Beijing Forest Studio

北京理工大学信息系统及安全对抗实验中心

面向操作系统的模糊测试

硕士研究生 杨语航

2025年01月19日

问题回溯

- 相关内容
  - 2024.09.03 张浩然《大模型赋能的模糊测试用例生成》
  - 2024.06.09 谢宁《基于突变的模糊测试》
  - 2024.05.26 邵思源《面向网络应用程序的模糊测试》

内容提要

- 预期收获
- 题目内涵解析
- 研究背景与意义
- 研究历史与现状
- 知识基础
- 算法原理
  - ACTOR
  - FuzzNG
- 特点总结与工作展望
- 参考文献

预期收获

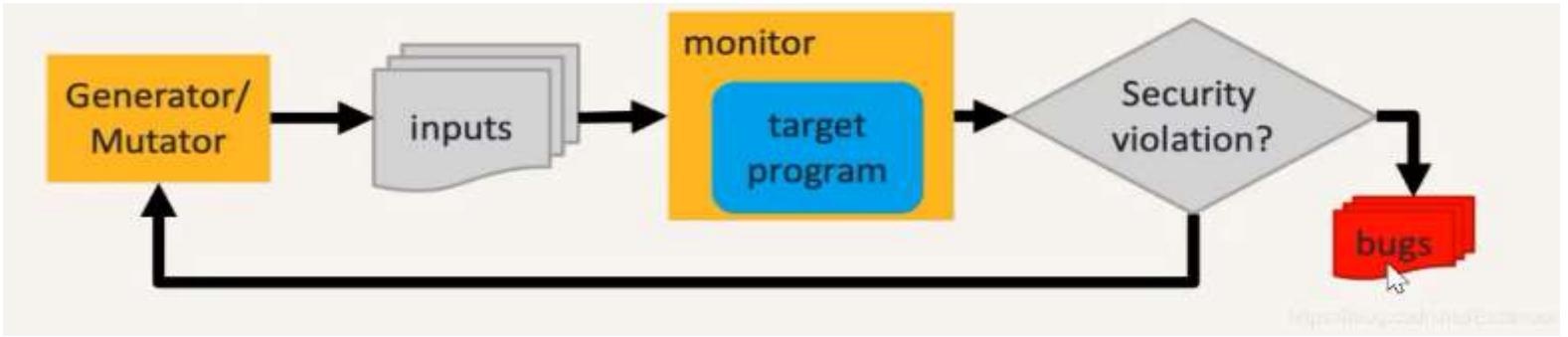
- 预期收获
  - 了解操作系统漏洞挖掘的特殊挑战和常见漏洞
  - 掌握操作系统模糊测试中的常用方法和基本概念
  - 理解两种前沿Linux内核的模糊测试技术原理与方法

内涵解析与研究目标

- 题目内涵解析（面向操作系统的模糊测试）
  - 操作系统：管理计算机软硬件资源的系统软件，提供用户接口并协调资源分配，主要功能包括资源管理、用户服务和硬件接口
  - 模糊测试：根据一定规则，产生大量随机数据然后输入到程序中并监视异常情况出现，以此发现可能的程序错误和漏洞

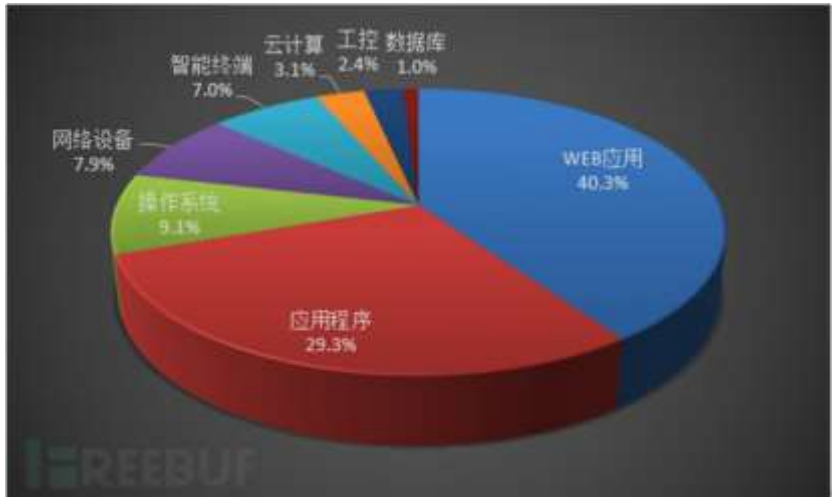
计算机系统的层次结构

- 研究目标
  - 以操作系统为研究对象，如Linux、Windows、移动操作系统等
  - 借助静态分析、符号执行及模糊测试等技术进行漏洞挖掘，增强操作系统鲁棒性

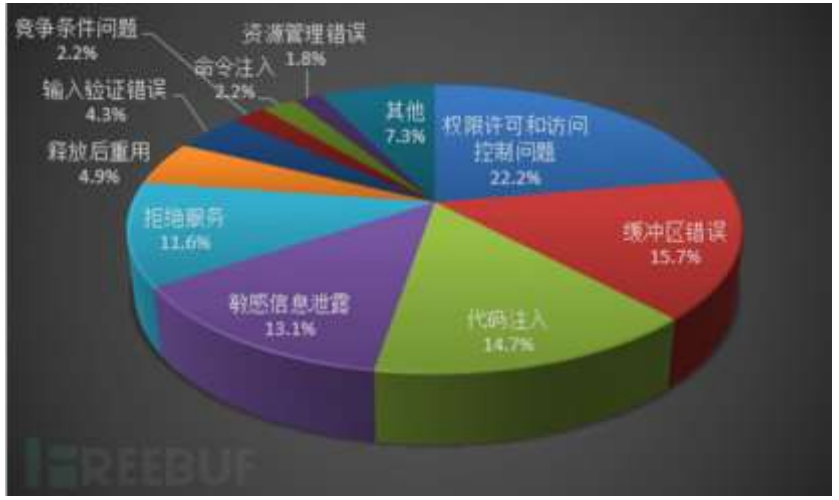


研究背景　操作系统生态安全问题越发复杂

- 研究背景
  - 重要性：操作系统是计算机系统的核心，其漏洞被恶意利用，威胁整个计算机安全
  - 广泛性：目前各类操作系统正被广泛使用在个人电脑、移动设备和企业服务器等，因此增强操作系统稳定性显得异常重要
  - 复杂性：1991年，Linux Kernel 0.01的源码行数约1万，然而仅2022年发布的5.19版本LOC超过了3300万行



