计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2021.20201018
58(5):944-963,2021

模糊测试技术综述

任泽众[1] 郑 晗[1] 张嘉元[2] 王文杰[1] 冯 涛[2] 王 鹤[3] 张玉清[1,3,4]
[1](中国科学院大学国家计算机网络入侵防范中心 北京 101408)
[2](兰州理工大学计算机与通信学院 兰州 730050)
[3](西安电子科技大学网络与信息安全学院 西安 710071)
[4](海南大学计算机与网络空间安全学院 海口 570228)
(zhangyq@ucas.ac.cn)

A Review of Fuzzing Techniques

Ren Zezhong[1], Zheng Han[1], Zhang Jiayuan[2], Wang Wenjie[1], Feng Tao[2], Wang He[3], and Zhang Yuqing[1,3,4]
[1](*National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, Beijing* 101408)
[2](*School of Computer and Communication, Lanzhou University of Technology, Lanzhou* 730050)
[3](*School of Cyber Engineering, Xidian University, Xi'an* 710071)
[4](*School of Computer Science and Cyberspace Security, Hainan University, Haikou* 570228)

Abstract Fuzzing is a security testing technique, which is playing an increasingly important role, especially in detecting vulnerabilities. Fuzzing has experienced rapid development in recent years. A large number of new achievements have emerged, so it is necessary to summarize and analyze relevant achievements to follow fuzzing's research frontier. Based on 4 top security conferences (IEEE S&P, USENIX Security, CCS, NDSS) about network and system security, we summarized fuzzing's basic workflow, including preprocessing, input building, input selection, evaluation, and post-fuzzing. We discussed each link's tasks, challenges, and the corresponding research results. We emphatically analyzed the fuzzing testing method based on coverage guidance, represented by the American Fuzzy Lop tool and its improvements. Using fuzzing testing technology in different fields will face vastly different challenges. We summarized the unique requirements and corresponding solutions for fuzzing testing in specific areas by sorting and analyzing the related literature. Mostly, we focused on the Internet of Things and the kernel security field because of their rapid development and importance. In recent years, the progress of anti-fuzzing testing technology and machine learning technology has brought challenges and opportunities to the development of fuzzing testing technology. These opportunities and challenges provide direction reference for the further research.

Key words fuzzing; basic working process; IoT security; kernel security; machine learning

摘 要 模糊测试是一种安全测试技术,主要用于检测安全漏洞,近几年模糊测试技术经历了快速发展,因此有必要对相关成果进行总结和分析.通过搜集和分析网络与系统安全国际四大顶级安全会议

收稿日期:2020-12-14;**修回日期**:2021-03-04
基金项目:国家重点研发计划项目(2018YFB0804701);国家自然科学基金项目(U1836210,61762060);甘肃省科技厅重点研发计划项目(20YF3GA016)
This work was supported by the National Key Research and Development Program of China (2018YFB0804701), the National Natural Science Foundation of China (U1836210, 61762060), and the Key Research and Development Program of the Science and Technology Department of Gansu Province of China (20YF3GA016).

(IEEE S&P,USENIX Security,CCS, NDSS)中相关的文章,总结出模糊测试的基本工作流程,包括:预处理、输入数据构造、输入选择、评估、结果分析这5个环节,针对每个环节中面临的任务以及挑战,结合相应的研究成果进行分析和总结,其中重点分析以American Fuzzy Lop工具及其改进成果为代表的,基于覆盖率引导的模糊测试方法.模糊测试技术在不同领域中使用时,面对着巨大的差异性,通过对相应文献进行整理和分析,总结出特定领域中使用模糊测试的独特需求以及相应的解决方法,重点关注物联网领域,以及内核安全领域.近些年反模糊测试技术以及机器学习技术的进步,给模糊测技术的发展带来了挑战和机遇,这些机遇和挑战为下一步的研究提供了方向参考.

关键词 模糊测试;基本工作流程;物联网安全;内核安全;机器学习

中图法分类号 TP319

"常用系统中可能会潜伏着严重的漏洞."[1]这一论述源自于模糊测试首次面世的论文中,它揭示了一个事实,隐藏的漏洞无处不在.漏洞可以引发严重的危害,2011年披露的震网[2]蠕虫病毒,利用软件漏洞损坏了伊朗核设施的离心机.2017年发生的WannaCry勒索攻击[3],通过利用漏洞,恶意加密了超过150个国家的至少23万台计算机中的文件,并以此进行勒索.2019年波音737MAX客机控制系统的漏洞被发现[4],在此之前,已经造成了2起坠机事件,导致346人死亡,这些事件凸显着漏洞检测技术的重要性.

模糊测试是漏洞检测技术的一种,通过使用针对目标程序生成的随机字符流,对目标程序进行多次测试,以检测可能存在的漏洞[1].

目前模糊测试已经成为漏洞检测的一种重要方法,在2019年Google Project Zero[5]的报告中,通过模糊测试发现的漏洞数目,比例高达37%.在安全界中,对于模糊测试技术的研究工作自始至终从未停止.

模糊测试相关的研究工作,一方面集中在提升模糊测试的漏洞检测能力,另一方面集中在拓宽模糊测试的应用范围.为了提升模糊测试的检测能力,业界引入了诸如符号执行、污点分析等程序分析技术,提升了模糊测试在覆盖率等评估指标上的表现,进而获得了更好的漏洞检测效果.Zhang等人[6]以及Wang等人[7]对该方向的定向检测部分进行了很好的总结.为了拓宽模糊测试的应用范围,模糊测试目前已被应用到了诸如物联网、内核安全等领域中,并为满足相应应用领域的特殊要求,诞生了不同的研究成果,具体将在第5节介绍.

目前,已有综述文章[8-12]对模糊测试做了归纳和整理,但自2018年起,模糊测试技术开始快速发展,有必要重新梳理和总结该领域的研究工作.

通过检索2010—2020年的网络与系统安全国际四大顶级会议(IEEE S&P,USENIX Security,CCS,NDSS),在图1中展示了模糊测试相关研究工作数量的逐年变化.可以看到,2018年以后的成果数量在急速地增加,因此基于这些文献,分析与归纳模糊测试的研究现状与方向有重要现实意义.

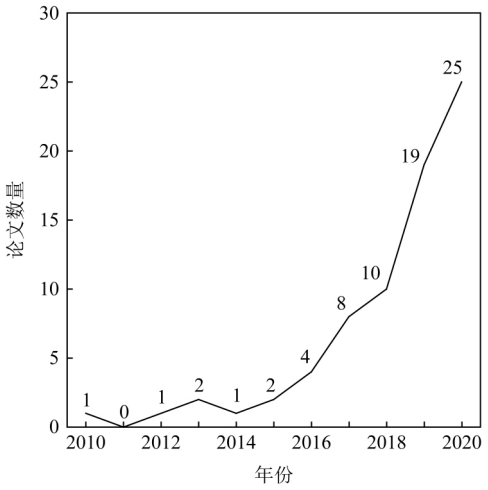

Fig. 1 The number of papers related to fuzzing published at famous security international conferences in the past ten years

图1 10年内在国际著名安全会议上发表的与fuzzing相关的论文数量

我们的主要贡献有4个方面:

1) 分析回顾了模糊测试的发展历史,选取模糊测试研究的重要时间节点.这些时间节点中即包含模糊测试发展史上开创性的工作,比如首次将模糊测试思想应用到不同领域中,也包含了诸多意义重大的研究成果,比如AFL(American fuzzy lop)[13]的诞生.

2) 目前一些综述文章提出了模糊测试的工作流程,但是并没有以此作为文章的脉络结构,读者

无法对模糊测试有一个整体工作流程的理解.对此,我们按照自己总结出的模糊测试基本工作流程,依次介绍每个环节的目的以及面临的挑战,并结合相应的研究成果介绍与讨论,从而使读者可以对模糊测试的整体工作流程有充分了解.

3) 模糊测试在不同的应用场景下,有着自己独特的使用需求.本文对于近年来不同领域的模糊测试研究进行了分析与整理,比如模糊测试在物联网、内核安全等领域的应用,基本上囊括了可以使用模糊测试技术的各个领域,为各个领域的安全从业人员使用模糊测试提供了参考.

4) 近年来随着反模糊测试、机器学习等相关领域的快速发展,给模糊测试技术领域带来了新的挑战和机遇.我们介绍了反模糊测试的思路,指出应该加强模糊测试平台建设,分析了机器学习同模糊测试结合的困难,并指明了模糊测试的下一步研究方向.

1 模糊测试

1.1 模糊测试的发展历程

为了给读者一个直观的印象,我们首先在图 2 中展示了模糊测试的发展历程,图 2 中的所有时间节点都会在本文详细介绍.如图 2 中所示,模糊测试诞生于 1988 年[1],并在那时设计了一个被称为 Fuzz 的工具,通过生成随机连续字符串对以字符串为输入的对象进行模糊测试,同时设计了一个名为 ptyjig 的工具,对输入有特殊要求的目标进行模糊测试.这时的模糊测试,其主要目的是尝试使用非常规的数据对目标的鲁棒性进行检测.

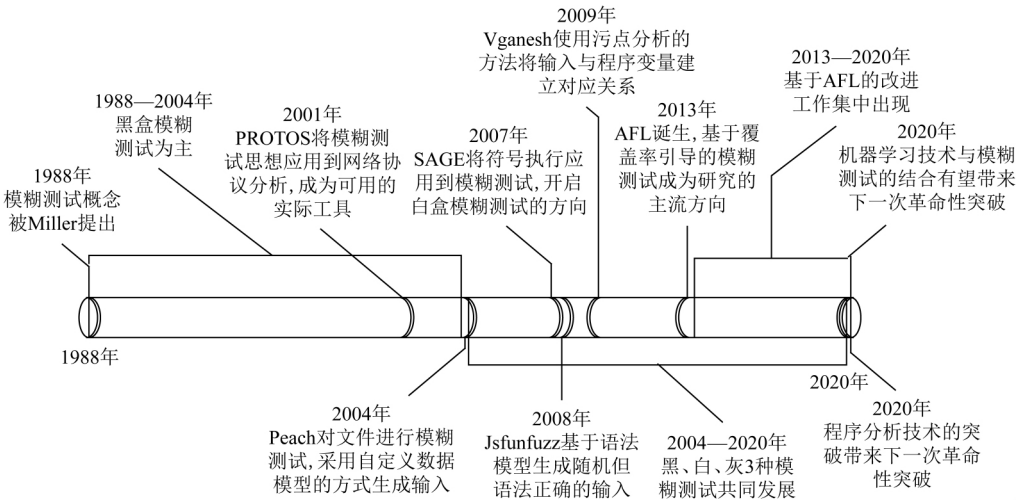

Fig. 2 The development of fuzzing

图 2 模糊测试的发展历程

在 1988—2004 年之间诞生的模糊测试方法大多是黑盒模糊测试,其中比较重要的有 Protos[14] 以及 Peach[15].Protos[14] 项目诞生于 2001 年,它将模糊测试首次应用到了网络协议的测试中,也是模糊测试技术成为实用性工具的开始.2004 年出现的 Peach[15] 是初期被应用在文件模糊测试的项目,之后历经改进,时至 2021 年仍然在被使用.此外,Peach[15] 可以由用户手工定义用于生成输入数据的数据模型,是基于语法生成输入思想的早期应用.

在 2007 年,受益于动态符号执行和测试数据生成技术的进步,诞生了 Sage[16] 模糊测试方法,该方法是使用符号执行的白盒模糊测试方法.通过使用符号执行技术,在程序运行过程中收集条件语句对输入的约束,通过用约束求解器进行求解产生新的输入.白盒模糊测试由于能够对目标内部情况有足够的了解,可以获得高质量的输入数据,绕过目标的输入检查,获得较高的覆盖率和深层漏洞检测效果.

