
南京大学

# 本 科 生 毕 业 论 文
# (申请学士学位)

| | |
|---|---|
| 论 文 题 目 | 基于 EMI 的编译器测试工具 |
| 作 者 姓 名 | 李想 |
| 学科、专业名称 | 计算机科学与技术 |
| 研 究 方 向 | 编译器测试 |
| 指 导 教 师 | 许畅 教授 |

**2017 年 6 月 2 日**

学　　号：131220088

论文答辩日期：2017 年 6 月 8 日

指 导 教 师：　　(签字)

# EMI based compiler testing tools

by

**Shawn Lee**

Supervised by

**Prof. Chang Xu**

*A dissertation submitted to*
*the graduate school of Nanjing University*
*in partial fulfillment of the requirements*
*for the degree of*
Bachelor of Science
*in*
Computer Science


Department of Computer Science and Technology
Nanjing University

June 2, 2017

# 南京大学本科生毕业论文中文摘要首页用纸

毕业论文题目：基于 EMI 的编译器测试工具

计算机科学与技术 专业 2013 级学士生 姓名：李想

指导教师 (姓名、职称)：许畅 教授

## 摘 要

本论文介绍了一种简单、通用的编译器检测方法：EMI(equivalence modulo inputs)。该方法基于以下两步来测试编译器：1. 动态地在给定测试输入集上运行得到程序控制流；2. 基于程序控制流，生成等效程序执行比对测试。事实上，这些不同的输入自然地帮助生成 EMI 变种，以帮助我们测试常规方法难以发现的编译错误。为了实现 EMI 算法去检测目标编译器正确性，我们分析程序的执行路径，并随机删改未执行的代码以生成等价变种程序并运行。我实现的基于 EMI 算法的 Rigel 工具发现了南京大学《编译原理》课程实验项目 C-- 编译器中许多常规方法未能发现的错误。

**关键词**：编译器测试；随机代码生成；自动化测试

# 南京大学本科生毕业论文英文摘要首页用纸

THESIS: EMI based compiler testing tools

SPECIALIZATION: Computer Science

POSTGRADUATE: Shawn Lee

MENTOR: Prof. Chang Xu

## Abstract

We introduce equivalence modulo inputs (EMI), a simple, widely applicable methodology for compiler verification. This method is based on these two steps: 1) dynamically executing a program on some test inputs to get its control flow. 2) based on control flow, generate its equivalent code for comparison. Indeed, the test inputs induce a natural collection of the original program's EMI variants, which can help differentially test any compiler and specifically target the difficult-to-find miscompilations.To create a practical implementation of EMI for validating C--(a tiny subset of C) compilers, we profile a program's test executions and stochastically prune its unexecuted code. My practical realization of the EMI concept targeting C-- compilers via the 'profile and prune' strategy has led to several confirmed, unique bugs for C-- compilers from 'Principles and Techniques' of Compilers of Nanjing University. In the past, these miscompilations were hard to notice.

**Keywords**: Compiler testing; code generation; automated testing

# 目　录

# 插图清单

# 第一章　引言

编译器是计算机科学发展史上最重要的软件系统之一。人类构建代码世界，都必须要通过编译器这个中间层来与机器交互。人们一直幻想“自动写代码”，殊不知，编译器其实就是这么一个自动编写代码的工具：人类通过高级语言描述目的，然后交由编译器去自动生成机器代码。因此编译器就是人类思想与计算机硬件之间的桥梁，人类借助编译器，驾驭强大的硬件计算资源，解放和发展生产力。

凡是上了一定规模的项目，就必然会存在 Bug，很遗憾，编译器也无能幸免。由于编译器的特殊性，一旦编译器出现 Bug ，往往更难排查，并且造成的损失也更大。因为当程序出现异常输出时，程序员往往会怀疑是自己代码的错误，而不会轻易归咎于编译器（事实上，绝大多数情况下错误的根源确实不在编译器）。现有的主流编译器就那么几种，一旦任一款编译器出现任何 Bug，将影响不计其数的项目。并且这是一种“无差别杀伤”，可能影响任何系统，包括金融、军事等核心领域，对人类社会造成巨大影响。

证明代码的正确性，工业界的做法是进行足够充分的测试，学术界的做法是进行理论的证明。

单纯追求“无 Bug”，理论上是可以实现的。如通过形式语言证明的 CompCert 编译器，它在 C99 标准的子集和部分硬件平台上，通过 Coq 证明了其生成代码的完备性，专门用于对可靠性要求极高的场景，但是它牺牲了灵活性、通用性和性能。

不过人类对性能的渴望是无止境的，并且就目前而言人们已经可以接受用极小的错误率来换取性能的提升。这也是为什么软件工程中，更多的是在写测试以检验代码“没有错”，而不是来证明代码“正确”。CompCert 只能对代码做有限的优化，生成的机器码质量较差，因此其远没有 GCC/Clang 流行。

现阶段编译器的研究热点，主要还是集中在优化上。而优化无可避免地会带来更多潜在的 Bug，所以充足的测试显得更为重要。测试的充分性，不仅体现在足够多的测试样例上，还要依赖先进完备的测试方法。下面我们就来介绍一种高效、可靠、通用的编译器测试方法——EMI(Equivalence Modulo Input)

## 1.1　EMI

EMI 是一种可靠且通用的编译器测试方法。它的基本思想是分析在给定输入集 I 上程序 P 执行的控制流，然后在保证控制流不被改变的情况下修改源程

序为 Q ，并重新执行。理论上，改造后的 Q 程序应该与源程序 P 是完全等效的，因此在输入集 I 上执行也理应得到完全相同的结果，即：

$$\forall i \in I, Comp(P)(i) === Comp(Q)(i)$$

显然我们可以得出结论：如果出现不等的情况，那一定是 Compiler 出错了（如果 P 到 Q 的转换是准确无误的话）。

尤其在测试编译器优化错误时更为有效。因为编译器的优化工作原理，往往是基于控制流分析来实施各种激进的优化策略。EMI 通过对源程序的修改，在保证程序等效性的同时，打乱了程序结构，使编译器优化错误更容易暴露出来。

总体而言，EMI 有如下的优点：

1. 它是通用的编译器测试方法，可以广泛用于各种语言的编译器。
2. 它测试效率更高，一个测试样例可以通过修改代码结构，生成多份测试代码。
3. 它可以使用现实世界的代码作为测试样例，一方面减轻了测试的负担，另一方面也使其测试更有现实意义。
4. 它的测试更全面，更容易覆盖边界情况，更容易触发隐藏很深的罕见 Bug，并且这个过程是全自动的。

## 1.2 Rigel

基于 EMI 算法的原理，我实现了一个用于检测编译原理课程实验的 C-- 编译器的测试工具：Rigel

EMI 算法是一套通用的编译器检测模型。具体应用到 C-- 编译器上，我的策略有：

- 将输入放在源码中。虽然我也实现了从外部读取输入的功能，但是这样的话对于每个源文件，都要另外准备一个输入文件，测试效率大大降低，并且也不利于批量自动生成合法的测试样例。可以证明的是，将输入写在源码中和从外部读取输入的测试效果是完全等效的。
- 通过插入 Watch point 函数的方式来统计得到语句的执行覆盖率。虽然说这样无法得到行级覆盖率，但事实上也没有必要非得与 Gcov 一样去统计行覆盖率。因为我们是对程序的 AST 进行操作，以 AST 节点为粒度单位反而比以源代码行号表示更加直观。并且实现起来，无需关注底层细节

（不像一般的代码覆盖率工具是通过插入二进制代码的方式实现），更容易且具有通用性。另一方面，统计覆盖率的过程，也同时可以看作是对编译器生成的程序进行运行时检测的过程。

- 通过插入、删除代码的方式，来生成 EMI 等效程序。相比于原论文中只采用“剪枝”的方法，我额外使用了插入/修改代码的方式，以体现测试的多样性。当然，可以保证，经过插入/修改代码后生成的代码与原程序等价。

Rigel 的详细实现细节请参见第三章。

## 1.3 总体工作

总的来说，本次毕业设计，我主要做了以下一些工作：

1. 随机生成合法的 C-- 程序
2. 统计 C-- 代码覆盖率
3. 对代码修改，生成等价程序
4. 自动化执行并测试 C-- 程序，以检验 C-- 编译器的错误

## 1.4 论文结构

本论文的结构大体如下：

在本章节中，我介绍了编译器测试的重要性和本次毕设项目的大体内容与实现思路。并于本章节的最后介绍了论文的组织结构。

第二章将通过几个具体的例子，来切实感受编译器 Bug 的隐蔽性及其带来的巨大影响。

第三章将详细介绍项目的实现细节，以及我认为比较重要的经验之谈和注意事项，并对代码结构进行梳理。

第四章将具体介绍如何运行本项目以及如何进行修改，以应对不同需求。同时将展示 Rigel 的实际运行效果。

第五、六、七章将介绍相关工作、结论和后续工作等。

# 第二章　示例

编译器错误大体分为两种。第一种就如常规的软件 Bug：输入合法源文件，编译器运行时报错崩溃；或者传入非法源文件时，编译器却不报错。第二种更加隐蔽，即编译器可以正常执行并生成代码，但是得到的代码却不符合源程序语义，即生成的程序有 Bug 。

