软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(7):2947−2963 [doi: 10.13328/j.cnki.jos.007333] [CSTR: 32375.14.jos.007333]

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

语义可感知的灰盒编译器模糊测试*

欧先飞[1,2], 蒋炎岩[1,2], 许 畅[1,2]

[1](计算机软件新技术全国重点实验室 (南京大学), 江苏 南京 210023)

[2](南京大学 计算机学院, 江苏 南京 210023)

通信作者: 蒋炎岩, E-mail: jyy@nju.edu.cn

摘 要: 模糊测试技术在软件质量保障、软件安全测试等领域起到重要作用. 然而, 在面对编译器这样输入语义复杂的系统时, 现有的模糊测试工具由于其变异策略中缺乏对语义的感知能力, 导致生成的程序难以通过编译器前端检查. 提出了一种语义可感知的灰盒模糊测试方法, 旨在提高模糊测试工具在编译器测试领域的效能. 设计并实现了一系列可保持输入语义合法性并探索上下文多样性的变异操作符, 并针对这些操作符的特点开发了高效的选择策略. 将这些策略与传统的灰盒模糊测试工具相结合, 实现了灰盒模糊测试工具 SemaAFL. 实验结果表明, 通过应用这些变异操作符, SemaAFL 在 GCC 和 Clang 编译器上的代码覆盖率相比 AFL++和同类工具 GrayC 提高了约 14.5% 和 11.2%. 在为期一周的实验期间, SemaAFL 发现并报告了 6 个以前未被发现的 GCC 和 Clang 缺陷.

关键词: 编译器测试; 语义可感知的模糊测试; 灰盒模糊测试

中图法分类号: TP311

中文引用格式: 欧先飞, 蒋炎岩, 许畅. 语义可感知的灰盒编译器模糊测试. 软件学报, 2025, 36(7): 2947–2963. http://www.jos.org.cn/1000-9825/7333.htm

英文引用格式: Ou XF, Jiang YY, Xu C. Semantic Aware Greybox Compiler Fuzz Testing. Ruan Jian Xue Bao/Journal of Software, 2025, 36(7): 2947–2963 (in Chinese). http://www.jos.org.cn/1000-9825/7333.htm

Semantic Aware Greybox Compiler Fuzz Testing

OU Xian-Fei[1,2], JIANG Yan-Yan[1,2], XU Chang[1,2]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](School of Computer Science, Nanjing University, Nanjing 210023, China)

Abstract: Fuzz testing techniques play a significant role in software quality assurance and software security testing. However, when dealing with systems like compilers that have complex input semantics, existing fuzz testing tools often struggle as a lack of semantic awareness in their mutation strategies leads to the generated programs failing to pass compiler frontend checks. This study proposes a semantically-aware greybox fuzz testing method, aiming at enhancing the efficiency of fuzz testing tools in the domain of compiler testing. It designs and implements a series of mutation operators that can maintain input semantic validity and explore contextual diversity, and develops efficient selection strategies according to the characteristics of these operators. The greybox fuzz testing tool SemaAFL is developed by integrating these strategies with traditional greybox fuzz testing tools. Experimental results indicate that by applying these mutation operators, SemaAFL achieves approximately 14.5% and 11.2% higher code coverage on GCC and Clang compilers compared to AFL++ and similar tools like GrayC. During a week-long experimental period, six previously unknown bugs in GCC and Clang are discovered and reported by SemaAFL.

Key words: compiler testing; semantic aware fuzz testing; greybox fuzz testing

---

* 基金项目: 国家重点研发计划 (2022YFB4501801); 国家自然科学基金 (62025202, 62272218); 江苏省前沿引领技术基础研究专项 (BK20202001)
本文由“新兴软件与系统的可信赖性与安全”专题特约编辑向剑文教授、陈厅教授、杨珉教授、周俊伟教授推荐.
收稿时间: 2024-08-18; 修改时间: 2024-10-15; 采用时间: 2024-11-25; jos 在线出版时间: 2024-12-10
CNKI 网络首发时间: 2025-04-21

近年来, 模糊测试领域取得了重要的技术进展[1−10]. 通过自动为待测软件生成大量测试用例, 模糊测试工具为大量软件发现了成千上万的关键性缺陷. 这些自动暴露的缺陷能够得到开发者的及时修复, 从而显著提高基础软件生态的质量和可靠性. 模糊测试技术现已被包括微软[11,12]和谷歌[13−15]在内的重要软件公司用于日夜不间断的安全测试和软件质量保障.

以代码覆盖率为导向的灰盒模糊测试技术是目前为止最成功的模糊测试技术之一[1], 其引起了学术界的广泛关注并在工业界广泛应用[16−19]. AFL++[20]作为最先进的灰盒模糊测试工具代表, 已在 Web 浏览器 (例如 Firefox、Internet Explorer)、网络工具 (例如 tcpdump、Wireshark)、图像处理器 (例如 ImageMagick、libtiff)、系统库 (如 OpenSSH、PCRE)、数据库[21]等软件系统中发现了大量漏洞. 如图 1 所示, 覆盖率导向的模糊测试技术以遗传算法为基础, 通过从种子池中抽取种子输入并生成变异后代的方式, 实现提升代码覆盖率的优化目标. 类似于生物种群的优胜劣汰, 新生成的优秀变异体测试输入 (探索到被测软件新分支) 会被保存在种子池中, 进入下一轮迭代搜索.

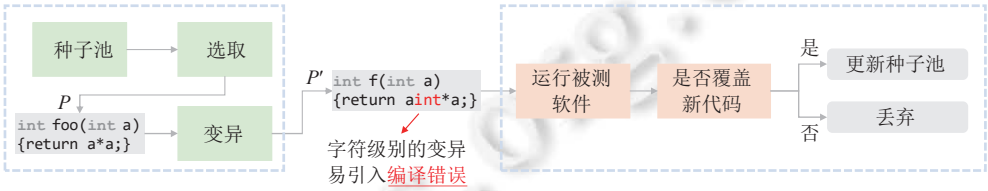

图 1 传统灰盒模糊测试工具的工作流程

然而灰盒模糊测试却难以有效应用于编译器测试, 特别是对于静态类型语言的编译器[1,22], 如 C/C++编译器. 如图 1 所示, 灰盒模糊测试中广泛采用了字符串级别[1,20]的变异操作符生成变异后代. 然而, 这些操作符无法理解输入程序的语义结构, 在随机位置变异极易破坏程序的合法性, 致使产生的绝大部分后代变异体均无法通过编译器前端检查 (语法检查、类型检查和语义检查等), 无法有效探索中间代码生成、中间代码优化、二进制代码生成等编译器中/后端功能模块的功能和行为, 覆盖率信息亦无法在此情形下提供搜索导向.

目前虽有一些工作致力于为灰盒模糊测试引入对结构化输入的感知能力, 如 Gramatron[23]、Superion[1]. 但到目前为止, 这些工作也仅支持语法级别的输入结构感知能力, 其测试能力主要在 XML 解析器等仅关注输入语法合法性的被测软件. 这些技术无法应对编译器等复杂软件系统对输入语义级别的合法性约束 (如程序的类型正确性、作用域正确性等), 易生成无法通过编译器前端检查的测试输入. 因此, 我们需要开发一种新的模糊测试方法, 能够理解并正确处理输入的语义结构, 以满足更复杂软件系统 (如编译器) 的测试需求. 这种方法不仅应提高模糊测试的有效性, 还应尽可能确保测试过程中对输入的语义正确性保持不变, 从而提升发现深层次、语义相关错误的能力.

本文尝试为灰盒模糊测试引入感知语义的能力, 即利用语义分析结果进行有导向的输入变异, 以弥补现有灰盒模糊测试在编译器测试领域的不足. 本文针对传统变异操作符不建模测试输入语义的局限, 将它们替换为基于语法树节点和语义分析结果进行变异的操作符, 从而实现在变换程序时大幅保持程序的语义合法性以及编译器深层次代码的高效探索. 尽管已有工作专门实现了基于语义信息的变异操作工具[22], 但为实际工业级灰盒模糊测试实现引入此类变异操作符, 依旧面临以下挑战.

挑战 1: 现有利用语义信息进行程序变异的工具在可用性和可靠性方面仍有不足, 难以应用到灰盒模糊测试. 在可用性方面, 现有最先进的语义级别程序变异工具 GrayC[22], 其支持的语义操作符数量和种类都相当有限, 在实际应用中超过 80% 的程序无法被其成功变异, 使图 1 中的模糊测试主循环长期处于缺少新变异输入的低效空转状态. 在可靠性方面, GrayC 对丰富语法特性和边界情况的处理不足, 在实践中很容易触发工具自身崩溃, 与工业级模糊测试工具实现不匹配.

为应对这一挑战, 本文在可用性和可靠性两个方面均进行了改进. 在可用性方面, 我们调研了 GrayC 无法变异的程序类型, 并设计新的变异操作符以扩大覆盖范围. 在可靠性方面, 我们通过大量人工测试和修复, 将变异操

作符的崩溃频率降至 GrayC 的 2% 以下, 并采用进程隔离机制将变异操作从 AFL++主程序中分离, 以防止模糊测试过程中断. 这些改进有效填补了基于语义信息的变异操作符与灰盒模糊测试之间的技术空白, 显著提升了灰盒模糊测试工具在复杂的编译器系统上的适用性和稳定性.