2008 年,为了对输入数据高度结构化的目标进行模糊检测,诞生了 jsfunfuzz[17].该方法根据语法模型生成随机的但语法正确的 JavaScript 代码,这种基于语法生成输入的思想,成为之后对输入数据高度结构化目标进行模糊测试的重要指导思想.

在使用符号执行的白盒模糊测试中,通常会受限于符号执行自身的问题,比如资源消耗过多以及路径爆炸等,导致模糊测试效率不高.白盒模糊测试因为需要充分了解目标内部信息,因此会消耗大量

的资源,比如消耗时间去分析程序内部细节,而且对于复杂的程序,得到详细而全面的内部信息是不现实的.为了提高模糊测试效率,诞生了继续改进白盒模糊测试和使用少量的目标内部信息进行模糊测试的2个方向.

沿着继续改进白盒模糊测试的方向,在2009年出现了Vganesh[18].通过使用污点分析技术替代了开销巨大的符号执行技术.污点分析技术,可以将程序攻击点中使用到的变量值,同输入数据特定部分建立联系,进而指导模糊测试的变异策略.该思想也被灰盒模糊测试借鉴使用.

使用少量目标内部信息进行模糊测试的思想带来了灰盒模糊测试,该方向最重要的成果是在2013年出现的AFL[13]模糊测试工具.AFL[13]是一款以覆盖率为导向的模糊测试工具,通过插桩的方法,采集输入数据对应的边覆盖率,作为模糊测试种子选取的衡量指标.AFL[13]通过使用进化算法以及精心构造的突变策略,获得了很好的模糊测试效果,发现了大量漏洞,也因此被研究人员所关注,在2013年至今诞生了大量衍生性工作,这些工作会在2.2.4节有详细介绍.

我们列举了模糊测试发展过程中比较重要的几个时间点,期间诞生了开创性的或者影响深远的工作,图2中还预测了未来模糊测试可能取得突破的方向,依据是:程序分析技术是与模糊测试结合最为紧密的领域,因此下一个对模糊测试产生重大影响的时间节点很可能来自两者的结合领域.另外,机器学习的强大威力在图像和自然语言处理等领域大放异彩,机器学习技术与模糊测试的结合很可能在未来给模糊测试带来性能上的巨大提升.

1.2 模糊测试基本工作流程

模糊测试的基本工作流程可以用来描述一个完整的模糊测试工作过程.如图3所示,该流程可以划分为5步:预处理(preprocessing)、输入构造(input building)、输入选择(input selection)、评估(evaluation)、结果分析(post-fuzzing).其中第1和第5个环节属于模糊测试开始前的准备工作和模糊测试结束后的收尾工作,实际测试环节是第2~4个环节.

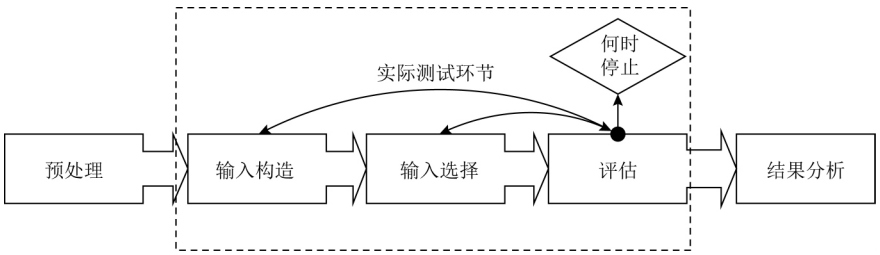

Fig. 3 Fuzzing basic working process

图3 模糊测试基本工作流程

按照工作流程,首先进入预处理环节,该环节的工作是:搜集目标相关信息并制定模糊测试的策略,为监控目标在测试中的运行状态做必要的准备.通常依赖于插桩、符号执行以及污点分析这类程序分析技术.

完成预处理后,在测试环节的工作包括:输入构造、输入选择以及评估3个子环节.

其中,输入构造的工作是:首先获取一定数量的种子,随后确定种子的能量分配策略、种子的优先级以及种子的突变策略,最后依据这些信息获得大量的输入数据.

得到大量输入数据后,可以考虑通过输入选择环节,对数据进行筛选.其主要工作是:尝试通过提前过滤掉无效的输入数据,以节省模糊测试的时间.从工作的目的来看,该环节适合使用机器学习技术,通过完成模式识别任务,在输入被执行前就提前判断数据的有效性.

作为测试环节的最后一步,评估部分的主要工作是:设计合适的实验,依照评估指标对模糊测试进行评估.合适的评估指标,可以给模糊测试带去执行结果的真实反馈,这是制定模糊测试执行策略的重要依据.同时,设计合适的实验更能反映模糊测试方法的实际性能,增加可信度.一个合适的模糊测试实验通常需要考虑:选择合适的模糊测试方法用作对比、选择合适的对象作为模糊测试的目标、选择合适的重复测试次数以及测试使用的超时时间等.

模糊测试结束以后,由于获得的结果无法被直接使用,还需要在结果分析环节进行处理.该环节的工作是:对获得的测试结果进行去重、复现、分析以及威胁性评估等工作,最终确定是否发现了有价值的漏洞.

2 预处理

预处理环节的目的是搜集目标相关信息,比如目标的输入数据格式、目标的内部结构,并为监控测试中目标的状态变化做必要的准备.该环节面临的挑战是:使用什么程序分析技术,以及模糊测试究竟需要对目标内部信息有多详细的了解.

接下来会先介绍预处理阶段经常使用的程序分析技术以及其优缺点.之后会根据预处理阶段对程序内部信息了解程度的不同,对模糊测试的类型进行划分,并结合相应研究成果,对每一类模糊测试做介绍.

2.1 分析技术与获取的信息

常用的程序分析技术包括:插桩、符号执行以及污点分析.

1) 插桩

插桩技术通过向目标的代码中合适的位置添加预设好的代码,获得程序的静态或动态执行信息.比如程序的抽象语法树,覆盖率以及函数内变量取值等.

常用的插桩技术分为动态插桩和静态插桩 2 种.静态插桩通常在源码或者中间代码的编译过程中进行,常见的比如通过 GCC 编译器在汇编语言上插桩,LLVM 在生成的中间语言 LLVM IR(low level virtual machine intermediate representation)上插桩.优点是节省时间、速度快,缺点是依赖于程序源码.动态插桩则是在运行的过程中对运行过的代码进行插桩.比如利用 QEMU[19] 等模拟技术,进行动态插桩,可以获得程序运行时的信息,缺点是资源的开销大.插桩技术已经被应用到模糊测试中,比如 AFL[13] 中就有静态插桩和动态插桩的 2 种模式,使得 AFL[13] 可以在源码以及二进制代码上进行模糊测试.

插桩技术的缺点是会带来资源的开销,比如在 UnTracer[20] 的论文中提到,相比于黑盒测试而言,以 AFL 为例的灰盒测试引入了将近 1300%的开销.

2) 符号执行

符号执行将程序行为的推理归结为逻辑领域的推理[21],通过构建一个表示程序执行的逻辑公式,可以同时推断一个程序在不同输入上的行为.该方法可以使模糊测试获得较好的覆盖率,并可以深入到程序深处,探寻可能存在漏洞的区域.

符号执行可以分为静态符号执行以及动态符号执行.静态符号执行通常会因为程序中循环和递归的存在,陷入到路径爆炸中,还会因为路径约束中包含诸如取 Hash 值等操作,导致约束求解失败.由于存在这 2 种问题,使用较多的是动态符号执行,动态符号执行通过对程序进行实际执行与符号化执行,维护程序的实际状态和符号化状态,通过将难以求解的约束替换为实际值,缓解了静态符号执行的问题,并按照深度优先的搜索策略对目标程序进行了探索.

动态符号执行也有自己的问题.①由于程序分支的存在,路径爆炸的问题仍然存在,程序越复杂,路径爆炸的问题就越严重.解决的一种办法是通过启发式的方法,选择比较重要的路径进行探索.②虽然动态符号执行使用实际值替换的方法,解决了一部分静态符号执行无法绕过的约束,但是也会丢失一些路径,造成探索结果的不完整.③所有的符号执行技术都受限于约束求解方法的能力,比如如何处理类似取余操作这样的非线性约束,仍然是符号执行面临的挑战.

为了能够将符号执行更好地应用到模糊测试中,近年来诞生了一些工作,比如 Pangolin[22] 通过允许符号执行重用之前的计算结果,Intriguer[23] 通过利用字段级的信息,都实现了对符号执行过程的加速.ILF[24] 通过使用神经网络,对由符号执行专家生成的大量高质量输入数据进行学习,得到了合适的模糊测试策略.

3) 污点分析

该技术会观测程序中,哪些程序数据受到了预先准备好的污染源(比如输入)的污染,目的是跟踪污染源和汇聚点(比如有敏感信息的程序数据)之间的信息流[21].

污点分析存在静态污点分析和动态污点分析 2 种.静态污点分析不需要程序实际运行,通过对程序静态分析,获得程序控制流图、抽象语法树等信息,依据数据流以及依赖关系进行污点分析;动态污点分析则是在程序实际执行的过程中,利用程序的动态执行信息进行污点分析.两者相比较而言,动态污点分析检测的可信度更高,但是检测结果是否全面,取决于动态污点分析对程序的覆盖情况,而且动态污点分析会消耗更多的资源;静态污点分析又会和符号执行一样,可能会陷入到路径爆炸中,而简化后的静态污点分析又存在着严重的过度污染问题.

将污点分析技术应用到模糊测试中,并降低其资源消耗是近年来的重要研究方向.比如 GREYONE[25] 尝试通过减少污点分析跟踪的对象、降低污点分析的开销、提升模糊测试的检测效率.

程序分析技术还有很多,本文只介绍了模糊测试中常用的3种.另外在黑盒测试中,只需要目标对象输入数据的格式等不涉及目标内部结构的信息.在一些特殊的应用场景下,预处理需要添加相应准备工作.比如在物联网设备上进行模糊测试会涉及到虚拟化等操作,相关内容会在第5节具体介绍.

2.2 模糊测试在预处理阶段的类型划分

根据模糊测试对程序内部信息分析的程度,现代的模糊测试方法可以划分为3类:黑盒模糊测试(black-box fuzzing)、灰盒模糊测试(grey-box fuzzing)和白盒模糊测试(white-box fuzzing).3种类型的模糊测试,没有优劣之分,只是有着各自的特点.本节对这3种模糊测试的特点以及相关的研究成果做介绍.

2.2.1 黑盒模糊测试

黑盒测试也被称为输入输出驱动(IO-driven)的测试,或者是数据驱动(data-driven)的测试.黑盒模糊测试不能对目标内部状态以及结构进行分析,只能获得诸如目标的输入数据格式等内部无关信息.此外在测试过程中,黑盒模糊测试无法跟踪目标内部的执行状态,只能通过检测目标的输出数据,对目标的状态进行判断.

对于黑盒模糊测试的研究是有意义的.首先,不是所有的被检测目标都是开源的,其次黑盒测试工具设计简单,开发和检测速度快[26].

黑盒模糊测试的问题是,由于不了解检测目标的内部信息,会生成大量无效输入,导致测试的覆盖率相对偏低,检测深层漏洞的能力有限.

黑盒模糊测试比较适用于输入数据高度结构化的目标以及复杂且难以分析的目标.黑盒模糊测试从最早的PROTOS[14],Peach[15],发展到如今的DELTA[27],IFuzzer[28],IMF[29],IoT Fuzzer[30],在网络、文件、内核以及物联网模糊测试上都有着重要的应用.

