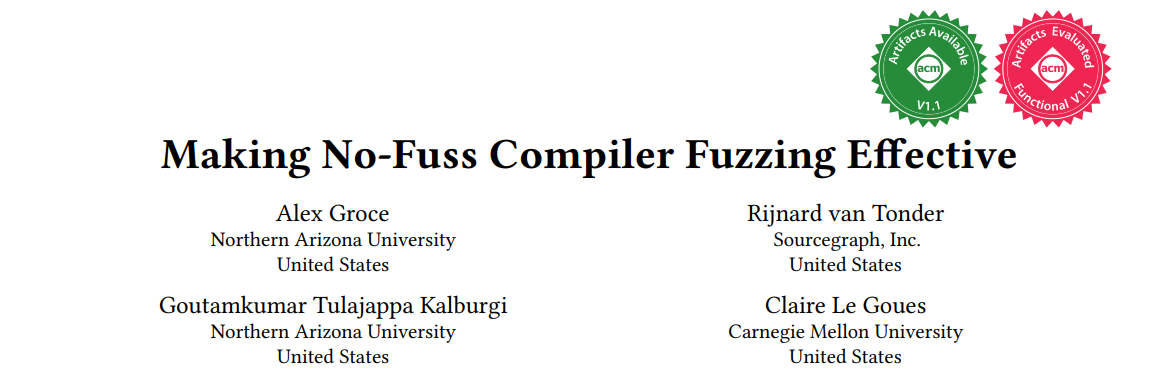
# 论文复现：Making No-Fuss Compiler Fuzzing Effective[[[1]](#endnote-0)]

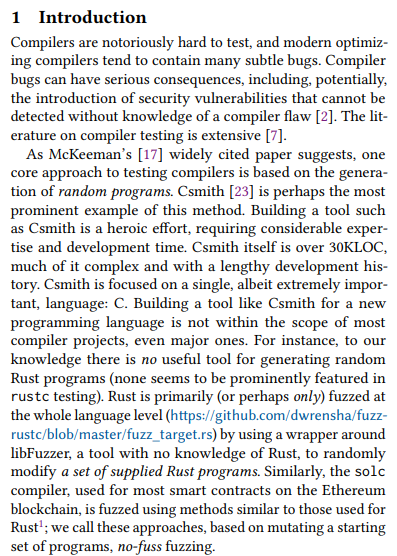
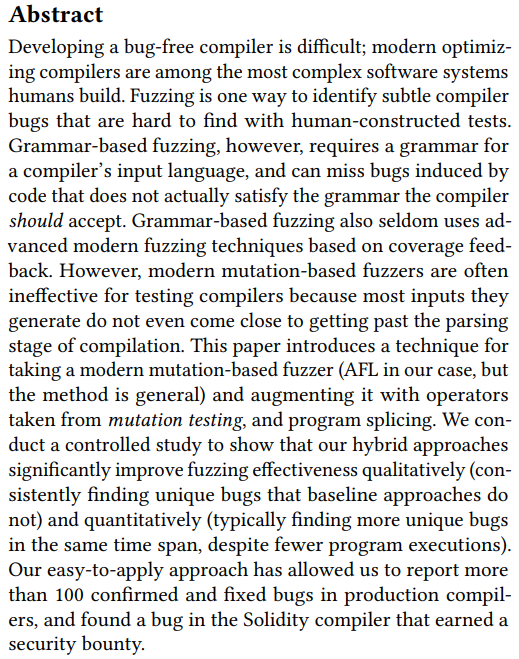
论文简介：

本人选择的复现论文题为Making No-Fuss Compiler Fuzzing Effective（使no-fuss编译器模糊测试变得有效），由美国北亚利桑那大学的Alex Groce和Goutamkumar Tulajappa Kalburgi所著。该篇论文的研究中心主要是放在对于编译器的模糊测试研究上，编译器作为连接顶层与操作系统的重要环节，看似与计算机体系结构的硬件细节并没有联系，然而，计算机体系结构中的指令集架构（如x86、ARM等）直接影响着编译器生成的机器代码。编译器需要将高级语言代码转换为特定指令集架构的机器码，以便计算机能够理解和执行；编译器还能根据不同的计算机体系结构采取不同的优化策略。例如，针对不同的处理器架构可能会选择不同的指令序列或优化技术，以提高程序执行效率；计算机体系结构对内存管理、寻址方式和内存访问速度等有重要影响。而编译器需要考虑这些因素来优化程序的内存访问模式，以确保程序在特定计算机体系结构下能够高效运行；计算机体系结构中的支持并行处理的特性（如多核处理器）需要编译器能够生成充分利用这些特性的代码，以实现并行计算和优化性能；不同的计算机体系结构可能有不同的指针大小和数据类型表示方式。而编译器需要了解这些特性并相应地生成适配特定体系结构的代码。那么，作为体系结构的重要桥梁，编译器的安全性、稳定性与适应性自然需要得到很重要的保障，此时，模糊测试的重要性和必要性就体现出来了。接下来，笔者先简单梳理论文的研究背景，研究问题，采用的方法以及实现的本地复现结果。



