计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(9):19−31 [doi: 10.15888/j.cnki.csa.009218]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于系统调用序列学习的内核模糊测试①

张　阳[1,2], 范俊杰[1,2], 孙晓山[1,2], 张颖君[1,2], 程　亮[1,2]

[1](中国科学院大学, 北京 100049)
[2](中国科学院 软件研究所 可信计算与信息保障实验室, 北京 100190)
通信作者: 程　亮, E-mail: chengliang@iscas.ac.cn

摘　要: 操作系统内核是计算机系统中最基本的软件组件, 它控制和管理计算机硬件资源, 并提供访问和管理其他应用程序所需的接口和服务. 操作系统内核的安全性直接影响整个计算机系统的稳定性和可靠性. 内核模糊测试是一种高效、准确的安全漏洞检测方法. 然而目前内核模糊测试工作中, 存在系统调用间关系的计算开销过大且容易误判, 以及系统调用序列构造方式缺乏合理能量分配以至于很难探索低频系统调用的问题. 本文提出以 N-gram 模型学习系统调用间关系, 根据系统调用的出现频次信息和 TF-IDF 信息优先探索出现频次低或者 TF-IDF 值高的系统调用. 我们以极低的开销, 在 Linux 4.19 和 5.19 版本的 24 h 实验中分别提升了 15.8%、14.7% 的覆盖率. 此外, 我们挖掘到了一个已知 CVE (CVE-2022-3524)、8 个新崩溃, 其中一个获得了 CNNVD 编号 (CNNVD-2023-84723975).

关键词: 内核模糊测试; N-gram; TF-IDF; 系统安全; 系统调用

引用格式: 张阳,范俊杰,孙晓山,张颖君,程亮.基于系统调用序列学习的内核模糊测试.计算机系统应用,2023,32(9):19–31. http://www.c-s-a.org.cn/1003-3254/9218.html

Kernel Fuzzing Based on System Call Sequence Learning

ZHANG Yang[1,2], FAN Jun-Jie[1,2], SUN Xiao-Shan[1,2], ZHANG Ying-Jun[1,2], CHENG Liang[1,2]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)
[2](Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

Abstract: The operating system kernel is the most fundamental software component in a computer system. It controls and manages computer hardware resources and provides interfaces and services necessary for accessing and managing other applications. The security of the operating system kernel directly affects the stability and reliability of the entire computer system. Kernel fuzzing is an efficient and accurate security vulnerability detection method. However, in current kernel fuzzing work, the overhead of calculating the relationship between system calls is too high, or it is easy to misjudge the relationship between system calls. In addition, the existing method for constructing system call sequences lacks reasonable energy allocation, making it difficult to explore problems of low-frequency system calls. This study proposes to learn the relationship between system calls by using an N-gram model and prioritize the expansion of system calls with low frequency or high TF-IDF values based on the frequency and TF-IDF information of system call occurrences. With minimal overhead, this study achieves a coverage increase of 15.8% and 14.7% in 24-hour experiments on Linux versions 4.19 and 5.19, respectively. Besides, one known CVE (CVE-2022-3524) and eight new crashes are discovered, one of which is numbered CNNVD (CNNVD-2023-84723975).

Key words: kernel fuzzing; N-gram; TF-IDF; system security; system call

① 基金项目: 国家自然科学基金 (62072448)
收稿时间: 2023-02-17; 修改时间: 2023-03-14; 采用时间: 2023-03-30; csa 在线出版时间: 2023-07-14
CNKI 网络首发时间: 2023-07-17

1 引言

Linux 操作系统内核在高性能服务器以及 IoT 设备中普遍存在. 人们生活中所使用的网络服务、智能家居、工业中的工控系统都离不开操作系统内核的强力支撑.

2020 年一篇 Linux 内核报告[1] 指出, Linux 内核的发展离不开一些实用工具如 Sparse[2]、Strace[3] 以及一些持续性的内核测试工具如 Syzkaller[4]. 近年来内核 CVE[5] 缺乏标识[6], 依然很多内核代码缺陷亟待修复.

操作系统作为直接运行在计算机硬件之上的系统软件, 具有最高的内核级的运行权限, 为系统服务程序、应用软件提供基础服务, 其漏洞可以危害整个系统的安全性、稳定性, 带来难以估量的损失, 因此对操作系统进行充分测试, 排除具有安全隐患代码, 及时修复存在 bug 的代码是操作系统内核开发生命周期以及后期维护阶段一项重中之重的任务.

目前, 内核模糊测试是内核空间漏洞挖掘的主要方式. 在内核模糊测试中, 用户通过内核提供的接口发送大量精心构造的数据, 通过观察内核运行状态, 使用例如 KASAN 等工具捕捉各种异常以此判断内核中的潜在问题. 内核模糊测试从已知最早的 Tsys[7] 开始经历了从随机的内核模糊测试[8] 到类型感知的内核模糊测试[9–14] 再到基于覆盖率的内核模糊测试[8,15–17] 的发展过程. 内核模糊测试领域越来越多运用程序分析技术, 硬件特性以及仿真软件 QEMU[18] 辅助内核模糊测试[15,16] 的进行.

目前的内核模糊测试方案主要以系统调用为接口, 通过构造系统调用序列作为运行程序输入至内核执行. 程序运行过程中内核通过一些辅助工具如 KASAN、Kmemleak 等检测内核异常状态从而发现潜在 bug. Linux 内核提供了接近 400 个系统调用, 如果随机组合各种系统调用, 不仅搜索空间巨大, 而且将产生大量的无效输入. 系统调用间还存在一定的依赖关系, 这就意味着这些系统调用不能随意的组合, 而且一些特定的调用必须有前置依赖调用才能顺利执行从而达到更深的代码逻辑. 所以学习系统调用间关系, 构造更加容易被操作系统内核接收的系统调用序列、提高系统调用序列的构造质量是决定内核模糊测试取得更高效率的重要因素.

目前内核模糊测试的研究热点工具 Syzkaller[4] 通过专家知识编写的系统调用模版, 给予系统调用一定的语义信息. 然后 Syzkaller 通过分析系统调用的资源类型信息计算系统调用选择的权重得到系统调用选择矩阵 (由一个系统调用选择下一个系统调用的矩阵, 下文称选择矩阵), 通过选择矩阵随机构造系统调用序列.

Syzkaller 的选择矩阵的大小是根据可生成的系统调用个数决定的, 通常可生成的系统调用个数约为 2 000 个. 也就是在构造系统调用的过程中, 由某个系统调用扩展下一个系统调用的选择就有 2 000 个, 这个扩展空间较大而且按照此方式构造的调用序列往往无法通过内核检验通过执行. Syzkaller 通过动态修正选择矩阵, 可以增加实际通过内核运行的系统调用序列中系统调用的选择权重, 其修正方式是将出现在同一系统调用序列中的任意两两系统调用对应的选择矩阵元素的选择权重加一. 但是, 并非出现在已执行过的调用序列中的系统调用对含有依赖关系, 这种方式容易误判调用间的关系.

Syzkaller 的系统调用序列构造方式是首先随机挑选第 1 个系统调用, 然后从左往右顺序构造. 这种构造方式存在两个问题: 1) 随机构造第 1 个系统调用时, 每个系统调用都是被等概率选取的. 这个构造方式无法高效探索执行次数少的系统调用; 2) 当根据选择矩阵构造的系统调用序列能通过内核执行时会增加选择矩阵对应的权值, 当多次使用相同的第 1 个系统调用构造序列时, 容易构造重复的调用序列.

