基于种子智能生成的内核模糊测试模型

王明义[1], 甘水滔[2,3], 王晓锋[1,4], 刘 渊[1]

[1] 江南大学 人工智能与计算机学院 无锡 中国 214122
[2] 清华大学 网络研究院 北京 中国 100084
[3] 数学工程与先进计算国家重点实验室 无锡 中国 214083
[4] 鹏城实验室 深圳 中国 518005

摘要 操作系统具有庞大的用户群体, 因此使得内核漏洞具有极强的通用性。模糊测试作为一种高效的漏洞挖掘方法, 也被应用于操作系统内核, 并且已经取得不错的成果。但是, 目前流行的面向内核的模糊测试模型 Syzkaller 在生成种子时具有一定的盲目性, 无法自动产生具有依赖关系的系统调用, 制约了模糊测试的代码覆盖能力。为解决上述问题, 本文提出并实现了基于种子智能生成的内核模糊测试模型 SyzMix。该模型一方面结合 LSTM(Long Short-Term Memory)神经网络, 使用语法模板, 通过序列化操作和反序列化操作, 能自动生成更多蕴含潜在依赖关系的系统调用序列, 有效提高了种子执行的成功率; 另一方面, 通过静态分析方法获得系统调用显式依赖关系, 通过动态分析方法获得系统调用隐式依赖关系, 并通过上述依赖关系进一步优化种子内部系统调用关系, 结合测试用例的生成策略和变异策略, 显著提高了选择系统调用的准确性。综合上述方法, SyzMix 达到了更高的代码覆盖能力和代码覆盖加速比。为了验证模型的有效性和实用性, 利用 SyzMix 与 Syzkaller 在不同版本的内核中进行测试, 种子执行成功率提高了 16%, 选择系统调用的准确性提高了 88.8%, 内核代码覆盖率提高了 7.87%, 代码覆盖加速比达到了 132.3%。另外, SyzMix 在不同版本的内核中发现了 8 个的未知 bug, 并申请得到 CVE 编号 CVE-2021-45868。

关键词 模糊测试; 漏洞挖掘; 操作系统内核; 神经网络
中图法分类号 TP311 DOI 号 10.19363/J.cnki.cn10-1380/tn.2024.05.09

Kernel Fuzzing Model Base on Intelligent Seed Generation

WANG Mingyi[1], GAN Shuitao[2,3], WANG Xiaofeng[1,4], LIU Yuan[1]

[1] School of Artificial Intelligence and Computer Science, Jiangnan University, Wuxi 214122, China
[2] Institute for Network, Tsinghua University, Beijing 100084, China
[3] State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi 214083, China
[4] Peng Cheng Laboratory, Shenzhen 518005, China

Abstract Operating system has a large user base, which makes the kernel vulnerability very versatile. As an efficient vulnerability mining method, fuzzing has also been applied to the operating system kernel, and has achieved good results. However, Syzkaller, the popular kernel oriented fuzzing model, has some blindness in generating seeds, and cannot automatically generate system calls with dependencies, which will restrict the code coverage ability of fuzzing. To solve these problems, this paper proposes and implements a kernel fuzzing model SyzMix based on seed intelligent generation. On the one hand, the model combines with LSTM (long short term memory) neural network, uses syntax template, automatically generates more system call sequences with potential dependencies through serialization and deserialization, and effectively improves the success rate of seed execution; on the other hand, the explicit dependencies of system calls are obtained by static analysis method, and the implicit dependencies of system calls are obtained by dynamic analysis method, and further optimize the relationship of system calls within the seed through the above dependencies. Combined with the generation strategy and mutation strategy of test cases, the accuracy of selecting system calls is significantly improved. Based on the above methods, SyzMix achieves higher code coverage and the code coverage speed-up. To verify the validity and practicability of the model, SyzMix and Syzkaller are tested in different versions of the kernel. The success rate of seed execution is increased by 16%, the accuracy of selecting system calls is improved by 88.8%, the kernel code coverage is increased by 7.87%, and code coverage achieved a speed-up of 132.3%. In addition, SyzMix found eight unknown bugs in different versions of the kernel and requested CVE number CVE-2021-45868.

Key words fuzzing; vulnerability discovery; operating system kernel; neural network

通讯作者: 王晓锋, 博士, 教授, Email: wangxf@jiangnan.edu.cn。
本课题得到鹏城实验室重大任务项目(No. PCL2022A03), 国家自然科学基金项目(No. 62172191, No. 61972182)资助。
收稿日期: 2022-06-17; 修改日期: 2022-09-28; 定稿日期: 2024-01-11

1 介绍

Linux 基金会发布的 Linux 内核历史报告[1]显示, 截至 5.8 版本, Linux 的源代码已经由最初的 10239 行增加至 28442673 行, 由只支持单一的 i386 架构发展到可以支持超过 30 种不同的架构。由于 Linux 系统的开源特性以及强大的功能, 越来越多的厂商和用户不断为 Linux 内核提供源码。图 1 展示了从 2002—2019 年 Linux 内核源码提交数量以及为 Linux 内核贡献源码的团队数量, 图 2 展示了知名的厂商对 Linux 内核源码的贡献。

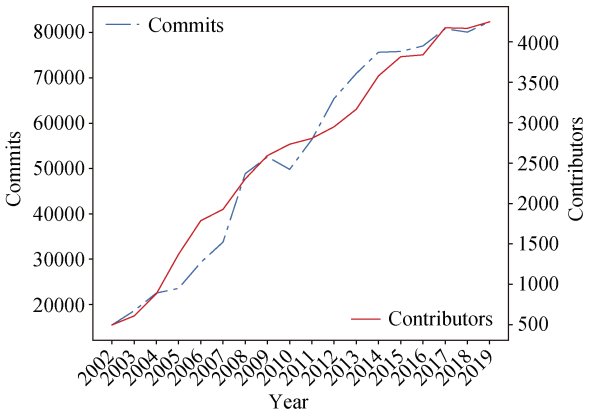

图 1 Linux 内核源码提交情况

Figure 1 Linux kernel source submission

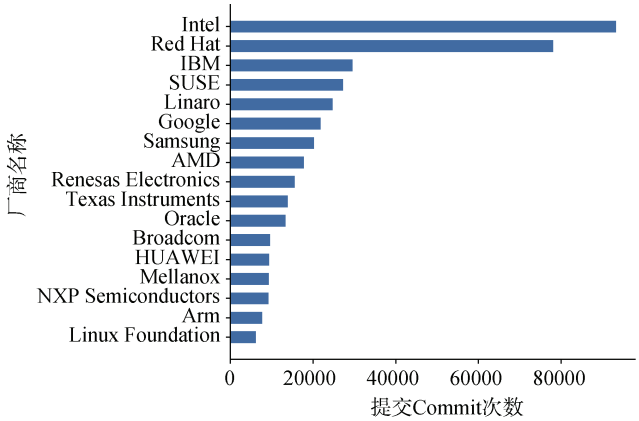

图 2 知名厂商对 Linux 源码的贡献情况

Figure 2 Contribution of well-known manufacturers to Linux source code

传统的内核开发人员需要通过手写测试套件的方法[2-3]消除内核漏洞, 但是随着代码规模的不断增加, 对 Linux 内核进行测试的难度也不断提高。

模糊测试(fuzzing)是一种通过构造非预期的输入数据并监视目标在运行过程中的异常结果来发现软件故障的方法[4]。但是, 一些经典的模糊测试工具(如 AFL[5])并不能直接对内核进行测试, 需要解决如下几个问题[6]: (1)当前的 Linux 内核代码量过于庞大, 并且不同组件之间有复杂的依赖关系; (2)测试用例输入时, Linux 内核需要通过系统调用接口处理验证; (3)Linux 内核中包含大量的状态空间, 一个单独的测试用例很难复现当时系统的状态。

当前流行的面向内核的模糊测试工具 Syzkaller[7]成功解决了这些问题。Syzkaller 使用自定义的语法模板 syzlang 生成对应的种子文件, 监控测试用例运行时的覆盖情况, 采集可以触发新路径的测试用例, 并调用不同的策略生成新的测试用例[8]。Syzkaller 组合使用了变异策略和生成策略, 但是在生成种子时具有一定的盲目性, 所生成的种子执行成功率不高。另外, Syzkaller 进行模糊测试的过程中, 不能高效选择具有依赖关系的系统调用。

针对上述问题, 本文提出一种基于种子智能生成的内核模糊测试模型 SyzMix。具体而言, 针对生成种子具有盲目性这一问题, SyzMix 通过对已有的种子进行分析, 结合 LSTM(Long Short-Term Memory)神经网络, 通过序列化和反序列化操作生成新的种子, 提高了种子执行的成功率。针对选择系统调用准确性不高的问题, SyzMix 创建并维护系统调用依赖关系表, 用来更准确地获得具有依赖关系的系统调用, 从而进一步对种子内部的系统调用关系进行优化。本文挑选了不同版本的 Linux 内核作为测试对象, 与 Syzkaller 进行对比, 种子执行成功率提高了 16%, 选择系统调用的准确性提高了 88.8%, 内核代码覆盖率提高了 7.87%, 代码覆盖加速比达到了 132.3%。另外, SyzMix 在不同内核中发现了 8 个未知 bug, 并申请得到 CVE 编号 CVE-2021-45868。

本文的贡献总结如下:

(1) 在种子生成阶段, 结合 LSTM 神经网络, 使用相应的语法模板, 通过序列化和反序列化操作, 生成更加智能的种子, 有效提高了种子执行的成功率。

(2) 在模糊测试阶段, 通过静态分析方法获得系统调用显式依赖关系, 通过动态分析方法获得系统调用隐式依赖关系, 创建并维护系统调用依赖关系表, 进一步对种子内部的系统调用关系进行优化, 结合测试用例的生成策略和变异策略, 显著提高了选择系统调用的准确性。

(3) 实现原型系统 SyzMix, 能有效提升面向 Linux 内核的代码覆盖率和代码覆盖加速比, 发现了 8 个未知 bug, 并申请得到 CVE 编号 CVE-2021-45868。

2 背景和相关工作

软件的安全性严重影响着整个计算机系统, 它

影响着人们的日常生活, 甚至会造成严重的财产损失[9-10]。模糊测试是一个重复输入、解析、修改测试用例并以此来找到软件漏洞的过程。自 21 世纪以来, 模糊测试逐渐成为评估软件安全性的主流做法之一, 已经发现了数以千计的漏洞, 覆盖了各种各样的软件类型[11]。

从测试用例的生成方式划分, 可以把模糊测试分为基于生成的模糊测试和基于变异的模糊测试[12]。基于生成的模糊测试根据程序输入的一些结构知识(比如输入语法、文件格式)从头开始生成新的测试用例; 基于变异的模糊测试需要得到一组有效的初始输入, 并通过对初始输入进行变异来生成新的测试用例。

从程序执行的不同进行划分, 可以把模糊测试分为黑盒模糊测试、白盒模糊测试、灰盒模糊测试[13]。黑盒模糊测试是指不清楚测试程序内部的工作流程, 只观察程序执行的输出; 白盒模糊测试是指可以获得测试程序内部的源代码、设计规范等详细信息, 并通过这些详细信息加速模糊测试的执行; 处于两者之间的为灰盒模糊测试, 它使用程序执行时的代码覆盖等信息对程序进行模糊测试。

与一般的软件相比, 操作系统的内核环境十分复杂, 包括许多中断和内核进程, 导致内核中存在不确定的执行状态[9], 使得一般的覆盖反馈机制不能轻易地应用。另外, 在模糊测试器对内核进行测试时, 由于操作系统的重启而导致的内核崩溃会严重影响模糊测试的效率[14]。业界提出了不同的方法对 Linux 内核进行模糊测试, 本节分别介绍面向数据处理和面向系统调用接口的内核模糊测试[15], 并说明本文提出的 SyzMix 和之前工作的区别。

2.1 面向数据处理的内核模糊测试

AFL-QEMU[16]使用 AFL 对闭源的二进制文件进行模糊测试, 模拟器会在每个基本块之前进行插桩, 并把覆盖率信息发送给 AFL。文献[17]提出了改进的方法, 在每个基本块的开头注入一段新的代码, 使原来的插桩程序成为模拟程序的一部分, 因此就不需要每个块都返回到模拟器中, 并实现了链式缓存机制, 显著提高了模糊测试的效率。

TriforceAFL[18]在 AFL-QEMU 基础上进行扩展, 在主机上运行 AFL 和 QEMU 虚拟机, 在客户机上运行目标程序。AFL 把生成的测试用例发送到目标程序, QEMU 虚拟机把覆盖信息发送回 AFL。目标程序通过 QEMU 虚拟机专用的系统调用与 TriforceAFL 进行通信。系统调用 fork 仅将被调用进程的状态移动到新的进程中, 并且可能使进程对象置于未定义的状态。文献[19]在 TriforceAFL 的基础上做出了一定的改进。模型首先使用 strace 获得系统调用情况, 然后通过循环神经网络训练得到的序列, 相比原始的 TriforceAFL, 在覆盖率和发现 crash 方面都有一定的提升。

Trinity[20]通过结合每个被测系统调用的特定知识来进行模糊测试。使用这个方法来减少无效测试用例的执行, 可以使模糊测试达到更深的代码层次, 提高了测试用例触发系统错误的概率。另外, Trinity 将限制位模式, 而不是将随机的位置传递给地址参数, 因此, 在大多数情况下, 它所提供的地址是页面对齐的。同时, 在生成测试用例的参数地址时, Trinity 也会创建更加可能引起错误的地址, 比如溢出 sizeof(int)的边界值, 这样的地址可能会导致内核发生 off-by-one 的错误。

kAFL[21]是一个与操作系统无关、使用硬件进行辅助的方式进行内核模糊测试的模型。它利用一个管理程序和英特尔处理器跟踪的功能实现, 允许模糊测试器独立于目标操作系统, 因此在操作系统崩溃的情况下, 也几乎不会增加额外的性能开销。在 Linux、MacOS、Windows 中, 都发现了系统组件的缺陷。

Unicorefuzz[22]证明了即使目标函数没有暴露在用户空间中, 使用模拟器对内核函数进行模糊测试也是可行的。只要函数不与硬件进行交互, 任何解析函数都能以覆盖率作为引导的方式进行模糊测试。但是 Unicorefuzz 使用了内存访问的钩子函数, 这严重影响了模糊测试的效率, 吞吐量仅为 AFL-QEMU 的 46%。

2.2 面向系统调用接口的内核模糊测试

Syzkaller 是当前十分流行的内核模糊测试工具, 它由 Google 开发并一直维护, 能够对多种操作系统平台进行模糊测试[7]。它对内核进行插桩编译, 并在一组虚拟机中运行内核[23]。Syzkaller 使用一系列测试程序来对内核进行测试。同时, 实现了一种系统调用的描述语言 syzlang 来定义系统调用的语法和语义, 但是所有的系统调用的描述都要手动编写[8]。为了构造这样的测试程序, Syzkaller 提供了生成新的序列和变异已经存在的序列两种策略。生成策略依靠语法模板, 挑选出系统调用函数原型, 并依次生成它们的参数。

Moonshine[24]提出了对输入的种子进行静态分析的方法, 先通过 strace 获取实际运行程序的覆盖情况, 在模糊测试运行之前对数据进行静态分析得到函数之间的依赖关系, 得到优质的种子来引导模糊测试的过程, 成功解决了 Syzkaller 需要手动编写种

子的难题。FastSyzkaller[25]首先采集大量的可以触发不同覆盖率的测试用例以及可以引起系统崩溃的测试用例, 然后使用 *n* 元模型(*n*-gram)生成新的种子, 进而指导模糊测试的执行。上述两种模型都是在模糊测试正式启动之前对输入的种子进行优化, 并且取得了一定的效果。

Syzkaller 也扩展支持了面向 Linux 内核中的 USB 驱动系统的模糊测试, 定义了多个针对 USB 驱动模块的系统调用, 在 USB 子系统中挖掘到超过 300 个未知的 bug[26]。Agamotto[27]实现了一种轻量级的虚拟机检查点的机制, 大幅提高了面向内核驱动程序模糊测试的效率。由于面向内核的模糊测试经常连续执行相似的测试用例, Agamotto 在执行测试用例的同时动态创建多个检查点, 并使用检查点来跳过部分执行步骤, 与 Syzkaller 相比, 在面向 USB 驱动程序的模糊测试中, 平均效率提高了 66.6%。

SyzVegas[6]在模糊测试执行时结合了增强学习的技术, 使用多臂赌博机(Multi-armed Bandits, MAB)算法, 根据内核的状态动态调整不同种子队列的使用概率, 调整任务序列的次序以及生成策略和变异策略的使用比例, 以可以接受的执行速度损耗为代价, 显著提高了代码的覆盖率。

Diskaller[28]在 Syzkaller 的基础上, 对模型的并行操作进行优化, 在资源不足的情况下, 多个 CPU 同时工作时, 覆盖率得到一定的提高。

Healer[29]是受 Syzkaller 启发, 使用 Rust 实现的一个面向内核的模糊测试工具, 与 Syzkaller 类似, Healer 使用语法模板 syzlang 提供的信息来生成符合参数要求和语义约束的系统调用。但是 Healer 移除了 Syzkaller 原有的用来指导选择的表, 通过动态移除系统调用尝试精简序列的方式检测覆盖率变化, 从而得到系统调用之间的关系来指导序列的生成和变异。

SyzScope[30]重点关注内核漏洞报告中安全影响程度。通过分析 syzbot 中报告的上千个低风险错误, 发现了其中 15%的错误实际上包含高风险的影响, 其中仍然有部分错误没有可用的补丁。SyzScope 通过对错误报告进行分析, 发现部分 WARNING 级别和 INFO 级别的报告以及 UAF/OOB 读的错误中, 实际含有 UAF/OOB 写的错误, 这可能导致控制流劫持以及任意内存写入, 对整个系统造成十分严重的后果。

2.3 SyzMix 主要特点

SyzMix 的主要工作体现在种子生成和种子内部系统调用序列的优化上。本节主要阐释 SyzMix 和相关工作 FastSyzkaller、Syzkaller 以及文献[19]上的区别。

在种子生成阶段, FastSyzkaller[24]使用 *n*-gram 语言模型生成新的种子文件, 但是没有在运行时对系统调用之间的依赖关系进行分析。SyzMix 使用了 LSTM 神经网络生成种子文件, LSTM 模型相比 *n*-gram 模型可以更好地处理序列建模的问题, 可以得到更加合适的系统调用 ID 序列, 提高种子文件执行的成功率。SyzMix 在运行时对系统调用之间的依赖关系进行分析, 并得到系统调用依赖关系表来得到更加优质的种子文件。