2.2.2 白盒模糊测试

白盒模糊测试会获得充足的目标内部信息,通常采用符号执行的方法.好处是可以生成高质量输入数据,在覆盖率以及程序的深层漏洞检测上有更好的表现.

但是实际应用的过程中,检测的目标比较复杂,符号执行容易陷入路径爆炸的问题中.另外,对目标程序细致全面的分析会消耗大量资源,严重影响模糊测试的效率.

为了尽量在不影响模糊测试效率的情况下,获得详细的目标内部信息,研究人员进行了很多改进工作,比如Driller[31]以及基于Driller[31]而诞生的QSYM[32],DigFuzz[33],Driller[31]中使用AFL[13]检索程序浅层的漏洞,当AFL[13]随机生成的输入无法继续深入探查程序的时候,任务转交给符号执行.QSYM[32]通过使用动态二进制转换,将符号执行与本地执行紧密的集成在一起,实现了更细粒度的指令级符号执行,解决了形成路径约束缓慢,甚至无法形成正确的约束的问题.如何正确快速的判断哪些程序的路径应该使用符号执行技术,是一个混合模糊测试器的优化问题.基于这个思想,DigFuzz[33]量化了处理每条路径的困难程度,将难以通过随机种子到达的路径转交给符号执行,较好的优化了在模糊测试中使用符号执行的策略.

近年来有一些研究尝试加速符号执行的速度,或者尽量将模糊测试和符号执行分离成2个过程.

为了加快符号执行,Pangolin[22]提出允许符号执行重用之前的计算结果,Intriguer[23]则通过利用字段级的信息,对符号执行在模糊测试中的使用作出了优化.

为了尽量避免因使用符号执行而影响模糊测试的效率,He等人[24]提出在模仿学习的框架之下,通过对学习任务的描述,从符号执行中学习一种高效、快速的模糊测试方法以及策略.在T-Fuzz[34]中通过设计算法,识别出程序中阻碍模糊测试继续进行的非关键校验和关键校验.通过二进制重写,覆盖掉非关键校验,最大限度地解除程序校验对模糊测试的限制,这是对程序而不是输入进行变异.符号执行在T-Fuzz[34]中只作为最后的验证方法以过滤误报.

2.2.3 灰盒模糊测试

灰盒模糊测试是白盒模糊测试的一种变体,它只能获得部分程序内部信息,用于模糊测试.其蕴含的思想是:对程序内部进行细致而全面的分析,并不是获得良好测试结果的必要条件,仅依靠有限的与测试目标相关的信息,再配合良好的测试策略,仍然可以获得令人满意的测试结果.

灰盒模糊测试并不是一个妥协后的方案,比如Matryoshka[35]通过使用良好的策略,解决了深度嵌套条件语句带来的路径约束问题,其效果要好于采用符号执行的白盒模糊测试.

灰盒模糊测试中,最重要的研究成果是AFL[13]模糊测试工具.它通过在编译时插桩,搜集模糊测试中边缘覆盖率信息,并使用了进化算法,将边覆盖率作为算法的适应函数(fitness function),使得模糊

测试可以沿着覆盖率增大的方向进行,极大地改善了模糊测试的效果.围绕着 AFL[13] 诞生了一系列的改进工作,2.2.4 节会对其中一些有代表性的成果做介绍.

总的来说,黑盒模糊测试工具是轻量级的测试工具,设计简单,测试速度快,但是检测效果并不理想,适用于难以对内部进行检测的对象,以及对于开发和检测时间有较高限制的情况下.白盒模糊测试工具是重量级的测试工具,更加智能,检测效果更好,对于深层的漏洞有更好的检测效果,但是无论是设计实现还是执行测试更加复杂与耗时,适用于可以分析内部信息的检测对象,以及对于深层漏洞有较高检测需求的任务.灰盒模糊测试并没有明确的定位,当对于目标内部信息分析较多的时候,灰盒模糊测试就更加偏向于白盒,反之则会偏向于黑盒.相比于白、黑盒模糊测试,灰盒模糊测试总体上来看会更有优势,通过灵活的设计,灰盒模糊测试可以在检测能力与资源消耗之间寻找一个合适的平衡点,获得最佳的检测效果.

2.2.4 AFL 及其改进工作

AFL[13] 虽然有一定效果,但是其设计上仍有缺陷,比如其能量分配(power schedule)的策略,以及种子选择的策略,都会影响 AFL[13] 的检测效率.而且 AFL[13] 更偏向于黑盒一些,对程序内部分析有限,这也限制了其检测深层漏洞的能力.所以通过对程序内部情况进一步分析,获得更好的检测效果也是 AFL[13] 改进的一个方向.针对 AFL 存在的不足,在 AFL[13] 诞生后,大量改进工作紧随其后.

在这些工作中 TriforceAFL[36],kAFL[37],PeriScope(fuzz)[38] 都是 AFL[13] 在内核模糊测试的使用,会在 5.2 节介绍.

在整体建模方面,AFLFast[39] 与 EcoFuzz[40] 都尝试通过对模糊测试进行更准确地建模以完成对模糊测试的能量分配.AFLFast[39] 将整个模糊测试过程建模为一个马尔可夫链(Markov Chain, MC),以此尝试对低频路径分配更多的能量.EcoFuzz[40] 进一步改进了建模,使用多臂老虎机(muti-armed bandit)模型,模糊测试需要依据现有信息从尝试不同策略与选择当前最优策略之间进行选择,以达到更合理的能量分配.

在符号执行方面,Driller[31] 将符号执行和模糊测试结合起来,设计合适策略使两者交替运行.QSYM[32] 使用动态二进制转换将符号执行与本机执行更好结合,以实现更细粒度、更快的指令级符号仿真.

符号执行对于模糊测试的效率影响较大.为了在不使用符号执行的情况下,解决路径约束问题,研究人员开始考虑使用污点分析,VUzzer[41] 在模糊测试过程中使用静态动态结合的方法,通过污点分析,指导种子变异.Angora[42] 则更进一步,基于大部分路径约束只由输入的少量字节决定这一思想提出了可伸缩的字节级的污点分析.并且通过与 AFL[13] 进行实验对比,得出使用上下文敏感的分支覆盖率统计更加有效的结论.此外 Angora[42] 还借鉴了机器学习中的梯度下降,进行导向性的突变.文中实验证明,Angora[42] 在保证效率的前提下有效地解决了路径约束问题.REDQUEEN[43] 通过使用观察到的输入与状态(input-to-state)一致性现象进一步优化了污点分析,降低了开销.GREYONE[25] 对 REDQUEEN[43] 进行了改进,通过使用分支的关键变量监视器(variable monitor)来避免复杂的静态分析.

在处理输入高度结构化的目标方向,Skyfire[44] 利用已有的样本知识,生成具有良好分布的种子输入.黑盒状态下,如何使用生成算法产生高质量数据是一个挑战,GLADE[45] 在这一方面作出了尝试.

在 2017 年诞生的 AFLGo[46] 中提出了定向模糊测试的概念,以帮助发现特定的漏洞.AFLGo[46] 使用模拟退火算法,有效地引导模糊测试到目标程序特定的位置进行检测.Hawkeye[47] 在 AFLGo[46] 的基础上,优化了路径的计算,同时引入了路径相似度概念以及自适应的种子变异策略,并优化了种子优先级调度.在实际测试的过程中实现了超过 AFL[13] 以及 AFLGo[46] 的测试效果.

在提升程序覆盖率上,CollAFL[48] 对于 AFL[13] 粗略的覆盖率统计方法做出了修改,通过利用程序控制流图以及精心构造的 Hash 算法,在统计覆盖率的时候有效地解决了路径碰撞问题,可以为模糊测试提供精确完整的路径覆盖信息,从而提高了发现新路径和漏洞的速度.FareFuzz[49] 认为基于 AFL[13] 一系列的各种模糊器仍然没有覆盖到足够广泛的程序状态,通过自动识别检测密度低的路径,以及一种新的动态生成的突变掩码创建算法,提高了模糊测试的覆盖率.

在实际使用的时候,并没有一个合适的标准规定某个具体情况下应该使用什么样的模糊测试工具,而是需要根据实际的情况具体分析.比如当模糊测试的目标比较简单且容易分析时,可使用结合了

符号执行的测试工具,如果目标有些复杂,就可以使用结合了污点分析的测试工具,如果目标非常复杂,这时候可以直接采用 AFL[13] 进行模糊测试.

AFL[13] 及一些代表性研究工作如表 1 所示,表 1 中总结了每个方法的特点以及对应论文中使用的实验对象.

Table 1 Fuzzer Based on the AFL

表 1 基于 AFL 的模糊测试方法

模糊测试方法	特性	实验对象
TriforceAFL[36]	基于 AFL 的 QEMU 模式,使用系统模拟技术实现对于分支信息的追踪.	Linux 内核
kAFL[37]	使用 Intel 的处理器跟踪(PT)技术以及 Inter 的硬件虚拟化特性(Inter VT-x),实现高效和独立于特定操作系统的模糊测试.	Linux 内核 Mac 内核 Windows 内核
PeriScope(Fuzz)[38]	研究利用外围设备以及驱动,间接的攻击内核程序的情况,既可以被动检测也可主动模拟攻击.	Andriod WiFi 驱动
AFLFast[39]	将整个模糊测试过程建模为一个马尔可夫链,并使用新的能量分配策略,解决能量分配不平衡问题.	GNU binutils
Driller[31]	使用 AFL 来发现浅层漏洞,当遇到 AFL 无法通过的程序检查后,使用符号执行技术生成输入来通过检查,使得测试得以继续.	DARPA CGC
Skyfire[44]	以语料库和语法作为输入,通过学习得到上下文敏感的概率语法(PCSG),指导生成有效的种子数据.	XML XSL JavaScript
GLADE[45]	通过使用一组给定的输入样例和对应的程序输出,在黑盒的状况下,得到程序输入语言的上下文无关的语法.配合基于语法的模糊测试器对具有结构化输入的程序展开测试.	URLs GNU Grep LISP XML
VUzzer[41]	通过对程序的控制和数据流进行轻量级的动态和静态分析,为种子突变过程提供信息反馈,提升模糊测试工具生成输入的质量.	DARPA CGC binaries 多种应用程序 LAVA 生成的含有漏洞的二进制测试用例
AFLGo[46]	提出定向模糊测试,根据种子与目标的距离,对不同种子分配不同能量,进而引导种子快速抵达目标并触发目标处的漏洞	补丁测试 LibXML2 LibMing
Hawkeye[47]	改进 AFLGo 的目标距离计算算法、引入函数相似度概念,对能量调度算法进行优化并对种子的变异粒度以及优先级调度进行了改进.	GNU Binutils MJS Oniguruma Fuzzer Test Suite
QSYM[32]	使用动态二进制翻译将符号仿真与本机执行紧密结合到一起,实现更细粒度、更快的指令级符号仿真.	LAVA-M Dropbox Lepton Ffmpeg OpenJPEG, etc.
Angora[42]	提出了上下文敏感的分支覆盖、可扩展的字节级污点跟踪、基于梯度下降的搜索等改进措施,在不使用符号执行的情况下解决路径约束,提高效率.	LAVA-M File-5.32 Jhead-3.00 Xmlwf(expat)-2.2.5,等
CollAFL[48]	使用控制流图和精心设计的 Hash 冲突解决方案,解决了路径冲突问题,从而可以提供精确、完整的覆盖信息.	24 个流行的开源 Linux 应用
FairFuzz[49]	在模糊测试过程中自动识别执行频率低的分支,通过使用一种新的突变掩码创建算法,引导突变偏向于产生覆盖低执行分支的输入.	Libjpeg-turbo-1.5.1 Libpng-1.6.29 Tcpdump-4.9.0 GNU binutils-2.28 Mupdf-1.9,等
T-Fuzz[34]	通过将程序中非必要约束覆盖掉,尽量减少模糊测试无法绕过的约束.由于改变了程序,为了防止误判,当触发漏洞时要使用符号执行工具进行验证.	LAVA-M Pngfix Tiffinfo Magick pdfohtml

续表 1

模糊测试方法	特性	实验对象
REDQUEEN[43]	介绍了一个轻量级的,但非常有效的污点跟踪和符号执行的替代方案,通过观察输入和程序状态关系,指导种子的变异过程,绕过程序检查.	LAVA-M DARPA's CGC Binutils Lodepng library
EcoFuzz[40]	将整个模糊测试建模为一个多臂老虎机问题,分为探索-求解 2 个部分,进而指导能量分配来解决能量分配不平衡问题.	GNU Binutils SNMP Libjpeg-turbo-1.5.3 Libxml2-2.9.9 Gif2png-2.5.13,等
GREYONE[25]	在 REDQUEEN 的基础上通过数据流分析,减少复杂的静态分析并使污点分析更加轻量级	19 个流行的开源 Linux 应用 LAVA-M

3 实际测试环节

在经历了必要的准备工作后,开始进行实际测试,该部分由输入构造、输入选择、以及评估 3 部分构成.

