Evaluating Fuzz Testing

ABSTRACT

模糊测试在发现真实软件中的安全关键错误方面取得了巨大的成功。最近，研究人员致力于设计新的模糊技术、策略和算法。这样的新想法主要是通过实验来评估的，所以一个重要的问题是：需要什么样的实验装置来产生可信的结果？我们调查了最近的研究文献，并评估了32篇模糊论文的实验评价。我们在每次评估中都发现了问题。然后，我们使用一个现有的模糊器进行了我们自己的广泛的实验评估。我们的结果表明，我们在现有的实验评估中发现的普遍问题确实可以转化为实际的错误或误导性评估。最后，我们提出了一些指导原则，希望这些指导原则将有助于改进模糊测试算法的实验评估，使报告的结果更加可靠。

1 INTRODUCTION

fuzz tester（或fuzzer）是一种工具，它迭代地、随机地生成用于测试目标程序的输入。尽管与更复杂的工具（包括SMT求解器、符号执行和静态分析）相比显得“幼稚”，但模糊器还是非常有效。例如，流行的fuzzer AFL已经被用来在流行程序中发现数百个bug[1]。将AFL与符号执行者angr进行比较，AFL发现在同一个语料库中，在24小时内多找出76%的错误（68对16）。fuzzers的成功使其成为一个热门的研究课题。

为什么我们认为绒毛起作用？虽然新思想的灵感可以从数学分析中获得，但模糊器主要是通过实验来评估的。当研究人员开发一种新的模糊算法（称之为a）时，他们必须从经验上证明它比现状更具优势。为此，他们必须选择：

•一个令人信服的基线模糊器B，可与之进行比较；

•基准套件目标程序样本；

•当a和B在基准测试套件上运行时测量的性能指标；理想情况下，这是通过崩溃输入识别的（可能可利用的）错误数量；

•一组有意义的配置参数，例如开始模糊化的种子文件，以及模糊化运行的超时时间（即持续时间）。

评估还应该考虑到模糊的基本随机性：由于使用随机性，在目标程序上运行的每个模糊可能产生不同于上一个的结果。因此，评估应该测量足够多的试验，以抽样代表模糊器性能的总体分布，使用统计测试[38]来确定a的测量改进比B的测量改进是真实的，而不是偶然的。

不执行这些步骤中的一个，或者在执行时不遵循建议的实践，可能会导致误导或不正确的结论。这样的结论浪费了从业者的时间，他们可能会从使用替代方法或配置中获得更多利益。他们还浪费了研究人员的时间，他们基于对评估参数的任意调整做出了过于强烈的假设。

我们研究了32篇最近发表的关于模糊测试的论文（见表1），这些论文通过阅读顶级会议记录和其他高质量的会议场所进行定位，并研究了它们的实验评估。我们发现没有一个模糊测试评估能够正确地执行上述所有步骤（尽管有些步骤很接近）。这在理论上是个坏消息，在进行了5万多个CPU小时的实验之后，我们认为在实践中也是个坏消息。以AFLFast[6]（as A）和AFL（as baseline B）为基础，对其性能进行了多种测试。我们选择了AFLFast，因为它是最新的技术进步；它的代码是公开的；我们相信我们有能力重新运行作者在自己的评估中描述的实验，并通过改变原始实验者没有的参数来扩展这些实验。这一选择也受到文献中AFL重要性的驱动：在我们检查的32篇论文中，有14篇在评估时使用AFL作为基线。我们针对三个binutils程序（nm、objdump和cxxfilt）和两个图像处理程序（gif2png和FFmpeg），它们用于先前的模糊评估[9、44、45、55、58]。我们发现，偏离上述配方的实验很容易导致得出错误的结论，原因如下：

同一配置下的模糊性能可能因运行而异。因此，比较单次运行

三分之二的考卷似乎没有给出完整的图片。例如，在nm上，一次AFL运行发现超过1200个崩溃输入，而一次AFLFast运行发现大约800个。然而，对比30次的中位数，却发现了一个不同的情况：（AFL）发生了400起车祸，（AFLFast）则接近1250起。然而，仅仅比较平均值仍然不够：我们发现，在某些情况下，通过统计测试，在性能上的明显差异在统计学上并不显著。

模糊性能在运行过程中可能会有所不同。这意味着，短时间的超时（如11个paper所用，少于5或6小时）可能会造成误导。例如，当使用空种子时，AFL发现13小时后gif2png没有崩溃，而AFLFast发现了近40个。但24小时后，AFL发现了39个，AFLFast发现了52个。当使用非空种子集时，在nm上，AFL在6小时的表现优于AFLFast，具有统计学意义，但在24小时后，趋势逆转。

我们同样发现了基于所用种子的实质性性能变化；例如，对于空种子AFLFast，在nm中发现了1000多个崩溃，但是对于小的非空种子，它只发现了24个，这在统计上与AFL发现的23个不可区分。然而，大多数论文对种子的选择都是漫不经心的，显然是假设任何种子都能同样有效，而没有提供细节。

谈到性能度量，我们检查的32篇论文中有14篇使用代码覆盖率来评估模糊效果。直观地说，覆盖更多的代码与发现更多的bug（19，30）相关，因此似乎值得这么做。但是相关性可能很弱[28]，所以最好直接测量发现的bug数量。然而，只有大约1家报纸采用了这种直接测量方法。大多数4篇论文都计算了发现的崩溃输入的数量，然后应用了一个启发式过程，试图消除触发相同bug的重复输入（为该bug保留一个“唯一”的输入）。最流行的两种启发式方法是AFL的覆盖率分布（7篇论文使用）和（模糊的）堆栈散列（36篇论文使用）。不幸的是，有理由相信这些重复数据消除启发式方法是无效的。

在另一个实验中，我们计算了一部分基本真理。我们对cxxfilt应用了从模糊版本到现在的所有补丁。我们将一个特定的补丁导致的所有输入组合在一起，然后优雅地退出[11]，确认这个补丁代表了一个单一的概念性错误修复。我们发现，所有57142个崩溃输入被认为是“唯一的”覆盖概况由9个不同的补丁解决。这代表了对错误数量的戏剧性的高估。最终，虽然AFLFast发现了比AFL更多的“独特”崩溃输入，但它在给定运行中发现更多独特错误的可能性略高。

堆栈散列做得更好，但仍然过多计算错误。而不是错误映射到，比如说在一个给定的试验中，500个AFL覆盖唯一的崩溃，它将映射到平均46个堆栈散列。堆栈散列也会受到错误否定的影响：大约16%的一个bug崩溃的散列是由另一个bug崩溃共享的。在五种情况下，只有一次崩溃发现了一个不同的错误，而该崩溃具有非唯一散列，这意味着“重复数据消除”将删除不同错误的证据。

这项实验是同类实验中最重要的一项，它表明依靠启发式方法来评估性能是不明智的。一个更好的方法是直接测量地面的真实性，就像我们在上面所做的那样，通过评估已知虫子的模糊器，或者使用合成套件，比如CGC[14]或LAVA[16]，就像我们研究的6篇论文所做的那样。（其他8篇论文部分考虑了基本事实，但通常作为“案例研究”与使用堆栈散列或覆盖率配置文件消除重复输入的一般声明一起进行。）

总的来说，模糊性能可能会随着目标程序的不同而变化，因此在一个多样的、有代表性的基准套件上进行评估是很重要的。在我们的实验中，我们发现AFLFast在binutils程序上的性能通常优于AFL（当使用空种子时，基本上与最初发布的结果相匹配），但在图像处理程序上没有提供统计上显著的优势。如果将这些项目纳入评估，读者可能会对其优势得出更微妙的结论。一般来说，很少有论文使用通用的、多样的基准测试套件；大约有6篇论文使用CGC或LAVA-M，2篇讨论了收集真实程序的方法，而其余的论文使用了一些精选的程序，这些选择在论文中几乎没有重叠。评估中使用的真实程序的中位数是7，最常用的程序（binutils）只有4篇论文共享（考虑版本时没有重叠）。因此，个人评价可能会在内部产生误导性的结论，而且结果很难跨论文进行比较。

