Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing阅读报告

一、相关背景

1. 基于覆盖率引导的模糊测试工具

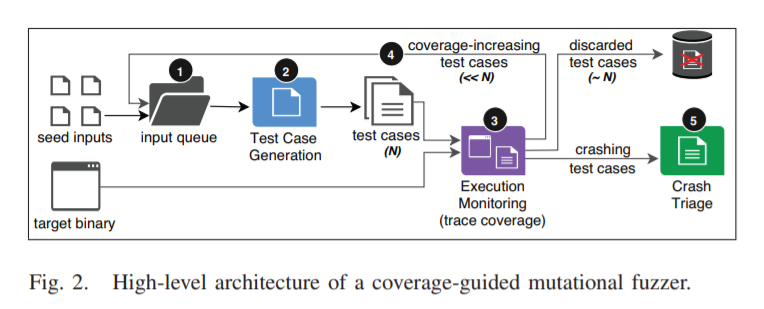
基于覆盖率引导的模糊测试工具需要通过测试用例在代码中的覆盖率的反馈来进行进一步测试用例的突变，而代码覆盖信息的获取需要通过插桩操作来完成。

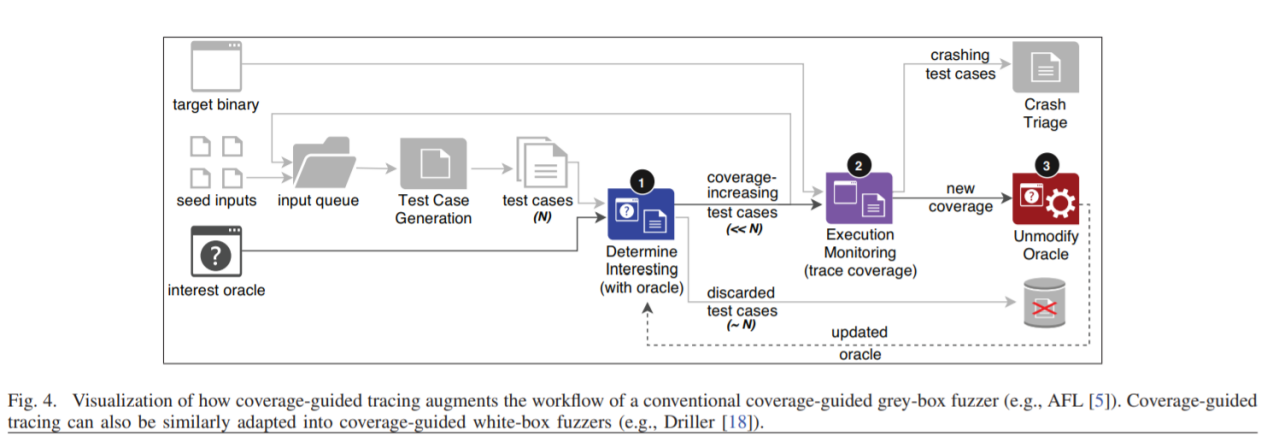
1. 常规的插桩、跟踪方法及其局限性
2. 全插桩、全跟踪：对每一个基本块都进行插桩操作，通过这种方法进行插桩虽然能够覆盖到所有的代码区域，获取完整的代码覆盖信息，但是由于在测试用例集中能够产生新覆盖的测试用例的占比不高，因此该方法会获取到许多无效的跟踪信息，并且每一个测试用例的跟踪过程中都需要执行程序运行本身以外的插桩代码，这将使得模糊测试工具的开销极大。
3. 随机插桩、随机跟踪：在所有基本块中随机选择部分基本块进行插桩操作，显然通过这种方法进行插桩会丢失部分的代码覆盖信息。

3、以覆盖率为指导进行跟踪

本文提出以覆盖率为指导的跟踪方法目的是要在不丢失代码覆盖信息的前提下减少执行跟踪操作的测试用例，类似于一个跟踪过滤器，只对那些会产生新的覆盖信息的测试用例进行后续跟踪。