3.1 输入构造

输入构造环节的目的是构造出可以被检测目标执行,用于模糊测试的输入数据.具体的挑战是如何在尽量满足语法语义检查的情况下,短时间内生成大量的输入,用以对目标做全面而深入分析.

为应对这样的挑战,目前获得输入数据的普遍方法是:首先得到一个数据 S,然后数据 S 按照一定的策略进行一定次数的变异,获得大量新数据 I,最后将 I 输入到被测试对象中进行测试.其中数据 S 被称为种子(seed),I 是测试实际使用的输入数据.这样做的好处是可以通过精心构造种子来保证输入的有效性,然后通过对种子进行合理的变异,获得大量的高质量输入数据.

为了获得足够多的高质量输入数据,需要经过种子获取、种子筛选、种子突变 3 个阶段.种子获取阶段将介绍种子的来源,种子筛选阶段会介绍种子的能量分配以及优先级分配策略,种子突变则会介绍相应的突变策略.

3.1.1 种子获取

种子可以直接使用提前准备好的高质量数据集,也可以通过模型生成得到.按照一定的策略从执行过的数据中进行选择,也是种子生成的常用手段.

模糊测试第 1 次使用的种子可以是事先准备好的高质量的输入数据,也可以是通过生成器生成得到.指导生成的策略可以是用户定义的,也可以是模糊测试器自行学习到的.比如为了给高度结构化的测试对象(比如 JavaScript)生成输入,Skyfire[44] 通过使用提前准备好的词典,学习得到一个概率性上下文敏感语法(probabilistic context-sensitive grammar),该语法包含有输入的语法和语义特征,被用做指导种子生成.

采用生成的方法获取种子,不一定总可以获得目标语言的语法作为前置条件.在黑盒条件下,GLADE[45] 通过使用少量输入样本,以及黑盒测试的输入输出信息,合成输入数据的上下文无关语法,自动生成有效的输入数据用于模糊测试.

在特定的使用环境下,模糊测试器的种子生成需要有特定的方法.比如针对 Android 本机系统服务中的漏洞,FANS[50] 通过收集目标服务中的所有接口,以及变量名称和类型,得到接口模型和依赖信息,依次生成有效的事务序列,对目标服务的接口进行模糊测试.在协议分析领域,Fiterau-Brostean 等人[51] 推断出协议实现的模型.该模型描述了如何实现响应格式正确的消息序列.在 JS 引擎安全领域,Montage[52] 优化了相应的数据生成.

生成的方法不仅可以获得种子,甚至可以直接生成模糊测试模型.比如传统模糊测试中,需要通过对库的静态分析或者动态跟踪才能将其纳入到模糊测试中,FuzzGen[53] 可以在给定的环境中,利用整个系统分析来推断库的界面,进而自动合成针对复杂库的模糊器的工具,之后结合 LibFuzzer[54] 可以实现更好的代码覆盖率并检测库深处的漏洞.除此之外,FuzzGen[53] 还可以合成遵循有效 API 序列的模糊器代码.

在进行模糊测试的过程中,按照策略选取一些执行过的输入数据作为种子也是一个普遍的做法.比如 AFL[13] 会使用进化算法,以提升覆盖率为主要引导方向,从执行过的输入数据中筛选合适的数据作为种子.值得注意的是由于进化算法常常会使模糊测试过程陷入到无果的随机突变序列中,对此

NEUZZ[55]提出了使用梯度引导的方式替代进化算法.常用的选取策略都关注输入数据的执行速度,所在路径频率以及深度,并没有直接关注模糊测试未能到达的路径.为了改进选取策略,CollAFL[48]提出:如果输入的执行路径中有许多未探索到的邻居节点的分支,则该路径的变异就很可能探索到未覆盖的区域;如果输入的执行路径中,存在很多未探索到的后代节点,则该路径的变异很可能引导模糊测试探索这些后代节点;如果输入的执行路径中有许多内存访问操作,则通过变异更容易触发内存损坏漏洞.

总的来说,可以在模糊测试开始前直接使用已有高质量数据作为种子,也可以通过合适的策略进行生成,还可以是在模糊测试中按照策略从输入数据中选择.获取良好构造的种子是模糊测试得到良好检测效果的重要保障.

3.1.2 种子的筛选

获得备选种子以后,需要对种子池里的备选种子进行筛选划分不同权重,以确定每个种子要用来生成多少输入,按照什么样的顺序从种子池中选择种子.这涉及2个概念:种子的能量分配(power schedules)和种子的优先级(priority).

1) 种子的能量分配(power schedule)

一个种子蕴含的能量,代表了该种子可以生成输入数据的多少.能量越高,由该种子变异得到的输入数据就会越多.能量分配策略面临的最大问题是能量分配不合理,这会直接影响模糊测试的执行速度以及覆盖率.

比如由于在AFL[13]的能量分配策略中,没有考虑到程序不同执行路径频率上的差异,造成种子的能量分配不合理.具体而言,在模糊测试执行一段时间后,有些路径会被高频次地访问到,而部分难以到达的路径因为只有少数种子能够到达而执行频率过低.这些难以到达的路径对应的种子显然需要更多的能量.这是AFL[13]无法做到的,也因此造成探索频率相对较低的路径上存在漏洞,需要更长时间的模糊测试才能暴露出来.

为了解决这一问题,AFLFast[39]将模糊测试过程,建模为对马尔可夫链状态空间的遍历过程.将种子变异导致的程序执行路径转移概率,视为马尔可夫链上的状态转移概率.然后通过制定能量分配策略,使模糊测试过程也就是对马尔可夫链状态空间的遍历过程,更倾向于访问低访问频率的区域,少访问高访问频率的区域,使得能量分配更加合理.通过这些改进措施,AFLFast[39]在维持和AFL[13]近似的检测能力的条件下,获得了更快的检测速度.

虽然提高了检测速度,但是AFLFast[39]的能量分配策略还不是很合理.首先,它无法根据模糊测试的过程灵活地调整能量分配策略,从而增加了发现新路径的平均能量消耗,其次,从一个种子开始,选择下一个种子,并对其分配一定的能量是博弈论中经典的"探索与利用"权衡问题,而非一个简单的概率问题,基于这样的想法,EcoFuzz[40]使用多臂老虎机模型(multi-armed bandit model)建模模糊测试的过程,通过使用基于平均损耗的自适应能量分配策略,有效地减少了能量损耗,在有限时间内实现了模糊测试覆盖率的最大化.

能量分配策略不止可以用于提升模糊测试执行速度,还可以引导模糊测试到指定的程序位置进行检测,比如AFLGo[46]针对传统的灰盒模糊测试存在的局限性,提出了导向型灰盒模糊测试(directed greybox fuzzing),用于对目标程序给定的位置进行模糊测试.AFLGo[46]通过使用基于模拟退火算法(simulated annealing)的能量调度方法,逐渐将能量更多地分配到距离目标更近的种子上,并减少距离目标较远的种子上的能量分配.AFLGo[46]相比于使用符号执行的白盒模糊测试和非直接灰盒模糊测试,在探查给定目标程序上获得了更好的效果.但是AFLGo[46]在路径选择上偏向于距离目标更近的路径,这可能会导致漏掉一些潜在的漏洞.

为了克服这个问题,Hawkeye[47]通过使用静态分析,全面地搜集与目标程序位置相关的调用图、函数以及基本块层面的距离信息.通过使用和AFLGo[46]相同的基本块路径距离(basic block trace distance)以及新增的覆盖函数相似度(covered function similarity)这2个指标,制定能量分配策略.对应覆盖函数相似度越大,基本块路径距离越小的种子将会被分配到更大的能量,产生更多的测试数据.

值得注意的是,AFLFast[39]的能量分配并不会显著地改善模糊测试器检测漏洞的能力极限,只是缩短了达到检测能力极限的时间,而AFLGo[46]以及Hawkeye[47]的能量分配更多的是有目标的引导模糊测试方向,而不是为了拓展程序的覆盖率.

2) 种子的优先级

种子的优先级决定了模糊测试从种子池中选择种子的顺序.

以经典的AFL[13]为例,AFL[13]引入了进化算法(genetic algorithm),该算法中使用了适应度函数

(fitness function),通过适应度函数可以对输入进行评估,并从中选取最佳的输入作为种子放入种子池,该种子池实际上是一个队列,种子按照入队顺序依次被用于模糊测试.这种方法,在一段时间后会因没有新种子的加入而被固定下来,形成一个循环,这也带来 AFL[13] 的一些问题.AFLFast[39] 在种子优先级的确定上做出了自己的改进,不再严格遵循先入队先执行的策略,而是会适当提升比较少被执行种子的优先级,也会提升能量分配较少的种子的优先级,尽量平衡在不同路径上的检测密度.

对于定向灰盒模糊测试,种子的优先级选择会有所不同,比如在 Hawkeye[47] 中,根据种子是否触发了新的执行路径,种子与目标种子的相似度(比如分配的能量),种子是否包含有目标函数,将种子分配到 3 个等级的队列中.

值得注意的是种子的长度是影响种子优先级的一个重要因素.通常而言,长度越短的输入数据,会占用越少的内存,加快模糊测试的速度.所以在对覆盖率增加贡献相同的情况下,长度更短的输入会获得更高的优先级.这一思想在 Rebert 等人[56] 的工作中通过详细的实验做出了验证,他们认为多个包含不同程序基本块的小文件,在效果和效率上要好于一个包含全部基本块的大文件,因而算法会更倾向于更小的输入.值得注意的是,AFL[13] 采用了在维持覆盖率不变的条件下,通过反复地删除种子的不同部分,来试图缩短种子的大小的方法.

更多的种子筛选策略在 Rebert[56] 的工作中通过大量的实验做出了对比,总的来说模糊测试中使用良好的种子筛选方法要好于随机采样的方法.

3.1.3 种子突变

通过种子获取和种子的筛选这 2 个环节后,获得了大量种子以及对应的能量分配和优先级策略,接下来需要在突变策略的指引下快速地生成大量输入数据.变异的策略直接决定了变异生成数据的好坏,变异策略太保守,导致覆盖率偏低,变异策略太激进,导致生成大量无效的输入,严重影响模糊测试的效率.