第一种 Bug 比较容易检测，因为可以及时发现，并且不需要执行编译生成的目标文件。我们可以将其当作一般的软件 Bug 来看待。

而第二种 Bug 就比较难以检测，并且危害更大。毕竟编译器是其他一切软件所依赖的基础设施，如果编译器生成了错误的代码，不但会导致编译生成的代码运行得到各种不确定性结果，产生巨大的破坏作用。更可怕的是此类 Bug 难以察觉与定位，并且会将程序员引入怀疑自己怀疑人生的无尽深渊。

## 2.1　Ken Thompson 编译器后门

说到编译器 Bug，讲一个有意思的故事。

当年 UNIX 的发明者 Ken Thompson 在贝尔实验室里的 UNIX 系统中植入后门，他走到任何一台 UNIX 机器，都能用自己的用户名密码登录。后来其它人去掉了 UNIX 内核的后门，并重写了编译器重新编译操作系统，均不管用。

原来，Ken Thompson 在 C 语言编译器里植入了一段恶意代码，使得它编译出来的程序都留有后门，这样他就可以任意登陆系统。更可怕的是，当这个编译器检测出是在编译自己时，将植入另一段恶意代码，使得编译得到的新编译器具有同样的性质（即在编译出的程序中植入后门）。只要使用过有后门的编译器，即使你的代码无比正确，得到的依然是有后门的程序。并且这个漏洞会一直传播下去，即使你能看见所有源码，或重写一个编译器来编译你的代码，也依然无法发现这个隐藏的漏洞，因为你的编译器依然要使用旧的编译器来编译。除非一开始所有工具都自己实现，不借助任何外部工具，不然永远无法绝对保证正确性，显然这是不可能的。

事后，Ken 在论文 *Reflections on Trusting Trust* [2] 中详细阐述了一个值得信赖的编译器是多么重要。

## 2.2 XcodeGhost 事件

上述的故事并未单纯止步于一个恶作剧。这个方法后来被人利用，爆发了大名鼎鼎的“XcodeGhost 事件”。事件的大致经过，是开发者下载了包含恶意代码的编译器，导致编译生成的所有程序都被植入后门，产生隐私泄露、广告点击等问题。由于是开发端的程序污染，普通用户毫无防备能力，因为即使是从官方渠道下载安装的应用也会有此风险。事件的影响颇为广泛，对用户隐私安全造成极大危害，许多知名产品都被波及，如微信、网易云音乐、滴滴、高德等装机量达数亿的软件。

由此可见编译器的重要性，它不仅仅会影响使用编译器的软件开发者，普通用户也会被间接地波及。所以对编译器进行全面、详尽的测试便显得尤为迫切。这不是一个只面向少数人的需求，而是一个可能影响数亿用户以及全球互联网基础设施的大问题。

## 2.3 几个常见编译器 Bug 演示

### 2.3.1 第一类，编译器自身运行错误

这类例子比较少见。因为如前文所说，此类 Bug 更容易被发现，因此编译器作者可能早早便发现并修复 Bug ，到用户层面时已经很难再撞见了。即便如此，人们还是有幸发现了 VC++ 编译器的一个内存泄露 Bug。在编译代码片段2.1时，编译器会无限申请内存，并最终卡死。

```
1 void test()
2 {
3     __asm { add eax
4     __asm { add eax
5 }
```

**Listing 2.1**: testasm

编译器的输出为代码片段2.2。

```
error C2414: illegal number of operands
error C2414: illegal number of operands
error C2400: inline assembler syntax
```

```
error in ‘opcode’; found ‘end of file’ fatal error C1060:
  compiler is out of heap space
```

**Listing 2.2**: vs error

同时，内存被耗光，如图2.1，2.2，2.3所示，编译器申请过当内存，并最终崩溃。

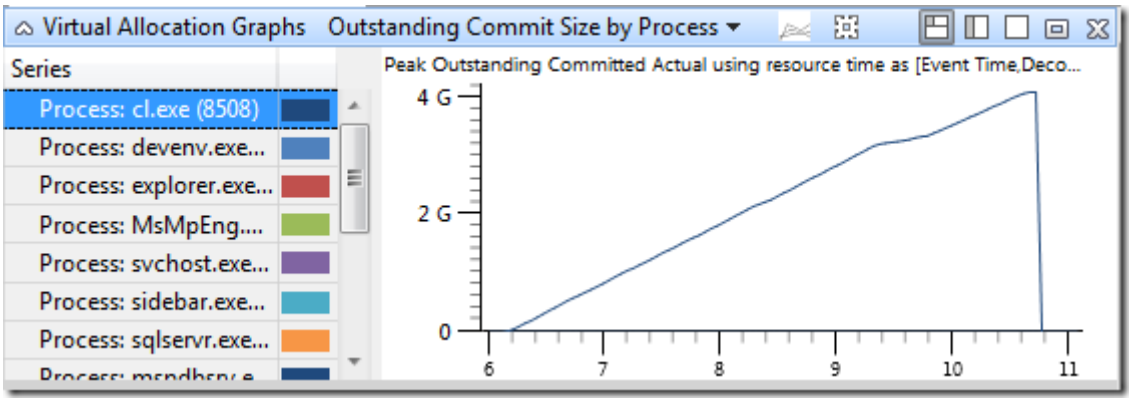


**Figure 2.1**: 内存使用情况

| | Count | Reserved Actual | Committed |
|---|---|---|---|
| C1XX.DLL!InvokeCompilerPassW | 128562 | 4253642752 | 4084768768 |
| \|- C1XX.DLL!Trap_CallMain | 128557 | 4253507584 | 4084752384 |
| \| \|- C1XX.DLL!CallMain | 128528 | 4231340032 | 4084269056 |
| \| \| \|- C1XX.DLL!Trap_main_compile | 128527 | 4231340032 | 4084264960 |
| \| \| \| \|- C1XX.DLL!main_compile | 128509 | 4159709184 | 4083564544 |
| \| \| \| \| \|- C1XX.DLL!CallPrimaryParser | 128506 | 4159709184 | 4083400704 |
| \| \| \| \| \| \|- C1XX.DLL!inline_assembler | 128476 | 4159700992 | 4082397184 |
| \| \| \| \| \| \| C1XX.DLL!asm_yylex | 128476 | 4159700992 | 4082397184 |
| \| \| \| \| \| \| C1XX.DLL!AllocAsmToken | 128476 | 4159700992 | 4082397184 |
| \| \| \| \| \| \| C1XX.DLL!VirtualHeap::AllocEven | 128476 | 4159700992 | 4082397184 |
| \| \| \| \| \| \| \|- C1XX.DLL!VirtualHeap::HeapExtend | 128467 | 4081057792 | 4082073600 |
| \| \| \| \| \| \| \| \|- KernelBase.dll!VirtualAlloc | 120683 | 0 | 3954540544 |
| \| \| \| \| \| \| \| \|- C1XX.DLL!VirtualHeap::Create | 7784 | 4081057792 | 127533056 |
| \| \| \| \| \| \| \| \| \|- C1XX.DLL!VirtualHeap::AllocEven | 3892 | 0 | 127533056 |
| \| \| \| \| \| \| \| \| \|- KernelBase.dll!VirtualAlloc | 3892 | 4081057792 | 0 |

**Figure 2.2**: 函数调用栈

由于微软的 VC++ 编译器并未开源，因此我们无法查阅源码以追踪是何原因触发了这个 Bug。只能期待微软的编译器团队来打补丁修复了（经测试，在 Visual Studio 2017 中，此问题已被修复）。

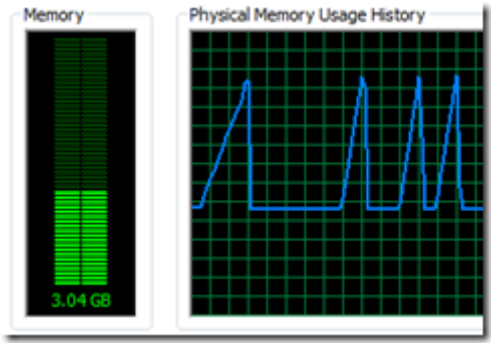


**Figure 2.3**: 资源管理器

### 2.3.2 第二类，编译生成错误代码

编译器生成错误的代码，这是编译器 Bug 中占比最高的类型了，数量巨大，诱因也五花八门。虽然很多，但是事实上这些 Bug 都很难被发现，造成的潜在破坏也更为巨大。编译生成的错误程序在绝大多数情况下都可以正常运行得到正确结果，但是在一些极端案例下便会显露其狰狞面目，让程序开发者怀疑自己、思考人生。