在目前与系统调用序列相关的研究中, Sun 等人的 Healer[19] 通过覆盖率变化学习有关系的系统调用, 并构建一个关系矩阵, 以矩阵值为 1 代表两个系统调用有关联. 然后该方法依据该关系矩阵生成调用序列, 但是, 通过覆盖率变化分析得到有关系的调用开销较大. Moonshine[20] 通过运行 Linux 测试集程序, 从真实程序中提取系统调用序列, 并尽可能精简这些系统调用序列, 该工作主要提供了 Syzkaller 运行的高质量初始种子. 杨鑫等人提出的 Dependkaller[21] 通过静态分析内核代码中共享内核数据信息的系统调用, 并通过程序信息赋予系统调用依赖权重, 该工作的静态分析容易产生误报且根据人为经验赋予权值不能客观反映调用间关系. FastSyzkaller[22] 通过 N-gram 模型分析触发崩溃的序列模式, 并得出触发崩溃的常见序列模式. 该工作对指导调用序列的生成并无修改.

现有工作中, 使用静态分析修正选择矩阵开销过

大且存在误报漏报, 使用动态执行序列分析调用间关系更加精确但也存在执行开销过大的问题. 目前能查阅到的内核模糊测试的文献资料中, 尚未有针对系统调用序列构造存在的问题做出相应修改的工作.

本文将使用 N-gram 模型计算 Syzkaller 运行中实际执行的语料库 (系统调用序列数据) 中的调用间关系信息, 通过概率模型获得更加准确的调用关系信息. 然后通过 N-gram 模型强化初始的选择矩阵, 最后通过随机游走建图的方式构造系统调用序列, 我们希望系统调用序列在满足局部的关系依赖的同时构造更加多样的系统调用序列, 从而提高覆盖率以及漏洞发现能力. 总体来说, 我们的设计有以下特点.

(1) 采用 N-gram 模型收集系统调用信息. 我们将系统调用的选择抽象为自然语言模型中的词语选择, 以实际运行的系统调用动态信息通过 N-gram 模型进行建模, 得到系统调用间的概率分布, 通过 N-gram 模型选择系统调用, 使得生成的系统调用序列更长从而达到更深的代码逻辑. 并且通过 N-gram 模型训练结果反馈到选择矩阵生成增强系统调用选择矩阵, 通过 N-gram 模型和增强的系统调用选择矩阵作为生成系统调用的依据.

(2) 采用随机游走方式生成调用序列. 我们根据 N-gram 模型以及增强的系统调用选择矩阵以随机游走的方式进行构建无向图, 然后使用拓扑排序从无向图中选择系统调用序列, 随机游走扩展系统调用根据系统调用的 IF-IDF 值按概率扩展. 保证构造序列在尽可能满足依赖关系的基础上尽可能“畸形”, 并且扩展更加“重要”的系统调用. 缓解 Syzkaller 缺少系统调用选择的能量分配导致构造相同覆盖率的调用序列带来的效率问题.

基于以上两点, 我们在 Syzkaller 的基础上实现了原型工具 Psyzkaller (Percentage Syzkaller), 在和 Syzkaller 工具的对比实验中, Psyzkaller 在 Linux 4.19 和 5.19 版本的 24 h 实验中分别提升了 15.8%、14.7% 的覆盖率. 此外, 我们挖掘到了一个已知 CVE (CVE-2022-3524)、8 个新崩溃, 其中一个获得了 CNNVD 编号 (CNNVD-2023-84723975).

本文第 1 节是引言. 第 2 节是背景, 主要讲述我们工作的基础工具 Syzkaller 的工作流程、系统调用序列构造的重要性以及 N-gram 模型. 第 3 节是方案设计, 主要讲述我们实现中 3 个模块的方法设计. 第 4 节是实现, 主要讲述我们工作的基础工具版本以及涉及的重要库文件. 第 5 节是实验评估, 主要从覆盖率、系统调用序列构造情况、漏洞发现能力以及额外时间开销 4 个方面进行工具有效性说明. 第 6 节是结束语.

2 背景

由于本文工作基于 Syzkaller 实现, 本节主要讲述 Syzkaller 的工作流程、系统调用序列构造重要性以及 N-gram 模型.

2.1 Syzkaller 的工作流程

Syzkaller 作为目前内核模糊测试研究的基准工具, 在 Linux、Android、OpenBSD、Windows 等操作系统中挖掘出的漏洞数以千计, 取得了一定的成果. 目前该工具依然在主流的内核版本持续模糊测试并将挖掘的漏洞公布在 syzbot[23] 中.

如图 1 展示了 Syzkaller 系统的程序结构, 红色字体代表了对应功能的配置选项. 整体来看, Syzkaller 分为两部分, 主机部分和虚拟机部分. 主机部分通过运行 syz-manager 来控制整个系统. syz-manager 可以创建虚拟机并通过 ssh 进行连接. 虚拟机部分通过 syz-fuzzer 生成和变异的系统调用序列作为内核的输入, syz-executor 将 syz-fuzzer 生成的系统调用序列交给内核执行. syz-fuzzer 收集系统调用执行的覆盖率信息并通过 RPC 的方式回传给 syz-manager 含有新覆盖的系统调用序列或者程序崩溃信息. syz-manager 将程序崩溃信息以及语料库信息存于本地数据库, 并开启了 Web 服务, 用户可以实时查阅 Web 界面查看模糊测试过程中的语料库信息、触发的崩溃等信息.

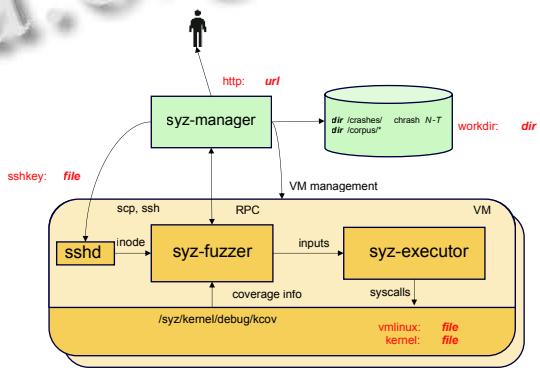

图 1 Syzkaller 流程图

2.2 系统调用序列构造的重要性

如下代码所示的是服务端程序建立网络通讯的系统调用序列, 该序列涉及 4 个系统调用. 其中 socket 系统调用用于建立一个套接字, 该套接字用于接下来

的网络通讯; bind 系统调用将该套接字绑定到一个地址, 并制定一个端口号; listen 系统调用, 使用该套接字监听连接请求; accept 系统调用用于请求到来时的处理流程.

```
sock_fd = socket(AF_INET, SOCK_STREAM, o)
bind((sock_fd, &addr, sizeof(addr))
listen(sock_fd, ···)
accept(sock_fd, &peer_addr, &size)
```

由于操作系统内核中存在“依赖”挑战[12], 系统调用间存在关联. 在这个例子中, 由于 socket 调用的输出是 bind、listen、accept 的输入, 我们称他们具有显式关系.

