软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(6):1996−2011 [doi: 10.13328/j.cnki.jos.006565]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于前馈神经网络的编译器测试用例生成方法*

徐浩然[1], 王勇军[1], 黄志坚[2], 解培岱[1], 范书珲[1]

[1](国防科技大学 计算机学院, 湖南 长沙 410073)

[2](军事科学院 系统工程研究院, 北京 100097)

通信作者: 王勇军, E-mail: wwyyjj1971@126.com

摘 要: 编译器模糊测试, 是测试编译器功能性与安全性的常用技术之一. 模糊测试器通过产生语法正确的测试用例, 对编译器的深层代码展开测试. 近来, 基于循环神经网络的深度学习模型被引入编译器模糊测试用例生成过程. 针对现有方法生成测试用例的语法正确率不足、生成效率低的问题, 提出一种基于前馈神经网络的编译器模糊测试用例生成方法, 并设计实现了原型工具 FAIR. 与现有的基于 token 序列学习的方法不同, FAIR 从抽象语法树中提取代码片段, 利用基于自注意力的前馈神经网络捕获代码片段之间的语法关联, 通过学习程序设计语言的生成式模型, 自动生成多样化的测试用例. 实验结果表明, FAIR 生成测试用例的解析通过率以及生成效率均优于同类型先进方法. 该方法显著提升了检测编译器软件缺陷的能力, 已成功检测出 GCC 和 LLVM 的 20 处软件缺陷. 此外, 该方法具有良好的可移植性, 简单移植后的 FAIR-JS 已在 JavaScript 引擎中检测到两处软件缺陷.

关键词: 软件缺陷; 编译器模糊测试; 深度学习; 前馈神经网络; 抽象语法树

中图法分类号: TP311

中文引用格式: 徐浩然, 王勇军, 黄志坚, 解培岱, 范书珲. 基于前馈神经网络的编译器测试用例生成方法. 软件学报, 2022, 33(6): 1996–2011. http://www.jos.org.cn/1000-9825/6565.htm

英文引用格式: Xu HR, Wang YJ, Huang ZJ, Xie PD, Fan SH. Compiler Fuzzing Test Case Generation with Feed-forward Neural Network. Ruan Jian Xue Bao/Journal of Software, 2022, 33(6): 1996–2011 (in Chinese). http://www.jos.org.cn/1000-9825/6565.htm

Compiler Fuzzing Test Case Generation with Feed-forward Neural Network

XU Hao-Ran[1], WANG Yong-Jun[1], HUANG Zhi-Jian[2], XIE Pei-Dai[1], FAN Shu-Hui[1]

[1](College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China)

[2](Institute of System Engineering, Academy of Military Sciences, Beijing 100097, China)

Abstract: Compiler fuzzing is one of the commonly used techniques to test the functionality and safety of compilers. The fuzzer produces grammatically correct test cases to test the deep parts of the compiler. Recently, recurrent neural networks-based deep learning methods have been introduced to the test case generation process. Aiming at the problems of insufficient grammatical accuracy and low generation efficiency when generating test cases, a method for generating compiler fuzzing test cases is proposed based on feed-forward neural networks, and the prototype tool FAIR is designed and implemented. Different from the method based on token sequence learning, FAIR extracts code fragments from the abstract syntax tree, and uses a self-attention-based feed-forward neural network to capture the grammatical associations between code fragments. After learning a generative model of the programming language, fair automatically produce diverse test cases. Experimental results show that FAIR is superior to its competitors in terms of grammatical accuracy and generation efficiency of generating test cases. The proposed method has significantly improved the ability to detect compiler software defects, and has successfully detected 20 software defects in GCC and LLVM. In addition, the method has soundportability. The simple ported FAIR-JS has detected 2 defects in the JavaScript engine.

Key words: software defect; compiler fuzzing; deep learning; feed-forward neural network; abstract syntax network

* 基金项目: 国家自然科学基金(61472439); 国家重点研发计划(2018YFB0204301)

本文由“系统软件安全”专题特约编辑杨珉教授、张超副教授、宋富副教授、张源副教授推荐.

收稿时间: 2021-09-05; 修改时间: 2021-10-15; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

编译器是软件开发工具链中的核心组件. 开源编译器, 如 GCC、LLVM 等不仅是开源软件的典型代表, 同时也是开源社区的重要基础软件, 直接服务于数以百万计的用户. 编译器中的软件缺陷潜在地影响着所有经过编译产生的程序, 可能导致正确的源代码被翻译为存在安全缺陷的可执行程序, 从而在安全敏感的环境中带来灾难性的后果. 鉴于编译器在软件开发中的基础性和重要性地位, 其可靠性与安全性在学术界和工业界均得到了广泛的研究与关注[1].

模糊测试技术通过生成大量的测试输入并监测目标程序执行状态来检测软件缺陷, 是检测编译器软件缺陷的有效方法之一. 编译器模糊测试面临的一个挑战是生成语法正确的测试用例. 研究表明[2], 编译器中的软件缺陷主要存在于中端和后端. 只有具备正确语法规范的测试用例才能通过编译器的解析阶段(词法分析和语法分析等), 避免在前端处理阶段就被丢弃, 从而无法对更为复杂、脆弱的中端和后端(如代码优化和代码生成等组件)展开测试. 早期的研究工作基于目标语言的语法规范产生测试用例. 例如, Csmith[3]基于给定的 C 语言概率语法随机产生的测试程序可以轻易地通过编译器的解析阶段. 然而, 该项工作指出了提供目标语言的语法规范是一项费时费力的工作. 此外, 这种方式生成测试用例的逻辑和代码与目标语言紧密绑定, 难以迁移到新的目标语言和编译器上.

近来, 通过将测试程序的生成问题建模为语言模型, 深度神经网络技术被引入到编译器模糊测试用例的生成过程. 研究者使用循环神经网络及其变种对字符级或 token 级的代码序列进行建模, 从大量样例程序中自动学习目标语言的语法规范. 这种方法无需提供目标语言的语法规范, 极大地节省了模糊测试器的开发时间. 模型接收部分源代码序列作为条件输入, 对后续代码序列进行预测. 模型捕获条件输入中的语法和语义信息越多, 就越容易生成具有正确语法和语义的代码. 然而, 循环神经网络通常按照时间步对输入进行序列化的逐项处理, 随着输入长度的增加, 循环神经网络及其变体面临着梯度消失的问题[4]. 因此, 种子程序中存在的长距离语法依赖关系, 给现有方法生成语法正确的测试用例时带来了新的挑战. 此外, 现有方法[2,5]将源代码视为普通文本序列, 没有考虑源代码中语义丰富的结构化信息. 如何充分利用源代码中的结构化信息, 以进一步增强模型生成有效测试用例的能力, 也是当前编译器模糊测试面临的挑战之一.

本文提出了一种新颖的基于前馈神经网络的编译器模糊测试用例生成方法以解决以上挑战. 该方法能够捕获源代码中广泛存在的长距离语法依赖关系, 利用抽象语法树中的结构化信息, 提高生成测试用例的有效性和生成效率. 与先前工作在字符级或 token 级构建语言模型不同, 为了充分利用源代码中的结构化信息, 本文从抽象语法树中提取子树构成代码片段序列, 以对 C 程序进行表示和生成. 序列中每一个代码片段的特征表示都将直接从该序列所有其他代码片段中学习得到, 从而实现对长距离语法依赖关系的有效嵌入. 为了实现这一目标, 本文采用基于自注意力的前馈神经网络来捕获代码片段之间的语法关联, 通过学习输入序列中一系列上下文感知的特征表征对后续的代码序列进行预测. 不同于循环神经网络中序列化的处理方式, 前馈结构的神经网络模型可以并行处理输入序列中的每一个单元, 从而通过并行运算提高生成测试用例的效率.

