

# 一种基于妨碍特征的模糊测试工具测评方法

郝高健<sup>1,2,3,4</sup>, 李丰<sup>1,2,3</sup>, 霍玮<sup>1,2,3,4</sup>, 邹维<sup>1,2,3,4</sup>

<sup>1</sup>中国科学院信息工程研究所 北京 中国 100093

<sup>2</sup>中国科学院网络测评技术重点实验室 北京 中国 100195

<sup>3</sup>网络安全防护技术北京市重点实验室 北京 中国 100195

<sup>4</sup>中国科学院大学 北京 中国 100049

**摘要** 模糊测试是一种高效的软件漏洞发现技术,在学术界和工业界有着丰富的研究成果和广泛的实践应用,产生了许多模糊测试工具。这些工具在技术特点及性能方面有着明显各异,需要通过测试来评估其效能,从而为工具选用以及改进提供指导。然而现有的模糊测试工具测评方法普遍存在一些情况下测评结果无法解释的问题。我们发现这与现有测评普遍忽略了模糊测试妨碍特征(Fuzzing-hampering Feature)有关。对此,本文深入研究妨碍特征对模糊测试的影响,归纳、提炼出5种妨碍特征,提出了一种将妨碍特征作为控制变量的、细粒度对比测评方法,并运用代码合成技术构建了包含118个目标程序的测试集Bench4I。经过对6款不同模糊测试工具的测评,结果表明,运用该方法可准确解释目标程序样本对被测工具功效的影响,进而推断工具的具体能力,有效提升了测评的可解释性。本文根据测评结果对实验中的被测工具提出了使用与改进建议,并实践了对QSYM的改进,取得了良好的效果。

**关键词** 模糊测试; 测评; 测试集; 软件漏洞

中图法分类号 TP311.5 DOI号 10.19363/J.cnki.cn10-1380/tn.2022.12.11

## Evaluating Fuzzers Based on Fuzzing-hampering Features

HAO Gaojian<sup>1,2,3,4</sup>, LI Feng<sup>1,2,3</sup>, HUO Wei<sup>1,2,3,4</sup>, ZOU Wei<sup>1,2,3,4</sup>

<sup>1</sup>Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China

<sup>2</sup>Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, Beijing 100195, China

<sup>3</sup>Beijing Key Laboratory of Network Security and Protection Technology, Beijing 100195, China

<sup>4</sup>University of Chinese Academy of Sciences, Beijing 100049, China

**Abstract** Fuzz testing is an efficient method to find security critical bugs. In recent years, a plenty of works about fuzz testing have been proposed in both industry and academia. A variety of fuzz testing tools have been developed. These tools differ in techniques and performance so that the evaluation of fuzzers is demanded to understand these tools. But many existing evaluations have problems of bad interpretability, which leads to limited findings from the evaluation results. In this paper, we find that the evaluation results can be affected by plenty of factors, including fuzzing-hampering features contained in the target programs. However, existing evaluations pay little attention on fuzzing-hampering features, which leads to the inability to explain the reasons behind the evaluation results, even causing unclear or erroneous conclusions. In this regard, we propose a method to evaluate fuzzers based on fuzzing-hampering features. Our method treats fuzzing-hampering features as one of the controlled variables and performs fine-grained comparative testing to find out the relationships between evaluation results and fuzzing-testing features to identify the reason causing the different results, making the evaluation more interpretable. We also develop a method to construct benchmarks with which fuzzing-hampering features can be a controlled variable during the evaluation. To implement the idea and show its effectiveness, we summarized 5 fuzzing-testing features, quantitatively defined how to calculate the indicator of the capabilities of a fuzzer and constructed a bug benchmark named Bench4I, which included 118 synthetic programs with different fuzzing-hampering features. In the experiment, we evaluated 6 fuzzers. It shows that the tools' detailed capabilities can be inferred according to the indicators calculated from the evaluation results so that the evaluation results become more interpretable. With the help of the evaluation, we also proposed several advices of using and improving these fuzzers. We put the improvement of QSYM into practice and gained a quite encouraging result.

**Key words** fuzz testing; evaluation; benchmark; security critical bug

通讯作者: 李丰, 博士, 副研究员, Email: lifeng@iie.ac.cn。

本课题得到中国科学院网络测评技术重点实验室和网络安全防护技术北京市重点实验室资助; 国家自然科学基金重点项目(No. 62032010)资助; 国家自然科学基金重点项目(No. U1836209)资助。

收稿日期: 2020-09-04; 修改日期: 2020-11-10; 定稿日期: 2022-12-07

1 引言

模糊测试(fuzz testing)是当前主流的软件漏洞挖掘技术之一,其基本原理是通过产生大量半有效的输入数据来尝试触发目标程序的异常,借此发现目标程序的漏洞<sup>[1]</sup>。AFL<sup>[2]</sup>作为目前最流行的模糊测试工具之一,已发现了上百个开源项目中的漏洞,谷歌 OSS-Fuzz<sup>[3]</sup>截至 2020 年 6 月已经发现了 300 个开源项目中的 20000 个漏洞。鉴于模糊测试在软件漏洞挖掘领域的突出表现,近年来,针对模糊测试的技术优化及改进层出不穷,表 1 统计了过去 4 年(2017 年 1 月至 2020 年 6 月)间发表在信息安全、软件工程领域的部分顶级学术会议上与模糊测试相关的论文数量,可以看出,关于模糊测试的研究工作持续受到关注且呈逐年上升趋势。学术界对模糊测试技术改进的持续关注,催生了大量模糊测试工具(表 2 归纳了近年来的模糊测试改进技术及其代表性工具),如何有效地对这些工具以及技术改进效果进行测评,成为亟待解决的问题。

表 1 发表于部分顶级会议的模糊测试相关论文数量  
Table 1 Research papers about fuzz testing published on part of the Top-level conferences

会议	2017	2018	2019	2020
USENIX	2	2	7	12
S&P	2	3	5	6
CCS	6	3	6	尚未公布
NDSS	1	4	6	4
ICSE	0	1	6	9
ASE	2	3	1	3
FSE	1	4	3	2
ISSTA	0	3	3	3
总和	14	23	37	39

目前常用的测评模糊测试技术改进效果的方法是选择一系列人工合成或基于真实程序构建的测试集,通过统计参与比对的模糊测试工具在规定时间内触发漏洞的数量、对目标程序代码的覆盖率等方面的差异,对工具效能进行评判。

在测试集构建方面,CGC<sup>[4]</sup>和 LAVA-M<sup>[5]</sup>是目前常用的采用人工合成方法构建的测试集,前者包含 296 个具备真实功能且存在漏洞的程序,后者则在 4 个 coreutils 程序的可达路径上分别注入了几十甚至上百个与输入相关且能够导致程序崩溃的漏洞。Google Fuzzer Test Suite<sup>[6]</sup>以及 FuzzBench<sup>[7]</sup>则使用 25 个真实程序构建测试集。表 1 统计的研究工作中有

23 篇论文使用了上述测试集。在效能评判方面,文献[8]调研的 32 篇论文的实验测评中有 28 篇将被测模糊测试工具触发的漏洞数作为效能评判指标,其中 12 篇论文结合了代码覆盖率来评判被测工具的效能。Yuwei Li 等人<sup>[9]</sup>进一步丰富并细化了模糊测试工具的评判指标,提出了触发漏洞的数量和质量、漏洞触发速度、漏洞触发稳定性、代码覆盖率和计算资源开销等量化指标,更加准确地反映了工具对测试集中目标程序的模糊测试效果。

合理的测试集和评价依据能够为模糊测试工具间的横向比对提供统一的规范,确保测评过程的公平性以及测评结果的可重现性。然而,目前的测评方法存在测评结论可解释性差的问题。以图 1 所示的两段代码为例,图(b)在图(a)的基础上,将条件约束中的魔数(magic value)检查替换为循环冗余校验(Cyclic redundancy check, CRC),而上述修改并不影响代码中漏洞的触发条件。AFL 以及实现了符号执行技术的 QSYM 对图(a)所示代码片段的模糊测试结果显示, QSYM 的漏洞发现效能显著优于 AFL(在第 5 节所述配置下,两者触发 bug 函数的时间开销分别为 1325 s 和 80 s),然而使用图(b)所示的代码进行测试,会发现两者触发 bug 函数所需的时间开销相当(在第 5 节所述配置下均为 1300 s 左右)。若忽略图(a)、图(b)的代码具体差异,仅观察测试结果,则难以解释 QSYM 的效能波动。此外,单独根据图(a)的测试结果所得结论与单独根据图(b)的测试结果所得结论甚至存在矛盾。反之,对于上述示例,若已知代码具体差异,根据测评结果数据则可以推导出 QSYM 实现的符号执行技术改进确实有助于模糊测试突破条件约束,而对于包含循环冗余校验的条件约束则无法有效生成可以达到 bug 函数的输入。

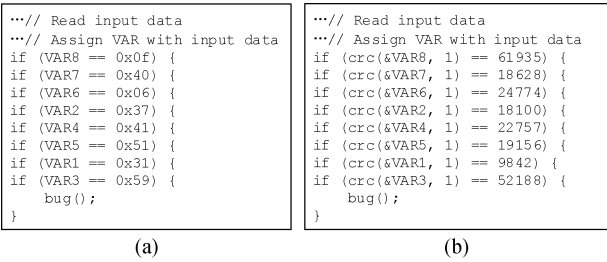


图 1 示例代码片段  
Figure 1 Example code

可见,知晓目标程序的代码特征信息不仅有助于判断工具对具备特定代码特征的目标程序的处理能力,也能更准确地验证工具的技术优势与瓶颈。本文将目标程序中包含的可能妨碍模糊测试策略、算

法或辅助技术发挥其应有功效的特征称为模糊测试妨碍特征,简称妨碍特征。现有的模糊测试工具测评方法虽然充分考虑了测试集包含的目标程序类型、漏洞数量、漏洞类型等因素,但普遍将目标程序当作一个黑盒,在测试集构建以及效能评判过程中普遍忽略了妨碍特征的影响,使妨碍特征成为影响测评结果的未知变量,这不仅导致测评结果可解释性差,也进一步影响了测评在工具使用及改进方面的指导作用。