文献[19]把神经网络应用于 TriforceAFL, 所使用的训练数据来自 strace 获取到的序列, 把长度为 1～16 的长序列作为输入。SyzMix 则把神经网络应用在更加先进的 Syzkaller 之上, 使用的训练数据来自实际执行的数据, 并且输入序列的长度在 1～10 之间, 并且绝大多数是长度<5 的较短序列, 并且对神经网络的结构进行了进一步的优化, 提高了在本模型中的适用性。

在模糊测试执行中, 相比 Syzkaller, SyzMix 做了两方面的优化工作。一方面, 对系统调用进行静态分析, 通过参数的调用关系得到参数类型树, 通过分析参数类型树获得系统调用之间的显式依赖关系, 提高选择系统调用的准确性。另一方面, 受到 Healer 工作[29]启发, 通过动态分析覆盖率的变化, 获得系统调用之间的隐式依赖关系, 增加种子内部系统调用的紧密性。

3 系统实现与设计

3.1 整体工作流程

本文所提出的模型 SyzMix 在 Syzkaller 基础上进行构建, 是一个能够智能生成种子并引导模糊测试执行过程的内核模糊测试模型。SyzMix 的整体流程如图 3 所示, 可以分为种子生成阶段和模糊测试阶段。

种子生成阶段对应图 3 中的上半部分, 具体可以分为三个步骤。第一步是反序列化过程, SyzMix 将采集大量语料库, 通过把输入语料库进行反序列化操作生成系统调用序列, 并把序列中的系统调用与不同的 ID 进行映射, 生成系统调用 ID 序列, 作为 LSTM 的输入。第二步是系统调用 ID 序列生成过程, SyzMix 将系统调用 ID 序列输入到 LSTM 神经网络中进行训练, 得到新的系统调用 ID 序列。第三步是序列化过程, SyzMix 使用语法模板 syzlang 生成对应的系统调用并生成所需要的参数, 得到新的种子文件, 作为模糊测试器的输入。在 3.2 节中将对种子生成阶段进行详细描述, 其中, 3.2.1 节、3.2.2 节、3.2.3 节中将分别对应反序列化过程、系统调用 ID 序列生

成过程、序列化过程三个步骤。

模糊测试阶段对应图 3 中的下半部分, SyzMix 通过对参数类型树进行静态分析获得系统调用显式依赖关系, 通过对系统调用覆盖率的变化进行监控, 动态地获得系统调用隐式依赖关系, 结合两类不同的依赖关系, SyzMix 创建并维护系统调用依赖关系表, 进一步对种子内部的系统调用关系进行优化。同时, SyzMix 会时刻监控被测试内核的状态, 分析内核输出信息, 捕获异常状况和内核崩溃状况, 对可以造成内核崩溃的程序进行复现。3.3 节会介绍 SyzMix 如何获得系统调用依赖关系表, 3.4 节会介绍如何使用系统调用依赖关系表。

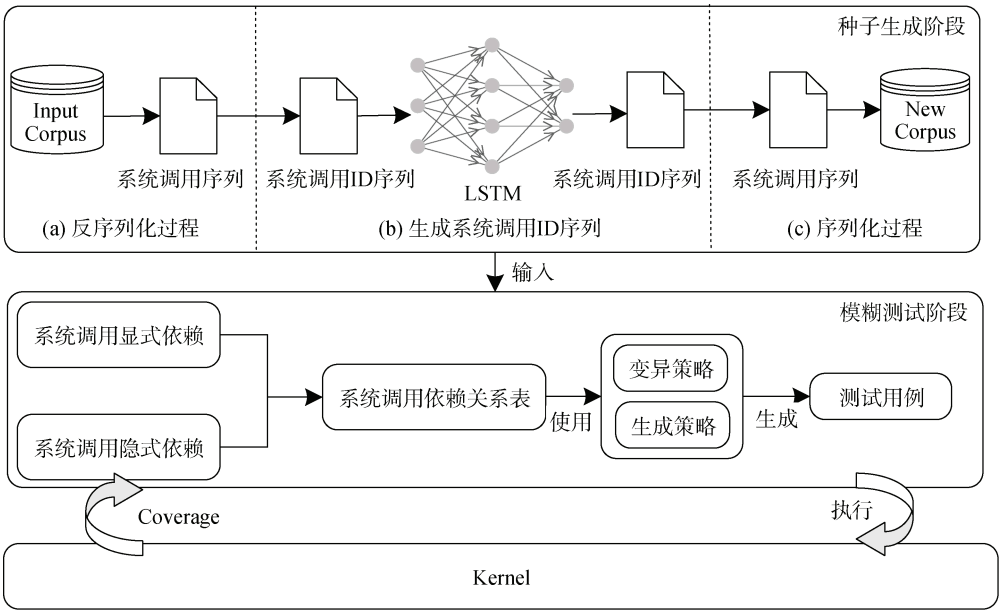

图 3 SyzMix 整体流程

Figure 3 SyzMix overall process

3.2 种子生成阶段

当前 Linux 内核中定义了超过 300 种系统调用, 假设每个种子的序列由 10 个系统调用组成, 就会出现超过 300^{10} 种结果, 然而 Syzkaller 每秒只能执行 100～1000 次, 稀缺的计算资源就要求更高质量的种子输入。并且随机组合的序列大多数都是无效的, 它们会因为没有对应的内核环境而提前退出, 执行不到更深层次的代码逻辑。因此, 生成内部关系紧密的种子对提高模糊测试的效率有正向效果。

种子生成阶段可以分为三个步骤。第一步, 把输入的语料库文件进行反序列化操作, 得到系统调用 ID 序列, 作为 LSTM 神经网络的输入。第二步, 使用 LSTM 神经网络生成新的系统调用 ID 序列。第三步, 把输出的系统调用 ID 序列进行序列化操作, 生成种子文件, 作为模糊测试器的输入。

3.2.1 反序列化过程

SyzMix 最初接收的文件是语料库文件, 语料库文件可以作为模糊测试器的输入和输出, 但是不可以被直接读取, 需要通过反序列化操作分解为若干个种子文件, 序列化过程如图 4 所示。

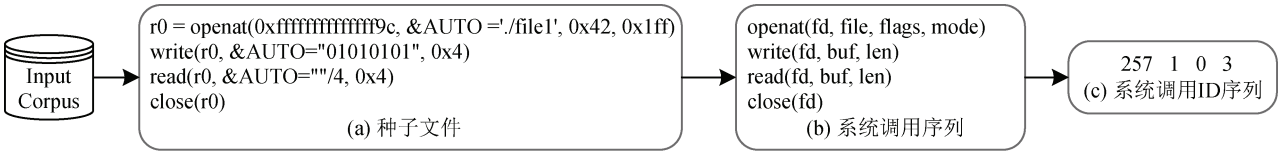

图 4 反序列化过程

Figure 4 Deserialization process

经过解析语料库文件, 能够得到种子文件, 如图 4(a)所示, 这种文件格式不能直接作为后续 LSTM 神经网络的输入。因此, 需要通过后续反序列化操作, 把种子文件转化为 LSTM 神经网络可以接受的输入。

首先遍历采集到的全部种子文件, 提取出全部不相同的系统调用名称, 忽略系统调用参数的区别, 把对应系统调用名称映射到标准语法模板 syzlang 中, 得到系统调用序列, 如图 4(b)所示。下一步再把得到

的结果与标识符进行映射, 得到系统调用 ID 序列, 如图 4(c)所示, 最终得到的系统调用 ID 序列可以作为 LSTM 神经网络的输入。

3.2.2 系统调用 ID 序列生成

为了生成更有意义的系统调用 ID 序列, SyzMix 使用 LSTM 神经网络来生成此类序列。LSTM 神经网络的输入为 3.2.1 节中得到的系统调用 ID 序列, 输出为新的系统调用 ID 序列。

LSTM 的结构如图 5 所示, LSTM 神经元包括用来长期记忆的细胞状态以及输入门、输出门、遗忘门。

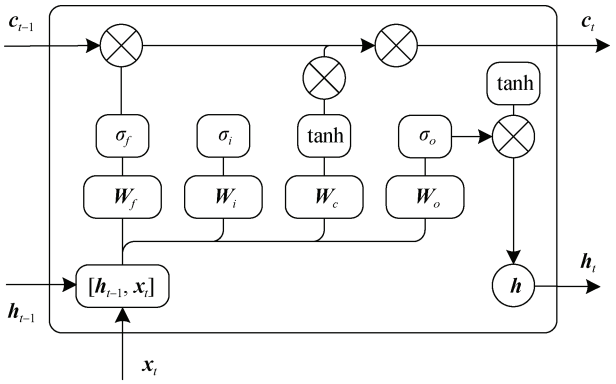

图 5 LSTM 结构

Figure 5 LSTM structure

若 $\boldsymbol{W}$ 表示权重矩阵, $\boldsymbol{b}$ 表示偏置向量, $\sigma(\boldsymbol{x})$ 表示激活函数, 则门函数表示为:

$$g(\boldsymbol{x}) = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{1}$$

其中 $\sigma(\boldsymbol{x}) = \dfrac{1}{1+\exp(-\boldsymbol{x})}$ 是非线性激活函数, 用来把输入值映射到 0～1 之间。