突变的具体操作大体上有 6 种:比特翻转(bit-flips)、简单算数运算(simple arithmetic)、覆盖(over-writing)、插入(inserting)、删除(deleting)、拼接(splice).比特翻转是指按比特位翻转,通常会按照一定的步长,连续翻转几个比特位,或者是在随机或特定的某个位置,对固定宽度的比特位进行翻转.简单算术运算是 AFL[13] 中使用的一种方法,该方法按照 8 b 的步长,依次按照 8 b,16 b,32 b 的宽度,从头开始进行加减操作.覆盖、插入是通过使用预设的值将种子中的一些部分进行覆盖,或者将其插入到种子的一些位置,预设的值可以是随机生成的,也可以是用户指定的.删除是删除种子的某些部分、拼接会选择 2 个差异较大的种子进行拼接,得到新的数据.

根据突变所依据的策略不同,我们将突变的方式划分为 2 种:黑盒突变(black-box mutation)和导向型突变(guided mutation).

1) 黑盒突变

所谓黑盒突变是指不依赖于目标相关信息,依照随机突变策略对种子进行突变的方法.这种类型的突变方式优点是可以快速大量地生成输入.AFL[13] 中大量使用了这种突变方式,通过精心构造的突变策略,获得良好的实际测试效果.这种方法不关心特定突变方式对程序状态的影响.黑盒突变常常作为整个突变策略的一部分出现.

2) 导向型突变

导向型突变是指有引导性的对种子进行突变,使突变后的输入尽可能地将程序引导到预期的程序状态,或者获得相应性能的提升.具体来说,导向性的突变又可以细分为程序状态导向型突变和性能导向型突变.

程序状态导向型突变,通过程序分析技术,得到种子与程序状态的关系,以此制定突变策略,针对感兴趣的程序状态生成相应的测试数据.

比如 VUzzer[41],通过使用静态和动态结合的程序分析方法,获得程序的控制流和数据流,控制流指导 VUzzer[41] 在种子突变的时候选择更深的路径,而数据流指导 VUzzer[41] 具体如何对种子进行突变.该突变方法被称为感知突变策略(application-aware mutation strategy).

确定突变位置是突变策略中的重要一环,通常使用污点分析等技术,寻找输入与程序状态的关系,进而确定突变位置.

比如 Angora[42] 使用可伸缩的字节级别的污点分析指明突变位置.VUzzer[41] 和 REDQUEEN[43] 也使用到了数据污染的相关技术,试图寻找合适的突变位置,但是他们只能分析输入数据或者输入数据的直接拷贝同程序分支的约束的关系,而无法有效处理非直接拷贝的情况.对于这个问题 GREYONE[25] 通过使用模糊测试驱动的污点推断技术(fuzzing-driven taint inference),获得污点属性(taint attributes),解决了该问题.

对于输入格式高度结构化的目标,传统的种子突变策略不太适用,需要保证既可以获得足够的输入数据,又不破坏种子的结构,比如在 Skyfire[44] 中,为了防止突变修改关键字段,只能选择种子的特定位置进行突变.ProFuzzer[57] 认为结构化的输入,可以根据不同的语义,划分成不同的字段,在不了解其底层语法和语义的情况下进行突变,会生成大量的无效输入.该方法自动恢复并分析对于模糊测试过程中发现漏洞至关重要的输入字段,智能地调整突变策略以重点对这些重要字段进行突变,从而提高了模糊测试的效率.

除了突变位置的选择外,覆盖、替换和插入所使用的备选字段的有效性也需要考虑到.比如 LangFuzz[58] 中,通过分析提供的输入目标语法以及足够多的有效输入样例,得到有效的代码段并存入代码段池中等待使用,同时 LangFuzz[58] 还可以使用步进算法自行生成有效的代码段.

此外在对 JS 引擎进行模糊测试的方案中,DIE[59] 使用轻量化的动态和静态程序分析技术,实现了被命名为结构保存(structure preservation)和类型保存(type preservation)的 2 种与程序状态相关的突变策略.

有时需要根据程序执行状态,及时调整突变策略.Hawkeye[47] 设计了自适应突变策略,策略中划分了轻微突变(fine mutation)以及粗粒度突变(coarse mutate)2 种类型,每种类型的突变方式都被分配了相应的概率,根据实际的执行情况,动态的调整 2 种突变方式的概率分配.

突变除了可以应用到种子上,还可以用于程序本身.T-Fuzz[34] 中通过对程序中非必要校验进行二进制的覆写,实现了种子对于程序非必要校验的绕过.

性能导向型突变,不会试图分析输入与程序内部状态的关系,而是根据输入数据同模糊测试评估指标的关系制定突变的策略.

比如在 AFL[13] 中进行连续字节翻转的时候,会衡量翻转每个字节时对于覆盖率提升的程度,如果覆盖率没有上升,则认为该字节对于覆盖率提升无意义.通过该过程可以获得一个 effectmap,用于存储种子中所有字节的有效性(以 0,1 区分),突变的时候如果该字节无效(为 0)则跳过突变,否则按策略进行突变.

总的来说,黑盒突变可以快速地获取大量输入数据.程序状态导向型的突变,更适合用于有目标导向的模糊测试.性能导向型的突变,可以在性能指标上有良好的体现.一个优秀的突变策略往往是将 3 种突变策略结合使用,针对不同的应用场景进行合适的策略组合.

3.2 输入选择

输入选择的主要目的是在输入被实际执行前,尽量将其中的无效数据滤除掉,以节省执行时间.虽然通过良好的策略,可以尽量保证输入数据的质量,但是无效数据仍然是多数,造成模糊测试运行缓慢效果不佳.

输入选择环节主要的挑战是如何在输入数据执行之前就识别出数据是否有效,这是一个模式识别的问题,使用机器学习技术是一个不错的选择.接下来我们对直接灰盒模糊测试中的输入选择工具 FuzzGuard[60] 进行介绍.

FuzzGuard[60] 是针对直接灰盒模糊测试而设计的,但是对其他类型的模糊测试仍然有启示作用.FuzzGuard[60] 通过使用机器学习技术,预测新产生的输入可否使得程序执行到有漏洞的代码处,将结果预测为不可达的输入滤除,配合之后的模糊测试工具进行高效地定向模糊测试.

FuzzerGuard[60] 中讨论了机器学习应用到模糊测试所要面对的一些问题,比如数据不平衡问题,具体来说输入数据中真正有效的数据只占少数,正负样本非常不平衡,这对于机器学习模型的训练是不利的.而且由于突变后种子执行路径变化的随机性,使用类似图像处理领域的数据扩展方法也是不可行的.FuzzGuard[60] 针对这一问题提出了步进式的训练方法(step-forwarding approach)一定程度上缓解了该问题.

基于覆盖率的模糊测试等其他模糊测试方法也可以尝试借鉴该思想,通过进一步筛选待执行的输入数据,提升模糊测试的效率.需要注意的是,这个环节不是必要的.如果增加输入选择的环节可以在有限的资源消耗下,大幅提升模糊测试整体效率,那么增加该环节才是有意义的.

3.3 评 估

评估的主要目的是选取一个合适的评估指标,用于评估模糊测试执行结果,从而帮助模糊测试制定合理的策略并反映模糊测试检测漏洞的真实能力.

现阶段的研究都会聚焦于模糊测试器在 2 个指标上的表现:覆盖率和暴露漏洞平均时间.

选择了合适的评估指标以后,如何设计实验,以评估一个模糊工具是否有效?将是评估环节中最大的挑战.

3.3.1 覆盖率

覆盖率是软件测试中的一个衡量指标,指的是在测试过程中,对象被覆盖到的数目占总数的比例.通常而言,高覆盖率更可能发现更多的隐藏漏洞,众多研究因此集中在覆盖率提升上.

在覆盖率使用的对象上,可以有不同选择.比如AFL[13]使用上下文无关的边覆盖率作为评估指标.相比较而言,Angora[42]使用上下文敏感的分支覆盖率.此外VUzzer[41]使用了块覆盖率替代边覆盖率作为覆盖率评估的对象.

对于复杂程序的覆盖信息统计是一个重要环节,传统的采用位图跟踪的方法比较粗略,直接使用Hash算法,也会带来严重的冲突问题.覆盖信息记录不够完整与详细,会造成覆盖率的不准确.CollAFL[48]通过使用足够大的存储空间,以及精心设计的Hash冲突解决方案,确保每一个基本块都有一个独特的ID,每一个边都有一个唯一的Hash值,很好地解决了冲突问题,为模糊测试提供更精确完整的覆盖信息.

一味地提升覆盖率是唯一正确的研究方向吗?虽然这从意图找全所有潜在漏洞的目标上来看是有道理的,然而TortoiseFuzz[61]认为专注于提升覆盖率,而将已经覆盖到的内容视为同等重要的这一思想是不合理的,这可能导致在某些不包含漏洞的边或基本块上浪费过多的检测时间.而且使得模糊测试更容易受到反模糊测试技术的影响.TortoiseFuzz[61]提出了边缘安全敏感性的概念,通过从函数、循环与基本块3个层次评估边缘安全敏感性,并依次对输入数据进行优先级排序.

此外,研究人员能够在覆盖率提升的工作中起什么作用,同样值得思考.IJON[62]提出,可以通过人工介入,发现阻碍模糊工具的常量,进而使用一段自定义注释解决该障碍,以使得覆盖率能够快速提升.

覆盖率的统计是需要消耗资源的.研究人员发现,用于标识代码覆盖率的插桩技术,对于程序的执行效率有着非常严重的影响.相比于黑盒测试而言,灰盒测试(以AFL[13]为例)引入了将近1300%的开销.如何降低开销是需要考虑的问题.对此,UnTracer[20]发现在测试过程中,只有不到万分之一的测试样例,能够带来覆盖率的增长,可以说对于大部分种子,覆盖率跟踪都是无效的.换而言之,我们只跟踪覆盖率增长的部分,就可以有效减少开销.

3.3.2 暴露漏洞平均时间

对于部分特殊场景,例如需要根据漏洞报告复现某个已知未知的漏洞之类的情况,覆盖率不再是一个合适的评估指标.此外,在一个程序中漏洞分布并不均匀,部分代码更有可能存在漏洞,而部分代码可能根本没有漏洞.纯粹以覆盖率为指标,可能会使模糊工具在不存在漏洞的部分浪费过多精力.基于这一特点,定向灰盒模糊的概念被提出,它旨在快速找到一个特定位置的漏洞.它不再使用代码覆盖率为主要指标,而是以一种暴露漏洞平均时间的指标来进行评估.

暴露漏洞平均时间被Böhme[46]提出的AFLGo[46]选做评估指标.在与基准模糊工具AFL[13]的对比实验中,AFLGo[46]复现单个漏洞的时间远比AFL少,这证明了该评估指标的有效性.此后Hawakeye[47]同样使用了这一指标来证明其性能.Parmesan[63]在通过sanitizer来发现潜在漏洞的研究工作中,也是用暴露漏洞平均时间作为评估指标同Angora[42]等定向灰盒模糊工具进行比较.

暴露漏洞平均时间相对于传统的覆盖率指标,更接近模糊工具的本质.模糊工具的目标就是发现漏洞,覆盖率这一指标只是基于覆盖率越高可能触发更多漏洞的假设.当然暴露漏洞平均时间也有自身的局限性,由于定向模糊的原因,发现的漏洞数目可能有限,而覆盖率指标有助于模糊工具发现更多的漏洞.

3.3.3 验证实验的设计

应该如何设置实验以评估一个模糊工具是否有效?对此,Klees等人[64]进行了实验并提出了一些相应标准.

