大连理工大学
Dalian University Of Technology

面向编译器测试及调试的程序构造方法研究

（大连理工大学博士学位论文答辩）

答辩人 涂浩新　　**导 师** 江贺 教授

软件学院2019级博士生

2023年12月6日星期三

目录

01

研究背景与动机

Background & Motivation

研究背景：编译器是重要的系统软件

研究背景：编译器是重要的系统软件

编译器是一种计算机程序，将某种程序语言编写成的源代码（高级语言）转换成另一种机器代码（低级语言）

研究背景：编译器是重要的系统软件

编译器是一种计算机程序，将某种程序语言编写成的源代码（高级语言）转换成另一种机器代码（低级语言）

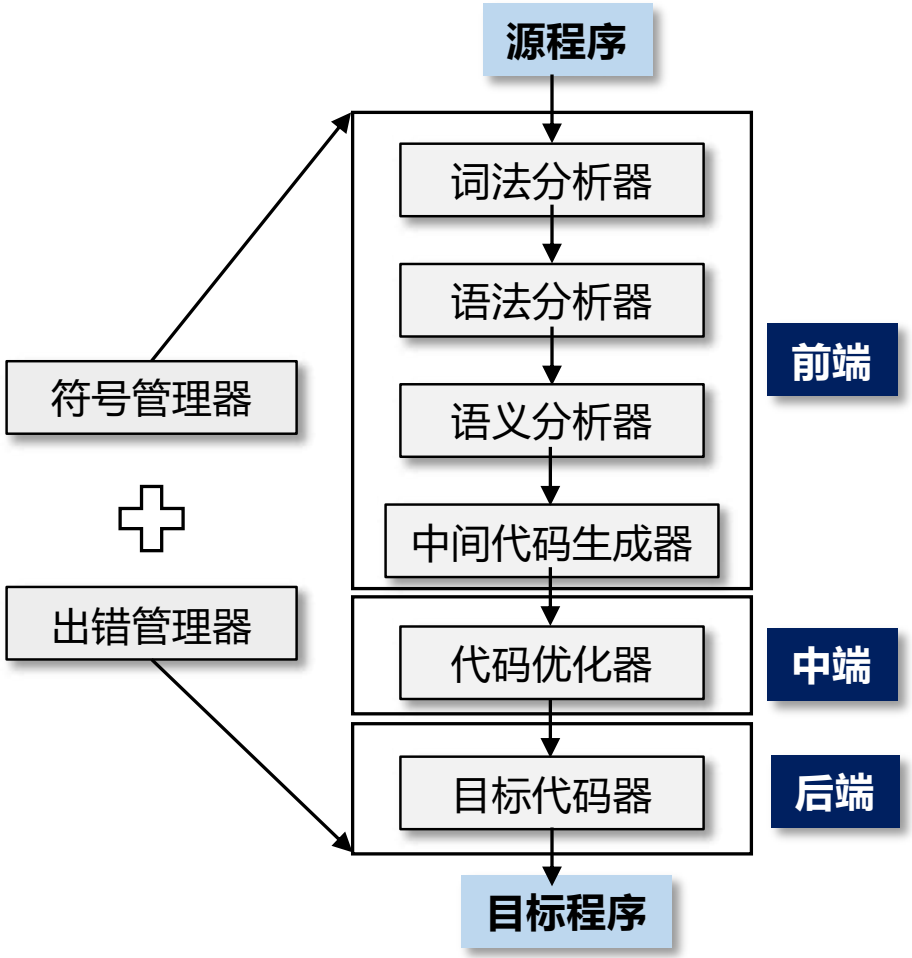


图1 典型的编译器架构

研究背景：编译器是重要的系统软件

编译器是一种计算机程序，将某种程序语言编写成的源代码（高级语言）转换成另一种机器代码（低级语言）

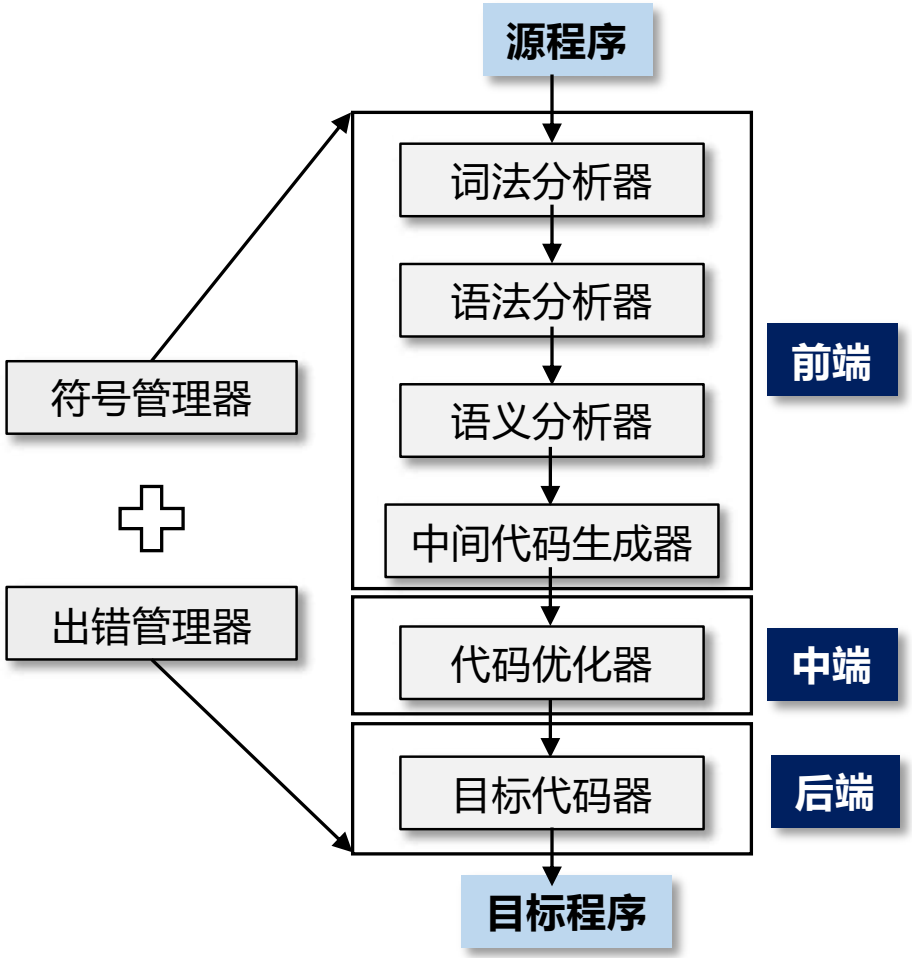


图1 典型的编译器架构

系统软件: Linux操作系统，数据库

应用软件：微信, zoom, 腾讯会议

研究背景：编译器是重要的系统软件

编译器是一种计算机程序，将某种程序语言编写成的源代码（高级语言）转换成另一种机器代码（低级语言）

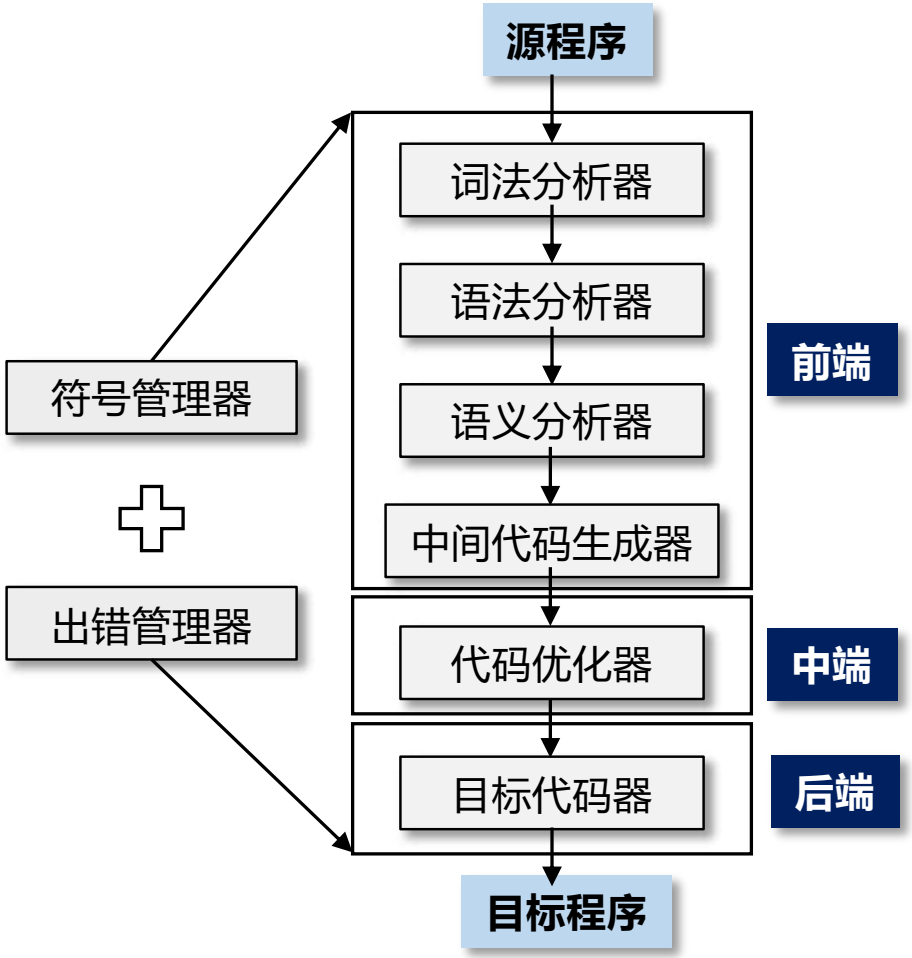
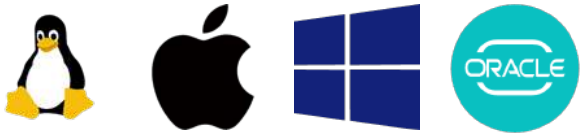
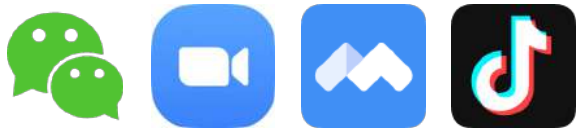


图1 典型的编译器架构

系统软件: Linux操作系统，数据库



应用软件：微信, zoom, 腾讯会议



- 工业和信息化部发布《**“十四五”软件和信息技术服务业发展规划**》中提出到2025年, 软件行业收入突破**14万亿元**
- 习主席发表的《**加强基础研究实现高水平科技自立自强**》中提到“要打好科技仪器设备、操作系统和**基础软件**国产化攻坚战”

研究背景：编译器缺陷的危害

编译器缺陷是指编译器源代码中存在的编程错误，会对编译后的软件造成严重可靠性/安全性威胁

[1] Sun, Chengnian, Vu Le, Qirun Zhang, and Zhendong Su. "Toward understanding compiler bugs in GCC and LLVM." In *Proceedings of the 25th international symposium on software testing and analysis*, pp. 294-305. 2016.

研究背景：编译器缺陷的危害

编译器缺陷是指编译器源代码中存在的编程错误，会对编译后的软件造成严重可靠性/安全性威胁

工业界

学术界

[1] Sun, Chengnian, Vu Le, Qirun Zhang, and Zhendong Su. "Toward understanding compiler bugs in GCC and LLVM." In *Proceedings of the 25th international symposium on software testing and analysis*, pp. 294-305. 2016.

研究背景：编译器缺陷的危害

编译器缺陷是指编译器源代码中存在的编程错误，会对编译后的软件造成严重可靠性/安全性威胁

工业界

操作系统　手机软件　安全相关系统

有缺陷编译器

崩溃/错误编译/性能故障等

增加程序调试难度

或者

导致严重应用故障

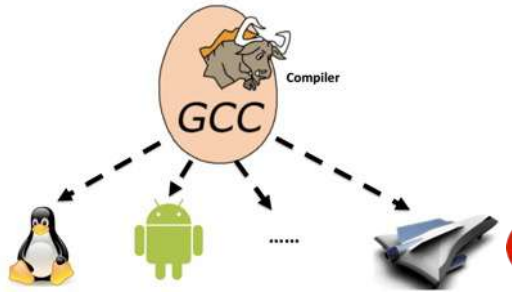
学术界

[1] Sun, Chengnian, Vu Le, Qirun Zhang, and Zhendong Su. "Toward understanding compiler bugs in GCC and LLVM." In *Proceedings of the 25th international symposium on software testing and analysis*, pp. 294-305. 2016.

研究背景：编译器缺陷的危害

编译器缺陷是指编译器源代码中存在的编程错误，会对编译后的软件造成严重可靠性/安全性威胁

工业界



操作系统　手机软件　安全相关系统

有缺陷编译器

崩溃/错误编译/性能故障等

增加程序调试难度

或者

导致严重应用故障

学术界

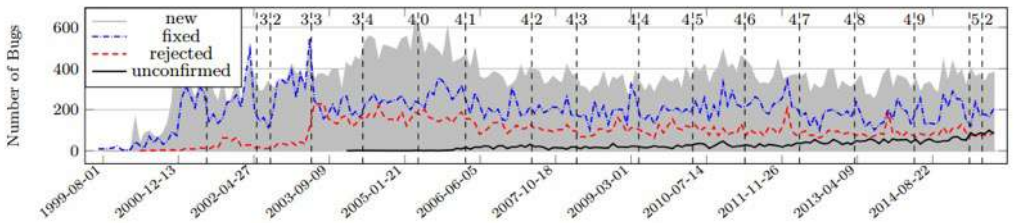


图2 GCC编译器中的缺陷

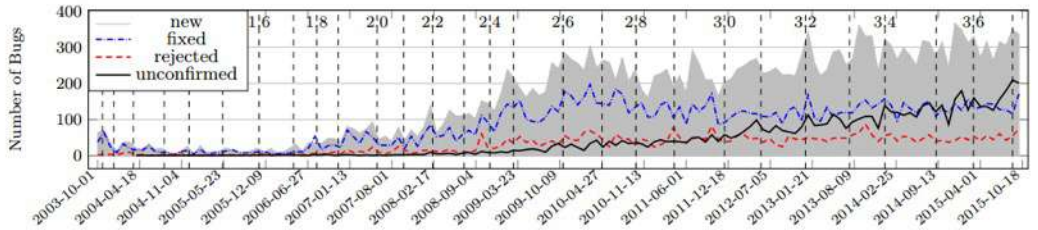


图3 LLVM编译器中的缺陷

实证研究表明，尽管编译器经过了大量测试，编译器中仍然存在缺陷[1]，因此保障编译器的质量至关重要

[1] Sun, Chengnian, Vu Le, Qirun Zhang, and Zhendong Su. "Toward understanding compiler bugs in GCC and LLVM." In *Proceedings of the 25th international symposium on software testing and analysis*, pp. 294-305. 2016.

研究背景：编译器质量保障之测试与调试 (1/2)

研究背景：编译器质量保障之测试与调试 (1/2)

研究者通常采用基于程序构造的方法对编译器进行测试（缺陷检测）和调试（缺陷定位）

研究背景：编译器质量保障之测试与调试 (1/2)

研究者通常采用基于程序构造的方法对编译器进行测试（缺陷检测）和调试（缺陷定位）

研究背景：编译器质量保障之测试与调试 (1/2)

研究者通常采用基于程序构造的方法对编译器进行测试（缺陷检测）和调试（缺陷定位）

检测主要目标 构造多样化的测试程序测试编译器

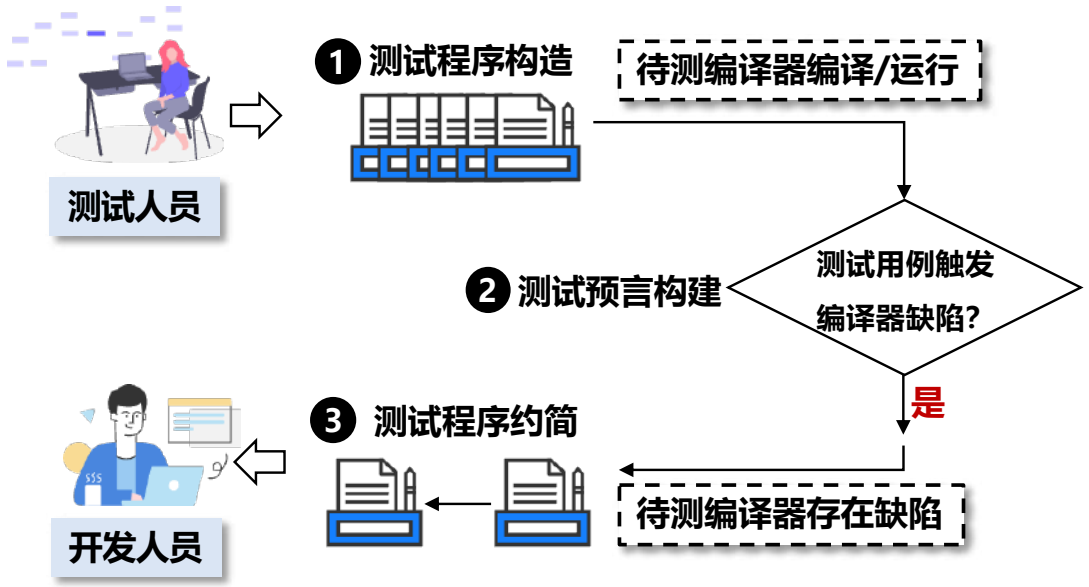


图4 编译器缺陷检测基本流程

研究背景：编译器质量保障之测试与调试 (1/2)

研究者通常采用基于程序构造的方法对编译器进行测试（缺陷检测）和调试（缺陷定位）

检测主要目标 构造多样化的测试程序测试编译器

测试程序构造 构造测试输入

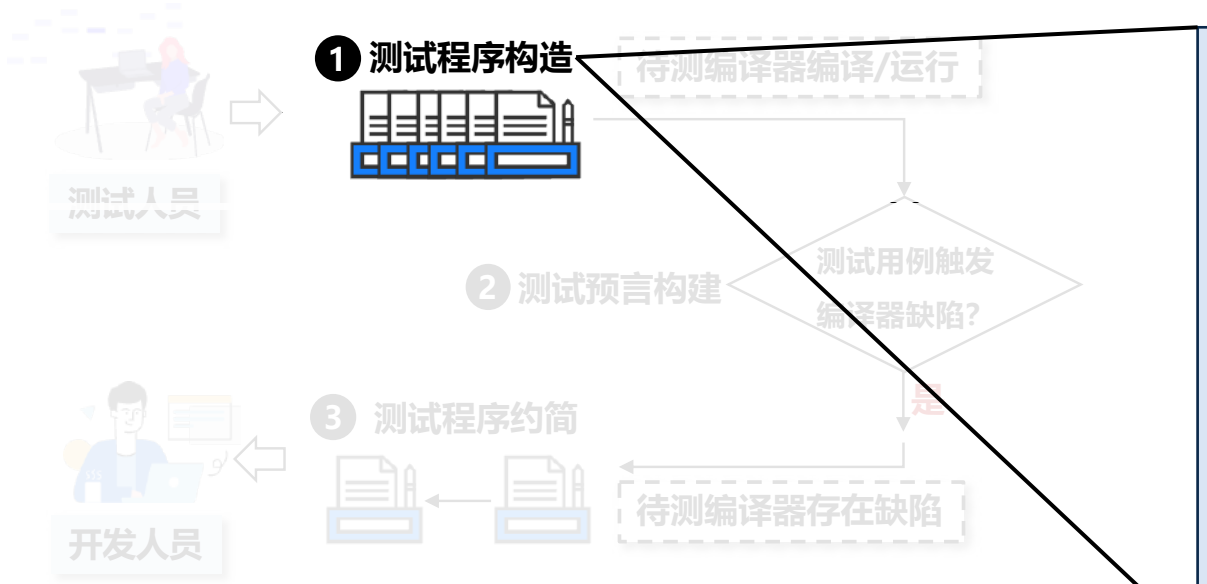


图4 编译器缺陷检测基本流程

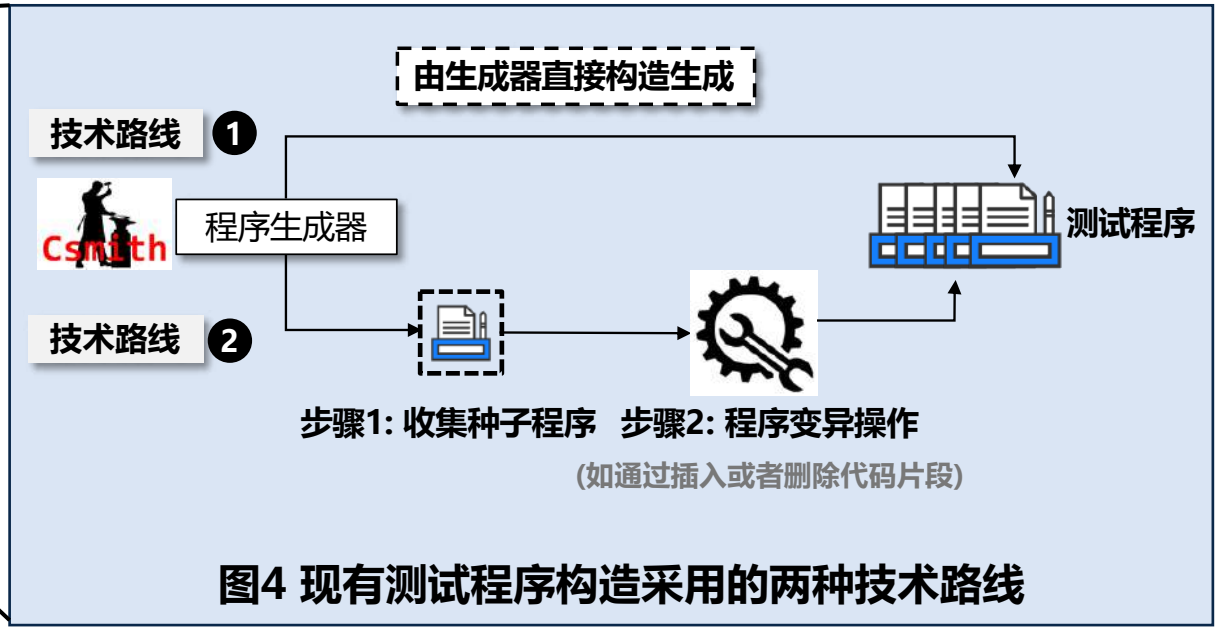


图4 现有测试程序构造采用的两种技术路线

研究背景：编译器质量保障之测试与调试 (1/2)