分别使用 $\boldsymbol{i}$ 、 $\boldsymbol{f}$ 、 $\boldsymbol{o}$ 表示输入门、遗忘门、输出门。公式(2)～(6)表示了 LSTM 的向前计算过程。

公式(2)～(3)表示了输入门的计算过程, 输出门的值如公式(5)所示, 神经元的输出如公式(6)所示。

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_{xi}\boldsymbol{x}_t + \boldsymbol{W}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i) \tag{2}$$

$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}_{xf}\boldsymbol{x}_t + \boldsymbol{W}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f) \tag{3}$$

$$\boldsymbol{c}_t = \boldsymbol{f}_t \times \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \times \tanh(\boldsymbol{W}_{xc}\boldsymbol{x}_t + \boldsymbol{W}_{hc}\boldsymbol{x}_{t-1} + \boldsymbol{b}_c) \tag{4}$$

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_{xo}\boldsymbol{x}_t + \boldsymbol{W}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_o) \tag{5}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \times \tanh(\boldsymbol{c}_t) \tag{6}$$

为了取得更加有效的系统调用 ID 序列, 在 LSTM 训练的过程中, 会抽取一定数量的系统调用 ID 序列直接进行序列化过程生成种子文件, 将它们交给 SyzMix 执行, 计算种子文件执行的成功率, 确定最合适的训练次数, 防止过拟合。

3.2.3 序列化过程

由 LSTM 获得的系统调用 ID 序列不能直接作为模糊测试的输入, 需要对参数进行赋值、通过合法性检测后所生成的种子才能作为模糊测试的输入。

首先, 找到与系统调用 ID 序列对应的系统调用, 然后参考语法模板确定参数表, 根据参数类型的不同赋予不同的初始值, 通过这种方法可以排除大量无效的种子文件。

这一步称为序列化过程, 是反序列化过程的逆过程, 这一步的初始输入为系统调用 ID 序列, 如图 4(c)中的形态, 通过这一过程, 把系统调用 ID 序列转化带有参数表的系统调用序列, 如图 4(b)中的形态, 再根据参数类型的不同生成不同的参数, 转化为图 4(a)中的形态, 最后生成可以作为模糊测试的输入语料库文件。

因此, 从整体的角度观察, SyzMix 的种子生成阶段, 就是通过解析已存在的语料库生成新的语料库的过程。

3.3 获得系统调用依赖关系表

种子生成阶段中生成的种子不一定完全符合系统调用之间的依赖关系, 依然有可能产生错误的输出, 这种情况会对覆盖率产生负面的影响。因此需要在模糊测试过程中不断对种子进行调整, 进一步对种子内部的系统调用关系进行优化, 从而进行更加高效的模糊测试。

因此在模糊测试正式启动前, SyzMix 会创建一个大小为 $n \times n$ 的矩阵 $\boldsymbol{M}$, 用来表示系统调用之间的依赖关系, 并把它命名为系统调用依赖关系表, 其中 n 为系统调用的总数。SyzMix 在创建系统调用依赖关系表时, 需要计算出系统调用之间的显式依赖关系和隐式依赖关系。显式依赖关系通过对不同系统调用的参数依赖进行静态分析获得, 隐式依赖关系通过监控测试用例被执行时的覆盖率变化来获得。

3.3.1 获得系统调用显式依赖关系

SyzMix 遍历所有系统调用的参数表, 并访问参数的祖先类型, 得到参数类型树, 用来获得系统调用之间的显式依赖关系。参数类型树如图 6 中所示, 假设系统调用 close()需要 fd 类型的输入参数, 而系统调用 pipe()的参数返回类型为 fd_btf, 通过查找参数类型树, 发现 fd 是 fd_btf 的父类, 我们就可以说 close()显式依赖于 pipe()。

算法 1.计算系统调用 c_{out}, c_{in} 的依赖性

输入: 参数类型树 *ParaTree*, 系统调用 c_{out}, c_{in}, 系统调用依赖关系表 *M*

```
1: FOR para_out IN c_out.outputPara
2:     FOR para_in IN c_in.inputPara
3:         paraArr = visitParaTree(ParaTree, para_out)
4:         IF paraArr.CONTAIN(para_in)
5:             M[out, in] = prioHigh
6:             RETURN
```

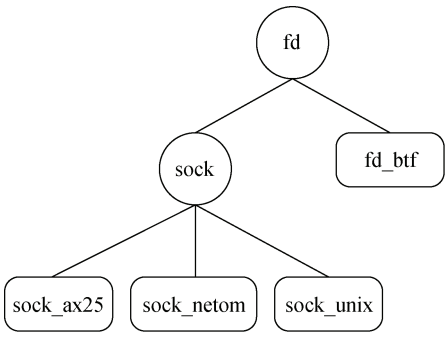

图 6　参数类型树

Figure 6　Parameter type tree

对于有显式依赖关系的系统调用 c_{out}、c_{in}, SyzMix 会把系统调用依赖关系表对应的位置 $M[out, in]$ 置为高优先级。算法 1 介绍了如何计算系统调用 c_{out} 与 c_{in} 之间的显式依赖关系。依次取出 c_{in} 的每个输入参数 $para_{in}$ 以及 c_{out} 的每个输出参数 $para_{out}$ (第 1～2 行), 然后递归访问参数类型树得到 $para_{out}$ 的所有祖先类数据, 赋值到数组 *paraArr* 中(第 3 行), 如果数组 *paraArr* 中包含参数 $para_{in}$, 就说明 c_{in} 显式依赖于 c_{out}, 于是把系统调用依赖关系表 M 中对应的位置 $M[out, in]$ 置为高优先级 prioHigh 并返回(第 4～6 行), 否则进入下一次循环。

下面我们将用具体的例子解释如何获得系统调用显式依赖关系。由图 6 可知, fd 是 sock、fd_btf 的父类, 而 sock 又是 sock_ax25、sock_netrom、sock_unix 的父类。图 7 展示了几个系统调用的模板, 接下来我们将用前面的序号代表后续的每一个系统调用。从图 7 中可以看出, 0 的输出参数类型为 sock_unix, 1 的输出参数类型为 fd_btf; 2 需要输入参数类型 fd, 3 需要输入的参数类型为 sock_unix。结合图 6 表示的参数类型树, 0 的输出参数类型 sock_unix 正是 3 的输入参数类型, 所以 3 依赖 0; 由于 2 需要输入参数类型为 fd, 同时 fd 是 sock、fd_btf 的父类, 所以 2 对 0、1 都具有依赖性。

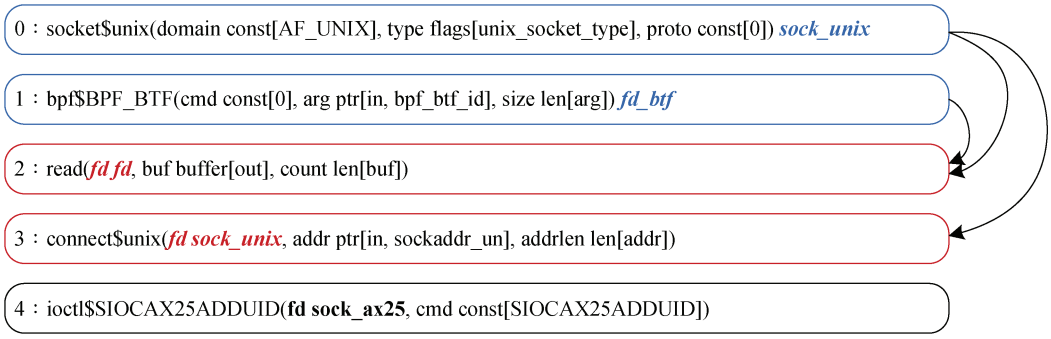

图 7　获取系统调用显式依赖关系的过程

Figure 7　Procedure for obtaining explicit dependencies of system call

3.3.2　获得系统调用隐式依赖关系

静态的系统调用依赖关系表对模糊测试的引导效果会随着模糊测试过程的不断进行而效率降低, 所以要在模糊测试的过程中对系统调用依赖关系表进行动态调整。

获得系统调用隐式依赖关系受到 Healer[28]中的动态分析部分启发。区别在于 Healer 会对得到的信息进行实时动态更新, 即在精简序列的同时更新系统调用之间的影响关系。而 SyzMix 提出了一种延时更新的机制, 即在执行种子序列获得覆盖率时会对系统调用序列进行精简(算法 2), 但并不立即进行更新, 而是在模糊测试器重启时、整个模糊测试过程开始之前一次性遍历全部的测试用例, 对每个测试用例进行分析, 得到全部的差异, 并且更新系统调用依赖关系表(算法 3)。在 4.4 节中会通过实验比较两种方法对 SyzMix 的影响。

在模糊测试的执行过程中, SyzMix 会维护测试用例 *case* 的队列 Q, 每个测试用例 *case* 由两部分组成, 第一部分是系统调用序列 *Prog*, 另一部分是这个系统调用序列的核心系统调用所在的位置 *CallIndex*。核心系统调用是指该系统调用处于当前顺序时, 有可能发现新的覆盖率。*CallIndex* 表示该系统调用所处的位置。SyzMix 通过删减核心系统调用之前的系统调用并监控核心系统调用覆盖率的变化, 从而得到显式依赖没有捕获的系统调用隐式依赖关系。