Syzkaller 会提供一个系统调用选择矩阵, 选择矩阵中元素的大小代表系统调用间的依赖强度, 也就是一个系统调用有多大的概率选择它的下一个系统调用. 例如, 当选取 socket 作为第一个系统调用时, Syzkaller 将根据代表 socket 调用的那行矩阵元素大小, 选取下一个系统调用. 在这个例子中, 需要构造完整的网络通讯的程序逻辑, 必须根据上述所示的调用序列进行构造. 选择有效的序列并建立有效的探索路径具有重要价值.

1) 合理构造序列将排除无效输入, 增加模糊测试效率. Linux 中的系统调用约有 400 个, 其中 Syzkaller 中专家知识编写的调用模版约有 5 000 个, 排列组合的方式构造序列搜索空间巨大.

2) 一个有效且精简过后的序列, 能构成更深的代码逻辑, 挖掘更深层次的代码缺陷.

2.3 N-gram 模型

N-gram 模型来源于自然语言处理领域, 主要用于评估一个句子的合理性. 内核模糊测试中, 通过构造系统序列进行测试. 而系统调调用间存在依赖关系, 系统调用序列的合理性影响模糊测试效率. 我们可以通过 N-gram 模型使用到序列构造上以满足调用间的依赖关系构造更长的序列.

这里我们将每个系统调用看作是一个词, 假设一个系统调用序列 S 由 T 个系统调用组成:

$$S = w_1, w_2, w_3, \cdots, w_{T-1}, w_T \tag{1}$$

一个系统调用序列出现的概率为:

$$\begin{aligned} p(S) &= P(w_1, w_2, w_3, \cdots, w_{T-1}, w_T) \\ &= \prod_{t=1}^{T} P(w_t | w_1, w_2, w_3, \cdots, w_{T-1}) \end{aligned} \tag{2}$$

根据马尔可夫假设, 第 n 个系统调用只和前 n–1 个系统调用有关, 则式 (2) 改进为:

$$\begin{aligned} p(S) &= P(w_1, w_2, w_3, \cdots, w_{T-1}, w_T) \\ &= \prod_{t=1}^{T} P(w_t | w_{t-n+1}, w_{t-n+2}, \cdots, w_{t-1}) \end{aligned} \tag{3}$$

以 bigram 为例, 也就是 n 为 2 的情况, 每个系统调用仅与前一个系统调用有关联, 此时:

$$p(S) = P(w_1, w_2, w_3, \cdots, w_{T-1}, w_T) = \prod_{t=1}^{T} P(w_t | w_{t-1}) \tag{4}$$

根据最大似然估计:

$$p(w_t | w_{t-1}) = \frac{C(w_t, w_{t-1})}{C(w_{t-1})} \tag{5}$$

结合式 (4) 和式 (5) 可得:

$$p(S) = \prod_{t=1}^{T} p(w_t | w_{t-1}) = \prod_{t=1}^{T} \frac{C(w_t, w_{t-1})}{C(w_{t-1})} \tag{6}$$

其中, $C(w_i)$代表系统调用w_i在语料库中出现的次数. 式 (6) 是我们运用 N-gram 模型的基础.

3 方案设计

如图 2 所示, 我们在 Syzkaller 的基础上实现了我们的原型工具 Psyzkaller, 其中黄色部分为原有工作, 蓝色部分为我们的增加或者改进的模块. 我们的改进主要包括 3 个模块: 1) 语料库分析模块负责根据语料库的动态执行信息在初始选择矩阵的基础上生成调用子序列作为训练集训练 N-gram 模型; 2) 选择矩阵增强模块负责将系统调用序列执行的动态信息反馈至初始选择矩阵得到增强的选择矩阵; 3) 调用序列生成模块负责根据 N-gram 模型以及增强的选择矩阵采用随机游走构造调用序列.

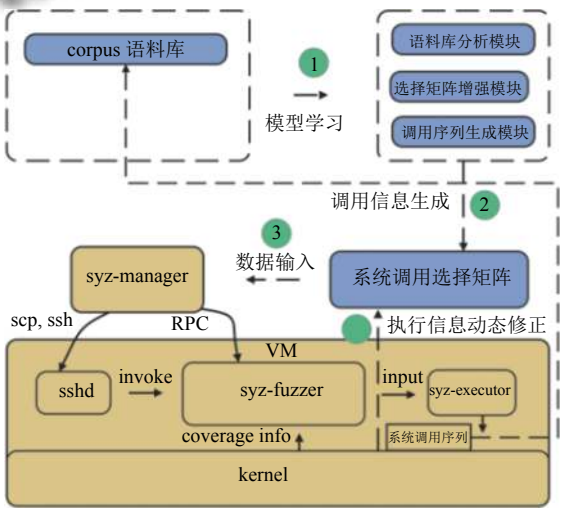

图 2 Psyzkaller 流程图

3.1 语料分析模块

语料库分析模块通过分析语料库中的调用序列训练 N-gram 模型. 语料分析模块由初始语料建图、深度遍历提取子序列以及 N-gram 模型训练 3 部分组成.

(1) 初始语料库建图

初始语料来自于 Syzkaller 种子程序的运行得到的系统调用序列. 我们首先提取调用序列初始选择矩阵中对应的子图. 以 F 代表系统调用序列, 长度为 T, w_{n_i}代表一个系统调用, 其中下标n_i是该调用在选择矩阵中的行 (列), M 为选择矩阵. 假设一个序列 F 为:

$$F = w_{n_1}, w_{n_2}, w_{n_3}, \cdots, w_{n_{T-1}}, w_{n_T} \tag{7}$$

则我们选取 M 的n_i行列组成的矩阵为 M_{sub}.

也就是在初始选择矩阵中提取调用序列每个系统调用表示的行和列. 得到的子图是初始选择矩阵的若干行列组成的邻接矩阵表示, 矩阵中元素信息为系统调用的选择权重. 初始选择矩阵包含着系统调用间的静态信息, 而系统调用序列是运行的动态信息, 我们提取调用序列对应的子图是为了将动静态信息结合并为下一步遍历工作做好数据基础.

(2) 深度搜索提取子序列

从选择矩阵中构建完成子图之后, 我们从系统调用序列的起始节点深度遍历该调用序列获得潜在子序列. 资源分析通过静态分析得到的系统调用间互相选择的权重. 这个分析结果代表系统调用间可能存在联系, 语料库中的系统调用序列代表真实通过运行的序列. 我们通过实际运行的系统调用序列遍历这个静态子图, 得到潜在的子序列. 通过语料库中系统调用序列数据遍历静态子图得到的子序列含有程序的动静态信息并且可以扩展数据集. 深度遍历搜索提取算法具体步骤如下.

1) 输入 F 为调用序列, M_{sub} 为由初始语料库建图得到的邻接矩阵, 用于判断系统调用节点间是否有可达的边.

2) 调用序列从未探索节点 *notvisit* 选取第一个系统调用节点 $syscall_i$ (算法 1 line 17).

3) 将选取的节点作为当前系统调用节点 *cur* (算法 1 line 4).

4) 探索路径 *path* 扩展当前系统调用节点 *cur*, 未探索节点集合 *notvisit* 去除扩展节点 (算法 1 line 6).

5) 如果当前系统调用节点无法扩展节点, 说明已经探索完成一条子序列, 将子序列存于 S (算法 1 line 8), 回到上一层递归函数 (算法 1 line 9).

6) 如果当前节点能继续扩展节点, 遍历 *notvisit* 节点 (剩余节点), 如果 $syscall_i$ 能够从当前节点到达, $syscall_i$ 作为选取节点 (算法 1 line 11). 进入递归 (算法 1 line 12).