研究者通常采用基于程序构造的方法对编译器进行测试（缺陷检测）和调试（缺陷定位）

检测主要目标 构造多样化的测试程序测试编译器

测试程序构造 构造测试输入

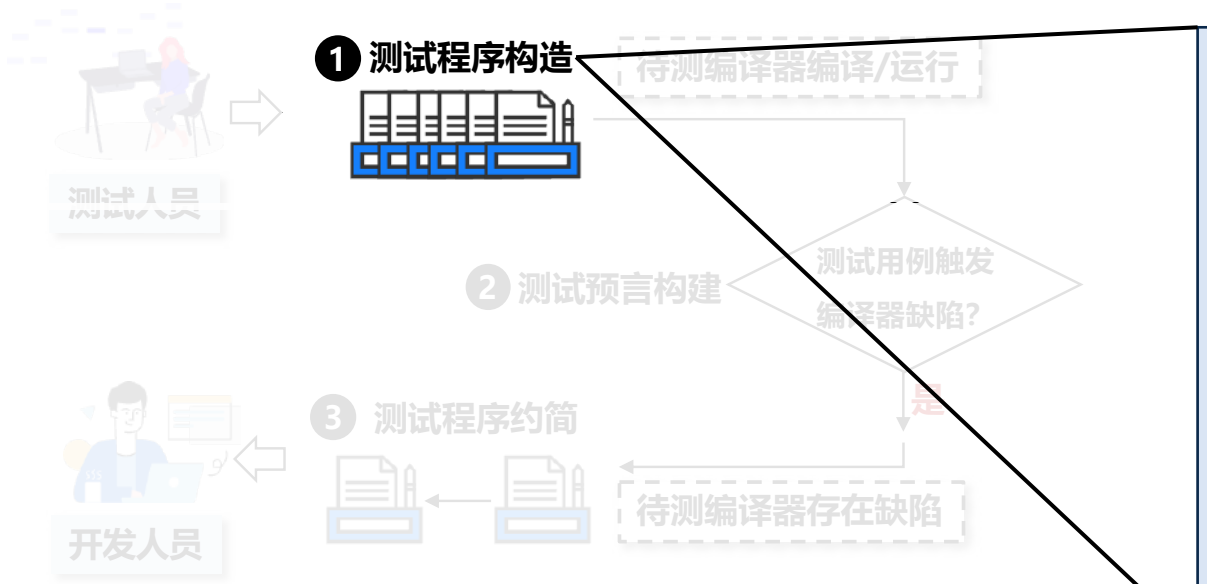


图4 编译器缺陷检测基本流程

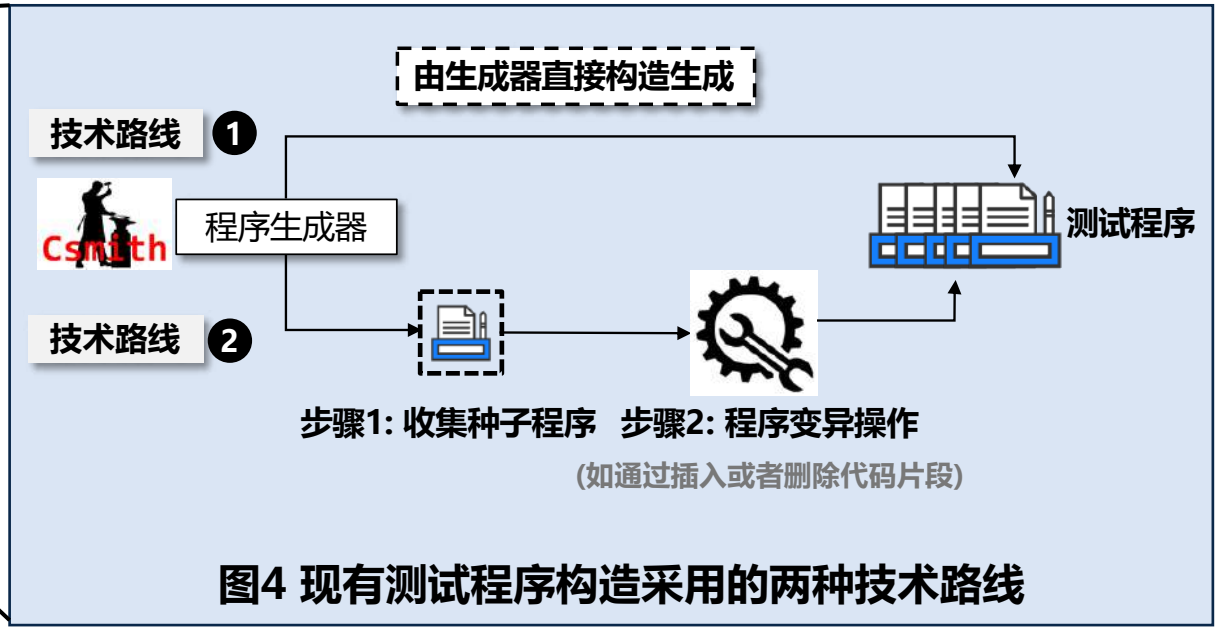


图4 现有测试程序构造采用的两种技术路线

构造多样化的测试用例是编译器测试流程中至关重要的一环

研究背景：编译器质量保障之测试与调试 (2/2)

研究背景：编译器质量保障之测试与调试 (2/2)

研究者通常采用基于程序构造的方法对编译器进行测试（缺陷检测）和调试（缺陷定位）

研究背景：编译器质量保障之测试与调试 (2/2)

研究者通常采用基于程序构造的方法对编译器进行测试（缺陷检测）和调试（缺陷定位）

研究背景：编译器质量保障之测试与调试 (2/2)

研究者通常采用基于程序构造的方法对编译器进行测试（缺陷检测）和调试（缺陷定位）

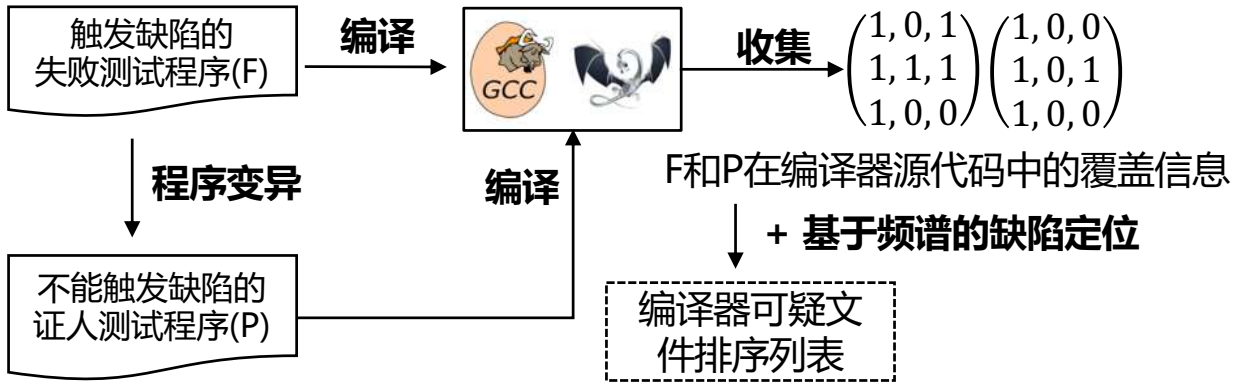


图5 基于程序构造的编译器缺陷定位基本流程

研究背景：编译器质量保障之测试与调试 (2/2)

研究者通常采用基于程序构造的方法对编译器进行测试（缺陷检测）和调试（缺陷定位）

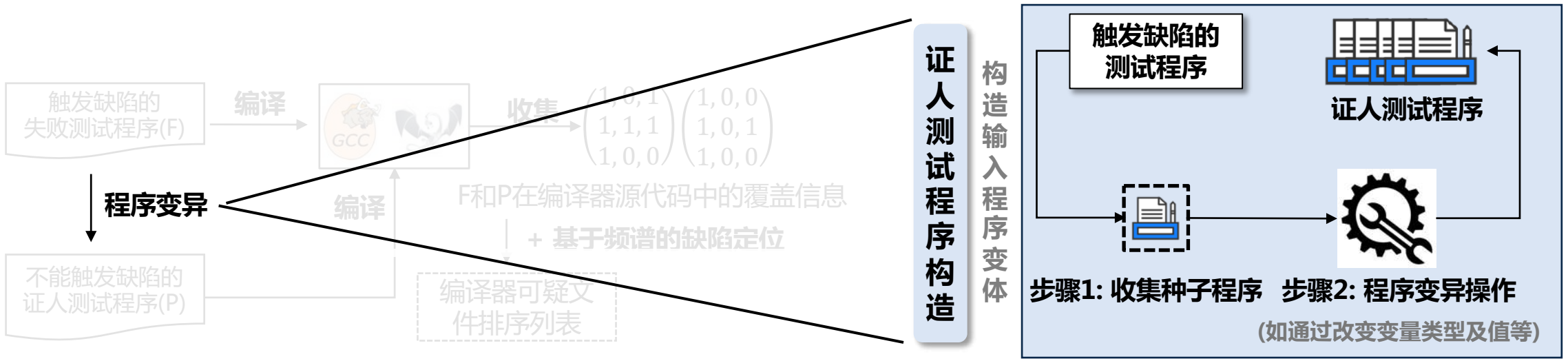


图5 基于程序构造的编译器缺陷定位基本流程

研究背景：编译器质量保障之测试与调试 (2/2)

研究者通常采用基于程序构造的方法对编译器进行测试（缺陷检测）和调试（缺陷定位）

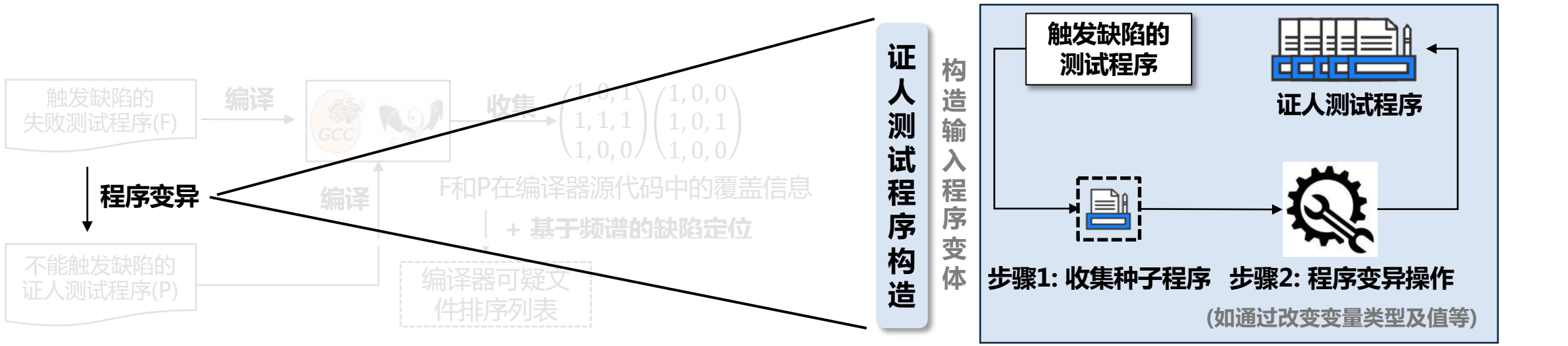
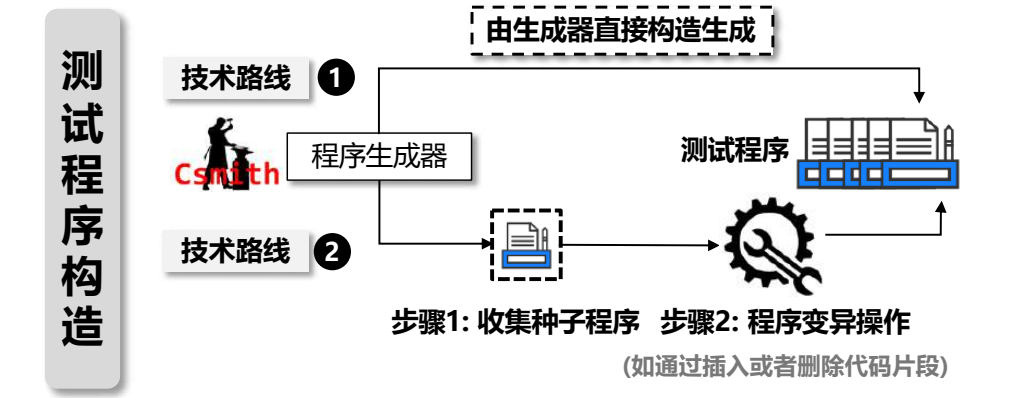


图5 基于程序构造的编译器缺陷定位基本流程



两点不同：输入的种子程序/变异策略

研究背景：编译器质量保障之测试与调试 (2/2)

研究者通常采用基于程序构造的方法对编译器进行测试（缺陷检测）和调试（缺陷定位）

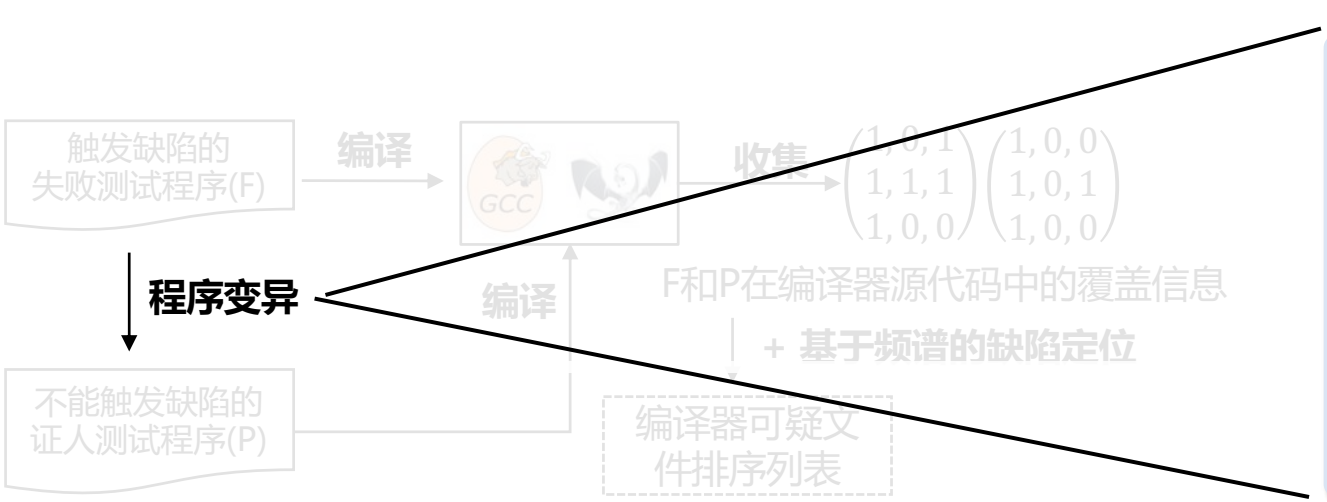
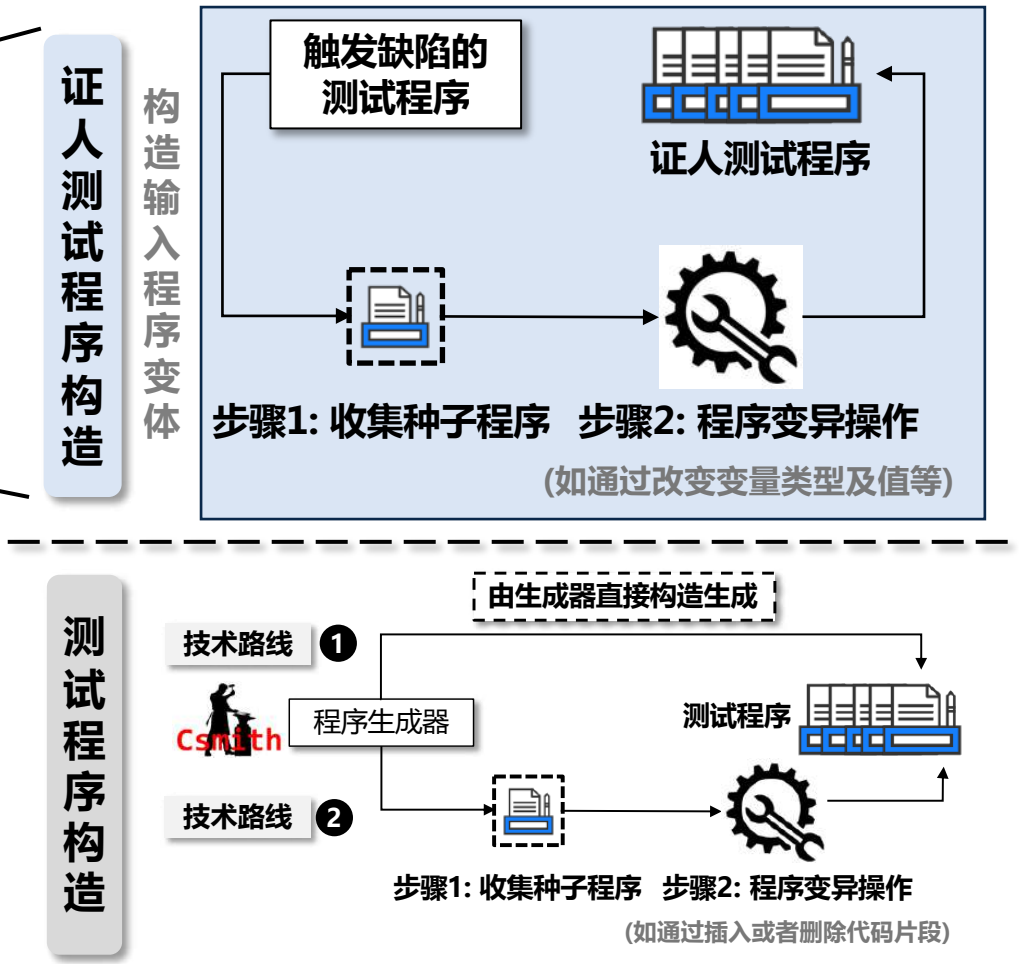


图5 基于程序构造的编译器缺陷定位基本流程



构造权衡多样化和相似化的证人测试程序是编译器缺陷定位过程中至关重要的一环

两点不同：输入的种子程序/变异策略

研究现状

研究现状

根据编译器基本结构（即前、中后端）及编译器质量保障的基本实现方法（测试及调试）

研究现状

根据编译器基本结构（即前、中后端）及编译器质量保障的基本实现方法（测试及调试）

➢ 面向编译器前端测试的程序构造方法

- **生成完全有效的测试程序:** 顺利通过前端分析，无法检测到潜在的前端缺陷

  Csmith [PLDI'2011], YARPGen[OOPLSA'2020]

- **生成词法无效的测试程序：**停止在前端分析早期词法分析阶段，无法检测到深层次的前端缺陷

  Dharma [Mozilla'2011], Grammarinator [TEST'2018]

研究现状

根据编译器基本结构（即前、中后端）及编译器质量保障的基本实现方法（测试及调试）

➢ 面向编译器前端测试的程序构造方法

- **生成完全有效的测试程序:** 顺利通过前端分析，无法检测到潜在的前端缺陷
- **生成词法无效的测试程序：**停止在前端分析早期词法分析阶段，无法检测到深层次的前端缺陷

Dharma [Mozilla'2011], Grammarinator [TEST'2018]