Klees等人[64]认为目前的评估方法存在诸多不足:比如大多数论文未能进行多次执行,而实验证明单次执行可能导致结果存在较大偏差.同时一些论文中并不是以发现的漏洞数目而是以去重后的崩溃状况(unique crash)作为评判标准.实际情况中,unique crash的数目远远比实际的漏洞数目要多.这可能导致对性能的高估.并且许多论文使用的测试超时时间过短,实验证明,较高的超时时间才能体现模糊测试方法的真实性能.最后许多论文都忽略了初始种子可能对实验产生的较大影响,而且不同论文对目标的选择也不统一,这对会对真实性能评估产生很大影响.

Klees等人[64]推荐研究人员在设计实验时遵循4个标准:1)要进行多次实验,并进行统计与检验以区获得结果的分布情况;2)应使用例如CGC/LAVA等具有确定错误的测试集或使用具有已知

漏洞的程序作为被测试对象;3)应考虑尝试各种种子输入,比如使用空种子;4)超时时间应至少设置为24 h,或尝试不同超时对性能造成的影响.

4 结果分析

结果分析发生在模糊测试结束以后,主要目的是对于模糊测试的输出信息进行分析和处理.面临的挑战是对于得到的输出状态,安全人员通常要手动去重,然后进行复现并分析根本原因,最后根据威胁程度,判断是否是有意义的漏洞.该过程高度依赖于相关领域知识以及必要的漏洞分析和复现能力.

为了加快结果分析环节的速度,节省安全研究人员的精力,将机器学习技术应用到该环节成为一个研究方向.Gary 等人[65]介绍了机器学习在该环节的应用情况,感兴趣的读者可以阅读相应文章.

该环节是模糊测试不可或缺的一部分,但不属于模糊测试主要工作,本文在这里不做过多讨论.

5 具体应用场景下的模糊测试

我们已经介绍了模糊测试的基本工作流程,让读者对模糊测试有一个整体的了解,然而在不同的领域中实际使用模糊测试时,通常会有着自身的特点.

本节会针对不同的领域,介绍模糊测试的研究成果,重点介绍近年来研究最为集中的 2 个领域,分别是物联网的模糊测试以及内核模糊测试,其他领域的研究会在 5.3 节中集中介绍.之所以选择这 2 个领域详细讨论,一方面是因为相应的研究成果相对较多,可以被集中论述,另一方面是因为物联网领域在近几年快速发展,留下了大量的安全隐患,而内核作为操作系统的核心程序,一旦存在漏洞,将带来致命的威胁,两者有必要单独讨论.

5.1 物联网中的模糊测试

物联网是近几年最受关注的研究领域之一,大量的物联网设备从实验室进入到普通人的家庭中,然而物联网领域快速地发展,遗留了大量安全问题.在复杂的物联网应用环境下使用模糊测试技术,面临着极大的需求和挑战.

物联网设备的威胁源自于哪里?针对这个问题,FIRM-AFL[66]认为通过利用物联网固件中的软件漏洞,实现针对物联网设备的攻击是物联网设备面临严重威胁的主要来源.

将模糊测试应用到物联网领域,首先要面对的问题就是特定物联网设备上运行的程序通常对于其实际硬件的配置有着高度的依赖性.简单的从固件中提取一个用户级别的程序,然后使用模糊测试进行检测,通常是行不通的.为了解决这个问题,IoTFuzzer[30]认为大多数物联网设备通过其官方移动应用程序进行控制,并且此类应用程序通常包含有与设备进行通信所使用协议的丰富信息,因此通过识别和重用特定于程序的逻辑(比如加密)来改变测试用例(尤其是消息字段),就能够有效地对物联网目标进行模糊测试,而无需依赖于有关其协议规范的任何知识.

检测速度是模糊测试的重要指标,而大部分的物联网设备远远无法满足模糊测试需要的吞吐率,为了提高吞吐率 AVATAR[67]通过提供更好的硬件组件支持来构建一个混合执行环境,使嵌入式固件的动态程序分析成为可能.为了进一步提高吞吐率,Firmadyne[68]为系统模式 QEMU 增加了物联网固件的硬件支持,并通过修改内核和驱动程序来处理由于缺乏实际硬件而导致的物联网异常,从而完全仿真系统.它同时支持 ARM 和 MIPS 这 2 种架构.

多项研究成果表明物联网设备采用全仿真的系统,可以获得最高的吞吐量,这是因为真实的物联网设备相比于桌面工作站或者服务器要慢得多.

FIRM-AFL[66]认为即使采用完全的系统仿真,吞吐量也远不能达到理想的水平.其开销主要源自于使用软件实现内存管理单元(SoftMMU)以及模拟系统调用.解决方法是通过启用可在系统仿真器中被仿真的 POSIX 兼容固件,并使用增强过程仿真技术.

除了吞吐率带来的挑战,模糊测试在物联网领域还要面对检测能力下降的挑战,比如 Muench 等人[69]在文中通过实验证明了内存损坏通常会导致嵌入式设备与台式机系统发生不同的行为.特别是在嵌入式设备上,内存损坏的影响通常不太明显,这大大降低了模糊测试的检测能力.

模糊测试应用于物联网领域有着迫切的需求,但是却受限于物联网领域独特的特点,从传统领域诞生的模糊测试技术很难直接使用到物联网领域,物联网设备的吞吐量通常难以达到模糊测试的要求,而且物联网领域中特别的异常状态对于模糊测试的检测能力也提出了挑战.

5.2 内核安全中的模糊测试

内核是构成计算机操作系统的核心程序,可以完全控制操作系统中的一切事物.内核中存在的漏洞,对整个操作系统有着致命的威胁.

内核模糊测试通常利用暴露出来的系统调用接口和外围接口,从用户空间进入到内核组件中进行模糊测试,以检测内核中可能存在的漏洞.

内核模糊测试面临一系列挑战,首先由于Windows内核程序以及很多相关组件的源代码并不开源,将导致传统的反馈机制不再适用.其次内核中的代码由于存在中断、多线程操作等机制,使得模糊测试变的很复杂.最后内核模糊测试一旦检测到程序的崩溃,将会导致整个操作系统重新启动,极大地影响了模糊测试的效率.

为了克服这些问题,产生了一些专门针对内核安全进行模糊测试的研究,比如 syzkaller[70] 是由Google开发的一种以获得高覆盖率为导向的内核模糊测试工具,目前实际使用比较多,并经常被用于对比实验中.Razzer[71] 是基于 syzkaller[70] 通过使用LLVM和修改后的SVF技术针对内核中存在的数据争用问题进行的模糊测试.TriforceAFL[36] 基于AFL[13] 的QEMU模式,在系统模拟器的帮助下,通过跟踪分支信息,对 Linux 的内核进行模糊测试.kAFL[37] 通过利用 Intel 提供的进程追踪技术,获取代码运行时的控制流信息,并通过使用 Intel 的硬件虚拟特征(VT-x)提升效率并使得 kAFL[37] 独立于特定的操作系统.

为了缓解由于内核代码执行时间过长,测试案例互相干扰,以及内核崩溃带来的内核模糊测试性能上的损失,Agamotto[72] 提出一个轻量化的虚拟机检查点作为新的原语,并动态的创建多个检查点,提高了内核模糊测试的吞吐量.通过使用 Linux 中的USB和PCI驱动程序作为外围攻击面对其评估,平均性能相比于 syzkaller[70],提升了 66.6%.

为了将混合模糊测试方法运用到内核模糊测试中,诞生了 HFL[73],实验中获得了同等数量资源下的相比于 syzkaller[70] 高出 26%的代码覆盖率,以及快 3 倍以上的速度.

对于内核安全的威胁可以是源自于外部固件.比如 PeriScope(fuzz)[38] 研究发现存在一些不通过系统调用的额外路径也会导致内核泄漏,比如通过破坏 WiFi 芯片组的固件向内核发送恶意输入.对此 PeriScope(fuzz)[38] 通过被动的检测驱动与其硬件的流量以及模拟攻击者在外围设备上的攻击行为,在实际实验中,对 2 款比较流行的芯片产品,通过WiFi组件和驱动进行模糊测试,发现了 15 个独特漏洞,其中 9 个是首次发现的.

考虑到大量的驱动程序是由第三方开发者提供的,安全性存疑,DIFUZE[74] 通过自动的识别驱动程序,将其映射到设备文件名,再自动地构造复杂的参数实例,对内核驱动程序进行了模糊测试.实验中在7款现代 Android 智能手机上检测出了内核中 32 个从未被发现的漏洞.

通过 5.2 节介绍,我们可以看到,传统的模糊测试应用到内核模糊测试上将面临着众多的问题,包括内核态代码的复杂执行环境,以及内核崩溃的处理问题.另外内核安全面临的威胁来源是不确定的,可以是外部固件,也可以是第三方开发的驱动程序,还可以是内核程序自身设计的问题.模糊测试在内核安全领域的应用还有待进一步的研究.

5.3 其他领域

模糊测试除了 5.1～5.2 节提到的在物联网以及内核安全领域的应用,在安卓系统安全,算法复杂度等方面也有着广泛的应用,这里我们集中介绍近年来模糊测试在多个领域的研究成果.

SpecFuzz[75] 使用动态推测执行来发现类似于 Spectre V1 等类型的漏洞.FANS[50] 使用接口模型和推断依赖关系生成测试用例,对安卓系统的本机服务进行模糊工具.Frankenstein[76] 通过构建一个固件模拟器,使用虚拟调制解调器对蓝牙协议进行了模糊测试.

USB驱动由于能够直接与内核交互而被视为具有高风险.然而对 USB 协议的模糊测试需要跨越软件与硬件,因此颇具难度.对此,USBFuzz[77] 提出了基于覆盖率的针对 USB 驱动的模糊测试.

灰盒测试的代码覆盖率并不能很好地反映多线程程序中存在的问题.MUZZ[55] 通过使用多种插桩技术,选择在单线程中表现更好的种子来探索多线程程序.对于多线程的数据竞争问题,KRACE[78] 引入新的覆盖率跟踪指标与进化算法,进行模糊测试.

此外还有针对文件系统的模糊工具 JANUS[79],针对错误处理代码进行模糊测试的 FIFUZZ[80],针对 Java 算法复杂度进行模糊测试的工具 HotFuzz[81],面向虚拟机监控器的模糊工具 HYPER-CUBE[82],针对算法复杂度进行模糊测试的 SlowFuzz[83],以及通过自然语言处理进行信息提取,进而进行基于语义模糊测试的 SemFuzz[84],还有针对 Web 浏览器 DOM 引擎的模糊测试工具 FreeDom[85].

模糊测试在各个领域都有着不同的应用,虽然各自有着自身独特的使用条件,但是整个模糊测试

的工作流程仍然是沿着我们提出的基本工作模型进行,只是在具体实现的时候,需要按照实际情况寻找合适的解决方案.

6 模糊测试的挑战与机遇

模糊测试在漏洞检测上起着重要的作用,但这一技术同样可以被攻击者利用,为恶意攻击者提供便利,因此针对模糊测试的对抗技术反模糊测试出现了,无论是反模糊测试自身的研究,还是其带给模糊测试技术的挑战,都是研究的重要方向.

除此之外,模糊测试的种类复杂繁多,每一个应用领域都有着自身独特的需求,能否将各种各样的模糊测试工具集成起来,形成一个通用的平台,是模糊测试实用化的一个重要挑战和方向.

近几年来,机器学习的相关技术飞速发展,已经在图像识别,自然语言处理等领域大放异彩,模糊测试与其相结合可以说是趋势所在,但是找到两者合适的结合点,最大限度发挥两者的优势仍然是一个需要研究的方向.