1. 方法概述





与常规的以覆盖率引导的模糊测试工具（Fig.2.）相比，本文提出的方法（Fig.4.）多了以下几个步骤：

1. Determine interesting（with oracle）：

insteresting oracle是修改后的二进制文件，与原始的二进制文件不同的是，它是在每个未覆盖的基本块的开始处插入一个中断信号后生成的二进制文件。因此这一步就是要找到那些未覆盖的基本块，插入中断信号，在第一次迭代之前，所有的基本块都未被覆盖，因此都会被插入这样的中断信号。

1. Execution Monitoring：

如果一个测试用例产生了新的覆盖信息，它就会触发这个基本块开始处的中断信号。那么就对这个种子进行完整的跟踪。

1. Unmodify Oracle：

对于该测试用例产生的新的覆盖的基本块，将其的中断信号撤销掉，以表示其已经被覆盖过，后续其他种子就不会在这里再次产生中断信号，从而产生无效的跟踪信息。

1. 算法实现

UnTracer是在AFL的基础上改进的，插桩产生两个单独的目标二进制文件，一个是用于辨别能够产生新覆盖的测试用例，一个是用于跟踪识别新的覆盖信息。如算法1所示：

line1：完成AFL的一些初始设置

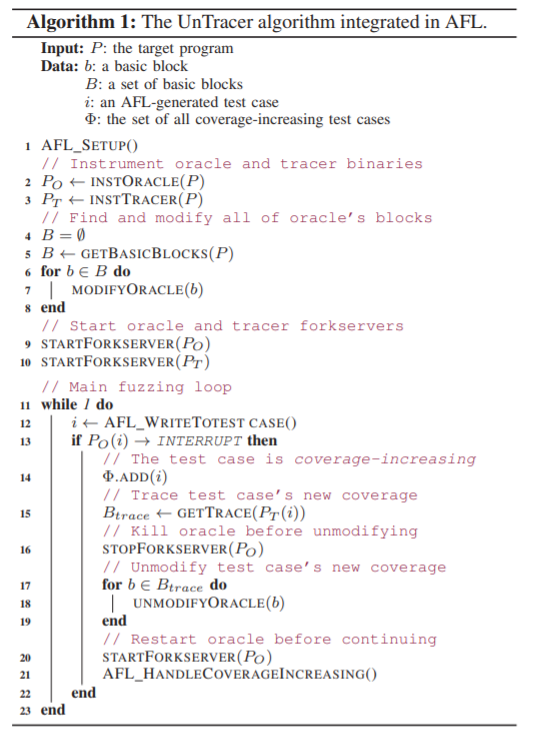
line2-3：对原始的目标二进制文件进行修改，产生（用于插入中断信号的目标二进制文件）、（用于实现跟踪的目标二进制文件）

line4-8：对oracle文件中所有的基本块插入中断信号

line9-10：开始oracle和tracer的forkserver

line13-19：进行测试用例的输入测试，如果输入的测试用例触发了中断信号，将该测试用例加入可产生新的覆盖信息的测试用例集中，对其进行跟踪，对新覆盖到的基本块撤销其中断信号。

line20-21：更新oracle，准备下一次迭代。



1. 具体实现
2. Forkserver Instrumentation
3. Interest Oracle Binary

Oracle就是在原始的目标二进制文件中的基本块中插入中断信号以实现产生新覆盖的测试用例的识别，而要实现这个插入就必须提前知道整个二进制文件的基本块地址，因此这里采用的是静态分析的方法来实现。

另外，这里采用的中断信号是SIGTRAP。因为对于中断信号而言有两个要求：一个是要避免与其他的关键信号产生冲突，另一个是中断指令的大小不能超过任何候选基本块的大小（因为采用的是替换源代码而不是增加代码的方式插入信号）；而SIGTRAP正好满足这两点要求，其二进制表示仅为1个字节。

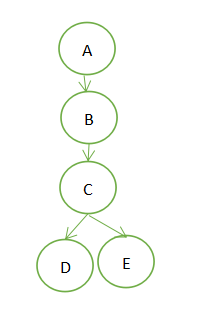
1. Tracer Binary（a coverage tracing-instrumented version of the target binary)