主要缺点：快速通过前端分析或只能检测到浅层次的前端缺陷

研究现状

根据编译器基本结构（即前、中后端）及编译器质量保障的基本实现方法（测试及调试）

➢ 面向编译器前端测试的程序构造方法

- **生成完全有效的测试程序:** 顺利通过前端分析，无法检测到潜在的前端缺陷

主要缺点：快速通过前端分析或只能检测到浅层次的前端缺陷

- **生成词法无效的测试程序：**停止在前端分析早期词法分析阶段，无法检测到深层次的前端缺陷

Dharma [Mozilla'2011], Grammarinator [TEST'2018]

➢ 面向编译器中后端测试的程序构造方法

- **基于生成的程序构造**：生成完全有效 (不含未定义行为的程序)

  Csmith [PLDI'2011], YARPGen[OOPSLA'2020]

- **基于变异的程序构造**：生成完全有效 (不含未定义行为的程序)

  Orion[PLDI'2014], Athena[OOPLSA'2015],Hermes[OOPLSA'2016]

研究现状

大连理工大学 Dalian University Of Technology　Lab.Oscar www.oscar-lab.org

根据编译器基本结构（即前、中后端）及编译器质量保障的基本实现方法（测试及调试）

➢ 面向编译器前端测试的程序构造方法

- **生成完全有效的测试程序:** 顺利通过前端分析，无法检测到潜在的前端缺陷

主要缺点：快速通过前端分析或只能检测到浅层次的前端缺陷

- **生成词法无效的测试程序：**停止在前端分析早期词法分析阶段，无法检测到深层次的前端缺陷

Dharma [Mozilla'2011], Grammarinator [TEST'2018]

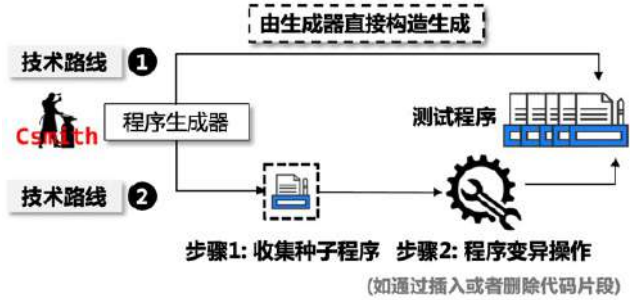
➢ 面向编译器中后端测试的程序构造方法

- **基于生成的程序构造**：生成完全有效 (不含未定义行为的程序)

  Csmith [PLDI'2011], YARPGen[OOPSLA'2020]

- **基于变异的程序构造**：生成完全有效 (不含未定义行为的程序)

  Orion[PLDI'2014], Athena[OOPLSA'2015],Hermes[OOPLSA'2016]



研究现状

根据编译器基本结构（即前、中后端）及编译器质量保障的基本实现方法（测试及调试）

➢ 面向编译器前端测试的程序构造方法

- **生成完全有效的测试程序:** 顺利通过前端分析，无法检测到潜在的前端缺陷

主要缺点：快速通过前端分析或只能检测到浅层次的前端缺陷

- **生成词法无效的测试程序：**停止在前端分析早期词法分析阶段，无法检测到深层次的前端缺陷

Dharma [Mozilla'2011], Grammarinator [TEST'2018]

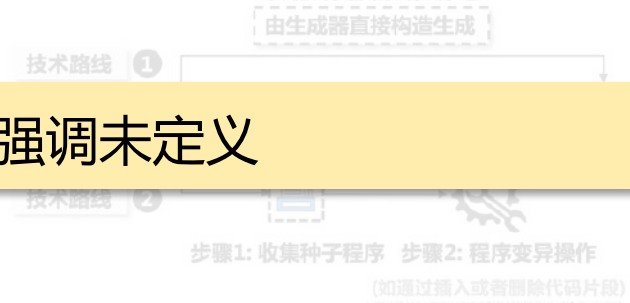
➢ 面向编译器中后端测试的程序构造方法

- **基于生成的程序构造**：生成完全有效 (不含未定义行为的程序)

主要缺点：程序生成器很难再检测出新缺陷或过于强调未定义

- **基于变异的程序构造**：生成完全有效 (不含未定义行为的程序)

Orion[PLDI'2014], Athena[OOPLSA'2015],Hermes[OOPLSA'2016]



研究现状

根据编译器基本结构（即前、中后端）及编译器质量保障的基本实现方法（测试及调试）

➢ 面向编译器前端测试的程序构造方法

- **生成完全有效的测试程序:** 顺利通过前端分析，无法检测到潜在的前端缺陷

主要缺点：快速通过前端分析或只能检测到浅层次的前端缺陷

- **生成词法无效的测试程序：**停止在前端分析早期词法分析阶段，无法检测到深层次的前端缺陷

Dharma [Mozilla'2011], Grammarinator [TEST'2018]

➢ 面向编译器中后端测试的程序构造方法

- **基于生成的程序构造**：生成完全有效 (不含未定义行为的程序)

主要缺点：程序生成器很难再检测出新缺陷或过于强调未定义

- **基于变异的程序构造**：生成完全有效 (不含未定义行为的程序)

Orion[PLDI'2014], Athena[OOPLSA'2015],Hermes[OOPLSA'2016]

➢ 面向编译器缺陷定位的程序构造方法

- **基于局部变异的构造技术**: 编写代码随机改变程序变量类型等，不关注证人测试程序的未定义行为

  DiWi [FSE'2019]

- **基于结构化变异的构造技术**: 编写随机改变程序的控制流等，不关注证人测试程序的未定义行为

  RecBi [ASE'2020]

研究现状

大连理工大学 Dalian University Of Technology

根据编译器基本结构（即前、中后端）及编译器质量保障的基本实现方法（测试及调试）

➢ 面向编译器前端测试的程序构造方法

- **生成完全有效的测试程序:** 顺利通过前端分析，无法检测到潜在的前端缺陷
- **生成词法无效的测试程序：**停止在前端分析早期词法分析阶段，无法检测到深层次的前端缺陷

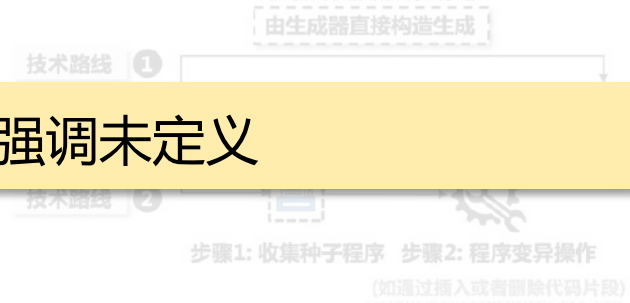
Dharma [Mozilla'2011], Grammarinator [TEST'2018]

主要缺点：快速通过前端分析或只能检测到浅层次的前端缺陷

➢ 面向编译器中后端测试的程序构造方法

- **基于生成的程序构造**：生成完全有效 (不含未定义行为的程序)
- **基于变异的程序构造**：生成完全有效 (不含未定义行为的程序)

Orion[PLDI'2014], Athena[OOPLSA'2015],Hermes[OOPLSA'2016]



主要缺点：程序生成器很难再检测出新缺陷或过于强调未定义

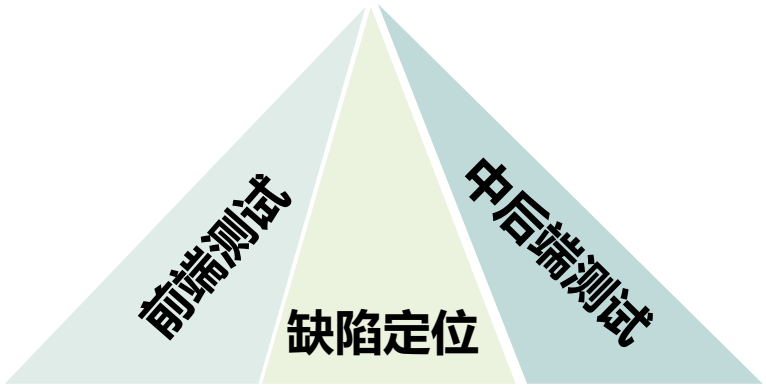
➢ 面向编译器缺陷定位的程序构造方法

- **基于局部变异的构造技术**: 编写代码随机改变程序变量类型等，不关注证人测试程序的未定义行为
- **基于结构化变异的构造技术**: 编写随机改变程序的控制流等，不关注证人测试程序的未定义行为

RecBi [ASE'2020]

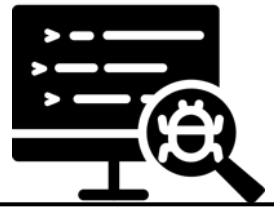
主要缺点：多样性不足、构造开销较大及不关注未定义行为

本文拟解决的关键问题



本文拟解决的关键问题

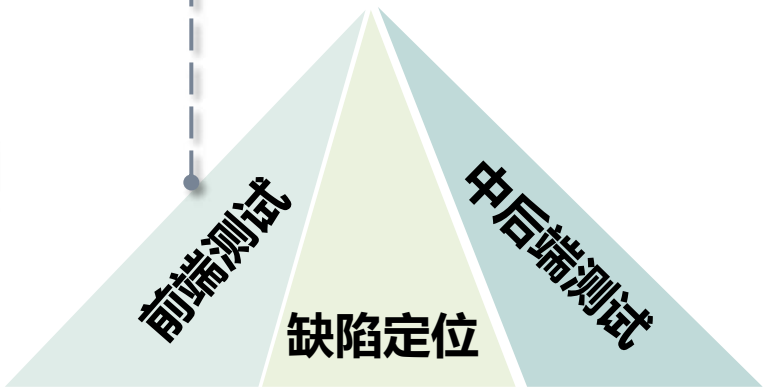
快速通过前端分析



只能检测浅层次缺陷

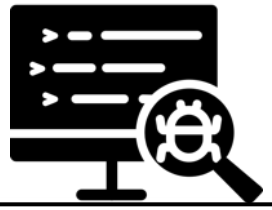
如何构造语法多样化的测试程序检测深层次的前端缺陷？

需要构造语法多样化的测试程序，即通过词法检查但因为其他错误而不能通过语义检查的程序



本文拟解决的关键问题

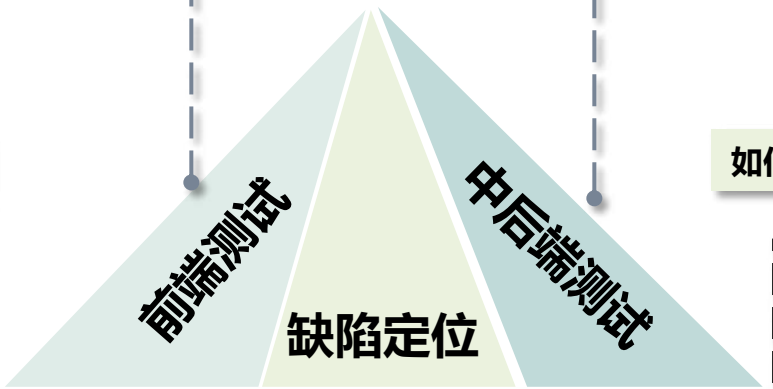
快速通过前端分析



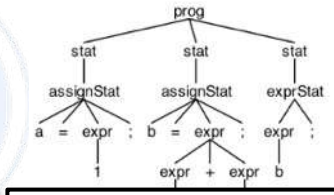
只能检测浅层次缺陷

如何构造语法多样化的测试程序检测深层次的前端缺陷？

需要构造语法多样化的测试程序，即通过词法检查但因为其他错误而不能通过语义检查的程序



程序生成器被驯化



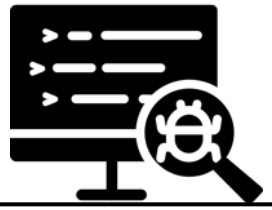
过于强调未定义

如何构造语义多样化的测试程序检测深层次的中后端缺陷？

需要构造语义多样化的程序，即通过语法检查单可能存在未定义行为的程序

本文拟解决的关键问题

快速通过前端分析



只能检测浅层次缺陷

如何构造语法多样化的测试程序检测深层次的前端缺陷？

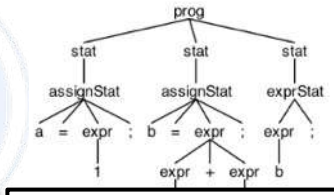
需要构造语法多样化的测试程序，即通过词法检查但因为其他错误而不能通过语义检查的程序

前端测试

缺陷定位

中后端测试

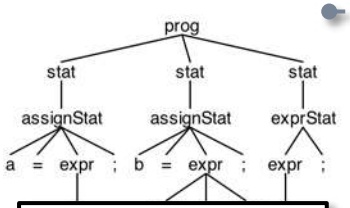
程序生成器被驯化



过于强调未定义

如何构造语义多样化的测试程序检测深层次的中后端缺陷？

需要构造语义多样化的程序，即通过语法检查单可能存在未定义行为的程序



语义多样性不足

构造开销大

不关注未定义行为

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缺陷定位？

需要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，且保证构造的开销尽可能小

研究内容及思路

研究内容及思路

大连理工大学 Dalian University Of Technology

Lab.Oscar www.oscar-lab.org

面向编译器测试及调试的程序构造方法研究

研究内容及思路

大连理工大学 Dalian University Of Technology

Lab.Oscar www.oscar-lab.org

面向编译器测试及调试的程序构造方法研究

应用场景

构造需求

存在挑战

研究思路

解决方案

编译器缺陷

编译器缺陷检测

编译器缺陷定位

研究内容及思路

面向编译器测试及调试的程序构造方法研究

应用场景

编译器测试（前端）

构造需求

触发深层次前端缺陷

（词法有效）语法多样性

存在挑战

设计多样化的生成结构 + 选择代表性的生成结构

研究思路

研究工作一

多粒度突变策略 + 等概率结构选择

解决方案

面向编译器前端测试的结构感知构造方法（第三章）

编译器缺陷

编译器缺陷检测

编译器缺陷定位

研究内容及思路

大连理工大学 Dalian University Of Technology

面向编译器测试及调试的程序构造方法研究

应用场景	**编译器测试**（前端）	**编译器测试**（中后端）
构造需求	**触发深层次前端缺陷** （词法有效）语法多样性	**触发深层次中后端缺陷** （语法有效）语义多样性
存在挑战	设计多样化的生成结构 + 选择代表性的生成结构	轻量合成多样化的程序片段 + 选择揭示缺陷的程序片段
研究思路	**研究工作一** 多粒度突变策略 + 等概率结构选择	**研究工作二** 高效复用内置代码 + 语法覆盖率选择
解决方案	**面向编译器前端测试的结构感知构造方法（第三章）**	**面向编译器中后端测试的再造生成器构造方法（第四章）**

编译器缺陷

编译器缺陷检测　　**编译器缺陷定位**

研究内容及思路

大连理工大学 Dalian University Of Technology　Lab.Oscar www.oscar-lab.org

面向编译器测试及调试的程序构造方法研究

应用场景	**编译器测试**（前端）	**编译器测试**（中后端）	**编译器调试**（缺陷定位）
构造需求	**触发深层次前端缺陷** （词法有效）语法多样性	**触发深层次中后端缺陷** （语法有效）语义多样性	**辅助隔离可疑文件** （语义有效）权衡多样性和相似性
存在挑战	设计多样化的生成结构 + 选择代表性的生成结构	轻量合成多样化的程序片段 + 选择揭示缺陷的程序片段	制定准确的提示模版 + 选择针对缺陷的特定提示
研究思路	**研究工作一** 多粒度突变策略 + 等概率结构选择	**研究工作二** 高效复用内置代码 + 语法覆盖率选择	**研究工作三** 静态程序分析技术 + 深度强化学习 + 轻量程序验证
解决方案	**面向编译器前端测试的结构感知构造方法（第三章）**	**面向编译器中后端测试的再造生成器构造方法（第四章）**	**面向编译器缺陷定位的大模型赋能构造方法（第五章）**

编译器缺陷

编译器缺陷检测　　**编译器缺陷定位**

研究内容及思路

大连理工大学 Dalian University Of Technology　Lab.Oscar www.oscar-lab.org

面向编译器测试及调试的程序构造方法研究

	编译器测试（前端）	编译器测试（中后端）	编译器调试（缺陷定位）
应用场景			
构造需求	**触发深层次前端缺陷** （词法有效）语法多样性	**触发深层次中后端缺陷** （语法有效）语义多样性	**辅助隔离可疑文件** （语义有效）权衡多样性和相似性
存在挑战	设计多样化的生成结构 + 选择代表性的生成结构	轻量合成多样化的程序片段 + 选择揭示缺陷的程序片段	制定准确的提示模版 + 选择针对缺陷的特定提示
研究思路	**研究工作一** 多粒度突变策略 + 等概率结构选择	**研究工作二** 高效复用内置代码 + 语法覆盖率选择	**研究工作三** 静态程序分析技术 + 深度强化学习 + 轻量程序验证
解决方案	**面向编译器前端测试的结构感知构造方法（第三章）**	**面向编译器中后端测试的再造生成器构造方法（第四章）**	**面向编译器缺陷定位的大模型赋能构造方法（第五章）**

编译器缺陷

编译器缺陷检测　　**编译器缺陷定位**

02

研究工作 (1/3)

Contents of Research

面向编译器前端测试的结构感知程序构造方法

研究背景及动机

研究背景及动机

背景

- **编译器前端的重要性:** 只有当前端分析正确地执行完后，后续的编译流程才能正确执行
- **研究现状:** C++编译器应用广泛且表明是存在缺陷最多的组件，较少工作关注C++编译器测试

研究背景及动机

背景

- **编译器前端的重要性:** 只有当前端分析正确地执行完后，后续的编译流程才能正确执行
- **研究现状:** C++编译器应用广泛且表明是存在缺陷最多的组件，较少工作关注C++编译器测试

动机

研究背景及动机

背景

- **编译器前端的重要性:** 只有当前端分析正确地执行完后，后续的编译流程才能正确执行
- **研究现状:** C++编译器应用广泛且表明是存在缺陷最多的组件，较少工作关注C++编译器测试

动机

进一步实证研究表明排名前四的C++缺陷和前端有关

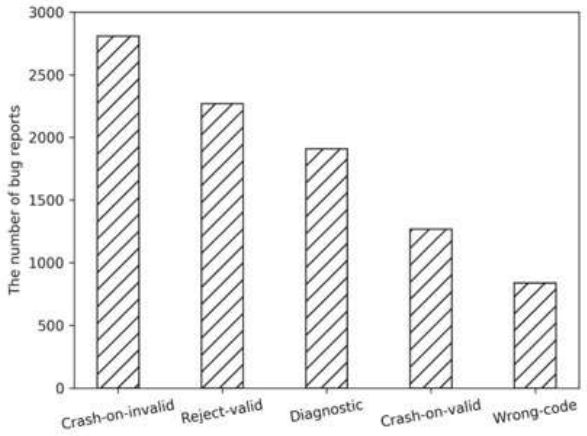


图1 和 C++相关排名前五的缺陷类型统计

研究背景及动机

背景

- **编译器前端的重要性:** 只有当前端分析正确地执行完后，后续的编译流程才能正确执行
- **研究现状:** C++编译器应用广泛且表明是存在缺陷最多的组件，较少工作关注C++编译器测试

动机

进一步实证研究表明排名前四的C++缺陷和前端有关

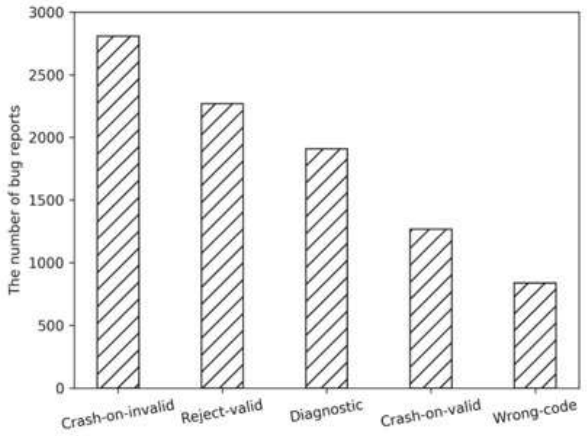


图1 和 C++相关排名前五的缺陷类型统计

现有方法构造出的测试程序要么快速通过前端分析或只能检测到浅层次的前端缺陷

研究背景及动机

背景

- **编译器前端的重要性:** 只有当前端分析正确地执行完后，后续的编译流程才能正确执行
- **研究现状:** C++编译器应用广泛且表明是存在缺陷最多的组件，较少工作关注C++编译器测试

动机

进一步实证研究表明排名前四的C++缺陷和前端有关

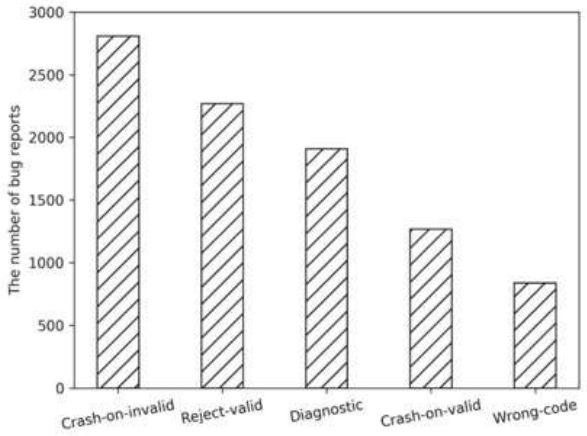


图1 和 C++相关排名前五的缺陷类型统计

现有方法构造出的测试程序要么快速通过前端分析或只能检测到浅层次的前端缺陷

科学问题：如何构造语法多样化的测试程序检测深层次编译器前端缺陷？

挑战

- **灵活化的生成结构设计：** 直接在源程序上突变无法保证程序的正确性，现有语法定义格式能力有限
- **代表性的生成结构选择：** 语法元素数量庞大，选择包含更多语法特性（即代表性）的具有挑战性

