

进化内核模糊测试研究综述

侍言^{1,2,3}, 羌卫中^{1,2,3}, 邹德清^{1,2,3}, 金海^{1,4}

- (1. 大数据技术与系统国家地方联合工程研究中心服务计算技术与系统教育部重点实验室, 湖北 武汉 430074;
2. 分布式系统安全湖北省重点实验室, 湖北 武汉 430074;
3. 华中科技大学网络空间安全学院, 湖北 武汉 430074;
4. 华中科技大学计算机科学与技术学院, 湖北 武汉 430074)

摘要: 模糊测试是一种通过生成随机、异常或无效的测试样例来检测软件或系统中潜在漏洞和错误的技术方法。内核作为一种高度复杂的软件系统, 由众多互相关联的模块、子系统和驱动程序所构成, 相比用户态应用程序, 将模糊测试应用于内核面临着代码庞大、接口复杂、运行时不确定等具有挑战性的问题。传统的模糊测试方法生成的输入只能简单地满足接口规范和显式调用依赖, 难以深入探索内核。进化内核模糊测试借助于启发式的进化策略, 在反馈机制的引导下动态地调整测试样例的生成和选择, 从而迭代式地生成质量更高的测试用例。对现有的进化内核模糊测试工作开展研究, 阐述了进化内核模糊测试的概念并总结了进化内核模糊测试的通用框架, 根据反馈机制类型对进化内核模糊测试工作进行分类和对比, 从反馈机制在运行时信息的收集、分析和利用等方面剖析反馈机制引导进化的原理, 对进化内核模糊测试的发展方向进行展望。

关键词: 内核; 模糊测试; 进化; 反馈

中图分类号: TP393

文献标志码: A

DOI: 10.11959/j.issn.2096-109x.2024001

Survey of evolutionary kernel fuzzing

SHI Yan^{1,2,3}, QIANG Weizhong^{1,2,3}, ZOU Deqing^{1,2,3}, JIN Hai^{1,4}

1. National Engineering Research Center for Big Data Technology and System Services Computing Technology and System Lab, Wuhan 430074, China
2. Hubei Key Laboratory of Distributed System Security, Wuhan 430074, China
3. School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China
4. School of Computer Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China

Abstract: Fuzzing is a technique that was used to detect potential vulnerabilities and errors in software or systems by generating random, abnormal, or invalid test cases. When applying fuzzing to the kernel, more complex and challenging obstacles were encountered compared to user-space applications. The kernel, being a highly intricate

收稿日期: 2023-02-23; 修回日期: 2023-08-16

通信作者: 羌卫中, wzqiang@hust.edu.cn

基金项目: 国家自然科学基金(62272181); 国家通用技术基础研究联合基金(U1936211)

Foundation Items: The National Natural Science Foundation of China (62272181), The Joint Funds of the National Natural Science Foundation of China (U1936211)

引用格式: 侍言, 羌卫中, 邹德清, 等. 进化内核模糊测试研究综述[J]. 网络与信息安全学报, 2024, 10(1): 1-21.

Citation Format: SHI Y, QIANG W Z, ZOU D Q, et al. Survey of evolutionary kernel fuzzing[J]. Chinese Journal of Network and Information Security, 2024, 10(1): 1-21.

software system, consists of numerous interconnected modules, subsystems, and device drivers, which presented challenges such as a massive codebase, complex interfaces, and runtime uncertainty. Traditional fuzzing methods could only generate inputs that simply satisfied interface specifications and explicit call dependencies, making it difficult to thoroughly explore the kernel. In contrast, evolutionary kernel fuzzing employed heuristic evolutionary strategies to dynamically adjust the generation and selection of test cases, guided by feedback mechanisms. This iterative process aimed to generate higher-quality test cases. Existing work on evolutionary kernel fuzzing was examined. The concept of evolutionary kernel fuzzing was explained, and its general framework was summarized. The existing work on evolutionary kernel fuzzing was classified and compared based on the type of feedback mechanism utilized. The principles of how feedback mechanisms guided evolution were analyzed from the perspectives of collecting, analyzing, and utilizing runtime information. Additionally, the development direction of evolutionary kernel fuzzing was discussed.

Keywords: kernel, fuzzing, evolutionary, feedback

0 引言

Linux 内核的高度灵活性和可定制性使其成为流行的开源操作系统内核之一。从服务器到嵌入式设备，从超级计算机到智能手机，Linux 内核广泛适用于各种不同的硬件设备和应用场景。Linux 内核在提供了丰富的功能与服务的同时，存在许多漏洞和安全问题。截至 2023 年 2 月，在 CVE 漏洞库中已经有 6 704 条 Linux 相关漏洞的记录。Linux 内核的开源特性和使用的广泛性使它成为内核安全研究的主要研究对象。

在典型的操作系统体系架构中，内核以最高特权级别运行，负责以安全、公平的方式管理和分配资源。内核中的漏洞利用会对整个操作系统产生难以估量的破坏，常见的危害如通过提权攻击使得数据泄露，通过拒绝服务漏洞威胁正常运行的服务。2022 年 3 月发现的 Linux 高危漏洞 DirtyPipe 利用系统调用 splice() 函数底层的零复制机制与管道通信机制，使得非特权用户可以越权向任意文件中写数据。

Linux 内核以社区化的开发模式来满足不断变化的硬件支持、应用场景和安全的需求，快速的发展与更新对 Linux 内核的稳定性和可靠性有着很高的要求。Linux 内核的安全验证面临着代码量庞大、交互逻辑复杂等问题，并且 Linux 内核中许多漏洞只会在运行时暴露，如数据竞争和竞态条件，因此需要自动化的动态测试技术对内核源码进行校验。

内核研究人员通常采用模糊测试来进行内核漏洞挖掘。模糊测试是一种自动化测试技术，通过为内核提供随机生成的输入来检测是否触发了异常或漏洞，TriforceAFL、Syzkaller、kAFL^[1]是内核模糊测试领域的代表性成果。与一般的用户态应用程序不同，内核拥有系统调用、驱动程序等复杂的接口，并且在中断、线程调度等机制的影响下，内核的运行状态是不确定的，因此仅依赖于语法规则生成的输入难以执行到内核的深层代码位置。

进化内核模糊测试是一种基于进化算法的内核模糊测试技术，能够在无人工干预的情况下自适应地调整测试用例的选择和生成，引导测试用例完成迭代式的进化。进化内核模糊测试构建完整的反馈回路，在内核执行时收集诸如代码覆盖率、状态转换等运行时信息，利用信息的反馈评估测试用例的质量，调整语料库的成员与结构，对变异策略进行优化，使之后新生成的输入更具备探索内核复杂执行空间的潜力。

目前学术界和产业界已经产出很多进化内核模糊测试工具，采用代码覆盖率、状态反馈等反馈机制引导进化，适配于内核的各种接口与组件。作为最典型的代码覆盖率引导的进化内核模糊测试器，Syzkaller 在内核的漏洞挖掘中取得了显著成果，从 2015 年开源至今已经发现数千个内核漏洞。

本文对 Linux 内核的进化模糊测试开展研究，首先，从灰盒模糊测试引入，对进化内核

模糊测试的概念进行阐述;然后,总结进化内核模糊测试的通用框架和 workflow,将反馈机制分为代码覆盖率反馈、状态反馈、定向反馈和特殊反馈 4 种类型;接着,按照反馈机制的类型对进化内核模糊测试工作进行分类和对比,分析和评估各种进化策略的设计思路与实际效用;最后,对未来进化内核模糊测试的发展方向进行展望与讨论。

1 相关定义

1.1 根据程序知识依赖程度分类的模糊测试类别

模糊测试器可以从目标程序中提取代码逻辑、分支条件等内部信息来指导测试用例的生成。根据对目标程序知识的依赖程度划分,模糊测试通常可以分为黑盒模糊测试(black box fuzzing)、白盒模糊测试(white box fuzzing)和灰盒模糊测试(grey box fuzzing)^[2]。

黑盒模糊测试不需要了解程序的内部实现,它通过为目标程序提供随机输入来观察程序的响应,以此来判断输入的有效性 with 程序状态。黑盒模糊测试从零开始生成输入,或者依靠静态语料库中的有效输入文件进行变异。这种模糊测试方法适用于输入格式复杂或高度结构化的测试目标,如 JavaScript 等语言编写的程序。同时,它也适用于目标程序很大且目标状态非确定性的场景,如浏览器 DOM 模糊测试器为 Chrome 生成 HTML 格式的输入。Trinity 是针对 Linux 内核系统调用接口的黑盒模糊测试工具,它基于参数格式知识生成更易被内核接受的系统调用。然而,由于 Trinity 没有考虑到内核上下文,因此生成的系统调用输入仍然存在许多问题。

与黑盒模糊测试相反的是白盒模糊测试,白盒模糊测试最初由 Godefroid 等^[3]提出,它从程序中提取充足的信息来辅助测试的进行,通常会结合符号执行、污点分析等技术^[4]。SAGE^[5]是最著名的白盒模糊测试器之一,它通过动态符号执行收集运行时条件语句的约束进行求解,为大规模的 Windows 应用程序生成测试用例,取得了很好的效果。白盒模糊测试考虑目标程序的逻辑约束和内部数据结构,因此更容易到达需要满足复杂约束的程序深层位置。然而,复杂的程序分析需

要耗费大量计算资源,这对模糊测试的效率有所影响。另外,符号执行中约束求解的不精确性会影响输入生成的效果。

灰盒模糊测试介于上述两者之间。灰盒模糊测试获取程序部分信息,结合运行时信息的反馈来引导种子逐步向更优方向变异。基于覆盖率的灰盒模糊测试(CGF, coverage-based grey box fuzzing)^[6]是其中著名且广泛应用的一项技术。通常,它会在程序编译时进行插桩,然后在程序执行时收集插桩代码反馈的覆盖率信息,用以指导种子的选择与变异,使得种子朝着可以使覆盖率增大的方向进化。

定向灰盒模糊测试(DGF, directed grey box fuzzing)在覆盖率引导的灰盒模糊测试基础上引入了对于目标代码位置的引导,AFLGO^[7]是其中的代表性工作。AFLGO 首先根据程序的过程内控制流图和函数调用图(CG, call graph)计算基本块到目标的过程间距离并在编译时将距离插桩,然后在程序运行时将执行过的基本块的距离聚合作为种子距离反馈回模糊测试器,模糊测试器赋予更接近目标的种子更多的能量,以引导模糊测试器不断逼近目标位置。AFLGO 应用了模拟退火算法来实现能量调度策略,防止因为片面追求局部最优而影响求解全局最优。

1.2 内核模糊测试所面临的问题

Linux 内核的模糊测试面临以下几个问题。

① Linux 内核代码规模庞大,更新快速。截止到 2022 年 1 月,Linux 源码已经超过 3 000 万行,收到了上千次的 commit。② Linux 内核涵盖众多的子系统,如设备驱动、文件系统等。每个子系统都有自己的接口和调用方式^[8],参数大多是内核中复杂的数据结构,如文件描述符、多级指针、嵌套数据结构等。③ Linux 内核在运行时需要对各种硬件设备、进程、内存等多种系统资源进行管理和调度,子系统间会产生复杂的交互行为,进而导致 Linux 内核复杂的执行状态。

