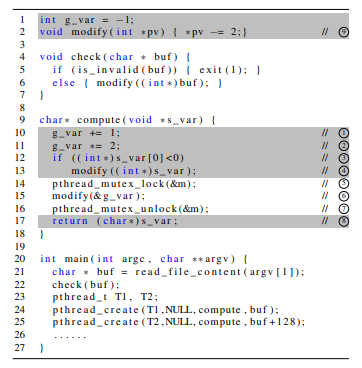
MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs阅读报告

1. 相关背景
   1. 主要问题

与顺序程序相比，多线程程序更容易包含严重的代码缺陷。一方面，线程交错的不确定性可能导致例如数据竞争与死锁的并发错误。这些错误可能导致程序产生异常结果或者被意外挂起。另一方面，一些在特定程序输入与线程交错的前提下产生的错误可能导致并发漏洞，造成例如内存破坏与信息泄漏的后果。

这些多线程相关的漏洞通常隐藏在复杂的程序流中，而现有的大部分模糊测试器都不关注影响多线程程序执行状态的线程交错。因此，主流的模糊测试器都无法很好地检测到多线程软件中存在的漏洞。

* 示例：



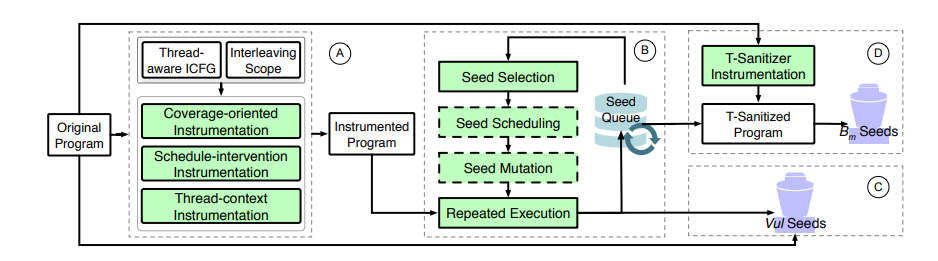
如果有两个线程T1和T2，那么在line10-11处标记为①和②的操作中可能出现以下三种情况：

1. T1:①→T2:①→T2:②→T1:① g\_var=4
2. T1:①→T2:①→T1:②→T2:② g\_var=4
3. T1:①→T1:②→T2:①→T2:② g\_var=2
   1. 解决思路

本文提出了一个面向多线程程序的灰盒模糊测试框架Muzz，采用了三种线程敏感的插桩策略，包括代码覆盖率引导的插桩、线程上下文敏感的插桩以及线程调度插桩。在模糊测试的过程中，这些插桩产生的运行时反馈可以帮助Muzz探索线程交错相关的多线程程序执行状态。此外，本文将这些多线程相关的目标并发错误归为两类：

* 并发漏洞：该类对应的是在多线程环境下产生的内存破坏漏洞，这些漏洞可以在模糊测试的阶段中被发现；
* 并发错误：该类对应的是例如数据竞争与死锁的并发错误。这类错误并不能在模糊测试的阶段被发现，因为这类错误并不一定会导致内存破坏，造成多线程程序的崩溃。本文提出的Muzz通过给例如TSan的并发错误检测工具重放fuzzing过程中生成的模糊测试种子来检测这些错误。

二、方法概述



总共有四个部分：

A：静态线程感知分析引导的插桩

该部分共提出三种插桩策略：

1. 基于覆盖率引导的插桩：通过在可疑的线程交错代码片段中增加更多插桩，记录线程交错相关的多线程程序执行状态覆盖率；
2. 线程上下文敏感的插桩：通过跟踪线程分叉（thread-forks）、线程死锁（thread-locks）、线程解锁（thread-unlocks）、线程连接（thread-joins）等线程函数的上下文来区分不同的线程标识ID。
3. 干预线程调度的插桩：可以动态调整每个线程的优先级。目的是通过干预线程调度使线程的交错情况实现多样化。

B：动态模糊测试

MUZZ优化了种子选择和重复执行，以生成更多与多线程相关的种子。

种子选择上，除了由基于覆盖的插桩提供的覆盖信息外，MUZZ还根据线程上下文工具提供的反馈对那些覆盖新的线程上下文的种子进行优先级排序。

在重复执行种子的过程中， 提高那些能够产生不确定性行为的种子的执行次数，跟踪交错执行状态。

C：漏洞分析

漏洞分析组件实现漏洞分类。

D：暴露并发漏洞

将fuzzing过程中的有意义的种子重放给并发bug检测器（例如TSan等等），试图找出。

1. 具体实现
2. 线程敏感的静态分析

静态分析的目的是为后续插桩等一系列操作提供一些轻量级的信息。

（1）线程敏感的ICFG构建

考虑例如POSIX标准Pthread的线程API的语义，Muzz生成了包含五类多线程信息的ICFG(inter-procedural control flow graph)：

