Diskaller: 基于覆盖率制导的操作系统内核漏洞并行挖掘模型

涂序文[1], 王晓锋[1], 甘水滔[2], 陈爱国[1]

[1] 江南大学物联网工程学院, 江苏 无锡 214122
[2] 数学工程与先进计算国家重点实验室, 江苏 无锡 214083

摘要 内核是操作系统的核心, 它构建了操作系统各类程序运行时需要的基础环境: 如进程调度、存储管理、文件系统、设备驱动和网络通信等。操作系统内核漏洞的存在可能使得计算机系统遭受拒绝服务、信息泄露、超级用户权限提升等攻击, 因此, 针对内核的漏洞挖掘一直是网络安全领域的研究热点。本文在现有的研究基础上, 提出一种基于覆盖率制导的内核漏洞并行模糊测试模型, 该模型以代码覆盖率为导向, 以计算节点和控制节点组成的星型结构作为并行模型, 各计算节点通过代码覆盖率对系统内核持续测试, 控制节点完成计算节点间代码覆盖率的收集与交互, 突破了传统测试模型对计算资源要求限制和数据竞争的瓶颈, 极大的提升了代码覆盖率及测试速度, 加快了漏洞挖掘的效率。为了验证模型的实用性及有效性, 利用 Diskaller 与 Syzkaller 和 Triforce 进行对比, 一定条件下 Diskaller 覆盖率较 Syzkaller 提升 12.8%, 执行速率提升 229%, 较 Triforce 覆盖率提升 335%, 执行速率提升 450%, 并且发现了 Linux 内核中两个先前未被发现的漏洞。

关键词 分布式; 模糊测试; 内核测试; 漏洞挖掘; 操作系统内核
中图法分类号 TP311 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2019.03.07

Diskaller: Kernel Vulnerability Parallel Mining Model Based on Coverage Guidance

TU Xuwen[1], WANG Xiaofeng[1], GAN Shuitao[2], CHEN Aiguo[1]

[1] School of Internet of Things Engineering, Jiangnan University, Wuxi, Jiangsu 214122, China
[2] State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, Jiangsu 214083, China

Abstract Kernel is the core of the operating system. It builds the basic environment that the operating system needs when running various programs such as process scheduling, storage management, file systems, device drivers, and network communications. The existence of kernel vulnerabilities in operating system may cause computer systems suffer from denial of service, information leakage, super user privilege escalation and other attacks. Therefore, vulnerabilities mining technique in the kernel has always been a research hotspot in the field of network security. Based on the existing researches, this paper proposes a kernel vulnerability parallel mining model based on coverage guidance. This model is based on code coverage rate, and uses a star structure composed of computing nodes and control nodes as the parallel model. The computing node continuously tests the system kernel through code coverage, and the control node complete the interaction of code coverage between computing nodes, which breaks through the bottleneck of the traditional model's limitation on the computing resource requirements, greatly improves the code coverage and testing speed, and accelerates the vulnerabilities Digging efficiency. In order to verify the practicability and effectiveness of the model, Diskaller is compared with Syzkaller and Triforce. Under certain conditions, the Diskaller coverage rate is 12.8% higher than Syzkaller, 335% higher than Triforce, the execution speed is increased by 229% compared with Syzkaller, and increased by 450% compared with Triforce, we discovered two previously undiscovered vulnerabilities in the Linux kernel by this method.

Key words distributed; fuzzing; kernel vulnerabilities; vulnerability discovery; operating system kernel

1 介绍

2017 年 5 月 12 日, 暗影经纪人组织(Shadow Brokers)入侵方程式组织(Equation Group)获取并泄漏微软高危系统协议漏洞 MS17-010[1], 该漏洞形成的主要原因是在内核函数 srv！SrvOs2FeaListToNt

通讯作者: 涂序文, 硕士, 研究生, Email: icytxw @gmail.com。
本课题得到国家重点研发计划项目(No. 2016YFB0800803); 国家自然科学基金项目(No. 61672264)资助。
收稿日期: 2018-04-18; 修改日期: 2018-05-27; 定稿日期: 2019-02-27

处理文件扩展属性(FEA)时, Windows SMBv1 容易受到大型非分页内核池内存中存在的缓冲区溢出漏洞攻击。不法分子使用漏洞利用程序 EternalBlue 和勒索软件 WannaCry 造成了迄今为止范围最大的勒索交费行为, 造成损失高达约 80 亿美元, 由此引发了安全研究人员对系统内核的极大关注。根据美国国家漏洞数据库统计, 2015、2016、2017 年内核相关的漏洞分别为 262、554、1195。由于操作系统内核的代码规模迅速增加, 其受到攻击的可能性正逐步扩大, 根据最近的一个报告[2], Linux 内核每天接收约 4 千行代码, 图 1 显示了 Linux 内核代码数量随版本的变化关系。随着代码规模的不断增加, 内核函数的交互关系日趋复杂, 面对千万行代码, 一些内核测试方法由于计算资源的限制, 面临覆盖率小, 执行效率低等问题。内核的模块化测试方法虽然不需要大量的计算资源, 但是这种方法难以检测模块与模块之间的动态执行关系, 无法系统的得到内核测试的完整结果。此外, 与面向用户模式的应用程序不同, 内核工作在内核态, 其本身提供测试软件的运行环境支持, 如果系统崩溃, 测试软件也将崩溃。由于内核的运行状态可随底层硬件的改变而发生变化, 例如经过交叉编译后, Linux 内核可分别运行于 Ubuntu 和 Android, 面对内核运行状态的多样性, 传统的软件测试方法很难移植到内核测试中。