下面，我将通过几个例子，来展现编译器会如何编译、修改我们的代码，最终导致不确定性或错误的结果。

#### 2.3.2.1 LLVM 的 Bug

```
struct tiny {
    char c;
    char d;
    char e;
};
void foo(struct tiny x)
{
    if (x.c != 1)
        abort();
    if (x.e != 1)
        abort();
}
int main()
{
```

```
    struct tiny s;
    s.c = 1;
    s.d = 1;
    s.e = 1;
    foo(s);
    return 0;
}
```

**Listing 2.3**: Reduced version of the code for bug reporting.

如代码片段2.3所示，如果正确编译的话，结构体中的成员都被初始化为 1，因此条件判断不会成立，即程序不会运行至 abort。

但是 LLVM 优化过程中，首先使用了全域数值编号的方式，将结构体 s 当成一个 32bit 的整数来加载。然后，LLVM 使用标量替换的方法，将结构体 s 的各个成员优化为栈上的局部变量。然而这两种优化方式会产生冲突，导致不确定性行为，继而导致结构体 s 并没有被正确初始化，引发程序运行时 abort。不过这个 Bug 并不一定会触发，只有在 LLVM 选择将 foo 函数内联时才会引发这个 Bug，而是否 inline，这又是由编译器选择的不确定性行为，导致极其难以复现和调试。

#### 2.3.2.2 GCC 的 Bug

不仅 LLVM 会有 Bug，GCC 也不能幸免。这是一个 GCC 的例子，如代码片段2.4所示。

```
int a, b, c, d, e;
int main()
{
    for (b = 4; b > -30; b--)
        for (; c;)
            for (;;) {
                e = a > 2147483647 - b;
                if (d)
                    break;
            }
    return 0;
}
```

**Listing 2.4**: GCC miscompiles this program to an infinite loop.

这个例子中，因为 C 语言规定，未初始化的全局变量值为 0（因为未初始化的变量是弱符号，在.bss 段中，动态加载器保证其值为 0），因此第二层循环不会执行，程序将很快执行完毕退出。

但是 GCC4.8 编译产生了错误的代码，使得程序死循环。这是由于 GCC 采取了*局部冗余变量消除*的技术，尝试将循环不变量外提以优化性能，所以将 2147483647 - b 这个表达式提取到了最外层循环体中。但如果这么做的话，当 b==1 时，会造成整数溢出，而整数溢出是一个 undefined 的行为，所以编译器可以“为所欲为”地按照它自己的理解去生成代码，这其中就可能生成导致死循环的代码。当然，编译器这么做的同时也会给出 Warning，提示此处有 undefined 行为。我们知道，其实代码并不会执行到那里，整数也不会溢出，所以此代码中并不存在未定义行为，这是一个错误的 Warning。

更为有意思的是，这段代码只在 GCC 的 -O2 优化选项时才会触发此 Bug，在更为激进的 -O3 选项下反而不会触发。

编译器暴露这个 Bug 的唯一马脚，或许就是给出了一条错误的 Warning，提示这里有一个 undefined 行为吧。可惜估计大多数程序员看到这条警告，都只会略微检查一下，在确定代码没有问题后便不再关心了。

有一句话叫：“Warning 是滋生 Bug 的温床。”看来这句话真是无比正确——Warning 不但孕育着代码的 Bug，也可能意味着编译器的 Bug！

## 2.4 本章小结

本章介绍了两类编译器 Bug：编译器自身运行时的错误以及编译器生成的程序运行错误。

并列举了多个编译器不同类型 Bug 的例子，细致分析了这些 Bug 的触发条件、症状以及触发原因。再次阐述了编译器 Bug 的隐蔽性以及对编译器进行详尽测试的重要性。

# 第三章　实现

在这一章节，我将详细介绍 EMI 算法，以及 Rigel 项目的整体设计与实现细节。同时也将介绍对现有 Csmith 项目和 IR Simulator 项目的修改。并将重点介绍实现过程中的难点与创新之处。

## 3.1　EMI 算法

### 3.1.1　EMI 等价

正如在第一章所说，EMI 是一种对各语言编译器通用的测试方法。所以在此小节我将不依赖于具体程序语言来详细介绍 EMI 的概念和方法。

EMI 的概念是，在给定输入集上，程序/模块完全等价，即在此输入集上有完全相同的执行结果和副作用。所以我们先定义两个程序 EMI 等价的概念：

给定两个程序 P, Q，$\forall i \in I\, P(i) === Q(i)$ ，即在此输入集上有完全相同的执行效果（同时包括结果和副作用），则称 Q 为 P 的 EMI 变种。

### 3.1.2　产生 EMI 变种

对于一个程序 P，产生其 EMI 变种有多种方式，但最终我们需要保证的是，生成的新程序 Q 在输入集上有完全相同的执行效果。

因此，生成 EMI 变种面临两大难题：

- 采用何种方式来生成与源程序 EMI 等价的新程序？
- 如何验证生成的新程序与源程序 EMI 等价？

对于第一个问题，我们可以想到，将那些程序运行过程中不会执行到的“死代码”删除或做一些合法的修改，是不会对原有的代码逻辑造成影响的。此外，在某些位置插入一些完全没有副作用的代码，也不会对源程序的逻辑造成影响。依此方法便可以生成与源程序 EMI 等价的新程序。

对于第二个问题，其实也比较容易解决。虽然我们知道肯定无法穷举所有输入来证明两个程序完全等价，但是事实上我们也没有必要这么做，我们只要在一定的输入数据集 I 上进行测试即可。如果对于 I 中的任意输入 i，都可以验证两个程序得到相同结果，那么我们就可以说两个程序在输入集 I 上 EMI 等价（EMI 原始定义其实就是在一定输入集合上的等价）。并且我们只需要保证“拒真率”为 0，“认假率”不为 0 也没有太大的影响。毕竟所有的测试都不能保证

一定能将所有 Bug 都全部查出来，那就变成了代码正确性证明了。如果想要查出更多 Bug，只要多随机运行几次测试即可。从这个角度来看，EMI 算法也算是一种随机算法了。

### 3.1.3 算法详解

非常感谢 Prof. Zhendong Su 提出的 EMI 算法模型，下面我想冒昧地引用一下 *Compiler Validation via Equivalence Modulo Inputs* [1] 论文中的算法图示，来详细介绍 EMI 编译器检测算法的详细实现步骤。

#### 3.1.3.1 Main process for compiler validation

**Algorithm 3.1** Main process for compiler validation [1]

```
1  procedure Validate (Compiler Comp, TestProgram P, InputSet I):
2      begin
           /* Step 1: Extract coverage and output */
3          P_exe := Comp.Compile(P,"-O0")      /* without opt. */
4          C := ⋃_{i∈I} C_i, where C_i := Coverage(P_exe.Execute(i))
5          IO := {⟨i, P_exe.Execute(i)⟩ | i ∈ I}
           /* Step 2: Generate variants and verify */
6          for 1..MAX_ITER do
7              P' := GenVariant(P,C)
               /* Validate Comp's configurations */
8              foreach σ ∈ Comp.Configurations() do
9                  P'_exe := Comp.Compile(P', σ)
10                 foreach ⟨i,o⟩ ∈ IO do
11                     if P'_exe.Execute(i) ≠ o then
                           /* Found a miscompilation */
12                         ReportBug (Comp, σ, P, P', i)
```

我们先来看算法3.1，EMI 编译器验证算法的主体执行流程：

先做准备工作，将待测程序使用 -O0 参数编译，即不开启任何优化，这样可以保证最大程度的正确性。然后在给定输入集 I 上执行，得到在这个输入集上执行后的代码覆盖率以及用作参照的输出结果。事实上，编译得到的这个程序是否正确，以及此程序运行得到的是否是正确结果并不重要，因为只要对比发现不同结果，一定是编译器某个地方出错了。让编译器“自证”错误，这也是 EMI 算法的亮点之一。

随后的工作就是多次迭代，生成 EMI 等价变种来进行测试验证。先生成 EMI 等价变种程序，然后在每一种编译器参数下（譬如 -O1，-O2，-O3），编译

生成的等价源码，得到可执行文件。然后在输入集上执行新生成的程序，输出结果应与之前的参照输出完全相同。如果和之前的输出结果不等，那么显然就可以汇报一个编译器 Bug 了。

#### 3.1.3.2 Generate EMI variant

上述流程中被一带而过但却是最重要的一步，就是生成 EMI 等价变种。算法3.2中详细介绍了生成 EMI 等价变种程序的流程。

**Algorithm 3.2** Generate EMI variant [1]

```
1  function GenVariant (TestProgram P, Coverage C): Variant P′:
2      begin
3          P′ := P
4          foreach s ∈ P′.Statements() do
5              PruneVisit (P′, s, C)
6          return P′
7  procedure PruneVisit (TestProgram P′, Statement s, Coverage C):
8      begin
           /* Delete this statement when applicable */
9          if s ∉ C and FlipCoin(s) then
10             P′.Delete (s)
11             return
           /* Otherwise, traverse s's children */
12         foreach s′ ∈ s.ChildStatements() do
13             PruneVisit (P′, s′, C)
```