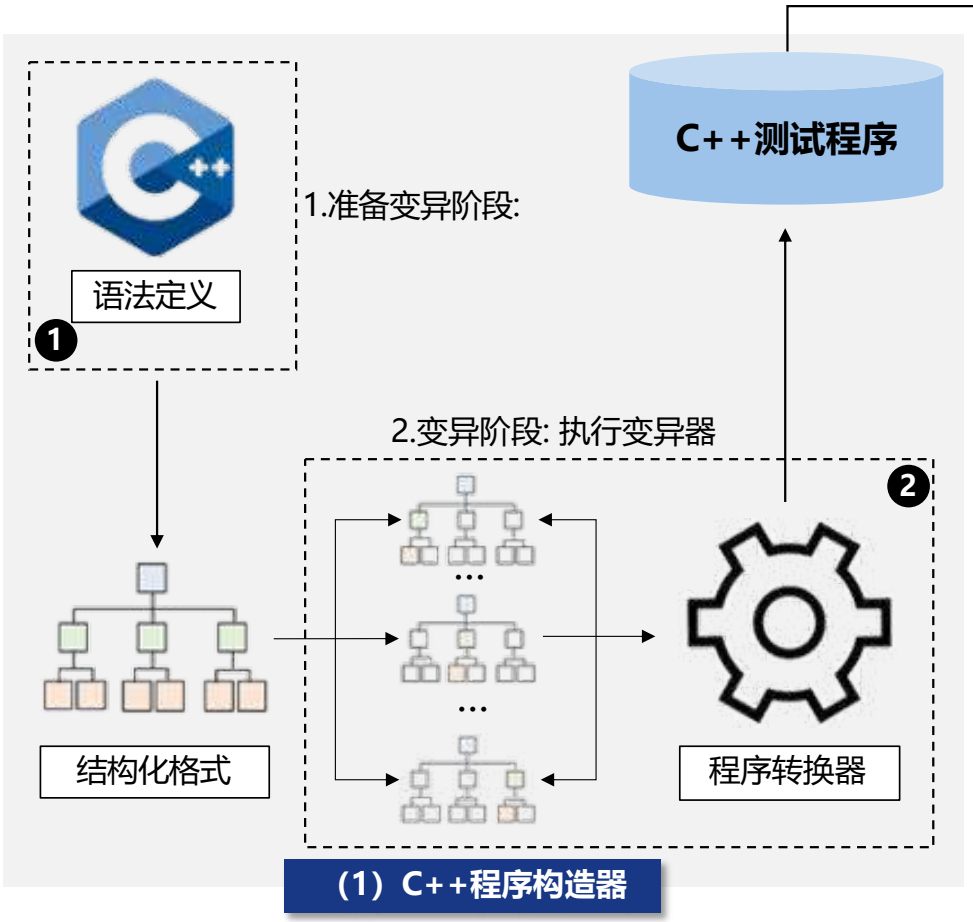
本文方法: CCOFT

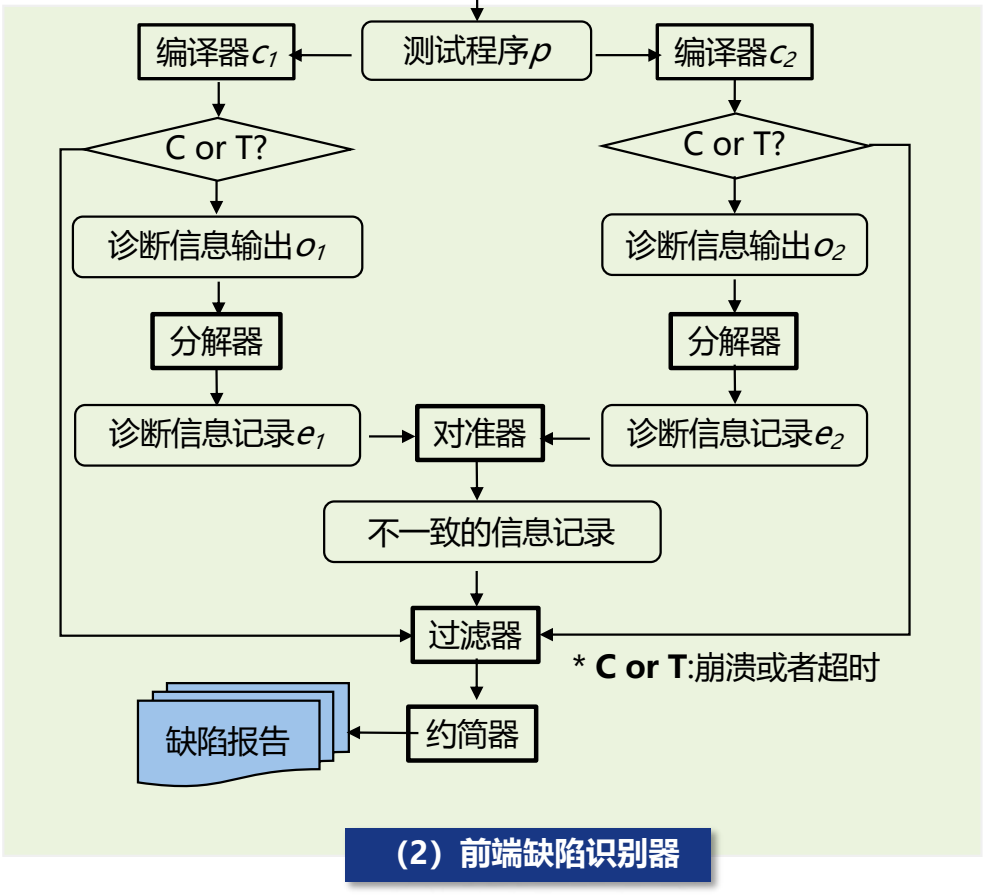
本文方法: CCOFT

为了检测深层次前端缺陷，本文提出结构感知的构造方法----CCOFT = C++ COmpiler Front-end Tester

本文方法: CCOFT

为了检测深层次前端缺陷，本文提出结构感知的构造方法----CCOFT = C++ COmpiler Front-end Tester

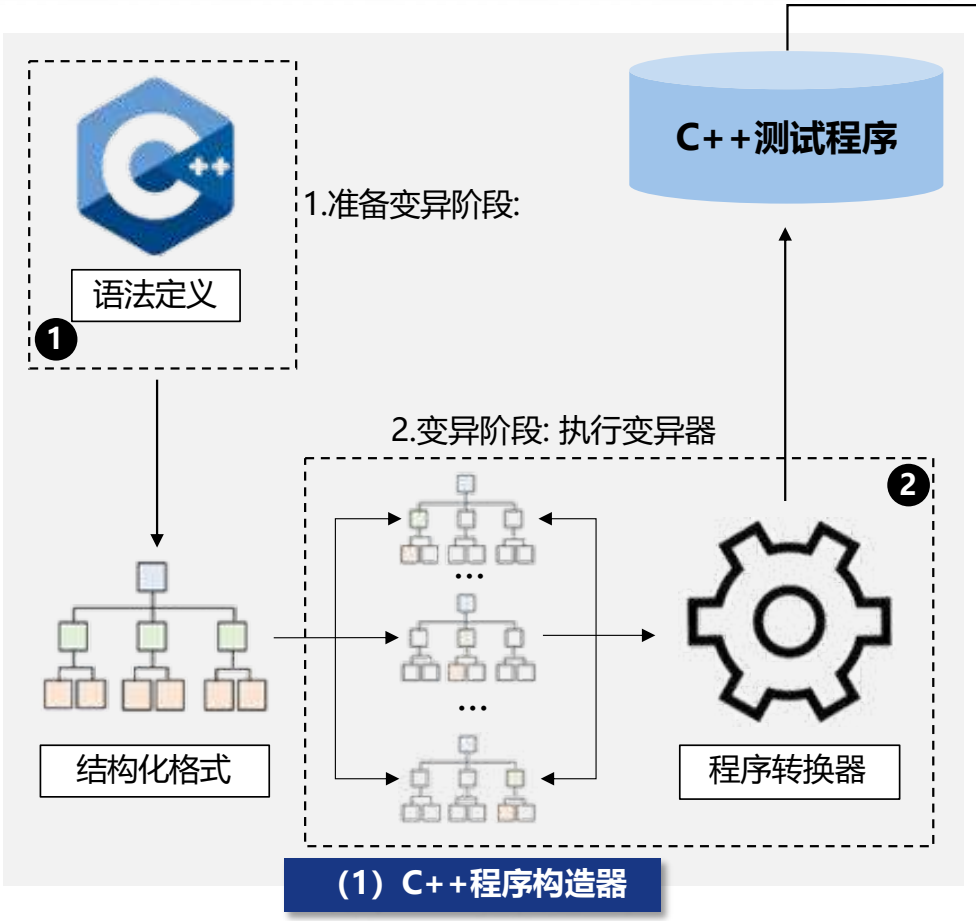


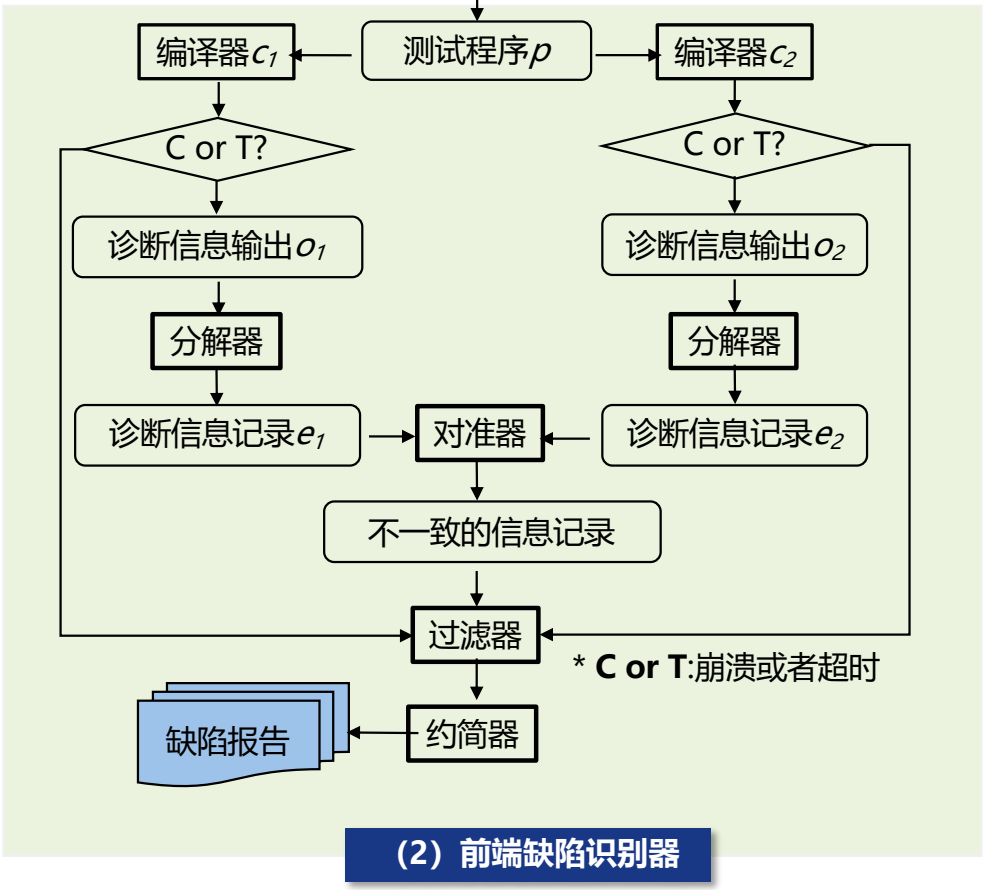


本文方法: CCOFT

为了检测深层次前端缺陷，本文提出结构感知的构造方法----CCOFT = C++ COmpiler Front-end Tester

(1) C++程序构造器: 通过将语法定义转换为灵活的结构定义，在灵活结构上执行多粒度变异操作，最终生成多样化的测试程序



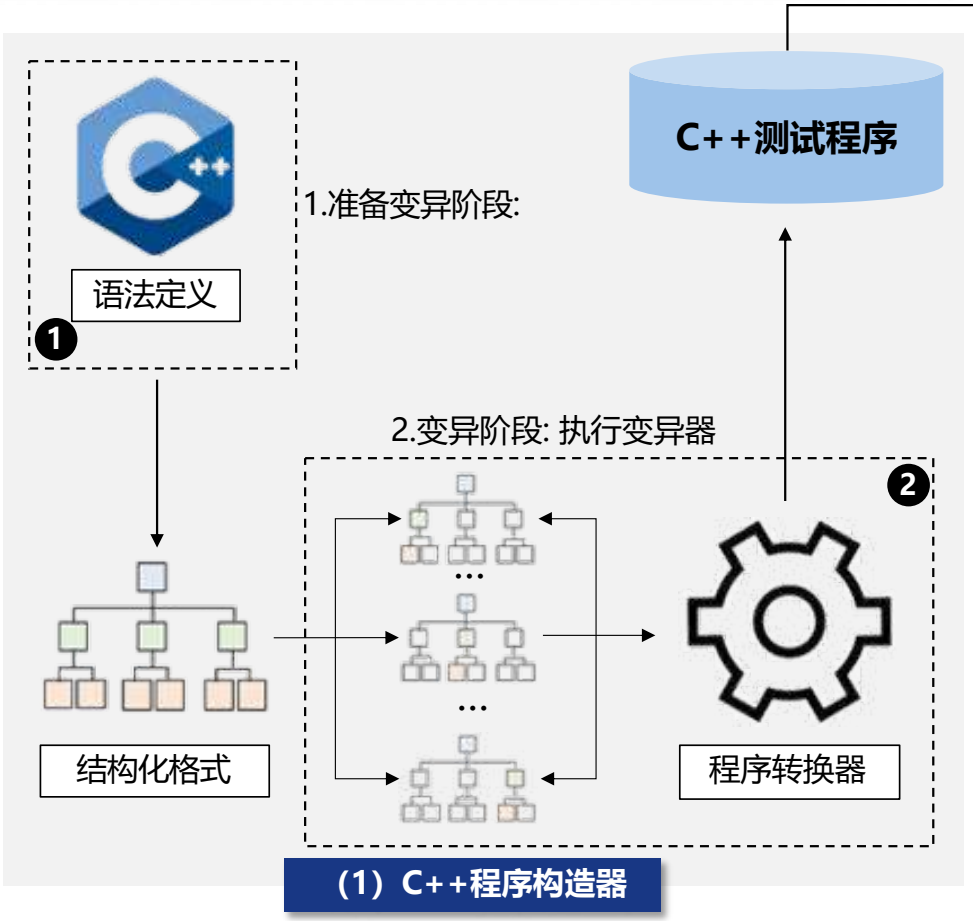


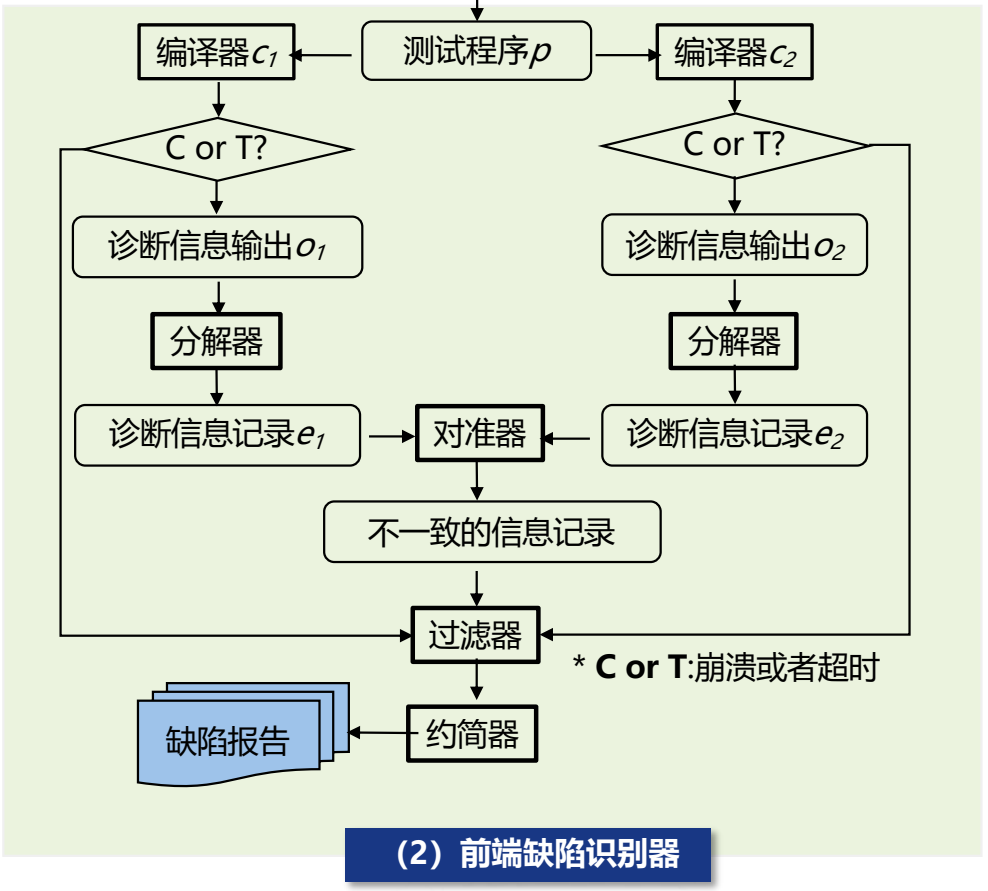
本文方法: CCOFT

为了检测深层次前端缺陷，本文提出结构感知的构造方法----CCOFT = C++ COmpiler Front-end Tester

(1) C++程序构造器: 通过将语法定义转换为灵活的结构定义，在灵活结构上执行多粒度变异操作，最终生成多样化的测试程序

(2) 前端缺陷识别器: 编译器输出信息对齐，结合差分策略有效识别各种编译器前端缺陷





CCOFT关键技术

CCOFT由三个关键技术组成: 结构感知模版构建+结构感知变异+差分及对齐算法

CCOFT关键技术

CCOFT由三个关键技术组成: 结构感知模版构建+结构感知变异+差分及对齐算法

为了解决“设计灵活化的生成结构”挑战

CCOFT关键技术

CCOFT由三个关键技术组成: 结构感知模版构建+结构感知变异+差分及对齐算法

为了解决“设计灵活化的生成结构”挑战

结构感知模版的构建

- **语法结构转换：**将语法结构转换为特定的结构化格式
- **变异执行准备：**支持多粒度突变，包括粗粒度的结构化变异和细粒度的变量值变异

CCOFT关键技术

CCOFT由三个关键技术组成: 结构感知模版构建+结构感知变异+差分及对齐算法

为了解决“设计灵活化的生成结构”挑战

为了解决“选择多样化的生成结构”挑战

结构感知模版的构建

- **语法结构转换：**将语法结构转换为特定的结构化格式
- **变异执行准备：**支持多粒度突变，包括粗粒度的结构化变异和细粒度的变量值变异

CCOFT关键技术

CCOFT由三个关键技术组成: 结构感知模版构建+结构感知变异+差分及对齐算法

为了解决“设计灵活化的生成结构”挑战

结构感知模版的构建

- **语法结构转换：**将语法结构转换为特定的结构化格式
- **变异执行准备：**支持多粒度突变，包括粗粒度的结构化变异和细粒度的变量值变异

为了解决“选择多样化的生成结构”挑战

结构感知的变异操作

- **等概率选择策略：**采用相同的概率选择不同的语法元素
- **结构代码转换：**将多样化的结构转换为有效的测试程序

CCOFT关键技术

大连理工大学 Dalian University Of Technology　Lab.Oscar www.oscar-lab.org

CCOFT由三个关键技术组成: 结构感知模版构建+结构感知变异+差分及对齐算法

为了解决“设计灵活化的生成结构”挑战

结构感知模版的构建

- **语法结构转换：**将语法结构转换为特定的结构化格式
- **变异执行准备：**支持多粒度突变，包括粗粒度的结构化变异和细粒度的变量值变异

为了解决“选择多样化的生成结构”挑战

结构感知的变异操作

- **等概率选择策略：**采用相同的概率选择不同的语法元素
- **结构代码转换：**将多样化的结构转换为有效的测试程序

基于差分测试与信息对齐算法的前端缺陷识别

- **崩溃或超时缺陷检测，跨版本策略与跨编译器策略，跨标准策略**
- **编译器输出信息分解算法：**将编译器输出记录成对
- **错误诊断信息记录对准算法：**提取不一致的信息
- **崩溃缺陷与不一致信息过滤算法：**过滤重复信息

