Code Generation [J]. IEEE Transactions on Software Engineering, 2023, 49 (7): 3675–3691.

[122] Nijkamp E, Pang B, Hayashi H, et al. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis [C]. In Proceedings of the 11th International Conference on Learning

Representations (ICLR), 2023: 1–25.

[123] Fried D, Aghajanyan A, Lin J, et al. Incoder: A Generative Model for Code Infilling and Synthesis [C]. In Proceedings of the 7th International Conference on Learning Representations (ICLR), 2022: 1–26.

[124] Xu F F, Alon U, Neubig G, et al. A Systematic Evaluation of Large Language Models of Code [C]. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (ISMP), 2022: 1–10.

[125] Touvron H, Lavril T, Izacard G, et al. LLaMA: Open and Efficient Foundation Language Models [EB/OL]. 2023. https://arxiv.org/abs/2302.13971.

[126] Taori R, Gulrajani I, Zhang T, et al. Stanford Alpaca: An Instruction-following LLaMA model [EB/OL]. , 2023. https://github.com/tatsu-lab/stanford_alpaca.

[127] Chiang W-L, Li Z, Lin Z, et al. Vicuna: An Open-source Chatbot Impressing GPT-4 with 90%* ChatGpt Quality [EB/OL]. 2023. https://lmsys.org/blog/2023-03-30-vicuna/.

[128] Anand Y, Nussbaum Z, Duderstadt B, et al. GPT4ALL: Training an Assistant-style Chatbot with Large Scale Data Distillation from GPT-3.5-Turbo [EB/OL]. , 2023. https://github.com/nomic-ai/gpt4all.

[129] McCabe T J. A Complexity Measure [J]. IEEE Transactions on Software Engineering, 1976, 2 (4): 308–320.

[130] Newman C D, Sage T, Collard M L, et al. Srcslice: A Tool for Efficient Static Forward Slicing [C]. In Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE), 2016: 621–624.

[131] Kaelbling L P, Littman M L, Moore A W. Reinforcement Learning: A Survey [J]. Journal of Artificial Intelligence Research, 1996, 4 (1): 237–285.

[132] Sutton R S, Barto A G. Reinforcement Learning: An Introduction [M]. USA: MIT Press, 2018:3-23.

[133] Mnih V, Kavukcuoglu K, Silver D, et al. Playing Atari with Deep Reinforcement Learning [C]. In Proceedings of the Workshop of Deep Learning on International Conference on Neural Information Processing Systems (NIPS), 2013: 1–9.

[134] Sutton R S, McAllester D, Singh S, et al. Policy Gradient Methods for Reinforcement Learning with Function Approximation [C]. In Proceedings of the 12th International Conference on Neural Information Processing Systems (NIPS), 1999: 1057–1063.

[135] KONDA V. Actor-critic Algorithms [J]. Advances in Neural Information Processing Systems, 2000, 12 (1): 1008–1014.

[136] Mnih V, Badia A P, Mirza M, et al. Asynchronous Methods for Deep Reinforcement Learning [C]. In Proceedings of the International Conference On Machine Learning (ICML), 2016: 1928–1937.

[137] Grondman I, Busoniu L, Lopes G A, et al. A Survey of Actor-critic Reinforcement Learning: Standard and Natural Policy Gradients [J]. IEEE Transactions on Systems, Man, and Cybernetics, 2012,

42 (6): 1291–1307.

[138] Kirchner F, Kosmatov N, Prevosto V, et al. Frama-C: A Software Analysis Perspective [J]. Formal Aspects of Computing, 2015, 27 (3): 573–609.

[139] Jeffrey D, Gupta N, Gupta R. Fault Localization Using Value Replacement [C]. In Proceedings of the 2008 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2008: 167–178.

[140] Pearson S, Campos J, Just R, et al. Evaluating and Improving Fault Localization [C]. In Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE), 2017: 609–620.

[141] Ma W, Liu S, Wang W, et al. The Scope of ChatGPT in Software Engineering: A Thorough Investigation [EB/OL]. 2023. https://arxiv.org/abs/2305.12138.

[142] Almazrouei E, Cappelli A, Cojocaru R, et al. Falcon-40B: an Open Large Language Model with State-of-the-art Performance [EB/OL]. 2023. https://huggingface.co/tiiuae/falcon-40b.

[143] Penedo G, Malartic Q, Hesslow D, et al. The RefinedWeb dataset for Falcon LLM: Outperforming Curated Corpora with Web Data, and Web Data Only [EB/OL]. 2023. https://arxiv.org/abs/2306.01116.