挑战 2: 现有灰盒模糊测试工具中的变异操作符选择策略无法匹配语义感知的变异操作符. 当前模糊测试工具中, 变异直接在字符级别进行, 变异总能获得新的输入变体, 因此随机选择即可满足测试需求. 然而, 语义级别的变异操作符通常带有使用前置条件 (如修改分支语句的前提是程序存在分支语句), 随机选择将导致无效的变异尝试. 此外, 基于语义信息的变异操作符可进行变异的变异点通常较少 (如分支语句), 同一操作符应用在同一个程序上多次后极易生成重复的程序. 这两个问题使灰盒模糊测试工具常用的随机策略效率有数量级的提升空间.

为应对这一挑战, 本文设计了一套动态权重匹配算法, 从随机尝试操作符和输入的组合开始, 随着模糊测试的进展, 基于操作成功与否的历史记录动态调整输入程序和操作符组合的权重, 并在生成变异后代时给予高权重组合以更高的选择概率.

综上所述, 本文主要贡献如下.

(1) 为灰盒模糊测试引入了基于语义信息的变异操作符, 有效填补了灰盒模糊测试在处理需要高度语义化输入的软件系统中的应用空白.

(2) 为知名模糊测试工具 AFL++实现基于语义信息的变异操作, 并将 AFL++扩展为工具 SemaAFL. 相比 AFL++和 GrayC, SemaAFL 在 GCC 和 Clang 上的代码覆盖率分别提高了 14.5% 和 11.2%. 此外, SemaAFL 在最新版本的 GCC-14 和 Clang-18 上发现并上报了 6 个缺陷, 均已获得开发者确认.

本文第 1 节阐述基于语义信息的变异操作符相对于传统灰盒模糊测试中字符串级变异操作符的优势. 第 2 节详细介绍基于语义信息的变异操作符的设计理念和具体实现方案. 第 3 节重点探讨变异操作符的选择策略. 第 4 节展示实验结果, 其中包括我们的工具 SemaAFL 成功发现的 6 个已获 GCC/Clang 开发团队确认的新漏洞. 第 5 节分析当前工具的局限性, 并探讨未来向多语言支持扩展的可能性. 第 6 节详细介绍模糊测试领域的相关研究工作.

1 基于语义信息的变异操作符的优势

在灰盒模糊测试中, 变异操作通过修改测试输入来探索待测程序的执行路径. 对于编译器这一特殊测试对象, 由于大部分编译器代码只在完成语法和语义检查后才执行, 因此保持输入的语义合法性是探索编译器不同行为的关键. 这样可以触发编译器的后端逻辑, 避免在前端就被基本的语法语义检查拦截.

传统灰盒模糊测试中的字符级变异操作符在保持输入程序合法性方面存在局限. 这些操作符会盲目地插入或删除字符串, 破坏程序的词法结构, 更难以维持语义合法性. 因此, 生成的程序在探索编译器行为上效率低下, 通常只能触及浅层的语法语义检查逻辑. 尽管一些工作如 Gramatron[23]和 Superion[1]致力于为灰盒模糊测试引入结构化输入感知能力, 但这些方法仅支持语法级别的感知, 对程序的语义结构仍缺乏理解.

图 2 中①展示了 AFL++基于字典的字符级变异操作. 该操作从预设字典中选取 char 字符串, 将其插入第 6 行变量 i 定义尾部. 生成的程序因 char 和 for 直接相连而在语法上不合法, 在编译器前端的语法检查阶段被拒绝. 实践中, 此操作未能探索到新的编译器分支.

图 2 中②展示了语法级变异操作. 该操作以语法树为单位, 将第 8 行表达式节点 p-&q 替换为 a. 虽然生成的程序通过语法检查, 但因尝试将数组类型和指针类型直接相加而无法通过语义检查. 编译器处理该表达式时会报错“检测到非法的加法操作数”. 实践中, 此操作同样未能探索新的编译器分支.

图 2 中③展示了本文实现的“降低数组维度”变异操作符效果. 该操作符选中数组 a, 将其定义降为与数组元素相同的简单指针类型. 得益于对程序语义的感知能力, 该操作在修改 a 的定义后, 还相应调整了所有对 a 元素的访问为直接对 a 的访问. 这不仅保证了变异后程序的语义合法性, 还触发了循环展开优化, 成功探索到新的编译器分支.

如图 2 所示, 利用语义级别信息的变异操作符具有多重优势. 首先, 它能有效感知和维护程序变异前后的语义

合法性, 确保变异后的程序顺利通过编译器的前端检查; 其次, 它能诱导程序展现不同的优化模式, 从而触发编译器的多样化优化行为. 相比之下, 字符串级别和语法级别的变异操作在实践中往往生成大量语义不合法的程序, 这些程序通常在编译器运行初期就被前端检查拒绝, 难以深入探索编译器的行为.

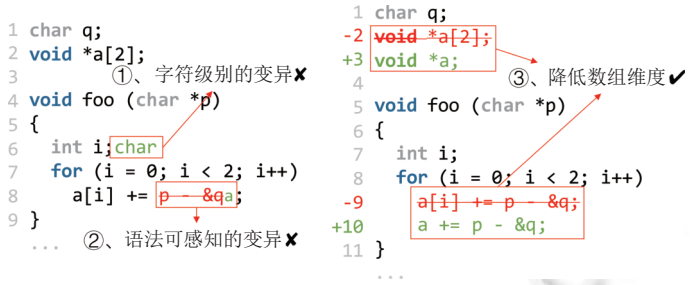

图 2 对 GCC 测试集中 pr21173 进行字符级别、语法可感知和语义可感知的变异效果对比

2 语义可感知灰盒模糊测试工具的设计实现与集成适配

图 3 展示了如何为灰盒模糊测试工具引入利用语义信息的能力. 相较于图 1 的模糊测试流程, 我们将原本字符串级别的变异模块替换为语义级别的变异模块. 为了实现这一替换, 我们需要设计实现一系列能够覆盖各种场景的基于语义信息的变异操作符, 并且, 将这些变异操作符集成适配进图 3 所示的变异模块.

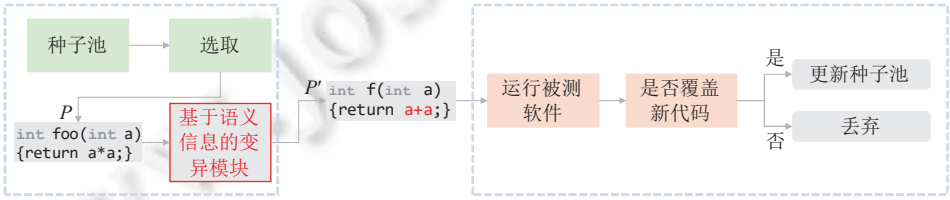

图 3 语义可感知的灰盒模糊工作流程

需要注意的是, 目前最先进的基于语义信息的程序变异工具 GrayC[22]仅实现了 5 个语义级别的变异操作符, 其在实践中只有约 17% 的情况下可以变异出不一样的程序. 且 GrayC[22]所实现的操作符实践中会有大量的崩溃, 在可用性和可靠性上均有明显的不足, 难以集成适配到灰盒模糊测试工具中. 因此, 我们选择自行设计实现变异操作符.

2.1 设计实现与可用性增强

如图 4 所示, 我们会同时借助人工和 AI 来快速头脑风暴, 生成一个初始的变异操作符集合. 我们将变异操作符视为一个三元组<名称, 功能描述, 代码实现>, 这里生成的初始变异操作符是指名称和功能描述部分. 随后, 对这些操作符的功能描述进行人工分析处理. 一些语义重复的, 诸如“消除未使用变量”和“移除未使用变量”, 我们会进行去重. 再之后, 我们会基于自身对编译器的专业知识, 筛选更利于探索编译器行为的操作符, 并进行人工实现. 这一步结束后, 我们共计获得了 43 个变异操作符.

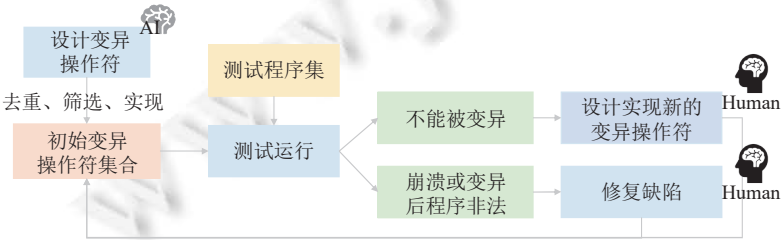

图 4 基于语义信息的变异操作符的设计和实现流程

表 1 展示了这 43 个初始变异操作符中的一小部分, 这些操作符涵盖了表达式、语句、函数、变量等多个方面, 并且其中包含一些相当复杂的操作, 如表 1 所示的“扁平化高维数组”. 不过即便如此, 在我们所搜集的测试程序集中, 依旧有不少的程序无法被所有这些操作符变异, 这也驱使我们去添加更多的变异操作符来覆盖这些无法被变异的程序.