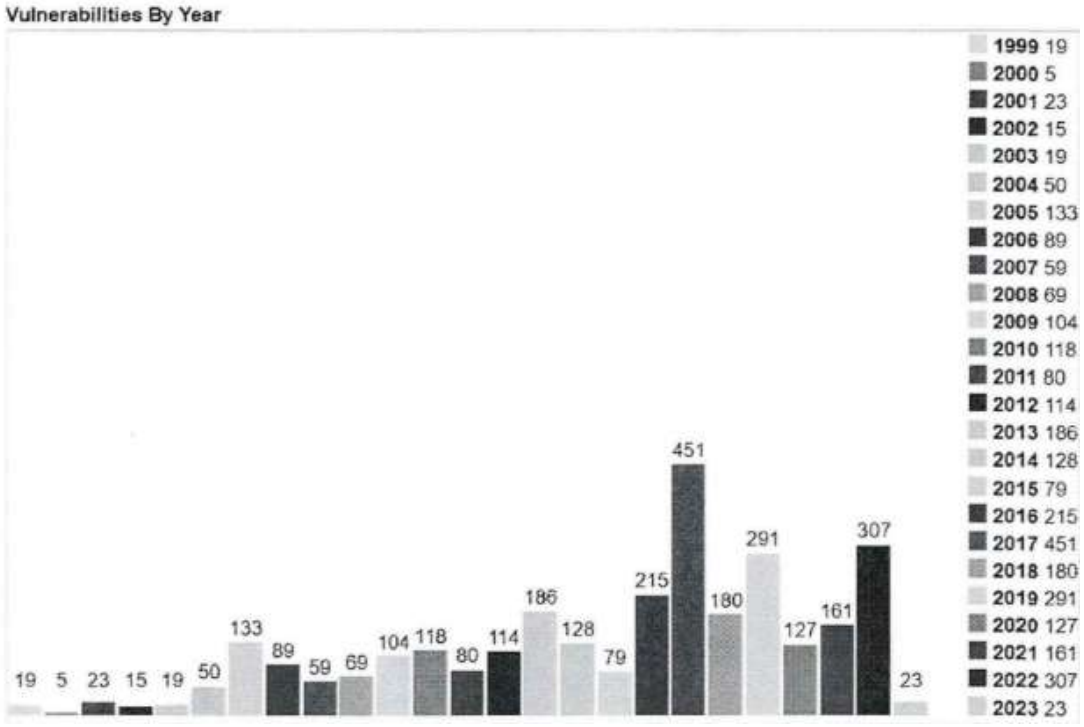
2023年漏洞影响对象占比



操作系统中漏洞类型占比

研究意义

- 研究意义
  - 操作系统稳定性提高
    - 及时发现OS的潜在漏洞和缺陷
    - 提升安全防护能力，保护用户数据隐私
  - 促进系统优化改进
    - 推动技术优化，适应安全威胁和用户需求
    - 加速安全更新迭代
  - 降低漏洞挖掘门槛
    - 推动形成成熟的模糊测试工具
    - 大幅减少漏洞挖掘专业经验需求



Linux操作系统内核漏洞统计1999-2023

发现操作系统漏洞，促进系统优化改进，降低漏洞挖掘门槛

研究历史　面向操作系统的模糊测试

2006 Peter等人首次开发针对Linux内核的原型静态分析工具，可高效处理大约三万行的源代码，相当于一个小的内核驱动源代码在包含引用的头文件和展开所有宏之后的大小

2011 Sim等人针对嵌入式Linux进行模糊测试，提出了一种自适应随机测试(ART)方法来生成测试用例，通过在输入域中均匀分布测试输入来提高命中故障的机会

2017 Machiry等人提出一种用于Linux内核驱动程序的静态分析工具，通过执行流敏感、上下文敏感和字段敏感的指针和污点分析，进行通用漏洞检测

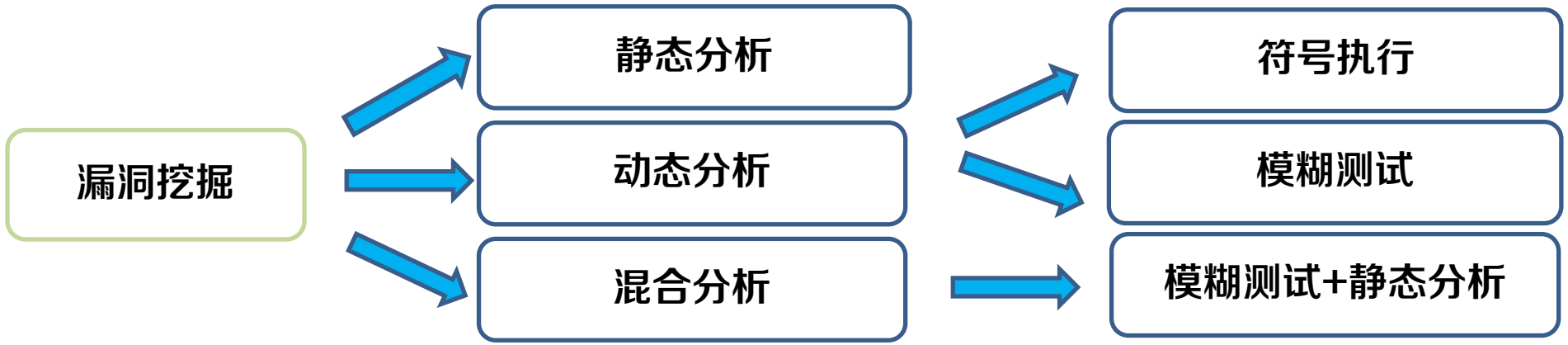
2018 Pailoo等人提出通过从真实程序的系统调用跟踪中提取种子来提高模糊测试效果的框架MoonShine，作为Syzkaller的扩展进行设计和实现

2020 Kim等人首次将模糊测试和符号执行结合，基于内核模糊测试工具Syzkaller和符号执行器S2E实现，在覆盖率和效率上超越当时最新的方法

2022 Zhao等人结合模糊测试和静态分析提出一种新型状态感知模糊测试工具StateFuzz，利用静态分析识别状态变量，优先考虑能触发新状态的测试用例

2023 Fleischer等人注重执行代码所采取的动作和顺序，首次将模板引导的合成应用于测试用例生成，模板由领域特定语言编写，精确控制动作顺序和组合生成输入，从而触发那些依赖特定动作序列的漏洞

2023 Bulekov等人提出一种可在没有系统调用描述的情况下对Linux内核进行模糊测试的工具FuzzNG，利用内核设计基本特性来重塑和简化输入空间

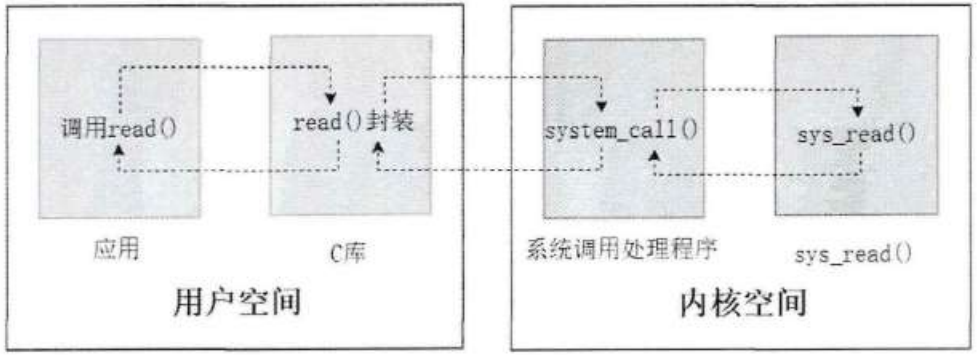


研究现状

- 漏洞挖掘方法
  - 静态分析：无需运行程序，通常针对源代码，分析其词法、语法、语义特征、数据流等，来发现潜在的漏洞，缺点是误报率较高
  - 符号执行：将程序输入符号化，对执行路径构建约束集合，通过求解约束方程确定能够执行某些路径的输入，缺点是分支很多时，面临“状态爆炸”
  - 模糊测试-覆盖率引导：目标是最大化代码覆盖率
    - 与静态分析结合：借助静态分析提供针对性输入，优化输入空间
    - 与符号执行结合：帮助解决“状态爆炸”的问题，但会受到诸如浮点运算精度的限制，且符号执行存在无法模拟部分系统调用的情况
- 漏洞挖掘对象
  - 常见有Web、应用程序、操作系统、数据库、网络协议等