6.1 反模糊测试

目前主流的模糊测试技术通常依赖于 4 个前提:1)单次执行速度要足够快;2)模糊工具可以获得覆盖率的反馈;3)目标程序中的路径约束可以被符号求解;4)崩溃可以被模糊工具所检测到.

基于这些依赖条件,为了实现反模糊测试,要在不对正常执行造成过大的影响的条件下,减缓模糊测试的执行速度,或者通过干扰覆盖率图,使得模糊工具无法从覆盖率图中获取有效信息,也可以干扰混合执行,使得正常能够被符号执行或污点分析求解的内容变得无法求解.

Güler 等人[86]和 Jung 等人[87]都提出了在低频执行路径插入延时代码,以减缓模糊测试速度的方案.并通过将常量比较等价替换为 Hash 后的比较干扰混合执行.在干扰覆盖率获取的方向上,他们都提出用无意义的代码填充覆盖率图,进而干扰模糊工具的判断.

反模糊测试技术一方面可以阻止恶意模糊测试对安全的威胁,另一方面也给今后的模糊测试技术提出了新的挑战.

6.2 模糊工具的集成

通过 6.1 节的内容可以看到,研究人员针对不同的领域提出不同的模糊测试方法,产生了种类繁多的模糊测试工具.研发出一种适用于所有场景的模糊工具并不现实,如何将不同模糊测试工具进行整合,构造一个通用模糊测试平台,是模糊测试实用化的一个重要研究方向.

6.3 机器学习在模糊测试的应用

机器学习技术在图像识别以及自然语言处理领域有着广泛的应用,将其使用到模糊测试领域是一个非常值得探索的领域,但是仍然有很多挑战需要应对.

首先要面对的就是数据源问题,模糊测试虽然可以生成大量的输入数据,但是大多是即时生成的,无法在不同的测试目标之间通用,这就对于构建通用数据集,方便模型训练,造成了很大的阻碍,也导致了无法生成一个提前训练好的,可被多种模糊测试使用的预训练模型,只能使用模糊测试实时生成的数据训练模型.

其次就是数据不平衡问题,模糊测试生成的数据中,能够真正实现测试目的的数据,在数量上相对偏少,正负样本的极度不平衡,使用步进式训练的方法只能缓解该问题,但无法从根本上解决.

最后,执行速度对于模糊测试来说非常重要,而复杂机器学习模型的训练往往会耗费大量时间,并且已有的不使用机器学习技术的模糊测试工具在性能上尚能满足人们的需求,模糊测试对于机器学习技术的需要并不迫切.

关于机器学习与模糊测试结合的内容,感兴趣的读者可以阅读 Saavedra 等人[65]以及 Wang 等人[88]的文章,这里不做过多的叙述.

7 结束语

模糊测试自诞生以来,能力不断提升,应用领域不断扩展,今日的模糊测试工具,已经逐渐成为漏洞发掘的重要工具.

本文对模糊测试进行了全面的总结.我们介绍了模糊测试流程中,各个环节的研究情况.也总结了具体场景下的模糊测试取得的进展.最终对模糊测试面临的机遇和挑战进行展望.

模糊测试的设计时刻体现着博弈的思想,是“精确”一些还是“模糊”一些,不同的策略可以带来不同的效果,总的来说模糊测试与其说是一门技术,不如说是一门艺术.

参 考 文 献

[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities [J]. Communications of the ACM, 1990, 33(12): 32-44

[2] Langner R. Stuxnet: Dissecting a cyberwarfare weapon [J]. IEEE Security & Privacy, 2011, 9(3): 49-51

[3] Mohurle S, Patil M. A brief study of WannaCry threat: Ransomware attack 2017 [J]. International Journal of Advanced Research in Computer Science, 2017, 8(5): 1938-1940

[4] DeFazio P A, Larsen R. The design, development & certification of the boeing 737 max [OL]. 2020 [2020-11-11]. https://transportation. house. gov/imo/media/doc/2020. 09. 15% 20FINAL% 20737% 20MAX% 20Report% 20for% 20Public%20Release.pdf

[5] Hawkes B. Project zero five years of 'make 0day hard' [OL]. Las Vegas: Black hat, 2019 [2020-11-11]. https://i. blackhat.com/USA-19/Thursday/us-19-Hawkes-Project-Zero-Five-Years-Of-Make-0day-Hard.pdf

[6] Zhang Yan, Zhang Junwen, Zhang Dalin, et al. Survey of directed fuzzy technology [C/OL] //Proc of the 9th IEEE Intl Conf on Software Engineering and Service Science (ICSESS). Piscataway, NJ: IEEE, 2018 [2020-11-11]. https://ieeexplore.ieee.org/abstract/document/8663772

[7] Wang Pengfei, Zhou Xu. Sok: The progress, challenges, and perspectives of directed greybox fuzzing [J]. CoRR preprint, CoRR: abs/200511907, 2020

[8] Li Jun, Zhao Bodong, Zhang Chao. Fuzzing: A survey [J]. Cybersecurity, 2018, 1(1): 1-6

[9] Liang Hongliang, Pei Xiaoxiao, Jia Xiaodong, et al. Fuzzing state of the art [J]. IEEE Transactions on Reliability, 2018, 67(3): 1199-1218

[10] Man'es V J M, Han H, Han C, et al. The art, science, and engineering of fuzzing: A survey [J/OL]. IEEE Transactions on Software Engineering, 2019 [2020-11-11]. https://ieeexplore.ieee.org/abstract/document/8863940

[11] Böhme M, Cadar C, Abhik R. Fuzzing challenges and reflections [J/OL]. IEEE Software, 2020 [2020-11-11]. https://ieeexplore.ieee.org/document/9166552

[12] Godefroid P. Fuzzing: Hack, art, and science [J]. Communications of the ACM, 2020, 63(2): 70-76

[13] Zalewski M. American fuzzy lop [CP/OL]. [2020-11-11]. https://lcamtuf.coredump.cx/afl/

[14] Kaksonen R, Laakso M, Takanen A. Software Security Assessment Through Specification Mutations and Fault Injection [M]. Berlin: Springer, 2001: 173-183

[15] Eddington M. Peach fuzzing platform [CP/OL]. [2020-11-11]. https://community.peachfuzzer.com/WhatIsPeach.html

[16] Godefroid P. Random testing for security: Blackbox vs whitebox fuzzing [C/OL] //Proc of the 22nd IEEE/ACM Int Conf on Automated Software Engineering (ASE 2007). Piscataway, NJ: IEEE, 2007 [2020-11-11]. https://doi.org/10.1145/1292414.1292416

[17] Security M. Funfuzz [CP/OL]. [2020-11-11]. https://github.com/MozillaSecurity/funfuzz

[18] Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing [C] //Proc of the 31st IEEE Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2009: 474-484

[19] Bellard F. QEMU a fast and portable dynamic translator [C] //Proc of USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2005: 41-46

[20] Nagy S, Hicks M. Full-speed fuzzing reducing fuzzing overhead through coverage-guided tracing [C] //Proc of 2019 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2019: 787-802

[21] Schwartz E J, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask) [C] //Proc of 2010 IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2010: 317-331

[22] Huang Heqing, Yao Peisen, Wu Rongxin, et al. Pangolin incremental hybrid fuzzing with polyhedral path abstraction [C] //Proc of the 2020 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2020: 1613-1627

[23] Cho M, Kim S, Kwon T. Intriguer field-level constraint solving for hybrid fuzzing [C] //Proc of the 2019 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2019: 515-530

[24] He Jingxuan, Balunović M, Ambroladze N, et al. Learning to fuzz from symbolic execution with application to smart contracts [C] //Proc of the 2019 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2019: 531-548

[25] Gan Shuitao, Zhang Chao, Chen Peng, et al. GREYONE data flow sensitive fuzzing [C] //Proc of the 29th USENIX Security Symposium. Berkeley, CA: USENIX Association, 2020: 2577-2594

[26] Godefroid P. From blackbox fuzzing to whitebox fuzzing towards verification [C/OL] //Proc of the 2010 Int Symp on Software Testing and Analysis. 2010 [2020-11-11]. https://citeseerx. ist. psu. edu/viewdoc/download?doi = 10. 1. 1. 675. 8778&rep=rep1&type=pdf

[27] Lee S, Yoon C, Lee C, et al. DELTA a security assessment framework for software-defined networks [C/OL] //Proc of NDSS 2017. 2017 [2020-11-11]. https://dblp. org/db/conf/ndss/ndss2017.html#LeeYLSYP17

[28] Veggalam S, Rawat S, Haller I, et al. IFuzzer an evolutionary interpreter fuzzer using genetic programming [C] //Proc of European Symp on Research in Computer Security. Berlin: Springer, 2016: 581-601

[29] Han H, Cha S. IMF: Inferred model-based fuzzer [C] //Proc of the 2017 ACM SIGSAC. New York: ACM, 2017: 2345-2358

[30] Chen Jiongyi, Diao Wenrui, Zhao Qingchuan, et al. IoT Fuzzer discovering memory corruptions in IoT through app-based fuzzing [C/OL] //Proc of NDSS 2018. San Diego, CA: University of California, 2018 [2020-11-11]. https://dblp.org/db/conf/ndss/ndss2018.html#TsirantonakisII18

[31] Stephens N, Grosen J, Salls C, et al. Driller augmenting fuzzing through selective symbolic execution [C/OL] //Proc of NDSS 2016. San Diego, CA: University of California, 2016 [2020-11-11]. https://dblp.org/db/conf/ndss/ndss2016.html#StephensGSD WCSK16

[32] Yun I, Lee S, Xu Meng, et al. QSYM a practical concolic execution engine tailored for hybrid fuzzing [C] //Proc of the 27th USENIX Security Symp. Berkeley, CA: USENIX Association, 2018: 745-761

[33] Zhao Lei, Duan Yue, Yin Heng, et al. Send hardest problems my way probabilistic path prioritization for hybrid fuzzing [C/OL] //Proc of NDSS 2019. San Diego, CA: University of California, 2019 [2020-11-11]. https://dblp.org/db/conf/ndss/ndss2019.html#ZhaoDYX19

[34] Peng Hui, Yan S, Payer M. T-Fuzz: Fuzzing by program transformation [C] //Proc of 2018 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2018: 697-710

[35] Chen Peng, Liu Jianzhong, Chen Hao. Matryoshka: Fuzzing deeply nested branches [C] //Proc of the 2019 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2019: 499-513

[36] Nccgroup. TriforceaFL [CP/OL]. [2020-11-12]. https://github.com/nccgroup/TriforceAFL

[37] Schumilo S, Aschermann C, Gawlik R, et al. kAFL hardware-assisted feedback fuzzing for OS kernels [C] //Proc of the 26th USENIX Security Symp. Berkeley, CA: USENIX Association, 2017: 167-182

[38] Song D, Hetzelt F, Das D, et al. PeriScope an effective probing and fuzzing framework for the hardware-os boundary [C/OL] //Proc of NDSS 2019. 2019 [2020-11-11]. https://dblp.org/db/conf/ndss/ndss2019.html#ZhaoDYX19

[39] Böhme M, Pham V T, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain [J]. IEEE Transactions on Software Engineering, 2017, 45(5): 489-506

[40] Yue Tai, Wang Pengfei, Tang Yong, et al. EcoFuzz adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit [C] //Proc of the 29th USENIX Security Symp. Berkeley, CA: USENIX Association, 2020: 2307-2324