表 1 本文实现的部分变异操作符

变异操作符	功能描述
修改表达式操作符	修改表达式的操作符, 并保证语义合法, 如a+b改为a–b
简单内联	将被调用函数内联至掉用处
分支转多路分支	将if分支语句转化为switch语句
翻转内联修饰符	若选定函数为内联函数, 则移除内联修饰符, 反之添加内联修饰符
表达式拷贝	对在同一个作用域内类型相同的两个表达式, 将其中一个替换为另一个
扁平化高维数组	将多维数组a[M][N][...]降为一维数组, 并相应的修改所有引用
分支条件翻转	将if分支语句的判断表达式取反

2.1.1 可用性增强

对于我们所设计和实现的操作符而言, 可用性问题主要包含两个方面, 一是有一部分测试程序无法被任何一个我们的操作符所变异, 另一个是操作符由于实现当中的缺陷, 会导致输出程序非法亦或是操作符自身崩溃. 针对这两个问题, 如图 4 所示, 我们会采用如下的解决方案来缓解.

(1) 无法被变异的程序. 在获得表 1 所示的初始的操作符集合 (包括名称, 功能描述和代码实现) 之后, 如图 4 所示, 我们会在搜集来的测试程序上进行大量的测试运行. 如果某个程序未能被所有操作符成功变异, 我们会人工分析该程序, 设计出可以覆盖该程序的变异操作符. 该过程会持续迭代, 直至所有测试程序都能被至少一个操作符变异. 这一过程迭代结束后, 共计获得了 72 个变异操作符.

(2) 变异操作符的缺陷. 为了降低操作符实现中的缺陷带来的影响, 我们在测试运行的时候也会搜集分析缺陷. 如图 4 所示, 如果测试过程中, 在某个变异操作符出现了崩溃或者变异出了不可编译的程序, 我们会人工介入并修复这个变异操作符.

2.2 集成适配与可靠性增强

为了将这些操作符和模糊测试相结合, 我们还需要将这些操作符集成进模糊测试工具中. 这里我们选择了目前最先进的灰盒模糊测试工具 AFL++[20]. AFL++为二次开发提供了一套插件机制, 我们可以将已实现的变异操作符按 AFL++的接口描述打包成动态链接库, 然后将其作为基于语义信息的变异模块交由 AFL++加载运行. 具体来说, 我们需要实现如下接口.

```
1. // 自定义的变异操作模块
2. size_t afl_custom_havoc_mutation(my_mutator_t *data, …);
```

如算法 1 所示, 在具体实现过程中, 我们在该函数内部会先通过 select 函数选出合适的种子程序 p 和变异操作符 o, 随后我们通过 request_mutation 函数将变异操作符 o 应用于程序 p 获得 p', 之后通过 update 函数更新一下 select 函数的选择权重, 最后将变异后的程序 p' 返回给 AFL++进行后续的模糊测试, 包括覆盖率更新、超时检查、崩溃检查以及种子程序池更新等.

算法 1. 自定义变异操作.

输出: 变异后的程序 p'.

```
1. function afl_custom_havoc_mutation() {
2.     p,o ← select()
3.     p' ← request_mutation(p,o)
```

4. update(p, o, p')
5. return p'
6. }

2.2.1 可靠性增强

由于基于语义信息的变异操作符实现起来相对字符级别操作复杂很多, 正确性很难得到完全保证, 实践中总会有一定比例的崩溃出现. 尽管在图 4 中, 我们已经通过持续的测试运行和缺陷修复来尽可能地减少崩溃, 却始终无法完全避免. 又由于通过自定义接口实现的变异操作符模块直接在 AFL++的进程空间中被调用. 因此一旦崩溃发生, 便会导致 AFL++停止运行, 这对于一个需要长时间执行的模糊测试工具而言是不可接受的.

针对经常出现的崩溃中断模糊测试工具这一可靠性问题, 我们使用了进程隔离的方式. 在调用自定义接口时克隆出一个子进程, 将所有操作放在子进程中完成, 并将结果通过进程间通信返回给父进程, 则基于语义信息的变异操作符的崩溃都会被隔离在子进程内, 从而大幅增加了基于语义信息的变异操作符的可用性.

我们将进程隔离实现在算法 1 的第 3 行. 具体来说, 我们实现了一个变异服务器, 专门处理程序变异请求. 算法 1 在第 3 行通过 request_mutation 向这个变异服务器发送请求. 如果服务器发生崩溃或卡死, 无法及时响应请求, request_mutation 会重启这个服务器并返回 $\perp$ 用以区分.

如算法 2 所示, 变异服务器循环处理来自客户端的程序变异请求. 服务器的工作流程开始于等待客户端通过 S_{req} 信号量发起的变异请求. 一旦接收到请求, 服务器从共享内存中提取待变异程序 p 和变异操作符 o. 随后, 服务器执行变异操作, 该操作可能修改程序 p 的结构或行为, 根据变异结果, 服务器将变异后的程序 p' 写回共享内存. 完成这些操作后, 服务器通过 S_{res} 信号量通知客户端, 变异处理已完成.

算法 2. 变异服务器处理变异请求.

全局: 共享内存 M_s, 请求信号量 S_{req}, 回复信号量 S_{res};
输出: 变异后的程序 p'.
1. function mutation_server() {
2. while (¬ terminated()) {
3. sem_wait(S_{req})
4. $p, o \leftarrow$ fetch_request(M_s)
5. $p' \leftarrow$ mutate(p, o) // 这一步会出现崩溃
6. write_response(M_s, p')
7. sem_signal(S_{res})
8. }
9. }

相对应地, 变异客户端的职责是构建变异请求, 并从服务器获取处理结果. 如算法 3 所示, 客户端首先将待变异的程序 p 和变异操作符 o 写入共享内存 M_s 中, 然后通过 S_{req} 信号量通知服务器发起变异请求. 在通知服务器后, 客户端等待服务器的响应. 这里使用了带超时机制的信号量等待函数 timed_sem_wait, 如果在规定的超时时间 T_{tmout} 内没有收到服务器的响应, 则认为变异处理失败. 此时, 客户端需要重置信号量 S_{req} 和 S_{res} 的状态, 并重新启动变异服务器, 以确保系统能够继续正常工作. 如果客户端成功接收到服务器的响应, 则从共享内存中读取变异后的程序 p' 并将其返回.

值得一提的是, 上述的变异客户端/服务器框架可以兼容多进程进行并行处理. 具体而言, 可以创建多个变异服务器实例, 每个实例独立处理一部分变异请求. 通过并行处理, 可以充分利用多核 CPU 的计算能力, 从而缩短变异操作的总体耗时. 当然, 引入并行处理也会带来一些新的问题, 例如需要对共享资源 (如共享内存) 的访问进行

同步和互斥控制, 以避免竞态条件的发生. 此外, 还需要对负载均衡进行合理设计, 以确保各个服务器实例的工作量基本均衡, 从而达到最优的整体性能.

算法 3. 客户端请求变异操作.

全局: 共享内存 M_s, 请求信号量 S_{req}, 回复信号量 S_{res};
超参: 最大等待时间 T_{tmout};
输入: 待变异的程序 p, 变异操作符 o;
输出: 变异后的程序 p'.

```
1. function request_mutation(p,o) {
2.     write_request(M_s,p,o)
3.     sem_signal(S_req)
4.     if (¬ timed_sem_wait(S_res,T_tmout)) {
5.         /* reset semaphores S_req and S_res */
6.         /* restart mutation server */
7.         return ⊥
8.     }
9.     p' ← fetch_response(M_s)
10.    return p'
11. }
```

3 基于语义信息的操作符选择策略

灰盒模糊测试工具内部所实现的变异操作符选择策略往往是完全随机的. 由于其所支持的都是字符级别的变异操作符, 如翻转位、交换字节, 这些变异操作符直接在最底层的字符流上进行操作, 因此无论如何被选择, 其总能以接近 100% 的概率变异成功并生成新的与输入不一样的程序. 但是同样的策略在语义级别的变异操作符上却会面临不小的困难.

挑战 1. 语义级别的变异操作符需要苛刻的前置语义条件. 灰盒模糊测试工具所实现的变异操作符直接在字符级别进行, 其能对任意输入进行变异, 但语义级别的变异操作往往需要输入满足很强的限定条件之后才能应用成功 (如“分支表达式反转”这样的语义级别的变异操作符需要输入中至少包含一个分支语句才能进行变异). 随机给定一个程序, 往往只有很少一部分语义级别的变异操作符可以被应用. 这使得灰盒模糊测试工具惯用的随机尝试策略对于语义级别的操作符效率非常低下, 随机选择往往会面临大量的失败变异.