此外,来自底层调度、同步原语的影响也会导致内核运行的不确定性^[9]。大部分的内核代码只能在特定的状态下(如全局变量被设定为特定的值时)才可以被执行到^[10]。

对于 Linux 内核的模糊测试而言, 黑盒模糊测试因为不考虑内核的内部实现, 在运行时无法得知 Linux 内核的实际运行情况, 所以很难生成内核可用的高质量测试用例。IMF^[11]虽然根据系统调用参数间的输入输出依赖关系建立模型, 通过基于模型的推断生成满足参数显式依赖的系统调用序列, 但是却无法满足系统调用间的隐式依赖^[12]。白盒模糊测试通常利用符号执行或类似的技术来生成输入, 但这对于大型程序在很大程度上是无效的^[13]。对于 Linux 内核这样的大型软件, 应用符号执行来探索有意义的路径是不实际的, 会发生路径爆炸、复杂约束无法求解等问题, 很难产生足够的有效输入。同时, 白盒模糊器盲目追求高代码覆盖的策略并不明智, 可能会浪费大量的时间和精力。

因此, 灰盒模糊测试最适用于 Linux 内核, 现有的进化内核模糊测试工作的框架是依照灰盒模糊测试的方案。在预处理时分析内核源码, 从中获取一定的初始信息, 如系统调用声明、ioctl 接口结构^[14]、控制流与数据流等, 以保证生成的测试用例满足基本的内核规范, 能够让内核接受并执行起来。在内核执行时通过插桩等方式获得内核运行时信息作为反馈, 评估当前测试用例的质量, 指导后续输入的变异生成。如此循环迭代, 从而通过复杂条件约束, 抵达更深层的代码位置, 探索到更多的内核状态。

1.3 基于反馈机制的进化内核模糊测试

1.3.1 进化模糊测试

进化模糊测试是一种受进化算法启发的启发式方法, 通常采用遗传算法的模式, 其中最为关键的部分就是反馈机制。代码覆盖率引导是应用最普遍的一种反馈方式, 以著名的基于覆盖率引导的进化灰盒模糊测试工作 AFL 为例。

AFL 面向用户态应用程序, 支持有源码和无源码两种模式。在有源码的情况下, AFL 可以选择编译阶段的 LLVM 模式插桩或汇编阶段的普通模式插桩, 以获取程序运行时的边覆盖率 (又称分支覆盖率)。边覆盖率如图 1 所示, A 和 B 是两个基本块, A→B 是一条边, 它表示程序从基本块 A 执行到基本块 B。在无源码的情况下, AFL 的 QEMU 模式通过 TCG (tiny code generator) 对

二进制文件插桩。AFL 将源基本块和目的基本块组合成为元组 (tuple), 以位图 (bitmap) 的形式来记录元组信息, 从而统计边覆盖率。位图中同时记录元组的命中数, 当出现新的元组或发生对已有元组的新命中时, AFL 会认为当前测试用例产生了新的覆盖状态, 这样的测试用例被 AFL 看作有趣的。

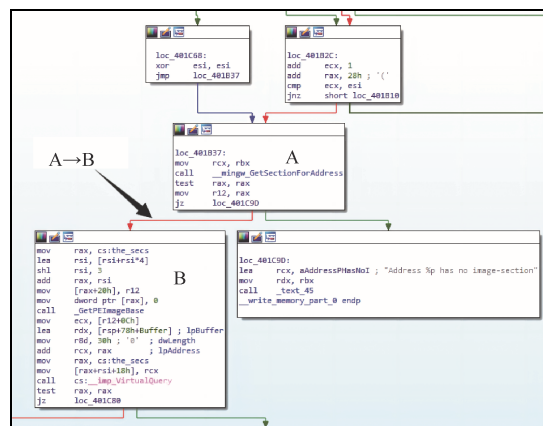


图 1 边覆盖率

Figure 1 Edge coverage

AFL 应用遗传算法的机制来构建反馈回路。遗传算法模拟种群的进化过程, 通过选择、变异等机制, 在每轮迭代中保留部分优良个体, 将它们作为下一轮的种子。适应度函数是遗传算法中用于评价个体的标准。对于 AFL 而言, 适应度函数即是否触发了新的覆盖状态。来自插桩的反馈告知模糊测试器当前轮次输入的运行结果, AFL 将有趣的输入加入种子池中, 然后挑选种子进行变异操作, 生成新一轮的测试输入, 这就是一次完整的反馈回路。AFL 的变异策略分为确定性策略和随机性策略。确定性策略包括位翻转、算数加减、字典替换与插入等, 旨在生成紧凑的测试用例, 只对原输入做较小改动。随机性策略包括算术运算、删除、拼接等。当确定性策略对于发现新路径无增益时, 将会一直执行随机性策略, 从而深入探究纯粹的随机行为。在这种反馈机制的帮助下, AFL 可以以增大覆盖率为目标不断进化, 参考已有的研究工作^[15]给出如图 2 所示的 AFL 框架。

反馈驱动被广泛应用于进化灰盒模糊测试, 不只 AFL 的衍生工具, 如 AFL++^[16]、kAFL、

WinAFL 等, LibFuzzer 和 Honggfuzz 也都是典型的覆盖率引导的进化灰盒模糊测试工具^[17]。LibFuzzer 是一个进程内的、覆盖率引导的、进化的模糊测试引擎, 它的功能是对库函数进行测试。在测试时, LibFuzzer 与被测库链接, 通过特定的入口点将测试用例提供给被测函数, 覆盖率信息则是由 LLVM 内置的 Sanitizer Coverage 插桩提供的。Honggfuzz 支持多种代码覆盖率反馈方式, 总

体可分为基于硬件和基于软件两种。基于硬件的 Honggfuzz 包括 Intel BTS (branch trace store) 和 Intel PT (processor trace), 以及利用 perf 子系统支持的 CPU 所实现的分支计数、指令计数等; 基于软件的 Honggfuzz 如 Sanitizercoverage 插桩、运行时插桩。依托于上述多种覆盖率反馈方式的支持, Honggfuzz 可以对 GNU/Linux、Windows、FreeBSD 等系统进行模糊测试。

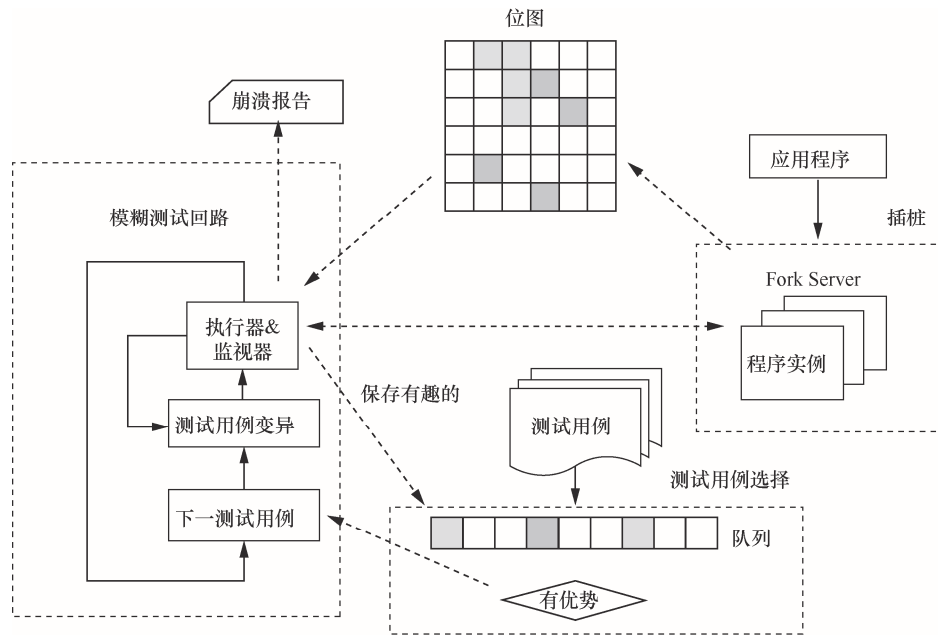


图 2 AFL 框架
Figure 2 The framework of AFL

1.3.2 进化内核模糊测试的反馈机制

代码覆盖率反馈在 Linux 内核中不像在用户态程序中易于应用, 目前仅支持两种代码覆盖率测量方式: GCOV^[18]和 KCOV。GCOV 是 GCC 编译器的覆盖率测量工具, 用于调试、最小化内核配置等。KCOV 专门为模糊测试设计并内置于内核, 通过 debugfs 文件系统的接口导出覆盖率信息。KCOV 在使用时可以指定启用和中止覆盖率收集, 因此可以捕获单个系统调用的覆盖率。Syzkaller 的代码覆盖率引导就是基于 KCOV 实现的。当前内核中需要代码覆盖率信息的模糊测试工具基本是基于 Syzkaller 或 AFL 的, 依赖它们已经实现的反馈机制来开展进一步的工作。

对于模糊测试而言, 衡量有效性最直接的指标就是它所发现的安全问题的数量和严重性。但

是模糊测试器并非每轮运行都可以发现问题, 因此需要一个指标来表示通过模糊测试发现问题的可行性, 代码覆盖率由此被引入。Del Frate^[19]等认为, 在测试中获得的代码覆盖率与软件可靠性之间存在简单的相关性; 但是也有工作认为代码覆盖和故障覆盖之间并不总是有明确的相关性^[20], 代码覆盖只是评估故障覆盖的中等指标^[21]。

在内核模糊测试中应用代码覆盖率反馈可以看作一种近似方案^[22], 覆盖率的提升并不是内核模糊测试的根本目的, 真正的目的是遍历尽可能多的内核运行状态, 以发现内核执行时的问题。但是内核中代码地位并非等同, 有时候只需要能够抵达关键位置即可发现问题, 其他分支是否可以满足条件约束并不是很重要。并且, 仅以覆盖率为导向来调整种子生成是一种比较笼统的方案, 在以覆盖率为导向的策略下, 侧重点被放在

在进化内核模糊测试的流程中, 首先需要对内核源码进行预处理。这个过程包括使用静态分析和符号执行等程序分析方法来获取与内核相关的知识, 如提取接口、变量、约束等; 同时, 需要对内核源码进行插桩, 以便在运行时能够收集代码覆盖和状态覆盖等信息; 另外, 需要设置必要的内核编译选项, 如开启 KCOV 的覆盖率收集和启用内核 sanitizer 等。