本文实现了一个编译器模糊测试原型工具 FAIR (feed-forward neural network based compiler fuzzer), 以代码片段为基本单位学习 C 程序的生成式模型. 为了评估 FAIR 的有效性, 本文将原型工具应用于 C 语言编译器的模糊测试. 实验结果表明: 与同类型的先进方法相比, FAIR 能够在单位时间内生成更多的测试程序, 并且显著提升了生成语法正确测试程序的比例, 进而大幅度提升了生成测试用例的解析通过率. 得益于较高的解析通过率和生成效率, FAIR 成功地在 GCC 和 LLVM 中发现了 20 处软件缺陷, 其中 3 处影响该编译器 2014 年以来的全部版本. 本文提出的方法具有较好的可移植性. 方法移植到 JavaScript 引擎的模糊测试后, 已在 ChakraCore 引擎中检测到了 2 处软件缺陷并获得开发团队确认.

本文主要贡献如下: (1) 提出了一种新颖的基于前馈神经网络的编译器模糊测试用例生成方法, 显著提高了生成测试用例的有效性和生成效率; (2) 实现了编译器模糊测试原型工具 FAIR, 对本文提出方法进行了实验验证; (3) FAIR 在主流的 C 语言编译器中检测到 20 处软件缺陷, 良好的可移植性使 FAIR 在 JavaScript 引擎中检测到 2 处软件缺陷. 相关信息均已提交软件开发团队.

本文第 1 节介绍编译器模糊测试的研究背景. 第 2 节介绍所提方法的框架. 第 3 节介绍 FAIR 与模型训练

有关的内容. 第 4 节介绍 FAIR 测试用例生成和模糊测试部分的内容. 第 5 节介绍本文的实证研究并对实验结果进行讨论. 第 6 节总结全文并展望未来的工作.

1 相关工作

编译器是使用最广泛的大型软件之一. 近年来, 经过研究者们的持续努力, 编译器的性能和可靠性得到了显著提升. 然而, 由于编译器的复杂性, 软件缺陷仍然是影响其安全性和可靠性的重要威胁. 编译器软件缺陷可以分为两种类型[6]: 一种类型会导致编译器程序崩溃(crash); 另一种会导致编译器生成与程序员意图不一致的错误代码, 称为错误编译(miscompilation). 差异测试(differential testing)[2,3]是检测错误编译类型缺陷的有效方法之一. 差异测试通过使用不同的编译器软件或编译优化等级对同一测试用例进行编译, 并对比输出程序的运行结果, 可以检测出潜在的错误编译问题. 除差异测试的方法外, Le 等人[6–8]提出并应用了等价取模测试(equivalence modulo inputs)的方法, 在主流编译器中检测出了一系列错误编译类型的软件缺陷. 本文工作聚焦于编译器崩溃类型 bug 的检测.

模糊测试是一种致力于发现目标软件中安全缺陷的有效技术. 测试过程中, 模糊测试器不断运行目标软件, 并使用产生的测试用例作为软件输入, 监测目标软件是否出现可能暴露代码缺陷的异常行为, 例如程序崩溃(crash)等. 测试用例的质量是影响模糊测试效果的关键因素之一. 根据生成测试用例方式的不同, 模糊测试器可以分为两类: 基于变异的模糊测试器和基于生成的模糊测试器. 基于变异的模糊测试器[9]通过对种子输入进行修改产生新的测试输入, 这种方式能够有效地对处理非结构化数据类型的软件(如图像和视频处理软件)进行测试. 对于接收结构化数据输入的编译器来说, 基于随机变异方式生成的测试输入难以通过编译器的解析阶段, 而基于生成的模糊测试器基于给定的语法规范产生的测试用例可以通过目标软件的合法性检查, 因此能够轻易通过编译器的解析阶段. 例如, Csmith 使用给定的上下文无关文法, 通过随机选择生成规则产生新的 C 程序. LangFuzz[10]基于上下文无关文法从程序集合中提取代码片段, 随后通过组合代码片段的方式产生新的 JavaScript 引擎测试用例. 然而, 构建模糊测试器所需的语法规范十分低效. 以 Csmith 为例, 该软件历经数年开发完成, 包含了数千行手工编写的 C++代码.

由于循环神经网络具有处理序列数据的优越能力, 近来, 研究者们将测试用例生成建模为语言模型[11]问题, 将循环神经网络作为测试用例生成模型的基础组件[12]以生成编译器模糊测试用例. 在 Godefroid 等人首次将循环神经网络学习 PDF 文件的格式规范, 并使用学习到的字符级生成式模型应用于 PDF 解析器的模糊测试[13]后, Cummins 等人提出了一种使用 LSTM 模型[14]自动学习 OpenCL 编程语言语法规范并生成相应编译器模糊测试用例的方法[2]. 该工作实现的原型工具 DeepSmith 通过学习 OpenCL 语言 token 级的语言模型, 能够基于前缀 token 序列预测后续的 token 进而生成测试代码. Liu 等人基于 LSTM 的序列到序列(seq2seq)模型, 从样例程序中学习字符级的 C 语言生成模型[5]. 以上方法无需提供目标语言的语法规范, 可以节省大量的模糊测试器开发时间.

然而, 由于循环神经网络需要学习序列化地传递信息, 使得基于循环神经网络的方法难以捕获代码序列中的长距离语法依赖关系[15]. 以 C 程序为例, 成对的括号是源代码中一种广泛存在的长距离语法依赖关系. 左括号可能位于程序起始处附近, 与之匹配的右括号则可能位于程序的结尾处. 特别是当程序中存在多级嵌套的左括号时, 基于循环神经网络的方法难以生成正确数量的右括号进行闭合, 从而导致生成存在语法错误的测试用例. 变量的定义和使用同样存在类似的问题. 图 1 展示了两个因未能有效捕获源程序中的长距离语法依赖关系而产生存在语法错误 C 程序的示例. 图 1(a)中程序因模型未能生成正确数量的右括号对左括号进行闭合导致语法错误; 图 1(b)中程序因模型未能捕获变量定义语句中的类型信息, 导致生成的赋值语句出现语法错误. 以上两个测试用例在编译器前端的语法分析阶段就被丢弃, 从而无法到达中端和后端的处理流程. 近来, 针对循环神经网络的梯度消失问题, 研究者们引入了注意力机制加以缓解. 目前, 注意力机制已经在机器翻译[16,17]、语言模型[18,19]、文本分类[20]以及文本总结[21,22]等领域得到了广泛应用.

源代码中含有丰富的结构化信息, 如何充分利用这些信息增强深度学习模型以生成有效的测试程序的能

力, 是当前亟待解决的问题[23]. 相比于使用源代码的字符流或 token 流, Uri 等人提出使用抽象语法树的节点序列学习源代码的表征(code representations), 在程序属性预测[24]、代码补全任务[25]以及源代码漏洞检测[26,27]等领域均提升了模型预测性能.