挑战 2. 语义级别的变异操作符多次调用后容易生成重复的程序. 对于字符级别的变异操作符而言, 如交换字节, 其在实践中几乎总能生成与变异前不一样的程序. 这里可以从概率层面做个简单的计算, 以交换字节为例, 想要出现重复的程序, 其概率低至 $1/\text{size}(p)^2$ ($\text{size}(p)$ 为输入程序的大小, 其通常不小于 100), 这样的概率已经非常之低, 可以忽略不计. 而对于语义级别的变异操作符而言, 给定一个程序可选的变异点经常很少. 以移除语句为例, 其可选的变异点与给定程序内所包含的语句数量一致, 如果选定的种子程序只有一条语句, 那么移除语句这一变异操作符在变异一次之后即无法再贡献新的不一样的程序.

在这两个挑战下, 我们无法直接使用灰盒模糊测试工具既有的简单随机变异操作符选择策略. 因此, 我们开发了一套动态权重选择算法, 这套算法会根据观测到的事件给程序 p 和变异操作符 o 动态调整选择权重. 在选择操作符和种子程序的时候, 会根据权重去加权采样出一组<程序, 操作符>对进行变异. 由于我们的算法同时覆盖了操作符选择和种子程序选择, 因此, 在实现中, 我们会同时将 AFL++的输入选择策略和操作符选择策略替换为我们所实现的基于动态权重的选择策略. 同样的, 由于我们替换掉了 AFL++自身的选择策略, 这会引入新的挑战.

挑战 3. 覆盖新分支时的权重奖励. 在灰盒模糊测试工具中, 分支覆盖信息用以指导灰盒模糊测试如何选择种子程序. 由于我们替换了灰盒模糊测试工具中的选择策略, 因此我们的算法需要承担起这部分功能. 更具体地, 我们需要在当前变异操作符 o 应用在程序 p 上所生成的程序 p' 探索到新的分支时, 进行权重调整, 以奖励这样的行为.

3.1 动态权重选择算法

给定种子程序池 P 和变异操作符集合 O, 该算法的重心是维护一个权重映射 $W=\{(p,o)\mapsto w|(p,o)\in P\times O\}$. 在模糊测试进行到选择种子程序和变异操作符这一阶段的时候, 通过 W 加权采样出一组 (p,o), 其中 $p\in P$, $o\in O$, 并将操作符 o 应用于 p 获得新的程序 p'.

我们将选择操作符和种子程序表述为算法 4 中的 select 函数 (对应前文算法 1 的第 2 行所调用的函数), 给定动态维护的权重映射 $W=\{(p,o)\mapsto w|(p,o)\in P\times O\}$, 在进行选择时, 选择算法根据当前权重采样一个 (p,o) 组合 (算法 4 第 6 行) 返回给算法 1 进行变异.

算法 4. 寻找适合的基于语义信息的变异操作符和种子程序.

全局: 种子程序池 P, 变异操作符集合 O, 权重映射 $W:P\times O\mapsto weight$;
输出: (p,o) 一组可以进行变异操作的操作符 o 和程序 p.

```
1. function select() {
2.     E ← {(p,o)|W[e] =⊥}
3.     if (random(0,1) < |E| / |P×O|) {
4.         return randomly_select(E)
5.     } else {
6.         return weighted_sample(P×O − E, W)
7.     }
8. }
```

权重映射 W 初始化所有权重为 $\perp$, 表示当前没有历史信息可供参考, 所有权重均未知. 这样的设计是为了在整个模糊测试刚开始的阶段, 以及新的程序被加入的时候, 区分出没有变异历史记录的 (p,o) 组合. 对于这部分组合, 我们以一定概率进行完全随机采样 (算法 4 第 4 行), 该概率会随着权重映射 W 中有记录的项数占比提升而降低. 对应算法 4 第 3 行 $|E|$ 越大, $\text{random}(0,1)<|E|/|P\times O|$ 的概率越小, 进行第 4 行 $\text{randomly\_select}(E)$ 的概率也越小. 且当 $|E|=|P\times O|$ 时, $\text{random}(0,1)<|E|/|P\times O|$ 的概率为 0, 意味着当所有组合都有记录时, 不再进行完全随机采样.

3.1.1 权重更新策略

前文描述了在给定权重映射 W 的情况下, 如何进行采样以选出合适的程序 p 和变异操作符 o. 而对于权重映射 W 如何进行更新, 这里会详细展开说明. 具体来说, 给定变异操作符 o, 种子程序 p, 变异后的程序 p' 以及权重映射 W, 我们会就表 2 所示的事件对权重映射 W 进行调整.

表 2 动态权重算法关注的事件和会采取的策略

事件	采取动作
用 o 变异 p 失败或生成重复程序	给组合 (p,o) 降低权重
用 o 变异 p 已多次失败且权重低于阈值	将组合 (p,o) 从权重映射 W 中移除
成功变异但未覆盖新分支	给组合 (p,o) 增加权重
p' 覆盖了新的分支	给组合 (p,o) 增加权重, 并对所有 $o\in O$ 将 (p',o) 加入权重映射 W

算法 5 描述了如何基于表 2 所示的事件对权重映射 W 进行调整更新, 对应权重更新函数记为 update, 该函数同样也对应前文算法 1 第 4 行所调用的函数. 该函数用以在操作符 o 被用于变异程序 p 之后, 根据变异的结果和

分支覆盖的信息来更新组合 (p,o) 的权重.

算法 5. 更新变异操作符权重映射.

全局: 种子程序池 P, 变异操作符集合 O, 权重映射 W, 变异历史记录 $H:[(p,o,p')^*]$;
超参: N_d, $N_\perp \in \mathbb{N}$, $w_t, w_s, w_c \in \mathbb{R}$;
输入: 被选中的程序 p, 被选中的变异操作符 o, 和变异后的程序 p'.

```
1. function update(p,o,p') {
2.   if (W[(p,o)] =⊥) {
3.     W[(p,o)] ← 1
4.   }
5.   if (p' =⊥ ∨count((∗,∗,p'),H) > N_d) {          // 用 o 变异 p 失败或生成重复程序
6.     W[(p,o)] ← max(1,W[(p,o)] − w_s)
7.     if (count((p,o,⊥),H) > N_⊥ ∧ W[(p,o)] ⩽ w_t) { // 用 o 变异 p 已多次失败且权重低于阈值
8.       W ← W − {(p,o)}
9.     }
10.  } else if (||brCover(p') − ∪_{q∈P} brCover(q)|| > 0) {// p' 覆盖了新的分支
11.    P ← P ∪ {p'}
12.    W ← W ∪ {(p',o) ↦ ⊥ | o ∈ O}
13.    W[(p,o)] ← W[(p,o)] + w_c
14.  } else {
15.    W[(p,o)] ← W[(p,o)] + w_s                    // 成功变异但未覆盖新分支
16.  }
17. }
```

首先, 我们在第 2 行检测组合 (p,o) 的权重是否为 $\perp$, 如果是, 则将权重 $W[(p,o)]$ 初始化为 1. 接着, 如果用操作符 o 变异程序 p 失败或生成了重复的程序 (第 5 行), 我们会在第 6 行给组合 (p,o) 的权重减去一个惩罚值 w_s. 如果对于同一个组合 (p,o), 变异失败的次数超过了阈值 $N_\perp$, 并且当前权重低于另一个阈值 w_t(第 7 行), 我们会在第 8 行将该组合从权重映射 W 中完全移除.

如果变异成功, 并且生成的程序 p' 覆盖了新的分支 (第 10 行), 我们不仅会在第 13 行给组合 (p,o) 的权重加上一个更大的奖励值 w_c, 还会在第 11、12 行将 p' 加入到种子程序池 P 中, 并对所有的操作符 o, 将新的组合 (p',o) 加入到权重映射 W 中, 以探索从这个新的有价值的种子程序开始的变异可能性. 如果变异成功, 但生成的程序 p' 并未覆盖任何新的分支 (第 14 行), 我们会在第 15 行给组合 (p,o) 的权重加上一个奖励值 w_s, 以鼓励该组合的进一步尝试.

通过这种动态调整权重的方式, 我们可以不断地根据变异的反馈来优化种子程序和变异操作符的选择, 将更多的计算资源投入到更有可能产生有价值变异的组合上, 提高模糊测试的效率.

4 实验评估

我们将前文所述的语义可感知的灰盒模糊测试实现为工具 SemaAFL. 为了全面评估 SemaAFL 的性能表现, 我们设计了一系列实验, 并选取了具有代表性的测试工具 AFL++和 GrayC 作为性能对比的基准. 这些实验会横向对比 SemaAFL 与其他工具在可用性/可靠性、代码覆盖率方面的优劣. 通过这些实验结果, 我们可以判断感知语义的策略对提升模糊测试性能的贡献, 并明确 SemaAFL 的优势和不足. 具体而言, 本节的评估会围绕这 3 个问题展开.

(1) SemaAFL 在可靠性和可用性方面与其他测试工具的横向对比.

(2) SemaAFL 在编译器代码覆盖率方面与其他测试工具的横向对比.

(3) SemaAFL 在探测现代编译器中的缺陷方面表现如何.

综上所述, 本节的实验评估将从可靠性和可用性、代码覆盖率和实际找缺陷的能力 3 个层面, 全方位地考察 SemaAFL 的优势和局限性, 为后续工作的改进提供重要参考.