7) 如果本层递归结束, 则进行回溯, 撤销处理结果 (算法 1 line 13、14).

8) 计算输出为潜在子序列集合 S.

伪代码如算法 1 所示.

算法 1. 深度遍历搜索提取序列

Input: 调用序列 F、子选择矩阵 M_{sub}.
Output: 潜在子序列 S.

```
1. Function recursion(syscall) //递归函数
2.     cur=syscall
3.     path.apppend(cur)
4.     notvisit=notvisit–cur
5.     if syscall don't have successor node
6.         S.append(path)
7.         return
8.     end if
9.     for syscall_i in novisit can be reached from cur
10.        recursion(syscall_i)
11.        path.erase(syscall_i)
12.        notvisit=notvisit+syscall_i
13.    end for
14. end Function
15. notvisit = F
16. path=∅; S=∅
17. for syscall_i in notvisit
18.     recursion(syscall_i)
19. end for
```

深度搜索提取子序列旨在实际运行的调用序列中, 挖掘子序列. 以图 3 为例, 假设我们有调用序列 1-2-3-4 (F), 提取的子图 (M_{sub}) 如图 3 所示. 则我们从系统调用结点 1 作为第一个节点开始在 M_{sub} 上深度遍历 (算法 1 line 17、18), 此时 *path* 为“1” (算法 1 line 2、3). 通过 for 循环遍历, 从节点 1 能到达的节点是节点 2 和节点 3, 我们首先选择节点 2, 进入递归函数, 此时 *path* 为“1-2” (算法 1 line 2、3). 以 2 为当前节点只能探索到节点 4 (算法 1 line 9), 所以以系统调用节点 4 进入递归函数后, 终止条件 (算法 1 line 5)满足, 得到一条子序列“1-2-4” (算法 1 line 6). 然后回溯至节点 2, *path* 为“1-2” (算法 1 line 11) 由于其可达节点 (4) 已经探索完, for 循

环结束, 所以继续回溯至节点 1, 此时 *path* 为“1” (算法 1 line 11). 接着从未探索的可达节点 3 开始探索, 同理可以探索“1-3-2”“1-3-4”两条子序列. 接着我们分别以 2、3、4 为第 1 个节点开始探索 (算法 1 line 17、18), 按照这样的规则, 我们提取的潜在子序列有“1-2-4”“1-3-4”“1-3-2-4”“2-4”“2-1-3-4”“3-2-4”“3-4”.

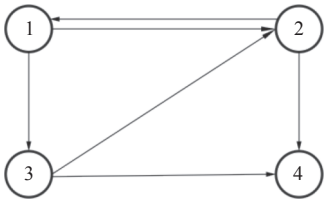

图 3 深度搜索示意图

(3) N-gram 模型训练

N-gram 模型来源于自然语言处理领域, 相关的背景知识在第 2 节中已经阐述. 由于在我们的使用场景中, 系统调用序列语料库中的数量为数万之内, 无法支撑复杂的深度学习模型以及 N 值较大的 N-gram 模型训练; 而且, 在我们的实验中, 语料库调用序列长度为 2 的调用序列占比最大, 我们采取 n=2 的 bigram 模型进行训练. N-gram 模型训练的具体算法如下.

1) 输入 S 为潜在子序列.

2) 遍历每个子序列.

3) 在每个子序列中, 提取相邻的两个系统调用, 并使用二维 hash 存储 (算法 2 line 5、7), 哈希表的两个维度分别是两个关联系统调用标号 (callnum)(Psyzkaller 对可生成的系统调用的标记), 数值大小为出现次数.

4) 语料库数据增加 100 条开始训练.

5) 计算输出为 bigram 模型 N.

伪代码如算法 2.

算法 2. N-gram 模型训练

Input: 潜在子序列 S.
Output: N //N-gram 模型.

```
1. N=∅ //N 为二维哈希表
2. for each s in S
3.     for i in range (1, s.length−1)
4.         if N[s[i].callnum][s[i−1].callnum]==0
5.             N[s[i].callnum][s[i−1].callnum]=1
6.         else
7.             N[s[i].callnum][s[i−1].callnum]++
8.         end if
9.     end for
10. end for
```

通过 N-gram 模型的训练, 我们得到强相关的调用序列对以及某个系统调用选择下一个的概率, 根据大数定律, 出现次数足够多的调用对的依赖关系可能更强. 在构造系统调用序列的过程中, 我们根据模型训练得到的数据依概率选取系统调用, 更容易构造出长序列.

3.2 选择矩阵增强模块

选择矩阵增强模块的目的在于将系统调用序列动态信息和初始的选择矩阵相结合, 使得选择矩阵包含系统调用的静态信息以及其运行的动态信息, 更加客观合理地反映系统调用间的关系. 选择矩阵增强的算法具体步骤如下.

1) 输入 N 为 N-gram 模型, 输入 M 为选择矩阵.

2) 遍历 N-gram 模型的第 1 个维度 $N[i]$, i 代表系统调用标号, $N[i]$ 代表与 i 系统调用相关的所有系统调用集合.

3) 如果 N-gram 模型中, i、j 系统调用有关联, 也就是 $N[i][j]$ 不为零, 则 *sum* 对应增加初始矩阵的权值 $M[i][j]$ (算法 3 line 5、6).

4) 增加出现 i、j 系统调用的选择权值, 对于 $M[i]$ 中的值, 如果 $N[i][j]$ 有值, 则 $M'[i][j]$ 计算为 $M[i][j]$+*sum**($N[i][j]$/$N[i][*]$), 其中 $N[i][*]$ 为 $N[i]$ 行的数值之和 (算法 3 line 10、11).

5) 计算输出为 M'.

伪代码如算法 3 所示.

算法 3. 选择矩阵增强

Input: N-gram 模型 N、选择矩阵 M.
Output: 增强矩阵 M'.

```
1. for n_i in N //n_i 为系统调用标号 i 所在哈希表 n_i=N[i]
2.     M[i]←M //M 中系统标号为 i 的行
3.     sum=0
4.     for j in range len(n_i)
5.         if N[i][j]!=0
6.             sum+=M[i][j]
7.         end if
8.     end for
9.     for val in range M[i]
10.        if N[i][j]!=0
11.            M'[i][j]=val+sum*(N[i][j]/N[i][*])
12.        end if
13.    end for
14. end for
```

由于我们采用了 N-gram 模型关注于相邻调用的关系, 原系统的增加调用序列任意两个调用间的动态更新选择矩阵的方案不再适用. 我们采用 N-gram 模型

信息更新选择矩阵. 我们的增强算法保障无 N-gram 模型数据的调用的相对权重不变 (算法 3 line 10), 此外我们根据 N-gram 模型数据再分配调用的选择权重 (算法 3 line 11). 该算法经模拟迭代验证能将选择矩阵中的系统调用的选择权重收敛至该调用对应的 N-gram 权重大小.

3.3 调用序列生成模块

TF-IDF (term frequency-inverse document frequency) 值的大小代表一个系统调用的“关键性”, 其在语料库中越少出现其值就越高, 而一个在语料越少出现的系统调用, 从该调用扩展将更可能增加覆盖率. 而采取随机游走的方式, 能产生更多样且“畸形”的调用序列.