[144] Yu Y, Jones J A, Harrold M J. An Empirical Study of the Effects of Test-suite Reduction on Fault Localization [C]. In Proceedings of the 30th IEEE/ACM International Conference on Software Engineering (ICSE), 2008: 201–210.

[145] Xu X, Debroy V, Eric Wong W, et al. Ties within Fault Localization Rankings: Exposing and Addressing the Problem [J]. International Journal of Software Engineering and Knowledge Engineering, 2011, 21 (6): 803–827.

[146] Wen M, Chen J, Tian Y, et al. Historical Spectrum-based Fault Localization [J]. IEEE Transactions on Software Engineering, 2019, 47 (11): 2348–2368.

[147] Lee J, Kim Y, Song Y, et al. Taming Undefined Behavior in LLVM [J]. ACM SIGPLAN Notices, 2017, 52 (6): 633–647.

[148] Wang X, Zeldovich N, Kaashoek M F, et al. Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior [C]. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP), 2013: 260–275.

[149] Li S, Su Z. Finding Unstable Code via Compiler-Driven Differential Testing [C]. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2023: 238‑251.

攻读博士学位期间科研项目及科研成果

已发表论文

[1] **Haoxin Tu**, He Jiang, Zhide Zhou, *et al*. Detecting C++ Compiler Front-end Bugs via Grammar Mutation and Differential Testing [J]. ***IEEE Transactions on Reliability***, 2022, 72(1):343-357. DOI:10.1109/TR.2022.3171220 (SCI 检索号:000799578000001) (JCR Q1)（本学位论文第三章）

[2] **Haoxin Tu**, He Jiang, Xiaochen Li, *et al*. RemGen: Remanufacturing A Random Program Generator for Compiler Testing [C]. ***IEEE International Symposium on Software Reliability Engineering***, 2022, 529-540. DOI:10.1109/ISSRE55969.2022.00057 (SCI 检索号:000909299200046) (CCF-B 国际会议)（本学位论文第四章）

待发表论文

[1] **Haoxin Tu**, Zhide Zhou, He Jiang, *et al*. LLM4CBI: Taming LLMs to Generate Effective Test Programs for Compiler Bug Isolation [J]. ***IEEE Transactions on Software Engineering***, 2023. (Major Revision) (CCF-A 国际期刊)（本学位论文第五章）

发明专利及软件著作权

[1] 江贺, **涂浩新**, 高越, 林浩, 周志德, 任志磊. 一种基于大语言模型赋能的编译器缺陷定位方法: 中国. 发明专利: 2023113619856.（本学位论文第五章）

[2] 江贺, **涂浩新**, 周志德, 任志磊. 基于结构感知变异及差分策略的编译器前端缺陷检测系统. 中国. 软件著作权: 2023R11L2030184.（本学位论文第三章）

[3] 江贺, **涂浩新**, 李晓晨, 周志德, 任志磊. 基于再制造生成器的编译器中后端缺陷检测系统. 中国. 软件著作权: 2023R11L2030765.（本学位论文第四章）

[4] 江贺, **涂浩新**, 周志德, 任志磊. 基于大语言模型赋能的编译器缺陷定位系统. 中国. 软件著作权: 2023R11L2030388.（本学位论文第五章）

获得奖励

[1] “第十八届全国软件与应用学术会议 NASAC2019 命题型竞赛”, 国家级, 三等奖, 2019.11. 完成人排序: 1/4.

[2] “第十六届中国研究生数学建模竞赛”, 国家级, 三等奖, 2019.12. 完成人排序: 1/3.

[3] “大连理工大学优秀研究生称号”, 校级, 2022.11. 完成人排序: 1/1.

[4] “大连理工大学博士生一等学业奖学金”, 校级, 2022.12. 完成人排序: 1/1.

[5] “博士研究生国家奖学金”, 国家级, 2022.12. 完成人排序: 1/1.

参与科研项目

[1] 国家自然科学基金优秀青年科学基金项目 (61722202): 智能软件工程, 2018.1-2020.12, 负责人: 江贺.

[2] 北京控制工程研究所合作项目：无人系统智能交互软件, 2020.3-2020.12, 负责人: 江贺.

[3] 国家重点研发计划课题 (2018YFB1003903)：基于代码风格与编程规范的代码现场检测与智能改进技术，2018.5–2021.4, 负责人: 江贺.