我们的研究（在第3节中概述）表明，在模糊化方面有意义的科学进展要求算法改进的主张得到更可靠证据的支持。我们所看到的32篇论文中的每一个评价都缺乏这方面的一些重要方面。本文提出了今后论文评价应遵循的一些明确的指导原则。特别是，研究人员应该进行多个试验并使用统计试验（第4节）；他们应该评估不同的种子（第5节），并且应该考虑更长的（24小时对5小时）超时（第6节）；他们应该评估使用基本事实而不是“独特崩溃”等启发式方法的错误发现性能（第4节7条）。最后，我们主张建立和采用一个良好的模糊基准，并勾画出它可能的样子。手工选择一些特定的目标，并将它们从一张纸变为另一张纸的做法是有问题的（第8节）。一个设计良好且一致同意的基准可以解决这个问题。我们还发现了其他值得研究的问题，包括建立更好的重复数据消除启发法（最近感兴趣的一个主题[42，51]），以及使用相关领域的算法思想，如SAT求解。

2 BACKGROUND

有许多不同的动态分析可以被描述为“模糊”，模糊器的一个统一特征是它们操作并产生具体的输入。否则，fuzzer可能被实例化为许多不同的设计选择和许多不同的参数设置。在本节中，我们概述了fuzzers的基本工作原理，然后介绍了32篇最近发表的论文的进展

这是模糊评价研究的核心。

初始化一个新的种子语料库。

isDone：根据实现目标的进度或超时来确定模糊化是否应该停止。

选择：从队列中至少选择一个候选种子进行突变。

变异：从至少一个种子和迄今为止对程序的任何观察，产生一个新的候选种子。评估：评估程序中的种子以生成观察结果。

干扰：确定对变异种子进行评估所产生的观察结果是否表明应保留输入。

**Fuzzing Procedure**

大多数现代模糊器都遵循图1所示的过程。这个过程首先选择一个“种子”输入的语料库来测试目标程序。然后，fuzzer重复地对这些输入进行变异，并评估被测程序。如果结果产生“有趣”的行为，fuzzer会保留变异的输入以备将来使用，并记录观察到的内容。最终，fuzzer会停止，要么是因为达到了某个特定的目标（例如，找到某种bug），要么是因为达到了超时。

不同的fuzzer在运行被测程序时记录不同的观察结果。在“黑盒”模糊器中，只需观察一次：程序是否崩溃。在“灰盒”模糊化中，观察还包括关于执行的中间信息，例如，执行期间所采取的分支，由直接按顺序执行的基本块标识符对确定。“白盒”模糊器可以利用应用程序源（或二进制）代码的语义进行观察和修改，可能涉及复杂的推理。收集额外的观察结果会增加开销。不同的fuzzer做出不同的选择，希望用更高的开销换取更好的bug发现效率。

通常，fuzzer的最终目标是生成导致程序崩溃的输入。在一些fuzzer配置中，isDone会检查队列是否有任何崩溃，如果有，它会中断循环。其他fuzzer配置试图收集尽可能多的不同崩溃，因此不会在第一次崩溃后停止。例如，默认情况下，libfuzzer[34]在发现崩溃时将停止，而AFL将继续并尝试发现不同的崩溃。其他类型的观察也是可取的，例如更长的运行时间，可能表明存在算法复杂性漏洞[41]。在任何这些情况下，模糊器的输出都是一些具体的输入和配置，可以从模糊器外部使用它们来重现观察结果。这允许软件开发人员确认、重现和调试问题。

**2.2 Recent Advances in Fuzzing**

模糊测试的有效性使其成为一个活跃的研究领域。通过文献检索，我们发现在2012年至2018年间发表的32篇论文提出并研究了核心模糊算法的各个部分的改进；自2016年以来，我们审查的32篇论文中有25篇发表。为了找到这些论文，我们从10篇发表在顶级安全场所的高影响力模糊论文开始。然后我们在这些论文中来回寻找引文。作为一项明智的检查，我们还对2012年以来发表的论文的标题和摘要进行了关键词搜索。最后，基于目标域对相关度进行判断，并提出了改进方案，对不适合的论文进行过滤。

表1按时间顺序列出了这些论文。在这里，我们简要地总结了这些论文的主题，按照它们最显著的改进目标的模糊化过程的一部分来组织。最后，我们感兴趣的是这些论文如何评价他们声称的改进，下一节将对此进行更多的讨论。

initSeedCorpus Skyfire[53]和Orthrus[49]建议通过对程序进行预先分析，以引导用于创建语料库和协助突变者的信息，从而改进初始种子选择。QuickFuzz[20，21]允许通过使用语法来指定有效或有趣的输入的结构来进行种子生成。DIFUZE进行了一次前期静态分析，以在模糊化之前识别设备驱动程序的输入结构[13]。

变异。symfunz[9]使用符号执行器来确定要变异的种子的位数。一些其他的工作会发生变异，以了解关于程序行为的污染级别的观察，特别是变异程序使用的输入[8，10，33，44]。当其他fuzzer使用预定义的数据突变策略（如位翻转或rand替换）时，gentutor使用被测程序的片段，这些片段通过动态切片将输入解析或操作为突变器[29]。SDF利用种子本身的特性来指导突变[35]。有时，语法被用来指导变异[2357]。Chizpurfle的[27]mutator利用了Java级语言构造的知识来帮助对Android系统服务进行进程内模糊化。

eval。Driller[50]和MAYHEM[8]观察到，程序中的一些有条件守卫很难通过暴力猜测来满足，因此（偶尔）在评估阶段调用一个符号执行器来越过它们。S2F还在eval期间使用了一个符号执行器[58]。其他的工作集中在通过改变操作系统来提高eval的速度[56]或者使用不同的低级原语来观察执行的效果[23，25，47]。T-Fuzz[39]将转换程序以删除对阻止访问新代码的输入的检查。MEDS[24]执行更细粒度的运行时分析，以在模糊期间检测错误。

isInteresting。虽然大多数论文关注的是崩溃，但一些工作改变了观察结果，将不同类别的程序行为视为有趣的，例如更长的运行时间[41]，或不同的行为[40]。Steelix[33]和Angora[10]对该程序进行了测试，以便通过观察揭示有关满足条件进展的细粒度信息。Dowser和VUzzer[22，44]使用静态分析为程序点分配不同的奖励，根据的可能是hood估计，经过该点将导致漏洞，或者到达CFG中的更深点。

choose。一些作品根据是否到达程序的特定区域来选择下一个候选输入[5，6，32，44]。其他的工作探索了选择候选种子的不同算法[45，55]。

3 OVERVIEW AND EXPERIMENTAL SETUP

本文的目的是评估现有的模糊测试算法实验评估的研究实践。如导言中所述，评估模糊测试算法a需要几个步骤：（a）选择要比较的基线算法B；（B）选择要测试的有代表性的目标程序集；（c）选择如何测量a与B的性能，理想情况下是发现错误；（d）填写算法参数，例如选择种子文件以及算法应该运行多长时间；和（e）对A和B执行多次运行并统计比较它们的性能。

关于模糊测试的研究论文在如何执行这些步骤上有很大的不同。对于第2.2节中介绍的32篇论文中的每一篇，表1指出了用于评估的基准程序；用于比较的基准模糊器；每个配置执行的试验次数；是否考虑了性能差异；如何将崩溃的输入映射到错误（如果有的话）；是否测量代码覆盖率以判断性能；如何选择种子文件；每次试用使用的超时时间（即允许fuzzer运行多长时间）。标题中给出了表格中每个单元格的说明；空白单元格表示论文的评估没有提到这一项。