如算法3.2所示，先生成一份源程序副本，然后遍历每一条语句，进行随机删除操作。具体根据什么条件删除，以及如何删除，在 PruneVisit 中体现：

如果一个语句没有被执行到（即之前运行时覆盖率为 0），那么就以一定的概率随机删除它并返回。否则，就遍历该语句的所有子语句，递归进行这个随机删除操作。

我在前面的章节也有提及，EMI 并不一定只采用随机删除死代码这一个方法，还可以进行随机修改、插入代码等等操作，只要保证修改后的代码和原程序等价即可。事实上，在 Rigel 项目中也是这么做的，尽可能让修改多样化，挖掘 EMI 方法的潜力。

#### 3.1.3.3 综合图示

更精炼一点来介绍，用图3.1和图3.2便可以概括。

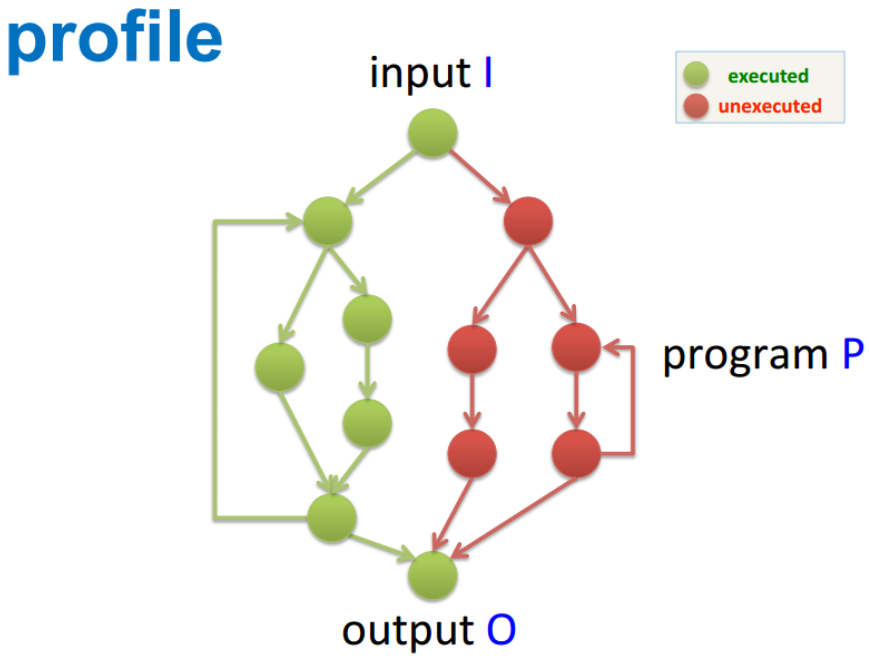


**Figure 3.1**: 统计覆盖率 [1]

图中显示的是程序执行后的控制流，绿色标注的是执行覆盖到的部分，红色标注的是死代码。如图3.1所示，先在输入集 I 上执行代码，得到覆盖率信息以及输出结果 O。图3.2表达的是对死代码进行随机删改，然后重新在输入集 I 上执行，应该与原程序得到完全相同的结果，即两者等价。

## 3.2 Rigel ——C-- 编译器检测工具

### 3.2.1 项目综述

> Rigel（参宿七）是 Orion（猎户星座）中最明亮的一颗恒星。从 1943 年以来，它的光谱就被当成其它恒星光谱分类的校准光谱之一。参宿七是一颗重要的航海星，因为它很明亮，又可以轻松地定出赤道的位置，这也意味着在全球各地的海洋都看得见它。

Rigel 是一款基于 EMI 算法，依赖于 LLVM/Csmith 实现的自动化 C-- 编译器检测工具。可用于南京大学《编译原理》课程实验项目 C-- 编译器的测试与验证。它具有自动生成 C-- 源代码、词法/语法/语义检查、自动化随机测试等功能。

在这一小节，我将详细介绍我实现的基于 EMI 方法的 C-- 编译器检测工具——Rigel 的实现细节与原理。

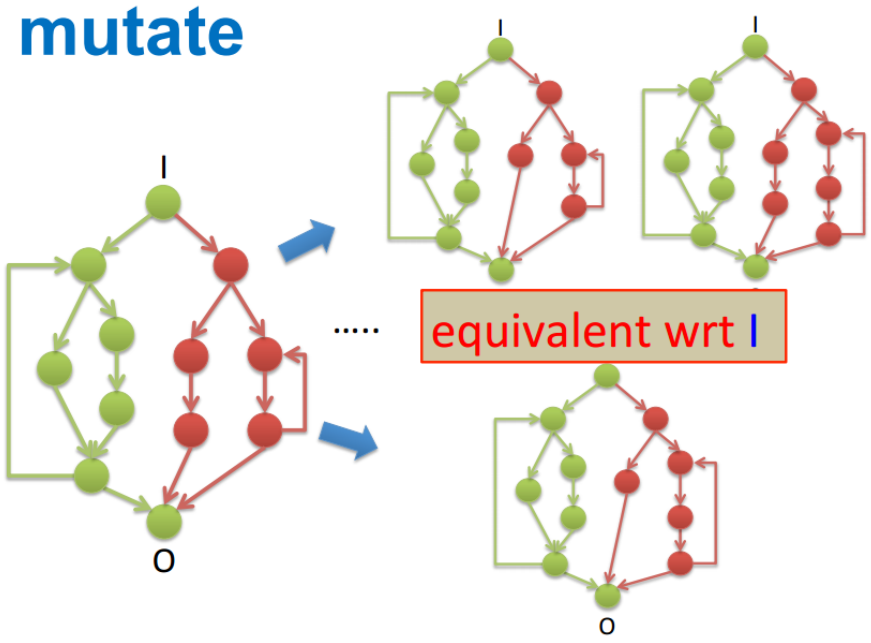


**Figure 3.2**: 产生 EMI 变种 [1]

### 3.2.2 代码框架

Talk is cheap, 用代码说话，整个项目的工程目录如列表3.1所示。

```
$ tree /a /f
Rigel:
|   .gitignore
|   .gitmodules
|   clang_helper.c
|   clang_helper.h
|   CMakeLists.txt
|   common.h
|   cmmsmith_help.txt
|   emi.c
|   emi.h
|   emi.png
|   Rigel
|   gen_cmm.c
|   gen_cmm.h
|   irsim-cli.py
```

```
|   inter.ir
|   main.c
|   Makefile
|   outline.md
|   ReadMe.md
|   Dockerfile
|   .travis.yml
|
+---csmith
|   +---...
|   +---...
|   \---...
|
+---generated_cmm
|       0.c
|       1.c
|       2.c
|       xxx.c
|
\---tmp_dir
        inserted_watch_point.c
        temp_file_with_rw.c
        random_gen_file.c
```

**Listing 3.1**: 项目文件

下面我将有选择性地详细介绍各个文件的意义与功能。

### csmith/ & cmmsmith_help.txt

csmith/ 是 Csmith 项目的目录，我将 Csmith 项目 fork 后作为 git submodule 引入进来，作为 Rigel 项目的一个子项目。此项目中代码文件较多，因此无法详细列举说明。

Csmith 项目原本作用是随机生成合法的 C++ 程序，因此需要对其进行改造，使其能够生成合法的 C-- 程序，这也是本次毕业设计一大难点所在。虽然改造后的程序生成的已经是 C-- 语言代码了（因此或许可以称之为 Cmmsmith），

但是出于对原作者的尊重，该目录依然保留了 Csmith 的名称。

cmmsmith_help.txt 是 Csmith(Cmmsmith) 程序的帮助文件，里面记录了一些配置参数。正常情况下你无需了解这个文件的内容，Rigel 会自动调用 Csmith(Cmmsmith) 去生成合法的 C-- 程序。但如果你想要独立运行 Csmith(Cmmsmith) 项目，不妨看一下这个文件以了解其参数与用法。

**generated_cmm/**

这个文件夹用于输出 Csmith(Cmmsmith) 所生成的合法的 C-- 文法代码。Rigel 运行时将读取该文件夹下的所有文件作为测试用例对编译器进行测试。这个文件夹可以为空，由 Rigel 调用 Csmith(Cmmsmith) 来生成代码，也可以自己手动编写测试样例放进去。不过要注意的是，文件名需以.c 为后缀。

**tmp_dir/**

顾名思义，这是一个临时目录，用于放置项目运行时生成的一些中间文件。你无需关心其中的内容，Rigel 会自动清理这些临时文件。不过当目标编译器未能通过测试，或 Rigel 项目崩溃（这个概率很低）时，这里的文件便对 Debug 很有帮助。你可以在这里找到具体是怎样的代码使得目标编译器未能通过测试，从而复现 Bug，以助于修正编译器。