算法 2.精简系统调用序列

```
输入: 测试用例队列 Q
1: case = dequeue(Q)
2: IF validate(case.Prog) == false
3:     RETURN
4: IF case.CallIndex == 0 ||
   hasNewCoverage(case.Prog[case.CallIndex])==false
5:     RETURN
6: cov = getCoverage(case.Prog[case.CallIndex])
7: p = case.Prog
8: FOR i = 0; i < case.CallIndex; i++
9:     RemoveCall(p, i)
10:    newCov = getCoverage(case.Prog[case.CallIndex–1])
11:    IF cov == newCov
12:        case.Prog = p
13:        case.CallIndex = case.CallIndex – 1
14:        i = i – 1
15:    ELSE
16:        p = case.Prog
```

精简系统调用序列如算法 2 所示。每一个位于测试用例队列 Q 中的测试用例 *case* 依次出队(第 1 行), SyzMix 首先对测试用例 *Prog* 部分中的每个系统调用进行静态检测, 保证它们的语法结构符合语法模板, 并且每一个参数都进行了合适的填充; 如果不符合语法规则, 直接丢弃并返回(第 2～3 行)。如果核心系统调用位于系统调用序列的第一个位置或者没有出现新的覆盖情况, 也会直接返回(第 4～5 行)。接下来计算核心系统调用的覆盖情况(第 6 行), 并把整个系统调用序列 *Prog* 拷贝到 p 中(第 7 行)。然后尝试移除每个位置的系统调用并监控核心系统调用的覆盖情况, 如果移除第 i 个系统调用时覆盖率发生变化, 说明第 i 个系统调用和核心系统调用存在隐式关系, 第 i 个系统调用不能被移除, 就把 p 还原; 否则, 把修改后的 p 赋值回 *Prog*(第 11～16 行)。

算法 3.更新系统调用依赖关系表

```
输入: 系统调用依赖关系表 M, 测试用例队列 Q
1: WHILE !isEmpty(Q)
2:     case = dequeue(Q)
3:     FOR front = 0; front < length(case) – 1; front++
4:         FOR rear=front+1;rear<length(case); rear++
5:             IF M[front, rear] != prioHigh
6:                 M[front, rear] = prioHigh
```

每次模糊测试重新启动时, SyzMix 就会检查测试用例队列 Q 并更新系统调用依赖关系表 M, 具体过程如算法所示。只要测试用例队列 Q 不为空, 就依次出队得到测试用例 Q(第 1～2 行), 检查依赖关系表 M 对应的位置是否为 prioHigh(第 3～5 行), 否则, 就把对应位置置为 prioHigh(第 6 行)。

3.4 使用系统调用依赖关系表

SyzMix 使用的变异算法与传统的模糊测试相似, 包括在序列中插入新生成的系统调用、按位翻转、两个不同的序列进行拼接、变异系统调用参数、随机移除序列中的系统调用。

SyzMix 使用的生成策略是随机在当前序列中选择一个系统调用作为参考, 生成新的系统调用, 放置在被选择的系统调用下一个位置, 并不断循环生成策略直到达到预定义数量。生成新的系统调用时, 为了保证新的系统调用序列能正常执行, 关键在于使用正确的挑选策略保证新的系统调用依赖已存在的序列。系统调用依赖关系表就是通过影响系统调用的挑选过程, 指导测试用例的变异和生成, 相比随机挑选, 能选择出相关性更强的系统调用, 有利于测试用例到达更深层的逻辑, 从而提高代码的覆盖率, 触发更多的内核崩溃。

算法 4. 生成并插入新的系统调用

```
输入: 系统调用序列 Prog, 系统调用依赖关系表 M
1: IF Prog != NULL
2:     call = rand(Prog)
3: ELSE
4:     call = randAll()
5: prioArr = M[call, :]
6: newId = choosePrio(prioArr)
7: newCall = generate(newId)
8: IF Prog != NULL
9:     insert(Prog, newCall, call)
10: ELSE
11:    Prog = newCall
```

算法 4 描述了生成并插入新的系统调用的过程, 这个算法在变异策略和生成策略中均有使用。首先, 要选择一个系统调用 *call* 作为新生成的系统调用的依赖, 如果系统调用不为空, 就在当前序列中进行挑选; 否则, 就在全部的系统调用中随机抽取(第 1～4 行); 然后取出系统调用依赖关系表 M 中的系统调用 *call* 对应的依赖关系数组 *prioArr*, 根据数组中的权值选择出新的系统调用 ID 序号 *newId*, 并生成对应的系统调用 *newCall*(第 5～7 行)。最后, 把 *newCall* 插入到 *Prog* 中, 如果 *Prog* 不为空, 就把 *newCall* 插入到 *call* 后面; 否则, 把 *newCall* 赋值到 *Prog*(第 8～11 行)。

4 实验与测试分析

4.1 实验设置

本文所提出的模型 SyzMix 是在 Syzkaller 基础上进行构建的, 因此将主要与目前流行的内核模糊测试工具 Syzkaller 进行实验对比。分别从生成种子的有效性、选择系统调用的准确性、代码覆盖能力、代码覆盖加速比、漏洞发现能力等方面进行实验。同时, 为了验证本文提出的各个部分都对覆盖率的提升起到以一定的作用, 将在 4.5.1 节中设计 SyzMix-LSTM、SyzMix-static 分别进行对比。

另外, 为了说明本文提出神经网络的有效性, 将在 4.2 节中与文献[19]中的模型进行对比。最后将在 4.7 节中探究此神经网络在 Healer 中的适用性。

生成种子的成功率可以验证 LSTM 神经网络在 SyzMix 中的适用性, 种子执行的有效性则对模糊测试至关重要, 因为随机生成的种子很难被成功执行。选择系统调用的准确性可以展示系统调用依赖关系表的有效性, 选择依赖关系更强的系统调用组成序列, 能有效提高模糊测试的效率, 减少不相关的系统调用对整个过程的误导。代码覆盖率和代码覆盖加速比是能从整体上体现 SyzMix 的模糊测试效果, 是体现模糊测试模型好坏最关键的标准。漏洞发现能力则可以体现模型的实际价值。

在进行测试时, 使用了四个不同版本的内核 4.19、内核 5.11、内核 5.16、内核 5.18 作为测试对象。内核 4.19 是 4.x 版本中最后一个长期维护的版本, 也是 Debian10 使用的内核版本。内核 5.11 是 2021 年发布的第一个内核版本, 增加了对 WiFi 6 的支持, 可以更好的发挥 AMD CPU 的性能。在进行模型测试时, 内核 5.16 是最新的稳定版内核, 优化了 FUTEX2 等系统调用。内核 5.18 是当时正在更新的内核版本, 把内核中的 C 语言标准从 C89 升级到 C11。

4.2 种子执行成功率的提高

生成种子的成功率是衡量 LSTM 神经网络在 SyzMix 中适用性的重要指标, 生成种子的成功率越高, 说明种子更加适合整个模型, 它指的是种子中的系统调用执行的成功率。在种子生成之后, 正式模糊测试启动之前, 我们对生成的种子进行测试, 计算它们的执行成功率, 以此来评估 LSTM 神经网络的效果。

SyzMix 所需要的种子是由若干系统调用组成的序列。我们依次执行每个序列, 并监控序列中每个系统调用返回的错误码 *errno*, 如果 *errno* 不为 0, 就认为这个系统调用执行失败。种子执行成功率的计算公式如下:

$$success = errno_0 / syscall_{sum} \tag{7}$$

$errno_0$ 表示返回值为 0 的系统调用的数量, $syscall_{sum}$ 表示所有种子中系统调用的总数。

生成种子部分使用 Pytorch 实现, 使用 GTX 960M 进行训练和生成。训练数据来自预先执行 48 小时的 SyzMix 所生成的测试用例。使用的 LSTM 网络的 batch_size 设置为 8, 隐藏层神经元数量设置为 128, 学习率设置为 0.001, 训练的 epoch 设置为 75. 输入的种子长度为 1～10, 并且绝大多数长度小于 5. 为量化训练效果, 训练时记录每个 epoch 的困惑度 (*perplexity*)。困惑度的计算方法如下:

$$perplexity = \exp(Loss / num) \tag{8}$$

其中 *Loss* 为反向传播过程中获得的损失函数, *num* 表示要预测的系统调用 ID 序列的长度。

同时, 为了证明本文中的神经网络优于文献[19]中提到的神经网络结构, 根据文中描述, 把隐藏层神经元数量设置为 512, 把输入的系统调用的长度设置为 1～16, 且长度偏长的序列居多, 每 10 个 epoch 测试一次种子执行的成功率, 把模型命名为 modified-TriforceAFL。

训练过程如图 8 所示, 实线部分表示 SyzMix 的困惑度和种子执行的成功率, 虚线部分表示 modified-TriforceAFL 的困惑度和种子执行的成功率。Syzkaller 的种子执行成功率等于 epoch 为 0 的值, 在 epoch 为 50 时, modified-TriforceAFL 的种子执行成功率达到最大值 37%, 提高了 11%; 在 epoch 为 60 时, SyzMix 的种子执行成功率达到最大值 42%, 相比 Syzkaller 提高了 16%, 证明了神经网络的有效性。