知识基础

- 系统调用（Syscall）
  - 定义：用户空间程序与内核交互的接口，本质是函数
  - 作用：用户空间程序通过Syscall请求内核执行特定操作，如文件操作open\write、进程控制fork、通信exit等
- 内核函数（Kernel function）
  - 定义：内核内部使用的函数，用户空间程序无法直接调用
  - 关系：系统调用是内核函数的封装，系统调用是桥梁，内核函数是具体执行任务的工具
- 动作（Action）
  - 定义：关注系统调用的具体效果，表示为三元组
  - 关系：一个系统调用可以对应一个或多个动作
  - 示例：堆动作 $\mathrm{A} = \langle A_a, addr, size \rangle$



- 系统调用描述（System-Call Descriptions）
  - 定义：使用领域特定语言编写的对系统调用的详细说明
  - 作用1：详细记录了系统调用所需的参数类型、返回值、约束条件等许多信息
  - 作用2：描述了系统调用如何管理资源，以及不同系统调用之间的依赖关系
  - 作用3：帮助内核模糊测试器生成有效的测试用例
  - 示例：使用Syzlang对ioct1编写的部分系统调用描述

```
ioctl$VHOST_SET_VRING_ADDR(fd fd_vhost,
                           cmd const[VHOST_SET_VRING_ADDR],
                           arg ptr[in, vhost_vring_addr])

VHOST_SET_VRING_ADDR = 1076408081

vhost_vring_addr {
 index flags[vhost_vring_index, int32]
 flags int32[0:1]
  desc_user_addr ptr64[out, array[int8]]
  used_user_addr ptr64[out, array[int8]]
  avail_user_addr ptr64[out, array[int8]]
  log_guest_addrs flags[kvm_guest_addrs, int64]
}
```

【 USENIX 】
{ACTOR}:{Action-Guided} Kernel Fuzzing

ACTOR TIPO

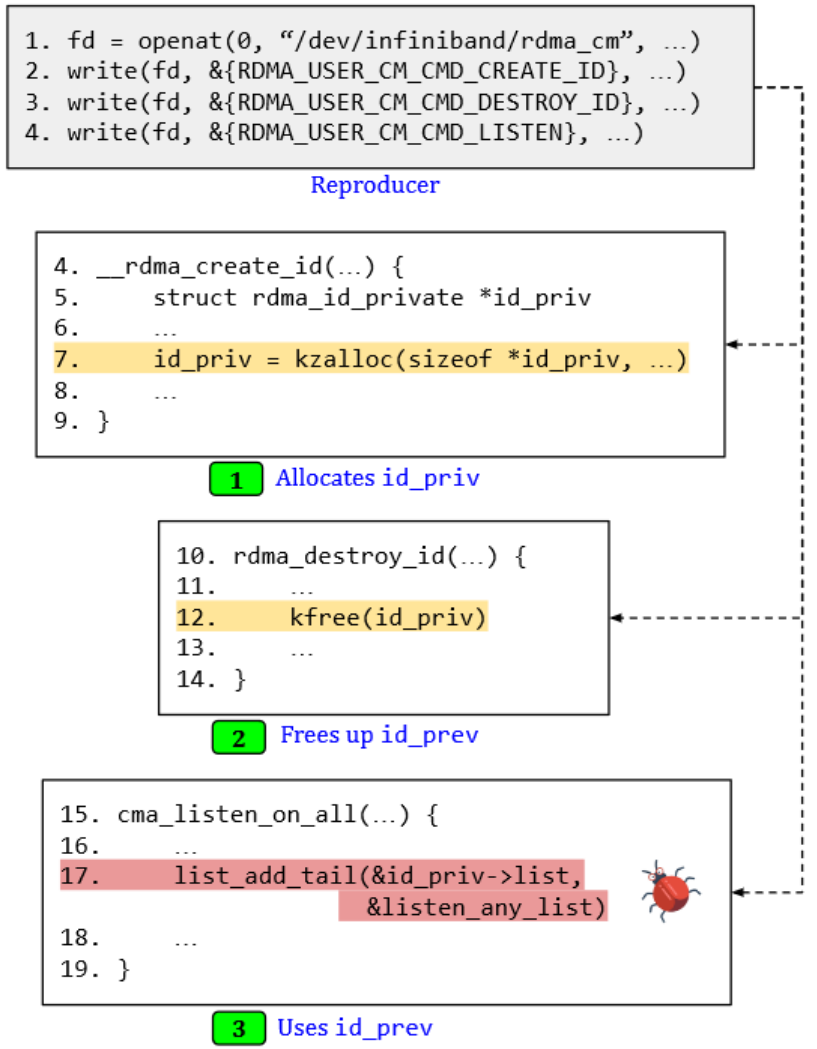
T	目标	提高内核模糊测试效率
I	输入	初始测试用例*n、预定义Bug模板*n
P	处理	1. 执行初始测试用例，并进行动作挖掘处理 2. 程序合成，利用动作组及预定义Bug模板合成输入用例 3. 检测目标状态，并保留高质量输入用例作为次轮输入种子
O	输出	目标内核的崩溃报告*1

P	问题	现有覆盖率引导模糊器难以发现需要特定动作顺序触发的Bug
C	条件	基于覆盖率引导模糊器、需要预定义Bug模板和静态语义分析
D	难点	1. 如何选择最有可能触发漏洞的动作序列 2. 如何准确对通用动作进行语义标记
L	水平	USENIX 2023（CCF-A）

ACTOR 创新性引入

- **Coverage-Guided**
  - 思路：保留高覆盖率测试用例，以此为基础进行变异等算法，仅考虑如何最大化代码覆盖率
  - 问题：已覆盖含Bug的代码块，但无法触发漏洞
- **Action-Guided**
  - 观点1：成功触发漏洞和执行代码的动作类型有关
  - 观点2：成功触发漏洞和这些动作的执行顺序有关
  - 分析：执行代码的动作类型确保输入能到达含Bug代码块，特定的动作顺序更可能触发潜在错误