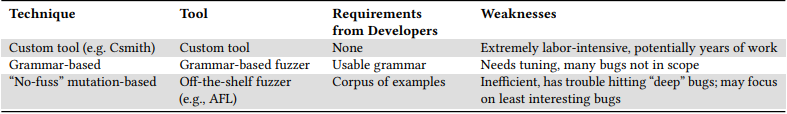
研究背景：

在摘要和论文介绍中，论文表示开发一个没有bug的编译器是很困难的。现代优化编译器是人类构造的最复杂的软件系统之一。模糊化是一种识别微妙编译器错误的而在人工构建的测试中很难找到错误的方法。然而，基于语法的模糊化需要编译器的输入语言，并且可能会错过由实际上不符合语法的代码编译器应该接受。基于语法的模糊处理也很少使用基于覆盖反馈的先进的现代模糊处理技术。然而，现代基于突变的模糊器通常对测试编译器无效，因为大多数输入generate甚至没有通过解析汇编阶段。不仅如此，编译器也是出了名的难测试的,现代优化编译器往往包含许多微妙的漏洞。编译器漏洞可能会产生严重的后果，包括可能引入安全漏洞，如果没有对编译器缺陷的了解，则无法检测到这些漏洞。正如 McKeeman 的广泛引用的论文所建议的那样，测试编译器的一种核心方法是基于随机程序的生成。Csmith是这种方法中最著名的例子。构建像 Csmith 这样的工具是一项伟大的成就，需要相当的专业知识和开发时间。Csmith本身就有超过 30KLOC 的代码，其中大部分都很复杂，并有着漫长的开发历史。Csmith关注的是单个的，但是却极其重要的语言：C。为新的编程语言构建像 Csmith 这样的工具并不在大多数编译器项目的范围之内，即使是主要的大型的编译器项目也不例外，它们也没有一个备用的随机测试和编译器/语言专家团队，所以构建类似csmith的工具是不可能的。这意味着从头开始生成有效程序的唯一方法是使用一个工具，它将语法作为输入，并生成满足语法的随机输出。然而，这种方法存在多重问题。首先，在许多情况下，由语法产生的程序，如果不广泛关注调整生产的概率等，将大多是无趣的(有趣的概念应当为：由于输入的变异而导致编译器进入全新的路径或做出新的应对行为;无趣则相反：与变异前的道路无异)。将这些问题结合在一起，往往会使大多数编译器在查找bug方面效率低下，并且容易发现不那么有趣的bug。因此，本篇论文便以此为背景，并且以寻求一种方法，能够提高编译器模糊测试的效果，同时避免增加开发人员的负担为动机而出发探讨研究。



研究问题：

本篇论文的研究主要关注于改进编译器模糊测试的有效性。由研究背景部分提出的问题可见，在传统的编译器模糊测试中，通常使用基于变异的方法，如AFL（American Fuzzy Lop），但这些方法最初设计用于测试二进制格式，其在测试编译器方面存在一些局限性。因此，这项研究旨在解决编译器模糊测试的有效性问题，而无需开发人员提供特定的语法或字典。主要问题是如何利用变异测试思想，以改善基于AFL的编译器模糊测试的性能，并尝试减少开发人员的负担。通过提出的新策略，试图在编译器模糊测试中提高漏洞发现的效率和数量，同时减少测试过程对开发资源的依赖。

(几种传统的编译器模糊测试方法对比)

采用方法：

首先定义no-fuss模糊测试的概念： Rust程序主要（或者可能仅仅）在整个语言层面上进行模糊测试，通过使用围绕着 libFuzzer 的包装器随机修改一组提供的 Rust 程序。类似地，用于以太坊区块链上的大多数智能合约的 solc 编译器也使用类似于 Rust 的方法进行模糊测试；我们称这些基于变异起始程序集的方法为 no-fuss 模糊测试

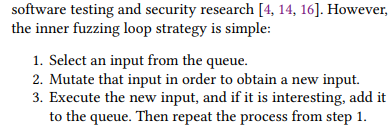
AFL基本模糊测试的内部模糊测试循环策略（算法 ）：

1. 从队列中选择一个输入。

2. 变异该输入以获得新的输入。

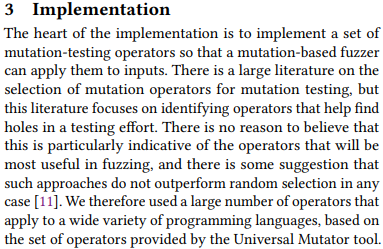
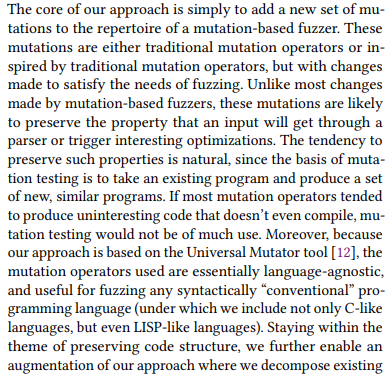
3. 执行新的输入，如果它有趣，则将其添加到队列中。然后从第1步开始重复该过程。

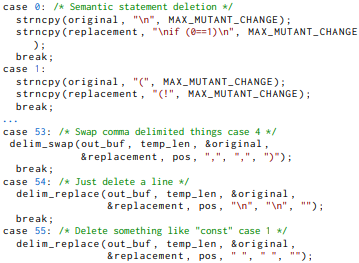
在步骤3中使编译器崩溃的输入被报告为bug。使用这样的模糊器通常是非常简单的，只需要1)使用特殊的工具构建编译器2和2)找到一组用作语料库的初始程序。



***论文中提到：*** 研究团队的工作重点是改进这个过程的第2步，对模糊测试的其他方面的实现细节视为不重要。特别是，在编译器模糊测试的文献中，大多数变异方法存在的问题是，例如基于字节的转换几乎总是将执行有趣编译器行为的程序编译成无法通过解析的程序。另一种替代的变异方法是所谓的“havoc”风格变异，通常涉及解决约束或遵循污点，但在编译器的情况下往往效果不佳，因为关系太复杂而难以解决/跟踪。第二种常见的方法是提供一个语言中有意义的字节序列的词典，这既对编译器开发人员来说是负担，效果也有限：例如，词典不能帮助模糊测试器删除诸如语句或块等代码的子单元。于是，研究团队提出了一种新的方法来为源代码生成大量有用的、有趣的突变，而不需要付出使编译器实际上不可行的分析代价，也不需要编译器开发人员付出任何额外的努力。研究团队的方法的核心是简单地向基于变异的模糊测试器的指令表中添加一组新的变异。这些变异要么是传统的变异算子，要么是受传统变异算子启发但经过更改以满足模糊测试的需求。与大多数基于变异的模糊测试器所做的更改不同，这些变异很可能保留输入将通过解析器或触发有趣优化的属性。此外，由于该方法是基于通用变异器工具，所使用的变异算子本质上是与语言无关的，并且适用于模糊测试任何语法上“传统”的编程语言（不仅包括类似C的语言，还包括类似LISP的语言）。为了保持代码结构的主题，他们进一步实现了一种扩展他们方法的方法，其中将现有的测试程序分解为组成部分片段，然后在运行时使用这些部分片段来合成和变异新的输入。

这种方法的实现的核心是实现一组变异测试算子，以便基于变异的模糊测试器可以将它们应用于输入。简单来说，该实现是基于正则表达式的文本近似方法，类似于Universal Mutator采用的方法。团队没有调用Python编写的变异工具，因为它相对较慢，而是使用低级别的C字符串库手工制作了所有语言（Universal Mutator的"universal"规则）和"C-like"语言的变异操作符的近似实现。大多数操作符通过选择要查找的字符串和要替换的字符串来实现；变异器会在原始字符串的随机位置找到一个出现，并将其替换为替换字符串。其他操作符需要更复杂的字符串操作，例如删除以分号分隔的语句或交换函数参数。然而，关键是，所有操作仅涉及基本的C字符串操作，并且对要变异的整个文本进行不超过4次线性扫描。在最坏的情况下，大多数操作仅需要进行一次线性扫描，并且大多数扫描在扫描输入的大片段之前终止。当选择的操作无法应用时（例如，要替换的字符串不存在），会尝试另一个操作，最多尝试一定次数