**CMakeLists.txt & Makefile**

这是项目的构建文件，如果只需要在 Linux 环境下运行，只需要使用 Makefile 就足够了。不过如果有跨平台需求，那么可以使用 CMake 来进行构建。

**irsim-cli.py & inter.ir**

这是经过改造的 IR Simulator 程序，负责运行生成的中间代码并验证输出是否相同，这也是本次项目一个关键步骤。我对原有的小程序进行了改造，去掉了图形界面和交互式接口，只保留了命令行调用的方式。Rigel 将调用这个 Python 小程序，用以执行待测编译器生成的 IR 代码，并收集输出结果来验证原程序与新程序是否 EMI 等价，从而检验编译器的正确性。

inter.ir 文件是编译器生成的中间代码文件。按照《编译原理》课程实验的要求，实验手册规定生成的中间代码文件置于此，因此我们便约定中间代码生成至这个地方。不过你并不需要关心这个文件的内容，因为它也属于临时文件。当目标编译器未能通过测试时，你或许可以查看一下这个文件，以定位测试失败的位置。

**common.h**

这是项目的公共头文件，用于包含标准库头文件、定义一些项目中使用的常量、以及定义用于调试的宏。

此外，这里还定义了项目的一些功能开关。Rigel 实现了许多功能，但是默认没有全部开启。如果有特殊需求，可以修改这里的一些宏开关，打开/关闭一些功能并重新编译，以实现定制化的功能。**注意，当你要修改代码时，请确保清醒地知道自己在做什么。**

**clang_helper.{c,h}**

这是与 Clang 库（不是 Clang 执行程序）进行交互的相关代码。本项目中借助 Clang 库来进行词法、语法分析，以生成 AST (抽象语法树)。然后遍历 AST，以达到修改 C-- 代码并生成其 EMI 变种的目的。在统计代码覆盖率时，也参考了 AST 上的节点类型信息，用以决定插入 watch point 函数的位置。可以说，Clang 库是本项目中最重要的外部依赖。

**gen_cmm.{c,h}**

这是负责生成合法 C-- 程序的代码，代码主要功能是与 Csmith(Cmmsmith) 进行交互，传入恰当的运行参数，以向指定目录生成给定数量的合法 C-- 源文件。

**emi.{c,h}**

这是项目的主要逻辑代码，完整实现了基于 EMI 方法的编译器测试功能。具体来讲，有如下主要功能：

- 对 C-- 代码进行词法/语法分析和语义检测，以判断编译器前端部分的正确性。
- 统计代码执行覆盖率。
- 执行生成的 EMI 等价变种程序，并比对输出结果。
- 报告编译器 Bug，并定位错误位置。

当然，还提供了一个简洁的对外接口，以方便从外部调用。

**main.c**

这是 Rigel 的入口函数，同时也可以认为是本项目的测试入口。在这里定义了待测编译器的路径、生成的中间文件保存的位置以及随机数种子。然后依次调用自动生成 C-- 程序的函数和对编译器进行 EMI 检测的函数，并在最后统一汇报测试结果。当然，如果将本项目编译为库用以在外部调用的话，则无需关心 main 函数的内容，它只是最上层的一个包装。

**Rigel**

这是最终编译生成的 Rigel 的可执行文件，直接运行即可。程序完成的工作如刚才 main 函数所述：自动调用 Csmith(Cmmsmith) 生成测试代码，并调用相关函数基于 EMI 方法对编译器进行随机迭代测试。如果执行过程中待测编译器有任何 Bug ，你可以选择程序立即退出或是等待全部测试完毕后统一汇报错误。

**emi.png & outline.md**

这是项目的说明文档以及一些必要的配图。作为文档的一部分，因此也包含在项目之中。其中使用伪代码（代码片段3.2）描述了整个项目的执行逻辑，虽然和原始的 EMI 方法大同小异，但是依然有很多改动和创新点。

```
cc=COMPILER_TO_CHECK
for each test.c:
    CHECK_SYNTAX_CHECK:
    check_src_validation_by_clang(add read()/write() delcation
      .)
        and compile_by_cc (they should be same)

    GET_CODE_COV:
    insert _my_func in test.c(use clang to find where to insert
      )
    cc compile test_inserted.c > a.ir
    for all inputs:
        a.ir executed by irsim-cli.py
        parse stdout and get coverage.
    get code coverage <input, output>
```

```
ITERATE:
random delete(or change) unexecuted code, get test_new.c
for all optimization level:
    cc compile test_new.c > a_new.ir
    for all inputs:
        a_new.ir executed by irsim-cli.py
        assert(a_new.ir(input)==output[input])
```

**Listing 3.2**: 执行流程

**ReadMe.md**

项目的简短介绍文档。包含了项目的简单介绍以及持续集成的构建状态信息。

**Dockerfile**

这是项目的 Dockerfile。本项目采用容器化的管理方式以方便开发、测试和部署。为了方便调用 Clang 库，项目基于 LLVM-3.5 镜像来构建，镜像中同时安装了 Python 3.4 等工具。在确保了部署环境的隔离性的同时，也使本项目具有跨平台移植性和快速迭代部署能力。对代码进行任何修改后，使用 Docker 重新构建，即可得到最新的镜像。运行该镜像，即相当于运行本项目，一切都很方便简洁。

**.travis.yml**

这是 Travis CI 的配置文件。我们同时使用 Docker 和 Travis CI 来做持续集成，保证项目代码的鲁棒性，确保每次 commit/push 都是健康可控的，这也是对代码质量的极大保障。

### 3.2.3 函数功能说明

介绍完项目框架后，再来详细介绍项目的实现细节，本节将从具体的函数出发，深入剖析 EMI 的算法与实现。

**void generate_tests(const char *dst_dir, int cmm_file_num);**

如函数名所描述，此函数负责向指定目录随机生成指定数量的 C-- 测试程序。如果目标目录不存在则会自动新建文件夹，生成的文件名以数字命名，形

如：1.c/2.c/3...。

**void generate_cmm(const char *outfile, unsigned seed);**

这个函数负责根据指定的随机数种子，生成 C-- 程序源代码。显然，generate_tests 是通过调用它来具体生成 C-- 程序源文件的。这是相对独立的函数，只需要和 Csmith(Cmmsmith) 进行交互。也就意味着，你可以在外部独立调用此函数生成 C-- 程序用作其他用途。并且此函数是确定性的，只要传入相同的种子，便一定会得到完全相同的结果，方便调试。

**int sem_check_by_clang(const char *test_file);**

使用 Clang 对 C-- 源程序进行辅助语义检查。如果源程序无任何词法/语法/语义错误，则返回 0，否则返回非 0 值。该函数会自动向源程序添加 read/write 函数的声明，以使其满足 C 语法要求。

**void traverse_cmm_ast(const char *test_file);**

如函数名所述，该函数将遍历 C-- 源程序的抽象语法树。在遍历每个节点的过程中，执行相关函数（譬如判断代码是否可以删除、是否被覆盖等）。同样，此函数借助 Clang 生成的 AST 来实现，不用自己重新实现一遍 parser。

**void test_compiler(const char *cc_path, const char *test_case_dir);**

这个函数的功能是使用指定测试样例（通过 generate_tests 函数生成）来测试目标编译器。完全基于 EMI 方法进行测试，其中调用了如下几个函数：

**static void test_cmm_file(const char *cc_path, const char *test_file)**

根据一个给定的 C-- 源文件，基于 EMI 方法来测试编译器。

**static void insert_watch_point(const char *in_file, const char *out_file)/static void count_coverage(const char *cc_path, const char *test_file, const char *input_file, char *origin_output);**

插入监视函数，并执行以得到代码执行覆盖率。

**static void random_test(const char *cc_path, const char *cmm_flag, const char *test_file, const char *origin_output, const char *input_file);**

根据代码执行覆盖率，对代码进行随机删改并测试，验证与原代码编译执行后的输出是否相同。这可以算作 EMI 的主体算法的实现。几个参数含义如下：

- const char *cc_path

  编译器程序路径。

- const char *cmm_flag

  编译器编译参数。

- const char *test_file

  待测试的 C-- 源文件。

- const char *origin_output

  没有经过任何修改、按照默认编译参数编译执行得到的输出结果。用于比较判断正确性。

- const char *input_file

  显然，自动测试过程中，是没有办法交互的。待测试的代码如需从 stdin 读取输入，则需要将输入预先存放在一个文件中，并将文件名传入。Rigel 会自动读取此输入，并重定向至 IR Simulator 的 stdin 执行。如果无需输入文件，则传入 NULL 即可。

  考虑到绝大多数情况下，是无需读入输入文件的，因此在编译时，可以定义 ENABLE_INPUT 宏，决定是否允许传入输入文件。

**static int cmm_compile(const char *cc_path, const char *cmm_flag, const char *test_file);**

这个函数负责根据给定的编译器执行程序路径、编译参数以及待编译文件路径，编译生成中间代码。目前编译参数只支持 “-O0”, “-O1”, “-O2”, “-O3”, 如果编译器不支持编译参数，则可以传入 NULL 指针。因为历史原因，为保证代码和 IR Simulator 之间的兼容性，编译生成的中间代码默认保存在当前目录下的 inter.ir 文件中，同时约定 Rigel 去寻找这个文件并执行。