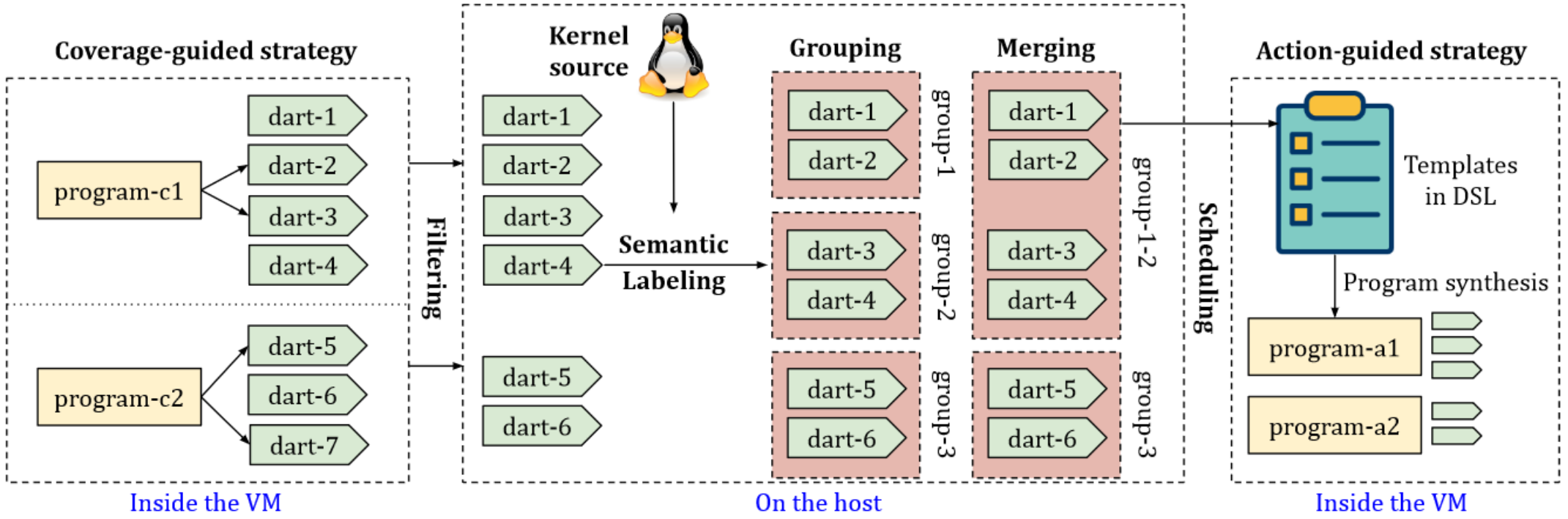
Coverage-Guided可能会执行：openat ⇒ write



Use-After-Free (UAF)

ACTOR 算法原理图

- 思路分析
  - 步骤1：使用覆盖率引导策略生成测试用例，逐步提高代码覆盖率
  - 步骤2：在生成的测试用例中，记录和分析执行的动作
  - 步骤3：动作引导策略根据记录的动作信息，生成特定的动作序列，高效发现漏洞



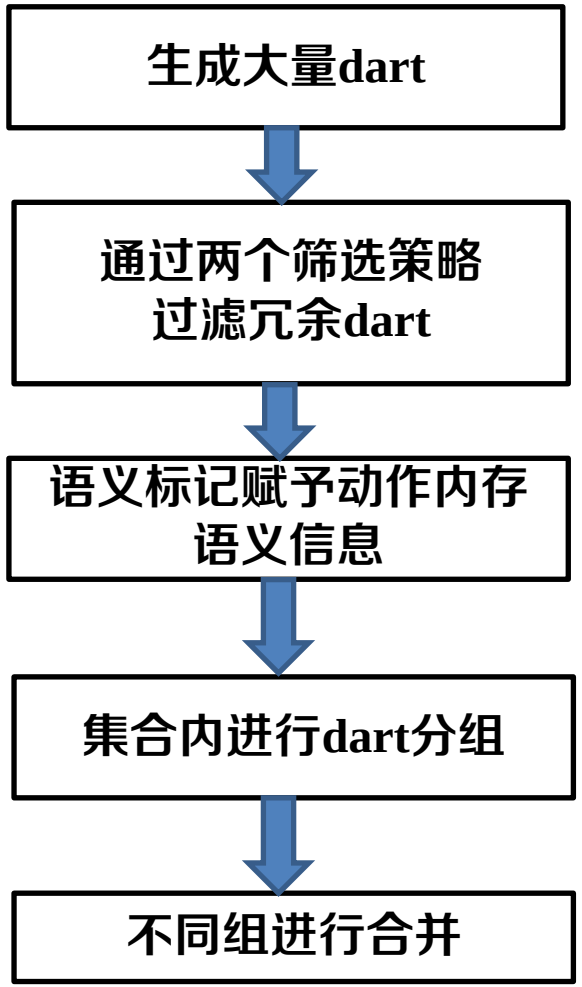
算法原理 动作挖掘

- 动作挖掘（Action Mining）
  - Dart生成：将Syscall及其参数与触发的动作相关联，生成一个三元组

    Dart d = ⟨s, a, Δ⟩，s为Syscall，a为动作类型，Δ为堆栈跟踪
  - Dart筛选：一个测试用例P可以产生大量冗余、重复的dart
    - 策略1-保留相关：仅保留在已观察到的堆分配内存上操作的dart
      - 假设测试用例P的全部dart为集合D，其中的堆分配集合为$A = \{alloc(d) | d \in D\}$
      - 通过查询某个dart操作的内存地址是否在A中，决定是否保留
    - 策略2-记录首次：仅记录Syscall每个分配的第一个读/写dart
  - 语义标记：初始的dart一般缺乏语义信息，其中的动作是通用的， 如：A_r/A_w
    - 语义恢复：利用静态分析从源代码中收集关于内存访问的语义信息
    - 语义细化：将这些信息注入模糊测试循环，使得A_r/A_w变为A_{pr}/A_{iw}等

算法原理 动作挖掘

- 动作挖掘
  - Dart分组：将dart集合D分为许多组，每个组由两个部分组成，一个负责分配堆内存的dart $d_a \in D$，其他所有在$alloc(d_a)$上进行操作的dart d
    - 例如：组合G可以表示为：$G = \{d | addr(d) \in alloc(d_a)\}$
    - 目的：确保组中的dart操作相同的内存区域，提高生成的测试用例的相关性
  - Dart组合并
    - 分组的对象：同一集合中的dart
    - 合并的对象：不同集合中的dart组
    - 目的：扩大组范围，学习不同测试用例的Syscall之间的关系，提高测试用例多样性



- 程序合成
  - 输入1：利用领域特定语言编写的Bug模板
    - 模板由一系列三元组形成，如：$bt = \{< at, id, repeat >\}$
    - 三元组中的at表示期望的动作类型， id是标识符用于连接多个三元组， $repeat$ 用于指定当前dart重复次数
  - 输入2：动作挖掘中产生的相关dart组
  - 输出：从组中选择适当的dart实例化Bug模板，生成可能触发Bug的测试用例

Bug class	Template	Bug class	Template
Use After Free	$\mathcal{A}_a \rightarrow \mathcal{A}_d \rightarrow [\mathcal{A}_r \mid \mathcal{A}_w]$	Null Ptr Deref	$\mathcal{A}_a^x \rightarrow \mathcal{A}_d^x$
Double Free	$\mathcal{A}_a \rightarrow \mathcal{A}_d \rightarrow \mathcal{A}_d$	Invalid Free	$\mathcal{A}_d$
Out of Bounds (1)	$\mathcal{A}_a \rightarrow \mathcal{A}_{iw}^* \rightarrow \mathcal{A}_{ir}$	Memory Leak (1)	$\mathcal{A}_a^*$
Out of Bounds (2)	$\mathcal{A}_a \rightarrow \mathcal{A}_{pw}^* \rightarrow \mathcal{A}_{pr}$	Memory Leak (2)	$\mathcal{A}_a \rightarrow \mathcal{A}_{pw} \rightarrow \mathcal{A}_d$
Uninitialized Read	$\mathcal{A}_a \rightarrow \mathcal{A}_r$	-	-