使用增强选择矩阵一定程度上保障系统调用满足依赖条件, 模糊测试往往需要较为“畸形”的输入. Syzkaller 中采取随机生成第 1 个系统调用, 往后逐个构造的方式构造调用序列, 这种方式容易选择相同的第 1 个调用, 从前往后生成调用序列容易产生重复的调用序列. 因此我们在调用序列生成部分做了一些修改. 首先我们按照语料库出现系统调用的情况, 首先选择语料库更少出现的系统调用, 然后随机游走建图, 扩展 TF-IDF 值更高的系统调用. 最后拓扑排序生成具有依赖关系的一个调用序列.

原始 Syzkaller 随机选取第 1 个系统调用, 然后向后生成随机的系统调用. 随机选取第一个系统调用缺乏能量分配策略. 当其使用某个系统调用作为第 1 个系统调用构造序列并长时间的 fuzz 后, 该系统调用相关的覆盖率将趋于饱和. 因此我们统计每个系统调用在语料库出现的次数, 将更多的能量赋予更少出现的系统调用中. 从前往后生成的构造方式中, 容易生成相同的调用序列, 所以我们采取随机游走的方式构建系统调用序列.

我们的工作不是完全的随机游走, 构造序列过程中, 我们计算构造序列中的 TF-IDF 权值, 以下公式 w_i代表一个系统调用, $|S|$代表语料库中所有的序列个数, 其计算公式如下:

$$TF_i = \frac{w_i}{\sum_{t=1}^{T} w_t} \tag{8}$$

TF 为某个系统调用在本次构造中出现的频率:

$$IDF_i = \log \frac{|S|}{|\{w_i \in S_i\}+1|} \tag{9}$$

IDF 为该系统调用在整个语料库中出现的逆频率值. TF-IDF 权值更大代表该系统调用在整个语料库中出现更少, 更可能是比较“关键”的系统调用将有更大的概率被扩展.

调用序列生成的具体步骤如下.

1) 输入 N 为 N-gram 模型, M'为增强选择矩阵, L 为构建序列长度.

2) 根据每个系统调用在语料库中的出现次数, 按概率选择第 1 个更少出现系统调用, G 扩展该调用 (算法 4 line 3), 其中 G 中节点表示系统调用, 其有向边表示两个系统调用存在选择关系.

3) 计算 G 中调用序列的 TF-IDF 值, TF-IDF 值更高, 代表该系统调用 C'在所有语料库中更重要, 将有更大概率被扩展 (算法 4 line 6).

4) 选择该调用前向或者后向进行插入 (算法 4 line 7), 根据$\varepsilon = 0.3$的概率使用增强选择矩阵 M' (算法 4 line 8), 0.7 的概率使用 N-gram 模型扩展 (算法 4 line 9). 扩展得到扩展的调用 C'', 调用 C''导入 G 中. 重复步骤 3).

5) 将 G 拓扑排序, 生成长度为 L 的系统调用序列 P (算法 4 line 11).

6) 计算输出为 L 长度的系统调用序列 P.

伪代码描述如算法 4 所示.

算法 4. 调用序列生成算法

Input: N-gram 模型 N、增强选择矩阵 M'、构建序列长度 L.
Output: 系统调用序列 P.

```
1. ε=0.3
2. list=∅
3. choose first system C call with energy and G append C
4. list.append(C), G.append(C)
5. while(list.size<L)
6.     select one call C' in G by TF-IDF value
7.     choose a direction forward or backward to append C' and G append C'
8.     it has the ε probability to use the M' to append the G
9.     it has the 1−ε probability to use the N to append G
10. end while
11. P = TopoSort(G) //拓扑排序
```

我们通过 N-gram 模型训练得到出现更频繁的系统调用对信息. 这个信息在生成长系统调用序列时更有优势, 如在生成第 2.2 节所示的网络通讯系统调用时, 使用 N-gram 模型扩展将更大概率生成该系统调用序列, 探索更深层次的内核代码逻辑.

4 实现

基于 Syzkaller 的 5bc3be51 提交版本实现了 Psyzkaller, 总共编写约2 200 行的 go 代码实现上述算法, 引用了 go 语言实现的 TF-IDF 库[24].

5 实验评估

为了验证改进工作的有效性, 我们通过 Psyzkaller 和 Syzkaller 在 Linux 内核 4.19、5.19 版本的覆盖率、语料库中的系统调用序列分布情况、触发崩溃个数、以及 Psyzkaler 的额外时间开销来说明我们工具的有效性. 我们的实验评估主要回答以下 4 个问题.

1) Psyzkaller 的覆盖率如何?

2) Psyzkaller 系统调用序列的构造情况如何?

3) Psyzkaller 的漏洞发现能力如何, 有无发现真实的漏洞?

4) Psyzkaller 的额外时间开销如何?

5.1 实验设置

我们实验运行在 Intel(R) Xeon(R) Gold 5218 @ 2.30 GHz 的 CPU、128 GB 内存、Ubuntu 20.04 的环境中, Psyzkaller 以及 Syzkaller 在相同内核配置下运行 24 h 三次取平均整值作为实验数据. 由于与我们工作强相关的工作中, Dependkaller 并未提供源码, Moonshine 在种子精简上做工作, Healer 基于旧版本的 Syzkaller 且缺乏更新, 经我们测试其实验效果并不我们选取的的基准 Syzkaller 效果好, 所以我们的工作只和 Syzkaller 做比较. 由于 5.19 是 5.x 向 6.x 过渡的测试版本, 5.x 的旧版本特性和 6.x 的新特性都可以在该内核代码中有所体现, 所以我们选取 5.19 版本的内核作为测试内核. 为了说明我们的工具的相同时间内能触发更多的内核崩溃, 我们选取 4.19 版本内核作为测试内核.

5.2 覆盖率对比

Syzkaller 通过将程序运行的 trace 中的相邻的 PC (programer counter) 值也就是程序运行经过的相邻地址做个哈希作为边覆盖信息. 我们沿用了这个边覆盖统计信息. 在 24 h 实验中, 如图 4、图 5 所示, Psyzkaller 的边覆盖率对比 Syzkaller 在内核 4.19、5.19 版本中分别提升了 15.8%、14.7% 的覆盖率. 其中在 5.19 版本增加的覆盖率主要分布在 net 文件中当中, 表 1 展示这些覆盖率差额.

我们进一步探索了 Linux 内核 net 模块已覆盖部分的调用序列情况, 其调用序列长度多为 2 个及以上. 构造 2 个调用长度以及长序列是使用 N-gram 模型扩展系统调用的优势.