**static int sem_check(const char *cc_path, const char *cmm_flag, const char *test_file);**

Rigel 借助了 Clang 库来进行辅助词法/语法/语义检查。考虑到 C-- 文法为 C 文法的子集（这一点，在改造后的 Csmith(Cmmsmith) 中得以保证），因此可以期待待测编译器应该与 Clang 得到相同的检查结果。如果拒真或认假，那么就可以认为待测编译器有错误。

其实这一步和 EMI 算法没有关系，即使不借助 Clang 进行辅助语法检查，依然可以单纯使用 EMI 方法来检验编译器错误。但是为了更好的检测效果，引入 Clang 进行辅助检查是一件有益无害的事情。

**static void exec_ir_code(const char *inserted_ir, char *output_buf, const char *input_file);**

这个函数调用 IR Simulator 来执行生成的中间代码。执行过程中如果死循环则会报错退出。

**static void exec_ir_for_cov(const char *inserted_ir, char *output_buf, const char *input_file);**

这个函数同样是调用 IR Simulator 执行生成的中间代码。不过这个函数被调用时，执行中间代码的目的是得到源代码的执行覆盖率，因此对于每个待测源文件，这个函数只会被调用一次。

本着 DRY(Don’t Repeat Yourself) 原则，此函数和 exec_ir_code 都依赖 exec_ir_driver 函数来进行代理调用。

### 3.2.4 其他工作

#### 3.2.4.1 对 Csmith 的修改

原版的 Csmith 功能非常强大，它支持庞大而臃肿的 C++ 语法，以及很多 C 语言中几乎很少用到的特性。在本项目中，这些强大的特性反倒是累赘，因此必须对 Csmith 做个性化的定制以生成合法的 C-- 文法程序，满足本项目需求。

首先，Csmith 与本项目有如下一些不兼容之处：

**数组初始化** C-- 文法里，是不支持数组直接初始化赋值的，而 C 语言中可以，并且经常这么做。同时 Csmith 里，没有配置选项可以关闭这个特性。因此必须手动修改，将原本数组初始化的代码，改写成定义数组、再循环赋值的形式。

**for 循环** 比较有趣的是，Csmith 生成的所有循环，都是用 for 来实现的，没有 while 语句。与此同时，C-- 却只支持 while 循环，因此需要将所有的 for 循环改写为 while 循环的形式。不过好在方法也比较简单：将 for 的初始化语句放置在 while 语句前，同时将循环变量自增语句放置在循环体最后，这样就可以近似地将 for 语句改写为 while 语句。之所以说近似，是因为两者之间并不完全等价，譬如：

- for 语句引入独立的作用域，在 for 语句中定义的循环变量作用域只在 for 语句中有效，而 while 语句则必须将循环变量定义在外部作用域中。
- for 语句执行过程中，如果遇到 continue 语句，最后的循环自增语句依然执行，而通过此方法模拟的 while 语句则直接跳转至下一轮循环，没有机会执行循环变量自增语句。

不过好在 C-- 中所有变量都是全局作用域，并且不支持 continue 关键词，因此不会出现歧义。可以安全地使用这个方法将 for 语句改写为 while 语句。

**自增/自减/复合赋值运算符** 在 C 语言中，形如 ++i/i++/i+=1 这样的自增/自减/复合赋值运算符是非常常见的。因此，Csmith 中同样没有给出配置参数，而是默认就会生成带自增/自减/复合赋值运算符的语句。然而 C-- 文法中却不支持自增/自减/复合赋值运算符，所以需要将此类运算符转换为普通的二元运算和赋值语句。譬如将 i+=1 替换为 i=i+1。将 ++i/i++ 替换为 i=i+1，但这么做会有一点问题：当自增运算不是语句而是作为表达式时，i++ 与 i=i+1 等价，但和 ++i 不等价。不过好在我们保证不会生成这种严重不规范的代码，因此这也不是一个问题。

**volatile/void/union…等关键词** 由于 C-- 只支持有限的 C 语言文法，而标准 C 语言 (C89) 中，有 24 个关键词，而 C-- 中只有 struct, return, if, else, while, int, float 这七个关键词。所以需要限制 Csmith(Cmmsmith) 只生成含有这七个关键词的代码。

**include/宏/声明** C-- 文法非常简单，不支持如 include、宏定义等预处理命令。同样也不支持函数和变量的声明，所以也要将这部分内容删去。

**read/write 函数** C-- 内建了 read/write 函数供输入输出使用，并且无需声明。同时，write 函数也是获取 C-- 代码执行结果的唯二方法之一（另一个是通过 main 函数的返回值）。因此生成的代码中，要调用 write 函数将结果打印，这样我们才能了解到代码的执行情况。

**只有 int 类型** 由于在实验三中，我们限制了程序中只会有十进制的 int 类型字面量，并且输入输出都为 int 类型。所以我们同样需要在 Csmith(Cmmsmith) 中限制生成的代码只有十进制的 int 类型字面量，并且所有变量以及输入输出都应为 int 类型。

#### 3.2.4.2 对 IR Simulator 的修改

由于 EMI 算法的要求，在两个地方需要执行编译生成的代码：

- 将待测源代码编译执行，得到代码执行覆盖率。
- 编译执行生成的 EMI 变种，检验输出是否与原输出相同。

然而，原有的 IR Simulator 小程序并不能直接调用，必须加以改造，以适应项目需要。主要做了如下改动：

- 去掉 GUI，只保留核心的虚拟机部分。
- 升级 Python2 为 Python3。
- 改造输入输出部分，以方便统计执行覆盖率及比较输出是否相等。

## 3.3 本章小结

本章先介绍了 EMI 的定义，分析其可行性，并详细介绍了算法的执行步骤。随后重点介绍基于 EMI 算法原理实现的 C-- 编译器检测工具——Rigel，对项目设计架构进行梳理，并对重点逻辑代码进行详细阐述，同时介绍了对 Rigel 所依赖的外部项目做出的一些修改。集中展现了本次毕业设计的重点、难点与亮点工作。

# 第四章　运行以及结果

这一章节将集中演示 Rigel 的运行方式，并给出几个例子来演示代码的输出结果。同时也将介绍一下如何修改代码以适应不同的需求。

## 4.1　构建

首先，来讲一下项目的构建方式。因为项目用到了 Clang 等工具，手动去配置环境显然比较麻烦，因此我们采用了 Docker 来构建和部署。

项目代码中已经包含了 Dockerfile，因此，只需要在 shell 中按照代码清单4.1执行，即可构建项目（宿主机需安装 docker，并可以访问互联网）。

```
git clone --recursive https://github.com/lxiange/Rigel.git
cd Rigel
docker build -t rigel/v0.1 .
docker images
```

**Listing 4.1**: 构建指令

依此步骤执行，屏幕上会打印许多编译日志信息，见图4.1。

如果没有出错的话，最后一条命令会在屏幕上显示构建完成的镜像，如图4.2。

## 4.2　运行

我们已经得到了包含项目源代码、执行文件及其依赖环境的镜像，下一步就是运行此镜像。

首先，在当前目录下新建一个 compilers 目录，用以放置待测编译器的二进制执行文件。由于 docker 类似于虚拟机，不能直接通过命令行读取宿主机文件，因此必须通过此种挂载文件夹的方式来传递文件。准备好后，在 shell 中输入代码清单4.2执行。

```
docker run --rm -v `pwd`/compilers:/rigel/compilers rigel:v0.1
```

**Listing 4.2**: 自动运行指令

Rigel 会自动寻找放置在 compilers 文件夹下的待测 C-- 编译器。运行此命令，则会对该编译器进行自动化随机测试。

当然，你也可以选择手动进入容器的方式来运行代码，这样你可以获得更多信息，或对代码进行修改并重新编译。命令参见代码清单4.3。

```
docker run -it --rm -v `pwd`/compilers:/rigel/compilers rigel:
  v0.1 bash
```

**Listing 4.3**: 手动运行指令

这样，你便进入了一个类似虚拟机的运行环境，此环境下已经准备好了代码运行的所有依赖项目。项目相关文件都在 /Rigel 文件夹下，除此之外，此环境与宿主机是完全隔离的。

## 4.3 输出

可以正常运行后，下面来解析一下程序的输出信息。

项目编译时，可以设置 ENABLE_LOG 宏，选择是否输出详细日志。默认我们尽可能输出更详细的信息，如果只关心测试结果的对错，可以将 ENABLE_LOG 宏设置为 0。

代码中采用了彩色输出，调试输出为黄色，错误报告为红色，测试全部通过时则为绿色输出。如图4.3所示。

其中，调试输出的格式为：

**{所在源文件}@{所在行号}/{所在函数}: {调试输出内容}**

错误输出会提示 Bug 的类型，目前支持：

- COMPILE_TIME_OUT: 编译超时
- CODE_DEAD_LOOP: 代码死循环
- NO_IR_CODE_FOUND: 没有生成 IR 代码
- OUTPUT_MISMATCH: EMI 变体程序输出与原输出不同
- FALSE_ACCEPT: 接受了非法的 C-- 源文件
- TRUE_REJECT: 拒绝了合法的 C-- 源文件