* TFork是例如pthread\_create的线程fork函数调用点的集合；
* TJoin是例如pthread\_join的线程join函数调用点的集合；
* TLock是例如pthread\_mutex\_lock的线程lock函数调用点的集合；
* TUnLock是例如pthread\_mutex\_unlock的线程unlock函数调用点的集合；
* TShareVar是不同线程间共享变量的集合。

（2）可疑的线程交错代码片段

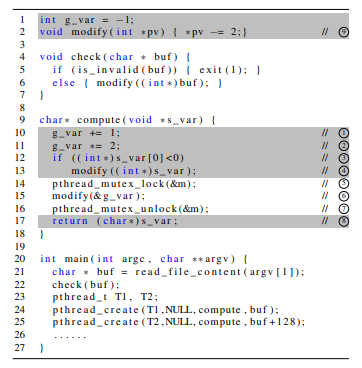
根据三个条件决定其属于可疑的线程交错的代码片段：

C1：语句在TFork调用之后且在TJoin调用之前被执行；（排除与多线程无关代码，例如校验或错误处理代码）

C2：语句在TLock调用之前或者在TUnLock调用之后被执行；（排除被锁保护的语句）

C3：语句至少读写了不同线程间共享的一个变量。（排除只读操作）

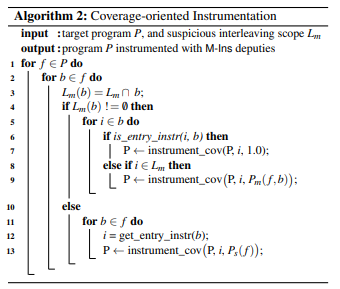
* 示例：



根据C1，可以排除掉3-9、18-27行。根据C2，可以排除掉第15行。根据C3，可以排除掉第14行、第16行。剩下的第1、2、10、11、12、13、17即为可能发生线程切换的位置，将它们归为集合。

1. 基于覆盖引导的插桩

对可疑的线程交错代码片段中的每条指令进行插桩仍然会产生大量的开销，这会极大地影响执行效率。Muzz利用概率选择插桩。



表示函数的cyclomatic complexity的值，该值越大，存在漏洞的概率越大。表示该函数的CFG中边的数量。表示该函数中的基本块的块数。

通常的最优上界为10。所以基础概率计算方法如下：

Line8-9：有些语句在集合中，但此处的线程交错实际上并没有改变共享变量的值。那么这些线程交错也并没有必要去探索。所以对于集合中语句Muzz对每条指令进行概率选择插桩，其概率如下：（其中默认为0.33）

Line11-13：有些语句不在集合中，但仍有插桩的必要。例如：校验语句，以防初始种子都被拒绝但没有反馈。这类语句的插桩概率如下：（其中默认为0.5）

2、线程上下文敏感的插桩

由于代码覆盖率插桩并不考虑线程的ID，Muzz采用线程上下文插桩来区分不同线程ID作为额外的运行时反馈。这些上下文是在TLock、TUnLock与TJoin调用的时候被记录的。

1. 干预线程调度的插桩

该插桩的目的是为了使线程交错实现多样化。Muzz采用例如POSIX的ptherad\_setschedparam等API在TFork调用的时候为每个线程指定一个随机的优先级。

1. 种子选择

Muzz采取的策略是优先选择探索了新的代码覆盖率或者线程上下文的种子。在实际运行中，Muzz将这个问题简化为：在遍历种子队列时，是否对队列最前端的种子t进行突变。

这里对种子进行两方面的判断，has\_new\_mt\_ctx()判断其是否有产生新的线程上下文，has\_new\_trace()判断其是否覆盖到新的路径。

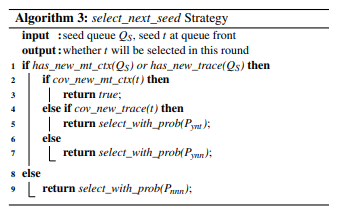
Line1：对种子队列进行上述两方面进行判断；

Line2：最前端的种子t如果产生了新的线程上下文，那么必然选择该种子进行突变；

Line4-5：种子t产生了新的路径，但没有产生新的线程上下文，则该种子有的概率被选择并进行突变。

Line6-7：种子t没有产生新路径，也没有产生新的线程上下文，则该种子有的概率被选择并进行突变。

Line8-9：整个种子队列没有产生新路径，也没有产生新的线程上下文,那么最前端的种子t有的概率被选择并进行突变。



1. 重复运行的次数

由于随机的线程交错，多线程程序可能会表现出不确定的行为。也就是，会有不确定行为的种子。对于这类种子，Muzz则增加其运行次数。因此，Muzz会重复执行模糊测试种子次。