例如，表1中的AFLFast[6]行显示，AFLFast的评估使用6个实际程序作为基准（第2列）；使用AFL作为基准模糊器（第3列）；在不报告任何方差的情况下运行每个实验8次（第4列）；测量并报告碰撞，但也进行了人工分类，以获得基本事实（第6栏）；未测量代码覆盖率（第7栏）；使用空文件作为唯一输入种子（第8栏）；并将6小时和24小时设置为不同实验的超时时间（第9栏）。

这些评价中哪些是“好的”，哪些不是，从某种意义上说，它们获得的证据支持声称的技术进步？在接下来的章节中，我们将从理论和经验两方面评估评估结果，并进行实验，证明糟糕的选择会导致对算法适应度的误导或错误结论。在某些情况下，我们认为，关于评价的“最佳”选择仍然是一个悬而未决的问题，但在其他情况下，显然应该采取某种特定的办法（或至少不应采取某些幼稚的办法）。总的来说，我们认为现有的每一项评估都缺乏某种重要的方式。

我们在这一节结束时描述了我们自己的实验设置。

fuzzer。在我们的实验中，我们使用AFL（标准配置参数）2.43b作为基线B，AFLFast[6]作为“高级”算法A。我们使用了2017年7月的AFLFast版本（从Github克隆而来），该版本基于AFL 2.43b版本。请注意，这些版本比Bóhme等人的原始论文[6]中使用的版本更新。最初的AFLFast的一些（但不是全部）想法被2.43b版本纳入AFL。这对我们来说不是问题，因为我们的目标不是重现AFLFast的结果，而是将其作为一个代表性的“高级”模糊器，以考虑（in）经验评估模糊器方法的有效性。（我们还注意到，AFL是我们所研究的14/32篇论文的基线，因此在我们的实验中使用它直接与使用它的那些评价相联系。）我们选择它和AFL是因为它们是开源的，易于构建，并且易于比较。我们偶尔也会考虑一种我们称之为AFLNaive的配置，即关闭覆盖跟踪的AFL（使用选项-n），有效地将AFL变成一个黑箱模糊器。

Benchmark programs.我们在实验中使用了以下基准程序：nm、objdump、cxxfilt（都来自binutils-2.26）、gif2png和FFmpeg。所有这些程序都是从最近对模糊技术的评估中获得的。FuzzSim中使用了FFmpeg-n0.5.10[55]。binutils-2.26是AFLFast评估的主题[6]，只有上面列出的三个程序有不可恢复的bug。gif2png-2.5.8是由VUzzer[44]测试的。1我们并不认为这是一个完整的基准测试套件；事实上，我们认为获得一个好的基准测试套件是一个开放的问题。我们只是使用这些程序来演示如何在不同的目标上进行测试，从而得出不同的结论。

绩效评估。在我们的实验中，我们测量了一个模糊器在一段时间内可能导致的“唯一”崩溃的数量，其中唯一性由AFL的覆盖概念决定。特别是，如果两个崩溃输入具有相同的（边缘）覆盖配置文件，则它们被认为是相同的。虽然这一措施并不罕见，但也有其问题；第7节详细讨论了原因。

平台和配置。我们的实验是在三台机器上进行的。机器I和II配备了12个2.9GHz的Intel Xenon CPU（每个CPU有2个逻辑核）和运行Ubuntu16.04的48GB RAM。机器III有24个2.4GHz CPU和110GB RAM，运行Red Hat Enterprise Linux服务器7.4。为了解释这些系统之间可能的差异，每个基准程序总是在同一台机器上测试，对于所有的fuzzer组合。我们的测试脚本利用了系统上的所有CPU，尽可能多地并行运行测试。每个CPU生成一个测试子进程，并通过CPU关联限制到它。每个试验都允许运行24小时，我们通常测量每个配置至少30个试验。我们还考虑了各种种子文件，包括空文件、随机选择的正确类型的文件和手动生成（但格式良好）的文件。

表1：过去模糊评价的总结。空白单元格表示论文的评估没有提到此项；-表示它不相关？意味着这个元素被提到了，但是没有足够的细节来说明它。基准测试：R表示现实世界的程序，C表示CGC数据集，L表示LAVA-M基准测试，S表示手动注入错误的程序，G表示Google fuzzer测试套件。基线：A表示AFL，B表示BFF[3]，L表示libfuzzer[34]，R表示Radamsa[43]，Z表示Zzuf[60]，O表示不超过1篇论文使用的其他基线。试验：试验次数。方差：C表示置信区间。崩溃：S表示用于在分类过程中对相关崩溃进行分组的堆栈散列，O表示用于分类的其他工具/方法，C表示用于区分崩溃的覆盖率配置文件，G表示根据基本事实进行分类的崩溃，G\*表示手动操作部分获得用于分类的基本事实。覆盖：L表示行/指令/基本块覆盖，M表示方法覆盖，E表示控制流边缘或分支覆盖，O表示其他覆盖信息。种子：R代表随机抽样的种子，M代表人工构建的种子，G代表自动生成的种子，N代表非空种子，但不清楚种子库是否有效，V代表假定存在有效种子，但不清楚如何获得种子库，E代表空种子，/意味着在不同的程序中使用了不同的种子，但在一个程序中只有一种种子。超时：以分钟（M）、小时（H）和/或天（D）为单位报告的时间。

4 STATISTICALLY SOUND COMPARISONS

所有现代模糊算法在执行测试时基本上都采用随机性，尤其是在执行突变时，但有时也采用其他方式。因此，仅仅运行fuzzer A和baseline B并比较它们的性能是不够的。相反，A和B都应该进行多次试验，并且应该判断它们之间的性能差异。

也许令人惊讶的是，表1显示了我们所考虑的大多数（32篇）模糊论文中的18篇没有提到试验的次数。根据上下文线索，我们的解释是他们每人做了一次试验。一个可能的理由是随机性“趋于均匀”，即，如果运行足够长的时间，随机选择将收敛，模糊器将找到相同数量的崩溃输入。从我们的实验中可以清楚地看到，这并不是真正的模糊性能，不同的运行会有很大的差异。

考虑图2所示的结果，图2显示了我们通过AFL（蓝色）和AFLFast（红色）在一段时间内（X轴）发现的累计crashes次数（Y轴），每个碰撞都以空种子开始。在每个图中，实线表示30次运行的中值结果，虚线表示最大和最小观测结果，以及中值95%置信区间的上下界[12]。（最外面的虚线是max/min，里面的是CIs。）

从这些地块的高度变化的表现可以清楚地看出，只考虑一次运行可能会导致错误的结论。例如，假设AFL的FFmpeg上的单次运行结果是最大值，在550次崩溃中排名第一，而AFLFast上的单次运行结果是最小值，在150次崩溃中排名第一（图2d）。仅仅比较这两个结果，我们可能认为AFLFast比AFL没有优势。或者我们可以观察到AFLFast的最大值和最小值，得出相反的结论。

进行多次试验并报告平均值更好，但不考虑方差也有问题。在表1中，我们可以看到14篇考虑了多次试验的论文中有11篇没有描述性能差异（在方差栏中有一个空白框）。相反，他们每个人在得出结论时都比较了A和B的“平均”表现（我们假设：算术平均），除了Dowser[22]报告了中位数，还有两个[20，59]没有提到“平均”是如何计算的。

问题是，如果方差足够大，平均值的差异可能在统计学上并不显著。解决方法是使用统计检验[38]。这样的测试表明性能差异是真实的，而不是偶然的。Arcuri和Briand[2]建议对于随机测试算法（比如

fuzzers），应该使用Mann-Whitney U检验来确定A和B的随机排序，即A的数据样本中的试验结果是否比B的结果更可能大。Mann-Whitney是非参数的，因为它不假设随机算法性能的分布；相比之下，标准t检验假设正态分布。