针对上述问题,本文将妨碍特征纳为模糊测试工具测评的控制变量之一,并为此构建具有妨碍特征标注信息的测试集,通过对比测试,量化妨碍特征对测评结果的影响,以此推断被测工具的能力,验证技术有效性,提升测评的可解释性,进而为模糊测试工具的使用与改进提供更详细的指导信息。为此,本文针对基于变异的模糊测试,从漏洞触发条件、模糊测试优化辅助技术对抗两个角度归纳了5种模糊测试妨碍特征,然后采用代码合成的方式生成包含这些妨碍特征的目标程序来构建测试集,用于模糊测试工具的测评。为了验证上述方法的效果,本文构建了包含118个目标程序的测试集Bench4I(Benchmark for Interpretable Fuzzing Evaluation),对AFL、AFLFast、MOPT、TortoiseFuzz、QSYM、Angora 6款技术特点各异的模糊测试工具进行了测评,在被测工具做了匿名的前提下对它们的能力进行了推断,并对它们的优缺点进行了分析。结果表明,被测工具对Bench4I中不同目标程序的模糊测试结果有明显的差异,且结果差异与目标程序的妨碍特征有明显的关联关系,能够明确地推断出被测工具的技术特点,提升测评结果的可解释性。最后,本文根据测评得到的结论,对部分被测工具提出了实际使用时的配置建议,以及进一步提升模糊测试效能的改进思路。

本文的主要贡献如下:

1) 提出了将目标程序的模糊测试妨碍特征纳入控制变量范畴的模糊测试工具测评方法,可从测评结果中获得单个妨碍特征对模糊测试的具体影响程度,以此推断工具的具体能力或技术特点,提升测评结果的可解释性。

2) 针对基于变异的模糊测试,从漏洞触发条件、模糊测试辅助优化技术对抗两个角度归纳了5种妨碍特征,量化地总结了这些特征对不同模糊测试技术的影响效果以及不同影响效果所对应的工具技术能力特点。实现了一套目标程序合成脚本,并用其合成了118个包含不同妨碍特征的目标程序,构建

了测试集Bench4I。

3) 使用Bench4I对6款技术特点各不相同的模糊测试工具进行了测评,对它们的能力进行了推断,并针对每个工具详细分析其技术优势与瓶颈,最后根据结论提出了工具实践使用时的配置建议以及改进思路。实践了QSYM的改进思路,对于LAVA-M中的目标程序,改进后的QSYM在2h内的漏洞触发数量平均比改进前多出约17个。

文章结构如下:第2节介绍相关工作;第3节对本文提出的模糊测试工具测评方法进行概述;第4节具体介绍本文归纳的5种模糊测试妨碍特征及测试集构建方法;第5节介绍测评实验并分析测评结果数据;第6节对全文进行总结并展望未来工作。

## 2 相关工作

### 2.1 模糊测试技术

模糊测试可分为三个主要环节:产生输入数据、运行目标程序、监控目标程序异常。不同的模糊测试技术对于这些环节的具体实现有所不同,从输入数据产生方式的角度,模糊测试可大致分为两类,如表2所示。

表2 模糊测试技术分类与特点  
Table 2 Techniques of fuzz testing

分类	代表性工具	技术特点
基于生成的模糊测试	Peach <sup>[10]</sup>	基于事先定义的格式模板生成输入数据
	Trinity <sup>[11]</sup>	
	vUSBf <sup>[12]</sup>	
	Radamsa <sup>[13]</sup>	盲目随机变异
	ZZUF <sup>[14]</sup>	
基于变异的模糊测试	AFL	N/A
	AFLFast <sup>[15]</sup>	代码覆盖反馈策略优化
	TortoiseFuzz <sup>[16]</sup>	
	CollAFL <sup>[17]</sup>	
	AFLGo <sup>[18]</sup>	变异操作调度优化
	MOPT <sup>[19]</sup>	
	QSYM <sup>[20]</sup>	符号执行辅助种子变异
	Driller <sup>[21]</sup>	
	GREYONE <sup>[22]</sup>	污点分析辅助种子变异
	Angora <sup>[23]</sup>	
	VUzzer <sup>[24]</sup>	数据类型推断辅助变异
	TIFF <sup>[25]</sup>	
	RNNfuzzer <sup>[26]</sup>	机器学习优化种子变异
	Neuzz <sup>[27]</sup>	

#### 2.1.1 基于生成的模糊测试

在产生输入数据的环节,基于生成的模糊测试(generational fuzzing)按照事先定义好的格式来生成输入数据,以满足目标程序对输入数据格式的要求。Peach 就是一款基于生成的模糊测试工具,它以

Peach Pit 来定义目标程序的输入数据格式模板。Trinity 和 vUSBf 则分别能够针对 Linux 内核与 USB 驱动程序生成输入数据。然而, 数据格式模板的定义通常依赖人工完成, 并且需要对目标程序有深入的了解, 不精确、不全面的数据格式定义会严重影响该类模糊测试的效果。

2.1.2 基于变异的模糊测试

基于变异的模糊测试(mutational fuzzing)通过对种子(seeds)进行变异来生成输入数据。种子是事先收集的目标程序的输入数据。基于变异的模糊测试会对种子执行变异操作来产生新的输入数据。这种方式不需要事先了解目标程序的输入数据格式, 具有更好的可扩展性。最原始的该类模糊测试不会对种子进行迭代更新, 变异操作也是盲目、随机的, 难以触发条件约束下的代码区域, 因此需要尽可能选取能够覆盖目标程序更多代码的输入数据集合作为初始种子以达到良好的模糊测试效果。

基于代码覆盖反馈的模糊测试是近年来应用最广的模糊测试技术之一, 其工作流程如算法 1 所示, 其中, 种子能量(energy)代表该种子被选取后所执行的变异操作次数。基于代码覆盖反馈的模糊测试会动态地收集目标程序当前的代码覆盖信息, 然后基于这些信息, 以某种方式衡量当前输入数据的“价值”, 进而判断是否将其作为新种子(对应算法 1 中的 isInteresting 函数), 使得对通过筛选的种子进行变异所产生的输入数据有更高概率触发漏洞。算法 1 的时间复杂度与被测程序中可行路径的数量成正比, 由于程序中通常包含循环、递归等导致路径数量呈指数级增加的因素, 因此, 模糊测试工具在实际使用时通常会设置一个运行时间上限(对应算法 1 中的第 14 行)。AFL 是经典的基于代码覆盖反馈的模糊测试工具, 它通过判断输入数据是否触发了新的代码区域来决定是否将其作为新的种子加入到种子队列。

在代码覆盖反馈技术的基础上进行改进的研究也有很多。AFLFast 将代码覆盖信息用于种子调度优先级和种子能量的计算(对应算法 1 中的 chooseNextSeed 和 assignEnergy 函数)。TortoiseFuzz 根据种子能够覆盖的代码的安全的相关性来计算其“价值”, 并为高“价值”的种子赋予更高的变异优先级和能量, 使得模糊测试更倾向于触发包含安全敏感代码的区域。AFLGo 基于代码覆盖反馈机制实现了导向模糊测试(directed fuzzing), 能够使模糊测试逐渐朝着目标代码区域靠近。上述研究改进了代码覆盖反馈指导策略, 但在种子变异上仍是盲目、随机的。为了对种子

算法 1.基于代码覆盖反馈的模糊测试

```
输入: 初始种子 S
输出: 能够使目标程序出现异常的数据 O
1: O = ∅
2: Q = S
3: REPEAT
4:   s = chooseNextSeed(Q)
5:   e = assignEnergy(s)
6:   FOR i = 1 to e DO
7:     s' = mutate(s)
8:     IF s' crashes THEN
9:       O = O ∪ s'
10:    ELSE IF isInteresting(s') THEN
11:      Q = Q ∪ s'
12:    END IF
13:  END FOR
14: UNTIL timeout or aborted
```

变异进行优化, MOPT 基于不同变异操作在不同时间对提升目标程序代码覆盖率的贡献有所不同这一事实, 利用粒子群优化算法(Particle Swarm Optimization, PSO)优化变异操作的调度。RNNFuzzer 和 Neuzz 则将模糊测试视为搜索问题, 利用机器学习对变异操作进行优化。另一方面, 混合模糊测试(hybrid fuzzing)通过融合符号执行、污点分析等技术来指导种子变异。例如 QSYM、Driller 利用符号执行对条件约束进行求解, 然后生成能够满足条件约束的输入数据。VUzzer、Angora 则利用污点追踪找到与条件约束有关的输入字节, 然后指导模糊测试重点对这些字节进行变异。

2.2 模糊测试工具测评

模糊测试工具的测评通常首先运行工具对目标程序执行模糊测试, 然后按照事先定义好的效能度量指标收集运行过程或结果数据, 最后对这些数据进行对比、分析, 得到测评结论。

2.2.1 测试集

现有模糊测试工具测评常用的测试集可分为两类。一类基于真实漏洞构建。例如 Google fuzzer testsuite 和 FuzzBench, 它们包含 25 个真实开源项目, 并将它们包含的真实漏洞作为“标准答案”, 后者在此基础上把 OSS-Fuzz 平台上的开源项目也作为测试集的一部分, 把平台上已经提交并通过审计的漏洞作为标准答案。这类测试集受真实漏洞数量和发现效率的约束, 测试集的规模受限且迭代更新缓慢。另一类测试集则通过漏洞植入等方式人工构造漏洞作为“标准答案”, 例如 Dolan-Gavitt 等人<sup>[5]</sup>实现的



LAVA(Large-scale Automated Vulnerability Addition)能够利用污点分析技术自动向真实程序中植入漏洞。

但是,上述所有测试集均未标注程序的代码特征信息,将目标程序视为黑盒,同时,现有测评方法也通常忽略目标程序对模糊测试的影响。对此,Xiaogang Zhu 等人<sup>[28]</sup>利用程序静态分析技术统计程序的路径条数、魔数检查个数、校验和检查个数作为额外标注信息,构建了 FEData 测试集。但是 FEData 归纳的特征类型及特征标注方法仍有很大局限性,目标程序中大量其它影响模糊测试的代码特征仍无法标注。