内核崩溃会导致整个系统瘫痪, 并且运行在 Ring0 级的内核难以与 Ring3 级的模糊测试器直接通信^[30], 所以内核模糊测试采用虚拟化技术来运行和监控被测内核, 使得内核可以运行在 Ring3 级, 常用技术如 QEMU-KVM。内核模糊测试通过特定的检测工具来监测和分析被测内核运行过程中的行为。KASAN (kernel address sanitizer) 用于监测 Linux 内核中的内存访问问题, 典型漏洞如越界访问 (OOB, out-of-bounds) 和释放后重用 (UAF, use-after-free), 此外还有用于检测未定义行为的 KUBSAN、用于检测内存泄露的 KMemleak 等。

测试用例生成器中的种子池 (语料库) 最开始由初始语料库生成, 初始语料库对于基于语法生成^[31]的模糊测试器不是必需的。测试用例生成器根据选择算法从种子池中挑选种子, 基于能量调度策略指定种子变异生成输入的次数^[32]。常用的变异策略包括位翻转、算术运算、拼接、插入等。对于以系统调用序列为输入的内核模糊测试器, 变异操作通过添加新的系统调用或修改系统调用参数的方式实现。测试用例生成器根据给定的种子生成测试用例, 通常将其构造为用户态可执行程序的形式。

在程序执行时, 用户态的输入程序会调用内核的内部接口, 与此同时被测内核的运行时信息会被反馈给测试用例生成器。这些反馈信息主要用于两个方面: 一方面, 用于评估当前种子的质量, 以判断是否将其加入种子池中; 另一方面, 用于指导种子的进化方向, 通常以指导变异策略的方式。例如, JANUS 根据状态反馈来决定系统调用中非平凡参数的变异生成。

鉴于进化内核模糊测试中反馈机制的重要性, 以反馈机制的类型作为进化内核模糊测试的

分类标准, 主要可以分为代码覆盖率反馈引导的内核模糊测试、状态反馈引导的内核模糊测试、定向内核模糊测试和特殊反馈引导的内核模糊测试 4 种类型。

2 代码覆盖率反馈引导的内核模糊测试

代码覆盖率反馈引导是进化内核模糊测试中最为经典的反馈机制。这类模糊测试器会记录输入的执行路径, 测量代码覆盖率的变化, 并将代码覆盖率的提高作为进化目标, 以探索更多新的路径。常用的代码覆盖率指标包括基本块覆盖率和边覆盖率等。基本块覆盖率是指测试执行过程中覆盖的基本块数占总基本块数的比例, 而边覆盖率是指测试执行过程中覆盖的控制流图中边占总边数的比例。相较于基本块覆盖率, 边覆盖率的优势在于它能够记录测试用例执行时基本块之间的跳转关系, 因此能够更全面地评估测试用例的覆盖情况。代码覆盖率反馈引导的内核模糊测试的代表性工作包括 Syzkaller、TriforceAFL、kAFL, 它们所实现的代码覆盖率反馈引导机制各不相同。

2.1 Syzkaller

Google 安全团队推出的 Syzkaller 是内核模糊测试的标志性工作之一。Syzkaller 是一个代码覆盖率引导的结构感知的进化内核模糊测试工具, 它通过执行系统调用序列来探索操作系统内核, 利用 KCOV 收集代码覆盖信息。被测内核在编译选项中要指定 CONFIG_KCOV=y 选项, 这样内核编译时会在每个基本块中插入插桩语句, 以 PC 值的低 32 位作为基本块标识。在内核执行时, 由 debugfs 子系统将基本块覆盖信息导出到 mmap() 函数创建的共享内存中。

为了有效地生成符合规范的系统调用序列, Syzkaller 定义了一种系统调用描述语言——syzlang。这种语言在描述系统调用时侧重于系统调用参数, 以参数名、类型和类型选项结合的形式来表示参数的长度、值域、方向等信息。为了处理系统调用间的显式依赖, syzlang 定义了资源 (resource) 类型。资源表示需要从一个系统调用的输出传递到另一个系统调用的输入的值, 如系统调用 close() 函数需要在其之前执行的系统调用 open() 函数或 pipe() 函数返回的值作为输入, 从而

建立一种生产者消费者模型,进而定义系统调用间的相对顺序。syzlang 还支持资源继承、系统调用变体、系统调用属性等特殊语法。

对于 Syzkaller 而言,一个测试用例程序是具有参数具体值的系统调用序列,经过转换后的 syzlang 描述被用来生成、变异、执行、最小化、序列化和反序列化程序。Syzkaller 根据系统调用描述、语料库和选择表来生成和修改程序^[33]。选择表的构建基于静态的参数类型分析和动态的系统调用组合出现频率分析,用以指示在程序中已有的系统调用后添加新的系统调用的优先级^[34]。

Syzkaller 从发现新的代码覆盖范围的程序中随机选择一个作为种子,并执行一系列随机变异。Syzkaller 使用多种变异方法,并为各种方法设置概率。变异方法包括翻转字节、插入随机字节、移除字节,以及在程序随机位置插入新的系统调用或已有的系统调用序列。此外, Syzkaller 能够根据内置的系统调用模板更改现有系统调用的参数。

Syzkaller 对产生新代码覆盖的程序进行校验,以确保新的覆盖范围可以可靠地复现。这些程序需要满足以下 3 个条件:① 不受内核的状态特性影响;② 不受执行中的非确定性影响;③ 不受多进程并发与交互影响^[35]。对于通过校验的程序, Syzkaller 对其进行最小化操作,即删除一些系统调用、缩短参数等,同时保持稳定的覆盖范围,然后将最小化的程序添加到种子池中。

2.2 TriforceAFL

TriforceAFL 和 kAFL 都是基于 AFL 的,但是 AFL 本身是一个用户空间的应用程序模糊测试器,不能直接用于内核模糊测试。Nossum 和 Casanovas^[36]认为插桩和共享内存的实现是 AFL 移植到内核的主要问题。他们将修改后的 AFL 应用到 Linux 文件系统驱动程序,启用内核的 GCOV 支持,以 AFL 插桩的方式构建驱动程序,使用驱动程序接口/dev/afl 获取模糊测试期间的代码覆盖反馈,并由此将数据传递给用户空间。模糊测试器运行在目标操作系统中,一旦发生崩溃,模糊测试进程会随之终止。

AFL 插桩的方式并不适用于构建内核的所有部分,并且这种方法需要 AFL 在内核模式下执行或者创建驱动程序接口以将插桩信息传递给用户空间,无法满足测试 Linux 内核的内部功能的需要。TriforceAFL 是 NCC GROUP 推出的 AFL 的补丁版本,它基于 AFL 的 QEMU 模式,扩展了 AFL 的用户态 QEMU 支持,从而可以在 QEMU 的全系统仿真模式下对操作系统内核进行模糊测试。与文献[36]的研究不同, TriforceAFL 不需要用 AFL 插桩方式编译内核,也无须考虑如何对内核核心代码部分插桩。同时, TriforceAFL 不需要在内核编译时开启覆盖支持,因此允许测试不能重新编译或难以插桩的复杂系统,理论上能够测试 QEMU 可模拟的任何操作系统。

TriforceAFL 使用虚拟化技术,在宿主机上运行 QEMU 和 AFL,同时在客户机中启动被测内核。虚拟机(VM)中运行的驱动程序(此处驱动程序特指充当测试用例的解码器并在 VM 内运行的用户空间程序)将序列化的测试用例进行反序列化并执行, QEMU 通过控制流图记录边跟踪信息, AFL 使用 QEMU Trace 的反馈来生成新的测试用例。AFL 将序列化的测试用例视为一般的数据缓冲区,执行由反馈驱动的非结构化通用变异方法。TriforceAFL 捕获的是边信息,而不仅仅是基本块执行信息,因此它继承了 AFL 的反馈和突变引擎的全部优势。

TLSF(triforce Linux syscall fuzzer)是基于 TriforceAFL 的进阶工作,它是特定于 Linux 的系统调用模糊测试器。TriforceAFL 的目标是跳转到有趣的基本块,而 TLSF 则是找到可以导致内核错误的例程。为了支持系统调用的变异, TLSF 添加了新的变异策略,支持文件描述符、缓冲区、进程号等系统调用参数的变异,可以依靠 AFL 的拼接将不同测试用例中的系统调用序列拼接在一起。越界读取漏洞 CVE-2016-4998 和内存损坏漏洞 CVE-2016-4997 都是 TLSF 发现的关键内核漏洞。

2.3 kAFL

与 TriforceAFL 一样, kAFL 也是操作系统(OS)无关的。kAFL 旨在创建一种支持各种内核(包括闭源)的反馈驱动模糊测试框架,提供一种

硬件辅助的代码覆盖率测量方法。kAFL 利用虚拟机监控 (VMM, virtual machine monitor) 程序来收集覆盖率信息, 证明独立于内核的反馈驱动的闭源内核模糊测试的可能性。

kAFL 借助了 Intel CPU 的两种硬件特性: Intel VT-x 和 Intel PT。前者用于硬件辅助虚拟化, 以提高虚拟化的效率; 后者提供了 Ring0 级代码执行的控制流信息, kAFL 利用这些信息来构建一个类似于 AFL 插桩的反馈机制。kAFL 实现的系统整体分为 3 个部分: 模糊测试逻辑组件、VM 和用户态代理。VM 由 Ring3 级组件 QEMU-PT 以及 Ring0 级组件 KVM-QT 组成。QEMU-PT 和 KVM-QT 分别为 QEMU 和 KVM 的修改版本。

模糊测试逻辑组件是 kAFL 的命令和控制组件, 它管理候选输入队列, 创建变异后的测试用例, 并对它们进行评估。模糊测试逻辑组件执行调度和变异的逻辑与 AFL 基本一致, 并且同样使用位图来存储基本块的跳转信息。QEMU-PT 与 KVM-QT 接口交互, 解码 PT 数据以构造和更新位图。

与 AFL 为基本块随机分配编号的方式不同, kAFL 使用基本块的地址来标识基本块。kAFL 维

护全局静态位图, 当模糊测试的一轮迭代结束时, 将位图与全局静态位图比较, 如果发现了一个新位, 则将其添加到全局位图中, 并将触发该新位的输入添加到队列中。当出现新的有趣的输入后, 进入确定性阶段, 对每个字节都进行变异; 在确定性阶段结束之后进入随机性阶段, 以多种方式变异。如果确定性阶段找到了新的输入, 那么随机性阶段将被往后推迟, 直到所有有趣的输入的确定性阶段都已完成。如果一个输入触发了一个全新的基本块跳转, 它将被赋予更高的优先级。

2.4 代码覆盖率反馈引导的其他工作

除了上述 3 种代表性的代码覆盖率反馈引导的进化内核模糊测试工具外, 内核模糊测试领域还有许多应用代码覆盖率反馈引导的工作, 大多基于 Syzkaller 或 AFL, 代码覆盖率引导的模糊测试如表 1 所示。这些工作在设计时制定了各自的进化目标, 借助代码覆盖率反馈引导模糊测试器逐渐趋近探索目标。为了对代码覆盖率这种较为泛化的引导策略进行补足, 这些工作通常结合静态分析、符号执行、动态分析等方法, 或者根据应用场景修改原有工具的代码覆盖率反馈机制,