回到我们的实验，我们可以看到简单比较平均值可能会得出错误的结论。例如，对于gif2png，AFLFast发现24小时后平均51次碰撞，而AFL为39次（相差12次）。但是，进行Mann-Whitney检验得到的p值大于0.05，这表明即使在30个24小时的试验中，这种差异也可能没有统计学意义。对于FFmpeg，AFLFast的中位数是369.5次崩溃，而AFL的中位数是382.5次，也有12次左右的崩溃，这次有利于AFL。同样，曼惠特尼认为这一差异是显著的。另一方面，对于纳米，AFLFast比AFL的优势极不可能偶然出现。

表1中方差列中带“C”的三篇论文至少给出了置信区间和平均值，最接近最佳实践。但即使在这里，他们也没有将他们的方法的性能与基线进行统计上的比较；他们让读者直观地判断这种差异。这是有帮助的，但不像（容易执行的）统计测试那样具有决定性。

讨论。虽然我们使用统计测试的建议应该没有争议，但是关于测试的最佳选择还有进一步的争论。特别是，两个可行的替代方案是排列测试[17]和基于引导的测试[7]。这些测试的工作方式是将测量数据视为总体人口的一种替代，系统地比较测量数据的排列和重新采样，以创建具有置信区间的排名。我们不清楚这种方法是否比曼惠特尼（Mann Whitney）更合适，所以我们遵循Arcuri和Briand[2]。

确定fuzzer A的性能中值大于fuzzer B是至关重要的，但相关的问题涉及效果大小。仅仅因为A可能比B好，并不能告诉我们它有多好。我们通过观察被测介质的差异来隐式地回答这个问题。统计方法也可用于确定这种差异代表真实差异的可能性。Arcuri和Briand提出了Vargha和Delaney的Aˆ12统计[52]（使用了Mann-Whitney计算的元素）。这里也可以使用Bootstrap方法。

5 SEED SELECTION

回想图1，在迭代选择和测试输入之前，fuzzer必须选择种子输入的初始语料库。大多数（32篇中的27篇，根据第2.2节）最近的论文关注于改进主模糊回路。如表1的seed列所示，大多数论文（30/32）使用了非空的seed语料库（带有G、R、M、V或N的条目）。一种流行的观点是，种子应该是格式良好的（“有效的”），较小的种子可能会驱动程序快速执行其预期的逻辑，而不是导致程序在解析/格式良好性测试时终止[31、44、45、53]。然而，很多时候并没有给出具体的种子使用细节。条目“V”出现9次，表示使用了有效的种子语料库，但未提供详细信息。条目“N”出现了10次，表示一个非空种子，但其内容再次没有详细信息。两篇论文[5，6]选择使用空种子（条目“E”）。当我们询问他们时，他们指出使用空种子是在输入配置中比较重要变量的简单方法。其他论文使用手工或算法构建的种子，或随机抽样的。

可能是初始种子库的细节是唯一重要的；例如，无论使用哪种种子，算法的改进都会得到反映。但也有可能在种子格式和算法选择之间存在着强烈和/或令人惊讶的交互作用，这可能会增加结果的细微差别[37]。事实上，这就是我们的研究结果。

我们用不同的种子测试了FFmpeg，包括空种子、现有视频文件的样本（“采样”种子）和随机生成的视频（“生成”种子）。对于采样的种子，从FFmpeg样本网站上抽取视频，分别从AVI、MP4、MPEG1和MPEG2子目录中抽取2个4个样本，然后将文件过滤掉，只包含小于1 MiB的种子，即AFL的最大种子大小，剩下9个采样的种子。这一组进一步缩减到最小的视频文件，以产生3个采样和1个采样的种子。对于made seeds，我们使用videogen（FFmpeg附带的一个工具）创建48个随机视频帧，使用audiogen（也附带）创建12秒音频，然后将它们与FFmpeg一起缝合为3个made MP4、MPG和AVI文件，每个文件的速度为4fps，从而生成视频和GIF文件。生成的种子是生成的MP4文件。我们还使用空种子和1个自制种子测试了nm、objdump和cxxfilt。对于nm和objdump，1-made种子是通过编译hello world C程序生成的。CXFILT的1个生成种子是由16个随机字符组成的文件，从字母集（大写和小写）、数字0到9和下划线中选择，这是标准化的C++名字的字母表。图3显示了这些不同的FFmpeg种子选择的结果。一个明显的趋势是，对于AFL和AFLFast来说，空种子比任何一组有效的、非空种子产生的崩溃性输入要多得多。另一方面，对于阿夫纳维来说，趋势是相反的。在使用非空种子的实验中，性能也有所不同。例如，图3（b）和图3（d）使用一个有效的种子（构造两种不同的方式）显示了非常不同的性能。前者在24小时后发现大约100起AFL和AFLFast事故，而后者发现不到5起。

表2的顶部放大了图3（a）和（b）中24小时标记处的数据。第一列表示使用的目标程序和模糊器；第二列（“空”）表示使用空种子时发现的崩溃的中位数；最后一列（“1-made”）表示使用有效种子时发现的崩溃的中位数。最后两列中的括号是用于统计检验AFLFast或AFL-naive性能与AFL的差异是真实的还是偶然的。对于AFL和AFLFast，一个空种子会产生数百个崩溃的输入，而对于AFLVINive，它不会产生任何输入。然而，如果我们使用1种或3种种子，AFLNaive发现的崩溃率明显高于AFL和AFLFast（5000比102/129）。

表2的其余部分在空栏中复制AFLFast评估的结果[6]，但随后在1-made栏中使用有效种子重新考虑。与AFLFast论文的结论相似，AFLFast在使用空种子时的碰撞检测能力优于AFL（具有统计学意义）。然而，当使用1粒种子时，AFLFast并没有那么好：它在nm上不再优于AFL，AFL和AFLFast通常都会发现更少的崩溃。

总之，很明显，一个模糊器在同一个程序上的性能会因使用的种子而有很大的不同。即使是有效的，但不同的种子可以诱导非常不同的行为。我们的结果表明，在评价一个算法时，考虑到种子的多样性是明智的。论文应具体说明如何收集种子，最好还是提供实际使用的种子。我们也认为应该考虑空种子，尽管它的使用违背了传统的智慧。从某种意义上说，这是最普遍的选择，因为空文件可以作为任何文件处理程序的输入。如果一个fuzzer能够很好地处理各种程序中的空种子，也许它也能很好地处理尚未测试的程序中的空种子。它从巨大的配置空间中取出一个重要的变量（即使用哪个文件作为种子）。

6 TIMEOUTS

另一个重要的问题是在一个特定的目标上运行一个fuzzer需要多长时间。表1的最后一列显示了先前的模糊器实验设置了非常不同的超时。一般从1小时到几天和几周不等。3种常见的选择是24小时（10篇论文）和5或6小时（7篇论文）。我们观察到，最近使用LAVA作为基准套件的文件选择了5小时作为超时时间，可能是因为原始LAVA文件中也有同样的选择[16]。6篇论文跑fuzzer的时间都超过了1天

我们考虑过的大多数报纸都毫无理由地报道了暂停。这意味着，超过一定的阈值，不需要更多的运行时间，因为算法之间的区别将很明显。然而，我们发现算法之间的相对性能会随着时间的推移而变化，并且过早终止实验可能会产生不完整的结果。例如，AFLFast的评估显示，AFL在6小时后没有在objdump中发现任何漏洞[6]，但运行AFL更长的时间似乎说明了另一个情况，如图2b所示。6小时后，AFL和AFLFast都开始以合理的速度发现崩溃。在gif2png上运行AFL显示了另一个有趣的结果，如图2e所示。即使在13小时后，AFL发现的车祸的中位数仍为0，但仅在7小时后，它发现了40起车祸。因为bug通常存在于程序的某些部分，所以fuzzing只在这些部分最终被发现时才检测出bug。图4显示了3个采样种子在nm上运行的AFL和AFLFast的结果。6小时后，没有一次AFL运行在nm中发现任何缺陷，而AFLFast发现的平均碰撞次数是4次；Mann Whitney说，这一差异是显著的。但24小时后，这一趋势发生了逆转：AFL发现14起车祸，而AFLFast仅发现8起。同样，这种差异也是显著的。