以代码片段(code fragment)为基本单位组建抽象语法树生成源程序的方法, 在代码执行引擎的模糊测试中得到了成功应用. LangFuzz 基于语法规范从样例程序集中提取代码片段构建代码片段池, 并通过代码片段替换的方式对 JavaScript 种子程序进行变异生成新的测试程序. IFuzzer[28]从样例程序中解析抽象语法树, 使用代码片段提取器提取抽象语法树的子树作为代码片段, 随后通过遗传算法基于给定的语法生成规则组合代码片段以产生新的测试程序. Han 等人[29]基于抽象语法树结构将种子程序分割为独立的代码块(code bricks), 通过语义感知的方式进行代码块的拼接以生成新的测试用例. Lee 等人[30]使用 LSTM 模型学习 JavaScript 程序代码片段的语言模型, 进而生成测试用例. 上述研究表明: 从源程序的抽象语法树中提取子树作为后续学习的代码片段可以有效地利用代码中丰富的结构化信息, 帮助生成语法正确的测试用例.

```
1  int main(·){                          int main(·){
2      int e[8];                             int e[8];
3      int x[8]={4,16,8,32,64,1,128,2};      int x[8]={4,16,8,32,64,2,1,128};
4      int i, j, t;                          int i, j, t;
5      for (i=0; i<7; i++) {                 for (i=0; i<7; i++) {
6          for (j=0; j<7; j++) {                 for (j=0; j<7; j++) {
7              if (x[j]>x[j+1]) {                    if (x[j]>x[j+1]) {
8                  t=x[j+1];                             t=x[j+1];
9                  x[j+1]=x[j];                          x[j+1]=x[j];
10                 x[j]=t;                               x[j]=t;
11                 //以上代码用作模型输入数据            }
12             }                                     }
13         }                                     }
14     }                                         //以上代码用作模型输入数据
15 }                                             e='c';
16     //...                                     //...
17 }                                         }
```

(a) (b)

图 1 存在语法错误的测试用例示例

2 FAIR 方法框架

本文提出了一种新颖的基于前馈神经网络的编译器模糊测试用例生成方法, 以解决现有的基于循环神经网络的方法面临长距离语法依赖关系时难以生成语法正确的测试用例的问题. 方法以抽象语法树子树为基本单位构建完整的测试程序, 充分利用抽象语法树中的结构化信息增强模型生成有效测试用例的能力.

本文实现了一款编译器模糊测试原型工具 FAIR, 架构总览如图 2 所示.

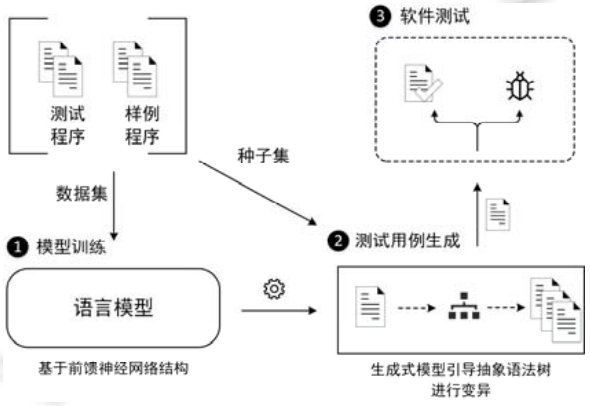


图 2 FAIR 设计总览

FAIR 由 3 部分组成, 包括学习模块、生成模块和测试模块. 学习模块以样例程序为数据集训练一个基于前馈神经网络的生成式语言模型, 以代码片段为基本单位构建抽象语法树从而生成测试程序. 代码片段为从样例程序中提取的抽象语法树子树, 是语言模型的基本单位. 生成模块基于种子集和学习得到的模型通过对抽象语法树变异的方式产生新的测试用例. 最后, 测试模块运行目标编译器加载测试用例并展开测试.

3 模型训练

使用深度学习模型从样例程序中学习 C 程序的模式, 是构建高质量语言模型的关键. 本节将依次介绍数据集的收集方式、代码片段的提取方式以及前馈神经网络的模型结构.

3.1 数据集与预处理

高质量的样例程序有助于神经网络模型更好地学习 C 程序的语法模式, 从而生成有效的测试程序. 在 FAIR 的实现过程中, 样例程序集既是模型训练的数据集, 也是后续模糊测试的种子集. 本文从开源编译器(GCC 与 LLVM 等)的测试套件中收集语法正确的测试程序, 构建 C 语言的样例程序集. 样例程序集中既包含测试编译器基本功能组件的测试程序, 也包含目标编译器的回归测试程序. 研究人员通过统计分析发现, 触发新的软件缺陷的测试用例含有较高比例的与回归测试集里相重合的代码片段[30]. 因此, 以回归测试程序作为样例程序学习目标语言的语言模型有利于生成触发编译器软件缺陷的程序.

为了减少数据集中噪声因素的影响, 本文对收集到的样例程序集进行预处理. 预处理包括移除种子程序中所有的注释代码, 并展开所有的宏代码.

由于 C 语言中标识符命名方式的灵活性, 数据集中含有大量的标识符, 导致源代码生成模型的词汇表非常大. 巨大的词汇表会严重影响代码模型的性能. 为了增强模型学习 C 语言语法的能力, 本文进一步对数据集进行标识符精简, 对数据集中所有开发者自定义的标识符按照一致的形式进行重命名. 例如, 根据声明的顺序, 将开发者定义的变量按照{*var*_0,*var*_1,*var*_2,...}的形式进行重命名. 将数据集中的函数名按照{*func*_0, *func*_1,*func*_2,...}的形式进行重命名. 除了常见的变量名和函数名, 本文还对其他类型的标识符进行了重命名, 包括结构体标识符、枚举标识符等.

3.2 代码片段提取

相比于一般的文本序列, 源代码有着严格的语法规范和丰富的结构化信息. 为了充分利用源代码中的结构化信息生成语法正确的测试用例, 本文基于抽象语法树中的代码片段构建 C 程序的语言模型.

每个语法正确的 C 程序都可以对应于一棵抽象语法树. FAIR 通过提取抽象语法树中高度为 1 的子树获得代码片段. 高度为 1 的语法树子树指的是抽象语法树中以每个非终结符为根节点、包含根节点的所有子节点以及根节点与子节点之间的连接边所构成的子树. 以图 3 为例, 这是从完整语法树中截取的一个函数定义的语法树子树. 在该抽象语法树中, 白色节点对应于源程序中的非终结符, 深色节点对应于终结符.

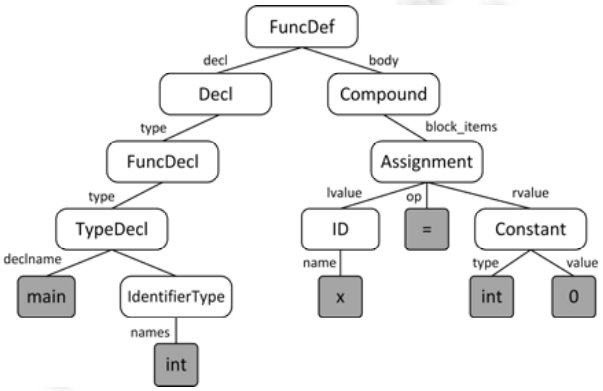


图 3 抽象语法树示例

从该语法树中提取得到的代码片段序列如图 4 所示, 例如, 非终结符 FuncDef 对应的代码片段为图 4 中的第 1 项, Assignment 节点对应的代码片段为图 4 中的第 7 项.

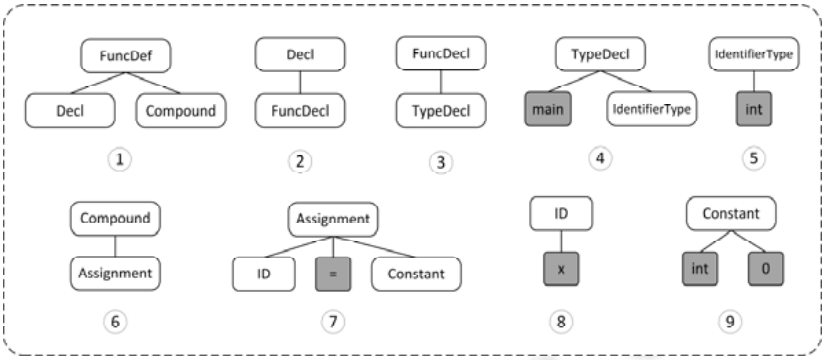


图 4 代码片段示例

由于语法树子树中包含非终结符及其子节点, 因此代码片段中含有关于该非终结符的展开规则. FAIR 将子树的根节点记为子树的类型, 即该代码片段的类型.