## 2.2.2 测评方法

George Klees 等人<sup>[4]</sup>调研了 32 篇关于模糊测试的论文,分析了文献中的实验部分,系统性地总结了若干模糊测试工具测评原则。首先,被测工具需对目标程序执行多次独立重复的模糊测试,再利用统计学方法对结果数据进行处理,其原因是模糊测试具有随机性,通过多次测试能够在一定程度上消除随机性对测评结果的影响。第二,由于初始种子对模糊测试效果影响很大,文章建议测评使用多组不同的初始种子进行多次测试,防止使用单一种子导致测评结果存在偏差,文章同时认为使用空文件作为初始种子也可以避免偏差,获得与被测工具真实效能相匹配的测评结果。第三,文章建议模糊测试的超时时限设置为 24 h 或以上,防止因运行时间过短而引入偏差。上述三点测评原则旨在消除偏差,确保测评能够反映被测工具的真实效能。

在效能度量指标上,最直观的是模糊测试触发的漏洞数量。大部分模糊测试工具能够捕获目标程序运行过程中的崩溃(crash),或使用 Address Sanitizer 和 Undefined Behavior Sanitizer(ASAN 和 UBSAN)<sup>[29]</sup>来捕获内存异常状态,以此监控内存破坏类漏洞的触发。模糊测试工具还需要对捕获到的重复漏洞触发进行去重,这可以通过异常调用栈比对等方式实现。此外,目标程序的代码覆盖率同样能够作为模糊测试工具效能的度量指标之一,因为通常来讲,模糊测试覆盖到的目标程序代码越多,触发漏洞的可能性越高。

## 3 方法概述

### 3.1 现有测评方法的缺陷

如前言部分的定义,我们将目标程序本身包含的可能妨碍模糊测试策略、算法或辅助技术发挥其应有功效的特征称为模糊测试妨碍特征。现有模糊测试工具测评方法普遍存在的问题是忽略了妨碍特

征对测评结果的影响。

在模糊测试研究领域,测评通常用于验证研究提出的新技术的有效性,令未采用及采用了新技术的工具对相同的目标程序进行模糊测试,然后对比结果。但是由于目标程序包含的妨碍特征会对测评结果产生影响,因此实际上我们不能单纯地根据测评结果判断新技术的有效性。例如前言部分所举的例子,QSYM 通过符号执行指导种子变异,提升了模糊测试的效率,但当目标程序包含符号执行无法求解的校验和检查时,QSYM 的漏洞发现效率会降低到与 AFL 相近。忽略目标程序中的妨碍特征会使其成为影响测评结果的未知变量,导致无法解释测评结果差异背后的原因。因此,我们需要改进现有测评方法,将目标程序中影响模糊测试的因素纳入考虑范围,达成更细粒度的测评,从而能够对被测工具的能力或技术特点做出推断。

### 3.2 改进思路

本文提出的测评方法的核心思路是将模糊测试妨碍特征纳为测评的控制变量之一。控制变量法常用于科学实验中,它只将实验中的某个单一因素作为自变量,其它因素作为控制变量(controlled variables)保持不变,从而把多因素问题转化为多个单因素问题,以便能够明确单个因素对因变量的影响。

对于模糊测试工具测评来说,因变量是测评结果数据,而目标程序作为模糊测试工具的作用对象,其本身包含的妨碍特征则是重要的影响因变量的因素之一,在测评中对妨碍特征实施变量控制即能够测出各妨碍特征对模糊测试的具体影响程度。例如图 2,在对目标程序 1 和 2 的模糊测试中,我们令其它条件都相同,并使程序 1 和 2 的妨碍特征差异  $\Delta feature$  仅包含单一妨碍特征(在图 2 的示例中, $\Delta feature$  为妨碍特征 2),此时便可确定导致结果 1 与结果 2 之间差异  $\Delta result$  的因素便是该妨碍特征(需利用多次独立重复试验在一定程度上消除模糊测试随机性对结果的影响)。

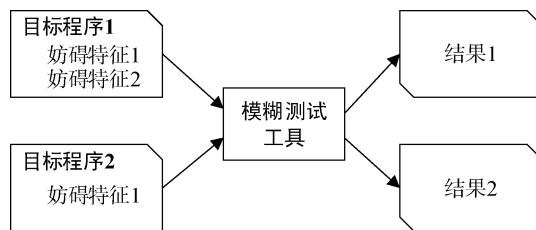


图 2 将模糊测试妨碍特征作为控制变量的对比测评  
Figure 2 Fuzzing-hampering features as controlled variables during evaluation

本文从漏洞触发条件、模糊测试优化辅助技术对抗两个角度归纳了 5 种妨碍特征(详见第 4 节), 并在测评中实现对妨碍特征的变量控制, 从而得到各妨碍特征  $\Delta feature$  对测评结果的具体影响程度, 即  $\Delta result$ , 再据此推断被测工具的能力或技术特点, 在更细粒度的层面对工具效能进行评估, 提高测评结果的可解释性。

### 3.3 测试集

为了实现上述测评思路, 首先需要对目标程序中的模糊测试妨碍特征进行标注, 此外, 不同目标程序的妨碍特征需存在类似于图 2 所示的差异, 以便能够在测评中实施对妨碍特征的变量控制。

真实程序代码规模大、复杂性高, 难以准确、全面地对其中的妨碍特征进行标注, 不同程序之间的妨碍特征差异也不可控。另一方面, 我们发现在例如文献[20-25]的模糊测试研究中使用了玩具程序(toy program)来说明模糊测试技术瓶颈, 这些玩具程序通常仅由几十至上百行代码构成, 包含特定妨碍特征, 复杂度不及真实程序, 但能够清晰地展示妨碍特征对模糊测试的影响。综上, 本文采用代码合成的方式生成包含不同妨碍特征的目标程序来构建测试集。代码合成具有高可控性, 使我们能够根据测评需求生成包含不同妨碍特征的目标程序, 达成对妨碍特征的变量控制。

有关模糊测试妨碍特征以及测试集的具体构建方法详见第 4 节。

### 3.4 测评流程

我们把模糊测试工具的测评分为 3 个主要环节: 环境参数设置、测评结果数据收集、结果数据分析。

#### 3.4.1 环境参数设置

测评的软、硬件环境及参数设置对测评结果是有影响的, 为了获得妨碍特征对测评结果的影响, 我们同样需要将这些因素作为控制变量, 使其在测评中保持一致。对此, 我们主要依照文献[4]提出的原则确定了种子文件选取、超时时限设置、独立重复测试次数、异常监控等具体设置, 详见第 5 节。

#### 3.4.2 测评结果数据收集

作为测评的因变量, 模糊测试工具运行的结果数据能够直接或间接地体现工具的效能。本文在测评中具体收集的数据如下:

1) 漏洞触发时耗/目标程序执行次数: 即从模糊测试开始执行到成功触发漏洞所耗费的时间(*time*)以及在这段时间内目标程序的执行次数(*execs*)。该结果数据能直观地反映被测工具触发漏洞的效率。

2) 代码触发时空分布: 以目标程序的运行次数代替时间维度, 以目标程序的基本块为空间维度, 代码触发时空分布描述了模糊测试中, 目标程序每次运行时各基本块被执行的情况。设函数  $e(bb, i)$  代表基本块  $bb$  在目标程序第  $i$  次运行中是否被触发, 函数值为 0 代表未触发, 为 1 代表被触发(包括被多次触发), 我们将该函数定义为代码触发时空分布函数。基于代码触发时空分布函数, 我们可以进一步定义许多其它表示, 例如函数  $f(x, bb) = \sum_{i=1}^x e(bb, i)$  表示随着目标程序运行次数的增加, 基本块  $bb$  被触发次数的变化趋势, 满足  $f(x, bb)=1$  的最小  $x$  值则表示基本块  $bb$  首次被触发时目标程序已经被运行的次数。相较于代码覆盖率, 代码触发时空分布数据能够区分不同基本块的触发情况, 包含更多信息量, 便于分析妨碍特征对模糊测试的影响。

#### 3.4.3 结果数据分析

我们将妨碍特征作为测评的控制变量, 着重从回答“同一工具对不同目标程序的模糊测试结果差异是什么原因导致的”的角度分析测评结果数据, 辅以不同工具间的对比, 提升测评的可解释性。

设  $\Delta result$  为模糊测试工具对两个目标程序的测试结果差异,  $\Delta feature$  为这两个目标程序包含的妨碍特征差异。把妨碍特征作为控制变量, 令  $\Delta feature$  仅包含单个妨碍特征, 便可以明确地把  $\Delta result$  归因于这单个妨碍特征上, 建立它与  $\Delta result$  之间的关联关系, 反映此妨碍特征对模糊测试的影响。例如  $\Delta result / \Delta feature$  可表示妨碍特征对测评结果的影响程度。由于妨碍特征对不同的模糊测试技术有不同的妨碍作用, 因此我们可以根据  $\Delta result / \Delta feature$  来推断模糊测试工具的能力(如表 3)。

## 4 测试集构建

为了实现本文提出的模糊测试工具测评方法, 我们归纳了 5 种模糊测试妨碍特征, 并利用代码合成的方式生成测试集, 具体如下。

### 4.1 模糊测试妨碍特征

本文从漏洞触发条件、模糊测试优化辅助技术对抗这两个角度, 针对目前应用更广泛的基于变异的模糊测试, 归纳了 5 种妨碍特征, 并按照 3.4.3 节所述, 量化了这些妨碍特征对不同的模糊测试技术的影响程度, 进而根据影响程度对模糊测试工具的能力进行推断, 如表 3 所示。

漏洞触发条件通常要求输入数据的特定字节取特定值, 我们将其称为输入数据的漏洞触发取值,

而模糊测试可以视为从输入空间中搜索漏洞触发取值点的过程, 直观地, 输入空间大小以及漏洞触发

取值个数影响着搜索的难度。综上, 从影响漏洞触发条件的角度, 本文定义了以下两个妨碍特征。

表 3 基于妨碍特征对模糊测试的影响推断模糊测试工具的能力

Table 3 Inferring capabilities of a fuzzer based on the influence of fuzzing-hampering features on fuzzing