常见的九种内存漏洞及对应Bug模板

实验设计　数据资源

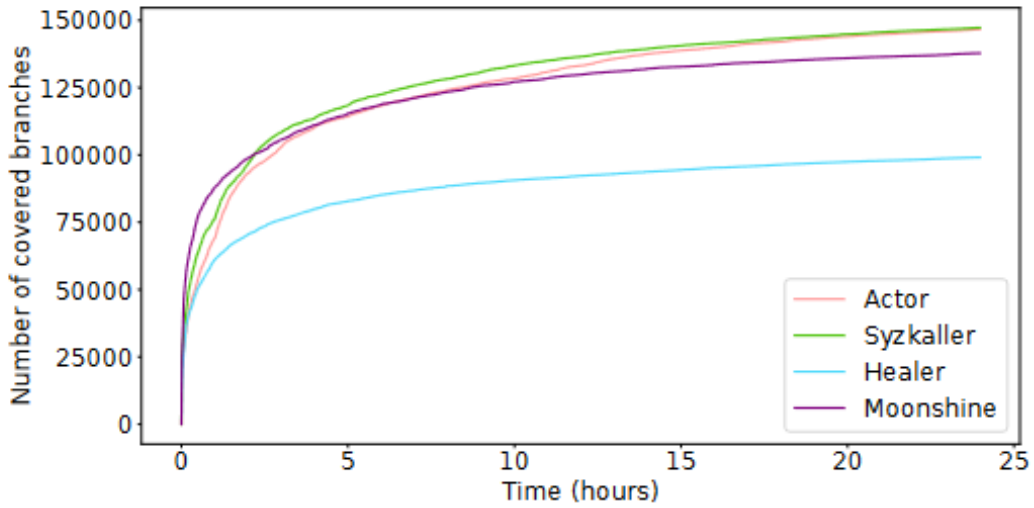
- 实验配置
  - 基础模糊器：Syzkaller
  - 测试对象：Linux kernel v5.4.206、v5.10.131、v5.19-rc6、v6.2-rc5
  - 动作收集：基于KASAN
  - 语义标记：基于LLVM 14
  - 硬件配置：2×Intel®Xeon ® E5-2690 V2 @ 3.00GHz CPU和256G内存的服务器
  - 操作系统：Ubuntu 20.04.4 LTS 64 bit OS
- 对比方法：
  - Syzkaller (现有工具，由Google团队开发维护)
  - Healer (2021)
  - MoonShine (2018)

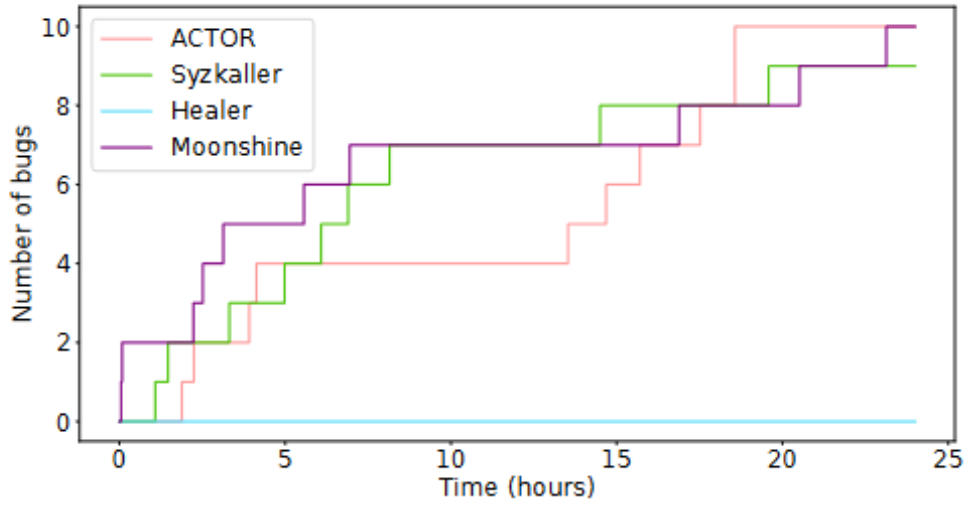
- RQ1：新漏洞发现
  - 右图列出ACTOR发现的漏洞描述、类型、对应内核版本、Bug状态及耗费时间
  - 共41个未知Bug，9个已被修复，一天内发现了15个

Kernel crash description	Crash type	Kernel version	Repro	Reported	Patched	Days
KASAN: use-after-free Read in drm_gem_object_release	Use-After-Free	v5.4.206	✓	✓	✓	0.73
UBSAN: undefined-behaviour in tdp_page_fault	Undef Behavior	v5.4.206	✗	-	-	0.73
general protection fault in sock_def_error_report	GPF	v5.4.206	✓	✓	-	2.96
general protection fault in dd_insert_requests	GPF	v5.4.206	✗	-	-	3.77
general protection fault in reset_interrupt	GPF	v5.4.206	✗	-	-	4.74
BUG: unable to handle kernel paging request in imageblit	Page Fault	v5.4.206	✓	✓	-	6.55
KASAN: use-after-free Read in tcp_retransmit_timer	Use-After-Free	v5.4.206	✗	-	-	6.45
general protection fault in vmx_vmenter	GPF	v5.4.206	✗	-	-	11.30
WARNING: ODEBUG bug in netdev_run_todo	Reachable Warning	v5.4.206	✗	-	-	8.54
UBSAN: undefined-behaviour in xprt_calc_majortimeo	Undef Behavior	v5.4.206	✗	-	-	11.40
KASAN: vmalloc-out-of-bounds Write in snd_pcm_hw_params	OOB access	v5.10.131	✓	✓	✓	0.03
KASAN: null-ptr-deref Write in rhashtable_free_and_destroy	Null ptr deref	v5.10.131	✗	-	-	1.01
WARNING: kmalloc bug in kvm_arch_prepare_memory_region	Reachable Warning	v5.10.131	✗	-	-	5.02
KASAN: use-after-free Read in post_one_notification	Use-After-Free	v5.10.131	✓	✓	✓	1.63
KASAN: vmalloc-out-of-bounds Write in imageblit	OOB access	v5.10.131	✗	-	-	6.49
INFO: task hung in gfs2_read_super	Reachable Info	v5.10.131	✓	✓	-	3.83
general protection fault in start_motor	GPF	v5.10.131	✓	✓	-	0.61
BUG: soft lockup in br_multicast_port_group_expired	Soft Lockup	v5.10.131	✗	-	-	4.45
KASAN: slab-out-of-bounds Read in ntfs_get_ea	OOB access	v5.19-rc6	✓	✓	✓	4.36
KASAN: vmalloc-out-of-bounds Read in cleanup_bitmap_list	OOB access	v5.19-rc6	✓	✓	-	4.11
KASAN: use-after-free Read in run_unpack	Use-After-Free	v5.19-rc6	✓	✓	✓	4.35
KASAN: use-after-free Read in __io_remove_buffers	Use-After-Free	v5.19-rc6	✓	✓	✓	4.78
KASAN: invalid-free in __io_uring_register	Invalid Free	v5.19-rc6	✓	✓	✓	7.29
KASAN: use-after-free Read in reiserfs_fill_super	Use-After-Free	v5.19-rc6	✓	✓		9.53
INFO: task hung in __bread_gfp	Reachable Info	v5.19-rc6	✓	✓	-	2.33
WARNING in inet_sock_destruct	Reachable Warning	v5.19-rc6	✓	✓	✓	6.57
kernel BUG in f2fs_new_node_page	Reachable Assertion	v5.19-rc6	✓	✓	✓	9.10
kernel BUG in ntfs_read_folio	Reachable Assertion	v5.19-rc6	✓	✓	-	0.13
INFO: task hung in __floppy_read_block_0	Reachable Info	v5.19-rc6	✓	✓	-	1.94
BUG: unable to handle kernel paging request in kvm_dev_ioctl	Page Fault	v6.2-rc5	✗	-	-	0.17
KMSAN: uninit-value in __dma_map_sg_attrs	Uninit Value	v6.2-rc5	✓	✓	-	0.94
KMSAN: uninit-value in sr_check_events	Uninit Value	v6.2-rc5	✓	✓	-	0.05
KMSAN: uninit-value in post_read_mst_fixup	Uninit Value	v6.2-rc5	✓	✓	-	0.25
general protection fault in get_cpu_entry_area	GPF	v6.2-rc5	✗	-	-	0.02
KMSAN: uninit-value in nilfs_add_checksums_on_logs	Uninit Value	v6.2-rc5	✓	✓	-	0.45
KMSAN: uninit-value in generic_bin_search	Uninit Value	v6.2-rc5	✓	✓	-	0.01
BUG: unable to handle kernel NULL pointer dereference in ntfs_iget5	Null ptr deref	v6.2-rc5	✓	✓	-	0.12
BUG: unable to handle kernel paging request in get_cpu_entry_area	Page Fault	v6.2-rc5	✗	-	-	0.02
BUG: unable to handle kernel paging request in bpf_ringbuf_alloc	Page Fault	v6.2-rc5	✗	-	-	2.47
KASAN: null-ptr-deref Read in gfs2_evict_inode	Null ptr deref	v6.2-rc5	✗	-	-	0.45
KASAN: null-ptr-deref Read in soft_cursor	Null ptr deref	v6.2-rc5	✗	-	-	2.33