其他已发表/在投论文 (与新加坡管理大学第二博士学位相关)

[1] **Haoxin Tu**, Lingxiao Jiang, Debin Gao, He Jiang, "Beyond a Joke: Dead Code Elimination Can Delete Live Code", ***NIER Track of International Conference on Software Engineering (ICSE-NIER 2024).*** (Accepted) (CCF-A 国际会议)

[2] **Haoxin Tu**, "Boosting Symbolic Execution for Heap-based Vulnerability Detection and Exploit Generation", ***in the Doctoral Symposium Track of International Conference on Software Engineering (ICSE 2023)***. (Accepted) (CCF-A 国际会议)

[3] Pansilu Pitigalaarachchi, Xuhua Ding, Haiqing Qiu, **Haoxin Tu**, Jiaqi Hong, and Lingxiao Jiang, "KRover: A Symbolic Execution Engine for Dynamic Kernel Analysis", ***in Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS 2023). Research Paper.*** (Accepted) (CCF-A 国际会议)

[4] **Haoxin Tu**, Lingxiao Jiang, Xuhua Ding, and He Jiang, "FastKLEE: Faster Symbolic Execution via Reducing Redundant Bound Checking of Type-Safe Pointers", ***in the Demonstrations Track of European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE 2022).*** (Accepted) (CCF-A 国际会议)

[5] **Haoxin Tu**, Lingxiao Jiang, Jiaqi Hong, Xuhua Ding, and He Jiang, "Concretely Mapped Symbolic Memory Locations for Memory Error Detection", ***Submitted to IEEE Transactions on Software Engineering***. 2023. (Major Revision) (CCF-A 国际期刊)

致　谢

求知若饥，虚心若愚。转眼进入博士学习尾声。这一段博士求学生涯，通过不断阅读和行路，我看见了世界的辽阔和自身的不足，更我让有机会从小小的家乡迈出勇闯世界的第一步。这一路不仅有苦有累有辛酸，也有好风光和无与伦比的良师益友。在此，仅对给予我帮助的家人、老师和朋友们表示深深的感谢。

首先，特别感谢我的导师江贺教授。作为大连理工大学的硕士及博士生，江老师不仅仅是我的硕博导师，还是我的人生导师。江老师对科研和学术研究的热爱，以及对软件工程研究方向的准确把握，使得实验室的同学们不断取得新成果和新突破。同时，江老师还给予了我们自由的学术环境和生活上的各种帮助，使得我们可以在自由轻松的环境下开展科学研究工作。除了学术上的帮助外，江老师严谨细心的处事风格也深深影响着我。在今后的学术生涯中，江老师必定是我追求的榜样，我将努力做到和江老师一样在学术道路上漫漫求索。此外，特别感谢江老师在我博士一年级时支持我参与软件学院与新加坡管理大学的双博士学位项目，没有导师的认可和持续帮助，我在短暂的五六年时光内顺利完成该项目的可能性几乎为零。

其次，感谢新加坡管理大学的 Lingxiao Jiang 教授和 Xuhua Ding 教授对我在新加坡交流时学术上的帮助。两位老师治学严谨以及追求卓越的学术态度深深影响着我。

再次，感谢课题组其他老师、师兄师姐以及师弟师妹。首先感谢任志磊教授。感谢任老师在我刚到大连时对我生活上的帮助以及在我论文撰写过程中提供的建议。任老师踏实勤奋的品格以及对科研的热爱执着为我树立了榜样。其次感谢实验室的师兄师姐。在我开始博士之路时，张静宣师兄、陈信师兄、李晓晨师兄、周志德师兄、刘东师兄、高国军师兄、赵旭师兄和唐艺璇师姐总是在我迷茫时为我指点迷津，你们认真刻苦的求学态度值得我学习。最后感谢实验室的师弟师妹们，如王尊、付刘伟、陈昊、李康乐、陈乐、林浩、高越等等，我们一起完成了很多有意义的工作。

最后，感谢我的家人尽其所能辛苦付出以及在我人生成长路上给予了我足够自由选择的权利。我长大了，他们也苍老了许多，以后的日子希望他们老的慢一些，让我有机会对他们多一些陪伴。最后，感谢我弟弟涂浩琮，感谢他在母亲身边的陪伴，未曾远离家乡，让我在求学之路上少了许多后顾之忧。

行文至此，唯有感谢。大工五六载光阴，二十载求学路，这一路有苦有乐，但总有良师益友相伴，不曾孤单。勇敢的人敢于追求理想，在大连理工大学的求学生涯即将告一段落，未来无论在世界何处，我会始终不忘初心，砥砺前行。

涂浩新

写于 2023 年 10 月 20 日

作者简介

姓名: 涂浩新

性别: 男

出生年月: 1996 年 9 月

民族: 汉族

籍贯: 江西省南昌市

研究方向: 软件测试与软件调试

简历：

2019.9—至今　大连理工大学　博士生

2020.8—至今　新加坡管理大学　博士生

2017.9-2019.6　大连理工大学　硕士

2013.9-2017.6　东北林业大学　本科