表 1 代码覆盖率引导的模糊测试
Table 1 Code coverage-guided fuzzing

基本工作	研究工作	研究内容	新发现的漏洞数
Syzkaller	Syzkaller	代码覆盖率引导的结构感知的进化内核模糊测试	—
	HFL	结合符号执行技术的混合模糊测试	24
	KOOBE	Linux 堆越界写漏洞的自动化利用	—
	StateFuzz	Linux 驱动程序的多维度反馈模糊测试	18
	RAZZER	动静态分析结合的数据竞争漏洞挖掘	16
	DR.Fuzz	基于语义通知机制的驱动程序测试	46
	FuzzUSB	USB gadget 栈的模糊测试	34
	FUZE	Linux 内核中 UAF 漏洞可利用性的评估	—
AFL ^[23]	TriforceAFL	基于 QEMU 的全系统仿真的内核模糊测试	—
	kAFL	Intel 硬件辅助的与 OS 无关的模糊测试框架	8
	X-AFL	主动模糊测试和被动模糊测试结合	—
	JANUS	二维输入空间的文件系统模糊测试	62
	HYDRA	可扩展的文件系统模糊测试框架	91
	USBFuzz	USB 驱动程序的模糊测试框架	26
	Unicorefuzz	基于 CPU 模拟的内核模糊测试	—
Both	Agamotto	基于动态虚拟机检查点原语实现模糊测试的性能优化	—

利用覆盖到的代码语句、函数、基本块的具体信息作为反馈。有些工作还会添加其他的反馈机制与代码覆盖率反馈一起引导。本节仅对这些工作的代码覆盖率反馈的相关部分进行研究。

2.4.1 基于 Syzkaller 的工作

基于 Syzkaller 的工作有 HFL、KOOBE^[37]、StateFuzz、RAZZER^[38]、Dr.Fuzz^[39]等。这些工作中应用了 Syzkaller 提供的代码覆盖率反馈的框架，但是在种子挑选、测试用例生成、反馈信息分析等环节进行了修改，以更贴合工作的实际需求。HFL、KOOBE 和 StateFuzz 应用了 Syzkaller 中的代码覆盖率引导的原有实现方式，RAZZER 对系统调用的覆盖信息进行分析与匹配，DR.Fuzz 结合 Intel PT 和 KCOV 实现驱动程序全阶段的覆盖率收集。

HFL 将符号执行引擎 S2E^[40]和 Syzkaller 结合实现了混合模糊测试，以解决间接控制转移、内部系统状态需求、嵌套参数这 3 个挑战。HFL 遵循一个通用的代码覆盖引导的模糊测试方案，以提升代码覆盖率为目标对输入变异。HFL 对内核中所有的代码块插桩，在执行程序时收集覆盖信息，跟踪模糊测试过程中所有覆盖到的代码块，如果执行到未覆盖的基本块，则将输入加入语料库中。

KOOBE 是一项漏洞自动化利用（AEG，automatic exploit generation）领域的工作，研究目标为 Linux 堆越界写漏洞，关注越界漏洞能力的提取。KOOBE 采用了一种能力导向和代码覆盖率引导相结合的模糊测试方案，它的代码覆盖率引导基于 Syzkaller 实现。考虑到覆盖率提升通常比发现新的能力更容易，这样会致使语料库中覆盖率引导的测试用例的占比过高，KOOBE 同时维护两个语料库队列，分别对应覆盖率增大和能力提升两个目标，以相同的概率从中选择种子，从而保证语料库的分布平衡。

StateFuzz 采用三维反馈机制来指导状态探索，3 个维度分别为代码覆盖率、状态变量值域和极值。它复用了 Syzkaller 的代码覆盖率反馈，当测试用例在执行中命中新代码时，就会发出反馈信号，以保存发现了新路径的种子。StateFuzz

设计了一种 3 层语料库，如果测试用例触发了新路径，则将其添加到语料库的第一层。StateFuzz 根据预定义的概率超参数选择语料库的某一层来取用种子，如果是第一层，则从该层随机选择一个种子进行进一步的突变。

RAZZER 的目的是挖掘内核中的数据竞争漏洞，采用了静态分析和动态分析相结合的方案，将模糊测试器引导到内核中潜在的数据竞争点，从而验证是否真的会有数据竞争发生。RAZZER 中的模糊测试分为单线程和多线程两个阶段，两个阶段分别拥有各自的生成器和执行器。单线程生成器以生成或变异的方式生成系统调用序列，称为 P_{st} 。在 P_{st} 执行时，RAZZER 通过 KCOV 收集每个系统调用的执行覆盖，判断是否与静态分析得到的候选竞争对匹配，如果匹配，则对 P_{st} 标注供多线程阶段使用。如果 P_{st} 的运行结果产生了新的覆盖，RAZZER 将保存此 P_{st} 并再将它反馈给单线程生成器。多线程生成器以带注释的 P_{st} 作为输入，输出 P_{st} 的多线程版本 P_{mt} ，同时结合带注释的候选竞争对信息，使用 hypercall 确定性地触发竞争。多线程执行器运行 P_{mt} 以确定是否真的发生数据竞争，并将发生了数据竞争的 P_{mt} 反馈给多线程生成器，在竞争语句后进行系统调用的变异以区分有危害性的数据竞争。

DR.Fuzz 实现了一种语义通知机制，以正确地构建与设备相关的数据结构来通过驱动程序的初始化验证链，从而在无设备的情况下启动驱动程序。语义通知机制还可以用于生成半错误形式的输入。代码覆盖率的提升通常采用深度优先策略，利用格式良好的输入来达到深处路径，而语义通知机制生成的格式错误的输入可以以宽度优先的方式来探索代码路径，进而提升代码覆盖率。由于驱动代码的加载是在内核启动时，而 KCOV 此时还未准备好读取覆盖率信息，因此 DR.Fuzz 使用 Intel PT 来跟踪启动阶段的执行，类似于 KAFL。在使用用户模式代理来测试驱动程序时，为了捕获单个系统调用的精确覆盖范围，覆盖率收集将切换回 KCOV。

FuzzUSB^[41]是针对 USB gadget 栈的模糊测试框架，使用了状态引导。它的代码覆盖率反馈模块沿用 Syzkaller，当 gadget 停驻在一个状态中

时, 模糊测试器在代码覆盖的引导下执行变异。为了提高有效性, 将在预处理阶段由符号执行得到的已求解的约束作为输入源, 帮助模糊测试器处理强分支条件。

应用 Syzkaller 的工作并不全要依赖于代码覆盖率反馈的进化机制, 也有的工作侧重于借助 Syzkaller 较为完善的系统调用生成与变异方法来生成系统调用。FUZE^[42]和 KOUBE 同属于漏洞自动化利用领域, 它是一个评估 Linux 内核中 UAF 漏洞可利用性的框架。FUZE 使用模糊测试为内核错误创建多样的上下文, 利用符号执行来探索不同上下文下 UAF 的可利用性。FUZE 模糊测试模块的目的是识别解引用悬垂指针的系统调用, 然后在原始 PoC 中的悬垂指针出现和解引用间插入这样的系统调用从而改变 PoC 的上下文。为此, FUZE 针对系统调用的生成与执行进行了 Linux 命名空间处理、共享 PoC 初始上下文的参数、检索与发生漏洞的对象相关的内核模块所涉及的系统调用的操作。

2.4.2 基于 AFL 的工作

基于 AFL 的工作诸如 X-AFL^[43]、JANUS、HYDRA^[44]、USBfuzz^[45]、Unicorefuzz^[46], 鉴于 AFL 仅支持用户态模糊测试, 这些工作需要要对 AFL 进行修改以适用于内核模糊测试。修改的重点通常在于仿照 AFL 的插桩机制为模糊测试器提供内核代码覆盖率, 同时会沿用 AFL 较为成熟的变异策略与代码覆盖率反馈机制。

X-AFL 利用被动模糊测试获取目标内核的运行日志和崩溃信息, 进而推断出保留了数据依赖和控制依赖的系统调用模型, 再对系统调用模型进行主动模糊测试, 使用系统调用跟踪来引导。X-AFL 基于 AFL 实现了序列引导模糊测试器 (SGF, sequence guided fuzzer), 为了对内核进行模糊测试, 它实现了 vForkSrv 来模拟 AFL 对程序插桩的功能。vForkSrv 是连通 SGF 以及 Android 设备的中间组件。SGF 基于 KCOV 提供的信息挑选和变异种子, KCOV 信息实质上是运行时调用跟踪 (RCT, runtime call trace)。被动模糊测试阶段从运行日志中收集发生内存错误时的调用跟踪, 即目标调用跟踪 (TCT, target call trace)。在测试用例运行时, 将 RCT 与 TCT 比较, 相似度

越高则对种子分配越多的能量。此外, 一般的代码覆盖率反馈机制也被应用在 X-AFL 中。

JANUS 是探索两维度输入空间的反馈驱动的模糊测试器, 它从文件系统映像和文件操作序列两个输入空间探索。JANUS 的模糊测试引擎在 AFL 的基础上扩展了映像变异器和系统调用模糊测试器。JANUS 以 libOS 作为执行器, 在构建 libOS 时实现了一个封装的 GCC 来选择性地对目标文件系统的源文件插桩, 以获取 AFL 式的运行时覆盖率反馈。运行时路径覆盖被构建为位图, 当发现新路径后, JANUS 将修剪后的映像、序列化程序和推测的映像状态保存为一个二进制输入, 以便后续的执行和变异。

HYDRA 和 JANUS 在输入构造、变异策略和执行环境上有一定的共通之处。HYDRA 将 JANUS 集成在其中, 以利用文件系统映像和系统调用序列两者变异时的协同。HYDRA 的种子由一个文件系统映像和一系列系统调用组成, 输入变异器会使映像和系统调用各自变异或同时变异来生成测试用例。如果产生了新的覆盖率或检查器将测试用例标记为有趣, 则将测试用例保存在种子池中。

USBfuzz 是一个 USB 驱动程序模糊测试的框架, 它的核心是使用软件模拟的 USB 设备向驱动程序提供随机的设备数据。USBfuzz 扩展了 KCOV 的静态插桩, 设计了一个 AFL 式的以驱动程序为中心的边覆盖率收集方案, 允许跨中断上下文的覆盖收集。USBfuzz 从单独的文件中读取数据来响应设备描述符和配置描述符的请求, 对于其他请求, 则从模糊测试器生成的文件中读取数据来响应。

Unicorefuzz 主要基于 AFL-Unicorn 来构建, 它使用了基于 QEMU 的轻量级 CPU 模拟框架 Unicorn。Unicorefuzz 利用 CPU 模拟来测试内核组件, 在对基本块逐个动态翻译时添加插桩代码, 以类似于编译时插桩的方式获取反馈信息。在执行翻译块时, 利用翻译块和基本块的对应关系, 使用程序计数器值作为每个新的基本块的反馈。