4.1 实验设置

为了回答第 1 个问题, 我们整理了一个包含 1 800 个测试程序的测试集, 涵盖了 C 语言的各类特性. 在这个集合上运行 SemaAFL、AFL++和 GrayC 这 3 个工具共计 5 遍, 每遍持续 24 h, 合计 15 个 CPU 日. 运行过程中会实时记录工具发生的崩溃和无效变异的数量及比例, 通过这些指标可以比较全面地评估 3 个工具的可靠性和可用性. 崩溃频率反映了工具的可靠性, 频繁崩溃说明工具本身可能存在缺陷且对测试用例的处理能力有限. 而无效变异程序的比例则反映了工具对具备不同特性的程序的适应性, 比例越低说明工具的通用性越好. 通过横向对比 3 个工具在这两项指标上的表现, 可以比较客观地评判 SemaAFL 的可靠性和可用性水平.

为了回答第 2 个问题, 我们用同样的 1 800 个 C 程序作为种子程序池, 对两个目前最广为使用的 C/C++编译器 GCC 和 Clang 进行模糊测试. 我们会在 GCC 和 Clang 最新的已发布版本上 (也就是 GCC-14 和 Clang-18) 测量 SemaAFL、AFL++和 GrayC 对编译器的实时分支覆盖率. 对所有的 3 个测试工具, 我们会在两个编译器上运行 5 遍, 每遍持续 24 h, 全部共计消耗 30 个 CPU 日.

为了回答第 3 个问题, 我们选择两个目前最广为使用的 C/C++编译器 GCC 和 Clang 作为测试对象, 并在它们最新发布的稳定版本 (即 GCC-14 和 Clang-18) 上运行 SemaAFL 进行为期一周的模糊测试. 在测试结束后, 我们会汇总 SemaAFL 报告的崩溃信息, 并人工分析缺陷成因并进行去重. 对于确认的触发编译器缺陷的测试用例, 我们会整理成符合 GCC 和 Clang 社区规范的缺陷报告, 提交给相应的开发者进行确认和修复.

上述所有实验都在运行 Ubuntu 22.04 的 DELL PowerEdge R6515 服务器上进行. 该服务器配备了一个 64 核 (128 线程) 的 AMD EPYC 7713P 处理器和 128 GiB 内存.

4.2 可用性和可靠性对比

4.2.1 可用性对比

如表 3 所示, SemaAFL 在 5 次 24 h 的实验中, 平均进行了超过 370 万次变异, 这一速度远超过同为基于语义信息的变异工具 GrayC, 并相当接近 AFL++. 这是因为 AFL++的所有变异均在字符级别进行, 无需进行复杂的分词和语法语义解析, 从而节省了大量时间. 在无效输出的对比上, SemaAFL 平均有 220 万次变异未能产生有效输出, 约占总变异次数的 58%. 这一比例在很大程度上反映了语义级别变异操作符苛刻的调用条件. 尽管 SemaAFL 的无效变异比例平均达到 58%, 但相比于同为语义级别变异工具的 GrayC 的 83%, SemaAFL 仍领先了接近 25%. 这得益于我们设计的操作符高效选择算法. 同时从表 3 可以看出, SemaAFL 无效输出的主要来源是操作符无输出, 占比为 92.4%. 而 GrayC 最大的来源却是输出与输入相同, 占比为 89.9%, 这是因为我们的实现在变异失败时不会有任何输出, 而 GrayC 在变异失败时会将输入原封不动输出. 对于 AFL++而言, 由于字符级别的变异天然就较难失败 (包括无输出、输出与输入相同等), 在我们所有的 5 次 24 h 实验中, 我们未曾观测到 AFL++的无效变异.

表 3 无效变异频率对比

工具	总变异次数	无输出	输出和输入相同	输出和历史输出相同	总无效变异次数	无效比例 (%)
SemaAFL	3 763 299	2 041 018	102 031	65 514	2 208 563	58.68
AFL++	4 512 455	0	0	0	0	0
GrayC	716 227	0	539 986	60 589	600 575	83.85

在变异生成的可编译程序数量上, 如表 4 所示, 在 24 h 的测试周期内, SemaAFL 远超 AFL++ (领先 12 倍) 和 GrayC (领先 88%), 这很大一部分是得益于与灰盒模糊测试工具的结合. 在变异生成的可编译程序比例上, SemaAFL 以高达 77% 的比例遥遥领先 AFL++, 且相当接近 GrayC. 这是因为 GrayC 仅设计实现了 5 个语义级别

的变异操作符, 其测试修复的工作量远低于我们的 72 个操作符. 如果投入更多的测试运行时间, 我们相信 SemaAFL 的可编译程序比例可以比肩甚至超越 GrayC. 此外, AFL++仅能维持约 4% 的可编译程序比例, 这是由于字符级别变异对程序结构的不可知性. 在表 4 中, 输出程序数量与变异次数正相关, 同时会受崩溃次数、havoc 机制的影响, 在这里不存在严格的等式关系.

表 4　输出程序和可编译程序吞吐量对比

工具	输出程序数量	可编译程序数量	可编译比例 (%)
SemaAFL	1 717 460	1 332 405	77.58
AFL++	2 772 711	110 353	3.98
GrayC	716 227	708 562	98.92

4.2.2　可靠性对比

在可靠性方面, 由于 AFL++的字符级别变异操作符实现逻辑非常简单, 易于人工验证. 如表 5 所示, 在整个 5 次 24 h 的对比实验中, 我们未曾观测到 AFL++的崩溃发生. 而相较而言, SemaAFL 平均在每 24 h 的时间段内仅发生了 4 821 次崩溃, 这一崩溃频率仅有 GrayC 的约 1/80. 同时由于我们采用了进程隔离, 这些崩溃没有对 SemaAFL 的模糊测试进程产生任何影响. 平均到每个崩溃而言, SemaAFL 发生一个崩溃需要耗费近 20 s 的时间, 而 GrayC 仅需要 0.22 s. 这里需要额外说明的是, 由于 GrayC 在实施变异前需要先将输入文件复制到输出文件, 然后再在输出文件上进行原地变异, 因此即使发生崩溃, 也至少能保留输出文件, 这也是表 3 中 GrayC 有高达 83% 的输出与输入相同的原因.

表 5　崩溃发生的频率对比

工具	总崩溃次数	每小时崩溃次数	平均每个崩溃发生的时间 (s)
SemaAFL	4 821	200	17.92
AFL++	0	0	—
GrayC	379 805	15 825	0.22

4.3　测试能力横向对比

图 5 展示了 SemaAFL、GrayC 和 AFL++ 在 GCC-14 和 Clang-18 上的平均覆盖率趋势. 总体而言, SemaAFL 在 GCC 和 Clang 上比 GrayC 和 AFL++ 的最佳结果提高了 14.5% 和 11.2%. 而 GrayC 仅通过 5 个精心设计的语义级别的变异操作符, 便在 GCC 和 Clang 上的表现优于 AFL++. 这些结果表明, SemaAFL 在探索编译器的不同行为上较 GrayC 和 AFL++具有显著优势.

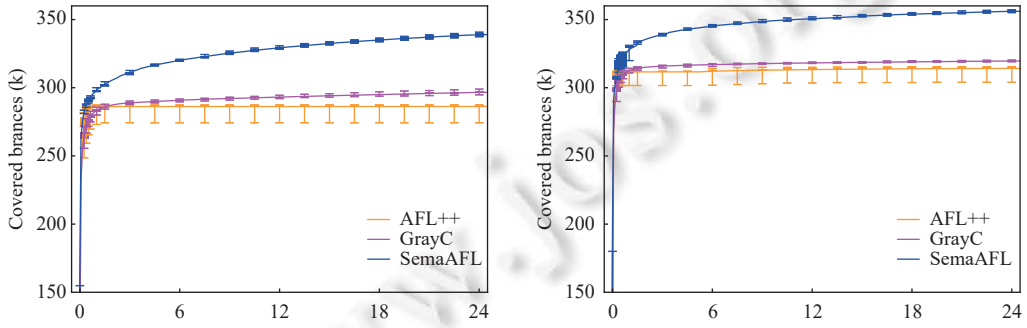

图 5　GCC-14 和 Clang-18 的各个测试工具在 24 h 内的覆盖率趋势

SemaAFL 的出色表现归因于其高效的语义级别的变异策略, 这使得它能够更深入地探索代码路径, 发现更多的边界情况和潜在的漏洞. 此外, GrayC 虽然没有 SemaAFL 表现出色, 其利用少量但高效的可感知语法的变异操

作符也展示了强大的能力, 特别是在处理复杂语法结构时表现尤为突出. 相比之下, AFL++ 尽管作为传统模糊测试工具同样表现出色, 但在感知语义方面略显不足.

这些结果表明, 当模糊测试工具能够利用语义信息进行变异时, 其探索代码覆盖率的能力能够得到显著提升, SemaAFL 成功例证了这一点. 这也进一步揭示了未来模糊测试工具的发展方向, 即结合语义信息的利用能力, 全面提升代码覆盖率和缺陷发现能力.