妨碍特征	妨碍特征对模糊测试的影响程度与能力推断			
	影响程度	推断	影响程度	推断
$N / N_V$	$\Delta execs$ 与 $\Delta(N / N_V)$ 成正比, 比例系数近似于 1 或大于 1	A: 模糊测试的种子变异算法近似于随机变异, 且比例系数越小, 算法优化程度越高	$\Delta execs / \Delta(N / N_V)$ 接近于 0	B: 模糊测试实现了某种技术对种子变异进行定向指导
$L$	$\Delta execs / \Delta L$ 大于 1	C: 模糊测试无法区分漏洞相关/无关字节	$\Delta execs / \Delta L \leq 1$	D: 模糊测试能够区分漏洞相关/无关字节, 从而只针对漏洞相关字节进行变异
NOI	$execs_{NOI}$ 为 $execs_{orig}$ 的两倍以上	E: 模糊测试实现了代码覆盖反馈指导策略, 模糊测试会陷入大量噪声路径中	$execs_{NOI}$ 约等于或小于 $execs_{orig}$	F: 模糊测试未实现代码覆盖反馈策略; 或模糊测试实现了某种技术消除了噪声路径的影响
SH	$execs_{SH}$ 为 $execs_{orig}$ 的两倍以上	G: 模糊测试实现了符号执行技术辅助模糊测试	$execs_{SH}$ 约等于或小于 $execs_{orig}$	H: 模糊测试并未实现符号执行技术; 或者实现的符号执行技术能克服该妨碍特征
TH	$execs_{TH}$ 为 $execs_{orig}$ 的两倍以上	I: 模糊测试实现了污点分析技术辅助模糊测试	$execs_{TH}$ 约等于或小于 $execs_{orig}$	J: 模糊测试并未实现污点分析技术; 或者实现的污点分析技术能克服该妨碍特征

(注: 对影响程度的划分以 AFL 的实际测试表现为参考进行设置。当考虑一个妨碍特征对模糊测试的影响时, 默认其它妨碍特征相同。)

**漏洞触发取值个数比( $N / N_V$ ):** 即输入空间包含的值的总数与空间中漏洞触发取值点的个数之比。例如图 4 所示代码中, VAR1 需要等于 0xab 来满足第 26 行的条件约束, VAR2 需要等于 0xcd 来满足第 27 行的条件约束, 输入数据的其余 6 个字节则可取任意值, 因此第 28 行漏洞的漏洞触发取值个数  $N_V$  为  $2^{48}$ , 输入空间包含的值的总数为  $N=2^{8*8}$ , 假设模糊测试对种子的变异是完全随机且独立的, 且每次变异成输入空间中任意一点的概率相同, 则  $N_V / N$  可代表每次变异产生的值能触发漏洞的概率,  $N / N_V$  则代表漏洞触发时模糊测试已执行的变异次数的期望值, 图 4 示例的该值为  $2^{16}$ , 它可以代表种子变异完全随机的情况下, 模糊测试触发漏洞的难度。在能力推断上, 我们设  $execs$  为漏洞触发时模糊测试已执行的变异次数, 若  $\Delta execs$  与  $\Delta(N / N_V)$  成正比, 比例系数近似于 1 或大于 1, 说明模糊测试对  $N / N_V$  的变化较为敏感, 可以推断种子变异算法有随机性质, 而当  $\Delta execs / \Delta(N / N_V)$  接近于 0 时, 说明模糊测试不再受  $N / N_V$  的影响, 可推断模糊测试工具实现了某种技术对种子变异进行定向指导。

**输入数据长度(L):** 即目标程序读取的输入数据所包含的字节数, 若目标程序读取的输入数据长度不固定, 则取其最小合法长度。输入数据可分为漏洞

相关字节和漏洞无关字节, 前者的取值决定着能否漏洞触发, 后者则无论取何值都无关乎漏洞的触发。在图 4 中, VAR1 和 VAR2 是漏洞相关变量, 构成这两个变量的字节即漏洞相关字节, 而构成 input 数组后 6 个元素的字节则是漏洞无关字节。输入数据长度除了能够决定输入空间的大小, 在漏洞相关字节相同的情况下, 输入数据长度越长, 模糊测试定位漏洞相关字节难度越高, 从而影响漏洞触发效率。因此在能力推断上,  $\Delta execs$  与  $\Delta L$  比值越大, 说明模糊测试受输入数据长度变化越明显, 参考 AFL 的实际表现, 我们拟定当该比值  $>1$  时推断模糊测试无法区分输入数据中的漏洞相关/无关字节, 而当该比值  $<1$  时推断模糊测试实现了某种技术进而能够在一定程度上区分漏洞相关/无关字节, 从而只针对漏洞相关字节进行变异而不受  $L$  变化的影响。

模糊测试的优化或辅助增强技术, 如代码覆盖反馈、符号执行、污点分析等, 可能存在缺陷, 目标程序的某些特征会使这些技术失效。因此, 从模糊测试优化辅助技术对抗这个角度, 本文归纳了针对代码覆盖反馈、符号执行、污点分析的妨碍特征。

**噪声路径(NOI):** 我们把程序执行路径分为噪声路径和漏洞路径。漏洞路径是漏洞触发过程中所执行的程序路径, 而与漏洞触发无关的程序路径则是

噪声路径。例如图 4 第 26~28 行代码属于漏洞路径, 而第 29~34 行代码属于噪声路径。若噪声路径形成一定规模, 代码覆盖反馈机制有可能会使模糊测试陷入其中, 导致漏洞发现效率降低。Jinho Jung 等人<sup>[30]</sup>和 Emre Güler 等人<sup>[31]</sup>均通过向目标程序中引入大量噪声路径来对抗模糊测试的代码覆盖反馈机制。在能力推断上, 假设两个目标程序除去噪声路径规模外, 其它条件完全相同, 设  $execs_{orig}$  为模糊测试触发原始目标程序的漏洞所需变异次数,  $execs_{NOI}$  为向原始目标程序引入大量噪声路径后, 触发其漏洞所需变异次数。参考 AFL 的实际表现, 我们拟定当  $execs_{NOI}$  为  $execs_{orig}$  的两倍以上时推断模糊测试可能实现了代码覆盖反馈指导策略, 导致  $execs_{NOI}$  增大。比值越小, 说明噪声路径对模糊测试的影响越小, 若  $execs_{NOI}$  约等于  $execs_{orig}$ , 则说明模糊测试可能未实现代码覆盖反馈策略, 也可能实现了某种技术, 如导向模糊测试, 来消除噪声路径的影响。

**符号执行妨碍特征(SH):** 这些特征通常是现有符号执行技术难以分析、求解的程序语句, 例如浮点运算、哈希函数等, 它们能够妨碍模糊测试利用符号执行技术指导种子变异来突破目标程序里的条件约束。在能力推断上, 我们设  $execs_{SH}$  为原始目标程序引入符号执行妨碍特征后, 触发其漏洞所需变异次数。我们拟定当  $execs_{SH}$  为  $execs_{orig}$  的两倍以上时推断模糊测试对该特征敏感, 进而推断其可能实现了符号执行技术。若  $execs_{SH}$  与  $execs_{orig}$  相接近, 则说明模糊测试可能未使用符号执行技术, 或者实现的技术有能力克服符号执行妨碍特征。

**污点分析妨碍特征(TH):** 这些特征通常是污点分析难以准确追踪的数据流, 例如隐式数据流, 它们能够妨碍模糊测试利用污点分析技术来指导种子变异。在能力推断上, 我们设  $execs_{TH}$  为向原始目标程序引入污点分析妨碍特征后, 触发其漏洞所需变异次数。我们拟定当  $execs_{TH}$  为  $execs_{orig}$  的两倍以上时推断模糊测试对该特征敏感, 进而推断其可能实现了污点分析技术。若  $execs_{TH}$  与  $execs_{orig}$  相接近, 则说明模糊测试可能未使用污点分析技术, 或者实现的技术有能力克服污点分析妨碍特征。

## 4.2 目标程序合成

本文基于 Python 实现了一套目标程序自动合成脚本, 其工作流程如图 3 所示。首先, 脚本会读取输入数据长度和漏洞相关字节数来生成配置文件。配置文件中定义了漏洞触发相关/无关变量的变量名、数据类型以及变量在输入数据中所占的字节, 然后基于这些变量生成条件约束并回写到配置文件里。

配置文件也可直接按规定格式进行手工定义。接下来, 脚本根据配置文件里定义的条件约束自动生成结构文件。结构文件定义了目标程序的漏洞触发路径和妨碍特征等配置信息。最后, 脚本根据结构文件生成包含妨碍特征的源代码片段, 将其植入到模板文件中, 构造出可编译可执行的目标程序。

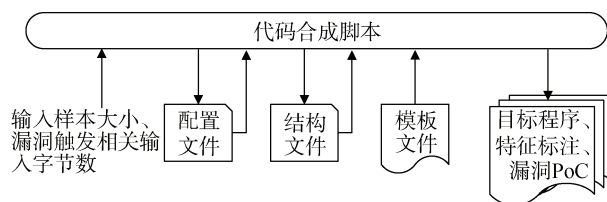


图 3 代码合成脚本工作流程

Figure 3 Workflow of the code generation Script

其中, 模板文件包含对主函数、通用功能函数、全局变量、漏洞函数等的定义, 并标记了代码植入点。脚本根据结构文件生成源代码的算法复杂度为  $O(n)$ ,  $n$  为结构文件内容的行数。结构文件生成的代码片段插入到模板文件的代码植入点即可形成一份完整的程序源代码。此外, 脚本能够在构造目标程序的过程中自动生成漏洞 PoC 以及妨碍特征的标注信息。在代码合成的过程中, 模糊测试妨碍特征的具体实现如下:

1) 输入数据长度  $L$  通过 `read` 函数实现。如图 4 第 20 行代码, 该函数的第 3 个参数决定了程序读入的字节长度。在代码合成过程中, 该参数的位置会被替换为输入数据长度。

```

1  int main(int argc, char** argv) {
2      unsigned char input[8];
3      ...
20     read(fd, input, 8);
21     unsigned char VAR1 = input[0];
22     unsigned char VAR2 = input[1];
23     unsigned char UTVAR3 = input[2];
24     unsigned int UTVAR4 = read_int(&input[3], 4);
25     unsigned char UTVAR5 = input[7];
26     if (VAR1 == 0xab){
27         if (VAR2 == 0xcd){
28             bug();
29         } else { //NOISE
30             ...
31         } else { //NOISE
32             if (UTVAR3 == 0xef)
33                 printf("hello\n");
34         }
35     }
36     ...