2.4.3 其他相关工作

与内核模糊测试中代码覆盖率相关的还有一

些其他工作，这些工作以代码覆盖率反馈引导进化策略的性能和效率为研究重点。

Agamotto^[47]通过动态虚拟机检查点原语来提高内核驱动程序模糊测试器的性能，采用从检查点恢复虚拟机的方式跳过内核崩溃后的重新启动和重复执行的测试用例的耗时部分。Agamotto 基于 QEMU 实现，使用 Syzkaller 对 USB 驱动程序测试。对于 PCI 驱动程序的测试，Agamotto 则利用了 AFL。与原有工作相比，Agamotto 显著提高了模糊测试的吞吐量。

Cho^[48]等基于先前的工作^[49]开发了一套评估系统，对 Syzkaller、Trinity、kAFL 等多种 Linux 内核模糊测试器进行评估。评估结果显示，传统的以更高代码覆盖率为目标的模糊测试器不会更多地执行与系统调用相关的基本块，因此建议内核模糊器在代码覆盖率测量时将系统调用相关的函数和基本块一并纳入考虑，从而更好地评估模糊测试性能或改进代码覆盖率反馈机制。

SNAP^[50]是一个定制化的硬件平台，它实现了硬件原语，以提高覆盖引导模糊测试的性能和精度。SNAP 利用处理器中的微体系结构优化，不仅支持透明化的跟踪，而且可以对执行状态进行更丰富的反馈，提供覆盖率跟踪和丰富的执行语义信息，同时可以做到性能开销几乎为零。

3 状态反馈引导的内核模糊测试

AFL 的设计者 lcamtuf 提到：“尽管状态相关的方法在纸面上是很吸引人的，但是会使得模糊测试器变得复杂脆弱，从而失去易用性和可靠性。以一种通用方法来定性和衡量程序的运行状态是理想情况，但是显然是无法实现的。相较于用户态程序，内核运行时的复杂性更是指数级增长，状态空间过于庞大。尽管难以定性地描述内核状态，但是内核状态是内核模糊测试必需要面对的问题”。

内核状态是操作系统内核在运行过程中所维护和管理的关键数据和信息，用于跟踪系统的各种活动和资源分配。这些状态包括进程的上下文信息、虚拟内存映射、文件描述符表、设备驱动程序状态等，它们共同构成了内核的运行时状态。

在执行系统调用时，内核会根据参数和请求，从一个状态转换到另一个状态。因此，系统调用的顺序与参数和内核状态是高度相关的。针对系统调用接口的内核模糊测试工具致力于生成符合规范与依赖的上下文敏感的系统调用序列，以达到更为复杂且一致的内核状态，相关的工作如 IMF、HFL、Moonshine、X-AFL。这些工作将内核状态作为一个表征系统调用质量的结果性指标，而不是作为运行时提供动态信息的反馈维度。Hao^[10]基于 Syzkaller 和 Dr.Checker^[51]，结合静态分析和人工分析来探索内核模糊测试中的依赖性挑战问题；将全局变量和任何可以从全局变量访问的堆内存（如指向堆对象的全局指针）定义为全局内存，并用全局内存作为内核状态，使得被系统调用改变的内核状态可见，从而更易于分析系统调用的执行路径与内核状态的相关性。

StateFuzz 对 IJON^[52]、AFLNet^[53]、InvsCov^[54]、StateAFL^[55]等探索程序状态的模糊测试工作进行总结，认为程序状态从本质上来说是程序的执行上下文，包括所有程序变量的值和所有内存与寄存器的值。在已有的状态反馈引导的内核模糊测试工作中，状态的表示大多借助筛选出的程序变量，以变量的值来表征状态，以值的变化来表示由某些操作引起的状态转换，进而指示内核的行为，如 JANUS、DR.Fuzz 和 StateFuzz。除此以外，也有其他的状态表示方法，状态反馈引导的模糊测试如表 2 所示。与代码覆盖率反馈引导类似，状态反馈引导的目标是在程序运行中覆盖尽可能多的状态。相比代码覆盖率，状态覆盖率包含更多的程序运行信息，并且状态的转换与数据流或控制流的跳转密切相关。

表 2 状态反馈引导的模糊测试
Table 2 State feedback-guided fuzzing

研究工作	状态表示	状态转换的原因
JANUS	文件对象	文件操作执行
FUZZUSB	I/O 操作	I/O 操作执行
DR.Fuzz	函数返回值	驱动程序初始化出现错误
StateFuzz	状态变量	驱动程序运行
VaultFuzzer	触发 bug 的状态	子系统运行

当前状态反馈主要应用于文件系统和驱动程序的模糊测试，这是因为它们相较内核中

的其他模块更为独立, 与内核的其他部分耦合性相对不高, 状态空间相对较小, 不需要考虑太多内核运行时交互的影响, 因此更容易设计出针对目标程序的状态反馈方案并取得良好效果。

3.1 文件系统

文件操作是在挂载的文件系统上执行的系统调用, 如 `open()`、`access()`、`mkdir()` 等。文件系统映像和文件操作互相存在隐式依赖关系: 文件系统的实时状态决定了一组系统调用可以对哪些文件对象进行操作, 而特定系统调用的执行会更改文件系统内的文件对象。

文件系统的状态可以用文件对象的运行时状态来表示, 因为文件系统的主要功能就是管理文件对象, 而文件操作的目标是文件描述符、文件路径这些文件对象。但是在文件系统运行时对文件对象的状态进行实时监测和提取在实现上过于烦琐。因此, JANUS 根据文件操作的目标对象和执行效果来推断文件对象的变化, 并以推测出的状态作为映像和文件操作这两个输入维度的结合点。

JANUS 在系统调用完成后推断出文件对象的实时状态, 然后使用当前状态作为反馈对系统调用进行生成和变异, 从而生成上下文敏感的文件操作。状态反馈主要用于指导系统调用中非平凡参数的生成。非平凡参数指正确值依赖于文件系统运行时上下文的参数, 这种参数主要与文件对象的状态中的 3 个部分相关, 即文件描述符、路径和扩展属性。当依据状态反馈生成新的系统调用后, 将它添加到原测试用例的末尾, 并根据该系统调用的功能更新状态, 以指示后续的文件操作变异。值得注意的是, JANUS 在系统调用参数变异时不会对目前已有的且对状态可能有影响的参数变异, 以保证状态的稳定性。

3.2 驱动程序

3.2.1 DR.Fuzz

代码覆盖率反馈并不适用于驱动程序的初始化阶段, 因为这个阶段的主要目标是通过验证而不是探索新的路径。在驱动程序中, 对于设备输入和内部函数的运行结果都会进行严格的检查, 当检查到错误时, 会有相应的错误处理流程, 常

用方法是将错误码作为返回值传递给上层调用者。此外, 驱动程序中有些函数的返回值是一个指向设备结构的指针。当这种类型的函数返回一个空值时, 意味着无法获得设备结构, 空值将被返回给调用者。调用者把返回值更改为非零, 并继续将其传递给上层。

驱动程序的初始化遵循统一的错误处理方式, 因此可以以驱动程序中函数的非零返回值作为指示驱动程序的错误状态的标志。DR.Fuzz 采用错误状态作为反馈, 为当前测试用例的执行状态提供充足的反馈信息。首先, 错误状态的反馈可以令模糊测试器得知本次初始化已失败, 从而提前开始下一次的执行。其次, 模糊测试器可以准确地知道是哪些输入导致了初始化失败, 从而指导后续的输入生成。

为了实现错误状态反馈, DR.Fuzz 首先查找并标记所有可能返回错误的驱动程序函数的位置, 然后在标记位置处插入桩代码。桩代码中包含的 `vmcall` 是一种特殊的指令, 它允许虚拟机监控程序与虚拟机之间进行通信。当虚拟机需要执行特殊的操作时, 它会调用 `vmcall`, 并将指令传递给 VMM。VMM 可以通过解释 `vmcall` 的参数来执行所需的操作, 如修改虚拟机的状态或向虚拟机提供服务。在运行时, 当驱动代码进入了错误状态, `vmcall` 会被调用而导致系统退出, 模糊测试器也会收到通知。`vmcall` 中携带的参数可以指示输入是否满足预期值, 状态分析器通过该参数来指导字节优先级突变, 以改进后续输入的变异。错误状态反馈从数据流的角度来指导模糊测试, 与从控制流的角度指导的代码覆盖率反馈相辅相成。

3.2.2 FuzzUSB

USB gadget 驱动程序的生命周期包含 3 个阶段 (配置、枚举和通信), 每个阶段都有自己的输入空间和用于数据传输的输入信道。每个 USB gadget 驱动都在其内部实现了一个细粒度的状态机。在配置和枚举阶段, USB gadget 驱动程序遵循 USB 协议规范通用的信息交换和状态变化; 但是在通信阶段, 不同的 gadget 的状态机各不相同。FuzzUSB 利用有限状态机来显式地表示目标程序的内部状态, 将每个执行了 I/O 操作的代码行作

为一个新的状态。在执行了新的 I/O 操作后, gadget 从先前的状态转换到与所执行的最新 I/O 操作对应的状态。

FuzzUSB 利用静态分析找到驱动程序内部控制状态机的代码位置并为符号执行提供路径的优先级,进而用符号执行提取出表示 gadget 驱动功能和行为的状态机。为了最大化地利用状态反馈引导的好处, FuzzUSB 使用了与 gadget 状态相关的状态覆盖和转换覆盖两种反馈,与代码覆盖中的基本块覆盖和边覆盖类似,状态覆盖的目的是访问到状态机的所有状态,转换覆盖则更关注状态转换的方向。

FuzzUSB 制定了复杂的变异策略以适配代码覆盖、状态覆盖、转换覆盖 3 种反馈引导下的多信道输入变异,定义了转换点间隔、选择下一目标状态等规则。FuzzUSB 在运行时利用提取的状态机和变异规则生成具有状态感知能力的输入,以提高对 USB gadget 栈进行模糊测试的效率和有效性。

3.2.3 StateFuzz

StateFuzz 提出了状态感知的 Linux 驱动程序模糊测试方案,以状态变量值的组合来粗粒度地表示程序状态。状态变量具有 3 个特征:有较长的生命周期;可以由用户更新以产生状态转移;可以影响程序的控制流或内存访问指针。在理想情况下,程序状态应该包含所有状态变量值的组合,但是组合的数目过于庞大,追踪这样的状态显然是不切实际的,因此 StateFuzz 采用了两种优化方案。

在代码覆盖率中,常用的边覆盖率是跟踪两个基本块之间的跳转,是两个相关基本块的组合,StateFuzz 由此得到灵感并以实验佐证,选择使用相关状态变量对来表征状态覆盖率。如果有控制流或内存访问指针同时被两个状态变量直接或间接地影响,那么这两个状态变量是相关的。另外,由于每个状态变量的值空间很大,因此将值空间切分为若干个值域。StateFuzz 在运行时跟踪每个相关变量对中的两个状态变量的值域跳转,如从 $(-1,0]$ 跳转到 $(0,1]$,同时会跟踪状态变量的极值作为补充。