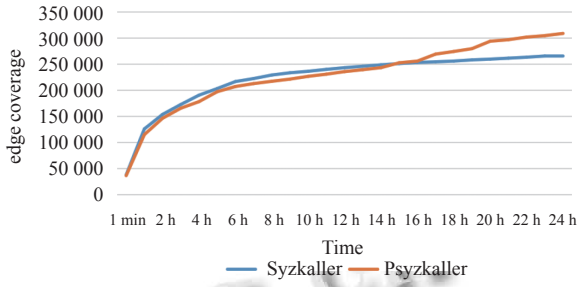

图 4 Psyzkaller 与 Syzkaller 在 Linux 内核 4.19 版本覆盖率对比

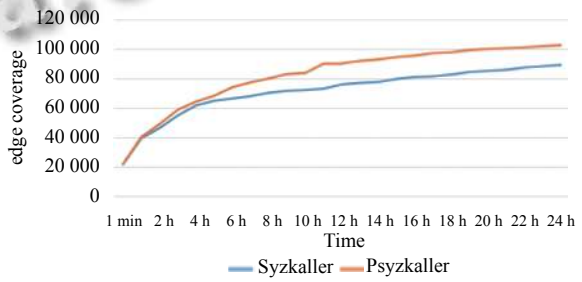

图 5 Psyzkaller 与 Syzkaller 在 Linux 内核 5.19 版本覆盖率对比

表 1 Psyzkaller 与 Syzkaller 覆盖率差异对比

文件名称	Syzkaller覆盖占比 (%)	Psyzkaller覆盖占比 (%)	边总数	差额 (取整)
net/core	24	27	19 171	575
net/ipv4	27	32	24 170	1 208
net/ipv6	36	38	16 350	327

5.3 系统调用序列分布情况

内核模糊测试会将触发新覆盖或者触发崩溃的系统调用序列加入语料库, 精简后的语料库的调用序列信息能说明 Psyzkaller 通过序列学习的方式构造序列的有效性. 我们以在 Linux 5.19 模糊测试得到的语料库为例, 图 6 为系统调用序列长度分布情况, 其长度分布在 1–20 之中, 其中长度为 2 的系统调用序列最多.

Psyzkaller 生成了系统调用序列 8 254 个, Syzkaller 生成有效序列 6 860 个, Psyzkaller 生成的系统调用序列高于 Syzkaller 约 20.3%. 由于 Psyzkaller 将调用序列及其参数信息作哈希统计序列个数, 相同的调用序列不同的参数会被视为不同的序列, 我们的方法没有对参数构造部分进行修改, 因此, 我们统计构造参数无关的序列个数, 如表 2 所示. 在我们的实验中, Psyzkaller 生成的参数无关的序列个数为 4 707 个, 比 Syzkaller

多约 26.5%. 可以看出我们的工具通过系统调用学习的方式构造系统调用序列取得了一定的效果.

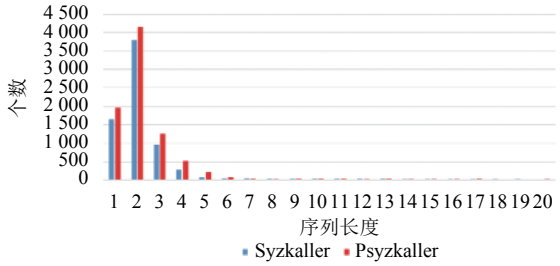

图 6 调用序列长度分布图

表 2 Psyzkaller 与 Syzkaller 调用序列总数对比

工具	序列个数	参数无关序列个数	参数无关序列个数占比 (%)
Syzkaller	6 860	3 720	54.2
Psyzkaller	8 254	4 707	57.0

FastSyzkaller[22] 指出, 5 个系统调用长度的调用序列是最容易触发崩溃的模式, 因此我们以 5 个系统调用为界, 视 5 个及以上系统调用序列为长序列. 我们的实验数据中 (表 3), Psyzkaller 生成了 381 个长序列, 比 Syzkaller 生成的长序列多约 101.6%, 可以看出 Psyzkaller 使用序列学习的方式在生成长序列方面占有优势, 能够探索更深的程序逻辑.

表 3 Psyzkaller 与 Syzkaller 长序列对比

工具	长序列个数
Syzkaller	189
Psyzkaller	381

5.4 漏洞发现能力

我们以相同时间内在相同版本内核中触发的崩溃数量以及是否发现新崩溃作为漏洞发现能力的度量标准. syzbot 是 Syzkaller 进行持续测试且汇报崩溃的平台, 使用 Syzkaller 相关工具的安全研究者在挖掘内核漏洞后将漏洞报送给内核相关维护者的同时会将 bug 崩溃信息抄送至 Syzkaller 讨论社区, 通过检索 syzbot、Syzkaller 社区邮件以及在 CVE 查询相关内核漏洞描述信息可以判断 Psyzkaller 触发的崩溃是否是最新的. Psyzkaller 发现了一个已知 CVE、8 个新崩溃, 我们将这 8 个崩溃信息提交给内核维护者的同时也与 syzbot 发布的已有内核 CVE 的崩溃信息进行比对, 确认它们都是新的. 我们对 8 个新崩溃中一个完成分析且稳定触发的漏洞提交至国家信息安全漏洞库并获得了 CNNVD 编号, 该漏洞已经完成研判并被评级为中危. 由于操作系统内核的复杂性, 相同的系统调用序列以及调用参数在不同的内核状态下运行会有不一样的运行结果, 这让复现崩溃构造 PoC 变得更加困难, 所以我们挖掘的 8 个新崩溃中, 7 个新崩溃目前在构造 PoC 阶段, 如表 4 所示.

表 4 Psyzkaller 挖掘的崩溃

内核版本	崩溃描述	漏洞状态	备注
5.19	memory leak in ipv6_flowlabel_opt	获CNNVD编号, 通过研判待披露	全新, CNNVD-2023-84723975, 评级中危
5.19	memory leak in ipv6_renew_options	已知CVE	CVE2022-3 524
5.19	memory leak in taskstats_exit	构造PoC	新崩溃
5.19	memory leak in copy_nid_process	构造PoC	新崩溃
5.19	memory leak in ipv6_sock_ac_join	构造PoC	新崩溃
5.19	Bug: soft lockup in e1000_watchdog	构造PoC	新崩溃
5.19	Bug: unable to handle kernel NULL pointer dereference in io_file_get normal	构造PoC	新崩溃
5.19	Bug: unable to handle kernel NULL pointer dereference in kernfs_get_inode	构造PoC	新崩溃
5.19	Bug: soft lockup in process_one_work	构造PoC	新崩溃

这里我们以我们提交的 CNNVD-2023-84723975 为例子, 展示该漏洞的调用栈. 其中$KERNEL 代表 Linux 主目录.

```
ipv6_flowlabel_opt ( $KERNEL/net/ipv6/ip6_flowlabel.c)
do_ipv6_setsockopt.constprop.0 ($KERNEL/net/ipv6/ipv6_sockglue.c)
ipv6_setsockopt ($KERNEL/net/ipv6/ipv6_sockglue.c)
sock_common_setsockopt ($KERNEL/net/core/sock.c)
__sys_setsockopt ($KEENEL/net/socket.c)
__x64_sys_setsockopt ($KERNEL/net/socket.c)
do_syscall_64 ($KERNEL/arch/x86/entry/common.c)
entry_SYSCALL_64_after_hwframe ($KERNEL/arch/x86/entry/
entry_64.S)
```

该崩溃的调用序列如下所示.

```
r0 = socket$inet6()
bind$inet6(r0, ···)
setsockopt$inet6_IPV6_FLOWLABEL_MGR(r0, ···)
setsockopt$inet6_group_source_req(r0, ···)
```

该例子和第 2.2 节中所示例子类似, 这些系统调用间具有较强关联关系, 构造序列时如果使用增强选择

矩阵拓展能强化出现的两两系统调用的选择权重, 如果根据 N-gram 模型拓展调用将有更大概率生成出该调用序列并更快到达变异环节得到崩溃.

在 Linux 4.19 版本 24 h 的模糊测试过程中, Psyzkaller 触发了 42 个崩溃, Syzkaller 触发了 35 个崩溃. 表 5 显示了部分崩溃的具体信息.