4.4 缺陷挖掘的能力

我们在一台第 4.1 节所述的服务器上运行 SemaAFL 一周时间, 合计在 GCC-14 和 Clang-18 上找到并上报了 6 个缺陷, 这些缺陷的详细信息如表 6 所示. 这 6 个缺陷均在 24 h 内被开发者确认, 且均为前所未知的缺陷 (尽在最近的数个版本可以触发).

表 6　已上报的 GCC-14 和 Clang-18 的缺陷

编译器	缺陷编号	缺陷概述
GCC	#115641	GCC crashes on function has attribute ‘__attribute__((const))’
GCC	#115642	internal compiler error: tree check: expected class ‘type’, have ‘exceptional’ (error_mark) in c_expr_sizeof_expr
GCC	#115644	ICE if redeclare a variable with different type
GCC	#115646	ICE in gen_conditions_for_pow_int_base, at tree-call-cdce.cc:587
Clang	#96624	UNREACHABLE executed: Non-canonical and dependent types shouldn’t get here
Clang	#96627	UNREACHABLE executed: Type not integer, floating, or complex

下面讨论两个上报的缺陷案例, 并展示基于语义信息的变异操作符是如何找到这两个缺陷的. 这两个缺陷均导致了编译器的崩溃, 一个在 GCC 的最新版本中, 一个在 Clang 的最新版本中.

缺陷案例 1: 图 6 展示了我们所发现的其中一个 GCC 的缺陷. 该缺陷会导致 GCC 在运行过程中崩溃. 我们通过“随机插入属性”这一语义级别的变异操作符, 给函数 f 插入了__attribute__((const)) 这一属性, 如图 6 第 8 行所示. 该属性会通知编译器函数 f 是纯函数, 不会依赖外部状态且具有相同输入时总是返回相同的输出. 在添加这一属性之后, GCC 崩溃并抛出了内部编译器错误 (ICE) 信息: expected tree that contains ‘decl common’ structure, have ‘call_expr’ in tree_could_trap_p.

缺陷案例 2: 图 7 展示了我们所发现的其中一个 Clang 的缺陷. 该缺陷会导致 Clang 在运行过程中崩溃. 我们通过“随机插入数学函数调用”这一语义级别的变异操作符, 给函数 f 的返回语句的返回表达式中的子表达式 c1 插入了对数学函数 abs 的调用, 如图 7 第 11、12 行所示. 在添加这一数学函数调用之后, Clang 崩溃并抛出了内部编译器错误 (ICE) 信息: UNREACHABLE executed: Type not integer, floating, or complex.

```
 1  #include <stdlib.h>
 2
 3  typedef struct {
 4    char hours, day, month;
 5    short year;
 6  } T;
 7
+8  __attribute__((const))
 9  T f (void) {
10    T now;
11    return now;
12  }
13
14  int main () {
15    if (f ().hours != 1 || f ().day != 2
16     || f ().month != 3 || f ().year != 4)
17      abort ();
18    return 0;
19  }
```

图 6　GCC 的崩溃缺陷 (编号#115641)

```
 1  typedef int v4si __attribute__ ((
 2    vector_size (1*sizeof(int))));
 3  typedef float v4sf __attribute__ ((
 4    vector_size (1*sizeof(float))));
 5
 6  __attribute__((noipa))
 7  v4si f(v4si a, v4si b, v4sf fa, v4sf fb)
 8  {
 9    const v4si c = fa < fb;
10    const v4si c1 = fa >= fb;
-11   return (c & a) | c1;
+12   return (c & a) | abs(c1);
13  }
```

图 7　Clang 的崩溃缺陷 (编号#96627)

5 讨 论

5.1 局限性

SemaAFL 专为 C/C++语言模糊测试而设计, 与 AFL 系列通用模糊测试工具相比, 其适用范围更为聚焦. 通用模糊测试工具采用字符串级别变异, 不依赖具体输入结构, 可应用于任意软件系统. 相比之下, SemaAFL 专注于 C/C++编译器测试, 原因如下: 首先, C/C++是系统级编程中最广泛使用的语言, 其编译器质量对整个软件生态系统的安全性和可靠性具有关键影响. 其次, 聚焦特定领域使我们能够精确建模程序的语法和语义结构合法性. 借助成熟的 C/C++语法语义分析工具 (同为编译器工具链的组成部分) 即可实现高效、可靠的语法和语义分析.

与 C/C++语言的绑定导致了 SemaAFL 实现方面的局限. 目前, SemaAFL 依赖 libclang 对 C/C++程序进行语法和语义分析, 因此无法解析非标准的语言扩展特性, 从而无法对这些特性进行有效测试. 例如, 虽然嵌套的函数定义并不属于 C/C++语言标准, 但在 GCC 等编译器中却作为一种语言扩展特性而被支持. 而 libclang 对这一特性就缺乏支持, 导致含有该特性的程序无法被正确解析.

5.2 多语言支持

SemaAFL 目前支持 C/C++编译器的模糊测试, 但其基于语义信息的变异操作符理念并不局限于特定编程语言. 主流编程语言普遍遵循相似的设计范式, 使得某些语义级别的操作符 (如“修改二元表达式”) 可适用于多种语言 (如 Java、Python、Go、Rust 等). 这为 SemaAFL 向其他语言的扩展提供了基础, 使我们能够在很大程度上复用现有实现, 而非完全重新开发.

具体来说, 若要适配新的编程语言, 我们需要为 SemaAFL 的变异操作符实现以下功能.

(1) AST 解析与重写: 目前, 最广泛使用的 C/C++解析器是 libClang, 它支持 C/C++标准语法及多种 GNU 扩展. 对于其他编程语言, 开发者可以利用 ANTLR[24]和 tree-sitter[25]等工具, 这些工具提供了比 libClang 更简易的 API.

(2) 语义查询与检查: 开发者可以复用使用语言服务器协议 (language server protocol)[26]实现的现有分析 (如标识符索引器和 linter). 此外, 这些分析不需要非常精确——保守的方法仍然可以产生有用的变异操作符, 尽管代价是会产生更多无效的程序.

以 Java 为例, 实现变异操作符的流程可概括为使用 ANTLR 解析程序为语法树, 遍历树以定位二元操作符节点, 确定操作符在源程序中的位置, 替换为其他合法操作符, 最后通过 LSP 进行语义合法性检查, 若合法则输出变异体.

然而, 将 SemaAFL 扩展至其他语言仍面临挑战. 每种语言都有其独特特性 (如 Java 的注解、C++的模板、Python 的装饰器), 这些语言特有元素可能需要定制处理逻辑, 难以直接跨语言移植. 尽管存在这些障碍, 我们认为 SemaAFL 的核心理念具有跨语言应用潜力. 通过精心设计和实现, 我们可逐步将 SemaAFL 扩展到更多编程语言.

6 相关工作

6.1 灰盒模糊测试

灰盒模糊测试作为一种有效的软件测试方法, 近年来得到了广泛的研究和应用. 最初的灰盒模糊测试通过随机变异格式正确的输入来探测程序是否会崩溃[27]. 随后, 研究者借鉴遗传算法思路, 引入代码覆盖率作为变异指导. 其中, AFL[2]是一个里程碑式的工作, 它使用轻量级插桩技术收集覆盖率信息, 大大提高了测试效率. AFL++[20]在其基础上整合了多个 AFL 变种的优秀特性, 进一步增强了工具的功能和适用性. CollAFL[5]通过改进覆盖率收集方案, 解决了路径碰撞问题, 提高了覆盖率精确度. FairFuzz[4]专注于提高对稀有分支的覆盖, 有助于发现深层次漏洞. Steelix[3]则致力于解决对比较操作的处理难题, 提高了对魔法字节的识别能力.

在此基础上, 一些研究者提出了更加专门化的方法. 例如, Angora[10]通过字节级别的污点分析和梯度下降算法

来解决复杂的路径约束问题. QSYM[28]结合了符号执行和具体执行, 提高了对复杂路径的探索能力. Matryoshka[29]提出了一种嵌套层次的模糊测试策略, 可以更有效地测试深层嵌套的程序结构. 此外, VUzzer[9]利用控制流和数据流分析来指导输入生成, 提高了对深层路径的覆盖. Redqueen[30]提出了一种基于输入到输出的推理技术, 能够有效地处理复杂的比较操作. MOPT[31]引入了基于粒子群优化的突变调度策略, 提高了模糊测试的效率.

除了以覆盖率为指导的工作, 还有一类以程序性能为指导的研究, 这些研究涵盖了多个性能相关的方面. SlowCoach[32]和 SlowFuzz[6]专注于发现可能导致程序性能下降的输入, 而 HotFuzz[33]则针对 Java 程序, 通过观察方法执行时间来识别性能问题. 在资源使用方面, PerfFuzz[7]通过最大化资源使用来发现性能瓶颈, MemLock[34]则专注于检测内存消耗异常, 可以发现潜在的内存泄漏和内存耗尽问题. 在安全方面, Pythia[35]致力于发现时间侧信道漏洞, 此外, Singularity[36]采用了创新的方法, 利用机器学习技术来预测输入对性能的影响. 这些多样化的研究方法共同构成了一个全面的性能导向模糊测试框架, 为开发者提供了有力的工具来发现和解决各种潜在的性能问题和安全漏洞.