实验结果: CCOFT的有效性 (1/2)

实验结果: CCOFT的有效性 (1/2)

对比策略: 与两个目前最好的方法进行对比，比较检测到的缺陷总数和唯一缺陷数（实验一）

实验结果: CCOFT的有效性 (1/2)

对比策略: 与两个目前最好的方法进行对比，比较检测到的缺陷总数和唯一缺陷数（实验一）

检测到的缺陷总数

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

实验结果: CCOFT的有效性 (1/2)

对比策略: 与两个目前最好的方法进行对比，比较检测到的缺陷总数和唯一缺陷数（实验一）

检测到的缺陷总数

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

实验结论：CCOFT具有更好的缺陷检测能力，对Dharma和Grammarinator的提高分别达 **135%** 和 **111%**

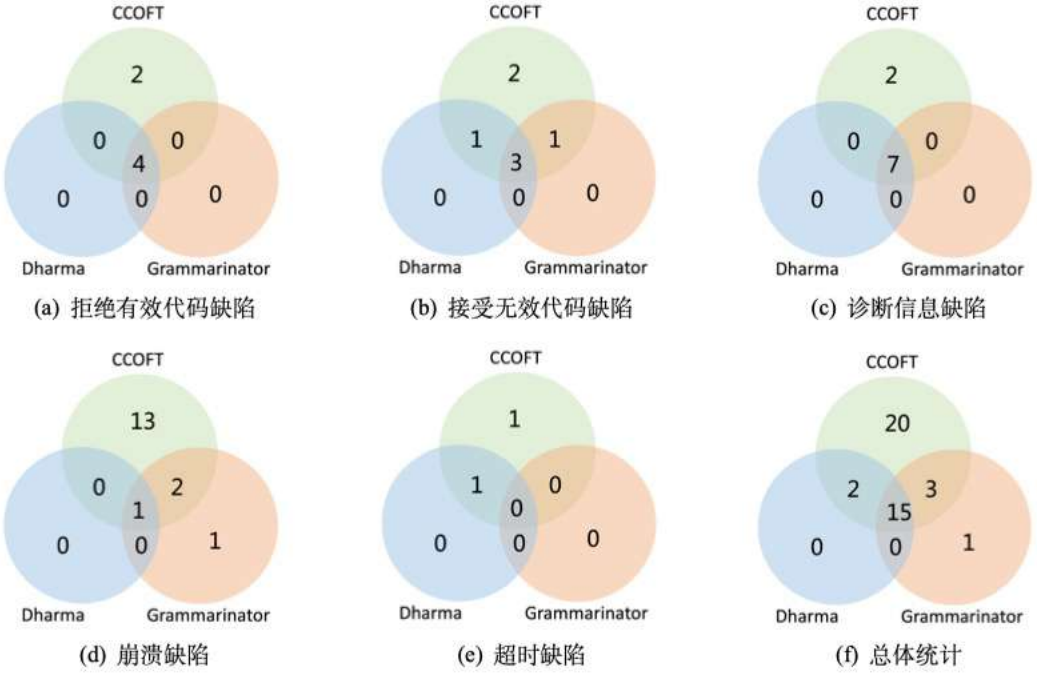
实验结果: CCOFT的有效性 (1/2)

对比策略: 与两个目前最好的方法进行对比，比较检测到的缺陷总数和唯一缺陷数（实验一）

检测到的缺陷总数

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

唯一缺陷数



实验结论：CCOFT具有更好的缺陷检测能力，对Dharma和Grammarinator的提高分别达 **135%** 和 **111%**

实验结果: CCOFT的有效性 (1/2)

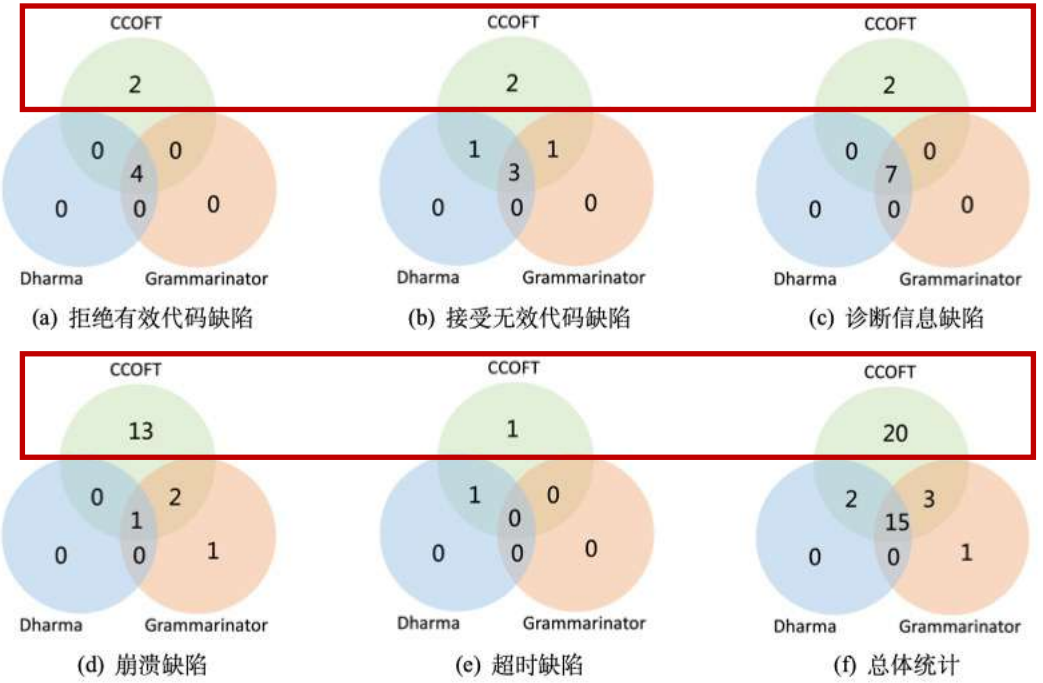
对比策略: 与两个目前最好的方法进行对比，比较检测到的缺陷总数和唯一缺陷数（实验一）

检测到的缺陷总数

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

实验结论：CCOFT具有更好的缺陷检测能力，对Dharma和Grammarinator的提高分别达 **135%** 和 **111%**

唯一缺陷数



实验结论：CCOFT可以检测到5个类型的不同编译器前端缺陷，且能检测到大部分（90%以上）的唯一缺陷

实验结果: 组件的有效性及实践检测能力 (2/2)

实验结果: 组件的有效性及实践检测能力 (2/2)

大连理工大学 Dalian University Of Technology

Lab.Oscar www.oscar-lab.org

组件有效性对比策略: 与使用新提出ECS策略的变体方法进行对比, **比较检测到的缺陷总数（实验二）**

实验结果: 组件的有效性及实践检测能力 (2/2)

组件有效性对比策略: 与使用新提出ECS策略的变体方法进行对比, **比较检测到的缺陷总数（实验二）**

检测到的缺陷总数

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

实验结果: 组件的有效性及实践检测能力 (2/2)

组件有效性对比策略: 与使用新提出ECS策略的变体方法进行对比, **比较检测到的缺陷总数（实验二）**

检测到的缺陷总数

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

实验结论：在相同的测试期间，**CCOFT** 比 CCOFT(¬ECS) 多检测出了 **82%** 的编译器缺陷，证实了新提出的ECS策略对**CCOFT**的积极贡献。

实验结果: 组件的有效性及实践检测能力 (2/2)

组件有效性对比策略: 与使用新提出ECS策略的变体方法进行对比, **比较检测到的缺陷总数（实验二）**

实践检测能力评估策略: 在最新版本的编译器上运行CCOFT，检查是否能**检测到新的前端缺陷（实验三）**

检测到的缺陷总数

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

实验结论：在相同的测试期间，**CCOFT** 比 CCOFT(¬ECS) 多检测出了 **82%** 的编译器缺陷，证实了新提出的ECS策略对**CCOFT**的积极贡献。

实验结果: 组件的有效性及实践检测能力 (2/2)

组件有效性对比策略: 与使用新提出ECS策略的变体方法进行对比, **比较检测到的缺陷总数（实验二）**

实践检测能力评估策略: 在最新版本的编译器上运行CCOFT，检查是否能**检测到新的前端缺陷（实验三）**

检测到的缺陷总数

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

检测到的前端缺陷总数

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

已确认/修复的缺陷类型

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

实验结论：在相同的测试期间，**CCOFT** 比 CCOFT(¬ECS) 多检测出了 **82%** 的编译器缺陷，证实了新提出的ECS策略对**CCOFT**的积极贡献。

实验结果: 组件的有效性及实践检测能力 (2/2)

组件有效性对比策略: 与使用新提出ECS策略的变体方法进行对比, **比较检测到的缺陷总数（实验二）**

实践检测能力评估策略: 在最新版本的编译器上运行CCOFT，检查是否能**检测到新的前端缺陷（实验三）**

检测到的缺陷总数

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

检测到的前端缺陷总数

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

已确认/修复的缺陷类型

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

实验结论：在相同的测试期间，**CCOFT** 比 CCOFT(¬ECS) 多检测出了 **82%** 的编译器缺陷，证实了新提出的ECS策略对**CCOFT**的积极贡献。

实验结论：

(1) **5** 种类型的 **136 个**编译器前端缺陷，其中**67个**被确认/修复

(2) 编译器开发者对作者提交的缺陷报告质量给予了积极肯定

02

研究工作 (2/3)

Contents of Research

面向编译器中后端测试的再造生成器程序构造方法

背景及动机

背景及动机

背景

- 编译器中后端的重要性: 中后端的缺陷通常难以察觉且可能带来巨大的可靠及安全威胁
- 程序生成器的地位: 是基于生成和基于变异方法的编译器中后端测试基石

背景及动机

大连理工大学 Dalian University Of Technology

Lab.Oscar www.oscar-lab.org

背景

- 编译器中后端的重要性: 中后端的缺陷通常难以察觉且可能带来巨大的可靠及安全威胁
- 程序生成器的地位: 是基于生成和基于变异方法的编译器中后端测试基石

动机

程序生成器很难再检测到新的中后端缺陷，能否提高程序生成器的有效性以提高其中后端缺陷检测能力？

背景及动机

背景

- 编译器中后端的重要性: 中后端的缺陷通常难以察觉且可能带来巨大的可靠及安全威胁
- 程序生成器的地位: 是基于生成和基于变异方法的编译器中后端测试基石

动机

程序生成器很难再检测到新的中后端缺陷，能否提高程序生成器的有效性以提高其中后端缺陷检测能力？

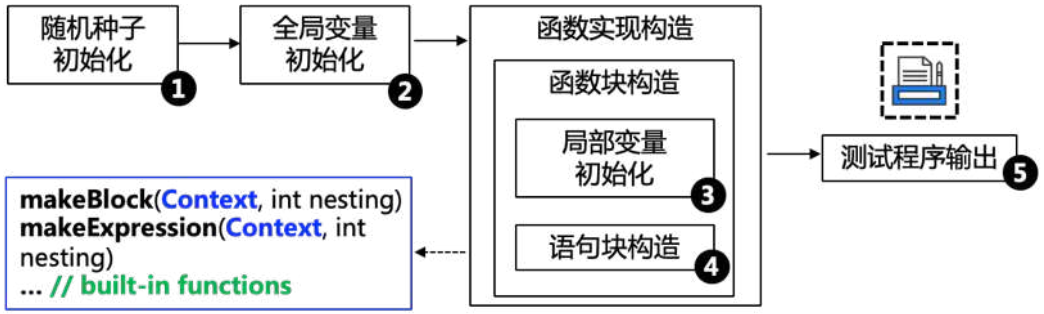


图1 典型的程序生成器工作原理

背景及动机

背景

- 编译器中后端的重要性: 中后端的缺陷通常难以察觉且可能带来巨大的可靠及安全威胁
- 程序生成器的地位: 是基于生成和基于变异方法的编译器中后端测试基石

动机

程序生成器很难再检测到新的中后端缺陷，能否提高程序生成器的有效性以提高其中后端缺陷检测能力？

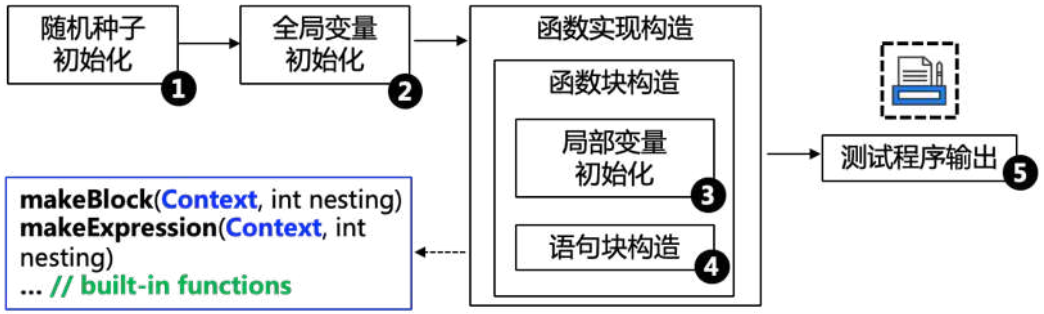


图1 典型的程序生成器工作原理

```
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} */
```

由makeBlock构造

图2 动机示例

背景及动机

大连理工大学 Dalian University Of Technology　lab.Oscar www.oscar-lab.org

背景

- 编译器中后端的重要性: 中后端的缺陷通常难以察觉且可能带来巨大的可靠及安全威胁
- 程序生成器的地位: 是基于生成和基于变异方法的编译器中后端测试基石

动机

程序生成器很难再检测到新的中后端缺陷，能否提高程序生成器的有效性以提高其中后端缺陷检测能力？

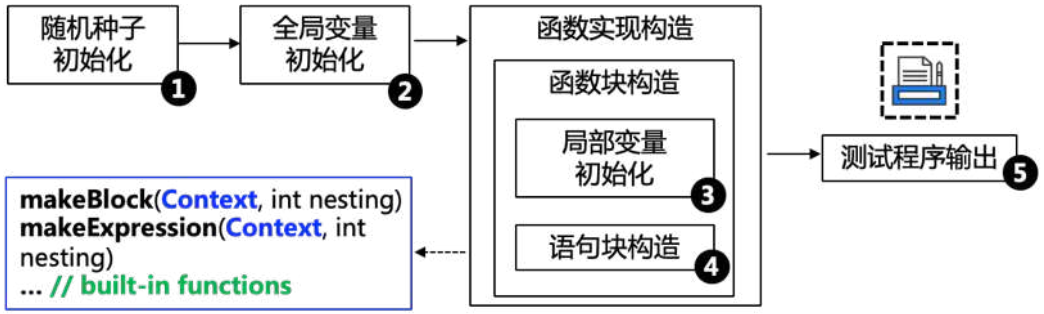


图1 典型的程序生成器工作原理

```
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} */
```

由makeBlock构造

图2 动机示例

科学问题：如何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？

（本文提出再制造的想法，将其运用到程序生成器中）

挑战

- **多样化的程序片段轻量化合成：**现有工作的合成方法都比较耗时且多样性无法衡量
- **揭示缺陷的程序片段选择：**合成的片段数量较大，如何选择尽可能揭示缺陷的片段？