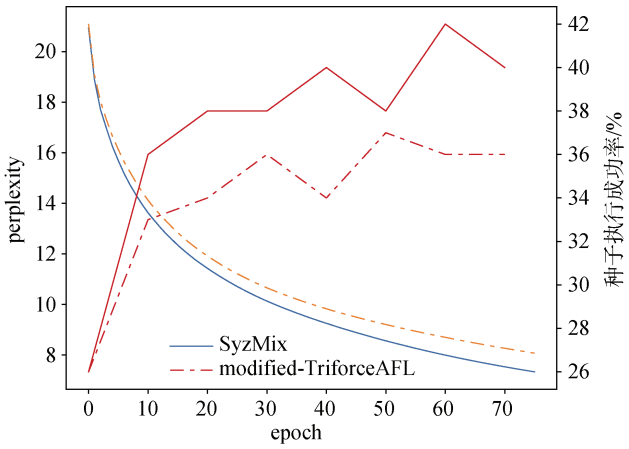

图 8 LSTM 训练过程

Figure 8 LSTM training process

为了进一步说明神经网络的有效性, 图 9(a)、图 9(b)、图 9(c)分别采集了 Syzkaller、modified- TriforceAFL、SyzMix 在各自种子执行成功率达到最大值时的种子执行情况。除了采集种子成功的情况, 还采集了几个明显导致种子执行错误的原因。SyzMix 相比 Syzkaller, 对于 9 号错误 Bad file descriptor、22 号错误 Invalid argument, 分别从 40%降低至 34%、从 24%降低至 7%, 都有不错的效果。但是对于 2 号错误 No such file or direction, 所占比例从 6%增加至 9%, 不过对整体结果影响不大, 是可以接受的。SyzMix 相比 modified-TriforceAFL, 2 号错误、22 号错误、25 号错误均有不同程度的下降。通过上述现象可以说明, 更加有序的系统调用顺序可以显著提高种子执行成功率。

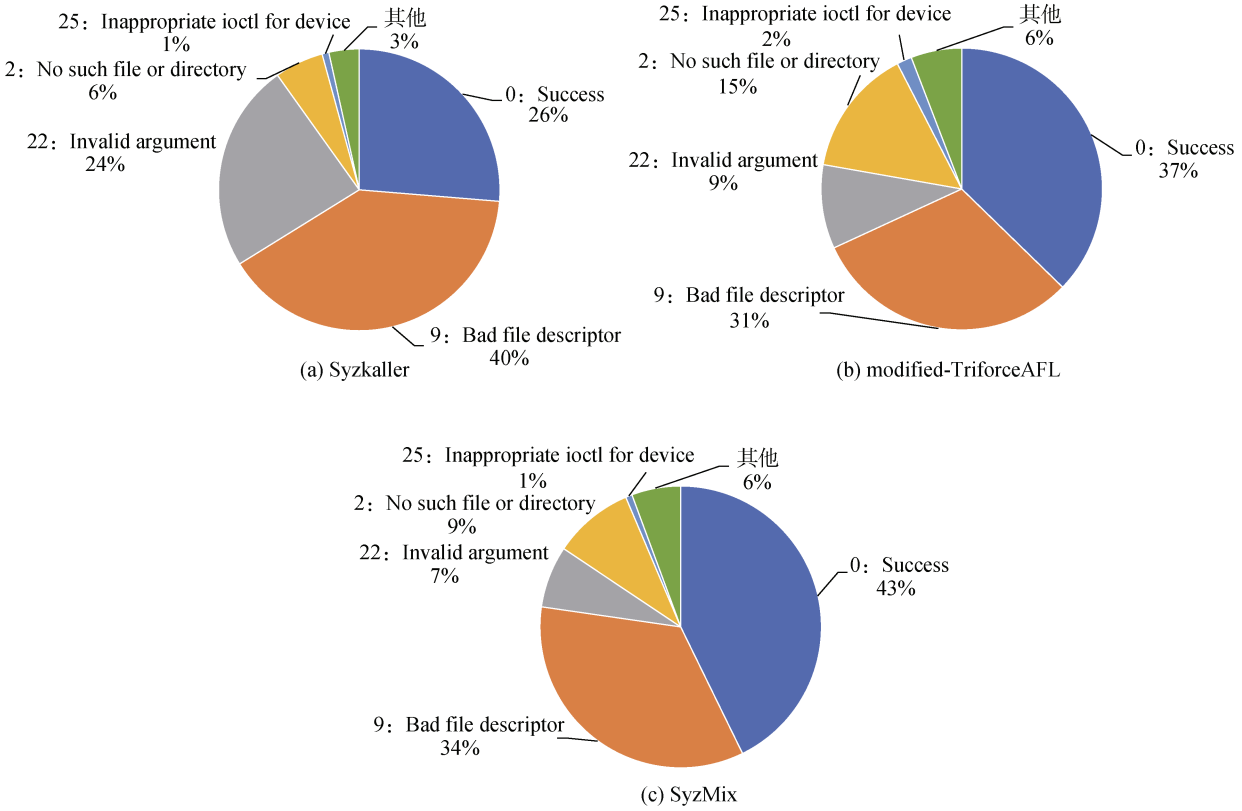

图 9　种子执行情况

Figure 9　Seed execution rate

4.3　选择系统调用的准确性

通过测试选择系统调用的准确性, 可以验证 SyzMix 中系统调用依赖关系表的有效性。选择系统调用的准确性是指以系统调用 c_{out} 为标准选择下一个系统调用时, 能选到依赖于 c_{out} 的系统调用 c_{in} 的次数。

通过分析系统调用的参数类型树可得, 系统调用 read()、write()、mmap()依赖于 open(), 系统调用 listen()、setsocket()依赖于 socket()。所以, 我们以 open()、socket()为基准, 选择下一个系统调用, 统计 open()所选择的系统调用为 read()、write()、mmap()的次数以及 socket()所选择的系统调用为 listen()、setsocket()的次数。

表 1 为循环执行选择程序 1×10^5 的结果。open()选择 read()、write()、mmap()的概率分别上涨了 141.6%、135.6%、127.3%; socket()选择 listen()、setsocket()的概率分别上涨了 57.6%、51.4%, 平均可以达到 88.8%的涨幅。

表 1　选择系统调用的准确性

Table 1　Accuracy of choosing system calls

	Syzkaller	SyzMix	提高的准确性(%)
open -> read	161	389	+141.6
open -> write	163	384	+135.6
open -> mmap	187	425	+127.3
socket -> listen	316	498	+57.6
socket -> setsocket	360	545	+51.4
平均	237.4	448.2	+88.8

4.4　动态更新策略的比较

本节中将会比较在 3.3.2 节获得系统调用隐式依赖关系时不同策略的效果。

根据系统调用依赖关系表更新时间的不同, 可

以分为实时更新策略和延时更新策略。其中实时更新参考策略了 Healer 中的动态更新方法。图 10(a)、10(b)分别表示了两种策略在内核 5.16、内核 5.18 中的覆盖率情况。模型 SyzMix-realtime、SyzMix-delay 分别使用了实时更新策略和延时更新策略, 其余的方面保持不变。

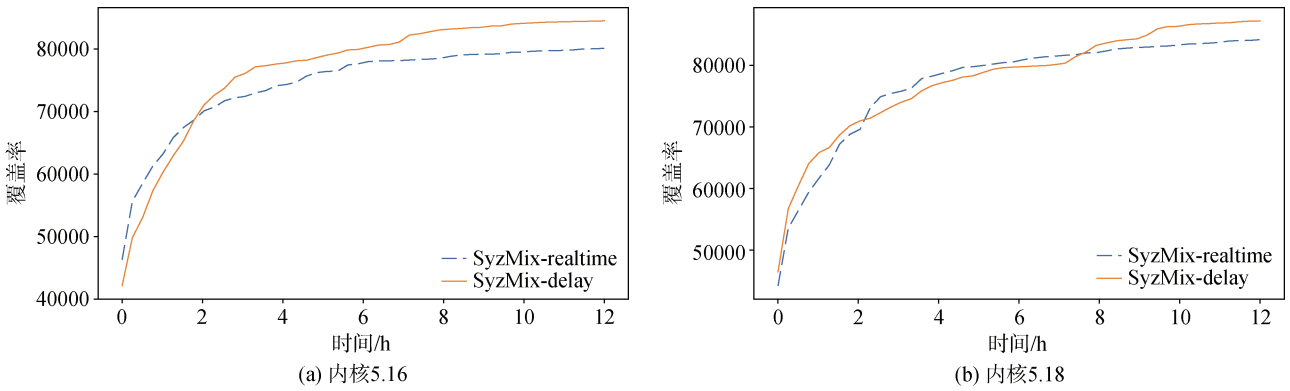

图 10 动态更新策略比较

Figure 10 Dynamic update policy comparison

在相同环境下执行 12 h 后, 相比实时更新的策略, 延时更新策略在内核 5.16、内核 5.18 中的覆盖率可分别提高 5.49%、3.61%。说明在当前模型中, 延时更新的策略是更加有效的。

4.5 代码覆盖率和代码覆盖加速比的提高

代码覆盖率和代码覆盖加速比是能从整体上体现 SyzMix 的模糊测试效果, 是体现模糊测试模型好坏最关键的标准。代码覆盖加速比指的是达到相同的覆盖率使用的时间, 所用的时间越短, 代码覆盖加速比越高。