实验设计　对比实验

- RQ2： 代码覆盖率
  - Syzkaller在代码覆盖率方面表现最佳，但ACTOR紧随其后，仅比Syzkaller低0.42%
- RQ3： 漏洞发现效率
  - Syzkaller发现9个漏洞
  - ACTOR和MoonShine各发现10个漏洞
  - Healer没有发现任何漏洞
  - ACTOR发现2个其他工具未发现的内存错误漏洞





算法总结 ACTOR

- 算法流程
  - 动作挖掘：记录Syscall和对应动作，语义标记，dart分组和合并
  - 程序合成：选择漏洞模板和dart组，实例化测试用例
  - 执行和反馈：执行新测试用例，收集内核反馈信息，优化后续输入
- 算法优势
  - 高覆盖率、高漏洞发现效率
  - 支持增加预定义Bug模板，有一定扩展性
- 算法不足
  - 静态分析可能无法完全准确识别内存访问语义信息，导致动作类型错误标记
  - 受到预定义Bug模板的限制，目前仅支持定义内核堆相关动作类型

【 NDSS 】

No Grammar, No Problem: Towards Fuzzing the Linux Kernel without System-Call Descriptions

FuzzNG TIPO

T	目标	无复杂系统调用描述的情况下实现高覆盖率的模糊测试
I	输入	测试用例*n、Agent-Config文件*1
P	处理	1. QEMU-Fuzz生成原始输入 2. NG-Agent解释原始输入为系统调用序列，mod-NG填充进程内存 3. NG-Agent收集覆盖率数据，评估模糊测试效率
O	输出	崩溃报告*1

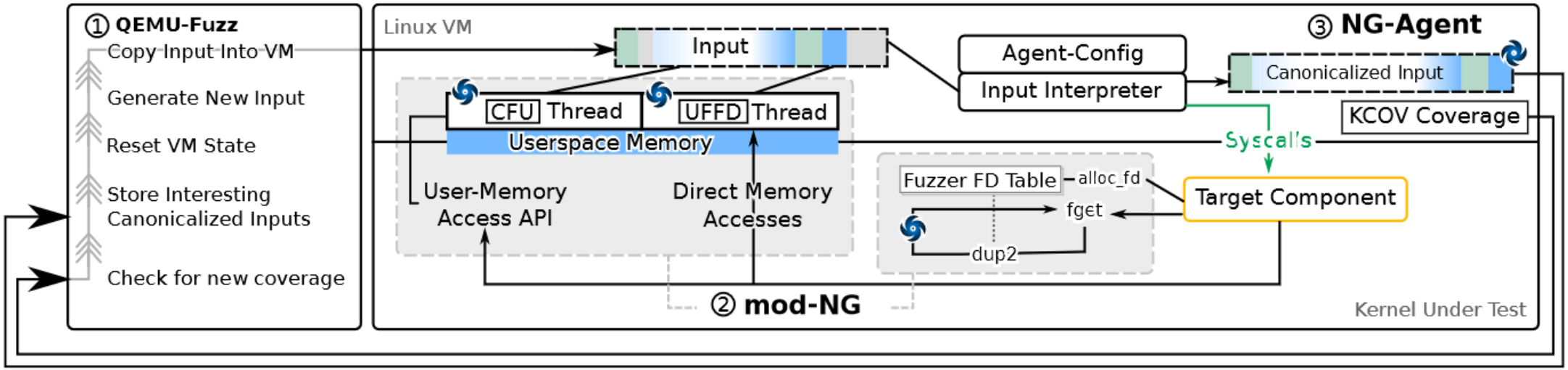
P	问题	当前先进的内核模糊测试需借助复杂的系统调用描述
C	条件	需要通用模糊测试器生成原始输入
D	难点	如何在缺乏详细系统调用描述的情况，生成有效输入
L	水平	NDSS 2023（CCF-A）

FuzzNG 创新性引入

- 传统内核模糊测试器
  - 背景1：内核模糊的测试用例输入一般由系统调用序列组成
  - 背景2：单个系统调用最多可接受六个参数，包括指针和文件描述符这类特殊参数
  - 问题：简单的对参数模糊很难生成有效的、可被内核执行的系统调用
  - 方法：借助系统调用描述，指导模糊测试器生成测试用例，如Syzkaller
  - 局限性：随着内核规模增大，系统调用描述渐渐显得不足；且过于依赖系统调用描述可能存在“过拟合”
- FuzzNG
  - 目标：无需复杂的系统调用描述和语法，实现高质量的测试用例
  - 方法：借助内核设计的基本特性，重塑和简化模糊测试输入空间

FuzzNG 算法原理图

- 算法原理图
  - QEMU-Fuzz：主要根据反馈信息生成新输入，并建立“快照”机制
  - mod-NG：对内核进行修改，hook 内存访问API 和fd API，来拦截和处理内核中的API，重塑和简化输入空间
  - NG-Agent：执行测试用例，并收集反馈信息