Psyzkaller 在 5.19 版本挖掘了 1 个已知 CVE, 8 个新崩溃, 其中一个新崩溃成功获得 CNNVD 编号并被评级为中危; 在 4.19 版本 24 h 的模糊测试过程中能触发更多的崩溃.

表 5 Psyzkaller 和 Syzkaller 24 h 在 Linux4.19 触发的崩溃 (节选)

崩溃描述	Syzkaller	Psyzkaller
KASAN: slab-out-of-bounds Read in vcs_write	√	√
KASAN: slab-out-of-bounds Read in vgacon_scrolldelta	×	√
KASAN: slab-out-of-bounds Write in vgacon_scroll	√	√
KASAN: stack-out-of-bounds Read in unwind_next_frame	×	√
KASAN: use-after-free Read in get_work_pool_id	×	√
KASAN: use-after-free Read in n_tty_receive_buf_common	√	√
KASAN: use-after-free Read in screen_glyph_unicode	√	√
KASAN: use-after-free Read in vc_do_resize	√	√
KASAN: use-after-free Read in vc_uniscr_check	√	√
KASAN: use-after-free Read in vcs_write	×	√
KASAN: use-after-free Write in con_shutdown	√	√
KASAN: use-after-free Write in do_con_write	√	√
WARNING in ext4_xattr_set_entry	×	√
WARNING in generic_make_request_checks	√	√
WARNING in loop_add	√	×
WARNING in netlbl_cipsov4_add	√	√
WARNING in notify_change	×	√
WARNING in restore_regulatory_settings	√	√
WARNING in vc_uniscr_alloc	√	√
WARNING in xfrm_policy_insert	√	√
WARNING: can't dereference registers at ADDR for ip interrupt_entry	√	√

5.5 额外时间开销

我们引入了 3 个模块, 需要计算系统调用序列的 TF-IDF 值, 训练 N-gram 模型, 以及以不同的构造方式构造系统调用. 本节将说明这些工作的时间开销. 我们从 Linux 4.19 进行模糊测试得到的语料库中随机抽取 100、500、10000、20000 个语料库作为实验数据.

Syzkaller 的动态修正方案首先遍历语料库数据, 然后使用两层 for 循环增加任意两两系统调用序列的权重. 其时间开销如表 6 所示. 随着料库数量的增加, 其算法时间复杂度开销差别为毫秒级别.

表 6 Syzkaller 动态修正时间开销 (s)

实验次数	语料库大小		
	500	10000	20000
1	0.152 0	0.157 4	0.152 0
2	0.153 2	0.148 1	0.145 8
3	0.170 5	0.182 2	0.164 2
4	0.128 2	0.122 1	0.159 4
5	0.127 0	0.126 7	0.124 7
平均值	0.146 2	0.147 3	0.149 2

表 7 展示了语料库大小分别为 500、10000、20000 统计计算 TF-IDF 值的时间开销情况. 可以看出尽管语料大小为 20000, TF-IDF 计算的开销也只有 2 s 左右.

表 7 TF-IDF 计算开销 (s)

实验次数	语料库大小		
	500	10000	20000
1	0.76	3.24	3.19
2	0.71	0.76	0.87
3	0.73	0.73	0.74
4	0.70	0.73	4.90
5	0.68	0.80	0.76
平均值	0.72	1.10	2.1

由于我们训练 N-gram 模型每新增 100 个语料进行计算, 所以这里我们统计 100 个语料库的训练时间开销. 表 8 实验数据显示我们 N-gram 模型增量训练开销非常小.

从 Syzkaller 的动态修正方案的时间开销, 以及我们更改该方案引入的 TF-IDF 计算以及 N-gram 模型增量开销, 可以看出, 我们 Psyzkaller 改进方案所使用的

额外时间开销很小. 由于我们改变了系统调用构造方式, 表 9 展示了在语料库大小为 20 000 的信息基础上, 使用 Syzkaller 构造方案以及 Psyzkaller 构造方案在生成长度为 2、12 的时间对比情况. 根据系统调用长度分布图, 长度为 2 的调用序列占比最高, 我们将长度 2 以及图 6 所示长序列分布的中位数 12 分别作为最常见的序列以及长序列的代表进行实验.

表 8 100 个语料库下 N-gram 模型增量训练开销 (s)

实验次数	时间开销
1	0.025
2	0.039
3	0.039
4	0.038
5	0.039
平均值	0.036

表 9 构造系统调用序列时间开销 (ms)

工具实验次数	Syzkaller		Psyzkaller	
	2	12	2	12
1	0.068	0.577	0.053	0.417
2	0.041	0.329	0.046	0.412
3	0.054	1.050	0.050	1.084
4	0.107	0.725	0.051	0.442
5	0.028	0.137	0.043	2.402
平均值	0.060	0.564	0.050	0.951

实验数据表明, 我们的构造方式在短序列的构造方式上相差不大, 长序列的构造时间开销虽然接近原方案开销的 2 倍, 但其开销成本也只多了约 0.4 ms. 综合以上 Psyzkaller 的开销实验说明, Psyzkaller 的额外开销很小, 完全不影响模糊测试进程.

6 相关工作

目前, 国内外很多工作基于 Syzkaller 展开. Sun 等人的 Healer[19] 通过覆盖率变化学习有关系的系统调用, 并通过构建一个关系矩阵, 以矩阵值为 1 代表两个系统调用有关联, 并通过该关系矩阵生成调用序列, 达到了一定的效果. Moonshine[20] 通过运行 Linux 测试集程序, 从真实程序中提取系统调用序列, 并尽可能精简这些系统调用序列, 为 Syzkaller 提供了初始的种子序列达到了覆盖率提升的效果. 杨鑫等人提出的 Denpendkaller[21] 通过静态分析内核代码中共享内核数据信息的系统调用, 并通过程序信息赋予系统调用依赖权重相较于 Syzkaller 提升了覆盖率以及漏洞挖掘能力. FastSyzkaller[22] 通过 N-gram 模型分析触发崩溃的序列模式, 并得出触发崩溃的常见序列模式.

除此以外, DIFUZE[25] 通过提取内核驱动信息, 在 7 个安卓系统上发现了 32 个之前的未知漏洞; Razzer[26] 通过修改 Hypervisor 以及加入断点旨在发掘竞争漏洞 SLAKE[27] 通过动静态结合分析内核重要数据结构, 并利用 slab 内存分配器挖掘内核漏洞; HFL[28] 通过模糊测试结合符号执行进行内核模糊测试; Kafl[17] 借助因特尔的 CPU 硬件特性进行模糊测试; IntellliGen[29] 通过自动化的方法构建驱动测试; Diskaller[30] 通过覆盖率引导以并行方式进行漏洞挖掘从而提升效率; Triforce-AFL[15] 在 AFL 的基础上使用 QEMU 工具对系统进行全模拟, 通过在 QEMU 模拟的 CPU 中一个特殊的指令 aflCall(0f24) 实现对操作系统进行模糊测试; TriforceLinuxSyscallFuzzer[31] 在 TriforceAFL 的基础上对系统调用进行模糊测试. 施鹤远等人[32] 详细介绍了使用覆盖率引导的 Fuzzing 技术来测试企业级 Linux 内核的实践经验, 包括测试框架的设计与实现、测试用例的生成和筛选、测试效果的评估和反馈等方面. Krace 是一项针对内核文件系统的数据竞争模糊测试技术的研究工作. Krace[33] 利用一种名为“Syscall Fuzzing”的技术来生成用于测试文件系统的系统调用序列, 并使用一种名为“K-Mutation”的技术来对这些系统调用进行变异, 以生成不同的测试用例. 此外, Krace 还使用了一种名为“事件排序”的技术来控制测试用例的执行顺序, 以便尽可能多地发现数据竞争问题. GREBE[34] 通过 LLVM 静态分析内核数据结构, 以及污点分析 syzbot 中的 60 个含有 PoC 的漏洞, 探索了这些 PoC 附近的程序漏洞, 提高了原有漏洞的可利用性. Dependency[12] 通过改写内核模糊测试工具 Syzkaller 和静态分析工具 Dr.Check 对内核的 4 个模块进行模糊测试, 然后通过人工分析出了造成内核模糊测试依赖挑战的 5 个根本原因.