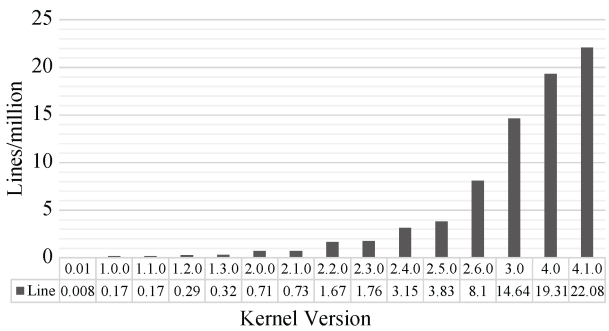


图 1 代码行随版本变化关系

Figure 1 The relationship between line and version

本文在现有的研究基础上, 提出一种基于覆盖率制导[3-4]的操作系统内核漏洞并行挖掘模型。该模型采用内核覆盖率计算工具 kcov 在编译内核时对源码进行插桩, 在对源码进行动态测试时得到覆盖率信息。计算节点根据覆盖率信息, 得到执行不同路径的测试例集合, 此测试例集合通过中央控制节点的收集与处理, 转发给其他的计算节点, 完成各计算节点间测试例的同步。模型充分结合各计算节点发现的路径信息, 突破了计算资源对操作系统内核测试的约束, 实现对操作系统内核的并行化测试。我们利用 Diskaller 与 Syzkaller 和 Triforce 进行对比, 一定条件下 Diskaller 覆盖率较 Syzkaller 提升 12.8%, 执行速率提升 229%, 较 Triforce 覆盖率提升 335%, 执行速率提升 450%。利用 Diskaller, 我们在较新的 Linux 内核中发现两个此前未被发现的 use-after-free 漏洞和多个已知的漏洞。

2 背景与相关工作

当前流行的漏洞挖掘方法主要有静态分析、符号执行、污点分析、模糊测试等, 从实际应用效果上看, 由于动态符号执行和污点分析受限于操作系统的代码复杂度, 因此, 本文在相关工作中侧重介绍静态分析和模糊测试在内核测试的应用情况。

2.1 静态分析

静态分析[5]是指在不运行代码的情况下, 利用词法分析、语法分析、控制流、数据流分析等程序分析手段来检测代码中可能出现的各类漏洞特征。根据程序代码数量、分析目的及粒度的差异, 静态程序分析的复杂程度高低不一, 由于静态分析无需构建动态执行环境, 使其能应用于大规模程序对象, 但程序中出现的指针别名问题、静态控制流图的不完整性问题、精确数据流信息的缺失问题, 也大大增加了静态分析的误报率。

从分析方法上看, 静态分析可分为基于规则和基于验证两种分析手段, 基于规则的静态分析方法是指通过对需要待检测的漏洞相关特性进行规则化, 然后遍历整个或部分预先建立好的代码状态空间, 检测是否存在符合漏洞特征规则的代码片段。在软件代码分析中, 基于规则的方法是静态分析中的主流方法, 它可以通过对大规模复杂代码进行摘要处理或稀疏化, 解决其可应用性。基于验证的静态分析方法通过对程序的正确属性进行抽象, 然后遍历整个程序状态空间, 检查其中是否有违背这些正确属性的情况。这类方法主要有模型检验、定理证明、静态符号执行、抽象解释等方法, 但都受限于状态空间爆炸, 一般作为辅助手段和基于规则的静态分析方法相结合, 以提升分析精度。

传统的内核漏洞静态分析工具, 如 Kint[6], 它首先将源程序的 C 语言代码利用编译器编译为中间语言, 再对编译后的中间语言进行边界检测、范围分析和污点分析等分析操作, 利用得到的条件约束表达式, 借助 STP 约束求解器进行值求解, 如果产生有效解, 则通过污点分析的分类, 输出脆弱点的详细信息以供后续分析。Kint 在 Linux 内核的整数溢出漏洞检测上具有良好的效果, 但它会因为别名分析、循环

展开等问题产生误报, 同时也会因约束求解器的有限求解能力产生漏报。

2017 年, Wang 等人[7]采用静态模型匹配的方法针对 Linux 内核代码的 double fetch 问题进行了一系列研究, double fetch 是内核与用户线程间的系统竞争引发的并发错误(concurrency bug), 包括死锁(deadlock), 原子性违例(atomicity violation), 顺序违例(order violation)等, 作者根据 double fetch 的定义从内核源码中进行模糊匹配, 它匹配多次调用 copy_from_user()或 get_user()等拷贝函数的内核函数, 这些拷贝函数的特点是会从同一地址读取用户数据, 然后根据这些匹配到的特征与细节, 进行精准模式匹配的 double fetch 检测。2018 年 Meng 等人[8]同样提出一种静态分析系统 DEADLINE 用于自动检测内核中的 double fetch 问题, DEADLINE 使用静态程序分析技术系统的在内核中查找多次读取用户内存空间的操作, 并采用专门的符号执行来检测多次读取用户内存空间的 double fetch 问题, 然而这类方法只能检测 double fetch 类型的问题, 具有一定局限性, 同时其静态分析产生的结果并不完全准确, 产生的误报会加大工作量, 因此无法将此方法在全面的内核测试过程中进行展开。

静态分析也常用来辅助动态分析。2012 年, M.J.Renzelmann 等人基于 S2E[9]开发 SymDrive[10]工具用于测试 Linux 和 FreeBSD 驱动程序, 实现了更完备的符号化的硬件来支持对设备驱动的测试, 该系统使用符号执行[11]来消除测试对硬件的需求, 同时在源码级采用静态分析和源到源的代码转换来辅助动态分析及符号执行, 通过修剪同一点产生的过多的分支路径来避免路径爆炸等问题, 减少测试新驱动程序的工作量, 但这种方法主要适用于驱动程序的测试, 同时 S2E 会真实的执行二进制命令, 所以每个测试执行的时间都很长。

2.2 模糊测试

模糊测试[12]是目前最流行的灰盒漏洞检测方法, 其基本思想是: 向待测程序提供大量特殊构造的或是随机的数据作为输入, 监视程序运行过程中的异常并记录导致异常的输入记录。辅助以人工分析, 基于导致异常的输入数据进一步定位软件中漏洞的位置。模糊测试很早就在软件工程中被采用, 最初被称为随机测试, 直到 1988 年, B.Miller 教授提出模糊测试的概念用于验证代码的质量和可靠性。2002 年, D.Aitel 等人[13]发布了一种开放源码的专门用于测试网络协议的模糊测试框架 SPIKE, 该框架采用基于块的方法, 具备描述可变长数据块的能力, 2004 年, Michael Zalewski 发布了针对浏览器进行 Fuzzing 测试的工具 Mangleme[14], 该工具能够不断产生畸形 HTML 脚本并驱动 Web 浏览器进行解析, 同年 M.Eddington 发布一款同时支持网络协议和文件格式的模糊测试框架 Peach。近年来, 随着 Afl[15], Honggfuzz[16]等基于反馈的模糊测试框架兴起, 模糊测试技术已经成为漏洞挖掘技术的主流技术之一。