FAIR 通过前序遍历抽象语法树获得输入程序的代码片段序列. 从抽象语法树的根节点开始, 每访问一个非终结符, 就意味着开始处理一个代码片段, 该非终结符节点即为代码片段的根节点. 通过前序遍历抽象语法树, 可以按照代码片段在源程序中的出现顺序获得对应的代码片段序列. 例如, 图3中抽象语法树所对应的代码片段序列如图 4 所示的序号标注.

3.3 模型结构

FAIR 对输入程序的代码片段序列进行建模, 基于序列中的已有代码片段预测后续的代码片段. 为了对代码片段序列中长距离的语法依赖关系进行有效建模, FAIR 采用了基于自注意力机制的前馈神经网络模型构建语法树片段的生成式模型. 模型架构如图 5 所示.

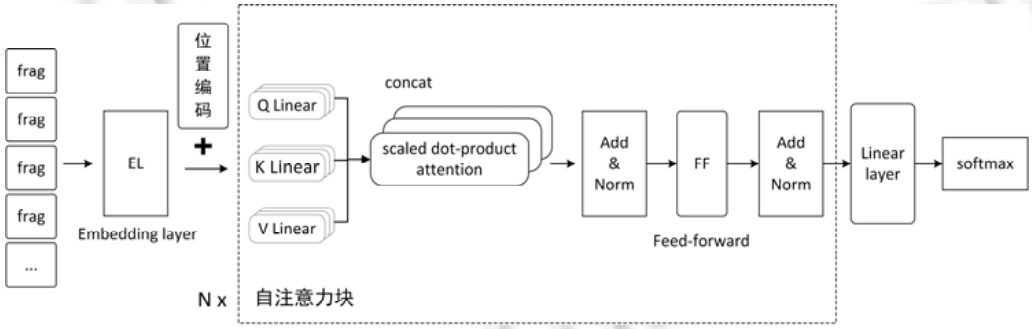


图 5 FAIR 神经网络模型

输入序列首先进入模型嵌入层(embedding layer). FAIR 使用学习到的代码片段嵌入将代码片段转换为向量表示. 输入序列中代码片段的先后顺序是一项重要的语义信息, 为了利用位置信息, 模型随后向输入序列的嵌入加入位置编码, 从而区分序列中出现在不同位置的代码片段. 具体而言, 本文采用广泛使用的正弦和余弦函数进行位置编码:

$$PosEnc(pos,2i)=\sin\left(\frac{pos}{10000^{2i/d}}\right) \tag{1}$$

$$PosEnc(pos,2i+1)=\cos\left(\frac{pos}{10000^{2i/d}}\right) \tag{2}$$

其中, *pos* 代表元素在序列中的位置, *d* 代表输入和输出的维度.

FAIR 采用的前馈网络模型由多个注意力块(attention blocks)组成, 其中每个注意力块由一系列相同的层堆叠组成, 每层包含一个多头自注意力子层和一个前馈网络子层. 在自注意力子层中, 模型通过对输入序列中向量的加权平均产生输出序列:

$$y_i = \sum_j w_{ij} x_j \tag{3}$$

其中, 权重 w_{ij} 来源于序列中代码片段向量之间的点积. 为了引入该子层的可控参数, 每个输入向量具有 3 种不同的使用方式, 分别称为查询(query)、键(key)和值(value).

具体来说, 基于位置编码后的序列, 通过使用线性层计算输入序列在 3 个维度上的映射, 得到输入向量 3 种不同使用方式下的新表示:

$$Q=W^q X \tag{4}$$

$$K=W^k X \tag{5}$$

$$V=W^v X \tag{6}$$

随后, 使用点积的方式计算每个输出向量加权权重. 其中, 查询向量用于计算自身输出的权重、键用于计算其他向量输出的权重、值作为加权和的一部分用于计算每个输出向量:

$$w'_{ij} = q_i^T k_j \tag{7}$$

$$w_{ij} = softmax(w'_{ij}) \tag{8}$$

$$y_i = \sum_j w_{ij} v_j \tag{9}$$

为了将权重的总和设定为1, 使用 *softmax* 函数作用于点积的结果. $\sqrt{d}$ 用于对点积的结果进行缩放, 避免 *softmax* 函数对数值过于敏感, 公式如下:

$$Attention(Q,K,V) = softmax\left(\frac{QK}{\sqrt{d}}V\right) \tag{10}$$

自注意力机制通过对输入序列中的每个向量进行加权平均的方式产生输出序列, 因而序列中任意两个代码片段之间的距离都可以看作是 1, 尽管该距离值在实际输入序列中可能很大. 这使得模型能够计算序列中任意两个代码片段之间的语法以及语义关联而不受距离因素的限制, 从而避免了在循环神经网络中由于序列元素距离过远导致的弱关联度问题.

为了联合使用多个表征子空间内的信息, FAIR 采用多头自注意力机制. 模型首先计算 *Q*, *K*, *V* 的线性变换, 然后再将结果反馈至自注意力机制中. 对于第 *j* 个头, 自注意力子层的输出按如下方式计算:

$$H_j(Q,K,V) = Attention(QW_j^Q, KW_j^K, VW_j^V) \tag{11}$$

模型随后对多个注意力头的结果进行拼接并作用于线性变换, 得到多头自注意力子层的输出:

$$H=concat(H_1,H_2,\ldots,H_h) \tag{12}$$

$$Y=HW^Y \tag{13}$$

多头注意力子层的输出将进入前馈网络层, 并添加归一化和残差连接得到注意力块的输出. 最后, 注意力块的输出被送入全连接层并作用 *softmax* 激活函数. FAIR 使用该输出预测下一个代码片段.

4 测试用例生成与模糊测试

本节将介绍 FAIR 基于学习到的语言模型以及种子程序, 生成新测试用例的方法以及如何对目标编译器进行测试.

4.1 测试用例生成方法

正如前文所述, 本文在模型训练阶段所使用的数据集同时也是模糊测试阶段的种子集. FAIR 通过基于给定的种子程序进行变异的方式产生新的测试用例.

算法 1 的 *TestCaseGeneration* 函数(第 1 行–17 行)展示了测试用例生成过程. 给定一个种子程序, FAIR 将其转换为抽象语法树后, 随机选择一个子树并进行裁剪. 裁剪时, 以选择的非终结符节点为根节点, 移除以该节点为根节点的所有子节点(第 4 行). 裁剪后, 基于裁剪点前的代码片段序列 *partialTree* 作为模型输入以预测后续的代码片段, 从而对裁剪位置进行补全(第 7 行).

算法 1. 测试用例生成方法.