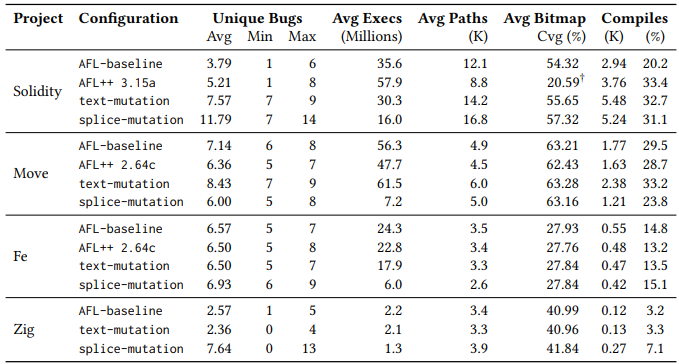
(部分实现)

***实验评估：***

**实验设置：**对四种编译器进行了评估，涉及 Solidity、Move、Fe 和 Zig3 语言。Solidity是在以太坊区块链上编写智能合约的知名语言，使用非常广泛。Fe是一种用于以太坊智能合约的实验性静态类型语言。Move是Facebook的智能合约语言，专为工业区块链解决方案 Diem 开发。Zig是一种新兴的系统语言，与 C、Rust、Nim 等静态类型语言及手动内存管理同处一个领域。Move 和 Fe 使用 Rust 实现，Solidity使用 C++ 实现，Zig 则是由C++和Zig本身混合实现的。

**模糊测试配置：**进行了四种模糊测试配置的比较评估。第一种配置是“快速且脏”(quick and dirty)模式下的标准AFL，是我们在所有项目中的对照基线。我们的方法直接集成到标准AFL中，因此我们与基线实现进行比较（这是进行合理比较模糊器评估的良好实践）。虽然AFL仍然是“无需麻烦”模糊测试的事实行业标准，但AFL++项目已经有了许多社区驱动的改进，通常可以优于标准AFL。因此，为了额外的对比，我们试图将结果与使用现有的AFL++工具的第二种模糊测试配置进行比较。我们成功地配置了AFL++用于我们四种编译器中的三种（不包括 Zig）。请注意，由于我们的技术没有在AFL++中实现，所以比较是不一致的，并且可能受到AFL++改进的干扰，这些改进可能会提升我们的方法。第三个配置纯粹应用字符串级别的变异，概率为75%。第四个配置将纯字符串级别变异策略与语法感知变异相结合，其中我们的AFL有33%的概率请求服务器生成新的输入（使用模板拼接），33%的概率在输入上执行字符串级别变异，以及34%的概率像平常一样运行AFL。

**模糊测试和持续时间：**我们对每个编译器的每种模糊测试配置运行了14次试验，以控制变异性和随机性。对于Solidity、Move和Fe编译器，我们运行了四种配置（包括AFL++），对于Zig运行了三种配置（不包括AFL++），共进行了210次试验。每次试验包括对单个核心的24小时模糊测试，从初始输入语料库开始。我们选择了24小时的试验，因为我们的目的是要观察那些旨在挖掘更深层漏洞的策略是否会产生（相对即时的）效果。我们的选择符合现有研究的观点，即在模糊测试活动的早期发现新漏洞的成本较低（更有可能）。如果我们的策略展现出任何显著的竞争优势，我们预期在受控环境中（24小时内）会早期显现。总体而言，我们的实验代表了210天的模糊测试，以展示具有我们的“无需麻烦”增强功能的模糊器在这些编译器上快速挖掘漏洞的性能。每个项目都在我们向上游仓库报告漏洞之前的早期提交时进行了模糊测试。



（该表显示了受控实验的主要结果：对每个项目进行了14次试验（每次试验持续24小时），使用了不同的配置：基线-AFL、AFL++、文本变异和splice变异。基线-AFL是标准的AFL；AFL++是社区驱动的增强型AFL。文本变异使用快速的基于字符串的变异算子（文本查找替换模式），在每个模糊输入上以75%的概率应用。剩余25%的时间，标准AFL对输入进行处理。splice变异是一种混合方法，

（1）以33%的概率应用文本变异算子；

（2）以33%的概率合成具有模板（splice）的语法感知输入；

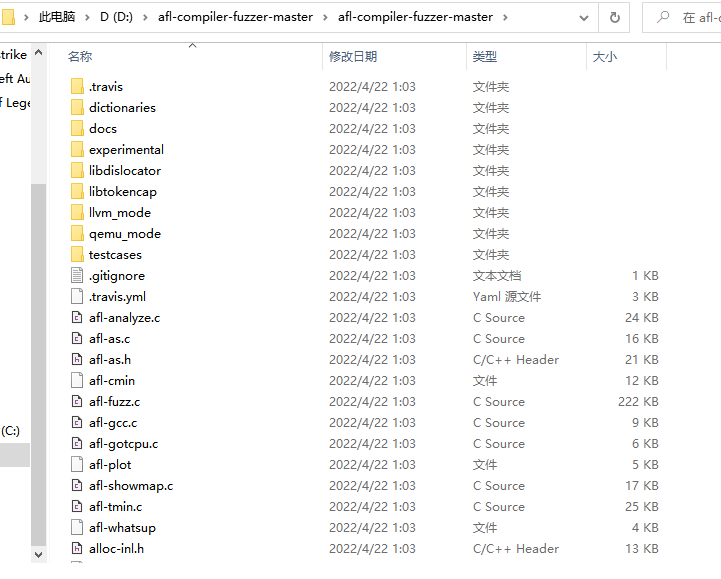
（3）剩余的34%时间使用标准AFL。† AFL++的更近期版本3.15a的工具设备与我们基线版本有所不同。）