极值反馈的维度是对值域维度的补充,因为

当缺少约束或约束不可解时,状态变量的值域无法被求解出。此外,新的极值的出现代表着程序进入了新的状态。StateFuzz 记录状态变量的上下界,当出现越界的新极值时,发出反馈信号,将该测试用例保留下来。

在预处理阶段,StateFuzz 利用静态分析来识别程序操作,并基于调用图分析来识别由多个程序操作访问的状态变量,然后在源代码的抽象语法树上执行静态符号以推断状态变量的值域。在编译期间,StateFuzz 在目标指针指向状态变量或别名的存储指令后插入跟踪代码,以跟踪值域跳转和极值,产生状态覆盖反馈。

StateFuzz 使用了代码覆盖率、状态变量值域、极值 3 种反馈方式,因此需要额外设计种子保存和种子选择策略。种子保存在 3 层语料库中,如果一个种子触发多种反馈信号,则保存在多层。与状态变量值域反馈相对应的是第二层,它以桶(bucket)的形式组织,桶由已执行的基本块地址的散列值进行索引。种子选择由预设好的超参数决定,语料库的 3 层具有相同的被选择的概率。

3.3 工业界应用

状态反馈的策略在工业界已经实践应用。VaultFuzzer 是赛博堡垒(HardenedVault)公司开发的针对 Linux 内核的基于状态导向的模糊测试工具,旨在解决对特定子系统的深度探索问题,为工控、车联网、云计算等对特定子系统有高稳定性要求的行业服务。为了避免资源浪费,VaultFuzzer 在覆盖率收集上添加了过滤,将目标收缩到子系统、文件和代码块级别。VaultFuzzer 的内核状态模式以 CLANG/LLVM 插桩的方式实现,将会触发 bug 的状态以测试用例的形式记录,并将这些测试用例作为种子,从而在执行中高效地映射代码路径的真实状态,实现状态的导向。

4 定向内核模糊测试

定向模糊测试自从在 AFLGO 中被提出后,逐渐成为模糊测试中一个重要的研究方向,如 Hawkeye^[56]、RDFuzz^[57]、BEACON^[58]都是该方向的工作。定向模糊测试的适应度函数的指标是种子的执行路径与目标位置的逼近程度,一般以控

制流距离或是否触发目标代码作为评估标准^[59]，将更逼近于目标的种子作为下一轮的迭代输入，渐进式地趋向目标位置。定向内核模糊测试以预先设定的目标为依据，模糊测试器在执行过程中专注于探索接近预设目标的路径。与传统的覆盖率导向方法不同，定向内核模糊测试并不简单地追求覆盖率的提升，而是更侧重于测试的针对性，从而能够更有效地发现潜在的内核问题。

以定向反馈引导作为进化策略的内核模糊测试工作有 SemFuzz^[60]、GREBE^[61]、CONZZER、VaultFuzzer，它们根据各自的研究目标指定了不同的目标位置，通过精心设计的反馈机制和变异策略引导模糊测试器逐步执行到目标点。

表 3 定向内核模糊测试
Table 3 Directed kernel fuzzing

研究工作	定向目标	距离指标
SemFuzz	漏洞函数和补丁代码	函数距离和基本块距离
GREBE	内核对象	关键变量相关覆盖率
CONZZER	线程交错	—
VaultFuzzer	目标函数	PC 值的权重

4.1 SemFuzz

SemFuzz 与 KOUBE、FUZE 一样，是漏洞自动化利用领域的工作，它的目的是利用 git 日志、CVE 等与漏洞相关的文本信息自动化地生成 PoC。SemFuzz 从 CVE 和补丁中提取版本、漏洞类型、漏洞函数、关键变量、系统调用等有用的语义信息，根据收集到的系统调用信息生成初始的系统调用序列并将其作为输入。

SemFuzz 的模糊测试流程分为两个阶段，分别应用了粗糙级别变异策略和细粒度变异策略。模糊测试器以初始输入运行，在运行时计算漏洞函数与执行跟踪间的距离，选择最短距离的输入作为新一轮的种子，不断迭代直到能够到达漏洞函数。SemFuzz 使用反向可达性分析来构造反向函数调用图，通过修改的 GCC 在内核编译时获取调用信息，结合运行时 KCOV 收集的执行到的函数信息，归一化地评估输入的优先级，值为 1 表示到达漏洞函数。

在输入到达漏洞函数后，SemFuzz 进入第二阶段，执行细粒度变异。与粗糙级别变异阶段类似，细粒度变异阶段利用控制流图和 KCOV 收集

的基本块覆盖信息，通过计算基本块之间的距离来评估输入的质量，从而到达漏洞函数中的补丁代码位置。SemFuzz 通过监视关键变量所依赖的内核函数的参数来执行细粒度变异，细粒度变异包括系统调用参数变异和重复已有的系统调用。

4.2 GREBE

与 SemFuzz 通过内核模糊测试实现漏洞复现以自动生成 PoC 的目的不同，GREBE 旨在利用定向模糊测试来探索触发漏洞的路径和上下文的多样化，使得 bug 能够显示出各种错误行为，以便探索它的可利用性。GREBE 从内核错误报告中提取与错误相关的结构和对象，以运行时对它们的命中作为反馈引导模糊测试。

GREBE 使用后向污点分析识别与内核崩溃或错误相关的对象，将关键对象的分配、释放、解引用的语句作为定向模糊测试中的目标点。对目标点所在基本块进行插桩，将十六进制最高有效位的值作为魔数从而与其他基本块区分。GREBE 将符合以下两个条件的种子加入语料库中：运行时到达了关键变量处；对关键变量操作语句有影响的系统调用的覆盖率有所提升。第二个条件使得模糊测试器可以累积内核状态。

GREBE 的变异策略借鉴了 Syzkaller 的方法，在 Syzkaller 根据模板生成系统调用的方式上进行了优化。GREBE 按照资源类型将系统调用分组，在组内再分为资源创建和资源使用两个子组。在变异时要么选用同一组中的系统调用替换，要么插入与种子程序中的资源相关联的系统调用。此外，在变异时保留系统调用中除常量、内存指针、校验码和资源外的系统调用参数。通过这样的变异优化，GREBE 在触发不同上下文和执行路径来探索内核错误的多种行为时，可以避免无意义地扩大探索空间，防止偏离关键变量的位置，提高效率 and 有效性。

4.3 CONZZER

CONZZER 是一个挖掘数据竞争漏洞的并发模糊测试框架，它利用并发调用对的反馈，使用定向模糊测试的方法来推断和覆盖新的可能的线程交错。CONZZER 定向的基本思想是：为已经覆盖到的线程交错提供一个合理的方向来推断其他可能的和罕见的线程交错。为了定向地探索线

程交错, CONZZER 使用邻接定向变异和断点控制方法。

邻接定向变异源于这样的洞察: 如果两个函数并发执行, 则与它们邻接执行的函数可能并发执行。以执行过程中覆盖到的并发调用对集合作为输入, 输出新的可能的并发调用对集合。以函数 A 和函数 B 的调用上下文组成的并发调用对为例, 将函数 A 的邻接调用上下文与函数 B 的调用上下文组合, 如果与现有并发调用对不重复, 则将其添加到输出集合中。

断点控制方法包含两个工作逻辑: 确定性断点控制和自适应断点控制。前者旨在覆盖变异得到的每个可能的并发调用对, 后者则将所有可能的调用对一起处理, 以类似于 TCP 拥塞控制的慢启动的方式有效地覆盖并发调用对。

CONZZER 可以与传统的输入驱动的模糊测试器协作, 如 AFL 和 Syzkaller, 程序输入来源于文件或系统调用序列。CONZZER 收集并尝试覆盖更多的并发调用对, 输入驱动的模糊测试器则负责收集代码覆盖率。当覆盖到新的并发调用对或新的路径分支时, 将种子标记为有趣并加入种子池中, 在选择种子时, 优先考虑覆盖到更多并发调用对的种子。

4.4 VaultFuzzer

VaultFuzzer 将内核源码转化为汇编代码和 LLVM IR 代码, 对它们以及整个内核镜像进行静态分析来获得目标函数的地址。程序计数器值的权重由 LLVM IR 的 CFG 得出, 根据 CFG 将不同的权重赋给不同的代码块, 以引导模糊测试的方向。

5 特殊反馈引导的内核模糊测试

特殊反馈机制是为了解决特定问题的需求而设计的, 构建方案时的每个步骤都是完全围绕问题本身来设计的, 因此特殊反馈机制局限于特定的应用场景, 不具备普适性。但得益于特殊反馈机制的定制化, 它往往是直接有效的, 而不像代码覆盖率反馈一样是宽泛近似的。表 4 总结了一些特殊反馈机制。

为了生成符合预期语义的系统调用序列, HFL 使用系统调用序列的有效顺序作为反馈。

HFL 首先对内核进行静态分析获得潜在的依赖对, 然后使用符号执行验证, 以区分真正的依赖对。此外, 它通过跟踪与符号化的系统调用参数相关的依赖值传播来检测参数值依赖性。一旦检索到系统调用序列的有效顺序, HFL 就会为模糊测试器提供反馈, 从而应用于之后的变异。

表 4 特殊反馈机制
Table 4 Specific feedback mechanism

研究工作	特殊反馈机制	目标问题
HFL	系统调用序列的有效顺序	生成符合预期语义的系统调用序列
KOOBE	能力引导	探索 OOB 能力
HYDRA	检查器信号	挖掘不同类型的文件系统错误
KRACE	别名指令对覆盖	探索并发维度
CONZZER	并发指令对覆盖	上下文敏感地探索并发维度

覆盖率引导的模糊测试方案对于探索 OOB 能力收效甚微, 因此 KOOBE 设计了一种能力引导的模糊测试方案: 给定一个 PoC 及其相应的 OOB 能力, 对其进行变异并收集 OOB 被触发时的能力反馈和覆盖反馈, 将带有新能力的种子注入符号跟踪引擎进行能力总结。当执行测试用例时, 收集 OOB 写入集的具体值作为能力反馈。OOB 能力的获取是通过轻量级的动态插桩实现的, 由 S2E 中的 QEMU 支持。

HYDRA 实现了一套可扩展的文件系统模糊测试框架, 为不同类型的文件系统问题设置了不同的检查器, 如一致性检查、POSIX 规范检查、逻辑检查和内存安全检查。HYDRA 的反馈引擎包含两个方面, 除了传统的分支覆盖率外, 还允许每个检查器注册自己的反馈格式, 如以布尔变量的形式指示这个测试用例是否触发了某个错误条件。执行时的反馈会逐渐将模糊测试引导向未探索的代码路径和检查器期望的状态, 从而发现 bug。

KRACE 除了使用分支覆盖率来捕获顺序维度中的代码探索情况外, 专门设计了别名指令对覆盖来捕获并发维度的探索进展。别名指令对是指在不同线程中可能会发生线程交错的内存访问指令对, 在执行过程中, KRACE 跟踪覆盖到的交错位置的数目作为反馈。KRACE 采用延迟注入来实现对内核调度的弱控制, 以别名指定对覆