考虑什么是合理的超时时间？从实用的角度来看，较短的超时是方便的，因为它们需要较少的总体硬件资源。在某些实际场景中，较短的时间可能更有用，例如，在正常开发过程中作为夜间运行的一部分。另一方面，如我们的实验所示，更长的运行时间可能会揭示出更普遍的性能趋势。随着运行时间的延长，特定的算法可能会更好；例如，它们可能会启动得很慢，但随着生成更多的测试，它们的错误查找能力会加快。例如，Skyfire花了几天的时间才使其更好的性能（超过AFL）变得清晰[53]。

我们认为，评估应该包括描述一段时间内性能的图表，正如我们所做的那样（如图4所示）。这些运行应该考虑至少24小时的超时；较短时间的性能可以很容易地从如此长的运行中提取出来。

讨论。除了记录特定时间的性能（例如，5、8和24小时的碰撞计数），还可以报告曲线下面积（AUC）作为一个较少标点的性能度量。例如，一个fuzzer在5秒内每秒发现一次崩溃，其AUC为12.5秒，而一个fuzzer在4到5秒内也发现了5次崩溃，其AUC为2.5秒。这些度量直观地反映了，在更早和更长的时间内发现碰撞比发现晚爆炸更可取。另一方面，这项措施可能更倾向于一个稳定的碰撞查找器，在5次碰撞时达到峰值，而不是在最后一口气发现10次；更多的碰撞不是更好吗？因此，AUC度量不能替代基于时间的性能图。。

7 PERFORMANCE MEASURES

到目前为止，我们关注于使用“唯一的”崩溃来测量fuzzer的性能，也就是说，使用导致“唯一的”崩溃的输入（很快定义）。由于崩溃是潜在严重错误的症状，根据崩溃输入的数量来度量模糊器似乎是很自然的。但是错误和崩溃的输入并不是一回事：许多不同的输入可能触发相同的错误。例如，缓冲区溢出可能会导致崩溃，无论输入数据由什么组成，只要该数据的大小超过缓冲区的长度。因此，简单地将崩溃的输入数作为性能度量可能会产生误导：fuzzer a可能会比fuzzer B发现更多的崩溃，但会发现相同或更少的实际错误。

因此，许多论文采用了一些策略来消除重复（或分类）崩溃，以便将它们映射到独特的错误。有两种流行的自动启发式方法可以做到这一点：使用AFL的覆盖率配置文件概念和使用堆栈散列。在表1中，在碰撞栏中标记为“C”（7张纸）和“S”（7张纸）。有四篇论文使用其他工具/方法进行分类，标记为“O”。例如，VUzzer还使用了一个名为！可用于评估由错误引起的崩溃的可利用性[44]。用户可以对转化为攻击的可能性很低的崩溃打折扣，因此显示fuzzer发现更危险的bug是有利的。我们实验中使用的重复数据消除策略对应于“C”。

不幸的是，正如我们在本节实验性地展示的那样，这些重复数据消除启发式方法实际上不善于根据根本原因对崩溃的输入进行聚类。

有几篇论文确实考虑了某种形式的基本事实。六篇论文把它作为主要的绩效指标，在表中标上“G”。通过选择基准程序，他们能够完美地将崩溃的输入映射到根本原因。另外八篇论文，在表中标记为“G\*”，试图对崩溃进行分类，以确定其根本原因，但这样做并不完全正确。通常，这种分类是作为“案例研究”进行的，通常既没有充分的依据，也没有完整的基本事实作为总体（数字）性能度量。

在接下来的三个小节中，我们将详细讨论性能度量，说明为什么使用启发式而不是实际的基本事实来比较fuzzer性能会导致误导或错误的结论。与直接测量错误不同，我们研究的论文中有近一半考虑了fuzzer执行（覆盖）目标程序重要部分的能力。这种度量可能比bug计数更具概括性，但不能替代它；我们将在本节末尾讨论它。

7.1 Ground Truth: Bugs Found

fuzzer的最终度量是它发现的不同bug的数量。如果fuzzer A通常发现的bug比基线B多，那么我们可以认为它更有效。一个关键的问题是：什么是（明显的）bug？这是一个主观问题，不容易回答。

我们设想开发人员最终将使用崩溃输入来调试和修复目标程序，以便不再发生崩溃。该修复可能不是特定于输入的，而是泛化的。例如，错误修复可能包括一个长度检查，以阻止缓冲区溢出的发生，这将对所有太长的输入有效。因此，如果目标p在给定输入I时崩溃，但在应用错误修复时不再崩溃，那么我们可以将I与修复所解决的错误关联起来[11]。此外，如果输入I1和I2都在p上引发崩溃，但一旦应用了错误修复，这两个输入就不再导致崩溃，那么我们知道这两个输入都会识别相同的错误（假设修复是适当的“最小”[26]）。

在目标程序上运行已知错误时，我们可以直接访问基本事实。这样的程序可能是有缺陷的旧版本，这些缺陷已经被修复，或者它们可能是合成的程序，或者是有综合引入的缺陷的程序。考虑到前一类，我们不知道以前的工作，使用旧程序及其相应的修复，以完全分流崩溃根据地面真相。在后一类中，九篇论文使用合成套件来确定地面真实性。最受欢迎的套房是CGC（Cyber Grand Challenge）[14]和LAVA-M[16]；我们将在下一节详细讨论这些套房。对于这两种情况，bug都被注入到原始程序中，触发特定bug的方式会在程序崩溃之前产生一个信号（如特定的错误消息）。因此，很明显哪个bug是由fuzzer生成的输入触发的。如果以前触发过该bug，则可以丢弃该输入。另外两篇论文使用了手工选择的程序，这些程序带有手动注入的漏洞。

7.2 AFL Coverage Profile

当无法获得基本的真实情况时，研究人员通常采用启发式方法消除重复的崩溃输入。AFL采用的方法，以及表1中7篇论文（标记为“C”）所使用的方法，是考虑具有与等效代码覆盖率配置文件相同的输入。如果碰撞的边缘覆盖范围包含以前任何碰撞中都看不到的边缘，或者丢失了以前所有观察到的碰撞中的边缘，则AFL将认为碰撞是“唯一的”

基于覆盖率配置文件对重复输入进行分类是有意义的：两个不同的bug可能有不同的覆盖率表示。另一方面，很容易想象一个bug可以由不同覆盖率配置文件的运行触发。例如，假设图5中程序中的函数崩溃将无条件地segfault。尽管程序中只有一个bug，但有两类输入将被视为不同的：以“a”开头的输入和不以“a”开头的输入。

根据事实进行评估。这种情况在实践中多久发生一次？我们使用AFL和AFLFast检查了cxxfilt的模糊运行生成的崩溃输入。自从我们的模糊版本发布以来，这段代码已经进行了多年的开发活动，因此（大多数）我们的模糊发现的错误已经被修补。我们使用git来标识更改用于编译cxxfilt的源文件的提交。然后，我们为每个提交构建了每个版本的cxxfilt。这产生了812个不同版本的cxfilt。然后，我们在每个不同版本的cxxfilt上运行每个崩溃输入（57142个），记录该版本是否崩溃。如果不是，我们认为输入是该程序版本修复的错误的表现。