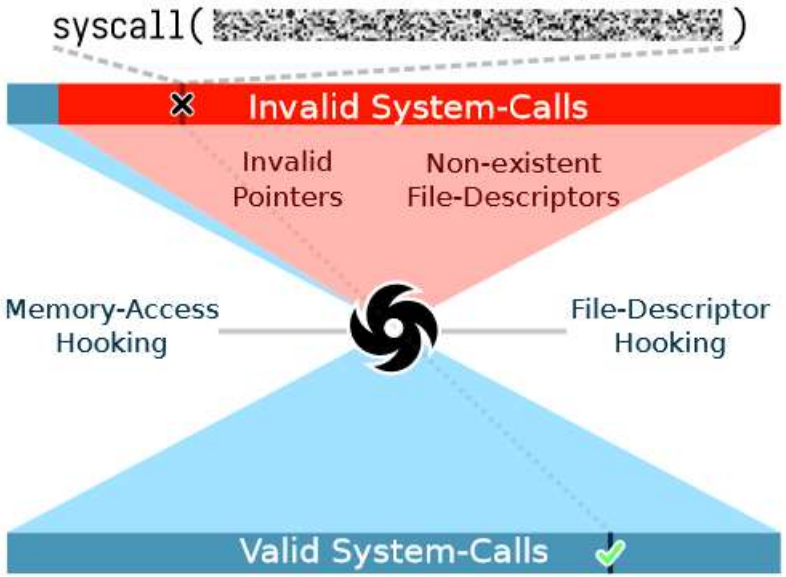


算法原理　重塑和简化输入空间

- 原始系统调用序列：含大量无效参数，难以探索内核深层次路径
- 内核API：内核函数，如Fdget查找文件、copy_from_user复制数据
- Hook：对内核API修改，起到拦截调用，修改信息的作用，类似“插桩”行为
- 示例：ssize_t write(int fd, const void *buf, size_t count)

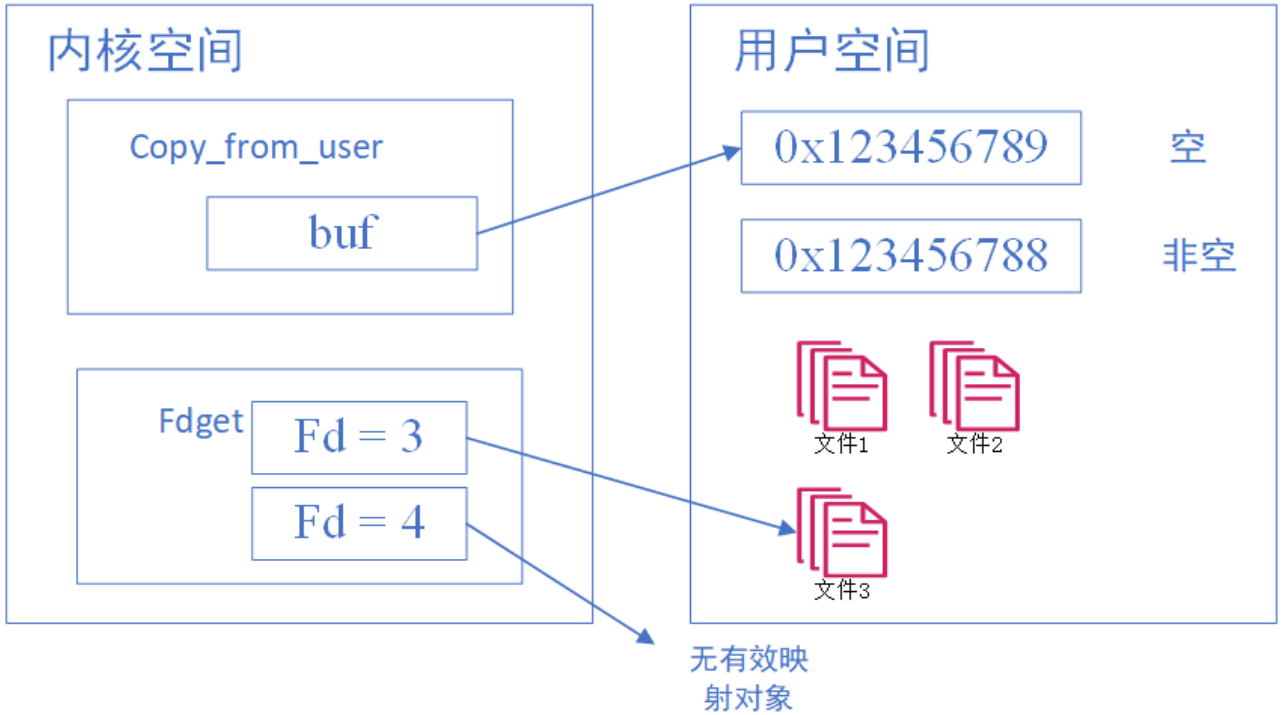
示例中的write系统调用，至少需要调用三个内核API

1. 调用Fdget，查找对应的文件
2. 调用copy_from_user，从用户空间复制数据
3. 最后调用写入数据到文件的API



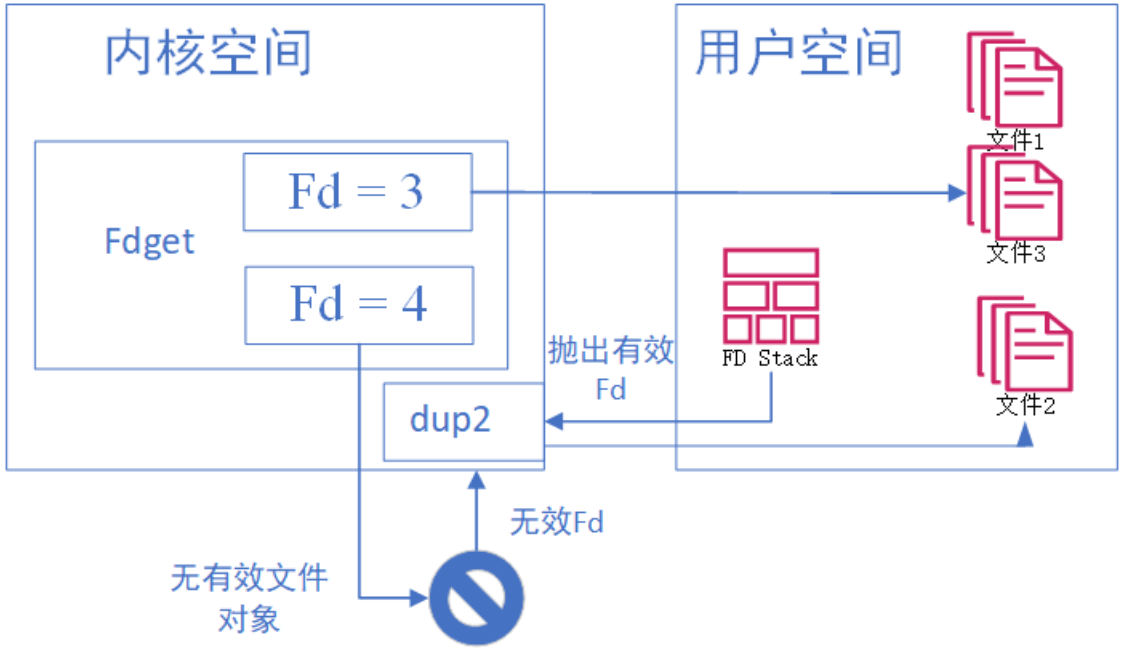
算法原理　重塑和简化输入空间

- 示例：ssize_t write(int fd, const void *buf, size_t count)
- Q1：如何确保指针没有指向空或内核地址呢?
  - 空指针，填充对应用户内存区域
  - 内核指针，更改指向为用户空间
- Q2：如何确保文件标识符fd对应有效文件呢?
  - Fd有效，不做处理
  - Fd无效，重新映射到有效文件