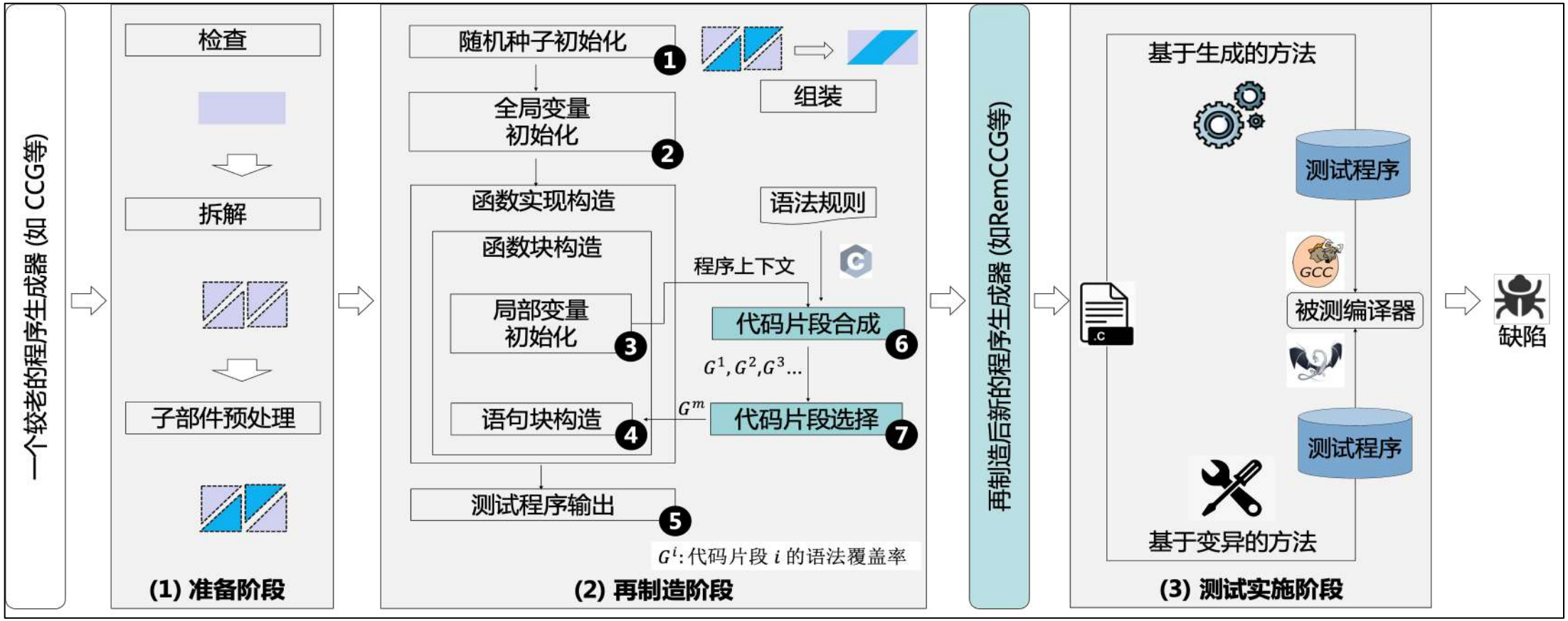
本文方法：RemGen

本文方法：RemGen

为了检测深层次中后端缺陷，本文提出再造生成器的构造方法----RemGen=Remanufacture Program Generators

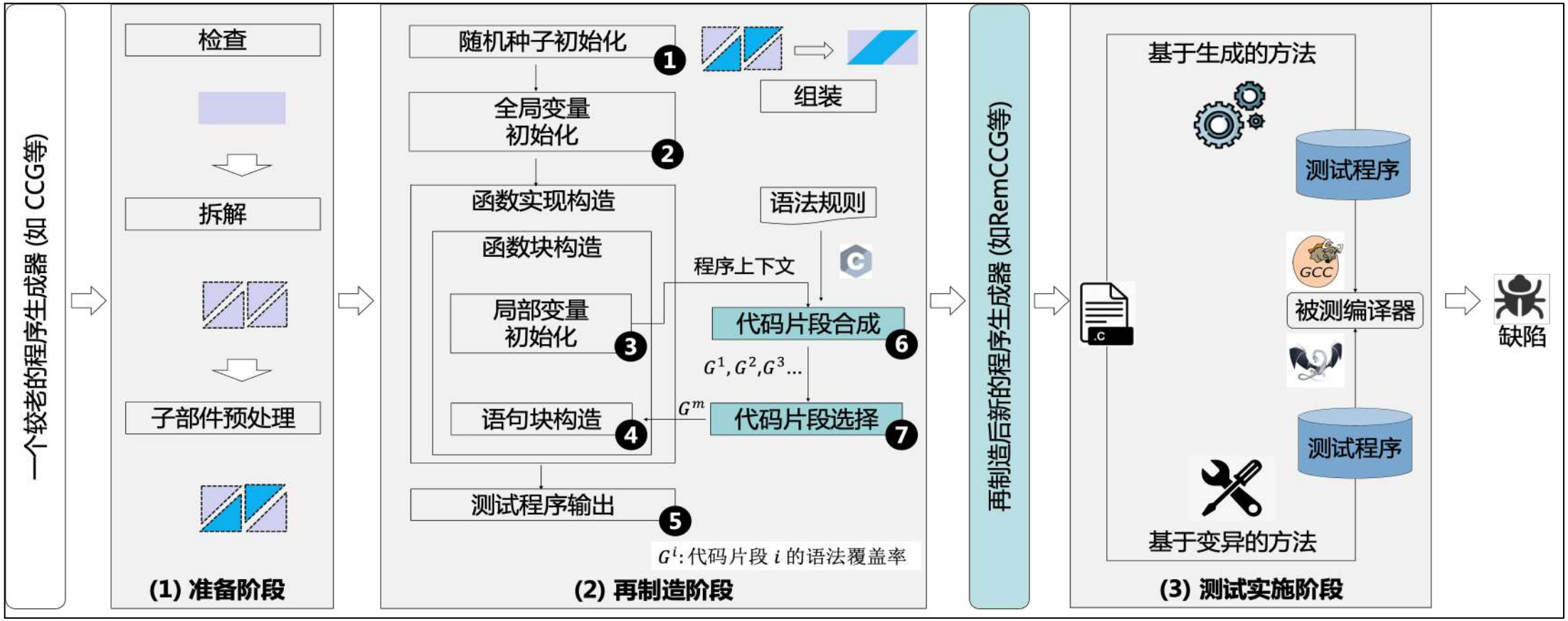
本文方法：RemGen

为了检测深层次中后端缺陷，本文提出再造生成器的构造方法----RemGen=Remanufacture Program Generators



本文方法：RemGen

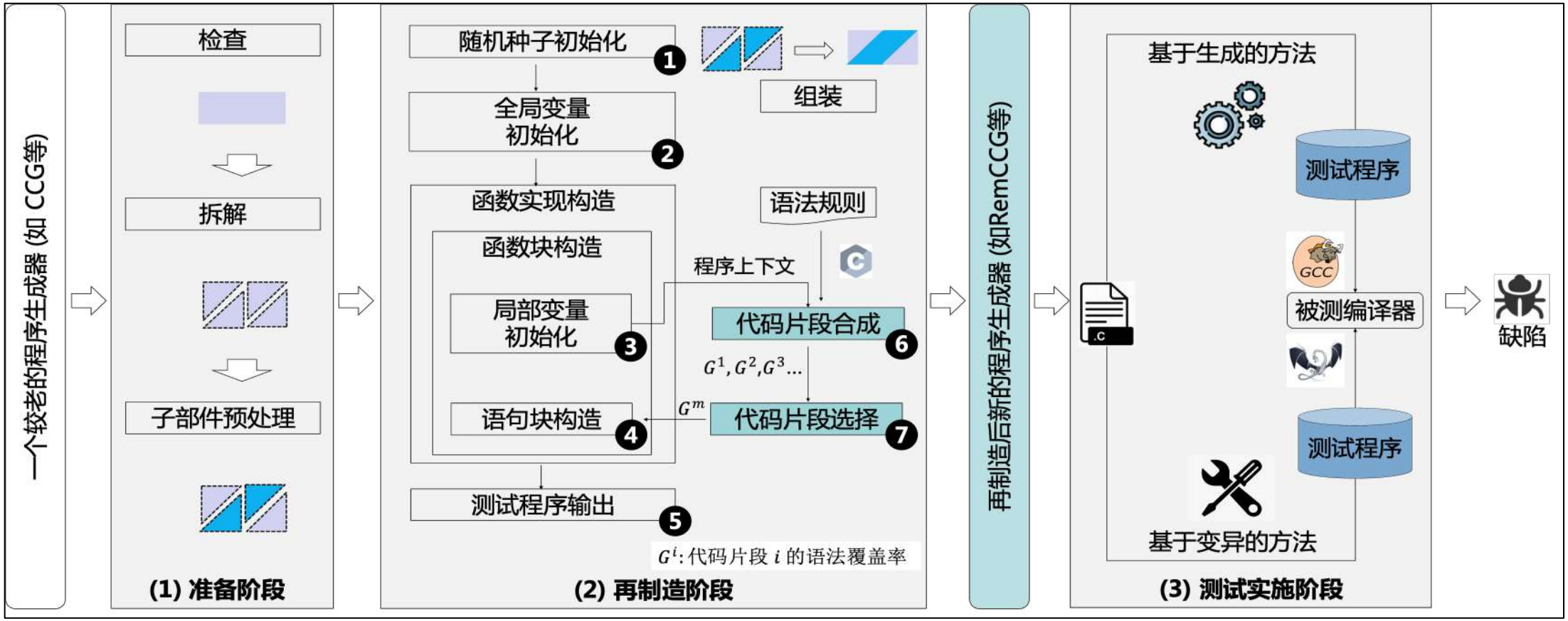
为了检测深层次中后端缺陷，本文提出再造生成器的构造方法----RemGen=Remanufacture Program Generators



(1) 准备阶段: 检查、拆卸和预处理拆卸后的程序生成器子组件

本文方法：RemGen

为了检测深层次中后端缺陷，本文提出再造生成器的构造方法----RemGen=Remanufacture Program Generators

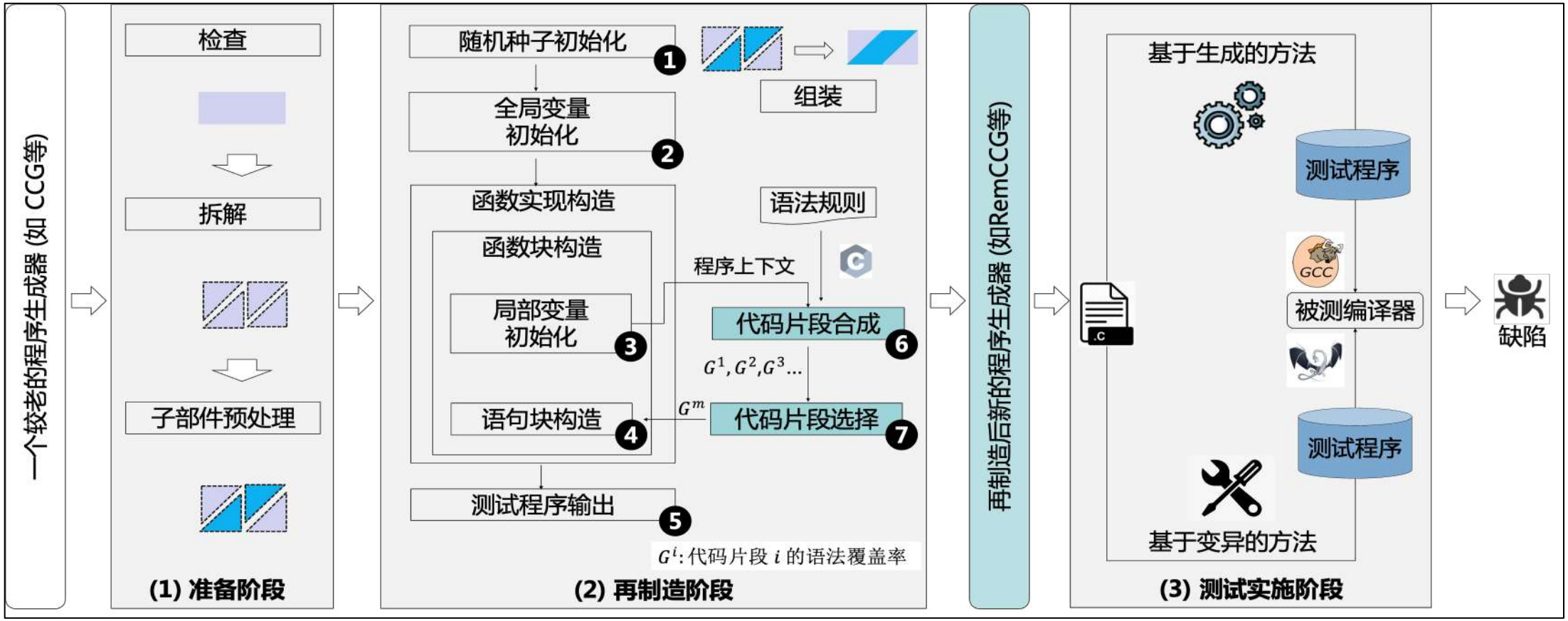


(1) 准备阶段: 检查、拆卸和预处理拆卸后的程序生成器子组件

(2) 再制造阶段: 将两个新组件整合到输入程序生成器的原始工作流程中, 最后重新组装

本文方法：RemGen

为了检测深层次中后端缺陷，本文提出再造生成器的构造方法----RemGen=Remanufacture Program Generators



(1) 准备阶段: 检查、拆卸和预处理拆卸后的程序生成器子组件

(2) 再制造阶段: 将两个新组件整合到输入程序生成器的原始工作流程中, 最后重新组装

(3) 测试阶段: 评估经再制造后新程序生成器的有效性

RemGen 关键技术

RemGen由三个关键技术组成: 再制造准备+多样化片段合成+揭示缺陷的片段选择

RemGen 关键技术

RemGen由三个关键技术组成: 再制造准备+多样化片段合成+揭示缺陷的片段选择

再制造准备

- **检查：** 代码是否公开/是否能够正常编译和运行
- **拆解：**了解各个函数的功能
- **子部件预处理：**收集上下文并将其作为外部变量

RemGen 关键技术

RemGen由三个关键技术组成: 再制造准备+多样化片段合成+揭示缺陷的片段选择

为了解决“轻量合成多样化的程序片段”挑战

再制造准备

- **检查：** 代码是否公开/是否能够正常编译和运行
- **拆解：**了解各个函数的功能
- **子部件预处理：**收集上下文并将其作为外部变量

RemGen 关键技术

RemGen由三个关键技术组成: 再制造准备+多样化片段合成+揭示缺陷的片段选择

为了解决“轻量合成多样化的程序片段”挑战

再制造准备

- **检查：** 代码是否公开/是否能够正常编译和运行
- **拆解：**了解各个函数的功能
- **子部件预处理：**收集上下文并将其作为外部变量

多样化程序片段合成组件

- **定义语法覆盖率：**量化生成程序片段的复杂性/表现力
- **上下文保留策略：**考虑全局上下文和局部上下文
- **代码片段合成：**语法辅助合成策略

RemGen 关键技术

RemGen由三个关键技术组成: 再制造准备+多样化片段合成+揭示缺陷的片段选择

为了解决“轻量合成多样化的程序片段”挑战

再制造准备

- **检查：** 代码是否公开/是否能够正常编译和运行
- **拆解：**了解各个函数的功能
- **子部件预处理：**收集上下文并将其作为外部变量

多样化程序片段合成组件

- **定义语法覆盖率：**量化生成程序片段的复杂性/表现力
- **上下文保留策略：**考虑全局上下文和局部上下文
- **代码片段合成：**语法辅助合成策略

为了解决“选择揭示缺陷的程序片段”挑战

RemGen 关键技术

RemGen由三个关键技术组成: 再制造准备+多样化片段合成+揭示缺陷的片段选择

为了解决“轻量合成多样化的程序片段”挑战

再制造准备

- **检查：** 代码是否公开/是否能够正常编译和运行
- **拆解：**了解各个函数的功能
- **子部件预处理：**收集上下文并将其作为外部变量

多样化程序片段合成组件

- **定义语法覆盖率：**量化生成程序片段的复杂性/表现力
- **上下文保留策略：**考虑全局上下文和局部上下文
- **代码片段合成：**语法辅助合成策略

为了解决“选择揭示缺陷的程序片段”挑战

揭示缺陷的程序片段选择组件

- **基于语法覆盖率的选择：**测量候选语法覆盖率和坐标原点之间的欧几里得距离
- **选择插入位置：**插入至函数体最后一个语句之后

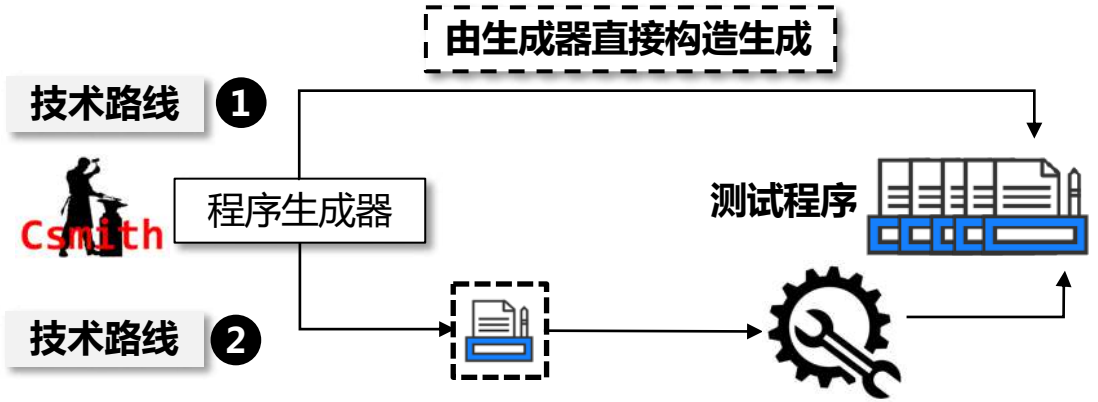
实验结果: RemGen 的有效性 (1/2)

实验结果: RemGen 的有效性 (1/2)

对比策略: 与两个目前最好的方法进行对比，比较检测到的缺陷总数（实验一）

实验结果: RemGen 的有效性 (1/2)

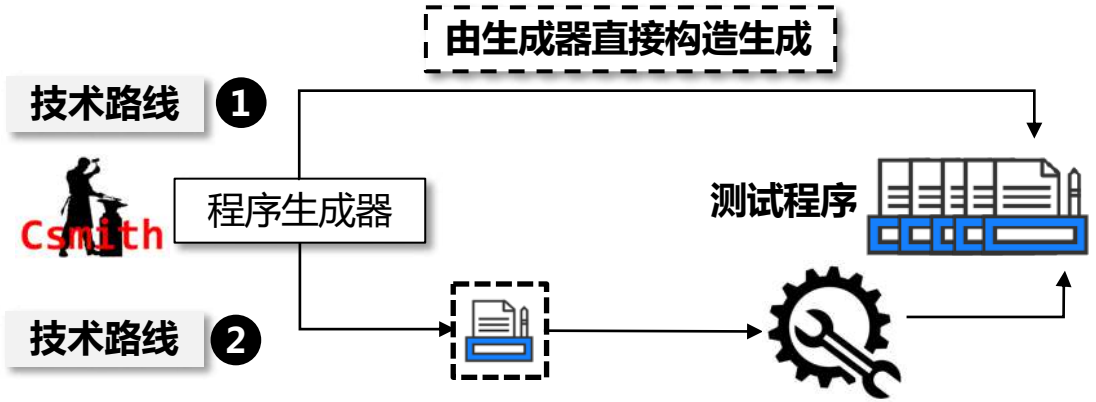
对比策略: 与两个目前最好的方法进行对比，比较检测到的缺陷总数 （实验一）



根据不同的技术路线进行两组实验，包括与基于生成以及基于变异的方法对比

实验结果: RemGen 的有效性 (1/2)

对比策略: 与两个目前最好的方法进行对比，比较检测到的缺陷总数 （实验一）



步骤1: 收集种子程序　步骤2: 程序变异操作

(如通过插入或者删除代码片段)

根据不同的技术路线进行两组实验，包括与基于生成以及基于变异的方法对比

与基于生成的方法对比结果

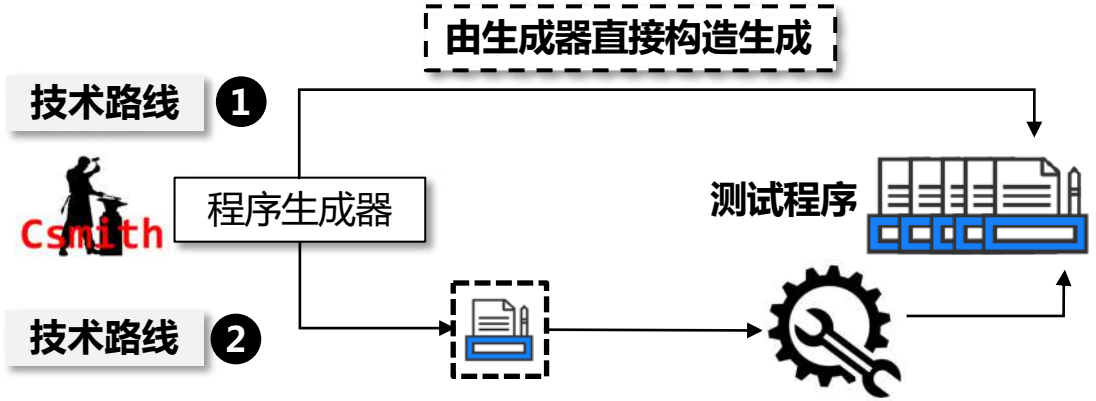
编译器	对比方法	Average Statistics			
		Crash	*Performance*	*Sum*	*Improvment*
GCC	CCG[111]	2.9	0.3	3.2	16%
	RemCCG	3.1	0.6	3.7	-
LLVM	CCG[111]	9.2	2.7	11.9	11%
	RemCCG	9.7	3.5	13.2	-

与基于变异的方法对比结果

编译器	对比方法	Average Statistics			
		Crash	*Performance*	*Sum*	*Improvment*
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	-

实验结果: RemGen 的有效性 (1/2)

对比策略: 与两个目前最好的方法进行对比，比较检测到的缺陷总数（实验一）



根据不同的技术路线进行两组实验，包括与基于生成以及基于变异的方法对比

与基于生成的方法对比结果

编译器	对比方法	Average Statistics			
		Crash	*Performance*	*Sum*	*Improvment*
GCC	CCG[111]	2.9	0.3	3.2	16%
	RemCCG	3.1	0.6	3.7	-
LLVM	CCG[111]	9.2	2.7	11.9	11%
	RemCCG	9.7	3.5	13.2	-

与基于变异的方法对比结果

编译器	对比方法	Average Statistics			
		Crash	*Performance*	*Sum*	*Improvment*
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	-

实验结论

（1）与基于生成的方法对比，**RemCCG**在GCC和LLVM编译器上分别比 CCG 提高 **16% 和 11%**

（2）与基于变异的方法对比，**RemCCG**在GCC和LLVM编译器上分别比 CCG 提高 **14% 和 11%**

实验结果: 组件的有效性及实践检测能力 (2/2)

实验结果: 组件的有效性及实践检测能力 (2/2)

组件有效性对比策略: 与各种的变体方法进行对比，**比较检测到的缺陷总数 （实验二）**

实验结果: 组件的有效性及实践检测能力 (2/2)

组件有效性对比策略: 与各种的变体方法进行对比，**比较检测到的缺陷总数 （实验二）**

检测到的缺陷总数

编译器	对比方法	Average Statistics			
		Crash	*Performance*	*Sum*	*Improvment*
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	-

实验结果: 组件的有效性及实践检测能力 (2/2)

组件有效性对比策略: 与各种的变体方法进行对比，**比较检测到的缺陷总数 （实验二）**

检测到的缺陷总数

编译器	对比方法	Average Statistics			
		Crash	*Performance*	*Sum*	*Improvment*
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	-

实验结论：RemCCG 中设计的两个组件均对 RemCCG 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%**

实验结果: 组件的有效性及实践检测能力 (2/2)

组件有效性对比策略: 与各种的变体方法进行对比，**比较检测到的缺陷总数 （实验二）**

实践检测能力评估策略: 在最新版本的编译器上运行 RemCCG, 检查是否能**检测到新的中后端缺陷（实验三）**

检测到的缺陷总数

编译器	对比方法	Average Statistics			
		Crash	*Performance*	*Sum*	*Improvment*
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	-

实验结论：RemCCG 中设计的两个组件均对 RemCCG 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%**

实验结果: 组件的有效性及实践检测能力 (2/2)

大连理工大学 Dalian University Of Technology

lab.Oscar www.oscar-lab.org

组件有效性对比策略: 与各种的变体方法进行对比，**比较检测到的缺陷总数 （实验二）**

实践检测能力评估策略: 在最新版本的编译器上运行 RemCCG, 检查是否能**检测到新的中后端缺陷（实验三）**

检测到的缺陷总数

编译器	对比方法	Average Statistics			
		Crash	*Performance*	*Sum*	*Improvment*
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	-

检测到的中后端缺陷总数

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

已确认/修复的缺陷类型

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

实验结论：RemCCG 中设计的两个组件均对 RemCCG 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%**

实验结果: 组件的有效性及实践检测能力 (2/2)

大连理工大学 Dalian University Of Technology

lab.Oscar www.oscar-lab.org

组件有效性对比策略: 与各种的变体方法进行对比，**比较检测到的缺陷总数 （实验二）**

实践检测能力评估策略: 在最新版本的编译器上运行 RemCCG, 检查是否能**检测到新的中后端缺陷（实验三）**

检测到的缺陷总数

编译器	对比方法	Average Statistics			
		Crash	*Performance*	*Sum*	*Improvment*
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	-

检测到的中后端缺陷总数

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

已确认/修复的缺陷类型

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

实验结论：RemCCG 中设计的两个组件均对 RemCCG 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%**

实验结论：RemCCG 在实际应用中能够有效地检测深层次的编译器中后端缺陷，总共报告了 GCC 和 LLVM编译器中**2**种类型的**56**个重要缺陷，其中**37**个缺陷已经被开发人员修复

02

研究工作 (3/3)

Contents of Research

面向编译器缺陷定位的大模型赋能程序构造方法

背景及动机

背景及动机

背景

- 编译器调试的必要性：快速有效地定位缺陷可以帮助开发者及时修复缺陷
- 证人测试程序构造: 主流编译器缺陷定位转化为程序构造问题

背景及动机

背景

- 编译器调试的必要性：快速有效地定位缺陷可以帮助开发者及时修复缺陷
- 证人测试程序构造: 主流编译器缺陷定位转化为程序构造问题

动机

现有方法生成的证人测试程序多样性不足、构造开销大且不关注含有未定义行为的程序

背景及动机

背景

- 编译器调试的必要性：快速有效地定位缺陷可以帮助开发者及时修复缺陷
- 证人测试程序构造: 主流编译器缺陷定位转化为程序构造问题

动机

现有方法生成的证人测试程序多样性不足、构造开销大且不关注含有未定义行为的程序

科学问题：如何构造语义完全有效权衡多样化和相似化的测试程序辅助编译器缺陷定位

(本文提出挖掘大模型潜能的想法，将其运用于证人测试程序构造)