为了确保我们的测试结果可信，我们采取了另外两个步骤。首先，我们确保非碰撞行为不是偶然的。当被触发时，内存错误和其他错误类别不一定会导致崩溃。例如，如果访问了未映射的内存，则越界数组读取只会使程序崩溃。因此，提交可能会改变程序的某些方面，从而在不实际修复错误的情况下消除崩溃。为了解决这个问题，我们使用地址消毒剂和未定义的行为消毒剂（ASAN和UBSAN）[48]编译了每个cxxfilt版本，它添加了对各种错误（包括内存错误）的动态检查。我们认为as an/UBSAN错误报告的出现是“崩溃”

第二，我们确保每个bug修复提交对应于一个bug修复，而不是多个bug修复。为此，我们手动检查将崩溃输入转换为非崩溃输入的每个提交，判断我们是否相信有多个不同的错误被修复（基于我们之前开发的原则[26]）。如果是这样，我们手动将提交拆分为较小的提交，每个修复一个。在我们的实验中，我们只需要这样做一次，一个将一批更改从cxxfilt的libiberty fork导入到主主干中的commit。5我们查看了组成此批的单个libiberty commit，以帮助我们确定如何拆分它。最终，我们将其分为五个不同的错误修复提交。

我们最后的方法产生了9个不同的错误修复提交，留下少量的输入仍然会导致当前版本的cxxfilt崩溃。图6组织这些结果。图中的每个条形图表示由AFL或AFLFast进行的24小时模糊试验。6对于每个条形图，y轴上条形图的大小是“唯一”（根据覆盖剖面）碰撞诱导输入的总数，当条被分割时，这些输入中的哪一个与我们的地面真实性分析发现的错误修复组合在一起。每个栏上方是该运行发现的Bug总数（即每个栏中的间隔数）。运行按运行中发现的唯一错误数排序。

我们可以看到，在运行期间发现的bug数量和运行中发现的崩溃输入数量之间充其量是弱相关的。这种相关性意味着，当从左向右移动时，碰撞计数将呈现更强劲的上升趋势。我们还可以看到，与AFL相比，AFLFast通常发现更多的“独特”崩溃输入，但每次运行发现的bug数量仅略高一些。Mann-Whitney发现，碰撞的差异在统计学上是显著的，p值为10-10，但bug的差异不是（但很接近），p值为0.066。

讨论。尽管我们采取了一些步骤来确保我们的分类符合基本事实，但我们可能仍然夸大或减少了实际的错误计数。作为前者的一个例子，我们注意到As an/UBSAN不能保证捕获所有内存安全违规，因此我们可能将偶然的更改归因于错误修复。我们发现了一些无法解释为什么提交修复了崩溃的情况，因此没有将提交与错误关联起来。另一方面，我们可能无法在一次提交中区分多个错误修复，可能是错误的，也可能是观察者的判断与我们的不同。在任何情况下，我们的计数和“唯一崩溃”之间的差异的大小意味着，即使计数稍有改变，“唯一崩溃”大量高估真实错误数的顶级结果也将保持不变。

如果我们在所有的实验中都使用了地面真实度测量，它可能会改变第4-6节中结果的特征。例如，在计算错误而不是“唯一”崩溃输入时，由于随机性（如图2）而导致的配置内的性能变化可能没有那么明显。在这种情况下，我们建议进行多次试验更为重要，因为模糊器A和B之间的微小性能差异可能需要许多试验来辨别。可能是由于种子不同而导致的性能差异（图3）也可能不那么明显如果一个种子发现了数百个崩溃，而另一个发现的则少得多，这是正确的，但最终所有的崩溃都对应于同一个bug。当计算错误而不是崩溃时，随着时间的推移，性能的变化也可能更小（图4）。另一方面，随着时间的推移，我们也有可能发现更多的变异，和/或不同的种子，而不是更少。在这两种情况下，我们认为第4-6节中的结果引起了足够的关注，即我们的建议应采用更长的超时时间和各种种子（包括空种子）进行测试，除非和直到具有地面真相数据的实验结果能够更清楚地说明情况。

7.3 Stack hashes

另一种常见的启发式重复数据消除技术是堆栈哈希[36]。我们认为有七篇论文使用了这种技术（表1中标记为“S”）。想法如下。假设我们的缓冲区

溢出错误存在于更大程序内部的函数中，例如库例程中。假设溢出立即导致segfault，则崩溃将始终发生在程序中的同一位置。更一般地说，崩溃可能取决于一些额外的程序上下文；例如，溢出缓冲区只有在特定调用函数中创建时才可能过小。在这种情况下，我们可以查看调用堆栈，而不仅仅是程序计数器，以将崩溃映射到特定的bug。为了忽略虚假的变化，我们将重点放在返回地址的标准化到其源代码位置。由于堆栈最靠近顶部的部分可能是最相关的，我们可能只将最新的N个堆栈帧与特定错误的触发关联起来。（N通常选择在3到5之间）可以对这些帧进行散列，以便与以前的错误（堆栈散列）进行快速比较。

只要相关上下文是唯一的，堆栈哈希就可以工作，并且在崩溃时仍在堆栈上。但是很容易看到这样的情况：如果不支持堆栈散列，则可能会导致计算不足或计算过多的真正错误。以图7中的代码为例，该代码的format函数中存在一个错误，该错误会破坏字符串s，最终导致输出函数崩溃（当s被prepare函数无辜地传递给它时）。format函数由函数f和g分别调用。

假设我们模糊这个程序并生成导致两个崩溃的输入，一个从f的调用开始，另一个从g的调用开始。将N设置为前3帧，堆栈哈希将正确地识别这两个输入对应于同一个错误，因为只有格式、准备和输出将在堆栈上。但是，将N设置为5会将输入视为不同的崩溃，因为现在一个堆栈包含f，而另一个堆栈包含g。另一方面，假设此程序有另一个错误函数，该函数在传递给prepare之前也会损坏。将N设置为2将不正确地合并由于该错误而导致的崩溃和由于错误格式而导致的崩溃，因为只考虑堆栈上的最后两个函数。

根据事实进行评估。我们通过比较堆栈散列的确定与之前实验中发现的错误的标签来衡量堆栈散列的有效性。我们的堆栈哈希实现使用地址清理器为cxxfilt的每个崩溃输入生成一个堆栈跟踪，并选择N作为堆栈上用于哈希的前3个条目。

我们的分析发现，堆栈哈希在消除重复数据输入方面比覆盖率配置文件有效得多，但仍然会高估发现的错误数。表3显示了堆栈哈希与我们标识的标签的比较结果。例如，考虑标签B，它表示31103个输入（第5列）。在这些输入中，产生了362个不同的堆栈散列（第2列）。如果堆栈散列度量是我们对cxxfilt中错误分布的唯一了解，我们将声称发现的错误比实际发现的多两个数量级。另一方面，堆栈散列对于标签H来说似乎做得很好：一个散列匹配所有568个输入。总之，在所有运行中，595个散列对应于9个bug，膨胀为66×，而57044个覆盖配置文件对应于9个bug的唯一输入，膨胀为6339×。4个崩溃输入分别与它们自己的“修复”提交相关联，但是当我们检查各自的代码更改时，我们看不出更改为什么要修复崩溃。因此，我们在表3中将这些输入列为“未知”。ASAN/UBSAN没有检测到所有可能的未定义行为，因此可能是编译之间的代码或数据布局更改或某种其他形式的非确定性正在抑制崩溃行为。编译器错误也是可能的。我们正在继续调查。

虽然堆栈散列并没有像AFL覆盖配置文件那样对bug进行过多计数，但它有一个严重的问题，即散列不是唯一的。例如，标签B中只有343个只匹配与B相关联的输入（第3列）。剩下的19个也匹配一些其他崩溃输入（第4列）。因此，如果将堆栈哈希用于重复数据消除，这些其他输入将被错误地丢弃。实际上，对于标签G来说，没有唯一的散列（第3列中有一个0）——它只是错误地匹配。总的来说，大约16%的哈希是不唯一的。8因此，基于堆栈哈希的重复数据消除将丢弃这些错误。