```

图 4 合成代码示例

Figure 4 An example of synthesized code

2) 漏洞触发取值个数  $N_V$  由漏洞触发路径上的条件约束决定。我们使用嵌套的 `if-else` 语句来构成漏洞路径, 如图 4, 漏洞放置在嵌套的最内层, 只有



当输入数据满足路径上的所有条件约束时, 程序才能运行到漏洞所在的基本块, 此时所有满足这些约束条件的输入数据取值个数即漏洞触发取值个数。

设  $\text{num}(\text{byte}_i, \text{cond})$  表示当输入数据满足条件约束  $\text{cond}$  时, 第  $i$  个字节所有可能取值的个数。所有满足

$\text{cond}$  的输入数据取值个数则为  $\prod_{i=1}^L \text{num}(\text{byte}_i, \text{cond})$ ,

其中  $L$  为输入数据长度。此外, 在我们生成的目标程序中, 一个漏洞相关变量只能出现在一个分支语句的条件约束中, 且一个分支语句的条件约束只能包含一个漏洞相关变量, 因此在本文构造的目标程序中, 漏洞相关变量与漏洞路径上的分支语句一一对应、数量相同。

3) 噪声路径基于漏洞无关变量生成。我们利用漏洞无关变量构造大量分支语句, 将其植入原始目标程序, 使模糊测试对漏洞无关变量的变异有高概率触发噪声路径下的新基本块, 误导代码覆盖反馈机制, 使模糊测试陷入到噪声路径中, 增加对漏洞无关字节的变异次数, 降低触发漏洞的概率。

4) 对于符号执行妨碍特征, 本文设计了两种实现方法。第一种方法利用循环冗余校验实现。我们在模板文件里定义了一个能够计算任意长度字节数组 CRC 校验和的函数, 然后利用该函数将条件约束逻辑表达式中的常量、变量替换为它们的 CRC 校验和, 从而引入符号执行妨碍特征。第二种方法利用一

维高斯函数  $f(x) = ae^{-\frac{(x-b)^2}{2c^2}}$  实现。假设我们希望漏洞相关变量  $\text{VAR}$  满足  $\text{VAR} \in (\text{lower}, \text{upper})$  时触发漏洞, 那么取  $b = (\text{lower} + \text{upper})/2$ ,  $c = \text{lower} - b$ , 并求解出  $f(\text{lower}) = 1$  时, 参数  $a$  的值, 然后将参数  $a$ 、 $b$ 、 $c$  代入  $f(x)$  得到一维高斯函数, 如图 5(a) 所示。通过上述方法得到的高斯函数, 当  $x \in (\text{lower}, \text{upper})$  时  $f(x) \geq 1$ , 否则  $0 < f(x) < 1$ , 此时构造如图 5(b) 所示代码, 可知当且仅当输入变量  $\text{VAR} \in (\text{lower}, \text{upper})$  时满足函数值  $\geq 1$  的条件约束, 从而触发漏洞。由于上述高斯函数参数的求解过程包含幂运算、浮点运算等现有符号执行技术难以处理的操作, 因此同样能够作为符号执行妨碍特征。

5) 对于污点分析妨碍特征, 我们采用隐式数据流来实现。如图 6, 输入变量  $\text{var}$  的值通过影响程序控制流来间接地影响变量  $\text{ch}$  的值, 进而影响数组  $\text{cc}$ , 即变量  $\text{var}$  通过隐式数据流将自身的值传递到了数组  $\text{cc}$  中。若污点分析技术无法识别隐式数据流, 就会导致其污点分析中断。

我们利用代码合成脚本构建了测试集 Bench4I (见表 4)。Bench4I 的目标程序按照特定格式命名, 例

```
// if lower<=VAR<=upper, return>=1
// else return 0
int f(int VAR, int lower, int upper) {
    double b = (lower + upper)/2.0; // b
    double c = pow((lower-b),2);    // 2c^2
    double fx = exp(-1.0);
    double a = 1.0 / fx;             // a
    double e = -1.0*pow((VAR-b),2)/c;
    double fx2 = a * exp(e);
    return (int)fx2;
}
```

(a)

```
//According to lower and upper,
//calculate a, b, c for Gaussian function "f"
...
if(f(VAR) >= 1)
    bug();
```

(b)

图 5 利用高斯函数实现符号执行妨碍特征

Figure 5 Using Gaussian function to implement fuzzing-hampering feature against symbolic execution

```
unsigned char cc[var_size];
for (i=0; i<var_size; i++) {
    ch = 0; temp = (var>>i*8) & 0x000000FF;
    for (int j=0; j<8; j++) {
        temp2 = temp & (1<<j);
        if (temp2!=0) ch |= 1<<j;
    }
    cc[var_size-1-i] = ch;
}
```

图 6 利用隐式数据流实现污点分析妨碍特征

Figure 6 Using implicit dataflow to implement fuzzing-hampering feature against taint analysis

如 “IS8\_TS4\_TV1\_\_1”, 其中第一个数字为输入数据长度, 第二个数字为漏洞相关字节数, 第三个数字为漏洞相关变量个数, 第四个数字是额外序号。

表 4 Bench4I 中的目标程序

Table 4 Target programs in Bench4I

测试子集名称	目标程序个数	妨碍特征
Bench4I-orig	29	$L, N / N_F$
Bench4I-noise	17	$L, N / N_F, \text{NOI}$
Bench4I-CRC	18	$L, N / N_F, \text{SH}$
Bench4I-gaussian	25	$L, N / N_F, \text{SH}$
Bench4I-IDF	29	$L, N / N_F, \text{TH}$
总和	118	$L, N / N_F, \text{NOI}, \text{SH}, \text{TH}$

Bench4I 包含 5 个子集: Bench4I-orig 由 29 个原始目标程序构成, 它们在输入数据长度、漏洞触发取值个数比上存在差异; Bench4I-noise 通过向原始目标程序中植入噪声路径构成, 噪声路径的植入点通常选择在构成漏洞路径的 if-else 分支语句中的 else 分支上, 类似于图 4 第 32、33 行代码; Bench4I-CRC 通过将原始目标程序条件约束中的常量和变量替换为

它们的 CRC 校验生成, 例如将 “var1 == 0xab” 替换为 “crc(var1) == crc(0xab)” ; Bench4I-gaussian 则如图 5 所示, 以高斯函数替换原始目标程序的分支条件约束来生成; Bench4I-IDF 则以图 6 所示方法在原始目标程序的漏洞相关变量与漏洞触发点之间引入隐式数据流。Bench4I 中的所有目标程序都由同一个模板文件合成, 除妨碍特征外不存在其它差异。

## 5 测评实验结果及分析

为了验证本文方法的效果, 我们使用 Bench4I 对六款技术特点各异的模糊测试工具进行了测评。测评中, 我们对这些工具进行了匿名, 然后根据测评结果数据对它们的能力或技术特点进行推断。

对于测评环境及参数, 我们主要依照文献[4]提出的原则进行设置。测评的硬件环境统一为 Intel Xeon E312xx 处理器(双核 2.4Ghz)、4GB 内存的虚拟机, 操作系统为 64 位 Ubuntu 16.04。初始种子集合统一包含 8 个种子文件, 这 8 个种子分别由  $n$  个 0x00、0x22、0x44、0x66、0x88、0xAA、0xCC、0xEE 构成, 其中  $n$  等于目标程序的输入数据长度。模糊测试超时时限由于 Bench4I 的目标程序代码规模小且仅包含一个漏洞, 因此设置为 2 h。独立重复测试次数设置为 5 次。异常监控方面, 本文使用 Clang 实现的 sanitizer 实现, 并将漏洞触发调用栈作为漏洞的唯一标识。

### 5.1 测评结果

我们分别使用六款模糊测试工具对 Bench4I 的目标程序进行模糊测试并记录结果数据, 包括模糊测试触发目标程序的漏洞所需变异次数, 即漏洞触发时目标程序被执行次数  $execs$ , 以及代码触发时空分布数据。按照 3.4.3 节所述, 对于包含单一妨碍特征差异的目标程序, 我们计算它们两两之间的  $\Delta execs$ , 以此建立妨碍特征与  $\Delta execs$  的关系, 量化妨碍特征对各模糊测试的影响程度, 如表 5 所示。

$\Delta execs / \Delta(N / N_V)$  代表漏洞触发取值个数比模糊测试结果的影响程度。在本测评中, 我们挑选 EBench4I-orig 中  $N / N_V$  互有差异的目标程序, 根据被测工具对它们的模糊测试结果计算所有  $\Delta execs / \Delta(N / N_V)$ , 并求它们的平均值。如表 5 所示, 工具 1、2、3、4 的  $\Delta execs / \Delta(N / N_V)$  值均  $>1$ , 因此根据表 3 做出推断 A, 而工具 5、6 的  $\Delta execs / \Delta(N / N_V)$  比值分别为 0.014 和 0.012, 接近于 0, 因此根据表 3 做出推断 B。

$\Delta execs / \Delta L$  代表输入数据长度对模糊测试结果的影响程度。在本测评中, 我们根据被测工具对目标程序 IS4\_TS1\_TV1\_\_1(输入数据长度为 4 字节)、IS4K\_TS1\_TV1\_\_1(输入数据长度为 4096 字节)和 IS10K\_TS1\_TV1\_\_1(输入数据长度 10240 字节)的模糊测试结果来计算。如表 5 所示, 工具 1 至 5 的  $\Delta execs / \Delta L$  均大于 1, 因此根据表 3 做出推断 C, 而工具 6 的  $\Delta execs / \Delta L$  仅为 0.12, 因此对其做出推断 D。

$execs_{NOI} / execs_{orig}$  是原始目标程序被植入噪声路径后与植入噪声路径前, 漏洞触发所需变异次数的比值, 能够反映噪声路径对模糊测试的影响程度。在本测评中, 我们根据被测工具对 Bench4I-noise 和 Bench4I-orig 的测试结果来计算  $execs_{NOI} / execs_{orig}$ 。如表 5 所示, 所有工具的  $execs_{NOI} / execs_{orig}$  都在 2 以上, 说明它们的漏洞触发效率都受到了噪声路径的较大影响, 因此可以做出推断 E。