[41] Rawat S, Jain V, Kumar A, et al. VUzzer application-aware evolutionary fuzzing [C/OL] //Proc of NDSS 2017. San Diego, CA: University of California, 2017 [2020-11-11]. https://dblp.org/db/conf/ndss/ndss2017.html#0001JKCGB17

[42] Chen Peng, Chen Hao. Angora efficient fuzzing by principled search [C] //Proc of 2018 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2018: 711-725

[43] Aschermann C, Schumilo S, Blazytko T, et al. REDQUEEN fuzzing with input-to-state correspondence [C/OL] //Proc of NDSS 2019. San Diego, CA: University of California, 2019 [2020-11-11]. https://dblp.org/db/conf/ndss/ndss2019.html#ZhaoDYX19

[44] Wang Junjie, Chen Bihuan, Wei Lei, et al. Skyfire data-driven seed generation for fuzzing [C] //Proc of 2017 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2017: 579-594

[45] Bastani O, Sharma R, Aiken A, et al. Synthesizing program input grammars [J]. ACM SIGPLAN Notices, 2017, 52(6): 95-110

[46] Böhme M, Pham V T, Nguyen M D, et al. Directed greybox fuzzing [C] //Proc of the 2017 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2017: 2329-2344

[47] Chen Hongxu, Xue Yinxing, Li Yuekang, et al. Hawkeye towards a desired directed grey-box fuzzer [C] //Proc of the 2018 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2018: 2095-2108

[48] Gan Shuitao, Zhang Chao, Qin Xiaojun, et al. CollAFL path sensitive fuzzing [C] //Proc of 2018 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2018: 679-696

[49] Lemieux C, Sen K. FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage [C] //Proc of the 33rd ACM/IEEE Int Conf on Automated Software Engineering. New York: ACM, 2018: 475-485

[50] Liu Baozheng, Zhang Chao, Gong Guang, et al. FANS fuzzing android native system services via automated interface analysis [C] //Proc of the 29th USENIX Security Symp. Berkeley, CA: USENIX Association, 2020: 307-323

[51] Fiterau-Brostean P, Jonsson B, Merget R, et al. Analysis of DTLS implementations using protocol state fuzzing [C] //Proc of the 29th USENIX Security Symp. Berkeley, CA: USENIX Association, 2020: 2523-2540

[52] Lee Suyoung, Han Hyungseok, Cha Kilsang, et al. Montage: A neural network language model-guided JavaScript engine fuzzer [C] //Proc of the 29th USENIX Security Symp. Berkeley, CA: USENIX Association, 2020: 2613-2630

[53] Ispoglou K, Austin D, Mohan V, et al. FuzzGen automatic fuzzer generation [C] //Proc of the 29th USENIX Security Symp Berkeley, CA: USENIX Association, 2020: 2271-2287

[54] Lattner C. LibFuzzer [CP/OL]. [2020-12-14]. http://llvm.org/docs/LibFuzzer.html

[55] She Dongdong, Pei Kexin, Epstein D, et al. NEUZZ efficient fuzzing with neural program smoothing [C] //Proc of 2019 IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2019: 803-817

[56] Rebert A, Cha S K, Avgerinos T, et al. Optimizing seed selection for fuzzing [C] //Proc of the 23rd USENIX Security Symp. Berkeley, CA: USENIX Association, 2014: 861-875

[57] You Wei, Wang Xueqiang, Ma Shiqing, et al. ProFuzzer on-the-fly input type probing for better zero-day vulnerability discovery [C] //Proc of 2019 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2019: 769-786

[58] Holler C, Herzig K, Zeller A. Fuzzing with code fragments [C] //Proc of the 21st USENIX Security Symp. Berkeley, CA: USENIX Association, 2012: 445-458

[59] Park S, Xu Wen, Yun I, et al. Fuzzing JavaScript engines with aspect-preserving mutation [C] //Proc of 2020 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2020: 1629-1642

[60] Zong Peiyuan, Lü Tao, Wang Dawei, et al. FuzzGuard filtering out unreachable inputs in directed grey-box fuzzing through deep learning [C] //Proc of the 29th USENIX Security Symp. Berkeley, CA: USENIX Association, 2020: 2255-2269

[61] Wang Yanhao, Jia Xiangkun, Liu Yuwei, et al. Not all coverage measurements are equal: Fuzzing by coverage accounting for Input prioritization [C/OL] //Proc of NDSS 2020. San Diego, CA: University of California, 2020 [2020-11-11]. https://dblp.org/db/conf/ndss/ndss2020.html#WangJLZBWS20

[62] Aschermann C, Schumlo S, Abbasi A, et al. Ijon exploring deep state spaces via fuzzing [C] //Proc of 2020 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2020: 1597-1612

[63] Österlund S, Razavi K, Bos H, et al. Parmesan sanitizer-guided greybox fuzzing [C] //Proc of the 29th USENIX Security Symp. Berkeley, CA: USENIX Association, 2020: 2289-2306

[64] Klees G, Ruef A, Cooper B, et al. Evaluating fuzz testing [C] //Proc of the 2018 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2018: 2123-2138

[65] Saavedra G J, Rodhouse K N, Dunlavy D M, et al. A review of machine learning applications in fuzzing [J]. CoRR preprint, CoRR: abs/190611133, 2019

[66] Zheng Yaowen, Davanian A, Yin Heng, et al. FIRM-AFL: High-throughput greybox fuzzing of iot firmware via augmented process emulation [C] //Proc of the 28th USENIX Security Symp. Berkeley, CA: USENIX Association, 2019: 1099-1114

[67] Zaddach J, Bruno L, Francillon A, et al. AVATAR a framework to support dynamic security analysis of embedded systems' firmwares [C/OL] //Proc of NDSS 2014. San Diego, CA: University of California, 2014 [2020-11-11]. https://dblp.org/db/conf/ndss/ndss2014.html#ZaddachBFB14

[68] Chen D D, Woo M, Brumley D, et al. Towards automated dynamic analysis for linux-based embedded firmware [C/OL] //Proc of NDSS 2016. San Diego, CA: University of California, 2016 [2020-11-11]. https://dblp.org/db/conf/ndss/ndss2016.html#ChenWBE16

[69] Muench M, Stijohann J, Kargl F, et al. What you corrupt is not what you crash challenges in fuzzing embedded devices [C/OL] //Proc of NDSS 2018. San Diego, CA: University of California, 2018 [2020-11-11]. https://dblp.org/db/conf/ndss/ndss2018.html#MuenchSKFB18

[70] Google. syzkaller [CP/OL]. [2020-11-11]. https://github.com/google/syzkaller

[71] Jeong D R, Kim K, Shivakumar B, et al. Razzer finding kernel race bugs through fuzzing [C] //Proc of 2019 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2019: 754-768

[72] Song D, Hetzelt F, Kim J, et al. Agamotto accelerating kernel driver fuzzing with lightweight virtual machine checkpoints [C] //Proc of the 29th USENIX Security Symp. Berkeley, CA: USENIX Association, 2020: 2541-2557

[73] Kim K, Jeong D R, Kim C H, et al. HFL: Hybrid fuzzing on the Linux kernel [C/OL] //Proc of NDSS 2020. San Diego, CA: University of California, 2020 [2020-11-11]. https://dblp.org/db/conf/ndss/ndss2020.html#WangJLZBWS20

[74] Corina J, Machiry A, Salls C, et al. DIFUZE interface aware fuzzing for kernel drivers [C] //Proc of the 2017 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2017: 2123-2138

[75] Oleksenko O, Trach B, Silberstein M, et al. SpecFuzz bringing spectre-type vulnerabilities to the surface [C] //Proc of the 29th USENIX Security Symp. Berkeley, CA: USENIX Association, 2020: 1481-1498

[76] Ruge J, Classen J, Gringoli F, et al. Frankenstein advanced wireless fuzzing to exploit new bluetooth escalation targets [C] //Proc of the 29th USENIX Security Symp. Berkeley, CA: USENIX Association, 2020: 19-36

[77] Peng Hui, Payer M. USBFuzz: A framework for fuzzing USB drivers by device emulation [C] //Proc of the 29th USENIX Security Symp. Berkeley, CA: USENIX Association, 2020: 2559-2575

[78] Xu Meng, Kashyap S, Zhao Hanqing, et al. KRACE data race fuzzing for kernel file systems [C] //Proc of 2020 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2020: 1643-1660

[79] Xu Wen, Moon H, Kashyap S, et al. Fuzzing file systems via two-dimensional input space exploration [C] //Proc of 2019 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2019: 818-834

[80] Jiang Zuming, Bai Jiaju, Lu Kangjie, et al. Fuzzing error handling code using context-sensitive software fault injection [C] //Proc of the 29th USENIX Security Symp. Berkeley, CA: USENIX Association, 2020: 2595-2612

[81] Blair W, Mambretti A, Arshad S, et al. HotFuzz discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing [J]. arXiv preprint, arXiv:200203416, 2020

[82] Schumilo S, Aschermann C, Abbasi A, et al. HYPER-CUBE high-dimensional hypervisor fuzzing [C/OL] //Proc of NDSS 2017. San Diego, CA: University of California, 2017 [2020-11-11]. https://dblp.org/db/conf/ndss/ndss2017.html #0001JKCGB17

[83] Petsios T, Zhao J, Keromytis A D, et al. SlowFuzz automated domain-independent detection of algorithmic complexity vulnerabilities [C] //Proc of the 2017 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2017: 2155-2168

[84] You Wei, Zong Peiyuan, Chen Kai, et al. SemFuzz semantics-based automatic generation of proof-of-concept exploits [C] //Proc of the 2017 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2017: 2139-2154

[85] Xu Wen, Park S, Kim T. FREEDOM engineering a state-of-the-art DOM fuzzer [C] //Proc of the 2020 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2020: 971-986

[86] Güler E, Aschermann C, Abbasi A, et al. AntiFuzz impeding fuzzing audits of binary executables [C] //Proc of the 28th USENIX Security Symposium. Berkeley, CA: USENIX Association, 2019: 1931-1947

[87] Jung Jinho, Hu Hong, Solodukhin D, et al. Fuzzification anti-fuzzing techniques [C] //Proc of the 28th USENIX Security Symp. Berkeley, CA: USENIX, 2019: 1913-1930

[88] Wang Yan, Jia Peng, Liu Luping, et al. A systematic review of fuzzing based on machine learning techniques [J]. CoRR preprint, CoRR: abs/190801262, 2019

Ren Zezhong, born in 1992. Master candidate. His main research interests include fuzzing and machine learning.

任泽众,1992 年生.硕士研究生.主要研究方向为模糊测试与机器学习.

Zheng Han, born in 1998. Master candidate. His main research interest is fuzzing technique.

郑　晗,1998 年生.硕士研究生.主要研究方向为模糊测试技术.

Zhang Jiayuan, born in 1997. Master candidate. Her main research interest is information security.

张嘉元,1997 年生.硕士研究生.主要研究方向为信息安全.

Wang Wenjie, born in 1964. PhD, associate professor. His main research interests include information security and intelligent information processing.

王文杰,1964 年生.博士,副教授.主要研究方向为信息安全与智能信息处理.

Feng Tao, born in 1970. PhD, professor. His main research interests include network and information security.

冯　涛,1970 年生.博士,教授.主要研究方向为网络与信息安全.

Wang He, born in 1987. PhD, lecturer. Her main research interests include cryptography, quantum cryptography and quantum communication protocols.

王　鹤,1987 年生.博士,讲师.主要研究方向为密码学、量子密码学和量子通信协议.

Zhang Yuqing, born in 1966. PhD, professor, PhD supervisor. His main research interest is information security.

张玉清,1966 年生.博士,教授,博士生导师.主要研究方向为信息安全.