**输入语料库和预处理：**所有模糊器试验都是在基于项目自己源代码树的输入上运行的。表2显示了一个总结。例如，Solidity基础语料库是位于test/libsolidity子目录中以.sol结尾的2,447个文件。对于使用语法感知输入生成的模糊测试试验，我们将基本语料库分解为独特的模板（Templ.）和具体程序片段（Frag.）。由于这个过程可能会生成非常大（因此较慢）的输入，我们移除所有大于4KB的模板和片段。对于Solidity来说，基本语料库分解成了9,308个模板和7,651个具体程序片段。

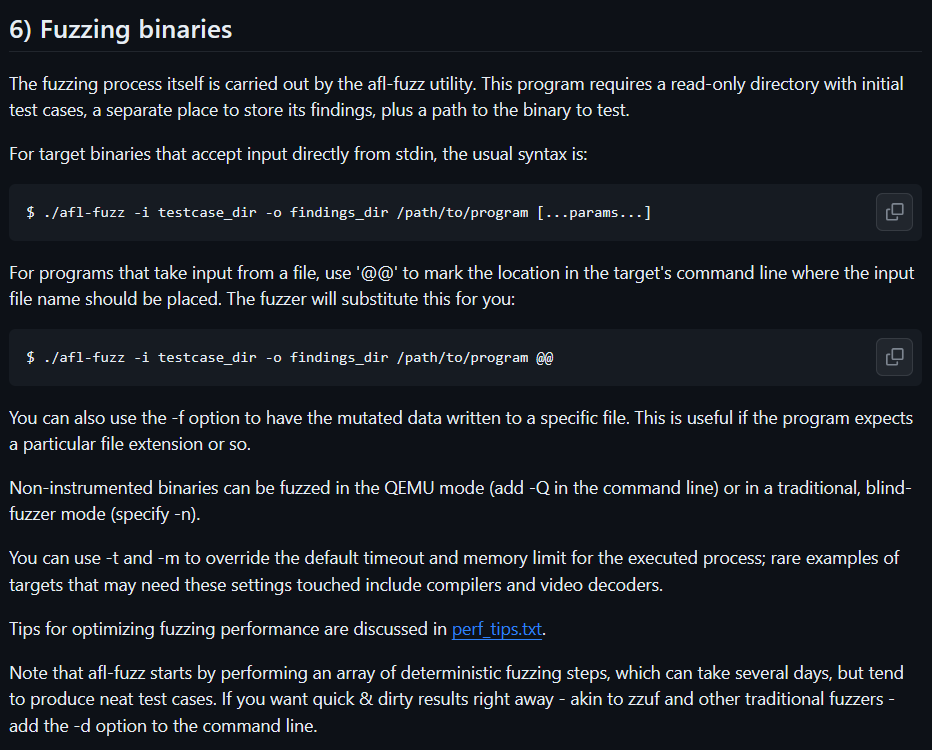
**总结:**使用自动化方法来发现漏洞是现代编译器开发的重要组成部分。由于各种原因，使用现成的模糊测试工具，尤其是AFL，是目前最广泛使用的方法。然而，基于变异的模糊测试最初用于测试二进制格式，而其传统限制了它们对编译器模糊测试的有效性。针对这个问题的大多数解决方案要么给编译器开发人员带来重大负担，要么不够有效，有时两者兼而有之。我们展示了通过借鉴变异测试的思想，可以显著提高基于AFL的编译器模糊测试的效果，而不需要强制开发人员提供语法或字典。我们所提出的两种配置中的一种对我们调查的所有编译器来说表现最好，在实验中有时效果显著（平均找到的漏洞数量比未改进的AFL的最佳版本多两倍以上）。研究团队提出的方法，报告了100多个以前未知的漏洞，这些漏洞已经因此得到修复，在重要的编译器项目中取得了成果。