讨论。表3显示了另一个有趣的趋势，在图6中也很明显，但不太准确。有些bug是由极少量的输入触发的，而另一些则是由极大量的输入触发的。错误G和I分别对应于2或10个输入，而错误B、D和E对应于超过10K个输入。先前的模糊研究已经发现了相似的错误分布[11]。虽然表3合并了所有测试的所有输入，但单独考虑每个测试（如图6所示），我们发现没有一次运行发现所有9个错误；所有运行都发现了错误B、D、E，但是没有一次运行发现超过5个额外的错误。

一个重要的开放性问题是，我们在cxxfilt中观察到的趋势是否适用于其他目标程序。要回答这个问题，就需要对我们在这里进行的味道进行更多的“基本事实”分析。假设它们成立，我们得出两个十个结论。首先，这些趋势强化了bug启发式的问题：在存在“罕见”输入的情况下，查找100个崩溃输入和101个（明显微不足道的差异）之间的差异可能代表查找1或2个唯一的bug（显著的差异）。其次，fuzzers可能受益于SAT解算器使用的一个算法技巧：通过丢弃一些当前状态并重新开始初始种子，随机“重新启动”搜索过程[46]，从而模拟运行单独试验的效果。面临的挑战是找出在重新启动时保留的模糊状态，以便保留重要的知识，但避免陷入局部最小值。

相关工作。van Tonder等人最近的工作。[51]还实验性地评估堆栈散列和覆盖概况对地面真相的有效性。和我们一样，他们将基本事实定义为通过特定代码补丁纠正的单个概念性错误。他们比较了覆盖率剖面和堆栈散列如何接近这一基本事实。和我们一样，他们发现两者都会高估真正的错误数量。当他们考虑不同的补丁和目标程序时，他们的研究是对我们的补充。然而，它们的崩溃输入集是通过对初始已知崩溃输入的突变生成的，而不是通过正常的模糊过程。因此，他们的数字并不像我们的数字那样，描述了在典型的模糊化用例中糟糕的重复数据消除策略的影响。

Pham等人。[42]还研究了N=1和N=∞的堆栈散列如何能够对通过符号执行识别的错误进行过多计数和欠计数。他们的兴趣是与他们自己的重复数据消除技术进行比较，因此他们的研究没有全面考虑基本事实。

7.4 Code Coverage

运行fuzzer来查找程序中的错误。如果一个fuzzer运行了很长一段时间，但没有发现bug，那么它的用户会认为它是不成功的。根据fuzzer发现的bug数量来评估fuzzer似乎是合乎逻辑的。然而，仅仅因为一个模糊者没有发现一个虫子，也许并不能告诉我们关于模糊者功效的全部故事。也许它的算法是正确的，但是很少或者根本没有错误可以发现，而模糊器只是变得不走运。

一种解决方案是取而代之（或者也）测量fuzzer A在基线B上对代码覆盖率的改进。Greybox fuzzers已经将优化覆盖率作为isintesting函数的一部分，因此显示改进的代码覆盖率肯定会指示模糊化的改进。这是有道理的。要在程序中的某个特定点找到崩溃，需要执行程序中的该点。先前对测试套件有效性的研究也表明，更高的覆盖率与错误发现有效性相关[19，30]。近一半的论文我们考虑了测量代码覆盖率；FairFuzz只评估使用代码（分支）覆盖率的性能[32]。

然而，最大化代码覆盖率与发现错误没有直接关系，这是没有根本原因的。虽然覆盖引导模糊器相对于黑盒模糊器的一般效果意味着存在很强的相关性，但特定算法可能会避开较高的覆盖率，而将注意力集中在可能存在错误的其他迹象上。例如，AFLGo[5]并不打算在全球范围内提高覆盖率，而是着眼于程序中特定的、可能出错的点。即使我们假设覆盖率和错误发现是相关的，这种相关性也可能很弱[28]。因此，覆盖率的次标准改进可能只会在错误发现效率方面产生微不足道的改进。

简言之，我们认为代码覆盖率作为一个次要的衡量标准是有意义的，但是根据发现的bug，基本事实应该始终是首要的。

8 TARGET PROGRAMS

我们想确定一个模糊算法通常比另一个更好，即它在从（大量）人口中提取的任何目标程序中发现错误的能力。一般性的声明通常是通过在一个声称代表总体的基准套件上测试fuzzer来实现的。其理念是，套房的良好表现应该转化为人群的良好表现。我们应该如何选择这样一个基准套件？

最近出版的著作考虑了各种各样的基准计划。大体上，它们分为两类，如表1第二列所示：真实程序和人工程序（或bug）。前者的例子包括Google fuzzer测试套件（“G”）[18]和实际程序的特别选择（“R”）。后者包括CGC（“C”）[14]、LAVA-M（“L”）[16]和人工选择的带有综合注入错误的程序（“S”）。一些论文的基准来自这两个类别（如VUzzer[44]和Steelix[33]）。正如我们下面讨论的，没有一个现有的基准选择是完全令人满意的，因此留下了开发一个良好的模糊基准的重要问题。

8.1 Real programs

根据表1，几乎所有的论文在评估中都使用了一些真实的程序。其中两篇论文[49，56]使用了Google Fuzzer测试套件[18]，一组现实世界中的程序和库，再加上一些工具，将fuzzing集中在一组已知的bug上。其他人在一组手工挑选的真实世界的程序上进行评估。

我们看到了两个问题，即真实程序被用作模糊目标的方式。首先，大多数论文只考虑少数目标项目，没有明确的理由说明其代表性。根据表1，程序的中位数是7。有时一小部分计算是合理的；例如，IMF是专门为模糊操作系统内核而设计的，因此它对单个“程序”（MacOS内核）的评估仍然很有趣。另一方面，大多数fuzzer的目标是应用于更大的群体（例如，所有的文件处理程序），因此7似乎是一个很小的数字。一个积极的异常值是FuzzSim，它使用了大量程序（超过100个），并解释了收集它们的方法。

作为少数不充分的一般目标所构成威胁的证据，考虑图2中报告的实验结果，这些结果与Bóhme等人[6]的结果相匹配。图的第一行显示了nm、objdump和cxxfilt的结果，这三个程序是Bóhme等人发现崩溃的三个程序。9将我们的注意力集中在这些程序上表明，在寻找崩溃的能力方面，AFLFast都优于AFL。然而，如果我们看第二排的数字，故事就不那么清楚了。对于FFmpeg和gif2png这两个用于其他模糊评价的程序，Mann-Whitney U检验显示AFL和AFLFast之间没有统计学差异。在我们的评估中包括这些项目削弱了任何关于AFLFast是AFL的改进的说法。

到目前为止，我们看到的使用真实程序的第二个问题是，很少有论文在同一版本上使用相同的目标。因此，很难在不同的报纸上进行非正式的比较。一组重叠的目标是binutils程序，用于一些评估[5，6，10，32]。多篇论文也考虑了FFmpeg和gif2png[9，44，45，55，58]。然而，没有人使用相同的版本。例如，这些文件中的binutils版本不同：AFLFast[6]和AFLGo[5]使用了2.26；FairFuzz[32]使用了2.28；Angora[10]使用了2.29。

Google Fuzzer套件的使用似乎解决了这两个问题：它包含25个已知错误的程序，并且是独立于任何给定的Fuzzer定义的。另一方面，它被设计成一种回归套件，不一定代表“在野外”的模糊；提供的工具和种子大多倾向于模糊者应该在几秒钟到几分钟内找到目标昆虫。

8.2 Suites of artificial programs (or bugs)

真正的程序是变化无常的，因为出现错误的可能性取决于许多因素。例如，正在积极开发的程序可能比那些相对稳定的程序有更多的错误（只是响应错误报告）。从某种意义上说，我们并不关心任何特定的程序集，而是一组有代表性的编程（反）模式，在这些模式中可能会出现bug。这种模式可以人工注入。有两个流行的套房可以做到这一点：CGC和LAVA-M。