$execs_{SH} / execs_{orig}$  是原始目标程序被植入符号执行妨碍特征后与植入前, 漏洞触发所需变异次数的比值, 能够反映符号执行妨碍特征对模糊测试的影响程度。在本测评中, 我们根据被测工具对 Bench4I-orig、Bench4I-CRC 和 Bench4I-gaussian 的模糊测试结果计算  $execs_{SH} / execs_{orig}$ 。如表 5 所示, 工具 5 的  $execs_{SH} / execs_{orig}$  为 28.6, 根据表 3 可以对其做出推断 G, 其余工具的  $execs_{SH} / execs_{orig}$  均接近于 1, 根据表 3 可做出推断 H。

表 5 妨碍特征对测评结果的影响程度及能力推断

Table 5 Fuzzing-hampering features' influences on evaluation results

	工具 1		工具 2		工具 3		工具 4		工具 5		工具 6	
	数值	推断	数值	推断	数值	推断	数值	推断	数值	推断	数值	推断
$\Delta execs / \Delta(N / N_V)$	20.3	A	14.8	A	5.6	A	14.7	A	0.014	B	0.012	B
$\Delta execs / \Delta L$	49.6	C	28.3	C	16.8	C	209.3	C	38.9	C	0.12	D
$execs_{NOI} / execs_{orig}$	$\geq 17$	E	4.8	E	$\geq 23$	E	17	E	50.2	E	57	E
$execs_{SH} / execs_{orig}$	1.17	H	0.68	H	1.01	H	0.17	H	28.6	G	1.05	H
$execs_{TH} / execs_{orig}$	0.49	J	1.29	J	1.56	J	0.54	J	4.9	I	63.4	I
能力推断(综合)	A, C, E, H, J		A, C, E, H, J		A, C, E, H, J		A, C, E, H, J		B, C, E, G, I		B, D, E, H, I	

$execs_{TH} / execs_{orig}$  是原始目标程序被植入污点分析妨碍特征后与植入前, 漏洞触发所需变异次数的比值, 能够反映污点分析妨碍特征对模糊测试的影响程度。在本测评中, 我们根据 Bench4I-orig 和 Bench4I-IDF 的模糊测试结果计算  $execs_{TH} / execs_{orig}$ 。如表 5 所示, 工具 5、6 的  $execs_{TH} / execs_{orig}$  分别为 4.9 和 63.4, 根据表 3 可以对其做出推断 I, 其余工具的  $execs_{TH} / execs_{orig}$  值均接近于 1, 可做出推断 J。

综上, 我们能够对工具 1 至 6 的能力做出以下推断: 工具 1~4 的种子变异算法近似于随机变异(推断 A)、无法区分漏洞相关/无关字节(推断 C)、实现了代码覆盖反馈指导策略且易受噪声路径的影响(推断 E), 结合推断 A 可确定它们并未实现符号执行或污点分析技术(推断 H、J); 工具 5、6 同样实现了代码覆盖反馈指导策略且易受噪声路径的影响(推断 E), 但实现了某种技术对种子变异进行定向指导(推断 B), 其中工具 5 实现了符号执行技术和污点分析技术(推断 G、D), 但无法区分漏洞相关/无关字节(推断 C), 而工具 6 实现了污点分析技术(推断 G)且能够区分漏洞相关/无关字节(推断 D), 但对符号执行妨碍特征不敏感的原因既可能是未实现符号执行技术, 也可能是其实现的符号执行技术有能力处理符号执行妨碍特征(推断 H)。

实际上, 工具 1~6 分别为 AFL、AFLFast、MOPT-AFL(MOPT 优化的 AFL)、TortoiseFuzz、QSYM 和 Angora, 我们根据测评结果以及表 3 做出的推断与这些模糊测试工具声明的能力基本相符, 这说明我们的测评方法能够准确地推断被测工具的能力。

## 5.2 结果分析

### 5.2.1 QSYM

根据 5.1 节的结论, 我们已知 QSYM 实现了代码覆盖反馈指导策略、符号执行技术和污点分析技术, 易受噪声路径的影响, 且无法区分漏洞相关/无关字节。为了进一步验证 QSYM 技术的有效性并分析其优缺点背后的具体原因, 为其实践使用与改进提供更多参考, 我们进行如下分析。

图 7(a)展示了符号执行妨碍特征对 QSYM 触发新代码区域的效率的影响。可以发现对于 AFL, 不论是否包含符号执行妨碍特征, 条件约束首次被满足所花费的运行次数较为接近, 而对于 QSYM 则存在很大差异。另一方面, 对于不包含符号执行妨碍特征的原始目标程序, QSYM 快速地通过了 8 层嵌套的分支条件约束并触发了漏洞, 但对于包含符号执行妨碍特征的目标程序, QSYM 的效率退化到与 AFL 一

致。由于 QSYM 实现的是附属(attach)在 AFL 上的运行机制, 这进一步说明了 QSYM 实现的符号执行技术对模糊测试效能提升的有效性。

对于输入数据长度分别为 4096、10240 的目标程序, 无论其是否包含符号执行妨碍特征, QSYM 对比 AFL 的漏洞发现效率并未出现显著提升, 表现出 QSYM 本身无法区分漏洞相关/无关字节的现象, 但根据  $execs_{TH} / execs_{orig}$  推断 QSYM 实现了污点分析技术, 且实际上 QSYM 也确实利用 libdft<sup>[32]</sup>实现了对污点变量的追踪, 本应具备区分漏洞相关/无关字节的能力。为了确认具体原因, 我们分析了 QSYM 生成的输入数据, 发现这些数据本身能够触发目标程序的新路径甚至直接触发漏洞, 说明 QSYM 在构造输入数据时能够区分漏洞相关/无关字节, 把约束求解得到的值插入到漏洞相关字节所在的位置。但是 QSYM 需附属在 AFL 上运行, 需要利用 AFL 的同步机制把自身产生的输入数据同步到 AFL 的种子队列中, 因此我们推断影响漏洞发现效率的原因是 AFL 本身的问题。AFL 的种子同步相关代码位于 afl-fuzz.c 的 sync\_fuzzers 函数中, 它调用 add\_to\_queue 函数将同步过来的输入数据放到种子队列的队尾, 如图 9(a)所示。由于 AFL 是顺序地从队列中取种子进行变异, 因此被同步到队尾的 QSYM 生成的输入数据会在最后才被取出用于模糊测试。输入数据包含的字节越多, AFL 对单个种子进行一轮变异的耗时越长, 导致位于队尾的 QSYM 产生的输入数据迟迟无法用于模糊测试, 从而无法有效提升模糊测试效率。

综上, 我们可知 QSYM 实现的符号执行技术能够切实有效地提升漏洞发现效率, 但仍受现有符号执行技术本身局限性的制约。此外, QSYM 利用 AFL 的种子同步机制将其构造的输入数据用于模糊测试, 但 AFL 本身对该同步机制的实现使得 QSYM 本身产生的输入数据无法在第一时间得以利用, 这限制了符号执行技术所带来的模糊测试效率提升效果, 这一缺陷会随着输入数据长度的提升而愈发显现。

### 5.2.2 Angora

根据 5.1 节的结论, 我们已知 Angora 实现了代码覆盖反馈指导策略、污点分析技术, 能够区分漏洞相关/无关字节, 但易受噪声路径的影响。此外, Angora 对符号执行妨碍特征不敏感的原因尚未确定, 既可能是未实现符号执行技术, 也可能是其技术有能力处理符号执行妨碍特征。为了进一步验证 Angora 技术的有效性并分析其优缺点背后的具体原因, 为其实践使用与改进提供更多参考, 我们进行如下分析。

首先, 为了明确 Angora 是否实现了符号执行技术, 我们额外构造了目标程序 IS4\_TS4\_TV1\_\_4-alter, 修改了原始目标程序的条件约束, 向魔数检查中加入符号执行技术能够求解的简单的算术运算, 将“if (VAR1 == 0x7E104EC3)”更改为“if (VAR1+cc == 0x7E103EC3)”, 其中 cc 的值是从文件中读取的常量, 以此防止算术运算被编译器优化, 若工具实现了符号执行技术, 则应该能够计算出满足条件约束的 VAR1 的取值, 进而构造输入数据, 快速触发条件约束下的漏洞函数。我们对比了 Angora、AFL 和 QSYM 对 IS4\_TS4\_TV1\_\_4-alter 的模糊测试结果, 发现 Angora 和 AFL 超时未触发漏洞, 而 QSYM 仅用 24 s 即触发漏洞, 这说明 Angora 对符号执行妨碍特征不敏感的原因同 AFL 一样, 是由于其并未实现符号执行技术。

图 7(b)展示了污点分析妨碍特征对 Angora 模糊测试效能的影响, 对于 IS16\_TS8\_TV8\_\_1, Angora 能够快速通过 8 层嵌套的条件分支语句, 而对于添加了污点分析妨碍特征的 IS16\_TS8\_TV8\_\_1-idf, Angora 的漏洞触发效率甚至低于 AFL, 在 2 h 仅通过了 3 层条件分支语句。此外, 我们发现 Angora 单位时间内对目标程序的执行次数要少于 AFL, 例如, AFL 平均每秒能够对 IS16\_TS8\_TV8\_\_1 执行 1345.16 次, 而 Angora 平均每秒仅对其执行 801.82 次。

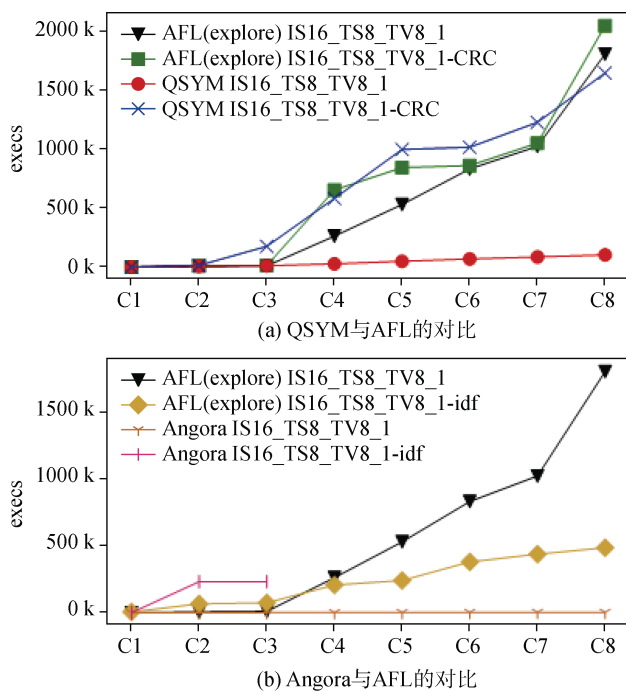


图 7 条件约束首次被满足平均花费的运行次数对比  
Figure 7 Comparison of the average number of executions when constraints are satisfied for the first time