4.5 节、4.6 节、4.7 节中使用的机器为 Dell PowerEdge R740, 操作系统为 64 位的 Ubantu18.04。内核所挂载的系统版本为 Debian Stretch, 内存大小设置为 40G, 并行进程数设置为 16, 在编译内核时开启 CONFIG_KCOV、CONFIG_DEBUG_INFO 等选项来获取内核源码的覆盖情况, 使用 KASAN 对内核进行动态内存检测, 完成挂载的系统将在 QEMU 虚拟机中运行。

由于本文的主要工作在于基于 LSTM 的系统调用序列顺序学习和基于类型的静态分析, 在所测试的模型中, 除了 Syzkaller 和本文提出的 SyzMix, 还设计了模型 SyzMix-LSTM、模型 SyzMix-static。其中模型 SyzMix-LSTM 在 Syzkaller 基础上增加了 LSTM 的部分, 用来测试种子生成部分的效果; 模型 SyzMix-static 在 Syzkaller 基础上增加了静态分析的部分, 用来测试基于类型的静态分析的效果。

为了实验的公平性, SyzMix 和 SyzMix-LSTM 使用的初始种子为第一阶段生成的 7035 个种子文件, Syzkaller 和 SyzMix-static 则使用相同数量的随机生成的种子进行测试。

4.5.1 覆盖率的提高

对不同内核进行覆盖率检测的测试时间设置为 12 h, 因为 12 h 之后覆盖率不会有明显的增长, 同时测试过程中以 15 min 为间隔进行覆盖率的记录。分别使用 Syzkaller、SyzMix-LSTM、SyzMix-static、SyzMix 进行测试, 覆盖率变化趋势如图 11 所示。

表 2 记录了在 12 h 时 Syzkaller、SyzMix-LSTM、SyzMix-static、SyzMix 的覆盖率情况。在内核 4.19、内核 5.11、内核 5.16、内核 5.18 中, 对比 Syzkaller, SyzMix-LSTM 可分别提高 8.44%、3.02%、3.86%、3.27%, 平均可提高 4.65%, 证明 LSTM 对覆盖率的提升具有正向的效果; SyzMix-static 可分别提高 9.25%、4.65%、5.05%、3.26%, 平均可提高 5.55%, 证

表 2 SyzMix SyzMix-LSTM SyzMix-static Syzkaller 覆盖率比较

Table 2 SyzMix SyzMix-LSTM SyzMix-static Syzkaller coverage compare

内核版本	Syzkaller	SyzMix-LSTM	SyzMix-LSTM 提高的覆盖率(%)	SyzMix-static	SyzMix-static 提高的覆盖率(%)	SyzMix	SyzMix 提高的覆盖率(%)
4.19	65991	71561	+8.44	72095	+9.25	72201	+9.41
5.11	72427	74614	+3.02	75795	+4.65	79595	+9.90
5.16	78178	81197	+3.86	82114	+5.05	84490	+8.07
5.18	83278	86003	+3.27	85989	+3.26	86677	+4.08
平均			+4.65		+5.55		+7.87

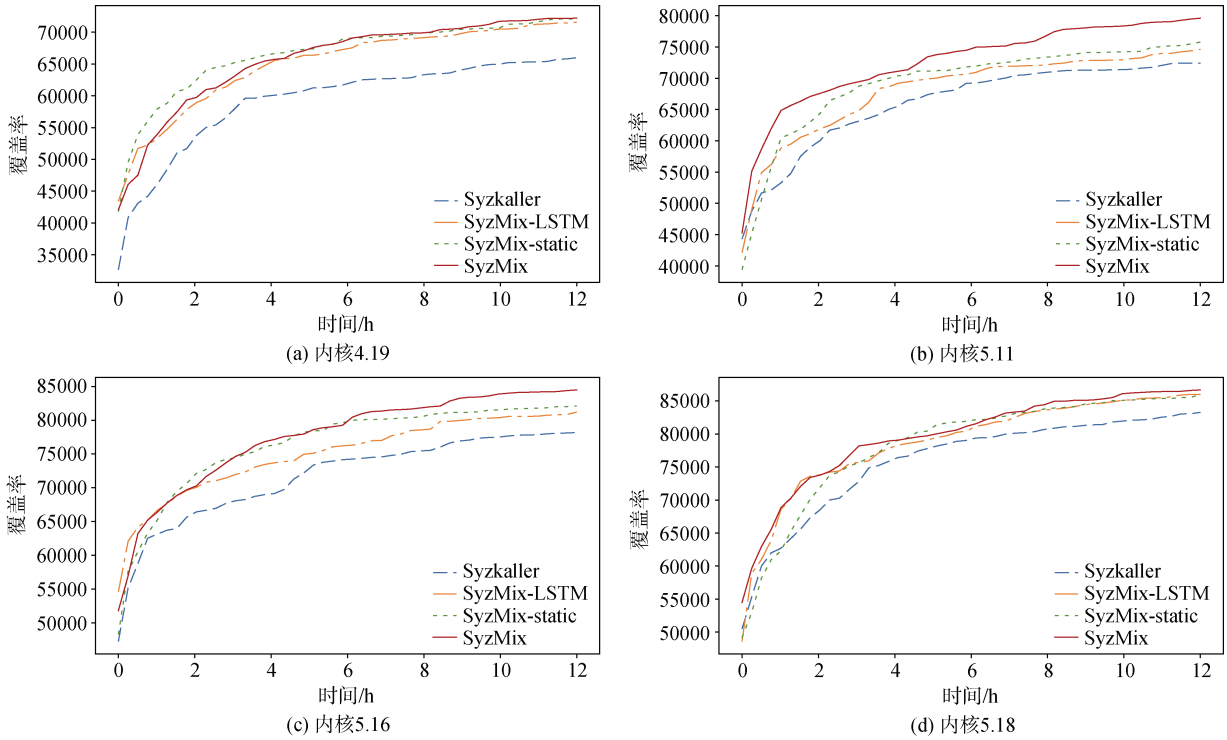

图 11　Syzkaller SyzMix-LSTM SyzMix-static SyzMix 覆盖率比较

Figure 11　Syzkaller SyzMix-LSTM SyzMix-static SyzMix coverage compare

明基于类型的静态分析对覆盖率的提升具有正向的效果; SyzMix 可分别提高 9.41%、9.90%、8.07%、4.08%, 平均可提高 7.87%。

4.5.2　代码覆盖加速比

代码覆盖加速比表示在达到相同的代码覆盖率时, SyzMix 相比 Syzkaller 可以节约的时间的比例。由于随时间增加, 覆盖率的提高缓慢, Syzkaller 在短时间内很难达到 SyzMix 在 12 h 时刻的覆盖率, 并且在实际测试过程中, 若以 12 h 为检测点, Syzkaller 在内核 4.19、内核 5.11 中在运行 48 h 后仍未达到对应的覆盖率, 所以把代码覆盖加速比的监测点设置为 SyzMix 在 6 h 时刻的覆盖率, 并记录 Syzkaller 需要多长时间可以达到这个覆盖率。

需要说明的是, 在计算代码覆盖加速比时, 没有把预先执行的时间以及模型训练时间进行折算, 实际的数值会小于下面给出的数据。但是由于不同内核要求输入种子的相似性, 神经网络训练出来的种子是具有可复用性的, 即在某一内核环境下训练出来的种子文件, 在其他版本的内核中依然有效。这种可复用性可以大幅降低训练的时间成本。

表 3　代码覆盖加速比

Table 3　Code coverage speed-up

内核版本	Syzkaller 所需要的时间	提高的效率
4.19	17h	+183.3%
5.11	15h30min	+158.3%
5.16	14h15min	+137.5%
5.18	9h	+50.0%
平均		+132.3%

表 3 中记录了 Syzkaller 要达到 SyzMix 在 6 h 时刻的覆盖率所需的时间, 在内核 4.19、内核 5.11、内核 5.16、内核 5.18 中, 分别需要 17 h、15h 30min、14h 15min、9h, 代码覆盖加速比可以达到 183.3%、158.3%、137.5%、50.0%, 平均代码覆盖加速比可以达到 132.3%。

4.6　漏洞发现能力

使用 SyzMix 对不同版本的内核进行模糊测试, SyzMix 能挖掘出新的 bug。表 4 列出了截止到本文写作时间, SyzMix 在内核 5.13、内核 5.15、内核 5.16 所发现的未知的 bug, 合计 8 个[①②], 包括 4 个 use-after-free、2 个 stack-out-of-bounds、1 个 null-ptr-

① https://bugzilla.kernel.org/buglist.cgi?email1=v.owl337%40gmail.com&emailreporter1=1&emailtype1=substring&query_format=advanced

② https://bugzilla.kernel.org/buglist.cgi?email1=6201613047%40stu.jiangnan.edu.cn&emailreporter1=1&emailtype1=substring&query_format=advanced

表 4 SyzMix 发现的 bug

Table 4 Bugs found by SyzMix

	序号	内核版本	类型
1	bug-213539	5.13-rc4	use-after-free
2	bug-213575	5.13-rc4	stack-out-of-bounds
3	bug-214463	5.15-rc-ksmbd-part2	use-after-free
4	bug-214655	5.15-rc-ksmbd-part2	use-after-free
5	bug-215405	5.16-rc6	use-after-free
6	bug-215421	5.16-rc4	bad-rss-counter-state
7	bug-215423	5.16-rc4	null-ptr-deref
8	bug-215517	5.16-rc4	stack-out-of-bounds

deref、1 个 bad-rss-counter-state。其中，在内核 5.15-rc-ksmbd-part2 中的 bug-214655 已经被确认并修复，并申请得到 CVE 编号 CVE-2021-45868[③]。