Linux 内核模糊测试最早可以追溯到 1991 年, 当时 Tin Le 发布了 tsys fuzzer, 仅由 250 行 C 代码组成。这个 fuzzer 的思想非常简单: 它随机挑选几个系统调用, 并赋给他们随机变量。目前这种类型的类型的内核测试框架依旧存在, 它们被统称为随机内核模糊测试。

1997 年, Koopman[17]首次提出基于函数参数类型感知的内核测试方法, 每一种类型, 例如 buffer, length 和 file 等, 都有一系列特定的值。它们根据系统调用的参数类型生成所有可能的测试组合, 并利用这些组合来测试系统调用。2010 年, Trinity[18]通过增加一些随机性来扩展这个生成测试例的想法。例如, 它随机生成一个整数值来作为一个 length 类型的参数, 而 Koopman 只有八个预定义的整数值。这个类别的模糊测试称为基于类型感知的内核模糊测试, Kernel fuzzing, Syzkaller, iknowthis, Triforce Linux Syscall Fuzzer, perf_fuzzer 等都属于这类。

一些模糊测试软件试图在系统运行程序时拦截系统调用, 这类模糊测试被称为基于钩子的内核模糊测试。如: IOCTL Fuzzer[19]通过自带驱动程序 hook 了 windows 特定系统调用 NtDeviceIoControl-File, 从而接管系统所有的 IOCTL 请求, 一旦这个请求符合配置文件定义的条件, IOCTL 请求的原始数据将被 IOCTL Fuzzer 用随机产生的数据替换。类似的模糊测试器还有 IOKit, PassiveFuzzFrame-workOSX 等。由于这类模糊测试器并不会产生具体的测试例, 因此, 就算它们发现了一个内核的异常, 也无法产生一个相应的程序来触发这个漏洞。

另外一些模糊测试器, 例如 Syzkaller[20]、TriforceAFL[21]、Kafl[22]、IMF[23]、Kernel Fuzz[24]等利用覆盖率指导测试函数的生成, 一旦测试例能够产生新的覆盖率, 它们将被加入到一个语料库中, 利用语料库中的测试例循环变异, 不断得到新的覆盖率信息, 这类测试通常被称作覆盖率驱动的内核模糊测试。同样的原理早已经用在比较先进的用户程序模糊测试工具, 如 AFL, Honggfuzz 等。其中 TriforceAFL、Kernel-Fuzzing 等工具利用 QEMU[25]动态插桩方法得到内核覆盖率信息, QEMU 动态插

桩是在二进制指令翻译过程中完成的, 当从 host 提取的指令块被翻译为 TCG 中间表示后, 在将其编译成 host 指定架构的指令时插入额外的分析代码, 这些分析代码在指令执行时将被调用。KAFL 利用基于 Inter PT(Inter Processor Trace,英特尔处理器追踪)实现的 QEMU-PT 追踪分支的跳转得到覆盖率信息, Inter PT 是英特尔架构在收集软件执行轨迹方面的一个扩展功能, 收集的信息包括控制流信息、执行模式等, 它可以在硬件级别追踪 CPU 执行的所有路径。Syzkaller 采用内核支持的 kcov[26]功能收集覆盖率。kcov 覆盖率收集是在一个任务基础上启用的, 因此能够提供准确而全面的覆盖率信息。

2.3 结论

利用静态程序分析技术在内核的漏洞检测上虽然能够取得一定的效果, 但它无法准确的得出代码在动态运行过程中触发的安全漏洞, 对于可能存在的安全漏洞, 还需要进行人工分析与验证, 对研究人员的要求较高。另外, 静态分析技术的误报率会随着代码规模的增加而递增, 其检测结果也具有片面性, 同时静态代码分析会产生状态爆炸问题, 不适用于对内核这类大规模程序进行全面的测试。相比于静态分析技术, 模糊测试技术具有自动化程度高、系统消耗低、误报率低等优点, 已经逐渐成为内核漏洞挖掘的主要方法。图 2 展示了不同种类内核模糊测试工具的发展历程, 从图中可以看出, 近几年来内核模糊测试的主要方法集中在基于类型感知的智能模糊测试上, 在此基础上, 基于覆盖率反馈的模糊测试技术已经成为内核漏洞挖掘的主流方法。