此外, 一些研究者开始探索如何将机器学习技术应用于灰盒模糊测试. 例如, Neuzz[37]使用神经网络来学习程序的边界行为, 从而更有效地生成测试用例. MTFuzz[38]则利用多任务学习来同时优化多个测试目标, 提高了测试的全面性. 在这一趋势下, 更多的工作开始涌现. 例如, Learn&Fuzz[39]使用序列到序列的学习模型来生成结构化的测试输入. DeepFuzz[40]利用深度强化学习来指导模糊测试过程, 提高了对深层漏洞的发现能力. FuzzGuard[41]使用机器学习来预测测试用例的有效性, 从而优化测试用例的选择. Cerebro[42]结合了程序分析和机器学习, 可以更准确地预测程序行为, 提高测试效率.

这些工作极大地推动了灰盒模糊测试的发展, 提高了漏洞发现的效率. 然而, 对于需要输入具有语义合法性的程序, 现有方法的能力仍然有限. 如何在保证变异后输入的语义合法性的同时提高灰盒模糊测试的效率, 仍是一个值得探索的问题. 未来的研究可能需要更深入地结合程序分析技术、机器学习方法以及领域特定知识, 以应对日益复杂的软件系统带来的挑战.

6.2 语法可感知的模糊测试

近年来, 学术界和业界日益关注提升灰盒模糊测试对测试输入语法合法性的感知能力. 这一趋势主要源于特定软件系统 (如 XML 解析器) 对输入语法合法性的严格要求. 为了应对这一挑战, 研究者们提出了多种创新方法. 例如, MongoDB 的模糊测试工具[43]对 JavaScript 测试输入的抽象语法树进行受控破坏. Superion[1]采用基于树的增量变异方式, 并具有良好的通用性. LangFuzz[44]和 IFuzzer[45]都使用语法来提取和重组代码片段, 生成新的测试输入. μ4SQLi[46]则专注于 SQL 语言, 使用专家设计的变异算子来生成语法正确且可执行的 SQL 语句. Skyfire[47]利用概率上下文敏感语法来生成结构化的种子输入, 提高了对复杂格式的测试效率. Nautilus[48]则针对解释器和编译器, 提出了一种基于语法的模糊测试方法, 可以生成语法正确的程序进行测试. Fuzzilli[49]专门针对 JavaScript 引擎设计, 使用定制的中间表示来生成有效的 JavaScript 代码. 此外, Gramatron[23]提出了一种基于文法的随机测试方法, 可以高效地生成符合特定语法的输入. Grimoire[50]结合了输入语法推断和基于语法的模糊测试, 可以在不需要预定义语法的情况下进行结构感知的模糊测试. Zest[51] 则提出了一种创新的生成器参数变异方法, 通过调整生成器的输入参数来实现输入变异, 同时保持测试输入的合法性. 这些方法各具特色, 共同推动了灰盒模糊测试在处理需求语法合法输入的待测程序方面的进步.

6.3 语义可感知的模糊测试

如第 1 节所述, 编译器这类复杂的软件系统不仅要求输入在语法上合法, 更需要在语义层面上合法 (例如, 变量必须在定义后才能使用). 大量代码在经过语法和语义检查后才会执行. 因此, 要探索编译器的不同行为, 关键在于保持输入的语义合法性. 然而, 第 6.2 节所述的语法可感知的模糊测试方法, 由于缺乏对程序语义的建模, 难以保障基本的语义合法性 (如变量定义使用顺序). 这导致变异后的程序往往只在语法上合法, 但在语义上不合法.

近年来, 研究者们在提高模糊测试的语义合法性保持能力和语义多样性探索能力方面也做出了诸多创新尝试. CodeAlchemist[52]采用了语义保持的代码合成技术, 能够生成语义有效的 JavaScript 代码用于测试. SPE[53]通过

枚举所有 α 不等价的程序, 深入探究了不同变量定义使用关系对编译器编译行为的影响. FCFUZZER[54]通过分阶段枚举同一程序框架下所有可能的函数调用组合, 来探索函数调用对编译行为的影响. JAttack[55]创新性地结合了程序框架和程序片段的概念, 通过预定义的程序结构和可插拔的代码片段, 生成语法正确且语义合理的 Java 测试用例. ClassFuzz[56]通过人工设计的 129 个变异操作符对 Java 类文件进行变异, 显著提高了 JVM 测试效率. GrayC[22]尝试通过人工设计可保持语义合法性的变异操作符来保障变异后程序的合法性. 然而, 由于可靠性和可用性问题, GrayC 在灰盒模糊测试中的应用效果仍有待提高. 尽管这些工作在提高模糊测试的语义合法性保持能力方面取得了显著进展, 但在面对 C/C++编译器这类对高度语义结构有严格要求的被测程序时, 它们的能力仍显不足. 如何在确保生成输入的语义合法性的同时, 有效提高模糊测试的覆盖率和缺陷发现能力, 仍然是一个亟需解决的关键问题.

References:

[1] Wang JJ, Chen BH, Wei L, Liu Y. Superion: Grammar-aware greybox fuzzing. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 724–735. [doi: 10.1109/ICSE.2019.00081]

[2] Zalewski M. American fuzzy lop. 2024. http://lcamtuf.coredump.cx/afl/

[3] Li YK, Chen BH, Chandramohan M, Lin SW, Liu Y, Tiu A. Steelix: Program-state based binary fuzzing. In: Proc. of the 11th Joint Meeting on Foundations of Software Engineering. Paderborn: ACM, 2017. 627–637. [doi: 10.1145/3106237.3106295]

[4] Lemieux C, Sen K. FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proc. of the 33rd ACM/IEEE Int'l Conf. on Automated Software Engineering. Montpellier: ACM, 2018. 475–485. [doi: 10.1145/3238147.3238176]

[5] Gan ST, Zhang C, Qin XJ, Tu XW, Li K, Pei ZY, Chen ZN. CollAFL: Path sensitive fuzzing. In: Proc. of the 2018 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2018. 679–696. [doi: 10.1109/SP.2018.00040]

[6] Petsios T, Zhao J, Keromytis AD, Jana S. SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. Dallas: ACM, 2017. 2155–2168. [doi: 10.1145/3133956.3134073]

[7] Lemieux C, Padhye R, Sen K, Song D. PerfFuzz: Automatically generating pathological inputs. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Amsterdam: ACM, 2018. 254–265. [doi: 10.1145/3213846.3213874]

[8] Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing. In: Proc. of the 31st IEEE Int'l Conf. on Software Engineering. Vancouver: IEEE, 2009. 474–484. [doi: 10.1109/ICSE.2009.5070546]

[9] Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H. VUzzer: Application-aware evolutionary fuzzing. In: Proc. of the 24th Network and Distributed System Security Symp. San Diego: NDSS, 2017. 1–14. [doi: 10.14722/ndss.2017.23404]

[10] Chen P, Chen H. Angora: Efficient fuzzing by principled search. In: Proc. of the 2018 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2018. 711–725. [doi: 10.1109/SP.2018.00046]

[11] Microsoft. Security development lifecycle (SDL) practices. 2024. https://www.microsoft.com/en-us/sdl/process/verification.aspx

[12] Bounimova E, Godefroid P, Molnar D. Billions and billions of constraints: Whitebox fuzz testing in production. In: Proc. of the 35th Int'l Conf. on Software Engineering. San Francisco: IEEE, 2013. 122–131. [doi: 10.1109/ICSE.2013.6606558]

[13] The chromium projects. 2024. https://www.chromium.org/Home/chromium-security/bugs

[14] Aizatsky M, Serebryany K, Chang O, Arya A, Whittaker M. OSS-Fuzz: Continuous fuzzing for open source software. 2024. https://github.com/google/oss-fuzz

[15] Chrome Security Team. ClusterFuzz. 2024. https://google.github.io/clusterfuzz/

[16] Manès VJM, Han H, Han C, Cha SK, Egele M, Schwartz EJ, Woo M. The art, science, and engineering of fuzzing: A survey. IEEE Trans. on Software Engineering, 2021, 47(11): 2312–2331. [doi: 10.1109/TSE.2019.2946563]

[17] Li J, Zhao BD, Zhang C. Fuzzing: A survey. Cybersecurity, 2018, 1(1): 6. [doi: 10.1186/s42400-018-0002-y]

[18] Zhu XG, Wen S, Camtepe S, Xiang Y. Fuzzing: A survey for roadmap. ACM Computing Surveys, 2022, 54(11s): 230. [doi: 10.1145/3512345]

[19] Zhao XQ, Qu HP, Xu JL, Li XH, Lv WJ, Wang GG. A systematic review of fuzzing. Soft Computing, 2024, 28(6): 5493–5522. [doi: 10.1007/s00500-023-09306-2]

[20] Fioraldi A, Maier D, Eißfeldt H, Heuse M. AFL++: Combining incremental steps of fuzzing research. In: Proc. of the 14th USENIX Conf. on Offensive Technologies. USENIX Association, 2020. 10.