挑战

- **制定精确的提示：**精确的提示是挖掘大模型潜能的重要组成部分
- **选择特定的提示：**每个触发缺陷的程序特征都不相同，使用统一的提示是无效的

背景及动机

背景

- 编译器调试的必要性：快速有效地定位缺陷可以帮助开发者及时修复缺陷
- 证人测试程序构造: 主流编译器缺陷定位转化为程序构造问题

动机

现有方法生成的证人测试程序多样性不足、构造开销大且不关注含有未定义行为的程序

```
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 }
```

(a) 失败测试程序

```
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 }
```

(b) 证人测试程序

科学问题：如何构造语义完全有效权衡多样化和相似化的测试程序辅助编译器缺陷定位

(本文提出挖掘大模型潜能的想法，将其运用于证人测试程序构造)

挑战

- **制定精确的提示：**精确的提示是挖掘大模型潜能的重要组成部分
- **选择特定的提示：**每个触发缺陷的程序特征都不相同，使用统一的提示是无效的

背景及动机

背景

- 编译器调试的必要性：快速有效地定位缺陷可以帮助开发者及时修复缺陷
- 证人测试程序构造: 主流编译器缺陷定位转化为程序构造问题

动机

现有方法生成的证人测试程序多样性不足、构造开销大且不关注含有未定义行为的程序

```
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 }
```

(a) 失败测试程序

```
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 }
```

由大模型构造

(b) 证人测试程序

科学问题：如何构造语义完全有效权衡多样化和相似化的测试程序辅助编译器缺陷定位

(本文提出挖掘大模型潜能的想法，将其运用于证人测试程序构造)

挑战

- **制定精确的提示：**精确的提示是挖掘大模型潜能的重要组成部分
- **选择特定的提示：**每个触发缺陷的程序特征都不相同，使用统一的提示是无效的

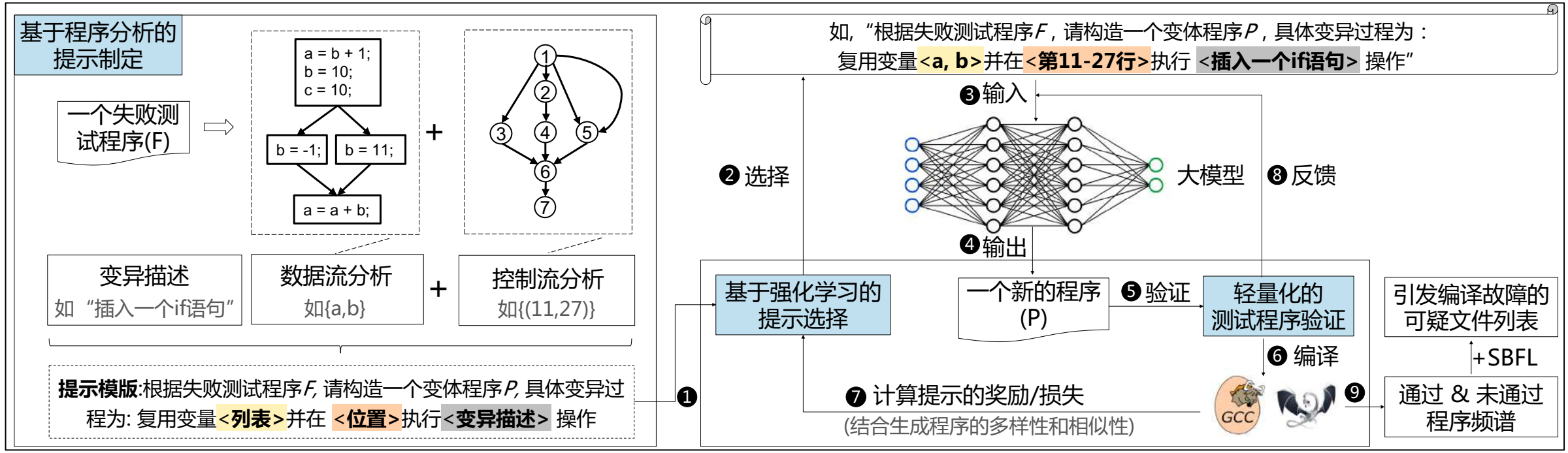
本文方法: LLM4CBI

本文方法: LLM4CBI

为了辅助定位缺陷，提出大模型赋能的构造方法----LLM4CBI (Large Language Models for Compiler Bug Isolation)

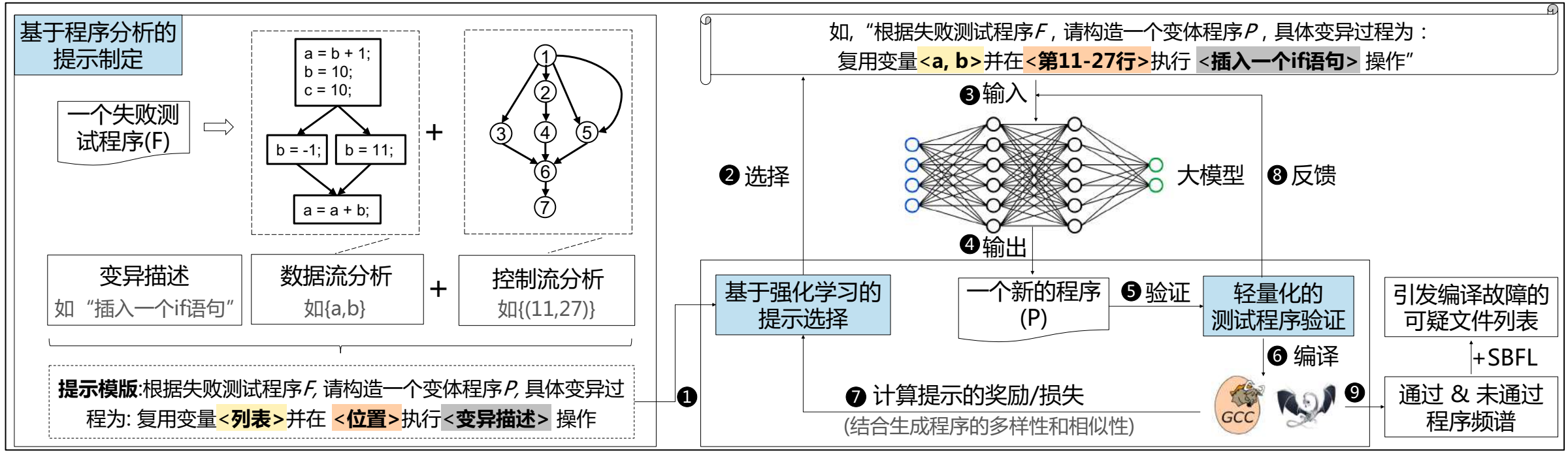
本文方法: LLM4CBI

为了辅助定位缺陷，提出大模型赋能的构造方法----LLM4CBI (Large Language Models for Compiler Bug Isolation)



本文方法: LLM4CBI

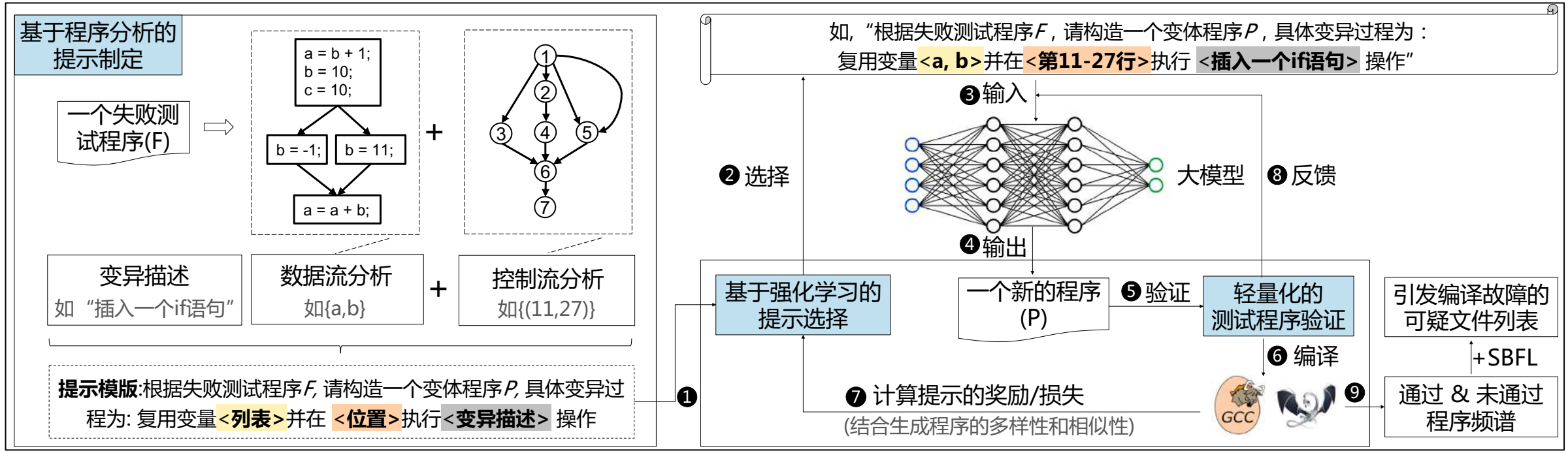
为了辅助定位缺陷，提出大模型赋能的构造方法----LLM4CBI (Large Language Models for Compiler Bug Isolation)



(1)基于程序分析的提示生成: 设计提示模版，使用数据流和控制流分析填充模版

本文方法: LLM4CBI

为了辅助定位缺陷，提出大模型赋能的构造方法----LLM4CBI (Large Language Models for Compiler Bug Isolation)

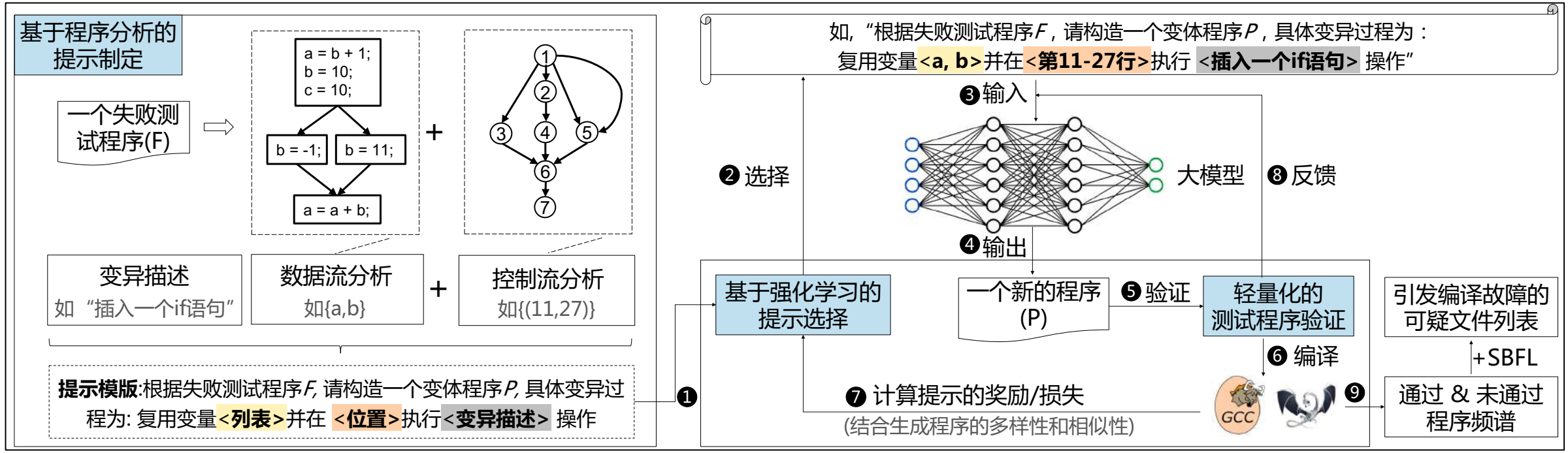


(1)基于程序分析的提示生成: 设计提示模版，使用数据流和控制流分析填充模版

(2) 基于强化学习的提示选择: 根据构造程序的多样性和相似性，计算提示奖励/损失

本文方法: LLM4CBI

为了辅助定位缺陷，提出大模型赋能的构造方法----LLM4CBI (Large Language Models for Compiler Bug Isolation)



(1)基于程序分析的提示生成: 设计提示模版，使用数据流和控制流分析填充模版

(2) 基于强化学习的提示选择: 根据构造程序的多样性和相似性，计算提示奖励/损失

(3) 轻量化测试程序验证: 对程序进行有效性验证和测试预言验证

LLM4CBI 关键技术

LLM4CBI由三个关键技术组成: 程序分析的提示生成+强化学习的提示选择+测试程序验证

LLM4CBI 关键技术

LLM4CBI由三个关键技术组成: 程序分析的提示生成+强化学习的提示选择+测试程序验证

为了解决“制定精确的提示”挑战

LLM4CBI 关键技术

LLM4CBI由三个关键技术组成: 程序分析的提示生成+强化学习的提示选择+测试程序验证

为了解决“制定精确的提示”挑战

基于程序分析的提示生成

- **提示模版的设计：** 考虑精确的突变描述
- **数据流分析：**找出最复杂的变量列表
- **控制流分析：**找出最复杂的语句位置

LLM4CBI 关键技术

LLM4CBI由三个关键技术组成: 程序分析的提示生成+强化学习的提示选择+测试程序验证

为了解决“制定精确的提示”挑战

基于程序分析的提示生成

- **提示模版的设计：** 考虑精确的突变描述
- **数据流分析：**找出最复杂的变量列表
- **控制流分析：**找出最复杂的语句位置

为了解决“选择特定的提示”挑战

LLM4CBI 关键技术

LLM4CBI由三个关键技术组成: 程序分析的提示生成+强化学习的提示选择+测试程序验证

为了解决“制定精确的提示”挑战

基于程序分析的提示生成

- **提示模版的设计：** 考虑精确的突变描述
- **数据流分析：**找出最复杂的变量列表
- **控制流分析：**找出最复杂的语句位置

为了解决“选择特定的提示”挑战

基于强化学习的提示选择

- **质量评价：** 组合相似性和多样性得分
- **计算奖励：**同时考虑历史的累积和未来的奖励
- **计算损失：**避免过于多样化或过于相似化

轻量化的测试程序验证

- **程序语义验证：**对包含任何未定义行为的测试程序进行静态形式化分析
- **测试预言验证：**检查生成的测试程序是否可以触发或者不能触发缺陷

实验结果:LLM4CBI的有效性 (1/3)

实验结果:LLM4CBI的有效性 (1/3)

对比策略: 与两个目前最好的方法进行对比，比较Top-N,MFR,MAR指标以及加速比（实验一）

实验结果:LLM4CBI的有效性 (1/3)

对比策略: 与两个目前最好的方法进行对比，比较Top-N,MFR,MAR指标以及加速比（实验一）

设置1: 相同时间

实验结果:LLM4CBI的有效性 (1/3)

对比策略: 与两个目前最好的方法进行对比，比较Top-N,MFR,MAR指标以及加速比（实验一）

设置1: 相同时间

与两种先进方法比较的编译器缺陷定位效果

Subject	Approach	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[9]	7	57.14	19	36.84	32	18.13	43	16.28	22.60	42.92	22.93	40.25
	RecBi[10]	8	37.50	24	8.33	36	13.89	45	11.11	19.67	34.42	20.13	31.94
	LLM4CBI	11	-	26	-	41	-	50	-	12.90	-	13.70	-
LLVM	DiWi[9]	4	150.00	21	33.33	27	22.22	40	20.00	27.05	49.24	27.06	48.63
	RecBi[10]	6	66.67	23	20.00	29	13.79	44	9.09	24.65	44.30	24.70	43.72
	LLM4CBI	10	-	24	-	33	-	48	-	13.73	-	13.90	-
ALL	DiWi[9]	11	90.91	41	35.14	59	25.42	83	18.07	24.83	46.36	25.00	44.79
	RecBi[10]	14	50.00	45	13.64	65	13.85	89	10.11	21.16	39.91	21.42	38.43
	LLM4CBI	21	-	48	-	74	-	98	-	13.32	-	13.80	-

注：列 "⇑ *" 表示 LLM4CBI 在各种指标上相较于比较方法的改进率（%）。

实验结果:LLM4CBI的有效性 (1/3)

对比策略: 与两个目前最好的方法进行对比，比较Top-N,MFR,MAR指标以及加速比（实验一）

设置1: 相同时间

设置2: 产生相同数量的测试程序

与两种先进方法比较的编译器缺陷定位效果

Subject	Approach	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[9]	7	57.14	19	36.84	32	18.13	43	16.28	22.60	42.92	22.93	40.25
	RecBi[10]	8	37.50	24	8.33	36	13.89	45	11.11	19.67	34.42	20.13	31.94
	LLM4CBI	11	-	26	-	41	-	50	-	12.90	-	13.70	-
LLVM	DiWi[9]	4	150.00	21	33.33	27	22.22	40	20.00	27.05	49.24	27.06	48.63
	RecBi[10]	6	66.67	23	20.00	29	13.79	44	9.09	24.65	44.30	24.70	43.72
	LLM4CBI	10	-	24	-	33	-	48	-	13.73	-	13.90	-
ALL	DiWi[9]	11	90.91	41	35.14	59	25.42	83	18.07	24.83	46.36	25.00	44.79
	RecBi[10]	14	50.00	45	13.64	65	13.85	89	10.11	21.16	39.91	21.42	38.43
	LLM4CBI	21	-	48	-	74	-	98	-	13.32	-	13.80	-

注：列 "⇑ *" 表示 LLM4CBI 在各种指标上相较于比较方法的改进率（%）。

实验结果:LLM4CBI的有效性 (1/3)

对比策略: 与两个目前最好的方法进行对比，比较Top-N,MFR,MAR指标以及加速比（实验一）

设置1: 相同时间

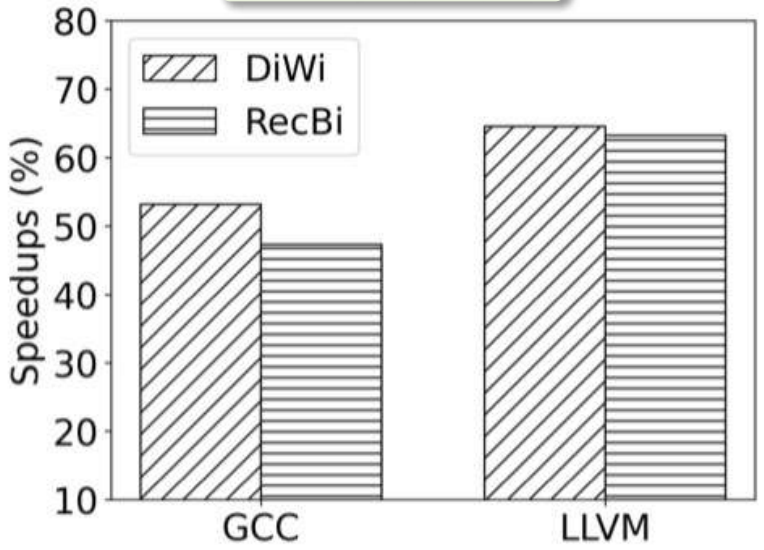
与两种先进方法比较的编译器缺陷定位效果

Subject	Approach	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[9]	7	57.14	19	36.84	32	18.13	43	16.28	22.60	42.92	22.93	40.25
	RecBi[10]	8	37.50	24	8.33	36	13.89	45	11.11	19.67	34.42	20.13	31.94
	LLM4CBI	11	-	26	-	41	-	50	-	12.90	-	13.70	-
LLVM	DiWi[9]	4	150.00	21	33.33	27	22.22	40	20.00	27.05	49.24	27.06	48.63
	RecBi[10]	6	66.67	23	20.00	29	13.79	44	9.09	24.65	44.30	24.70	43.72
	LLM4CBI	10	-	24	-	33	-	48	-	13.73	-	13.90	-
ALL	DiWi[9]	11	90.91	41	35.14	59	25.42	83	18.07	24.83	46.36	25.00	44.79
	RecBi[10]	14	50.00	45	13.64	65	13.85	89	10.11	21.16	39.91	21.42	38.43
	LLM4CBI	21	-	48	-	74	-	98	-	13.32	-	13.80	-

注：列 "⇑ *" 表示 LLM4CBI 在各种指标上相较于比较方法的改进率（%）。

设置2: 产生相同数量的测试程序

加速比比较结果



实验结果:LLM4CBI的有效性 (1/3)