利用Dyninst静态地使用forkserver来检测跟踪程序，在每个基本块中插入覆盖回调（coverage callback）用于覆盖跟踪。

这里存在两个问题会使得Tracer的开销增大：

①写：多次记录某个单独的基本块（通常出现在循环中）使得该部分开销增大。

②读：以下图为例，例如第一次产生的测试用例1产生的覆盖信息是ABCD，那么在读取覆盖信息准备撤销中断信号的时候读取出来的基本块就是ABCD，接着第二次产生的测试用例2产生的覆盖信息为ABCE时，那么读取的基本块就是ABCE，可以看到事实上在第二次的测试用例中ABC已经不是新的覆盖信息了。因此类似的反复读取也会使得该部分的开销增大。



所以这里做了一个优化：初始化一个全局的哈希结构图来跟踪所有被覆盖的基本块信息。在执行基本块时，回调使用哈希映射来查找，如果时已经被覆盖的基本块则不做操作，如果是之前未被记录的基本块则更新其跟踪日志和哈希图。

1. Unmodifying Oracle

对于新的覆盖的基本块执行unmodify oracle的操作，撤销其中的中断信号。

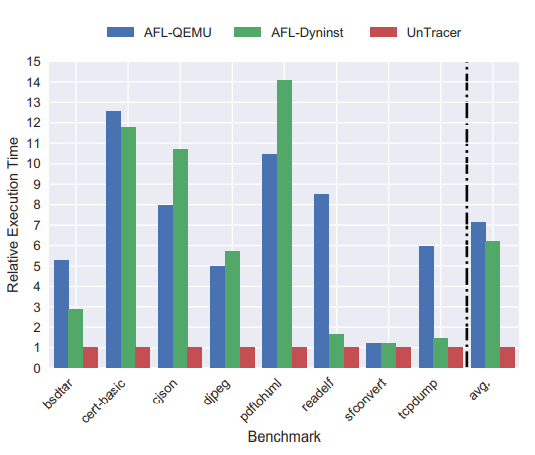
1. 实验评估

实验评估主要围绕以下三个问题：UnTracer的性能如何？主要是由哪些因素在产生开销？模糊测试时产生的开销和能够产生新的覆盖信息的测试用例占比的关系是如何？

对比的工具主要包括一些采用白盒/黑盒进行跟踪的模糊测试工具，AFL-Clang（white-box）、AFL-QEMU（black-box dynamically-instrumented）、AFL-Dyninst（black-box statically-instrumented）。使用QEMU作为基础跟踪器（因为我们的重点是黑盒跟踪）为每个基准生成五个数据集。

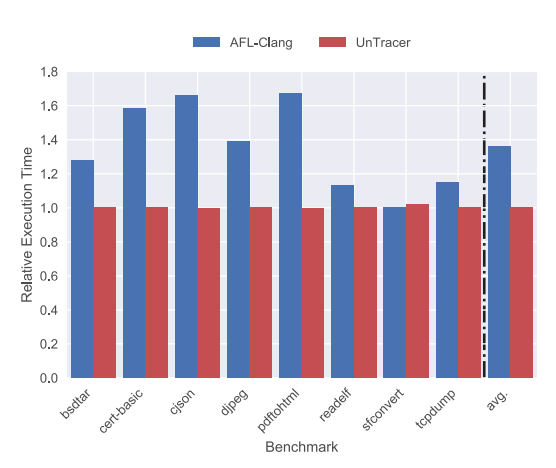
将UnTracer的性能与三种流行的与覆盖率无关的跟踪方法进行比较。 首先探讨两个黑盒二进制模糊跟踪器的性能：AFL-QEMU（动态）和AFLDyninst（静态）。 其次，将UnTracer的性能与白盒二进制模糊跟踪器AFLClang（静态汇编器跟踪）进行比较。