输入: AST 种子集(*AS*), 训练得到的前馈神经网络模型(*model*), 采样温度(*T*), 生成切片的最大数量(max);

输出: 新的测试用例.

```
1  function TestCaseGeneration
2      seed←SelectSeed(AS)
3      node←seed.root(·)
4      partialTree, prunedType←PruneSeed(seed)
5      count←0
6      while count<max do
7          weights←PredictNextFrag(model,partialTree,prunedType)
8          newFrag←Sampling(weights,T)
9          AppendFrag(node,newFrag)
10         if genFinished(·) then
11             break
12         end if
13         partialTree.append(newFrag)
14         cound←count+1
15     end while
16     return node
17 end function
18 function Sampling(weights,temperature)
19     weights←weights.div(temperature).exp(·)
20     choice←multinomial(weights)
21     return choice
22 end function
23 function AppendFrag(node,newFrag)
24     if IsPrunedNonterminal(node) then
25         node←newFrag
26         return
27     end if
28     C←node.child(·)
29     for child in C do
30         AppendFrag(child,newFrag)
31     end for
32 end function
```

图 6 展示了对语法树进行变异的示例. 在该示例中, 模型选择了语法树的 Decl 子树进行变异. 首先从语法树中移除 Decl 及其所有的子节点; 随后, 以该抽象语法树剩余部分的前序序列为条件输入, 模型依次预测产生 Decl 代码片段、TypeDecl 代码片段以及 Struct 代码片段, 依次添加到原语法树中完成对裁剪子树的变异.

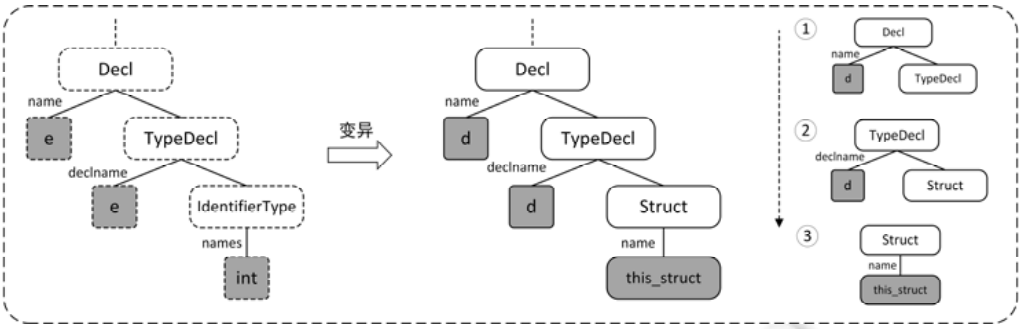


图 6 变异示例

从裁剪子树的根节点开始，FAIR 记录待预测的下一个代码片段的类型 *prunedType*，并在模型预测后基于类型过滤模型输出的概率分布，仅保留与目标类型一致的代码片段. 由于 FAIR 以代码片段(语法树子树)为基本生成单元，每生成一个代码片段后，通过该片段内子节点的信息即可判断该片段是否存在下一级待展开的非终结符，并通过下一级非终结符的类型信息确定下一级代码片段的类型. 相比于基于语法生成规则的生成方式，FAIR 在预测得到一个代码片段的同时也生成了该代码片段的根节点的展开规则. 这种生成方式有利于产生语法正确的测试用例.

每当模型预测得到一个代码片段，FAIR 通过 *AppendFrag* 函数将其加入裁剪后的抽象语法树中(第 23 行–32 行)，并检查此时语法树中是否仍存在待展开的非终结符(第 10 行): 如果不存在，表示语法树的补全已经完成，测试程序生成完毕；如果存在，则更新当前的前缀代码片段序列作为模型所需要的上下文信息，进行下一步的预测.

4.2 采样方法

当对以编译器为代表的复杂软件进行模糊测试时，一方面需要测试用例尽量符合规范，从而可以通过目标软件的解析阶段(如词法分析和语法分析等)；另一方面，又需要测试用例尽量多样化以触发目标软件中的缺陷. 为了持续地对目标编译器进行模糊测试，FAIR 通过对模型预测结果进行采样的方式生成多样化的输出程序.

常用的采样方式包括贪心采样和随机采样: 贪心采样是指总是从模型输出中选择概率最大的一项使用，该采样方式生成的测试程序格式往往是良好的，但缺少多样性；随机采样是指从模型输出的概率分布中随机选择一项使用. 相比于完全随机的方式，在模糊测试阶段，FAIR 从模型输出的多项式分布中进行采样，使得具有较高预测概率的代码片段有较大的概率被选择，同时，模型也有一定的概率来探索其他可能的输出.

算法 1 的 *Sampling* 函数(第 18 行–22 行)展示了从多项式分布采样的方法. 本文使用采样温度控制从多项式分布中采样的随机程度: 采样温度越低，代表模型越保守，越容易选择具有较大预测概率的结果；采样温度越高，代表模型越激进，越有可能选择具有较小预测概率的结果.

4.3 编译器测试

测试用例生成完成后，FAIR 运行目标编译器展开对测试用例的编译测试. 由于本文主要关注编译器中的软件缺陷，因此在运行编译器时，设定编译参数为*–S*，表示仅编译，不执行代码汇编和链接的处理. 本文在测试时为每个编译进程设置 10 s 的超时时间，如果超时但编译仍未完成，则终止该编译进程.

模糊测试中，FAIR 重点关注导致编译器崩溃的软件缺陷类型. 例如，GCC 和 LLVM 中定义了一类特殊类型的编译器软件缺陷，称为“内部编译器错误(internal compiler error)”. 该类型错误作为一种编译器反馈缺陷的机制，由编译器内部的缺陷代码引发，而非由源代码中的编程错误引发. 触发内部编译器错误时，编译器将引发崩溃并打印函数调用栈，以辅助对缺陷进行定位分析.

5 实证分析

5.1 实验设计

为了验证所提方法的有效性, 本文提出了 5 个问题并通过实验进行验证.

(1) FAIR 产生测试用例的解析通过率如何?

(2) FAIR 产生测试用例的效率如何?

(3) FAIR 检测编译器软件缺陷的能力与现有方法相比如何?

(4) 将 FAIR 应用于实际的编译器模糊测试中, 测试效果如何?

(5) 本文所提出的方法是否容易迁移到其他代码编译器或执行引擎?

生成测试用例的解析通过率反映了模型生成语法正确的测试用例的能力. 生成式模型训练完成后, 模型编码了 C 语言的语言模式, 本文使用解析通过率这一指标来衡量模型学习 C 语言模式的效果.

生成测试用例的效率反映了生成式模型在模糊测试中的运行效率. FAIR 通过生成多样化的测试用例来对编译器巨大的输入空间进行探索, 因此生成测试用例的效率也是影响模糊测试效果的重要因素之一.

与现有方法检测编译器软件缺陷能力的对比, 反映了本文所提出方法的有效性. 本节将 FAIR 与现有方法应用于真实编译器的模糊测试, 并对检测效果进行对比.

本文的目标是在现实的编译器中检测软件缺陷, 因此, 本节将 FAIR 应用于主流生产级编译器 GCC 与 LLVM 的模糊测试中, 并使用在主流编译器中检测到的软件缺陷数作为评估 FAIR 有效性的指标之一.

最后, 将论证本文所提出的方法是否具备良好的可移植性, 能够迁移到其他编译器或代码执行引擎.

由于基于 Liu 等人[5]实现的原型工具 DeepFuzz 开源代码未能复现出论文所描述的效果, 本文采用 DeepSmith 中的模型与方法作为实验基准. DeepFuzz 采用字符级的语言模型, 文献[5]中提到, 构建字符级的语言模型易于实现但却需要模型学习复杂的 token 级语法, 而 DeepSmith 是面向 OpenCL 编译器的 token 级语言模型, 因此本文将 DeepSmith 中的 token 级语言模型迁移到 C 语言编译器中作为实验基准. DeepSmith 采用循环神经网络 LSTM 模型, 在实现该模型的过程中, 设置嵌入层维度为 128, 隐藏层维度为 512. FAIR 中设置 6 个自注意力层, 层间 dropout 设置为 0.1, 多头注意力个数设置为 8, 嵌入层维度和隐藏层维度分别设置为 128 和 512. 模型训练过程中, DeepSmith 与 FAIR 的优化方法均采用带动量的随机梯度下降方法, 动量设置为 0.9.