## 4.4 示例

对于开发测试而言，测试一个完全正确或是错得离谱完全不能用的编译器是很无趣的，且对项目开发帮助不大，因为它们并不能展现 Rigel 的威力。最好的情况就是有一个只在部分情况下会出错的待测编译器，这也和现实情况

相吻合，毕竟 GCC/Clang 等编译器大多数情况下都是正确的，只在某些极端情况下会出错。在此感谢陈越琦同学提供了这么一款满足要求的 C-- 编译器 (Cmm-Compiler) 供开发测试使用。

下面我将展示几个正确或错误的例子，并详细解释说明其输出含义。

### 4.4.1 正确例子

图4.4，编译器通过了所有测试样例多轮的检测，暂时没有发现错误。

### 4.4.2 错误例子

#### 认假

图4.5，Rigel 发现待测编译器接受了错误的代码，因此报错退出。触发错误的代码见图4.6。

C-- 文法不支持隐式类型转换，因此 while 中的判断语句是不符合 C-- 语法规范的。然而待测编译器却没有报错，而是默默生成了中间代码。不过即便如此，这段代码也并不会死循环，但是编译器却生成了会死循环的代码。Rigel 正确地找出了这个 Bug。

#### 死循环

图4.7，生成的中间代码在 IR Simulator 中执行时超时，我们认为代码陷入死循环，因此报错退出。图4.8展示了触发了这个超时错误的代码。

显然，图4.8中的这段代码并不会死循环，应该很快就执行完毕退出。但是生成的 IR 代码却执行了相当长的时间还没有结束，这不符合语义。所以我们认为这是编译器的 Bug，它生成了死循环的代码。

#### 拒真

如图4.9所示，待测编译器提示 15、25、28、36 行有语法错误，因此中断编译，没有生成 IR 代码。然而 Rigel 认为这是合法的 C-- 代码，编译器理应正确编译生成 IR 代码，事与愿违，所以 Rigel 报了 *NO_IR_CODE_FOUND* 的错误。图4.10展示了被待测编译器拒绝的 EMI 变种 C-- 源文件。

显然，在图4.10中，15、25、28、36 行，都是完全合法的 C-- 代码，虽然代码看起来比较诡异，但却是完全合法的。很遗憾，待测编译器却拒绝了合法的代码，所以这是它的 Bug。

通过上述几个例子，我们充分感受到 Rigel 的强大威力，它能正确找出许多难以发现的潜在编译器 Bug。然而由于篇幅限制，我无法在此演示 Rigel 的全部功能，更多用法请参见文档与代码。

## 4.5 本章小结

本章详细介绍了 Rigel 的构建、运行方式，以及如何修改代码以适应不同需求。并通过几个实际的编译器错误的例子来帮助理解 Rigel 的输出格式，通过这几类常见的编译器 Bug，展现了 Rigel 的强大能力，既不会遗漏，更不会错杀。

```
ock.o csmith-Bookkeeper.o csmith-CFGEdge.o csmith-CGContext.o csmith-CGOptions.o csmi
.o csmith-CrestExtension.o csmith-DFSOutputMgr.o csmith-DFSProgramGenerator.o csmith-
ultRndNumGenerator.o csmith-DeltaMonitor.o csmith-DepthSpec.o csmith-Effect.o csmith-
ssionFuncall.o csmith-ExpressionVariable.o csmith-ExtensionMgr.o csmith-ExtensionValu
csmith-Finalization.o csmith-Function.o csmith-FunctionInvocation.o csmith-FunctionIn
eeExtension.o csmith-Lhs.o csmith-LinearSequence.o csmith-MspFilters.o csmith-OutputM
gramGenerator.o csmith-Reducer.o csmith-ReducerOutputMgr.o csmith-SafeOpFlags.o csmit
aSequence.o csmith-SplatExtension.o csmith-Statement.o csmith-StatementArrayOp.o csmi
r.o csmith-StatementFor.o csmith-StatementGoto.o csmith-StatementIf.o csmith-Statemen
mith-VectorFilter.o csmith-platform.o csmith-random.o csmith-util.o
make[2]: Leaving directory `/rigel/csmith/src'
make[2]: Entering directory `/rigel/csmith'
make[2]: Nothing to be done for `all-am'.
make[2]: Leaving directory `/rigel/csmith'
make[1]: Leaving directory `/rigel/csmith'
clang -lclang -std=c11 -Wall -Werror main.c gen_cmm.c emi.c clang_helper.c -o Rigel
CMakeLists.txt
Dockerfile
Makefile
Rigel
c_help.txt
clang_helper.c
clang_helper.h
common.h
csmith
emi.c
emi.h
emi.png
gen_cmm.c
gen_cmm.h
irsim-cli.py
main.c
outline.md
test_case
 ---> 6691fdfb7b32
Removing intermediate container 8818ce9688f9
Step 6 : VOLUME /rigel/compilers
 ---> Running in 2ab779795791
 ---> e87fe140230d
Removing intermediate container 2ab779795791
Step 7 : ENTRYPOINT /rigel/Rigel
 ---> Running in 1ad48c3c588b
 ---> 18597a33598c
Removing intermediate container 1ad48c3c588b
Successfully built 18597a33598c
```

**Figure 4.1**: 构建输出

```
→ Rigel git:(master) docker images
REPOSITORY                    TAG                 IMAGE ID            CREATED                  SIZE
rigel/v0.1                    latest              18597a33598c        Less than a second ago   1.838 GB
```

**Figure 4.2**: 镜像列表

```
emi.c@157/sem_check: syntax check pass, is_illegal: 0

emi.c@68/exec_ir_driver: exec command: timeout 5 python3 irsim-cli.py inter.ir
emi.c@71/exec_ir_driver: ir output buf: File loaded successfully.

emi.c@71/exec_ir_driver: ir output buf: stdout:  -1

emi.c@71/exec_ir_driver: ir output buf: Finish Program has exited gracefully.

emi.c@71/exec_ir_driver: ir output buf: Total instructions = 8

emi.c@71/exec_ir_driver: ir output buf: Simulation OK. Instruction count = 8

emi.c@71/exec_ir_driver: ir output buf: 0

emi.c@166/random_test: test cmm file:generated_cmm/0.c
emi.c@174/random_test:
choice: 2:
int w%d = 42343;
w%d = 1 * w%d;

clang_helper.c@87/sem_check_by_clang: tmp_dir/random_gen_file.c error_num:0

emi.c@125/cmm_compile: exec command: timeout 5 ./Cmm-Compiler-master/Code/parser tmp_dir/random_gen_file.c

emi.c@132/cmm_compile: found ir code
emi.c@137/cmm_compile: compiled tmp_dir/random_gen_file.c
emi.c@157/sem_check: syntax check pass, is_illegal: 0

emi.c@68/exec_ir_driver: exec command: timeout 5 python3 irsim-cli.py inter.ir
emi.c@71/exec_ir_driver: ir output buf: File loaded successfully.

emi.c@71/exec_ir_driver: ir output buf: stdout:  -1

emi.c@71/exec_ir_driver: ir output buf: Finish Program has exited gracefully.

emi.c@71/exec_ir_driver: ir output buf: Total instructions = 8

emi.c@71/exec_ir_driver: ir output buf: Simulation OK. Instruction count = 8

emi.c@71/exec_ir_driver: ir output buf: 0

Cong! No bug found!
You must be a lucky guy.
```

**Figure 4.4**: 测试通过输出

```
emi.c@285/test_compiler: meet file: generated_cmm/0.c

clang_helper.c@87/sem_check_by_clang: generated_cmm/0.c error_num:1

emi.c@125/cmm_compile: exec command: timeout 5 ./Cmm-Compiler-master/Code/parser

emi.c@132/cmm_compile: found ir code
emi.c@137/cmm_compile: compiled generated_cmm/0.c
emi.c@150/sem_check: false accepted.
emi.c@151/sem_check: found bug, BugType:5
bug detail: FALSE_ACCEPT
current seed: 1495294270
Rigel: emi.c:44: void report_bug(enum BugType): Assertion `0' failed.
```

**Figure 4.5**: 认假

```
root@1b4bf173e160:~/clang_vol/test_clang_ast# cat generated_cmm/0.c
int  func_1()
{
    int l_2 = 84715592;
    l_2 = (-8);
    while ((l_2 == 24))
    {
        int l_5 = 0;
        l_5 = 1;
        return l_2;
        l_2 = l_2+4;
    }
    return l_2;
}


int main ()
{
    write(func_1());

    return 0;
}
```