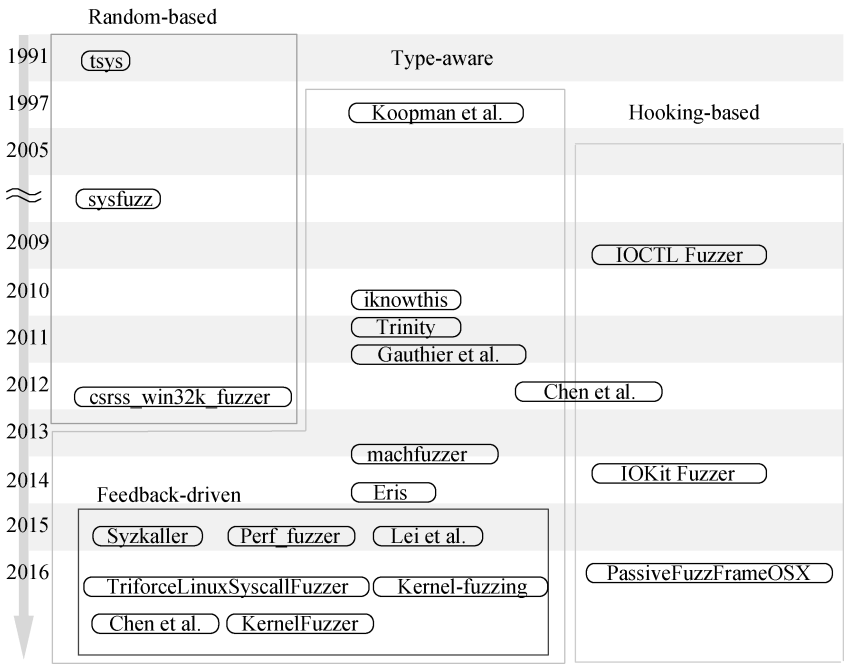


图 2 内核测试发展历程

Figure 2 The history of kernel fuzzing

3 Diskaller 模型设计和实现

Diskaller 是一个无监督的, 基于覆盖率制导的并行内核模糊测试模型, 它基于 Syzkaller 构建, 支持多种操作系统内核的测试, 如: Akaros, FreeBSD, Fuchsia, NetBSD, Linux 等。通过交叉编译, 还可以支持 Android 和 ARM 的测试。模型的总体结构可以分为两部分, 第一部分是 Host-1 到 Host-N 等组成的多个并行计算节点, 每个计算节点运行在单独的主机内, 彼此之间没有直接的数据交互; 第二部分是由 Host-Control 组成的中心控制节点, 控制节点负责从计算节点接收执行出新路径的测试例, 经过计算与处理后同步给其他的计算节点, 完成计算节点的并行数据交互, 整体结构如图 3 所示。

3.1 并行模块设计

并行计算(Paraller Computing)[27]是指同时使用多种计算资源解决计算问题的过程, 是提高计算机系统计算速度和处理能力的一种有效手段。它的基本思想是用多个处理器协同求解同一个问题, 即将被求解的问题分解为若干个部分, 各部分均由一个独立的处理器来并行计算。根据并行计算的思想, 我们利用多个计算节点构成 Diskaller 的并行模块, 由于每个并行模块的执行过程是一致的, 因此我们着重介绍每个并行模块的构成及其与控制节点的数据交互过程。

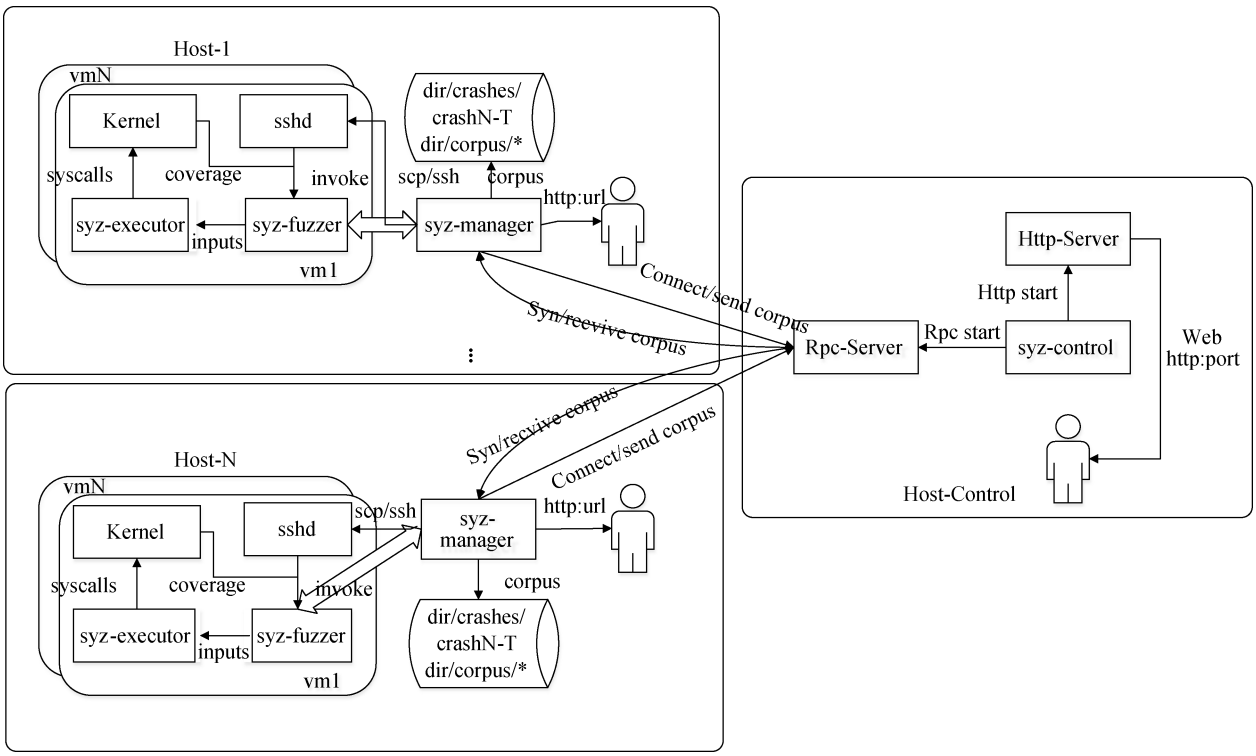


图 3　Diskaller 工作流程

Figure 3　The workflow of Diskaller

3.1.1　计算节点结构设计

计算节点之间的交互过程如图 4 所示, 计算节点由多个 Host 构成, 如图 3Host-1~Host-N, 每个 Host 工作过程如下: 首先, syz-manager 运行于宿主机上, 它启动并监控多个被测的 vm 实例, 每个 vm 实例都是一个虚拟机(如: qemu, kvm), syz-manager 进程通过 ssh 命令登录 vm 实例并通过 scp 命令将可执行二进制 syz-fuzzer 和 syz-executor 文件拷贝到 vm 实例上执行。同时, syz-manager 进程主要负责测试例和路径信息的记录与交互, 它接收每个 vm 实例传入的含有新路径的测试例并将其分发给其他的 vm 实例, 使得每个 vm 实例的语料库能够同步, 同时路径信息的同步避免了不同的 vm 重复执行含有相同路径的输入。另外, syz-manager 根据 syz-fuzzer 执行的输出情况, 结合预先定义的正则表达式来判断一次执行是否产生崩溃, 与 syz-fuzzer 和 syz-executor 不同的是, 它运行在稳定的宿主机环境, 不会受到模糊负载的白噪声干扰。syz-fuzzer 进程运行在不稳定的虚拟机中, 它引导模糊测试过程, 包括测试例的生成与变异。

syz-fuzzer 运行于 vm 实例上, 每次以共享内存的形式将单个测试例传递给 syz-executor 执行, 返回覆盖率信息和程序输出信息, 覆盖率信息通过 RPC(远程过程调用)[28]回传给 syz-manager, 由 syz-manager 统计完新的覆盖率信息后, 将得到的新覆盖率重新返回给 syz-fuzzer, 完成每个 vm 实例上覆盖率信息的同步。与此同时 syz-fuzzer 利用管道将测试例执行得到的输出结果传递给 syz-manager。为了便于 syz-manager 识别一个崩溃信息是由哪一个测试例触发的, syz-executor 将测试例和执行结果一同输出, 建立测试例与输出的一一对应关系。

syz-executor 每次执行都利用 kcov 记录内核函数的执行轨迹, 此执行轨迹经过一系列处理后, 作为路径覆盖率信息以共享内存的形式返回给 syz-fuzzer。用户可以通过网页查看测试例执行数量, 路径数量和崩溃数量等信息。