实验结果：

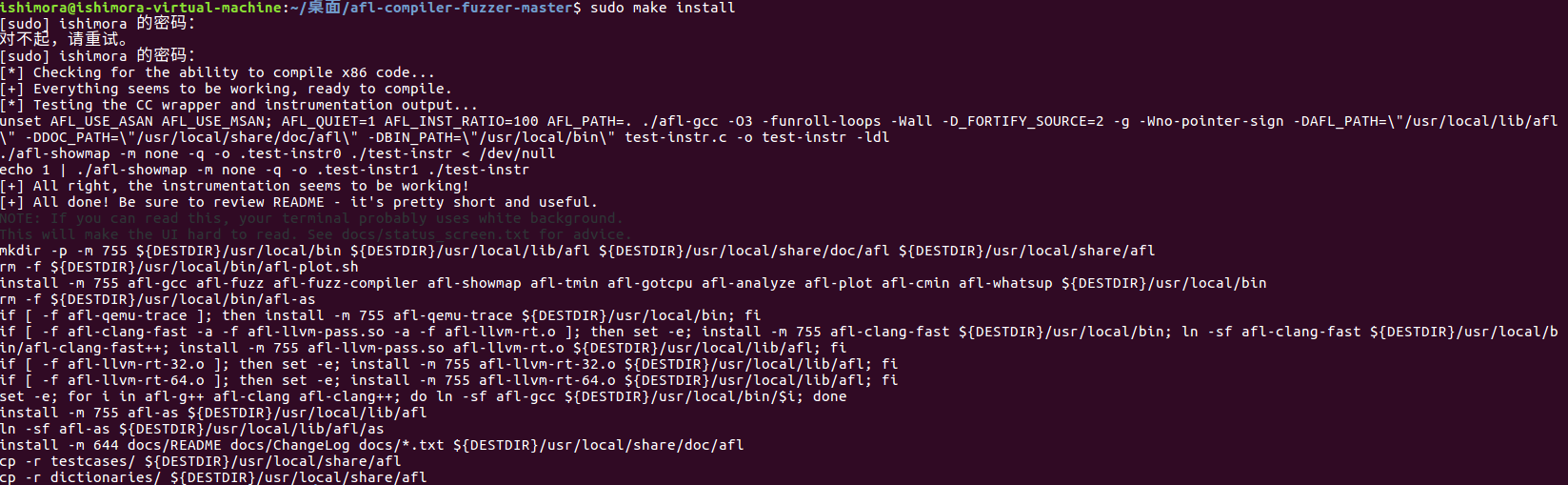
首先从github上下载afl模糊测试器源码



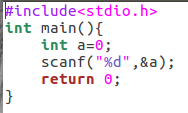
根据github提示进行操作



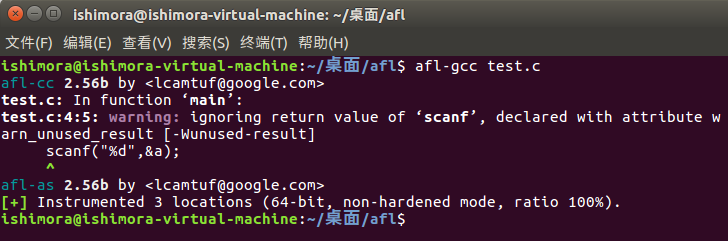
使用sudo make install命令在linux上安装afl模糊测试器



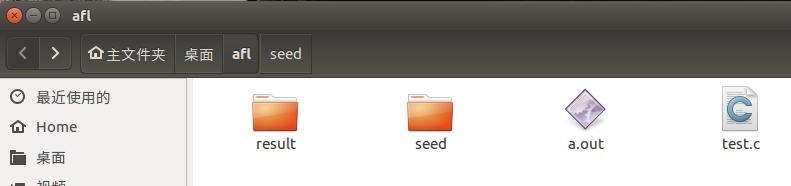
测试文件test.c源代码：

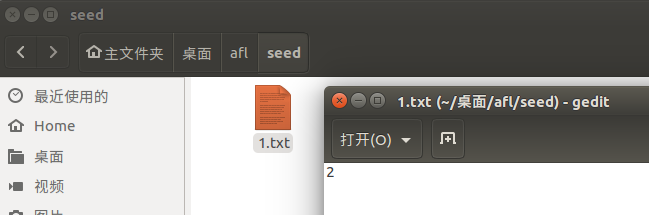


afl-gcc test.c （使用afl-gcc编译源文件）



seed目录下的种子文件：





种子文件中的字符串代表了对进行模糊测试的程序的原始输入（在afl模糊测试过程中，该输入会被不断地从队列中取出，进行变异，执行新的输入，如果是有趣的输入则添加至队列中继续变异）。参考：

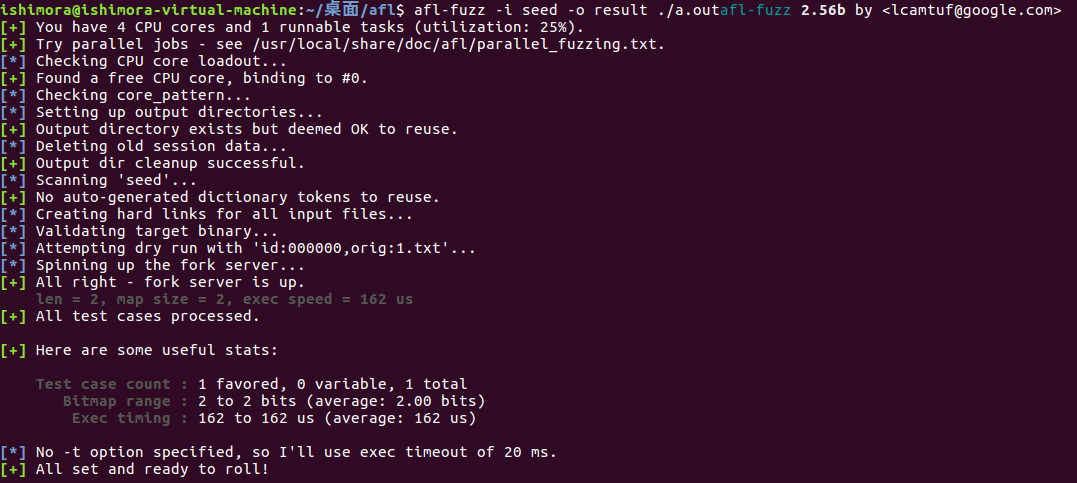
AFL基本模糊测试的内部模糊测试循环策略（算法 ）：

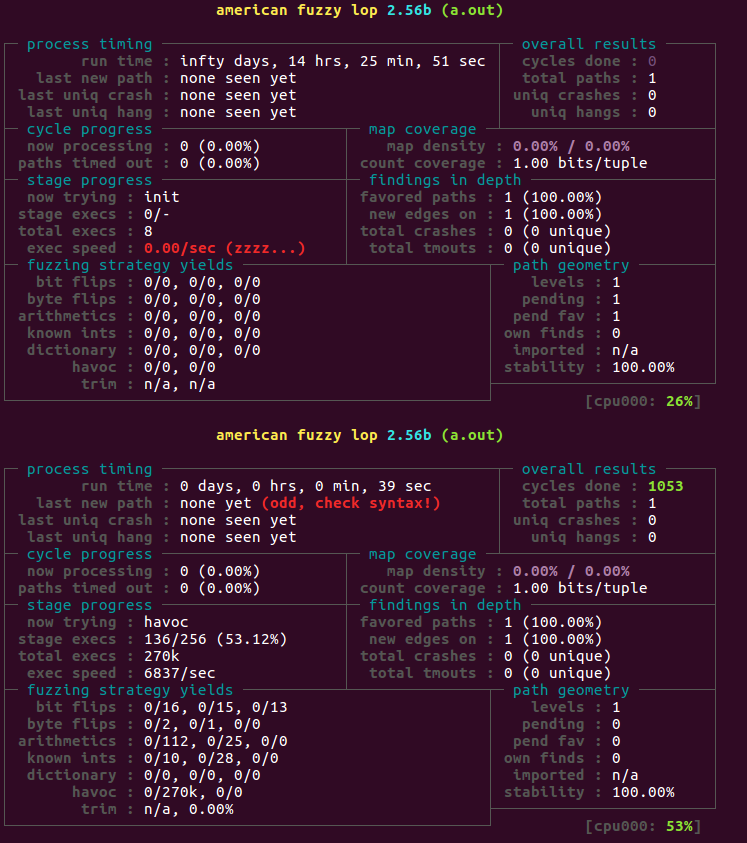
1. 从队列中选择一个输入。

2. 变异该输入以获得新的输入。

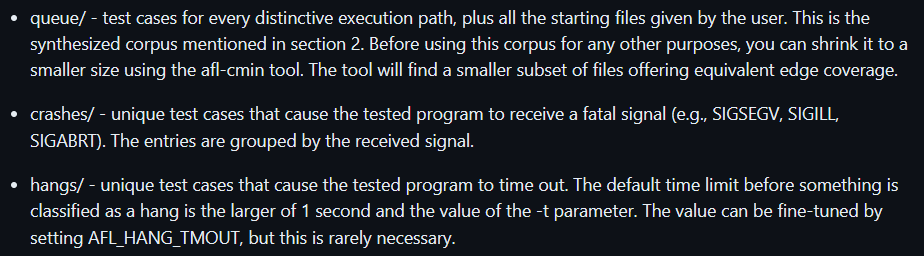
3. 执行新的输入，如果它有趣，则将其添加到队列中。然后从第1步开始重复该过程。