生成式模型接收条件输入作为上下文信息进而预测后续的代码序列. 条件输入越长, 模型可获取的上下文信息越多. 本文实验将各模型的条件输入最大长度设定为 64.

本文实验内容均开展于同一计算平台, 操作系统版本为 Ubuntu 20.04, 处理器为 Intel(R) Xeon(R) Silver 4210R CPU, 内存为 32 G, 显卡型号为 TITAN RTX, 开发语言及版本为 Python 3.8.

5.2 RQ1: 生成测试用例的解析通过率

生成测试用例的解析通过率反映了深度学习模型学习 C 语言模式的效果以及模型生成有效测试用例的能力. 生成能够通过编译器解析阶段的测试用例对于模糊测试过程至关重要, 只有能够通过解析阶段的测试用例, 才能对编译器中更为复杂、脆弱的中端和后端展开测试. 本节将评估 FAIR 生成测试用例的解析通过率, 此外, 还将评估采样方法以及采样温度对通过率的影响. 本文采用编译通过率对解析通过率进行近似. 实验过程中, 对 FAIR 生成的测试用例进行编译测试, 能够通过编译的即为通过了解析阶段. 需要说明的是: 本部分实验中, 导致编译失败的原因均为测试用例存在语法错误而非触发了编译器中的 bug. 我们使用 GCC 的正式版本 GCC-9 开展本部分实验.

Liu 等人[5]指出: 通过随机插入代码行方式生成测试用例的解析通过率最高, 这是因为该方式避免了从原始种子程序中删减代码, 有利于保护种子程序的原有代码结构. 因此, 本部分实验收集 DeepSmith 在随机插入代码行策略下的解析通过率进行对比, 这是 DeepSmith 生成测试用例解析通过率最高的一种策略. 实验结果如图 7 所示. FAIR 生成测试用例的解析通过率达到了 72.11%, 相比于 DeepSmith 提高了 28.61%. 一方面, 这是因为基于注意力机制的前馈神经网络模型捕获源代码长距离语法依赖关系的能力较强; 另一方面, 基于抽

象语法树代码片段的方式，有效利用了种子程序的结构化信息，增强了模型生成语法正确测试用例的能力.

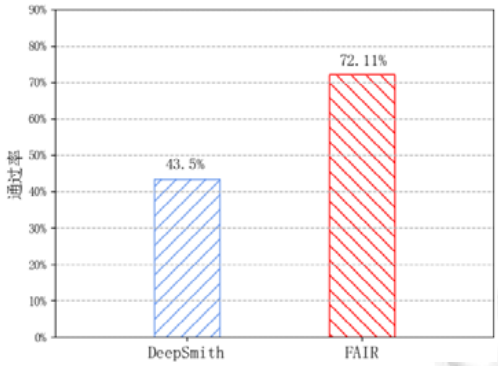
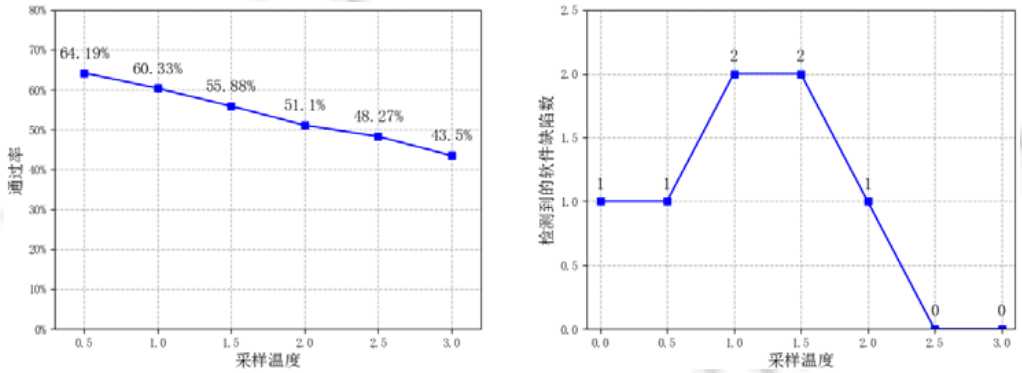


图 7　生成测试用例解析通过率的对比

从模糊测试的角度，对模型的输出进行采样能够为生成式模型带来多样化的输出. 本文所采用的随机采样方法可以将模型从种子集中学习到的多种代码模式联合起来，产生多样化的测试用例. 这种采样方式带来的副作用是模型的输出并不总是语法正确的，特别是在采样温度较高、模型较为激进时. 为了分析采样温度对生成测试用例解析通过率的影响，本文选择 6 个不同的采样温度值，并收集不同的采样温度下生成测试用例解析通过率的变化进行分析.

实验结果如图 8(a)所示. 随着采样温度的升高，生成式模型变得更激进，导致 FAIR 生成测试用例的通过率呈下降趋势. 当采样温度达到 2.5 时，FAIR 生成测试用例的解析通过率下降到 50%以下.



(a) 采样温度对解析通过率的影响　　(b) 采样温度对检测软件缺陷数的影响

图 8　采样温度对测试用例解析通过率以及检测软件缺陷的影响

5.3 RQ2: 生成测试用例的效率

本文使用单位时间内生成并测试的测试用例总数量评估不同模型与方法的效率. 实验中设置的时间跨度为 1 小时. 将对比 1 小时内 FAIR 与 DeepSmith 生成并测试的测试用例数. 本部分实验中的目标软件为 GCC-9，模型均采用贪心采样方法.

实验结果如图 9 所示. 受益于前馈神经网络结构，FAIR 在生成测试用例阶段基于条件输入预测后续 token 时，能够并行地对条件输入序列进行运算，无需序列化地对种子程序中的 token 序列进行逐项处理. 最终，相比于 DeepSmith 在相同的时间间隔内生成并测试了更多数量的测试用例，超过 DeepSmith 的 28.5%.

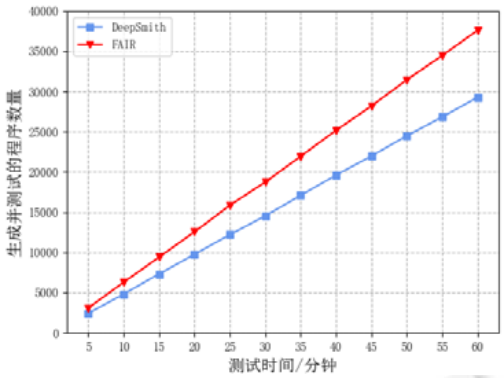


图 9　1 小时内生成并测试的程序数量

5.4 RQ3: 检测编译器软件缺陷的能力

本部分实验首先对模型采样温度对检测软件缺陷的影响进行了实证分析, 将 FAIR 在不同的采样温度下分别进行 100 个进程时(进程数×小时数)的模糊测试, 对检测到的编译器 bug 数进行了对比. 本部分实验选择正处于开发中、即将发布正式版的 GCC-12 作为目标软件进行测试.

实验结果如图 8(b)所示. 随着采样温度的上升, FAIR 检测到的软件缺陷数呈现先增加后减少的趋势. 当采样温度在 1.0–1.5 区间段内时, 模型检测到了最多数量的软件缺陷. 本部分实验结果表明, 对模型输出进行基于多项式分布的随机采样是有价值的. 随着采样温度的升高, 尽管模型生成测试程序的解析通过率有所下降, 但这有利于模型生成多样化的测试用例, 从而触发编译器中的软件缺陷.