为了防止测试过程中测试例的执行影响操作系统的正常运行, 每次通过 syz-manager 启动 vm 实例时, 我们都采用暂时性快照(snapshot)的方式, 启动 vm 实例后所有对它的修改都不会被保存。为了排除 vm 实例重启后, IP 地址浮动变化带来的影响, syz-manager 和每个 vm 实例之间都采用 net 端口映射的方式进行网络互通, syz-manager 每启动一个 vm 实例, 都使用计算节点的一个随机未使用的端口映射到 vm 实例的 22 端口, 完成 scp 命令的二进制拷贝和 ssh 命令的传输。这种方式无需指定每个 vm 实例的 IP 地址, 增加了框架的灵活性。

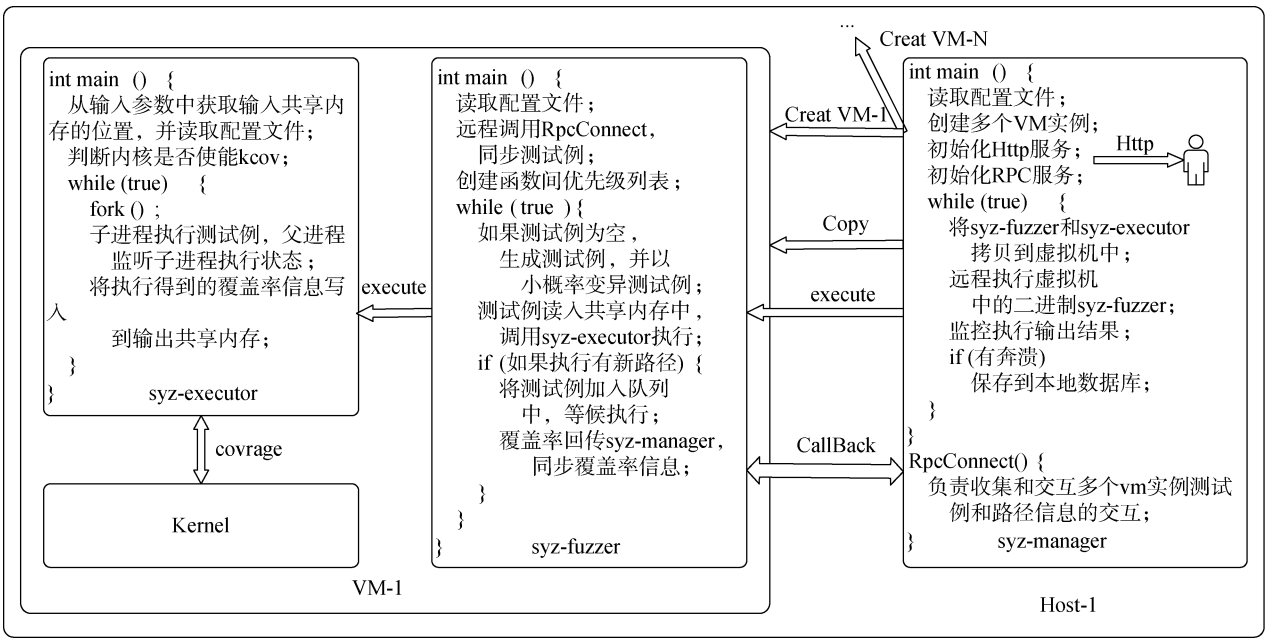


图 4　计算节点执行过程

Figure 4　Execution process of the compute node

3.1.2　控制节点结构设计

syz-control 运行于一台独立的宿主机上, 包含一个 RPC 进程 Rpc-Server 和一个 HTTP 进程 Http-Server。在 syz-manager 请求连接前, Rpc-Server 首先根据对方的 Mac 地址判断此连接是否是一个合法连接, 如果对方 Mac 地址的 md5 加密值与启动 syz-control 时配置的密钥一致, 那么我们认为这是一个合法连接。在一个连接是合法的基础上, syz-manager 会向 Rpc-Server 发送一个连接请求, 此请求会将 syz-manager 的 corpus 信息作为参数发送给 syz-control 的 Rpc-Server 进程, 同时在 Host-Control 进程上新建专属于此连接的文件夹。以 Host-1 主机连接 Host-Control 主机为例, Host-1 上的 syz-manager 向 syz-control 的 Rpc-Server 发出连接请求, 那么将在 Host-Control 主机工作目录下创建 Host-1 目录和一个保存全局 corpus 数据库, Host-1 目录用于记录此请求传入的每个测试例的摘要信息。摘要信息是通过单个测试例的 hash 运算得到的, 在数量上与连接 Rpc-Server 的计算节点中保存的 corpus 数量保持一致。如果传入的测试例在全局 corpus 数据库中不存在, 那么它同时会将完整的信息添加到全局 corpus 数据库中。

当连接请求结束, syz-manager 会向 Rpc-Server 发送一个同步请求。由于连接请求和同步请求之间存在时间差, 在这段时间差内每个 syz-manager 的 corpus 数量可能会发生变化: 当出现新的路径时, corpus 会增加; 当 corpus 中某个测试例的优先级低于后发现的一个测试例时, 这个测试例将会被优先级高的测试例替换, 因此同步请求将这些变化的测试例信息作为参数, syz-control 宿主机上 Host-1 目录中的 corpus 摘要信息和全局 corpus 数据库将根据这些变化进行更新, 更新的同时将全局数据库排除掉 Host-1 目录 corpus 信息的全局数据库记录值返回给 syz-manager, 完成 corpus 的同步过程。