afl-fuzz -i seed -o result ./a.out （执行afl模糊测试，从seed目录中寻找原始输入，将结果存放至result目录）

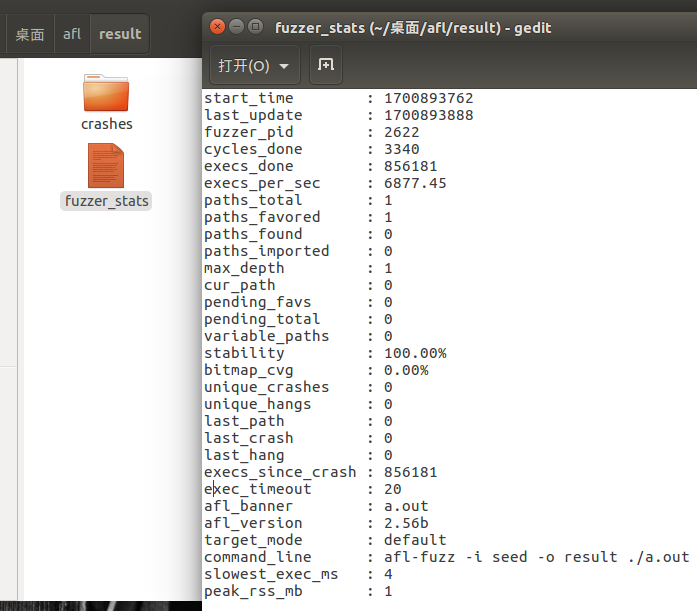




result文件中会有三个文件夹:



result中的fuzzer\_stats显示了当前测试的各项结果数据，包括开始时间，最后一次更新时间，测试器进程id号，完成的测试轮数，每秒执行的次数，路径总数，路径偏好，新路径查询等等。在crashes目录中，还可以找到使程序崩溃的日志

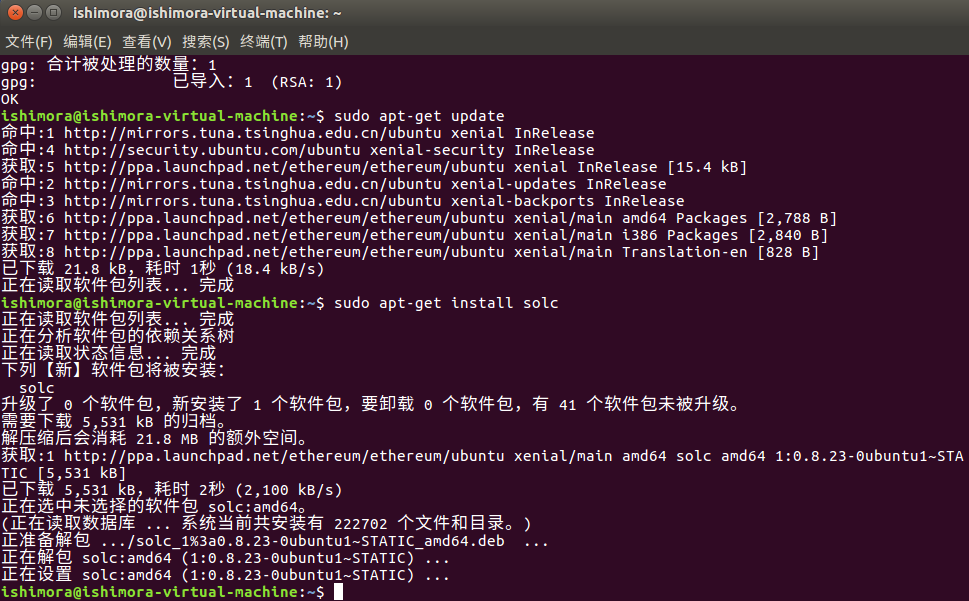


**进行完基本的afl模糊测试以后，进行论文复现，也即复现论文使用afl模糊测试对于Solidity，Move，Fe，Zig编译器的模糊测试工作**

首先从Solidity官网中找到Solidity的二进制安装包版本



按照教程获取solidity二进制稳定版本

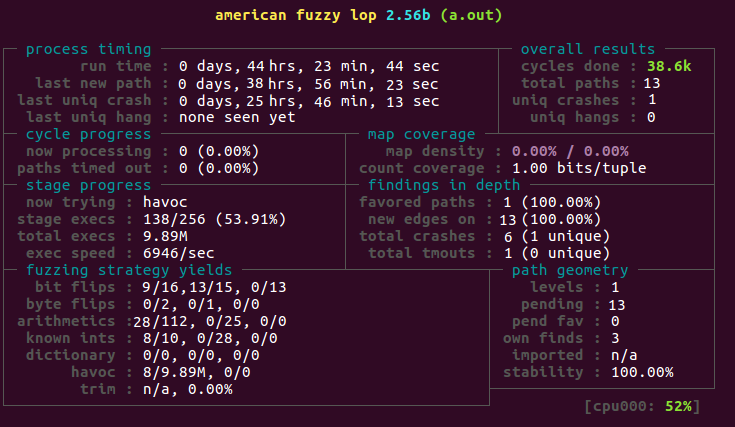


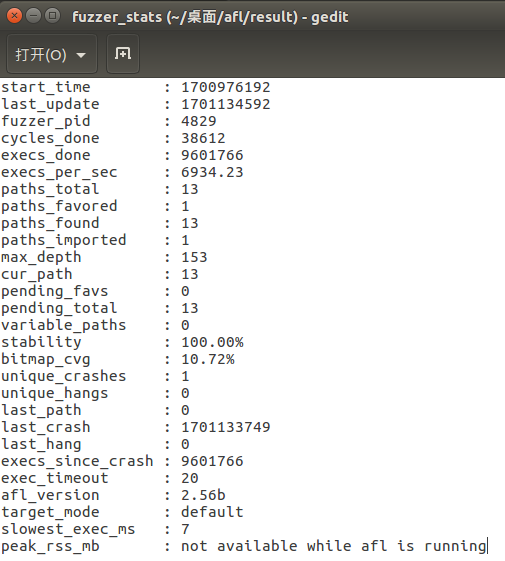
获取以后在build文件夹中得到Solidity的二进制文件，这些文件将能用作模糊测试的输入