其中为初始运行次数，其值为8。其中为常数，其值为32。是该种子在之前的执行中探索过的线程上下文的个数。

AFL中也有类似的策略，但用的公式不同，在AFL中：

当次运行中没有不确定行为时，为0，反之为1。

1. 评估
2. 实验设置

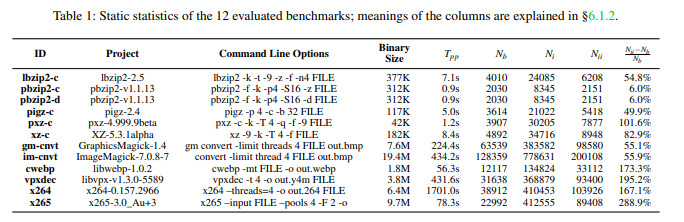
作者在评估中使用了四个灰盒模糊测试框架：Muzz、MAFL、AFL、MOpt。其中MAFL是Muzz的变体，其采用了AFL而非Muzz的代码覆盖率导向的插桩，其他插桩策略和后续方法与Muzz相同。

测试对象：pigz、lbzip2、pbzip2、xz、pxz、ImageMagick、GraphicsMagick 、libvpx、libwebp、x264、x265。

实验在4个Intel(R) Xeon(R) Platinum 8151 CPU@3.40GHz 28核工作站上进行，每个工作站运行64位Ubuntu 18.04 LTS。

在fuzzing期间，我们以24小时的时间预算，对所有12个基准测试程序运行上述每个fuzzers 6次。

因为所有被评估的程序都设置为使用4个工作线程运行，并且这些线程被映射到不同的内核。



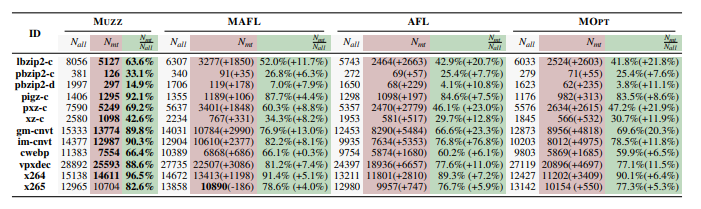
:静态分析预处理的时间 :基本块的数量

:指令总数 :Muzz插桩的指令数

:Muzz插桩比AFL（每个基本块均等插桩）多出的指令占比

2、实验面向三个研究问题

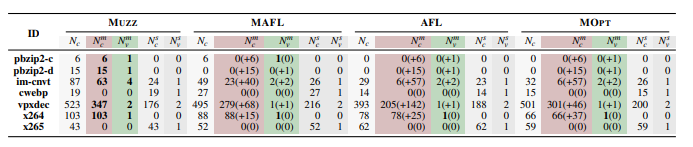
* RQ1：Muzz可以生成更加有效的模糊测试种子来探索更多的多线程程序执行状态吗？



:所有能产生不同路径的种子 :能够运行多线程上下文的种子

MUZZ在增加多线程相关种子的数量和百分比方面具有优势。所提出的三种线程感知插桩和动态运行时的策略有利于种子的生成。

* RQ2：Muzz检测并发漏洞的能力如何？



:proof-of-crash(POC)文件的数量

:与多线程相关的POC文件数量 :并发漏洞数量

:与多线程无关的POC文件数量 :非并发的漏洞的数量

MUZZ在执行更多与多线程相关的崩溃状态和检测并发漏洞方面表现出优越性。

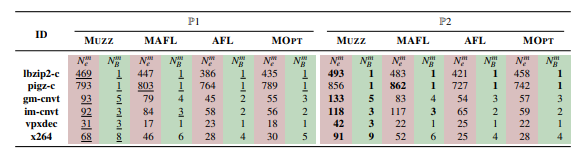
* RQ3：在检测并发错误的方面，Muzz生成的模糊测试种子的效果如何？

通过重放Muzz生成的模糊测试种子并使用例如TSan、Helgrind与UFO的并发错误检测工具来检测这些错误，以在两个小时的重放时间中尽可能地检测出更多的并发错误。

重放时间是固定的，但是提供了两种重放模式。

P1：队列中的每个种子循环执行一次，直到达到时间预算。

P2：重复执行依赖于Nc:每个种子在每轮循环中连续执行次。对于AFL生成的多线程相关种子，每回合重放5次;



比较MUZZ、MAFL、AFL和MOPT上的两种重放模式P1和P2的并发冲突()和并发bug()。

MUZZ在检测并发bug方面优于竞争对手，在fuzzing过程中计算的数值Nc有助于揭示这些bug。

1. 总结

本文提出了一种基于GBFs的模糊多线程程序线程感知种子生成技术MUZZ。采用了三种新的插桩策略，进而区分由线程交错引入的执行状态。根据这些插桩提供的反馈，MUZZ优化了动态策略，以强调不同类型的多线程环境。实验表明，MUZZ在生成有价值的种子、检测并发漏洞以及揭示并发bug方面优于其他灰盒模糊器，如AFL和MOPT。

针对特定场景特定情况下实现插桩的优化，结合TSan等工具在fuzzing的基础上试图找到更多漏洞的方法是本文较为新颖的点。