为了验证 FAIR 检测编译器软件缺陷的有效性, 本节使用 FAIR 和 DeepSmith 分别进行 400 个进程时(5 进程×80 小时)的模糊测试, 并对检测到的软件缺陷数进行对比. 此外, 还与基于语法生成 C 语言编译器测试用例的模糊测试器 Csmith 展开对比. DeepSmith 与 FAIR 均采用随机采样, 采样温度设置为 1.0.

实验结果如图 10 所示. 在相同的模糊测试时间段内, FAIR 检测到了最多数量的编译器软件缺陷. 与 Csmith 相比, DeepSmith 和 FAIR 能够基于种子集中的高质量种子进行变异, 产生了新的测试程序, 从而具有较强的生成高质量测试用例的能力. 与 DeepSmith 相比, FAIR 能够利用种子程序的结构化信息生成测试用例, 进一步提升了检测编译器软件缺陷的能力. 最终, 经过 400 个进程时的测试后, FAIR 共检测到 8 处编译器软件缺陷, DeepSmith 检测到了 4 处, Csmith 没有检测到软件缺陷.

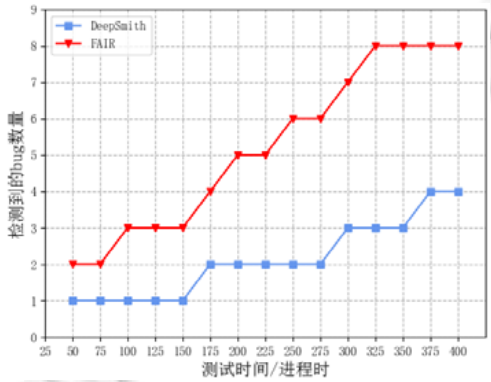


图 10　检测到的软件缺陷数对比

需要指出的是: 受模糊测试过程中变异子树选择以及采样方法等随机性因素的影响, 测试结果具有一定的随机性风险. 本部分实验在相同的实验环境下共开展了 3 次, 检测编译器软件缺陷的结果见表 1. 在去重后

的检测总数里, FAIR 检测到的 8 个编译器软件缺陷中有 6 个是 DeepSmith 没有检测到的, 其余 2 个重叠.本部分的实验结果表明, 本文所提出的方法是对现有方法的有力补充.

表 1 检测编译器软件缺陷能力对比

名称	Csmith	DeepSmith	FAIR
检测到的软件缺陷数-1	0	4	8
检测到的软件缺陷数-2	0	4	6
检测到的软件缺陷数-3	0	3	7
总数(去重)	0	4	8

5.5 RQ4: FAIR在生产级编译器中检测到的软件缺陷数

本节将 FAIR 应用于两款主流的生产级编译器的模糊测试: GCC 和 Clang/LLVM, 通过检测到的软件缺陷数验证本文所提出方法的有效性. 在不受支持的旧版本编译器中, 检测软件缺陷的价值是比较小的, 因此本文仅对在实验过程中仍受社区支持的编译器版本展开测试, 包括 GCC-9、GCC-10、GCC-11、LLVM-11 以及 LLVM-12. 此外, 还对正处于开发中的 GCC-12 也进行了模糊测试.

FAIR 成功检测出 20 个 C 语言编译器软件缺陷, 缺陷编号及影响的编译器版本见表 2, 相关信息已经提交相应的开发团队, 11 个已获得确认, 其中涉及 GCC 的软件缺陷中有 3 个被开发团队确认影响 4.9.0 (2014 年发布)以上的所有版本.

表 2 FAIR 检测到的编译器 bug

编号	受到影响的编译器版本
GCC-101171	10/11/12
GCC-101172	11/12
GCC-101196	12
GCC-101285	9/10/11/12
GCC-101313	9/10/11/12
GCC-101314	11/12
GCC-101364	9/10/11/12
GCC-101365	12
GCC-101437	12
GCC-101702	12
GCC-101858	9/10/11/12
LLVM-51154	12
LLVM-51314	11/12
LLVM-51334	11/12
LLVM-51342	11/12
LLVM-51356	11/12
LLVM-51370	12
LLVM-51372	11/12
LLVM-51373	11/12
LLVM-51374	11/12

接下来, 通过 FAIR 生成的 2 个具体的测试用例进一步阐述本文方法的有效性.

- Bug: GCC-101171.

测试用例及种子程序信息如下. 通过对语法树子树变异的方式构建新的测试程序比基于文本序列的方法在变异粒度上更加灵活, 能够轻松地在任意语法树节点级别进行变异. 本例中, FAIR 从种子程序中随机选择了 1 棵 TypeDecl 子树, 该子树对应于程序第 6 行的部分代码. 该子树变异后的代码对应于第 7 行的相应位置. 经过该变异, 测试用例成功触发了 GCC 的内部编译器错误. GCC-10–GCC-12 这 3 个版本均受该 bug 影响, 导致编译器 crash. 在向 GCC 开发团队提交了该 bug 的信息后, 该测试用例被 GCC 开发团队吸纳进了源代码的测试套件中, 测试用例编号为 101171.

```
1    extern void foo(void);
2    int x=0x1234;
3
```

```
4     int bar(·)
5     {
6   –     if (x!=(sizeof((long)0x1234)))
7   +     if (x!=(sizeof((enum t)0x1234)))
8             foo(·);
9     }
```

- Bug: GCC-101172.

该 bug 影响即将发布的 GCC-12. 测试用例及种子程序信息如下. FAIR 选择语法树的 1 棵 Decl 子树进行变异, 子树对应于种子程序的第 5 行, 该行代码声明了 *int* 型变量 *e*. 对该抽象语法树的变异过程如图 6 所示, 新生成的 Decl 子树对应程序的第 6 行. 变异后的测试程序成功触发了 GCC 的内部编译器错误. 向 GCC 开发团队提交了该 bug 的信息后, 该测试用例被 GCC 开发团队吸纳进了源代码的测试套件中, 测试用例编号为 101172.

```
1     union U {
2       int a[3];
3       struct S {
4         int a: 3;
5   –     int e;
6   +     struct this_struct d;
7       } b;
8     };
9     const union U hello={.a={1,2,3}};
10
11    void foo(·) {
12      int x=hello.b.a;
13    }
```

5.6 RQ5: 本文所提出的方法是否容易移植到其他编译器或执行引擎

FAIR 无需用户提供语法规范, 能够自动地从样例程序中学习目标程序设计语言的语言模型, 进而生成新的测试程序. 这使得 FAIR 能够比较容易地移植到其他程序设计语言的编译器或执行引擎的模糊测试. 为了验证所提方法的可移植性, 本文将 FAIR 移植到 JavaScript 代码执行引擎的模糊测试中, 共新增及修改代码 400 余行, 实现了模糊测试原型工具 FAIR-JS, 已经在 ChakraCore 引擎中检测到了两处软件缺陷(Issue-6716 和 Issue-6717), bug 信息已提交开发团队并获得确认.

实验结果表明: 本文所提出的测试用例生成方法能够快速地迁移到其他目标软件, 从而节省大量模糊测试器开发的人力与时间成本.

6 总结与展望

本文提出了一种基于前馈神经网络的编译器模糊测试用例生成方法, 并实现了一款原型工具 FAIR. 通过实证分析, FAIR 生成有效测试用例的能力优于同类型的方法, 并且检测编译器软件缺陷的能力也得到了显著提升. 将 FAIR 应用于生产级 C 语言编译器的模糊测试后, 已经发现并报告了 20 处软件缺陷. 此外, 本文所提出的方法具有较好的可移植性, 将 FAIR 迁移到 JavaScript 引擎的模糊测试仅需要增改 400 余行代码, 有效地节省了开发模糊测试器的成本. 在接下来的工作中, 我们考虑进一步完善原型工具 FAIR 的功能, 包括加入多元化的测试用例生成策略等. 当前, FAIR 仅监测目标编译器在编译测试用例时是否出现 crash 状态. 接下来的工作方向还包括在测试用例生成后, 结合差异测试方法, 为 FAIR 增加对错误编译类型软件缺陷的检测能力.