[21] Liang J, Wu ZY, Fu JZ, Zhu J, Jiang Y, Sun JG. Survey on database management system fuzzing techniques. Ruan Jian Xue Bao/Journal

of Software, 2025, 36(1): 399–423 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7048.htm [doi: 10.13328/j.cnki.jos.007048]

[22] Even-Mendoza K, Sharma A, Donaldson AF, Cadar C. GrayC: Greybox fuzzing of compilers and analysers for C. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 1219–1231. [doi: 10.1145/3597926.3598130]

[23] Srivastava P, Payer M. Gramatron: Effective grammar-aware fuzzing. In: Proc. of the 30th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2021. 244–256. [doi: 10.1145/3460319.3464814]

[24] Parr TJ, Quong RW. ANTLR: A predicated-*LL*(*k*) parser generator. Software: Practice and Experience, 1995, 25(7): 789–810. [doi: 10.1002/spe.4380250705]

[25] Max Brunsfeld. Tree-sitter. 2024. https://tree-sitter.github.io/tree-sitter/

[26] Bünder H. Decoupling language and editor-the impact of the language server protocol on textual domain-specific languages. In: Proc. of the 7th Int'l Conf. on Model-Driven Engineering and Software Development. Prague: ScitePress, 2019. 129–140. [doi: 10.5220/0007556301290140]

[27] Miller BP, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. Communications of the ACM, 1990, 33(12): 32–44. [doi: 10.1145/96267.96279]

[28] Yun I, Lee S, Xu M, Jang Y, Kim T. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In: Proc. of the 27th USENIX Conf. on Security Symp. Baltimore: USENIX Association, 2018. 745–761.

[29] Chen P, Liu JZ, Chen H. Matryoshka: Fuzzing deeply nested branches. In: Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security. London: ACM, 2019. 499–513. [doi: 10.1145/3319535.3363225]

[30] Aschermann C, Schumilo S, Blazytko T, Gawlik R, Holz T. Redqueen: Fuzzing with input-to-state correspondence. In: Proc. of the 26th Network and Distributed System Security Symp. 2019. [doi: 10.14722/ndss.2019.23371]

[31] Lyu CY, Ji SL, Zhang C, Li YW, Lee WH, Song Y, Beyah R. MOPT: Optimized mutation scheduling for fuzzers. In: Proc. of the 28th USENIX Conf. on Security Symp. Santa Clara: USENIX Association, 2019. 1949–1966.

[32] Chen YQ, Schwahn O, Natella R, Bradbury M, Suri N. SlowCoach: Mutating code to simulate performance bugs. In: Proc. of the 33rd IEEE Int'l Symp. on Software Reliability Engineering. Charlotte: IEEE, 2022. 274–285. [doi: 10.1109/ISSRE55969.2022.00035]

[33] Blair W, Mambretti A, Arshad S, Weissbacher M, Robertson W, Kirda E, Egele M. HotFuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. In: Proc. of the 27th Network and Distributed System Security Symp. San Diego: NDSS, 2020. 1–18. [doi: 10.14722/ndss.2020.24415]

[34] Wen C, Wang HJ, Li YK, Qin SC, Liu Y, Xu ZW, Chen HX, Xie XF, Pu GG, Liu T. MemLock: Memory usage guided fuzzing. In: Proc. of the 42nd Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 765–777. [doi: 10.1145/3377811.3380396]

[35] Atlidakis V, Geambasu R, Godefroid P, Polishchuk M, Ray B. Pythia: Grammar-based fuzzing of REST APIs with coverage-guided feedback and learning-based mutations. arXiv:2005.11498, 2020.

[36] Wei JY, Chen J, Feng Y, Ferles K, Dillig I. Singularity: Pattern fuzzing for worst case complexity. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Lake Buena Vista: ACM, 2018. 213–223. [doi: 10.1145/3236024.3236039]

[37] She DD, Pei KX, Epstein D, Yang JF, Ray B, Jana S. Neuzz: Efficient fuzzing with neural program smoothing. In: Proc. of the 2019 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2019. 803–817. [doi: 10.1109/SP.2019.00052]

[38] She DD, Krishna R, Yan L, Jana S, Ray B. MTFuzz: Fuzzing with a multi-task neural network. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 737–749. [doi: 10.1145/3368089.3409723]

[39] 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. Urbana-Champaign: IEEE, 2017. 50–59.

[40] Liu X, Li XT, Prajapati R, Wu DH. DeepFuzz: Automatic generation of syntax valid C programs for fuzz testing. In: Proc. of the 33rd AAAI Conf. on Artificial Intelligence. Honolulu: AAAI, 2019. 1044–1051. [doi: 10.1609/aaai.v33i01.33011044]

[41] Zong PY, Lv T, Wang DW, Deng ZZ, Liang RG, Chen K. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In: Proc. of the 29th USENIX Conf. on Security Symp. USENIX Association, 2020. 127.

[42] Li YK, Xue YX, Chen HX, Wu XH, Zhang C, Xie XF, Wang HJ, Liu Y. Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Tallinn: ACM, 2019. 533–544. [doi: 10.1145/3338906.3338975]

[43] Guo R. MongoDB's JavaScript Fuzzer: The fuzzer is for those edge cases that your testing didn't catch. Queue, 2017, 15(1): 38–56. [doi:

10.1145/3055301.3059007]
[44] Holler C, Herzig K, Zeller A. Fuzzing with code fragments. In: Proc. of the 21st USENIX Security Symp. Bellevue: USENIX Association, 2012. 38.
[45] Veggalam S, Rawat S, Haller I, Bos H. IFuzzer: An evolutionary interpreter fuzzer using genetic programming. In: Proc. of the 21st European Sym. on Research in Computer Security. Heraklion: Springer, 2016. 581–601. [doi: 10.1007/978-3-319-45744-4_29]
[46] Appelt D, Nguyen CD, Briand LC, Alshahwan N. Automated testing for SQL injection vulnerabilities: An input mutation approach. In: Proc. of the 2014 ACM Int'l Symp. on Software Testing and Analysis. San Jose: ACM, 2014. 259–269. [doi: 10.1145/2610384.2610403]
[47] Wang JJ, Chen BH, Wei L, Liu Y. Skyfire: Data-driven seed generation for fuzzing. In: Proc. of IEEE Symp. on Security and Privacy. San Jose: IEEE, 2017. 579–594. [doi: 10.1109/SP.2017.23]
[48] Aschermann C, Frassetto T, Holz T, Jauernig P, Sadeghi AR, Teuchert D. Nautilus: Fishing for deep bugs with grammars. In: Proc. of the 26th Network and Distributed System Security Symp. 2019. [doi: 10.14722/ndss.2019.23412]
[49] Groß S, Koch S, Bernhard L, Holz T, Johns M. Fuzzilli: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In: Proc. of the 30th Network and Distributed System Security Symp. San Diego: NDSS, 2023. 1–17. [doi: 10.14722/ndss.2023.24290]
[50] Blazytko T, Aschermann C, Schlögel M, Abbasi A, Schumilo S, Wörner S, Holz T. Grimoire: Synthesizing structure while fuzzing. In: Proc. of the 28th USENIX Conf. on Security Symp. Santa Clara: USENIX Association, 2019. 1985–2002.
[51] Padhye R, Lemieux C, Sen K, Papadakis M, Le Traon Y. Semantic fuzzing with zest. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: ACM, 2019. 329–340. [doi: 10.1145/3293882.3330576]
[52] Han H, Oh D, Cha SK. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In: Proc. of the 27th Network and Distributed System Security Symp. San Diego: NDSS, 2019. 1–15. [doi: 10.14722/ndss.2019.23263]
[53] Zhang QR, Sun CN, Su ZD. Skeletal program enumeration for rigorous compiler testing. In: Proc. of the 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Barcelona: ACM, 2017. 347–361. [doi: 10.1145/3062341.3062379]
[54] Xia XM, Feng Y. Detecting interpreter bugs via filling function calls in skeletal program enumeration. In: Proc. of the 34th IEEE Int'l Symp. on Software Reliability Engineering. Florence: IEEE, 2023. 612–622. [doi: 10.1109/ISSRE59848.2023.00066]
[55] Zang ZQ, Wiatrek N, Gligoric M, Shi A. Compiler testing using template java programs. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: ACM, 2022. 23. [doi: 10.1145/3551349.3556958]
[56] Chen YT, Su T, Sun CN, Su ZD, Zhao JJ. Coverage-directed differential testing of JVM implementations. In: Proc. of the 37th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Santa Barbara: ACM, 2016. 85–99. [doi: 10.1145/2908080.2908095]

附中文参考文献:

[21] 梁杰, 吴志镛, 符景洲, 朱娟, 姜宇, 孙家广. 数据库管理系统模糊测试技术研究综述. 软件学报, 2025, 36(1): 399–423. http://www.jos.org.cn/1000-9825/7048.htm [doi: 10.13328/j.cnki.jos.007048]

欧先飞(1996－), 男, 博士生, CCF 学生会员, 主要研究领域为编译器测试.

许畅(1977－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为软件测试与分析, 自适应软件系统.

蒋炎岩(1988－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为系统软件, 软件自动化.