CGC套件包含296个作为DARPA网络大挑战的一部分制作的bug程序[14]。这个套件是专门为评估错误查找工具而设计的，比如fuzz tester这个套件的程序执行实际的功能，并且有可利用的错误。LAVA（它代表大规模的自动漏洞添加）是一个将bug注入已知程序的工具[16]。该工具旨在沿可行路径添加崩溃、输入确定错误。LAVA的作者使用该工具创建了LAVA-M套件，其中包括四个注入bug的coreutils程序：base64、md5sum、uniq和who。与CGC程序不同，LAVA-M程序有很多：前三个程序中每个程序都有几十个，世卫组织有2000多个。对于这两个套件，如果一个fuzzer触发了一个bug，则会有一个指示它是哪一个的信号，这对于理解从可能的总数中找到多少个bug非常有用。

CGC和LAVA-M自从被引入以来，作为评价fuzzer的基准选择已经越来越受欢迎。在过去的两年中，CGC和LAVA-M分别被用于评价4个和5个模糊器。VUzzer[44]、Steelix[33]和T-Fuzz[39]在评估中使用了这两个基准。然而，有时CGC基准是子集：Driller[50]、VUzzer[44]和Steelix[33]分别在296个程序中的126、63和17个程序上进行评估。

虽然CGC程序是手工设计来模拟现实的，但这种模拟可能并不完美：在CGC程序上表现良好可能无法推广到实际程序。例如，cgcqce质询程序的平均大小为（仅）1774行代码，许多程序使用telnet风格的基于文本的协议。同样，LAVA-M注入的bug可能与“野外”发现的bug不够相似，现实世界软件开发背后的动机和环境可能无法转化为专门设计为不安全的合成基准。LAVA的作者写道，“LAVA未来的一个重要工作是使生成的语料库看起来更像真实程序中发现的bug。”事实上，在最近的实验[15]中，他们还表明，相对简单的技术可以有效地找到所有LAVA-M bug，它们遵循一个简单的模式。我们不知道任何独立评估这些套房被视为“真实”或“一般”程度的研究

8.3 Toward a Fuzzing Benchmark Suite

我们的评估使我们相信，确实需要一个可靠的、独立定义的基准测试套件，例如用于模糊测试的DaCapo[4]或SPEC10。这是一项足够大的任务，我们在本文中不打算承担。这应该是一个社区的努力。也就是说，我们确实对这一努力的结果有一些想法。

首先，我们认为这个套件应该有一系列的程序，这些程序应该有明确的指标来确定什么时候发现特定的错误，要么是因为错误是综合引入的（如LAVA-M和CGC），要么是因为它们以前在旧版本中被发现（如第7.2节中的地面真实性评估）。对基本事实的清楚了解避免了对同一个错误对应的输入进行过多计数，并允许评估工具的误报和漏报。我们倾向于使用带有已知bug的真实程序，因为它们的生态有效性更为可靠。

其次，这个套件应该足够大（包括程序数量和这些程序的大小），以代表整个目标人群。正确的数字是多少？这是一个悬而未决的问题。CGC包含300个小程序；Google Fuzzer套件有25个；大多数论文使用在7个左右。我们的感觉是7太小了，但它可能取决于选择哪一个7。也许25更接近正确的数字。

最后，测试方法应该建立一些防止过度拟合的防御机制。如果一个静态基准测试套件开始被普遍使用，工具可能会开始采用启发式方法和策略，这些方法和策略并不具有根本的优势，但却不成比例地应用于基准测试程序。解决这个问题的一种方法是有一个固定的标准套件和一个相对频繁变化的“可进化”部分。支持后者的一种方法是建立一个模糊化竞赛，类似于长期运行的SAT解决竞赛系列。11在这个方向上的一个努力是Rode0day，一个反复出现的错误发现竞赛。12由于模糊化研究人员事先不知道目标程序，应激励他们发展通用程序，可重复使用的技术。每个竞争对手的套件都可以加入到静态基准中，至少在一定程度上可以使套件更加健壮。一个挑战是如何规范地开发新的生态有效的目标。例如，Rode0day使用了自动化的错误插入技术，工具可能会对这些技术进行过度的调整（上面讨论的LAVA问题）。

9 CONCLUSIONS AND FUTURE WORK

模糊测试是一种很有前途的技术，已经被用来发现许多重要的错误和安全漏洞。这一前景促使越来越多的研究人员开发新的模糊测试算法。这类算法工作的证据主要是实验性的，因此重要的是，它来自一个有充分基础的实验方法。特别是，研究人员应使用一组有意义的配置参数（包括输入种子集和持续时间（超时））在一组通用目标程序上运行算法a，并与在相同条件下运行的基线算法B的性能进行比较，其中，性能定义为找到的（不同的）错误数。A和B必须运行足够的时间，以考虑模糊的内在随机性，并通过统计测试来判断性能。

本文对32篇近期的论文进行了综述，并对其实验方法进行了分析。我们发现没有一篇论文完全遵循上述方法。此外，我们使用AFLFast[6]（as A）和AFL[1]（as B）进行的实验结果说明了为什么不遵循此方法会导致误导或削弱结论。我们发现了

•大多数论文未能进行多次测试，而那些没有通过统计测试解释不同性能的论文。这是一个问题，因为我们的实验表明，运行到运行的性能可以有很大的不同。

•许多论文不是通过计算不同的错误来衡量fuzzer的性能，而是通过使用诸如AFL的覆盖率度量和堆栈散列等启发式方法来计算“独特的崩溃”。这是一个问题，因为我们所做的实验表明，启发式算法可以显著地高估错误的数量，并且确实可以通过错误地分组崩溃的输入来抑制错误。这意味着，明显的改善可能是适度的或虚幻的。许多论文都考虑了一些根本原因，但往往是作为一个“案例研究”，而不是绩效评估。

•许多报纸没有理由使用短时间的暂停。我们的实验表明，可能需要更长的超时才能完整地描述算法的性能。

•许多论文没有仔细考虑种子选择对算法改进的影响。我们的实验表明，根据使用的种子不同，性能也会有很大的不同。特别是，两个不同的非空输入不需要产生类似的性能，空种子可以比预期的更好地工作。

•关于目标方案的选择，论文种类繁多。越来越多的公司使用CGC和/或LAVA-M合成套件，这些套件的优点是它们是独立于给定算法定义的，通过模糊它们发现的错误可以可靠地计数（不需要崩溃重复数据消除策略）。其他的论文通常选择不相交的小程序集，这使得在不同的论文中比较结果变得很困难。我们的实验表明AFLFast在最初评估的目标上表现良好，但在其他论文使用的两个目标上表现并不比AFL好。

最终，我们的实验与最初的AFLFast论文[6]的积极结果大致吻合，但通过将评估范围扩大到不同的种子、更长的超时时间和不同的目标程序，AFLFast优越性的证据，至少对于我们测试的版本，被削弱了。启发式崩溃重复数据消除策略的价值值得怀疑，这进一步削弱了我们对改进声明的信心。我们认为，以前的许多论文都存在着一定程度的评价问题。因此，本文的一个关键结论是，模糊社区需要开始进行更严格的实验，以得出更可靠的结论。

具体来说，我们建议模糊测试评估应该包含以下元素：

•多个试验，用统计试验区分分布；

•一系列已知错误的基准目标程序（例如。，

LAVA-M，CGC，或旧程序的错误修复）；

•根据已知的错误来衡量性能，而不是基于AFL覆盖率分布或堆栈散列的启发式；块或边缘覆盖率可以用作二级度量；

•考虑各种（有充分记录的）种子选择-

包括空种子；

•超时至少24小时，或其他理由，以较少，随着时间的推移绘制性能。