综上, 我们可以确认 Angora 并未实现符号执行技术, 而实现的污点分析技术能够切实有效地提升漏洞发现效率, 但仍受现有污点分析技术本身局限性的制约。此外, 在污点分析技术失效的情况下, Angora 自身的种子变异算法运行效率并不如 AFL。

### 5.2.3 AFL、AFLFast、MOPT 与 TortoiseFuzz

根据 5.1 节的结论, AFL、AFLFast、MOPT-AFL 和 TortoiseFuzz 的能力推断相同, 但它们的测评结果数据仍存在一定差异。本节以 AFL 为基准(包括 AFL 的 exploit 模式和 explore 模式), 通过对比目标程序的代码触发时空分布等数据来确定这些模糊测试工具的不同之处, 分析它们的优缺点, 为实践使用与改进提供更多参考。

**漏洞触发效果对比:** 首先我们观察这些工具对漏洞相关变量个数为 1 的目标程序(命名中包含“TV1”, 以下简称为“TV1 程序”)的模糊测试结果。根据 3.2 节所述, 漏洞相关变量与漏洞路径上的分支语句一一对应、数量相同, 因此 TV1 程序的漏洞路径上仅包含 1 个分支语句, 在此情况下路径覆盖指导策略对漏洞发现效率的影响有限, 种子变异算法起主要作用。以  $N/N_V$  为  $2^{12}$  的 TV1 程序为例, AFL、AFLFast、MOPT-AFL 和 TortoiseFuzz 平均花费 74092、91924、34080 和 60252 次变异触发漏洞, 可知 MOPT-AFL 效率最佳, 对比 AFL 有较为明显的提升, 此外, 根据表 5, MOPT-AFL 的  $\Delta execs / \Delta(N/N_V)$  也是四者中最小, 因此可以推断: MOPT-AFL 对种子变异算法进行了优化(推断 K)。

对于漏洞路径包含多条分支语句的目标程序, 我们对比了其漏洞路径上的条件约束首次被满足时平均花费的运行次数。图 8 展示了被测工具对目标程序 IS16\_TS4\_TV4\_\_1 的路径触发效率, 可看出 AFLFast 的效率在其中为最低, 表 6 也说明 AFLFast 即使是在无噪声路径干扰的情况下, 对漏洞路径上的基本块的触发次数占比也是这些工具中最底的。由于本文构造的目标程序把漏洞放置于条件分支语句嵌套的最内层, 因而我们可以推断: AFLFast 在代码覆盖反馈指导策略上不倾向于对这种“深路径”的探索, 不利于发现处于路径深处的漏洞(推断 L)。

**种子变异调度策略对比:** 一个 cycle 表示模糊测试对种子队列中所有的种子都执行了一轮变异, 种子队列里包含的种子数量越多, 或者对单个种子执行的变异次数越多, 则每个 cycle 执行的变异次数越多。根据表 7 可知, 对于目标程序 IS8\_TS4\_TV4\_\_1, AFL 在 exploit 模式下的每个 cycle 包含 87406 次目标程序执行, 而 explore 模式的每个 cycle 仅包含

5271 次执行, 这意味着 exploit 模式专注于对单个种子进行大量变异, 而 explore 模式倾向于对整个种子队列进行快速轮询(完成了 109 个 cycle)。从上述角度来看, AFLFast、MOPT-AFL 和 TortoiseFuzz 对种子变异调度策略更接近于 AFL 的 explore 模式(推断 M)。

**噪声路径敏感性对比:** 我们根据测评中记录的代码触发时空分布数据, 计算漏洞路径上的基本块的触发次数占有基本块触发次数的比例, 如表 6 所示, 可知无论目标程序是否包含噪声路径, TortoiseFuzz 对漏洞路径上的基本块的触发次数占比均为最高, 且从漏洞触发效果上来看, TortoiseFuzz 对包含噪声路径的目标程序的漏洞触发成功率为四者中最高, 综上, TortoiseFuzz 受噪声路径的影响最小, 因此可以推断它对路径覆盖指导策略实现了某种优化, 降低了噪声路径对模糊测试触发漏洞的干扰作用(推断 N)。

实际上, AFLFast、MOPT 和 TortoiseFuzz 均通过对 AFL 的代码覆盖反馈指导策略或种子变异算法进行优化来提升模糊测试效能。AFLFast 修改了种子优先级与种子能量的计算算法, 使模糊测试对路径的探索以广度优先, 快速覆盖更多浅层路径(对应推断 L), MOPT 基于粒群算法优化了变异操作的调度(对应推断 K), TortoiseFuzz 针对能够触发敏感内存操作的种子, 提高其优先级和能量值, 使模糊测试导向可能包含漏洞的代码区域, 提升漏洞触发效率(对应推断 N)。

5.3 测评结论指导工具使用与改进

5.3.1 工具使用建议与改进思路

测评对模糊测试工具的使用或是改进均有着重要的参考价值与指导意义, 而基于妨碍特征的模糊测试工具测评能够得到更细粒度且更具可解释性的结论。我们根据 5.1、5.2 节得到的结论, 针对部分被测试工具提出以下使用建议和改进思路:

对于 QSYM, 除了现有符号执行技术本身的局限性之外, AFL 实现的种子同步机制无法第一时间应

用 QSYM 构造的输入数据, 当输入数据包含大量字节时, 无法有效提升模糊测试效率。针对此问题进行改进预期能收获效能提升。

对于 Angora, 我们发现当污点分析技术失效时, 其自身的种子变异算法运行效率不如 AFL, 因此在实践中我们建议使用 Angora 的 “--sync\_afl” 模式, 将 Angora 与 AFL 并行运行, 从而能够在污点分析失效时发挥 AFL 变异算法运行效率上的优势。

对于 AFLFast, 其种子变异调度倾向于快速轮询整个种子队列, 因此当需要快速挖掘目标程序浅层漏洞或初始种子数量较多时可以使用, 但不适用于挖掘路径深处的漏洞。在改进方面, 我们建议利用符号执行、污点分析等技术帮助其快速突破深路径的探索, 从而更好地发挥 AFLFast 的优势。

5.3.2 QSYM 改进

由 5.2.1 节的分析可知, QSYM 利用 AFL 的同步机制将自身产生的输入数据同步到 AFL 的种子队列尾部, 如图 9(a)所示, 导致 QSYM 生成的输入数据无法及时被用于模糊测试, 影响了符号执行技术所带来的模糊测试效能提升的效果, 这种影响会随着输入数据长度的提升而愈发显现。

针对上述问题, 我们修改了 AFL 的种子同步机制, 将 AFL 同步过来的输入数据插入到当前正在执行变异的种子之后, 如图 9(b)所示, 使 QSYM 产生的输入数据能在第一时间被用于模糊测试。

我们把 QSYM 附着在修改后的 AFL 上运行, 将其作为 altered QSYM 重新对 IS4K\_TS1\_TV1\_\_1 和 IS10K\_TS1\_TV1\_\_1 进行了测试, 结果如表 8 所示, 漏洞触发所需变异次数明显比修改前要少, 漏洞触发效率得到显著提升。为了进一步验证 altered QSYM 的效能表现, 我们用 LAVA-M 测试集对改进前后的 QSYM 进行了测评, 每个目标程序执行 2 h 模糊测试(其中 uniq 执行 5 h 模糊测试), 结果如图 10 所示, 可知 altered QSYM 能够更快地触发更多漏洞。虽然理论上当模糊测试运行足够长的时间后, QSYM 和 altered QSYM 发现的漏洞数量会趋于相

表 6 漏洞路径上的基本块的触发次数所占比例

Table 6 Percentage of executions of the basic blocks on vulnerability path (%)

	IS8_TS4_TV4__1	IS8_TS4_TV4__1-noise	IS16_TS4_TV4__1	IS16_TS4_TV4__1-noise
AFL(exploit)	11.548	0.232	13.305	0.343
AFL(explore)	13.136	0.239	12.082	0.190
AFLFast	3.443	0.272	2.210	0.323
MOPT-AFL	12.445	0.241	18.196	0.265
TortoiseFuzz	16.389	0.300	19.739	0.366



表 7 模糊测试完成的 cycle 数对比

Table 7 The number of cycles finished by fuzzers

	IS8_TS4_TV4__1	IS8_TS4_TV4__1-noise
AFL(exploit)	7.2 cycles (87406 execs/cycle)	1.35 cycles (3904329 execs/cycle)
AFL(explore)	109 cycles (5271 execs/cycle)	27.7 cycles (240075 execs/cycle)
AFLFast	25.5 cycles (26306 execs/cycle)	12.6 cycles (255767 execs/cycle)
MOPT-AFL	40.7 cycles (10545 execs/cycle)	26.3 cycles (309093 execs/cycle)
TortoiseFuzz	57 cycles (4951 execs/cycle)	22 cycles (218133 execs/cycle)

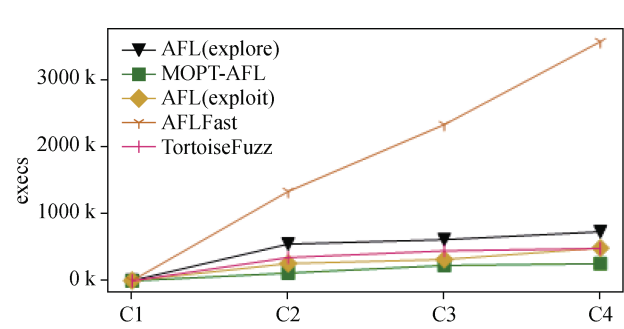


图 8 条件约束首次被满足平均花费的运行次数

Figure 8 Comparison of the average number of executions when constraints are satisfied for the first time