对比策略: 与两个目前最好的方法进行对比，比较Top-N,MFR,MAR指标以及加速比（实验一）

设置1: 相同时间

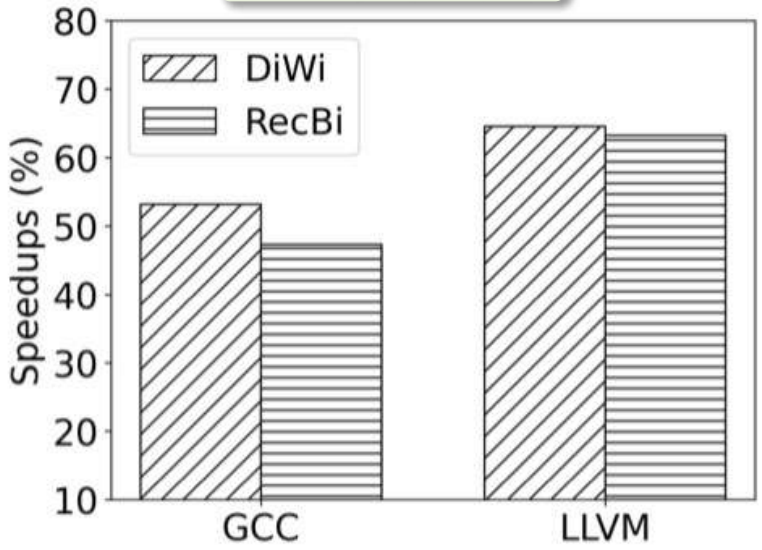
与两种先进方法比较的编译器缺陷定位效果

Subject	Approach	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[9]	7	57.14	19	36.84	32	18.13	43	16.28	22.60	42.92	22.93	40.25
	RecBi[10]	8	37.50	24	8.33	36	13.89	45	11.11	19.67	34.42	20.13	31.94
	LLM4CBI	11	-	26	-	41	-	50	-	12.90	-	13.70	-
LLVM	DiWi[9]	4	150.00	21	33.33	27	22.22	40	20.00	27.05	49.24	27.06	48.63
	RecBi[10]	6	66.67	23	20.00	29	13.79	44	9.09	24.65	44.30	24.70	43.72
	LLM4CBI	10	-	24	-	33	-	48	-	13.73	-	13.90	-
ALL	DiWi[9]	11	90.91	41	35.14	59	25.42	83	18.07	24.83	46.36	25.00	44.79
	RecBi[10]	14	50.00	45	13.64	65	13.85	89	10.11	21.16	39.91	21.42	38.43
	LLM4CBI	21	-	48	-	74	-	98	-	13.32	-	13.80	-

注：列 "⇑ *" 表示 LLM4CBI 在各种指标上相较于比较方法的改进率（%）。

设置2: 产生相同数量的测试程序

加速比比较结果



实验结论： **LLM4CBI** 在两种不同的实验配置中明显优于两种最先进的方法：**LLM4CBI** 比 DiWi 和 RecBi 更有效且高效地生成语义完全有效且权衡多样化和相似化的证人测试程序

实验结果: 新组件的有效性(2/3)

实验结果: 新组件的有效性(2/3)

对比策略: 与LLM4CBI的变体方法对比，比较Top-N,MFR,MAR指标 （实验二）

实验结果: 新组件的有效性(2/3)

对比策略: 与LLM4CBI的变体方法对比，比较Top-N,MFR,MAR指标 （实验二）

变体方法说明

- $LLM4CBI_{eq}$：使用现有的提示
- $LLM4CBI_{sq}$ ：使用特殊的提示
- $LLM4CBI_{rand}$ ：随机选择提示
- $LLM4CBI_{selnov}$ ：去掉程序验证组件

实验结果: 新组件的有效性(2/3)

对比策略: 与LLM4CBI的变体方法对比，比较Top-N,MFR,MAR指标 （实验二）

变体方法说明

- $LLM4CBI_{eq}$：使用现有的提示
- $LLM4CBI_{sq}$：使用特殊的提示
- $LLM4CBI_{rand}$：随机选择提示
- $LLM4CBI_{selnov}$：去掉程序验证组件

与两种先进方法比较的编译器缺陷定位效果

Subject	Approach	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	36	13.89	42	19.05	20.87	38.19	21.48	36.22
	LLM4CBI$_{sp}$	6	83.33	23	13.04	33	24.24	39	28.21	20.50	37.07	24.42	43.90
	LLM4CBI$_{rand}$	6	83.33	25	4.00	33	24.24	42	19.05	18.23	29.24	18.72	26.82
	LLM4CBI$_{selnov}$	8	37.50	22	18.18	35	17.14	38	31.58	19.38	33.44	19.62	30.17
	LLM4CBI	11	-	26	-	41	-	50	-	12.90	-	13.70	-
LLVM	LLM4CBI$_{ep}$	8	25.00	20	20.00	31	6.45	43	11.63	18.98	27.66	19.31	28.02
	LLM4CBI$_{sp}$	7	42.86	23	4.35	32	3.13	39	23.08	17.75	22.65	17.87	22.22
	LLM4CBI$_{rand}$	8	25.00	23	4.35	30	10.00	41	17.07	16.28	15.66	16.40	15.24
	LLM4CBI$_{selnov}$	8	25.00	20	20.00	32	3.13	45	6.67	14.85	7.54	14.88	6.59
	LLM4CBI	10	-	24	-	33	-	48	-	13.73	-	13.90	-
ALL	LLM4CBI$_{ep}$	13	61.54	44	13.64	67	10.45	85	15.29	19.93	31.17	20.40	32.34
	LLM4CBI$_{sp}$	12	75.00	46	8.70	65	13.85	78	25.64	19.13	30.38	21.15	34.74
	LLM4CBI$_{rand}$	14	50.00	49	2.04	67	10.45	83	18.07	17.26	22.83	17.56	21.41
	LLM4CBI$_{selnov}$	16	31.25	42	19.05	70	5.71	83	18.07	17.12	22.20	17.25	20.00
	LLM4CBI	21	-	50	-	74	-	98	-	13.32	-	13.80	-

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

实验结果: 新组件的有效性(2/3)

对比策略: 与LLM4CBI的变体方法对比，比较Top-N,MFR,MAR指标 （实验二）

变体方法说明

- $LLM4CBI_{eq}$ ： 使用现有的提示
- $LLM4CBI_{sq}$ ：使用特殊的提示
- $LLM4CBI_{rand}$ ：随机选择提示
- $LLM4CBI_{selnov}$ ：去掉程序验证组件

与两种先进方法比较的编译器缺陷定位效果

Subject	Approach	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	36	13.89	42	19.05	20.87	38.19	21.48	36.22
	$LLM4CBI_{sp}$	6	83.33	23	13.04	33	24.24	39	28.21	20.50	37.07	24.42	43.90
	$LLM4CBI_{rand}$	6	83.33	25	4.00	33	24.24	42	19.05	18.23	29.24	18.72	26.82
	$LLM4CBI_{selnov}$	8	37.50	22	18.18	35	17.14	38	31.58	19.38	33.44	19.62	30.17
	LLM4CBI	11	-	26	-	41	-	50	-	12.90	-	13.70	-
LLVM	$LLM4CBI_{ep}$	8	25.00	20	20.00	31	6.45	43	11.63	18.98	27.66	19.31	28.02
	$LLM4CBI_{sp}$	7	42.86	23	4.35	32	3.13	39	23.08	17.75	22.65	17.87	22.22
	$LLM4CBI_{rand}$	8	25.00	23	4.35	30	10.00	41	17.07	16.28	15.66	16.40	15.24
	$LLM4CBI_{selnov}$	8	25.00	20	20.00	32	3.13	45	6.67	14.85	7.54	14.88	6.59
	LLM4CBI	10	-	24	-	33	-	48	-	13.73	-	13.90	-
ALL	$LLM4CBI_{ep}$	13	61.54	44	13.64	67	10.45	85	15.29	19.93	31.17	20.40	32.34
	$LLM4CBI_{sp}$	12	75.00	46	8.70	65	13.85	78	25.64	19.13	30.38	21.15	34.74
	$LLM4CBI_{rand}$	14	50.00	49	2.04	67	10.45	83	18.07	17.26	22.83	17.56	21.41
	$LLM4CBI_{selnov}$	16	31.25	42	19.05	70	5.71	83	18.07	17.12	22.20	17.25	20.00
	LLM4CBI	21	-	50	-	74	-	98	-	13.32	-	13.80	-

注：列“⇑ *”表示 LLM4CBI 在各种指标上相较于其他变体方法的改进率（%）。

实验结论： LLM4CBI 中设计的三个新组件，包括精确的提示生成、记忆提示选择和轻量级测试程序验证，都有助于提高 **LLM4CBI** 缺陷定位的有效性

实验结果:LLM4CBI的可扩展性(3/3)

实验结果:LLM4CBI的可扩展性(3/3)

对比策略: 替换LLM4CBI中的大模型，比较Top-N指标 （实验三）

实验结果:LLM4CBI的可扩展性(3/3)

对比策略: 替换LLM4CBI中的大模型，比较Top-N指标 （实验三）

在LLM4CBI中评估的开源大语言模型列表

大模型	参数大小	发布日期	受欢迎程度 (GitHub)
Alpaca[154]	7B	March 2023	25.0K star
Vicuna[155]	7B	March 2023	22.8K star
GPT4ALL[156]	13B	March 2023	46K star

各种大模型的编译器缺陷定位效果对比结果

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 使用 GPT-3.5 作为内置的大模型。

实验结论：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。

05.总结与展望

三个研究工作的关系及主要成果

三个研究工作的联系和区别

大连理工大学 Dalian University Of Technology

Lab.Oscar www.oscar-lab.org

本文方法	CCOFT	RemGen	LLM4CBI
	 Detecting C++ Compiler Front-End Bugs via Grammar Mutation and Differential Testing	 Remgen: Remanufacturing a Random Program Generator for Compiler Testing	 LLM4CBI: Taming LLMs to Generate Effective Test Programs for Compiler Bug Isolation
面向场景	编译器前端缺陷检测	编译器中后端缺陷检测	编译器缺陷定位
构造需求			
语义有效：行为定义			
语义有效：类型匹配			
语法有效			
词法有效			

三个研究工作的联系和区别

本文方法	CCOFT	RemGen	LLM4CBI
			
面向场景	编译器前端缺陷检测	编译器中后端缺陷检测	编译器缺陷定位
构造需求	语法多样化		
语义有效：行为定义			
语义有效：类型匹配			
语法有效			
词法有效			

三个研究工作的联系和区别

本文方法	CCOFT	RemGen	LLM4CBI
	 Detecting C++ Compiler Front-End Bugs via Grammar Mutation and Differential Testing	 Remgen: Remanufacturing a Random Program Generator for Compiler Testing	 LLM4CBI: Taming LLMs to Generate Effective Test Programs for Compiler Bug Isolation
面向场景	编译器前端缺陷检测	编译器中后端缺陷检测	编译器缺陷定位
构造需求	语法多样化		
语义有效：行为定义	✗		
语义有效：类型匹配	✗		
语法有效	✗		
词法有效	✓		

三个研究工作的联系和区别

本文方法	CCOFT	RemGen	LLM4CBI
	 Detecting C++ Compiler Front-End Bugs via Grammar Mutation and Differential Testing	 Remgen: Remanufacturing a Random Program Generator for Compiler Testing	 LLM4CBI: Taming LLMs to Generate Effective Test Programs for Compiler Bug Isolation
面向场景	编译器前端缺陷检测	编译器中后端缺陷检测	编译器缺陷定位
构造需求	语法多样化	语义多样化	
语义有效 - 行为定义	✗		
语义有效 - 类型匹配	✗		
语法有效	✗		
词法有效	✓		

三个研究工作的联系和区别

本文方法	CCOFT	RemGen	LLM4CBI
	 Detecting C++ Compiler Front-End Bugs via Grammar Mutation and Differential Testing	 Remgen: Remanufacturing a Random Program Generator for Compiler Testing	 LLM4CBI: Taming LLMs to Generate Effective Test Programs for Compiler Bug Isolation
面向场景	编译器前端缺陷检测	编译器中后端缺陷检测	编译器缺陷定位
构造需求	语法多样化	语义多样化	
语义有效：行为定义	✗	✗	
语义有效：类型匹配	✗	✓	
语法有效	✗	✓	
词法有效	✓	✓	

三个研究工作的联系和区别

本文方法	CCOFT	RemGen	LLM4CBI
	Detecting C++ Compiler Front-End Bugs via Grammar Mutation and Differential Testing	Remgen: Remanufacturing a Random Program Generator for Compiler Testing	LLM4CBI: Taming LLMs to Generate Effective Test Programs for Compiler Bug Isolation
面向场景	编译器前端缺陷检测	编译器中后端缺陷检测	编译器缺陷定位
构造需求	语法多样化	语义多样化	权衡多样化与相似化
语义有效：行为定义	✗	✗	
语义有效：类型匹配	✗	✓	
语法有效	✗	✓	
词法有效	✓	✓	

三个研究工作的联系和区别

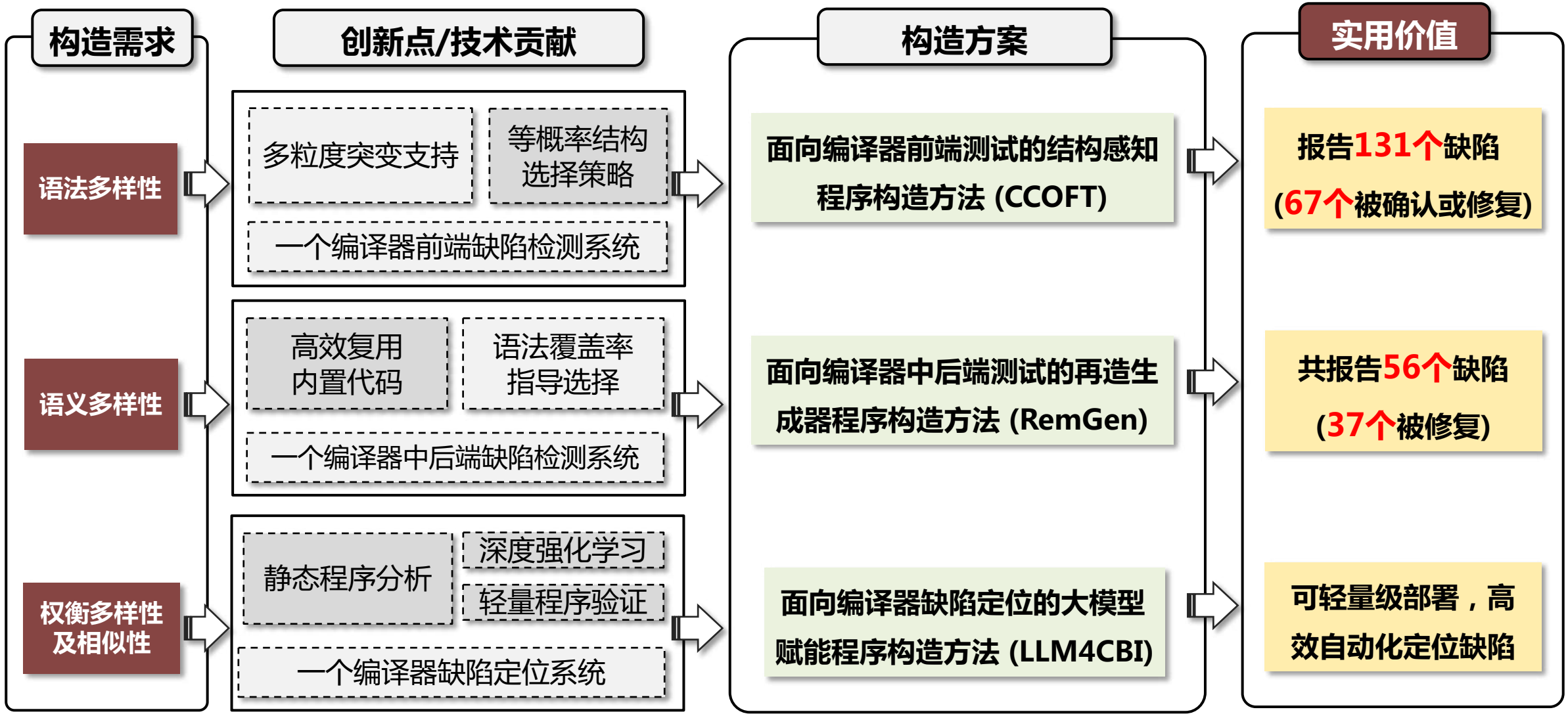
本文方法	CCOFT	RemGen	LLM4CBI
	Detecting C++ Compiler Front-End Bugs via Grammar Mutation and Differential Testing	Remgen: Remanufacturing a Random Program Generator for Compiler Testing	LLM4CBI: Taming LLMs to Generate Effective Test Programs for Compiler Bug Isolation
面向场景	编译器前端缺陷检测	编译器中后端缺陷检测	编译器缺陷定位
构造需求	语法多样化	语义多样化	权衡多样化与相似化
语义有效-行为定义	✗	✗	✓
语义有效-类型匹配	✗	✓	✓
语法有效	✗	✓	✓
词法有效	✓	✓	✓

三个研究工作的联系和区别

本文方法	CCOFT	RemGen	LLM4CBI
	Detecting C++ Compiler Front-End Bugs via Grammar Mutation and Differential Testing	Remgen: Remanufacturing a Random Program Generator for Compiler Testing	LLM4CBI: Taming LLMs to Generate Effective Test Programs for Compiler Bug Isolation
面向场景	编译器前端缺陷检测	编译器中后端缺陷检测	编译器缺陷定位
构造需求	语法多样化	语义多样化	权衡多样化与相似化
语义有效-行为定义	✗	✗	✓
语义有效-类型匹配	✗	✓	✓
语法有效	✗	✓	✓
词法有效	✓	✓	✓

有效性依次递增

创新点与本论文实用价值



面向编译器测试及调试的程序构造方法研究

未来研究方向

01 编译器缺陷检测

- 单一序列的测试
- 面向优化序列多样化的程序构造

02 编译器缺陷定位

- 细粒度的定位精度
- 进一步结合编译器缺陷历史缺陷信息

03 编译器缺陷修复

- 结合符号执行/静态分析/机器学习等多种技术的修复方法
- 结合编译器开发者的经验以及编译器缺陷仓库历史信息

博士期间学术成果总结

发表/待发表论文

- **Haoxin Tu**, He Jiang, Zhide Zhou, Yixuan Tang, Zhilei Ren, Lei Qiao, and Lingxiao Jiang, "*Detecting C++ Compiler Front-end Bugs via Grammar Mutation and Differential Testing*", in ***IEEE Transactions on Reliability***, 2022. (JCR Q1)（本学位论文第三章）
- **Haoxin Tu**, He Jiang, Xiaochen Li, Zhilei Ren, Zhide Zhou, and Lingxiao Jiang, "*RemGen: Remanufacturing A Random Program Generator for Compiler Testing*", in the ***33rd IEEE International Symposium on Software Reliability Engineering*** (ISSRE 2022). (CCF-B 国际会议)（本学位论文第四章）
- **Haoxin Tu**, Zhide Zhou, He Jiang, Imam Nur Bani Yusuf, Yuxian Li, and Lingxiao Jiang, "*LLM4CBI: Taming LLMs to Generate Effective Test Programs for Compiler Bug Isolation*", in ***IEEE Transactions on Software Engineering***. (Major Revision) (CCF-A 国际期刊)（本学位论文第五章）

专利及软件著作权

- 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: **发明专利**.（本学位论文第五章）
- 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. **软件著作权**.（本学位论文第三章）
- 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**. 软件著作权**.（本学位论文第四章）
- 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. **软件著作权**.（本学位论文第五章）

博士期间其他奖励及论文成果

博士期间参加的比赛及所获奖励

- “**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)
- “**华为杯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)
- “**大连理工大学博士生一等学业奖学金**”, 校级, 2022.11. (完成人排序: 1/1)
- “**大连理工大学优秀研究生称号**”, 校级, 2022.11. (完成人排序: 1/1)
- “**博士研究生国家奖学金**”, 国家级, 2022.12. (完成人排序: 1/1)

其他已发表/在投论文 (与第二学位论文相关)

- **Haoxin Tu**, Lingxiao Jiang, Debin Gao, He Jiang, “*Beyond a Joke: Dead Code Elimination Can Delete Live Code*”, Submitted to ***NIER Track of IEEE/ACM International Conference on Software Engineering*** **(ICSE-NIER 2024)** (Accepted) **(CCF-A 国际会议)**
- 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)
- **Haoxin Tu**, Lingxiao Jiang, Xuhua Ding, and He Jiang, “*FastKLEE: Faster Symbolic Execution via Reducing Redundant Bound Checking of Type-Safe Pointers*”, in the ***Tool Demonstrations Track of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*** **(ESEC/FSE 2022).** (Accepted) **(CCF-A 国际会议)**
- **Haoxin Tu**, Lingxiao Jiang, Jiaqi Hong, Xuhua Ding, and He Jiang, “*Concretely Mapped Symbolic Memory Locations for Memory Error Detection*”, on ***IEEE Transactions on Software Engineering***. (Major Revision) **(CCF-A 国际期刊)**

大连理工大学
Dalian University Of Technology

谢　谢！

请各位老师批评指正