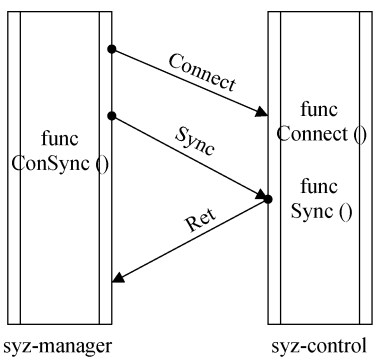


图 5　连接和同步过程

Figure 5　Process of Connection and Synchronization

3.1.3　并行数据交互

我们以图 5 为例阐述 syz-manager 与 syz-control 之间的数据交互原理。func ConSync() 函数是 syz-manager 中的一个函数, 它是一个单独的线程, 其主要作用是远程调用 syz-control 中的函数 func Connect()和函数 func Sync(), 调用 Connect()函数时传递的参数为 syz-manager 已有的 corpus 信息。func Connect()函数收到调用命令后, 将此连接传入的

corpus 提取摘要保存在单独记录此连接的目录中, 同时去重后保存在全局 corpus 数据库中, 记录摘要信息的目的是防止从此连接得到的 corpus 重新返回给 func ConSync()函数。连接过程结束后, func ConSync()函数统计 syz-manager 中变化的 corpus, 并将其作为参数传递给函数 func Sync(), func Sync()收到参数后, 更新此连接的数据库和全局数据库, 并将全局数据库排除在摘要信息中的测试例后, 返回给 syz-manager, 返回的值将被存入队列中等待执行。

具体考虑图 6 的情景, 假设 Connect 阶段每个 Host 给控制节点传递的测试例数量分别为 1700、1800、1900, 其中 Host-1 和 Host-2 重复 500 个, Host-1 和 Host-3 重复 600 个, Host-2 和 Host-3 重复 700 个, 三个 Host 共同重复 400 个, 那么存入控制节点全局数据库的测试例数量为 4000 个, 此时开始同步阶段。同步阶段若每个 Host 新增的测试例数量为 400、500、600, 删除的测试例数量为 100、200、300, 那么每个 Host 会根据这些变化信息来更新自身的 corpus 摘要信息, 其中每个 Host 目录所记录的 corpus 摘要将变为 2000、2100、2200, 全局数据库根据每个 Host 目录的 corpus 摘要信息进行增减, 变为 4600 个(假设新增的测试例有 300 个是重复的)。此后每个 Host 的摘要信息与全局数据库进行比较, 将不在摘要信息中的测试例作为新的测试例返回, 此时 Host-1、Host-2、Host-3 分别返回 2600、2500、2400 个 corpus。交互完成之后, 一直循环同步阶段。

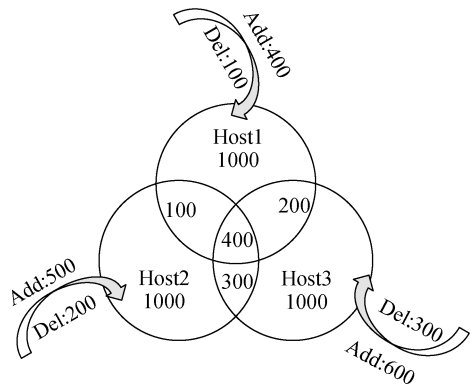


图 6 Host 间数据交互

Figure 6 Data interaction diagram between Hosts

3.2 覆盖率制导模型设计

3.2.1 基本块覆盖模型

Diskaller 以基本块(base block)为最小单位计算覆盖率信息, 通过将程序划分为一组代码段, 每一段代码中的第一条语句被执行, 则整段代码都被执行一次, 这样的代码段称为基本块。通常基本块具有如下性质:

1. 只有一个出口语句和一个入口语句
2. 执行时只能从入口语句入, 出口语句出
3. 同一基本块所有语句执行次数相同

根据基本块的性质, 我们可以判断基本块的最后一条语句一定是一个跳转语句, 其跳转目的地址是另一个基本块的第一条语句。如果此跳转包含在一个条件语句中, 那么该基本块就有两个基本块作为目的地址, 因此若把所有基本块当作一个节点, 那么一个程序中所有基本块构成一个有向图, 称为基本块图, 将一个基本块到相邻基本块的跳转, 称为有向图的边, 任意两个可达基本块所经过的边的组合, 称之为路径。为了避免路径爆炸, 我们规定一条新路径的发现与经过的边的组合顺序无关, 仅与此路径是否含有新的边有关。

3.2.2 覆盖率收集

Diskaller 覆盖率制导是基于 kcov 实现的, 相比于其他覆盖率收集方法, 通过 kcov 收集覆盖率信息具有准确度高, 灵活性强, 性能损耗低的优点, 同时 kcov 还支持数据流引导的模糊测试, 它通过 trace-cmp 选项动态追踪汇编的 cmp 指令, 得到它的两个操作数, 利用操作数的替换, 我们可以快速发现一条新的路径。通过 GCC 编译器在编译内核阶段启用 CONFIG_KCOV=y 选项, GCC 会将一个函数调用插入每个内核的每个基本块中, 图 7 展示的是插桩前的代码与插桩后的代码, 如果开启了 trace_cmp 选项, 编译器还会在 cmp 指令附近插入额外的代码进行标志。执行过程中, 如果插入的函数被执行到, 那么它将返回一个插入函数计算出的数值用于标志这个基本块, syz-executor 每执行一次系统调用都会收集一系列的基本块的信息。