盖作为反馈来确定是否需要更多的延迟调度。

KRACE 所使用的别名指令对覆盖只描述两个并发执行的指令的位置, 而没有考虑它们的执行上下文。因此, CONZZER 使用上下文敏感的并发调用对覆盖来表征线程交互和运行时的调用上下文, 为模糊测试器提供反馈。并发调用对考虑并发执行函数的调用上下文, 使用函数的运行时调用栈来描述, 因此它需要在程序执行时动态地识别和收集。

6 总结和展望

本文对目前已有的进化内核模糊测试工作开展研究, 阐明了模糊测试中进化的含义, 并总结了进化内核模糊测试的通用流程框架。针对进化内核模糊测试中的反馈机制, 将反馈机制分为代码覆盖率反馈、状态反馈、定向反馈和特殊反馈4种类型, 从反馈机制的设计、实现、效用等方面进行了剖析。按照反馈机制将进化内核模糊测试工作进行分类和对比, 如表5所示。

反馈机制通常包含反馈信息收集、信息分析、信息导向等方面。反馈信息收集通常是以程序分析与插桩反馈结合的方式实现的, 通过程序分析定位到插桩点并插入桩代码, 由桩代码在程序运行时反馈出运行时信息。信息分析一般通过反馈信息来判断当前测试用例是否为目标探索提供了增益, 如覆盖到新的路径、探索到新的状态等。信息导向利用分析得到的结果, 决定是否将当前种子加入种子池, 为其分配能量, 调整变异策略, 引导测试用例的进化方向。

进化内核模糊测试是一种受进化算法启发的启发式方法, 其进化的选择和方向都是由反馈机制来控制的。传统的代码覆盖率反馈虽然泛化, 只以覆盖更多路径作为唯一目标, 但仍是不可替代的, 因为它可以作为探索执行空间的通用评估指标。因此, 基本上进化内核模糊测试器中会引入代码覆盖率反馈的维度。理想的反馈机制应该是细化且具体的, 是针对特定问题、特定目标来定制的。状态反馈和定向反馈虽然遵循框架性的方法论, 但是具体工作并非如此, 尤其是特殊反馈。

由单一反馈机制引导到多反馈机制协同引导

是进化内核模糊测试的一个重点发展方向, 与代码覆盖率反馈结合成为普遍使用的方法。首先, 单一的反馈机制所能够提供的引导能力是有限的, 并且由于技术的固有缺陷, 反馈机制往往无法达到最理想的效果, 因此现有工作都在有意识地添加新的反馈维度来对单一反馈机制进行弥补: StateFuzz 使用极值反馈来补足状态变量值域无法由符号执行求解出的情况, FuzzUSB 使用转换覆盖来补充状态覆盖未关注到的控制流信息。其次, 多维度的输入空间需要多维度的反馈机制来适配, 这样才能够尽可能地遍历到目标程序的所有执行情况。例如, 并发漏洞模糊测试的系统调用序列和线程调度分属顺序维度和并发维度, 驱动程序模糊测试的输入分为上层系统调用输入和下层设备数据输入。因此, 多反馈机制是必要的, 可以提供更全面的引导和探索能力, 使模糊测试更加高效和准确。

当多个反馈机制并用时, 语料库的设计和种子的选择需要额外的考量, 典型工作如 StateFuzz 和 FuzzUSB。在选择种子时, 如果反馈机制的地位等同, 则一般是设置相同的选择概率; 如果某个维度的反馈更加重要, 则会优先考虑该维度的反馈, 并且在运行中得到该维度的反馈时会提高当前种子的优先级。种子的选择在多反馈机制协同的进化内核模糊测试中目前并没有得到深入且专门的研究, 但它确实是影响内核模糊测试效率的重要因素。在单维度的代码覆盖率反馈引导中, SYZVEGAS 利用多臂老虎机算法来动态优化 Syzkaller 的两个关键决策点(任务选择和种子选择), 从而提高了覆盖率并且降低了开销, 这为多维度反馈机制的进化决策的优化提供了参考。

目前的反馈机制在各自的维度独立地完成反馈回路, 各个反馈维度之间没有交互。然而, 如果各个反馈维度或输入维度之间能够产生互相的反馈, 共同指导进化方向, 将进一步提高模糊测试的有效性。JANUS 提供了二维输入空间互相反馈的示范, 文件系统映像和文件操作两个输入维度在状态反馈机制的协同下互通信息, 使得变异策略更加理想化和智能化。

除了多维度反馈协同引导进化外, 还有其他

表 5 进化内核模糊测试工作
Table 5 Evolutionary kernel fuzzing work

研究工作	测试用例生成	变异策略	反馈机制	反馈机制实现方式
Syzkaller	系统调用序列的生成与变异	插入、删除系统调用/系统调用参数变异	代码覆盖率反馈	KCOV
TriforceAFL	文件中读取+变异	AFL 变异策略	代码覆盖率反馈	QEMU Trace
TLSF	系统调用序列变异	系统调用参数变异/系统调用序列拼接	代码覆盖率反馈	QEMU Trace
kAFL	Linux 下以 ext4 映像作为初始输入+变异	AFL 变异策略	代码覆盖率反馈	Intel PT
HFL	系统调用序列的生成与变异+S2E 符号执行	只变异系统调用的入口点	代码覆盖率反馈 系统调用序列有效顺序反馈	KCOV SVF+S2E
KOOBE	Linux 堆越界写 PoC+变异双队列语料库, 概率选择	插入、删除系统调用/系统调用参数变异	代码覆盖率反馈 能力引导	KCOV S2E 的 QEMU 插件
StateFuzz	3 层语料库, 概率选择	插入、删除系统调用/系统调用参数变异	代码覆盖率反馈 状态变量值域反馈 极值反馈	KCOV LLVM Sancov+SVF
RAZZER	单线程: 系统调用序列的生成与变异多线程: $P_{st} > P_{ml}/P_{mt}$ 变异	插入、删除系统调用/系统调用参数变异	代码覆盖率反馈	KCOV
DR.Fuzz	语义通知机制	字节优先级变异	代码覆盖率反馈 错误状态反馈	KCOV+Intel PT LLVM
FuzzUSB	状态机+变异规则生成	多信道/多反馈变异策略	代码覆盖率反馈 状态覆盖反馈 转换覆盖反馈	KCOV dg llvm slicer+KLEE
X-AFL	被动: hook 系统调用并改变参数 主动: 模型生成	系统调用参数变异	代码覆盖率反馈 序列覆盖反馈	KCOV
JANUS	文件系统映像的元数据变异 系统调用序列的生成与变异	映像: 元数据变异 文件操作: 参数变异/末端插入新系统调用	代码覆盖率反馈 状态反馈	GCC wrapper 文件对象状态推断
HYDRA	文件系统映像的元数据变异 系统调用序列的生成与变异	映像: 元数据变异 文件操作: 参数变异/末端插入新系统调用	代码覆盖率反馈 检查器定义的信号	GCC wrapper 检查器插件
USBFuzz	设备/配置描述符作为种子+随机数据生成	AFL 变异策略	代码覆盖率反馈	KCOV
Unicorefuzz	人工设置	AFL 变异策略	代码覆盖率反馈	Unicorn
VaultFuzzer	系统调用序列的生成与变异	插入、删除系统调用/系统调用参数变异	代码覆盖率反馈 状态反馈 PC 权重定向	KCOV CLANG/LLVM
SemFuzz	提取 CVE/git 日志生成系统调用序列+变异	粗糙级别/细粒度变异	漏洞函数和补丁代码定向	修改的 GCC+KCOV
GREBE	内核错误报告的 PoC+变异	Syzkaller 策略+系统调用分组/参数值保留	内核对象定向	GCC plugin
Conzzer	并发调用对+程序输入	邻接定向变异+断点控制方法	并发调用对覆盖率 线程交错定向	CLANG/LLVM
KRACE	单个系统调用生成与变异+线程调度生成	插入、删除系统调用/系统调用参数变异/主次列表洗牌	代码覆盖率反馈 别名指令对覆盖率	LLVM

具有实践意义和价值的方法。DR.Fuzz 中特意生成半错误格式的输入来以宽度优先策略探索代码路径, 在生成输入时将深度优先与宽度优先相结合。SemFuzz 分阶段进行定向探索, 设置了不同的定向目标与变异策略, 使得定向反馈引导更具针对性。这些方法为进化内核模糊测试的发展提供了更多可能的思路。

7 结束语

进化内核模糊测试是内核漏洞挖掘最为有效的方法, 它使用精心设计的进化策略, 在反馈机制的引导下为内核生成迭代式进化的输入, 以探索内核的执行情况, 自动高效地挖掘漏洞。本文着眼于现有的进化内核模糊测试工作, 以反馈机

制类型为分类依据, 重点研究了现有进化内核模糊测试工作中反馈机制的实现机理, 并对基于反馈机制的进化内核模糊测试的发展方向进行了展望。

参考文献:

- [1] SCHUMILO S, ASCHERMANN C, GAWLIK R, et al. kAFL: hardware-assisted feedback fuzzing for OS kernels[C]//26th USENIX Security Symposium (USENIX Security 17). 2017: 167-182.
- [2] 任泽众, 郑晗, 张嘉元等. 模糊测试技术综述[J]. 计算机研究与发展, 2021, 58(5): 944-963.
REN Z Z, ZHENG H, ZHANG JY, et al. A review of fuzzing techniques [J]. Journal of Computer Research and Development, 2021, 58(5): 944-963.
- [3] GODEFROID P, LEVIN M Y, MOLNAR D A. Automated whitebox fuzz testing[C]//Proceedings of NDSS. 2008: 151-166.
- [4] LIANG H, PEI X, JIA X, et al. Fuzzing: state of the art[J]. IEEE Transactions on Reliability, 2018, 67(3): 1199-1218.
- [5] FUZZING I W. SAGE: whitebox fuzzing for security testing[J]. SAGE, 2012, 10(1).
- [6] PHAM V T, BÖHME M, SANTOSA A E, et al. Smart greybox fuzzing[J]. IEEE Transactions on Software Engineering, 2019, 47(9): 1980-1997.
- [7] BÖHME M, PHAM V T, NGUYEN M D, et al. Directed greybox fuzzing[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017: 2329-2344.
- [8] 李贺, 张超, 杨鑫, 等. 操作系统内核模糊测试技术综述[J]. 小型微型计算机系统, 2019, 40(9): 6.
LI H, ZHANG C, YANG X, et al. Survey of OS Kernel fuzzing[J]. Journal of Chinese Computer Systems. 2019, 40(9): 6.
- [9] PRAKASH A, VENKATARAMANI E, YIN H, et al. Manipulating semantic values in kernel data structures: attack assessments and implications[C]//Proceedings of 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 2013: 1-12.
- [10] HAO Y, ZHANG H, LI G, et al. Demystifying the dependency challenge in kernel fuzzing[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 659-671.
- [11] HAN H S, CHA S K. IMF: inferred model-based fuzzer[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017: 2345-2358.
- [12] PAILOOR S, ADAY A, JANA S. MoonShine: optimizing OS fuzzer seed selection with trace distillation[C]//Proceedings of 27th USENIX Security Symposium (USENIX Security 18). 2018: 729-743.
- [13] LI Y, JI S, LYU C, et al. V-fuzz: vulnerability-oriented evolutionary fuzzing[J]. arXiv preprint arXiv:1901.01142, 2019.
- [14] CORINA J, MACHIRY A, SALLS C, et al. Difuze: Interface aware fuzzing for kernel drivers[C]//Proceedings of 24th ACM SIGSAC Conference on Computer and Communications Security(CCS 2017). 2017: 2123-2138.
- [15] WANG Y, JIA X, LIU Y, et al. Not all coverage measurements are equal: fuzzing by coverage accounting for input prioritization[C]//Proceedings of NDSS. 2020.
- [16] FIORALDI A, MAIER D, EIBFELDT H, et al. AFL++: combining incremental steps of fuzzing research[C]//Proceedings of 14th USENIX Workshop on Offensive Technologies (WOOT 20). 2020.
- [17] PRAMANIK A, TAYADE A. Study and comparison of general purpose fuzzers[R]. University of Wisconsin-Madison, 2017.
- [18] HINDS N, LARSON P, FRANKE H, et al. Using code coverage tools in the Linux kernel[C]//ACM SIGSOFT Software Engineering Notes (ACM' 04). 2014: 70-81.
- [19] DEL FRATE F, GARG P, MATHUR A P, et al. On the correlation between code coverage and software reliability[C]//Proceedings of Sixth International Symposium on Software Reliability Engineering(ISSRE'95). 1995: 124-132.
- [20] BRIAND L, PFAHL D. Using simulation for assessing the real impact of test coverage on defect coverage[C]//Proceedings 10th International Symposium on Software Reliability Engineering. 1999: 148-157.
- [21] CAI X, LYU M R. The effect of code coverage on fault detection under different testing profiles[C]//Proceedings of the 1st International Workshop on Advances in Model-based Testing. 2005: 1-7.
- [22] LI J, ZHAO B, ZHANG C. Fuzzing: a survey[J]. Cybersecurity, 2018, 1(1): 1-13.
- [23] SHU G, HSU Y, LEE D. Detecting communication protocol security flaws by formal fuzz testing and machine learning[C]//Proceedings of 28th IFIP WG 6.1 International Conference Tokyo on Formal Techniques for Networked and Distributed Systems - FORTE 2008. 2008: 299-304.
- [24] KIM K, JEONG D R, KIM C H, et al. HFL: hybrid fuzzing on the linux kernel[C]//Proceedings of NDSS. 2020.
- [25] XU W, MOON H, KASHYAP S, et al. Fuzzing file systems via two-dimensional input space exploration[C]//Proceedings of 2019 IEEE Symposium on Security and Privacy (SP). 2019: 818-834.
- [26] KIM K, KIM T, WARRAICH E, et al. FuzzUSB: hybrid stateful fuzzing of usb gadget stacks[C]//Proceedings of 2022 IEEE Symposium on Security and Privacy (SP). 2022: 2212-2229.
- [27] ZHAO B, LI Z, QIN S, et al. StateFuzz: system call-based state-aware Linux driver fuzzing[C]//Proceedings of 31st USENIX Security Symposium (USENIX Security 22). 2022: 3273-3289.
- [28] XU M, KASHYAP S, ZHAO H, et al. KRACE: data race fuzzing for kernel file systems[C]//Proceedings of 2020 IEEE Symposium on Security and Privacy (SP). 2020: 1643-1660.
- [29] JIANG Z M, BAI J J, LU K, et al. Context-sensitive and directional concurrency fuzzing for data-race detection[C]//Proceedings of Network and Distributed Systems Security (NDSS) Symposium. 2022.

- [30] TAN Z, LU H. A systemic review of kernel fuzzing[C]//Proceedings of the 2020 International Conference on Cyberspace Innovation of Advanced Technologies. 2020: 283-289.
- [31] OEHLERT P. Violating assumptions with fuzzing[J]. IEEE Security & Privacy, 2005, 3(2): 58-62.
- [32] ZHANG K, XIAO X, ZHU X, et al. Path transitions tell more: optimizing fuzzing schedules via runtime program states[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 1658-1668.
- [33] SUN H, SHEN Y, WANG C, et al. Healer: relation learning guided kernel fuzzing[C]//Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 2021: 344-358.
- [34] 杨鑫, 张超, 李贺, 等. 基于系统调用依赖的 Linux 内核模糊测试技术研究[J]. 网络安全技术与应用, 2019 (11): 13-16.
- YANG X, ZHANG C, LI H, et al. Research on Linux kernel fuzzing technology based on system call dependency[J]. Network Security Technology & Application, 2019 (11): 13-16.
- [35] WANG D, ZHANG Z, ZHANG H, et al. SyzVegas: beating kernel fuzzing odds with reinforcement learning[C]//Proceedings of 30th USENIX Security Symposium (USENIX Security 21). 2021: 2741-2758.
- [36] NOSSUM V, CASASNOVAS Q. Filesystem fuzzing with american fuzzy lop[C]//Proceedings of Vault Linux Storage and File Systems Conference. 2016.
- [37] CHEN W, ZOU X, LI G, et al. KOUBE: towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities[C]//Proceedings of 29th USENIX Security Symposium (USENIX Security 20). 2020: 1093-1110.
- [38] JEONG D R, KIM K, SHIVAKUMAR B, et al. Razzer: finding kernel race bugs through fuzzing[C]//Proceedings of 2019 IEEE Symposium on Security and Privacy (SP). 2019: 754-768.
- [39] ZHAO W, LU K, WU Q, et al. Semantic-informed driver fuzzing without both the hardware devices and the emulators[C]//Proceedings of Network and Distributed Systems Security (NDSS) Symposium. 2022.
- [40] CHIPOUNOV V, KUZNETSOV V, CANDEA G. S2E: a platform for in-vivo multi-path analysis of software systems[J]. ACM Sigplan Notices, 2011, 46(3): 265-278.
- [41] KIM K, KIM T, WARRAICH E, et al. Fuzzusb: hybrid stateful fuzzing of usb gadget stacks[C]//Proceedings of 2022 IEEE Symposium on Security and Privacy (SP). 2022: 2212-2229.
- [42] WU W, CHEN Y, XU J, et al. FUZE: towards facilitating exploit generation for kernel use-after-free vulnerabilities[C]//Proceedings of 27th USENIX Security Symposium. 2018: 781-797.
- [43] LIANG H, CHEN Y, XIE Z, et al. X-afl: a kernel fuzzer combining passive and active fuzzing[C]//Proceedings of the 13th European workshop on Systems Security. 2020: 13-18.
- [44] KIM S, XU M, KASHYAP S, et al. Finding semantic bugs in file systems with an extensible fuzzing framework[C]//Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019: 147-161.
- [45] PENG H, PAYER M. USBFuzz: a framework for fuzzing USB drivers by device emulation[C]//Proceedings of 29th USENIX Security Symposium (USENIX Security 20). 2020: 2559-2575.
- [46] MAIER D, RADTKE B, HARREN B. Unicorefuzz: on the viability of emulation for kernel space fuzzing[C]//Proceedings of 13th USENIX Workshop on Offensive Technologies (WOOT 19). 2019.
- [47] SONG D, HETZELT F, KIM J, et al. Agamoto: accelerating kernel driver fuzzing with lightweight virtual machine checkpoints[C]//Proceedings of 29th USENIX Security Symposium. 2020: 2541-2557.
- [48] CHO M, JIN H, AN D, et al. Evaluating code coverage for kernel fuzzers via function call graph[J]. IEEE Access, 2021, 9: 157267-157277.
- [49] KIM S, JEONG S, CHO M, et al. Poster: evaluating code coverage for system call fuzzers[C]//Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019: 2689-2691.
- [50] DING R, KIM Y, SANG F, et al. Hardware support to improve fuzzing performance and precision[C]//Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. 2021: 2214-2228.
- [51] MACHIRY A, SPENSKY C, CORINA J, et al. DR. CHECKER: a soundy analysis for linux kernel drivers[C]//26th USENIX Security Symposium (USENIX Security 17). 2017: 1007-1024.
- [52] ASCHERMANN C, SCHUMILO S, ABBASI A, et al. Ijon: exploring deep state spaces via fuzzing[C]//2020 IEEE Symposium on Security and Privacy (SP). 2020: 1597-1612.
- [53] PHAM V T, BÖHME M, ROYCHOUDHURY A. AFLNet: a grey-box fuzzer for network protocols[C]//Proceedings of 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). 2020: 460-465.
- [54] FIORALDI A. Program state abstraction for feedback-driven fuzz testing using likely invariants[J]. arXiv preprint arXiv:2012.11182, 2020.
- [55] NATELLA R. Stateafl: greybox fuzzing for stateful network servers[J]. Empirical Software Engineering, 2022, 27(7): 191.
- [56] CHEN H, XUE Y, LI Y, et al. Hawkeye: towards a desired directed grey-box fuzzer[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018: 2095-2108.
- [57] YE J, LI R, ZHANG B. RDFuzz: accelerating directed fuzzing with intertwined schedule and optimized mutation[J]. Mathematical Problems in Engineering, 2020, 2020: 1-12.
- [58] HUANG H, GUO Y, SHI Q, et al. Beacon: directed grey-box fuzz-

ing with provable path pruning[C]//Proceedings of 2022 IEEE Symposium on Security and Privacy (SP). 2022: 36-50.

- [59] 杨克, 贺也平, 马恒太, 等. 有效覆盖引导的定向灰盒模糊测试[J]. 软件学报, 2021, 33(11): 3967-3982.

YANG K, HE Y P, MA H T, et al. Guiding irecteddd grey-box fuzzing by target-oriented valid coverage[J]. Journal of Software, 2022, 33(11): 3967-3982.

- [60] YOU W, ZONG P, CHEN K, et al. SemFuzz: semantics-based automatic generation of proof-of-concept exploits[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017: 2139-2154.

- [61] LIN Z, CHEN Y, WU Y, et al. GREBE: Unveiling exploitation potential for Linux kernel bugs[C]//2022 IEEE Symposium on Security and Privacy (SP). 2022: 2078-2095.

[作者简介]



侍言（1998-），男，江苏宿迁人，华中科技大学硕士生，主要研究方向为内核模糊测试。



王卫中（1977-），男，江苏南通人，博士，华中科技大学教授、博士生导师，主要研究方向为系统安全及软件安全。



邹德清（1975-），男，湖南湘潭人，华中科技大学教授、博士生导师，主要研究方向为云计算安全、网络攻防与漏洞检测、软件定义安全与主动防御。



金海（1966-），男，上海人，华中科技大学教授、博士生导师，主要研究方向为计算机系统结构、虚拟化技术、集群计算。