References:

[1] Chen JJ, Patra J, Pradel M, Xiong YF, Zhang HY, Hao D, Zhang L. A survey of compiler testing. ACM Computing Surveys (CSUR), 2020, 53(1): 1−36.

[2] Cummins C, Petoumenos P, Murray A, Leather, H. Compiler fuzzing through deep learning. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. 2018. 95−105.

[3] Yang XJ, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. In: Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2011. 283−294.

[4] Kolen JF, Kremer SC. Gradient flow in recurrent nets: The difficulty of learning LongTerm dependencies. In: A Field Guide to Dynamical Recurrent Networks. 2001. 237−243. [doi: 10.1109/9780470544037.ch14]

[5] Liu X, Li X, Prajapati R, Wu D. Deepfuzz: Automatic generation of syntax valid C programs for fuzz testing. In: Proc. of the AAAI Conf. on Artificial Intelligence. 2019, 33(1): 1044−1051.

[6] Le V, Afshari M, Su ZD. Compiler validation via equivalence modulo inputs. ACM SIGPLAN Notices, 2014, 49(6): 216−226.

[7] Le V, Sun CN, Su ZD. Finding deep compiler bugs via guided stochastic program mutation. ACM SIGPLAN Notices, 2015, 50(10): 386−399.

[8] Sun CN, Le V, Su ZD. Finding compiler bugs via live code mutation. In: Proc. of the 2016 ACM SIGPLAN Int'l Conf. on Object-oriented Programming, Systems, Languages, and Applications. 2016. 849−863.

[9] Chen P, Chen H. Angora: Efficient fuzzing by principled search. In: Proc. of the 2018 IEEE Symp. on Security and Privacy (SP). IEEE, 2018. 711−725.

[10] Holler C, Herzig K, Zeller A. Fuzzing with code fragments. In: Proc. of the 21st {USENIX} Security Symp. ({USENIX} Security 2012). 2012. 445−458.

[11] Wang NY, Ye YX, Liu L, Feng LZ, Bao T, Peng T. Language models based on deep learning: A review. Ruan Jian Xue Bao/ Journal of Software, 2021, 32(4): 1082−1115 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6169.htm [doi: 10.13328/j.cnki.jos.006169]

[12] Sutskever I, Martens J, Hinton GE. Generating text with recurrent neural networks. In: Proc. of the ICML. 2011.

[13] Godefroid P, Peleg H, Singh R. Learn&fuzz: Machine learning for input fuzzing. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2017. 50−59.

[14] Hochreiter S, Schmidhuber J. Long short-term memory. Neural Computation, 1997, 9(8): 1735−1780.

[15] Karpathy A, Johnson J, Fei-Fei L. Visualizing and understanding recurrent networks. arXiv preprint arXiv: 1506.02078, 2015.

[16] Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv: 1409. 0473, 2014.

[17] Luong MT, Pham H, Manning CD. Effective approaches to attention-based neural machine translation. arXiv preprint arXiv: 1508. 04025, 2015.

[18] Salton G, Ross R, Kelleher J. Attentive language models. In: Proc. of the 8th Int'l Joint Conf. on Natural Language Processing (Vol.1: Long Papers). 2017. 441−450.

[19] Al-Rfou R, Choe D, Constant N, Guo M, Jones L. Character-level language modeling with deeper self-attention. In: Proc. of the AAAI Conf. on Artificial Intelligence. 2019, 33(1): 3159−3166.

[20] Zheng W, Chen JZ, Wu XX, Chen X, Xia X. Empirical studies on deep-learning-based security bug report prediction methods. Ruan Jian Xue Bao/Journal of Software, 2020, 31(5): 1294−1313 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5954.htm [doi: 10.13328/j.cnki.jos.005954]

[21] Rush AM, Chopra S, Weston J. A neural attention model for abstractive sentence summarization. arXiv preprint arXiv: 1509.00685, 2015.

[22] Paulus R, Xiong C, Socher R. A deep reinforced model for abstractive summarization. arXiv preprint arXiv: 1705.04304, 2017.

[23] Hu X, Li G, Liu F, Jin Z. Program generation and code completion techniques based on deep learning: Literature review. Ruan Jian Xue Bao/Journal of Software, 2019, 30(5): 1206−1223 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5717. htm [doi: 10.13328/j.cnki.jos.005717]

[24] Alon U, Zilberstein M, Levy O, Yahav E. A general path-based representation for predicting program properties. ACM SIGPLAN Notices, 2018, 53(4): 404−419.

[25] Alon U, Sadaka R, Levy O, Yahav E. Structural language models of code. In: Proc. of the Int'l Conf. on Machine Learning. PMLR, 2020. 245−256.

[26] Chen ZX, Zou DQ, Li Z, Jin H. Intelligent vulnerability detection system based on abstract syntax tree. Journal of Cyber Security, 2020, 5(4): 1−13 (in Chinese with English abstract).

[27] Wang XM, Zhang T, Xin W, Hou CY. Source code defect detection based on deep learning. Journal of Beijing Institute of Technology (Natural Science Edition), 2019, 39(11): 1155−1159 (in Chinese with English abstract).

[28] Veggalam S, Rawat S, Haller I, Bos H. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In: Proc. of the European Symp. on Research in Computer Security. Cham: Springer, 2016. 581−601.

[29] Han HS, Oh DH, Cha SK. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In: Proc. of the NDSS. 2019.

[30] Lee S, Han HS, Cha SK, Son S. Montage: A neural network language model-guided javascript engine fuzzer. In: Proc. of the 29th {USENIX} Security Symp. ({USENIX} Security 2020). 2020. 2613−2630.

附中文参考文献:

[11] 王乃钰, 叶育鑫, 刘露, 凤丽洲, 包铁, 彭涛. 基于深度学习的语言模型研究进展. 软件学报, 2021, 32(4): 1082−1115. http://www.jos.org.cn/1000-9825/6169.htm [doi: 10.13328/j.cnki.jos.006169]

[20] 郑炜, 陈军正, 吴潇雪, 陈翔, 夏鑫. 基于深度学习的安全缺陷报告预测方法实证研究. 软件学报, 2020, 31(5): 1294−1313. http://www.jos.org.cn/1000-9825/5954.htm [doi: 10.13328/j.cnki.jos.005954]

[23] 胡星, 李戈, 刘芳, 金芝. 基于深度学习的程序生成与补全技术研究进展. 软件学报, 2019, 30(5): 1206−1223. http://www.jos.org.cn/1000-9825/5717.htm [doi: 10.13328/j.cnki.jos.005717]

[26] 陈肇炫, 邹德清, 李珍, 金海. 基于抽象语法树的智能化漏洞检测系统. 信息安全学报, 2020, 5(4): 1−13.

[27] 王晓萌, 张涛, 辛伟, 侯长玉. 深度学习源代码缺陷检测方法. 北京理工大学学报, 2019, 39(11): 1155−1159.

徐浩然(1993－), 男, 博士生, 主要研究领域为软件安全.

解培岱(1985－), 男, 博士, 副教授, 主要研究领域为软件安全, 恶意代码检测.

王勇军(1971－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为软件安全分析,网络威胁行为分析与检测.

范书珲(1993－), 女, 博士生, 主要研究领域为区块链安全, 软件安全.

黄志坚(1989－), 男, 博士, 工程师, 主要研究领域为软件测试, 网络安全.