（1）黑盒二进制跟踪



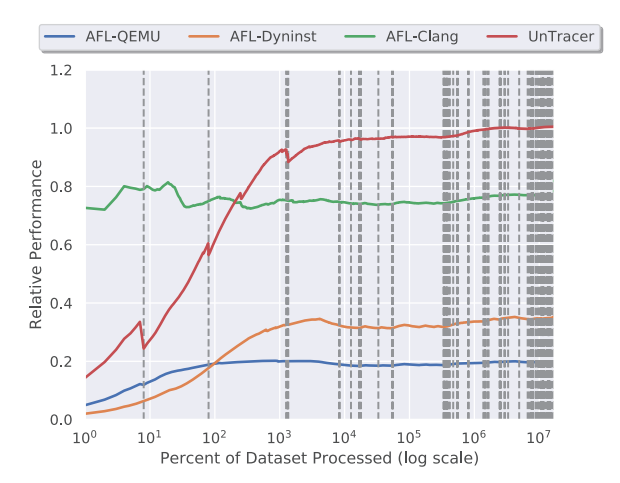
可以看到，在所有基准测试中，UnTracer的平均相对执行时间为1.003（开销为0.3％），而AFL-QEMU和AFL-Dyninst的平均相对执行时间分别为7.12（开销为612％）和6.18（开销为518％）。结果表明，UnTracer将跟踪黑盒二进制文件的开销最多减少了四个数量级。

（2）白盒二进制跟踪



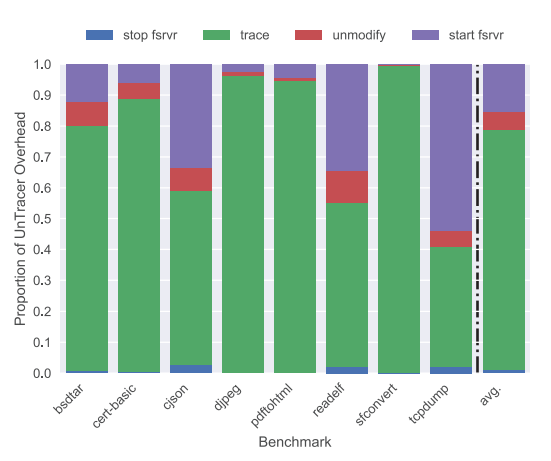
如图显示了UnTracer的基准开销，以及AFL的白盒二进制跟踪器AFL-Clang。在所有八个基准测试中，AFL-Clang的平均相对执行时间为1.36（36％的开销），而UnTracer的平均执行时间为1.003（0.3％的开销）（每个跟踪器的平均RSD小于4％）。

（3）



该图显示了UnTracer的开销如何随着时间的推移以及覆盖率增加的测试用例的变化而变化。在测试过程的早期，覆盖率增加的测试用例的速度就足够高，足以降低UnTracer的性能。随着时间的推移，增加覆盖范围的单个测试案例的影响不大，UnTracer的开销逐渐接近0％。此结果还表明，可以建立一个覆盖指导的混合跟踪模型，该模型始终跟踪初始测试用例，直到覆盖率增加的测试用例的速率减小到UnTracer变得有用为止。

（4）部分开销评估



该图显示了相对于总开销，所有四个组件的执行时间的细分。 该图显示，UnTracer开销的两个最大组成部分是覆盖范围跟踪和forkserver重新启动。

（5）混合模糊测试

例如，可以设定一个阈值，当可以产生新覆盖的测试用例占比大于该阈值则采用全跟踪的方式，如果小于该阈值则采用覆盖引导跟踪的方式。

1. 总结

本文提出的方法基于覆盖率增加的测试用例在模糊测试中占比少的情况，它可以通过修改目标二进制文件来使它们在测试用例产生新覆盖率时自动报告。显著地减少了模糊测试工具耗费在跟踪上的时间，突出了识别和利用模糊固有的不对称性的潜在优势。模糊测试依赖于执行许多测试用例，以期找到覆盖范围增加或产生崩溃的小子集，是一个比较有意思的改进方向。但是在具体实现方面的一些执行效率等方面还可以继续改善以提高执行速度和执行效率。