4.7 种子生成部分在 Healer 中的适用性

由于 Healer 和 SyzMix 都以 syzlang 格式的文件类型作为输入，本节将探究种子生成部分对 Healer 的适用性。需要说明的是，由于 SyzMix 使用 Go 语言实现，而 Healer 使用 Rust 实现，并且整体结构有较大差异，因此没有对种子执行部分进行移植。

Healer-LSTM 使用 LSTM 训练后得到的种子文件，而 Healer 使用相同数量的随机种子文件。相比 Healer，在内核 5.16、内核 5.18 执行 12 h 后，结果如图 12(a)、12(b)所示，Healer-LSTM 覆盖率可分别提高 4.77%、4.76%。本节实验可以说明种子生成部分在 Healer 上具有一定的适用性。

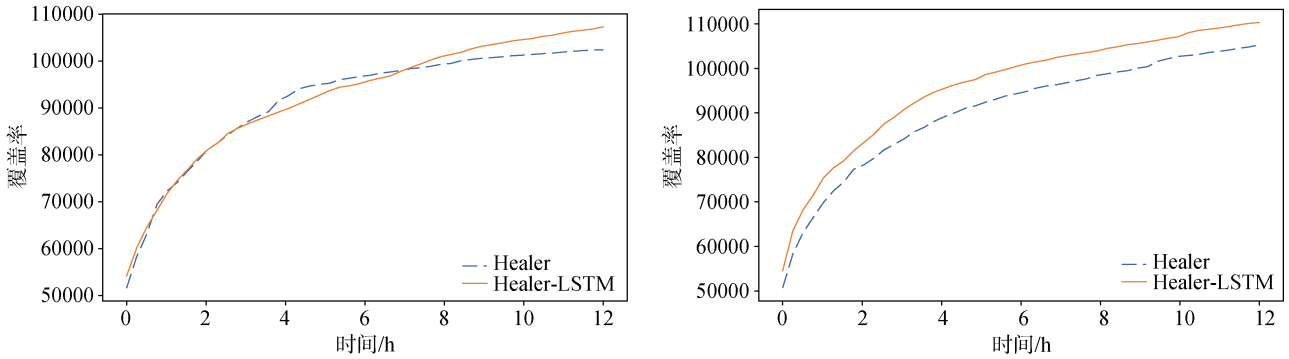

图 12 Healer、Healer-LSTM 覆盖率比较

Figure 12 Healer Healer-LSTM coverage compare

5 总结与讨论

本文介绍了内核相关的模糊测试技术，并在现有的框架 Syzkaller 上提出了一种基于种子智能生成的内核模糊测试模型 SyzMix。该模型引入 LSTM 神经网络，创建并使用了系统调用依赖关系表。实验结果表明，该模型相较于 Syzkaller，种子执行成功率更高，选择系统调用的准确性更好，效率更高，覆盖率更好，并且发现了 8 个新的 bug，申请得到 CVE 编号 CVE-2021-45868。

在下一步工作中，将使用更加先进的神经网络结构，对种子文件进行更加细粒度化的训练，提高种子执行的成功率，进一步提高代码覆盖率。

参考文献

[1] Linux Kernel History Report. https://www.linuxfoundation.org/wp-content/uploads/2020_kernel_history_report_082720.pdf. 2020.

[2] Testing Linux at a time. http://linux-test-project.github.io/. 2021.

[3] Linux Kernel Selftests. https://www.kernel.org/doc/html/v4.15/dev-tools/kselftest.html. 2021.

[4] Richard McNally, Ken Yiu, Duncan Grove. Fuzzing: The State of the Art. Technical report. Australia: DSTO Defence Science Organisation, DSTO-TN-1043. Jan. 2012.

[5] Michal Zalewski. American fuzzy lop: A novel type of compile time instrumention fuzzer. http://lcamtuf.coredump.cx/afl. May. 2015.

[6] Wang, D., Zhang, Z., Zhang, H., et al. SyzVegas: Beating kernel fuzzing odds with reinforcement learning[C]. *30th USENIX Security Symposium*, 2021: 2741-2758.

[7] Dmitry V yukov. Syzkaller: the next generation of kernel fuzzer. https://www.slideshare.net/Dmitry/syzkaller-the-next-gen-kernel-fuzzer. May. 2017.

[8] Dunk O. Applying OS Fuzzing Techniques to Unikernels. Technical report. BSc (Hons) Computer Science, University of Manchester. Aug.2021.

[9] Zhu X G, Wen S, Camtepe S, et al. Fuzzing: A Survey for Roadmap[J]. *ACM Computing Surveys*, 54(11s): 230.

[10] Lin G J, Wen S, Han Q L, et al. Software Vulnerability Detection Using Deep Neural Networks: A Survey[J]. *Proceedings of the*

① https://nvd.nist.gov/vuln/detail/CVE-2021-45868

IEEE, 2020, 108(10): 1825-1848.

[11] Godefroid P. Fuzzing[J]. *Communications of the ACM*, 2020, 63(2): 70-76.

[12] Ren Z Z, Zheng H, Zhang J Y, et al. A Review of Fuzzing Techniques[J]. *Journal of Computer Research and Development*, 2021, 58(5): 944-963.
(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.)

[13] Barkworth Ashley, Mcdonald Lucas, Ijaz Ul Haq Muhmmad. Survey of Software Fuzzing Techniques[M]. 2021.

[14] Liang H L, Pei X X, Jia X D, et al. Fuzzing: State of the Art[J]. *IEEE Transactions on Reliability*, 2018, 67(3): 1199-1218.

[15] Li H, Zhang C, Yang X, et al. Survey of OS Kernel Fuzzing[J]. *Journal of Chinese Computer Systems*, 2019, 40(9): 1994-1999.
(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): 1994-1999.)

[16] Oliver M. QEMU,Unicorn,Zelos,and AFL. https://insinuator.net/2020/07/qemu-unicorn-zelos-and-afl/. July.2020.

[17] ncc group. Project Triforce: Run AFL on Everything! https://www.nccgroup.trust/us/aboutus/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/. Jun.2016.

[18] Dava Jones. Trinity: A Linux Kernel System call fuzzer tester. http://codemonkey.org.uk/projects/trinity, May. 2016.

[19] Rao. Arvind. Linux Kernel System Call Fuzzing[D]. Masaryk University, 2018.

[20] Schumilo S, Aschermann C, Gawlik R, et al. KAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels[C]. *The 26th USENIX Conference on Security Symposium*, 2017: 167-182.

[21] Maier D, Radtke B, Harren B. Unicorefuzz: On the Viability of Emulation for Kernelspace Fuzzing[C]. *The 13th USENIX Conference on Offensive Technologies*, 2019: 8.

[22] Improving AFL's QEMU mode performance. https://abiondo.me/2018/09/21/improving-afl-qemu-mode/. 2018.09.21.

[23] Li J, Zhao B D, Zhang C. Fuzzing: A Survey[J]. *Cybersecurity*, 2018, 1(1): 6.

[24] Pailoor S, Aday A, Jana S. Moonshine: Optimizing OS Fuzzer Seed Selection with Trace Distillation[C]. *The 27th USENIX Conference on Security Symposium*, 2018: 729-743.

[25] Li D, Chen H. FastSyzkaller: Improving Fuzz Efficiency for Linux Kernel Fuzzing[J]. *Journal of Physics*: *Conference Series*, 2019, 1176: 022013.

[26] External usb fuzzing for linux kernel. https://github.com/google/syzkaller/blob/master/docs/linux/external_fuzzing_usb.md. 2019.

[27] Song D, Hetzelt F, Kim J, et al. Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints[C]. *The 29th USENIX Conference on Security Symposium*, 2020: 2541-2557.

[28] Tu X W, Wang X F, Gan S T, et al. Diskaller: Kernel Vulnerability Parallel Mining Model Based on Coverage Guidance[J]. *Journal of Cyber Security*, 2019, 4(2): 69-82.
(涂序文, 王晓锋, 甘水滔, 等. Diskaller: 基于覆盖率制导的操作系统内核漏洞并行挖掘模型[J]. *信息安全学报*, 2019, 4(2): 69-82.)

[29] Sun H, Shen Y H, Wang C, et al. HEALER: Relation Learning Guided Kernel Fuzzing[C]. *The ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021: 344-358.

[30] Xiaochen Zou, Guoren Li, Weiteng Chen, et al. SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel[C]. *31st USENIX Security Symposium*. 2022: 3201-3217.

王明义 于 2020 年在山东科技大学物联网专业获得学士学位。现在江南大学软件工程专业攻读硕士学位。研究领域为漏洞挖掘、系统安全。研究兴趣包括: 模糊测试、系统安全、漏洞挖掘等。Email: 6201613047@stu.jiangnan.edu.cn

甘水滔 博士, 长期从事系统安全和程序分析方法研究。Email: ganshuitao@gmail.com

王晓锋 于 2007 年在哈尔滨工业大学计算机系统与结构专业获得博士学位。现任江南大学教授。研究领域为网络仿真、网络安全。Email: wangxf@jiangnan.edu.cn

刘渊 江南大学工智能与计算机学院教授, CCF 高级会员, 博士生导师, 主要研究领域为网络安全, 数字媒体等。Email: lyuan1800@jiangnan.edu.cn