7 结束语

操作系统内核空间的稳定安全是维护网络空间安全稳定的一项重中之重的内容. 内核模糊测试是一种高效的漏洞挖掘工具, 能够精准发现内核空间的潜在安全隐患. 本文设计了一种基于系统调用序列学习的模糊测试方案. 通过真实运行的系统调用序列数据训练 N-gram 模型, 挖掘系统调用关系, 并通过该信息指

导生成系统调用序列. 整体来看, 我们的工作在 4.19 和 5.19 的内核版本中分别提升了 15.8%、14.7% 的覆盖率, 发现了 8 个新崩溃. 内核模糊测试仍然存在许多工作值得研究, 将来的工作中, 我们将进行系统调用序列构造与参数选取相关的研究, 进一步提升内核模糊测试的效率以及效果.

参考文献

1 Open Sistemas. Linux kernel history report. https://opensistemas.com/en/2020-linux-kernel-history-report/. (2023-02-20)[2023-03-19].

2 kernel.org. Sparse document. https://sparse.docs.kernel.org/. (2021-09-06)[2023-03-19].

3 Sourceforge.net. Smatch project website. http://smatch.sourceforge.net/. (2021-10-21)[2023-03-19].

4 Google. Syzkaller. https://github.com/google/syzkaller. (2023-02-24)[2023-03-19].

5 Vulnerability trends over time. https://www.cvedetails.com/product/20550/Canonical-Ubuntu-Linux.html?vendor_id=4781. [2023-03-19].

6 Ruohonen J, Rindell K. Empirical notes on the interaction between continuous kernel fuzzing and development. Proceedings of the 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). Berlin: IEEE, 2019. 276–281.

7 Le T. Tsys, similar to crashme, alpha release. https://groups.google.com/g/alt.sources/c/V_B37EtnWKQ/m/NztsljVYV84J. (1991-09-21)[2023-03-19].

8 van Sprundel I. Fuzzing: Breaking software in an automated fashion. https://events.ccc.de/congress/2005/fahrplan/attachments/582-paper_fuzzing.pdf. (2005-12-08).

9 Kleen A, Jones D. Trinity. https://github.com/kernelslacker/trinity. [2023-03-19].

10 Garn B, Simos DE. Eris: A tool for combinatorial testing of the Linux system call interface. Proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation Workshops. Cleveland: IEEE, 2014. 58–67.

11 Gauthier A, Mazin C, Iguchi-Cartigny J, *et al*. Enhancing fuzzing technique for OKL4 syscalls testing. Proceedings of the 6th International Conference on Availability, Reliability and Security. Vienna: IEEE, 2011. 728–733.

12 Hao Y, Zhang H, Li GR, *et al*. Demystifying the dependency challenge in kernel fuzzing. Proceedings of the 44th IEEE/ACM International Conference on Software Engineering. Pittsburgh: IEEE, 2022. 659–671.

13 Koopman P, Sung J, Dingman C, *et al*. Comparing operating systems using robustness benchmarks. Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems. Durham: IEEE, 1997. 72–79.

14 taviso. iknowthis. https://github.com/taviso/iknowthis. (2019-04-18)[2023-03-19].

15 nccgroup. TriforceAFL. https://github.com/nccgroup/TriforceAFL. (2017-05-31)[2023-03-19].

16 oracle. kernel-fuzzing. https://github.com/oracle/kernel-fuzzing. (2016-10-14)[2023-03-19].

17 Schumilo S, Aschermann C, Gawlik R, *et al*. kAFL: Hardware-assisted feedback fuzzing for OS kernels. Proceedings of the 26th USENIX Conference on Security Symposium. Vancouver: USENIX Association, 2017. 167–182.

18 Bellard F. QEMU. https://www.qemu.org. [2023-03-19].

19 Sun H, Shen YH, Wang C, *et al*. HEALER: Relation learning guided kernel fuzzing. Proceedings of the 28th ACM SIGOPS Symposium on Operating Systems Principles. ACM, 2021. 344–358.

20 Pailoor S, Aday A, Jana S. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. Proceedings of the 27th USENIX Conference on Security Symposium. Baltimore: USENIX Association, 2018. 729–743.

21 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. 网络安全技术与应用, 2019, (11): 13–16. [doi: 10.3969/j.issn.1009-6833.2019.11.010]

22 Li D, Chen H. FastSyzkaller: Improving fuzz efficiency for Linux kernel fuzzing. Journal of Physics: Conference Series, 2019, 1176(2): 022013.

23 Google. syzbot. https://syzkaller.appspot.com/upstrea. [2023-03-19].

24 Fan JJ. TF-IDF. https://github.com/Junjie-Fan/tfidf. [2023-03-19].

25 Corina J, Machiry A, Salls C, *et al*. DIFUZE: Interface aware fuzzing for kernel drivers. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas: ACM, 2017. 2123–2138.

26 Jeong DR, Kim K, Shivakumar B, *et al*. Razzer: Finding kernel race bugs through fuzzing. Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP). San Francisco: IEEE, 2019. 754–768.

27 Chen YQ, Xing XY. SLAKE: Facilitating slab manipulation for exploiting vulnerabilities in the Linux kernel. Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. London: ACM,

2019. 1707–1722.

28 Kim K, Jeong DR, Kim CH, *et al*. HFL: Hybrid fuzzing on the Linux kernel. Proceedings of the 27th Annual Network and Distributed System Security Symposium. San Diego: The Internet Society, 2020.

29 Zhang MR, Liu JZ, Ma FC, *et al*. IntelliGen: Automatic driver synthesis for fuzz testing. Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). Madrid: IEEE, 2021. 318–327.

30 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. 信息安全学报, 2019, 4(2): 69–82.

31 Hertz J. TriforceLinuxSyscallFuzzer. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer. (2017-01-05)[2023-03-19].

32 Liang J, Jiang Y, Chen YL, *et al*. PAFL: Extend fuzzing optimizations of single mode to industrial parallel mode. Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Lake Buena Vista: ACM, 2018. 809–814.

33 Xu M, Kashyap S, Zhao HQ, *et al*. Krace: Data race fuzzing for kernel file systems. Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP). San Francisco: IEEE, 2020. 1643–1660.

34 Lin ZP, Chen YQ, Wu YH, *et al*. GREBE: Unveiling exploitation potential for Linux kernel bugs. Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP). San Francisco: IEEE, 2022. 2078–2095.

(校对责编: 孙君艳)