**Figure 4.6**: 触发认假 Bug 的代码

```
emi.c@71/exec_ir_driver: ir output buf: stdout:  32432423

emi.c@71/exec_ir_driver: ir output buf: stdout:  16

emi.c@71/exec_ir_driver: ir output buf: stdout:  43253623

emi.c@71/exec_ir_driver: ir output buf: stdout:  32432423

emi.c@71/exec_ir_driver: ir output buf: stdout:  17

emi.c@71/exec_ir_driver: ir output buf: stdout:  43253623

emi.c@71/exec_ir_driver: ir output buf: stdout:  32432423

emi.c@71/exec_ir_driver: ir output buf: stdout:  15

emi.c@71/exec_ir_driver: ir output buf: stdout:  43253623

emi.c@71/exec_ir_driver: ir output buf: stdout:  32432423

emi.c@71/exec_ir_driver: ir output buf: stdout:  16

emi.c@71/exec_ir_driver: ir output buf: stdout:  43253623

emi.c@71/exec_ir_driver: ir output buf: stdout:  32432423

emi.c@71/exec_ir_driver: ir output buf: stdout:  17

emi.c@71/exec_ir_driver: ir output buf: stdout:
emi.c@93/exec_ir_driver: return code:124
emi.c@94/exec_ir_driver: found bug, BugType:2
bug detail: CODE_DEAD_LOOP
current seed: 1495294399
Rigel: emi.c:44: void report_bug(enum BugType): Assertion `0' failed.
```

**Figure 4.7**: 死循环

```
int  func_1()
{
    int l_2 = 79787384;
    int l_5 = 73269690;
    l_2 = (-16);
    while ((l_2 != (-6)))
    {
        int l_6 = 27321238;
        l_5 = 87778416;
        l_6 = 29393354;
        l_2 = l_2+1;
    }
    if (l_2)
    {
        return l_2;
    }
    else
    {
        int l_7 = 34750377;
        l_7 = l_7;
        l_7 = l_7;
        l_7 = l_7;
    }
    l_2 = 0;
    while ((l_2 < 14))
    {
        int l_10 = 71227432;
        return l_10;
        l_2 = l_2+8;
    }
    if (l_5)
    {
        return l_5;
    }
    else
    {
        return l_2;
    }
}
```

**Figure 4.8**: 触发死循环 Bug 的代码

```
emi.c@181/random_test:
choice: 1:


emi.c@174/random_test:
choice: 2:
int w%d = 42343;
w%d = 1 * w%d;

clang_helper.c@87/sem_check_by_clang: tmp_dir/random_gen_file.c error_num:0

emi.c@125/cmm_compile: exec command: timeout 5 ./Cmm-Compiler-master/Code/parser

Error type B at Line 15 : near 'int' syntax error, unexpected TYPE
Error type B at Line 25 : near 'int' syntax error, unexpected TYPE
Error type B at Line 28 : near 'int' syntax error, unexpected TYPE
Error type B at Line 36 : near 'int' syntax error, unexpected TYPE
emi.c@135/cmm_compile: found bug, BugType:3
bug detail: NO_IR_CODE_FOUND
current seed: 1495294521
```

**Figure 4.9**: 拒真

```c
int  func_1()
{
    int l_2 = (-4);
    int l_3[6];
    int i0;
int s1556993525 = 1;
s1556993525 = s1556993525 + 1;
    i0 = 0;
int w948630786 = 42343;
w948630786 = 1 * w948630786;
    while (i0 < 6)
    {
int w1956011939 = 42343;
w1956011939 = 1 * w1956011939;
        l_3[i0] = 53992236;

        i0 = i0 + 1;
    }
int s168278187 = 1;
s168278187 = s168278187 + 1;
    l_2 = l_2;
int s1398399802 = 1;
s1398399802 = s1398399802 + 1;
    l_2 = 5;

    while ((l_2 >= 1))
    {
        int i1;
        return l_3[l_2];
int w1357088467 = 42343;
w1357088467 = 1 * w1357088467;

    }
    return l_3[0];
}
```

**Figure 4.10**: 被拒的正确代码

# 第五章　相关工作

本次毕业设计以及 Rigel 项目，是基于大量现有编译器测试验证相关的研究工作实现的。没有他们的杰出成果，我一定无法完成我的项目。这一章节将感谢并详细介绍项目的相关工作。

## 5.1 EMI-based Compiler Testing[1]

首先，最直接需要感谢的项目，便是苏振东教授提出的基于 EMI 的编译器检测方法。他的工作卓有成效，在检测 GCC 和 LLVM 编译器上取得了显著的效果。本次毕设的核心思路便完全参考了他的此篇论文。换言之，我的 Rigel 项目，便是在南京大学《编译原理》课程实验 C-- 编译器测试的特定应用场景下，对其论文思路的一种简易实现。虽然很钦佩和感谢苏教授的研究工作，但是本项目的代码为完全独立实现，没有对原作代码进行任何参考。

## 5.2 LLVM

毫无疑问，在编译器领域，LLVM 是最杰出的项目之一。它不仅是一款杰出的 C/C++ 编译器，更是一套完整且功能强大的库。借助它，可以很方便地进行编程语言设计、代码分析、代码优化与生成等工作。本次实验中，借助了 LLVM 项目的前端：Clang 库来进行语法分析，这大大减轻的我的工作压力，因为我不再需要去手动实现 parser。同时，考虑到编译原理实验的 C-- 语言为 C 语言的子集，因此我也借助 Clang 来进行辅助语法检测。虽然本项目中只使用 LLVM 来生成 AST，但是 LLVM 还有很多激动人心的功能，并且有规范的 API 和文档，值得向大家推荐。

## 5.3 Csmith

Csmith 是一款用于自动随机生成合法 C++ 语言代码的工具。它也应该是目前唯一一款能够自动随机生成代码的工具。它的功能非常强大，几乎覆盖了庞大的 C++ 语法的方方面面。它也作为一款优秀的编译器检测工具，发现了许多 GCC 和 Clang 的 Bug。不过它并没有完全覆盖 C-- 语法，因此我对它做了一些改动，以适应我的项目需要。

## 5.4 IR Simulator

非常感谢某位学长 (学姐) 编写的这个虚拟机小程序。这个虚拟机程序，是运行 C-- 编译生成的中间代码文件的平台，同时也是 Rigel 运行时必须依赖的平台。就好像玩电脑游戏必须要有一台电脑一样，测试编译器生成代码的正确性，也必须有这么一个虚拟机平台来运行生成的中间代码。此项目代码非常工整规范，鲁棒性很强，模块化做得很好，以至于我修改移植时并没有费太大功夫，再次对前人的工作表示感谢。

# 第六章　结论和后续工作

本篇论文先通过几个例子表现了编译器测试的重要性，然后详细介绍了一种创新且卓有成效的 EMI(Equivalence Modulo Input) 编译器测试方法。我们先从算法层面详细介绍其核心思想以及进行可行性分析。随后介绍了我实现的基于 EMI 方法、用于编译原理课程实验 C-- 编译器的测试工具：Rigel。我们详细介绍了其实现细节、使用方法以及运行效果。同时针对项目实现过程中的一些难点问题以及创新点进行了着重介绍。最后，感谢并简单介绍了对本次毕业设计产生帮助的几个项目。

虽然本次毕业设计以及毕业论文到此就暂告一段落了，项目也取得了预期的效果。不过如果继续深入探索的话，依然可以有更多的工作可以做。譬如：

- 提供完善美观的 UI，方便用户交互。
- 添加更多配置选项，以支持更多语法特性。
- 优化日志和报错处理逻辑，方便用户更快定位 Bug 位置以及获取更多详细信息。

同时，从工程角度，Rigel 还不够完善。文档、测试还不够充足。我们一向能正确认知项目的优点与不足，以助于日后进一步的改进。

# 参考文献

[1] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Notices*, volume 49, pages 216–226. ACM, 2014.

[2] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

# 简历与科研成果

## 基本信息

李想，男，汉族，1995 年 9 月生，江苏省淮安市人。

## 教育背景

| | | |
|---|---|---|
| *2013* 年 *9* 月 — *2017* 年 *6* 月 | 南京大学计算机科学与技术系 | 本科 |
| *2010* 年 *9* 月 — *2013* 年 *6* 月 | 江苏省淮阴中学 | 高中 |

## 攻读学士学位期间完成的学术成果

1. 没有。

## 攻读学士学位期间参与的科研课题

1. 没有。

## 攻读学士学位期间获得的主要奖励与荣誉

1. 没有。

# 致　　谢

首先非常感谢我的导师——许畅教授，在我的毕业设计从选题到后期实现全过程中提供了细致入微的全面悉心指导。没有他的帮助，我一定无法顺利完成我的毕业设计。

其次需要感谢我的同学们（尤其感谢陈越琦同学提供他的编译器供我测试），他们是一群能力卓越同时又乐于助人的 Geek。和他们在一起的四年，学到了很多东西，获得了成长和锻炼。虽然即将彼此告别、各奔前程，但我依然庆幸在生命中最宝贵的四年里遇见最可爱的你们。

最后也是最想感谢的便是我的父母，没有你们每个月给我打生活费，我一定没有力量坚持毕业。谢谢你们对我坚定的物质鼓励，比精神激励更有成效。

谢谢你们！