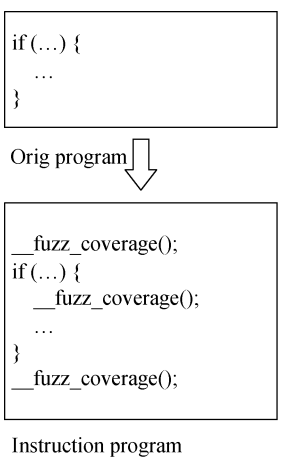


图 7 原始代码与插桩代码

Figure 7 Origin program and instruction program

假设 0xffffffff8100206a 是__fuzz_coverage()返回的一个标志基本块的值, kcov 将多次执行函数__fuzz_coverage()得到的形如 0xffffffff8100206a 的返回值存入数组 cover_data[]中。为了更详细的记录一条路径的信息, Diskaller 采用类似 Afl 块间转移的概念, 块间转移核心算法如下:

```
uint32_t prev = 0;
for (uint32_t i = 0; i < th->cover_size; i++) {
    uint32_t pc = (uint32_t)th->cover_data[i];
    uint32_t sig = pc ^ prev;
    prev = hash(pc);
    if (dedup(sig))
        continue;
    write_output(sig);
    nsig++;
}
…
static uint32 hash(uint32 a){
    a = (a ^ 61) ^ (a >> 16);
    a = a + (a << 3);
    a = a ^ (a >> 4);
    a = a * 0x27d4eb2d;
    a = a ^ (a >> 15);
    return a;
}
```

每次循环都从 cover_data[]数组中取一个标记基本块的值, 与前一个基本块标记的哈希值进行异或运算得到唯一的 sig, 作为块间转移的标志, 此标志即上文提到的基本块图的有向边。哈希运算的目的是为了区分两个相同块之间的方向, 假如有 A, B 两个块, 如果不进行 hash()运算, 那么 A^B 的结果将和 B^A 的结果一致, 而实际上 A->B 和 B->A 是两条不同的边。write_output(sig)将此基本块间的转移信息通过共享内存传递给 syz-fuzzer。

syz-fuzzer 根据 syz-executor 返回的 sig, 判断此返回的 sig 是否含有新的边, 如果含有一条新的边, 将其添加到 maxSignal 中并保存产生此边的输入。对于那些没有产生新块间转移的输入将会被抛弃。这种算法考虑了一个非常细粒度的、长期的对程序状态的探索, 同时它不必执行复杂的计算, 不必对整个复杂的执行流进行对比, 也避免了路径爆炸。为了说明这个算法是如何工作的, 我们考虑如下两条路径:

#1: A->B->C->D->E

#2: A->B->C->A->E

第一条路径将会产生 maxSignal(AB, BC, CD, DE), 其中 A, B, C, D, E 表示基本块, AB, BC, CD, DE 表示基本块间的转移。由于第二条路径产生了新的块间转移 CA, AE, 所以第二条路径将被视作一条新路径。同时更新 maxSignal (AB, BC, CD, DE, CA, AE), 那么对于如下一条路径将不会被 Diskaller 视为一条新路径, 尽管它看起来是条不同的执行路径:

#3: A->B->C->A->B->C->A->B->C->D->E

3.3 输入模型设计

为了避免大量无效的测试例, Diskaller 根据函数参数类型自动生成输入测试例, 参数类型由函数调用描述文档确定, 函数描述文档是根据系统头文件和系统调用参数类型预先定义的。我们以 Linux 的函数 read()为例, 介绍 Diskaller 从它开始产生一个测试例的完整过程。

首先, read()函数在 Linux 中的定义为: ssize_t read(int fd, void *buf, size_t count)。它的第一个参数是一个资源描述符类型, 第二个参数是一个指向内存缓冲区的指针, 第三个参数是长度类型, 表示缓冲区的长度。我们根据如下的语法规则编写描述文档:

```
syscallname "(" [arg ["," arg]*] ")" [type]
arg = argname type
argname = identifier
type = typename [ "[" type-options "]" ]
typename = "const" | "intN" | "intptr" | "flags" |
   "array" | "ptr" |"buffer" | "string" | "strconst"|
   "filename" | "len" |"bytesize" | "bytesizeN " |
   "bitsize" | "vma" | "proc"
type-options = [type-opt ["," type-opt]]
```

a) syscallname表示函数名, 在这里为read; b) arg表示read函数的参数, 可以有多个, 每个arg包含argname和type, 其中argname是由编写描述文档者任意定义的; c) type表示参数类型, 或者函数的返回类型, 它由typename和type-options构成。typename可以由16种预先定义的类型组成, 也可以由函数返回的特定类型组成, type-options表示typename的属性。

根据如上定义, 我们可以将read的描述文档写为read(fd fd, buf buffer[out], size len[buf]), 第一个参数 fd fd表示argname为fd, type为fd, 由于fd不在我们定义的16种类型中, 因此fd类型需要其他函数生成。第二个参数buf buffer[out], argname为buf, type为buffer, type-options为out, 其中buffer是预先定义的类型, 表示的就是一个指向内存缓冲区的指针, 指向的空间大小是随机分配的。out表示方向, 是针对read函数而言的, 表示read函数将从fd读取的数据输出(写入)到

buffer中, buffer的内容并不会影响到read函数的执行, 所以我们无需为它生成初始值, 也就是说, 方向可以决定我们是否需要给此参数生成初始值。如write函数的第二个参数我们定义为buf buffer[in]表示我们需要将buffer的内容输入到fd所指示的文件, 其内容会影响到write函数的执行过程, 因此我们应该给它一个随机初始值。read函数的第三个参数size表示大小, 类型为len, 也在预先定义的16中类型当中, 其type-options指定其他参数的argname, 表示指定参数的大小, 这里用来表示buf指向空间的大小。

由于 read 函数的第一个参数是一个资源描述符类型, 不在自定义的类型当中, 因此我们需要在函数列表中, 选择一个返回值类型为 fd 的函数, 这样的函数有很多个, 我们以优先级为概率从中随机选择一个, 假如此处选中 open 函数。open 函数在 linux 中的定义为: int open(const char * pathname, int flags, mode_t mode)。第一个参数为指向文件路径的指针, 第二个参数为文件的打开方式, 第三个参数是文件的读写权限。我们以同样的方式编写描述文档: open(file ptr[in, filename], flag flags [open_flags], mode flags[open_mode]) fd, 第一个参数的参数名为 file, 参数类型 ptr, 方向为 in, 指向的类型为 filename, 即指向文件名; 第二个参数名为 flag, 参数类型 flags, 属性为 open_flags, 这里 open_flags 不在预习定义的类型当中, 因此是一个自定义的属性, 其值由常量构成; 第三个参数名为 mode, 参数类型 flags, 属性为 open_mode, 也是一个自定义常量属性。通过上述步骤, 我们得到如下的描述文档:

```
open(file ptr[in, filename], flag flags
    [open_flags], mode flags[open_mode]) fd
read(fd fd, buf buffer[out], size len[buf])
open_flags=O_RDONLY,O_WRONLY,O_O
    DWR, O_APPEND, FASYNC …
open_mode=S_IRUSR,S_IWUSR,S_IXUSR,
    S_IXUSR, S_IRGRP, S_IWGRP …
```