算法原理　重塑和简化输入空间

- 通过Hook内核fd相关API，确保输入的系统调用中fd有效
  - 步骤1：通过hook内核alloc_fd函数，每次内核分配新的fd，记录该fd和对应的文件对象
  - 步骤2：根据步骤1中记录信息，维护一个FD stack，用于查找和映射fd
  - 步骤3：当内核调用fdget函数，hook拦截该调用，先检查初始fd是否有效
    - Fd有效：不做额外处理，正常返回对应文件对象
    - Fd无效：抛出FD stack中最上面的文件描述符，使用dup2系统调用将其映射到未分配的文件描述符上



实验设计　数据资源

- 实验配置
  - 基础模糊器：libFuzz
  - 测试对象：Linux kernel v5.12
  - 硬件配置：2×Intel®Xeon ® E5-2600 V3 和192G内存的服务器
  - 操作系统： Ubuntu 20.04.4 LTS 64 bit OS
- 对比方法
  - Syzkaller (现有工具，由Google团队开发维护)
  - Healer (2021)

实验设计　对比实验

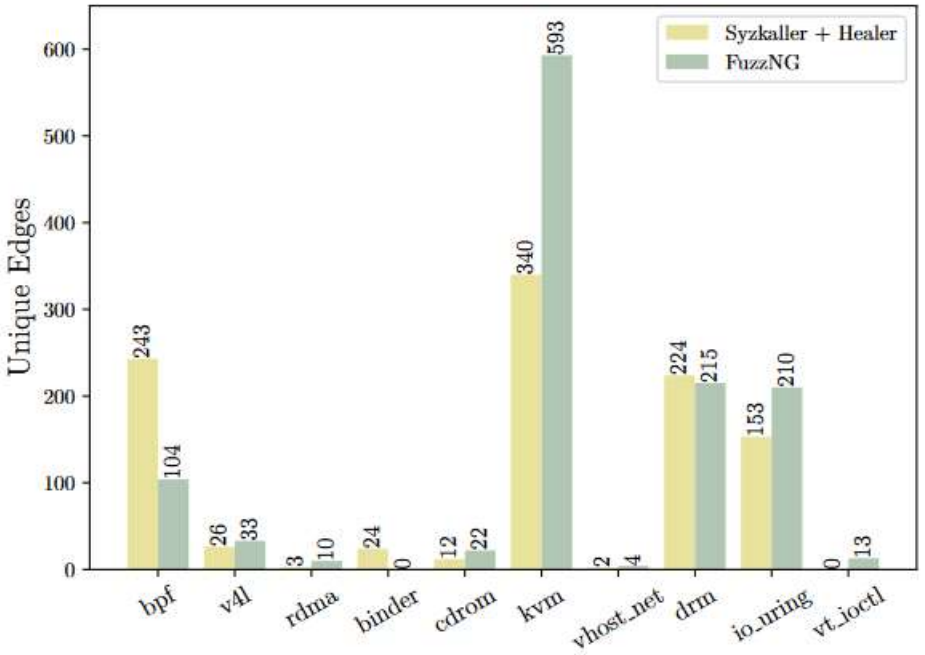
- RQ1：和其他模糊测试器比，FuzzNG的覆盖率如何?
- 实验结论
  - FuzzNG在两种平均覆盖率计算中均优于Syzkaller
- RQ2：与Syzkaller相比，FuzzNG的配置文件大小如何?
- 实验结论：
  - FuzzNG的Config LoC仅为Syzkaller的系统调用描述代码行数的1.69%

Component	Max Cov	Syzkaller		Healer	FuzzNG	
		Edge Count	Syzlang LoC	Edge Count	Edge Count	Config LoC
bpf	15359	3623	864	1121	3572	1
video4linux	1004	563	381	446	567	4
rdma	4014	562	1474	*	591	5
binder	2506	340	272	56	344	6
cdrom	956	138	351	120	144	5
kvm	34924	9213	891	8755	9468	7
vhost_net	415	218	157	210	225	9
drm	12503	2296	745	1978	2138	7
io_uring	3413	982	343	986	1003	6
vt_ioctl	332	142	381	138	162	9
Average				76.52%	**102.53%**	**1.67%**
Geo. Mean vs. Syzkaller				66.95%	**102.41%**	**1.09%**

实现了无复杂系统调用描述的情况下的高代码覆盖率

实验设计　对比实验

- 对比实验：FuzzNG在不同内核组件上唯一覆盖的代码路径
- 实验结果
  - FuzzNG在KVM组件显著覆盖最多的唯一路径
  - Syzkaller + Healer在bpf组件更有优势
  - 在70%的组件中FuzzNG在唯一路径覆盖数上超过或接近Syzkaller + Healer
- 结果分析
  - KVM组件涉及大量内核子系统交互，状态转换复杂，FuzzNG的重塑和简化输入空间策略较适合处理



FuzzNG最终在经过广泛测试的内核组件中发现了9个新漏洞

算法总结 FuzzNG

- 算法流程
  - 初始化：配置Agent-Config，完成内核API Hook设置
  - 执行：QEMU-Fuzz生成输入，NG-Agent解释输入，mod-NG有效化系统调用
  - 反馈：通过评估覆盖率保留“有趣的”测试用例，识别崩溃
- 算法优势
  - FuzzNG可在无详细系统调用描述的情况下，实现高覆盖率并发现新漏洞
  - FuzzNG的配置文件远小于Syzkaller的语法描述，对新组件支持容易
- 算法不足
  - 泛化能力不足，对内核版本和配置敏感
  - 对部分内核组件效果一般，如bpf

特点总结与未来展望

特点总结与未来展望

- **ACTOR**
  - 基于动作引导策略
  - 侧重于执行代码的动作类型和动作顺序
  - 关注对同一内存区域进行操作的动作
- **FuzzNG**
  - 基于覆盖率引导策略
  - 可在无复杂系统调用描述的情况下，实现高质量的测试用例
  - 关注涉及用户空间内存访问和文件描述符的系统调用
- 未来发展
  - 扩展ACTOR的支持的动作类型，目前仅支持堆相关的动作类型
  - 扩大FuzzNG对内核API 的Hook范围，目前仅针对CFD、UFFD和FD

参考文献

[1] Fleischer M, Das D, Bose P, et al. {ACTOR}:{Action-Guided} Kernel Fuzzing[C]. 32nd USENIX Security Symposium (USENIX Security 23). 2023: 5003-5020.

[2] Bulekov A, Das B, Hajnoczi S, et al. No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions[C]. Network and Distributed System Security (NDSS) Symposium. 2023.

[3] Raj A S, Gibbs W, Dong F, et al. Fuzz to the Future: Uncovering Occluded Future Vulnerabilities via Robust Fuzzing[C]. Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. 2024: 3719-3733.

知人者智，自知者明。胜人者有力，自胜者强。知足者富。强行者有志。不失其所者久。死而不亡者，寿。

谢谢！