使用以下命令对Solidity编译器进行模糊测试,input\_dir中指定用作输入的文件夹，output\_dir指定用作afl-fuzz的输出目录



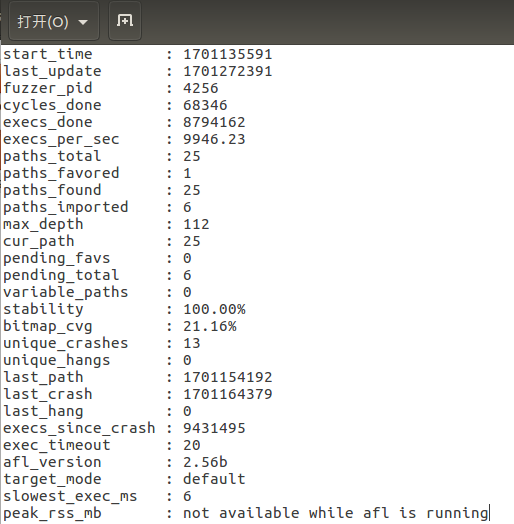
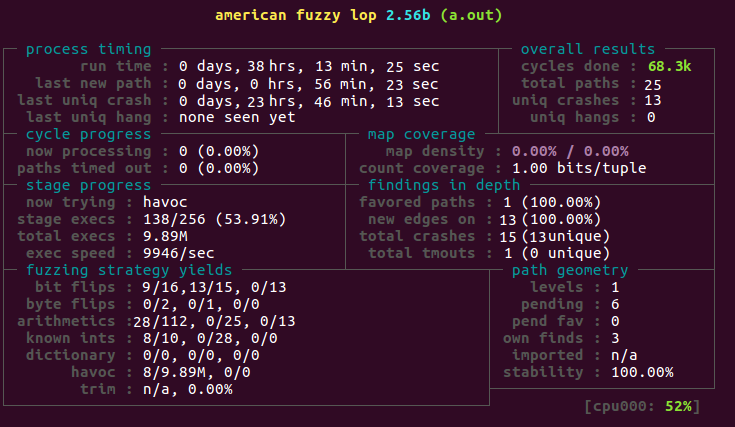
Solidity进行44小时后模糊测试结果:



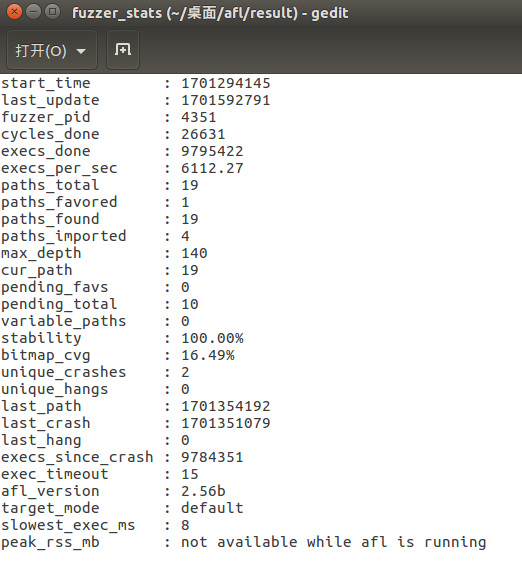
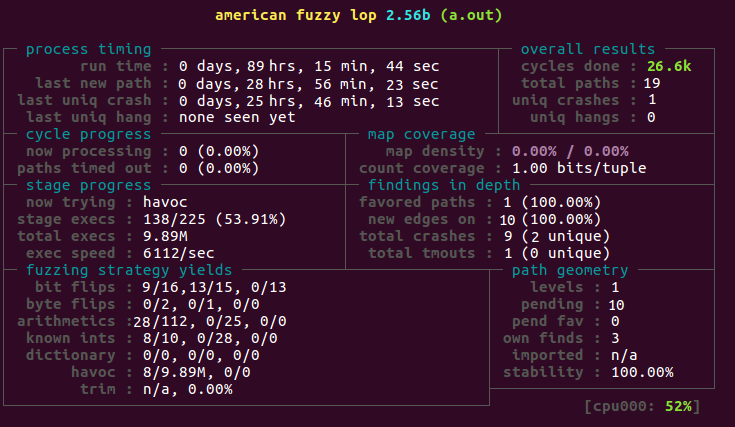


同理对Move、Fe编译器的测试如下：

Move:

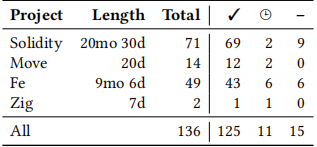


Fe:

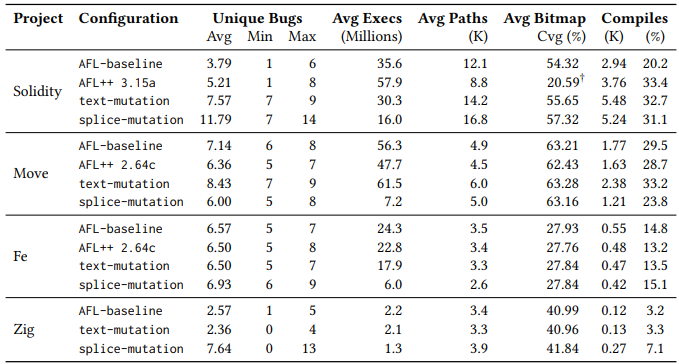


做出统计如下：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Project | Config | Avg Bugs | Avg execs(k) | Avg paths(k) | Avg bitmap cvg(%) | Complies(k) |
| Solidity | Afl-baseline | 1.74 | 96.0 | 13 | 10.72 | 2.16 |
| Move | Afl-baseline | 4.71 | 87.9 | 25 | 21.16 | 6.27 |
| Fe | Afl-baseline | 3.94 | 97.9 | 19 | 16.49 | 0.12 |

（注：原论文中不仅对Solidity、Move、Fe编译器做了测试，还对Zig做出测试，并且对每个编译器都做出了4种不同模糊测试器配置下的测试，不仅如此，基本都是在大于一个星期的测试时间内进行测试:，由于笔者资源和能力有限，只能对前三个编译器进行测试，并且测试时间都无法达到超过100h）