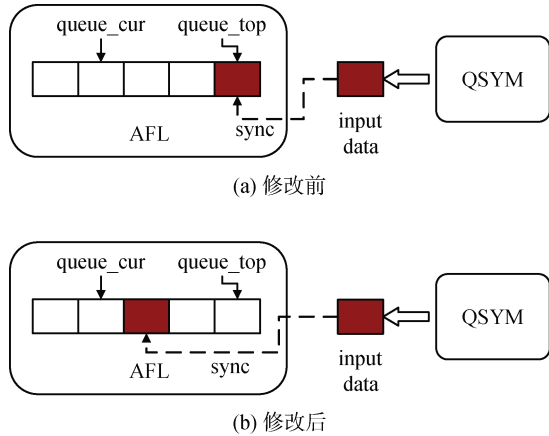


图 9 AFL 和 QSYM 之间的输入数据同步

Figure 9 Input data synchronization

(注: queue\_cur 指向当前正在进行变异的种子; queue\_top 始终指向队列的种子; sync 表示 AFL 对 QSYM 产生的输入数据进行同步)

等, 但后者能够尽早利用符号执行产生的输入数据, 使模糊测试更快地达到应有效果的“上限”, 在实践中可以据此更早地结束模糊测试, 节省时间和计算资源。

表 8 改进前后 QSYM 的漏洞触发效率对比

Table 8 Comparison of the evaluation results

	QSYM	altered QSYM
IS4K_TS1_TV1__1	489039 execs	110410 execs
IS10K_TS1_TV1__1	1408833 execs	262064 execs

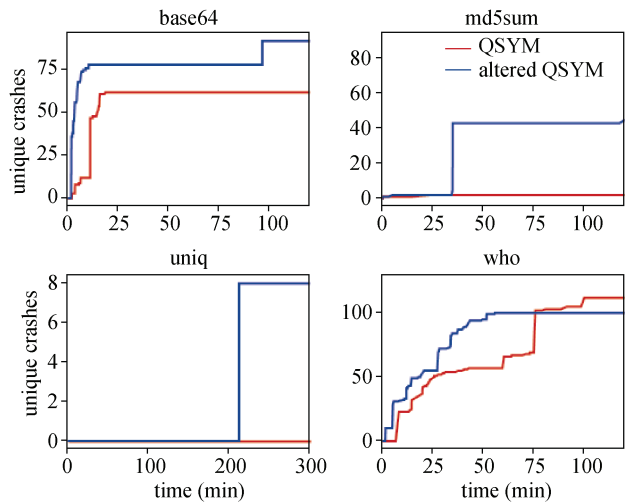


图 10 QSYM 修改前后对 LAVA-M 的漏洞触发情况

Figure 10 The bugs of LAVA-M triggered by QSYM and altered QSYM

## 6 总结

现有模糊测试工具测评普遍忽视了目标程序本身对测评结果的影响, 导致在一些情况下无法明确解释测评结果背后的原因, 从而无法得出明确的、具有良好可解释性的结论。本文针对上述问题, 构建了具有模糊测试妨碍特征标注信息的测试集, 将妨碍特征纳为测评的控制变量之一, 通过更细粒度的对比测试来建立测评结果与妨碍特征的关联关系推断被测工具的能力, 提升测评的可解释性。为此, 本文总结了 5 种妨碍特征, 并使用代码合成的方式基于这些特征构建了测试集 Bench4I, 对 6 款技术特点各不相同的模糊测试工具进行了测评, 通过对比分析推断工具的能力, 提升了测评的可解释性, 并基于测评结论对部分工具的实践使用与改进提供了建议, 并按照提出的思路实践了对 QSYM 的改进, 有效提升了其漏洞发现效率。

在未来工作中, 本文计划从模糊测试优化辅助技术对抗的角度, 针对更多模糊测试技术扩充妨碍特征的种类, 扩充 Bench4I 测试集, 使之能够在测评中支持对更多模糊测试技术的验证与能力推断。

## 参考文献

- [1] Manes V J M, Han H, Han C, et al. The Art, Science, and Engineering of Fuzzing: A Survey[EB/OL]. 2018: arXiv: 1812.00140. <https://arxiv.org/abs/1812.00140>.
- [2] American Fuzzy Lop. M. Zalewski. <https://lcamtuf.coredump.cx/afl>. Jun. 2020.
- [3] OSS-Fuzz. Google. <https://google.github.io/oss-fuzz>. Jun. 2020.
- [4] Cyber Grand Challenge. Defense Advanced Research Projects Agency (DARPA). <https://github.com/CyberGrandChallenge>. Jun. 2020.
- [5] Dolan-Gavitt B, Hulin P, Kirda E, et al. LAVA: large-scale automated vulnerability addition[C]. *2016 IEEE Symposium on Security and Privacy*, 2016: 110-121.
- [6] Google. <https://opensource.google/projects/fuzzer-test-suite>. Jun. 2020.
- [7] FuzzBench: Fuzzer Benchmarking As a Service. Google. <https://google.github.io/fuzzbench>. Jun. 2020.
- [8] Klees G, Ruef A, Cooper B, et al. Evaluating Fuzz Testing[C]. *The 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018: 2123-2138.
- [9] Li Y W, Ji S L, Chen Y, et al. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers[EB/OL]. 2020: arXiv: 2010.01785. <https://arxiv.org/abs/2010.01785>.
- [10] Peach Fuzzer. Peach Tech. <https://www.peach.tech/products/peach-fuzzer>. Jun. 2020.
- [11] Trinity: Linux system call fuzzer. D. Jones. <https://github.com/kernelslacker/trinity>. Jun. 2020.
- [12] A KVM/QEMU Based USB-fuzzing Framework. vusbf-Framework. <https://github.com/schumilo/vUSBf>. Jun. 2020.
- [13] A general-purpose fuzzer. Aki Helin. <https://gitlab.com/akihe/radamsa>. Jun. 2020.
- [14] ZZUF. S. Hocevar. <https://github.com/samhocevar/zzuf>. Jun. 2020.
- [15] Böhme M, Pham V T, Roychoudhury A. Coverage-Based Greybox Fuzzing as Markov Chain[J]. *IEEE Transactions on Software Engineering*, 2019, 45(5): 489-506.
- [16] TortoiseFuzz. <https://github.com/TortoiseFuzz/TortoiseFuzz>. Jul. 2020.
- [17] Gan S T, Zhang C, Qin X J, et al. CollAFL: path sensitive fuzzing[C]. *2018 IEEE Symposium on Security and Privacy*, 2018: 679-696.
- [18] Böhme M, Pham V T, Nguyen M D, et al. Directed Greybox Fuzzing[C]. *The 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017: 2329-2344.
- [19] C. Lyu, S. Ji, C. Zhang, et al. Optimized Mutation Scheduling for Fuzzers[C]. *28th USENIX Security Symposium*, 2019: 1949-1966.
- [20] Yun I, Lee S, Xu M, et al. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing[C]. *The 27th USENIX Conference on Security Symposium*, 2018: 745-761.
- [21] Stephens N, Grosen J, Salls C, et al. Driller: augmenting fuzzing through selective symbolic execution[C]. *Proceedings 2016 Network and Distributed System Security Symposium*, 2016: 1-16.
- [22] Gan S T, Zhang C, Chen P, et al. GREYONE: Data Flow Sensitive Fuzzing[C]. *The 29th USENIX Conference on Security Symposium*, 2020: 2577-2594.
- [23] Chen P, Chen H. Angora: efficient fuzzing by principled search[C]. *2018 IEEE Symposium on Security and Privacy*, 2018: 711-725.
- [24] S. Rawat, V. Jain, A. Kumar, et al. VUzzer: Application-aware Evolutionary Fuzzing[C]. *NDSS*, 2017: 1-14.
- [25] Jain V, Rawat S, Giuffrida C, et al. TIFF: using input type inference to improve fuzzing[C]. *The 34th Annual Computer Security Applications Conference*, 2018: 505-517.
- [26] Rajpal M, Blum W, Singh R. Not all Bytes are Equal: Neural Byte Sieve for Fuzzing[EB/OL]. 2017: arXiv: 1711.04596. <https://arxiv.org/abs/1711.04596>.
- [27] She D D, Pei K X, Epstein D, et al. NEUZZ: efficient fuzzing with neural program smoothing[C]. *2019 IEEE Symposium on Security and Privacy*, 2019: 803-817.
- [28] Zhu X G, Feng X T, Jiao T Y, et al. A Feature-Oriented Corpus for Understanding, Evaluating and Improving Fuzz Testing[C]. *The 2019 ACM Asia Conference on Computer and Communications Security*, 2019: 658-663.
- [29] Serebryany K, Bruening D, Potapenko A, et al. AddressSanitizer: A Fast Address Sanity Checker[C]. *The 2012 USENIX conference on Annual Technical Conference*, 2012: 28.
- [30] Jung J, Hu H, Solodukhin D, et al. FUZZIFICATION: Anti-Fuzzing Techniques[C]. *The 28th USENIX Conference on Security Symposium*, 2019: 1913-1930.
- [31] Güler E, Aschermann C, Abbasi A, et al. ANTIFUZZ: Impeding Fuzzing Audits of Binary Executables[C]. *The 28th USENIX Conference on Security Symposium*, 2019: 1931-1947.
- [32] V. P. Kemerlis, G. Portokalidis, et al. libdft: Practical dynamic data flow tracking for commodity systems[C]. *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012: 121-132.



郝高健 于 2013 年在重庆大学网络工程专业获得学士学位。现在中国科学院大学网络空间安全学院网络空间安全专业攻读博士学位。研究领域为软件安全分析理论与技术。研究兴趣包括漏洞发现工具测评等。Email: haogaojian@iie.ac.cn



李丰 于 2013 年在中国科学院大学获博士学位, 现为中国科学院信息工程研究所副研究员, 研究方向为程序分析与软件漏洞挖掘。Email: lifeng@iie.ac.cn



**霍玮** 于 2010 年在中国科学院计算技术研究所获博士学位。现任中国科学院信息工程研究所博士生导师, 中国科学院青年创新促进会成员。主要研究领域包括软件漏洞挖掘、利用和安全评测、基于大数据及知识图谱的软件安全分析、信息系统安全分析等。Email: huowei@iie.ac.cn



**邹维** 研究员、博士生导师。现任中国科学院信息工程研究所副所长。中国科学院优秀指导教师, 中国计算机学会优秀博士学位论文指导教师。研究领域包括网络与软件安全、网络空间安全评测。Email: zouwei@iie.ac.cn