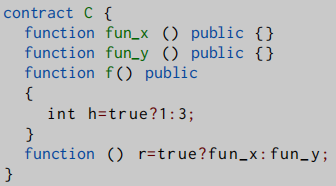
与原论文结果进行比较：



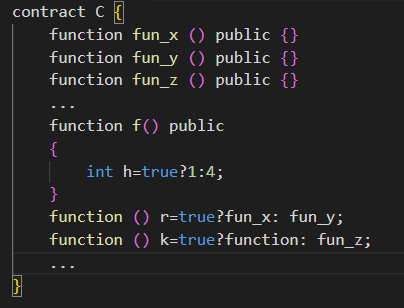
可见除了测试器配置以及编译器种类上不如原论文以外，检测出的平均bug数比原文少很多，推测是编译器已经就这些bug进行了修复，平均运行次数以k为单位要比原文少很多，检测路径时可能出现bug，使得平均路径数与原文的路径相差较大，avg bitmap结果参数也总体比原结果低出许多，总体而言，复现结果不算完整，也谈不上完美。

**技术改进**

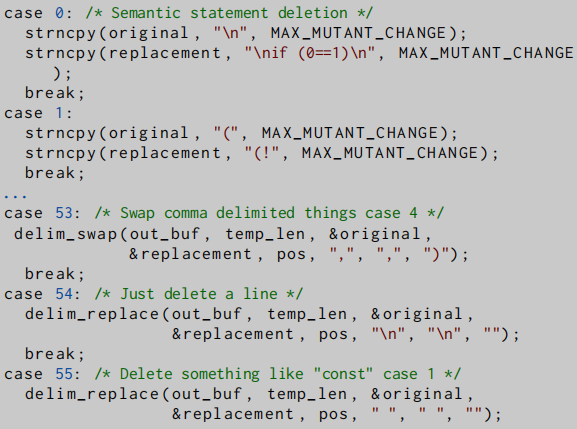
①首先，论文中提出了对于传统模糊测试中的一种新奇的改进：基于改变通用模糊器修改（突变）输入的方式。然后通过从突变测试领域中提取的大量修改来增加这些工具所做的主要基于字节的修改集，这只以可能保留理想propertiesÐ的方式修改代码，比如通过解析器的能力。（下图是由这样的方法生成的这样一个输入，它生成了一个语法格式良好的程序，可以在编译器的优化例程中触发更深层次的行为。）



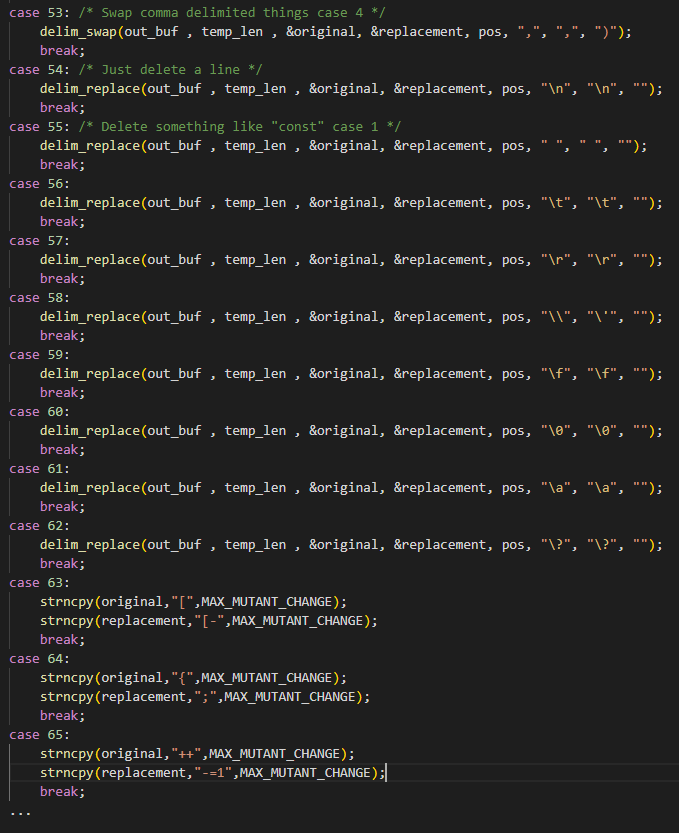
那么，笔者通过对不同突变程序的增加，使得模糊测试器在探索不同的道路时，对于不同道路的选择更丰富和更具有多样性,对于最终的检测结果也更容易查找出编译器的差漏：



②其次，论文还实现了对于各种字符串的随机排列的随机变异，该实现的核心是实现一组突变测试操作符，以便基于突变的模糊器可以将它们应用于输入。有大量关于在突变测试中选择突变操作符的文献，但这些文献集中在识别有助于在测试工作中发现漏洞的操作符。但没有理由相信这表明在模糊化过程中选用了最有用的操作符，而且有一些建议表明，这种方法在任何情况下都不会优于随机选择。因此，基于通用突变体工具提供的操作符集，论文使用了大量适用于各种编程语言的操作符。



那么本人在参考过原论文的字符串随机变异的实现以后，也增加了应当应用随机变异的案例与情况，有利于丰富随机变异以后模糊测试器对于多道路的选择的多样性，成功查找差错的概率应得到有力的提升：



# 总结

在模糊测试编译器时，涉及到编译器优化问题，因此我需要去了解不同计算机体系结构上的指令级并行性。某些优化可能将会依赖于特定的指令集和处理器架构；编译器工作原理与计算机内存模型息息相关。在进行编译器的模糊测试时，我了解到了不同体系结构下的内存层次结构、缓存体系和优化技术对于编译器的优化以及测试都可能会有影响;学习完对编译器的模糊测试以后，我更深入地了解了硬件特性对编译器行为和性能的影响。如不同架构的处理器对于内存访问速度、指令执行效率等方面的影响可能会影响编译器的优化策略。不仅如此，不同的计算机体系结构有不同的指令集架构（如x86、ARM等），这些架构在编译器的工作和代码生成方面可能存在差异。通过模糊测试编译器，我了解了这些差异以及对代码生成和性能等的影响。总而言之，通过复现论文对编译器的新型AFL模糊测试，我更深入地了解到不同计算机体系结构之间的差异，以及这些差异对编译器和程序性能的影响。这种知识可能将让我更具备优化程序、编写更高效代码以及理解底层硬件和软件交互的能力

1. [] Alex Groce, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. 2022. Making no-fuss compiler fuzzing effective. In Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC 2022). Association for Computing Machinery, New York, NY, USA, 194–204. https://doi.org/10.1145/3497776.3517765 [↑](#endnote-ref-0)