根据上述描述文档, Diskaller 将能够自动生成如下的测试函数集:

```
r0 = open(&(0x7f0000000000)= "./file0", 0x3, 0x9)
read(r0, &(0x7f0000000140)= "", 0x42)
```

生成的 open 函数第一个参数是一个指针类型, 指向 filename, 对比 Linux 中 open 函数的第一个参数类型 char *, 我们缩小了第一个参数的范围, 使得其只能产生标准的指向文件名的指针, 畸形测试例则由变异函数生成, 这样能够减少无效参数的干扰, 增大了函数参数的命中率。空间的申请和初始值的设定都交由处理 ptr 类型的函数完成, 这里的初始值为“./file0”。open 函数的其他两个参数 flags 和 mode 根据选项 open_flags 和 open_mode 的值取随机个数异或, fd 类型的返回值 r0 将被用于 read 函数。read 函数第一个参数 r0 在执行时会被 open 函数的返回值替换, 第二个参数是指针类型, 这个指针存放在空间 0x7f0000000140, 大小为 0x42, 第三个参数表示第二个参数的大小。

由于测试例是根据参数类型生成的, 而每种类型都在描述文档中预先定义, 通过这种方式生成的测试例排除了大量的无效输入。对于生成的测试集, sys-fuzzer 将采用无限循环的方式进行测试, 每生成一个测试例, 将传递给 syz-executor 程序执行, 如果某一个函数调用有新的覆盖率产生, 则会将此测试例加入到语料库中, 并继续下一次测试。

3.4 脆弱性监测模型设计

为了监测程序动态执行过程中是否产生奔溃, 面向用户程序的漏洞挖掘工具通常的操作是 fork()子进程, 子进程执行测试例进行动态测试, 父进程监听子进程的执行状态, 根据子进程执行的返回结果来判断此次执行是否发生异常。相比于面向用户程序的漏洞挖掘工具, 内核漏洞挖掘工具更难监控每次测试的执行结果, 因为内核一旦崩溃, 依赖内核运行的监控进程将会失效。

为了实时监测执行结果, 判断每次执行是否有 BUG 发现, syz-manager 通过管道方式记录每次的执行输出信息。我们采用如下的策略判断输出的信息中是否含有可能存在的 BUG: 首先对输出的信息进行字符匹配, 如果输出结果匹配预先定义的字符串, 那么我们再对此输出进行正则匹配, 用于提取详细的信息, 接下来将提取的地址信息用 addr2line 工具转换为源码位置信息, 提供用户可理解的程序执行轨迹, 最后将这个测试例和日志存入 syz-manager 的 crash 目录中, 部分匹配规则见表 1。如果测试过程中, 内核异常结束后没有任何输出, 我们通过触发超时时间来控制 vm 实例的重启, 同时保存奔溃前执行得到的日志信息。

4 实验设计与测试分析

4.1 实验设计

为了验证 Diskaller 的有效性, 本文将此模型与 Syzkaller、TriforceLinuxAfl 进行了对比。实验中所

表 1 输出结果正则匹配规则

Table 1 Output regular matching rules

Type	String	Regular Expression
BUG	BUG KASAN:	BUG: KASAN: ([a-z\\-]+) in {{FUNC}}(?:.*\\n)+?.*(Read|Write) of size (?:[0-9]+)
	BUG KASAN	"BUG: KASAN: double-free or invalid-free in {{FUNC}}
	BUG: workqueue leaked lock	BUG: workqueue leaked lock or atomic(?:.*\\n)+?" + ".*last function: ([a-zA-Z0-9_]+)\\n
	BUG: .* locks	BUG: .*still has locks held!(?:.*\\n)+?.*{{PC}} +{{FUNC}}
kernel panic	Kernel panic	Kernel panic – not syncing: Couldn't open N_TTY ldisc for [^]+ --- error –[0-9]+"
	Kernel panic	"Kernel panic – not syncing: (.*)"
kernel BUG	kernel BUG	kernel BUG at {{FUNC}}/([a-zA-Z0-9_]+)\.c
	Kernel BUG	Kernel BUG (.*)
kmalloc BUG	BUG kmalloc-	BUG kmalloc-.*: Object already free
divide error	divide error:	divide error: (?:.*\\n)+?.*RIP: [0-9]+:(?:{{PC}} +{{PC}} +)?{{FUNC}}
ubsan	UBSAN:	"UBSAN: (.*)"

(注: 其中 PC 表示"\\[\\<[0-9a-f]+\\>\\]", FUNC 表示"([a-zA-Z0-9_]+)(?:\\.\\+)")

有测试操作系统都为 Ubuntu16.04, 被测内核为 Linux v4.16。实验将处理器数量模拟为计算资源, 由于实验需要的内存较小, 故此次对比实验中不予考虑内存的变化, 统一设置为 8G。为了验证 Diskaller 并行挖掘模型的有效性与实用性, 我们分别进行了三组实验, 并给每组实验的 Diskaller 都分配四台 Host。第一组实验, Diskaller 每台 Host 处理器的数量均为 1, Syzkaller 和 TriforceAFL 处理器的数量都为 4; 第二组实验, Diskaller 每台 Host 处理器的数量均为 2, Syzkaller 和 TriforceAFL 处理器数量为 8; 第三组实验, Diskaller 每台 Host 处理器数量均为 4, Syskaller 和 TriforceAFL 处理器数量为 16。为了模拟计算资源不足的情况, 我们给每组实验的 Syzkaller 和 TriforceAfl 添加处理器数量为 2、4、8 的对比实验。我们将上述情形以表二的形式列出: 当需要的总处理器数量为 4、8、16 时, Diskaller 分别提供 4*1、4*2、4*4 个处理器, 其中 A*B 中的 A 表示 Host 的数量, B 表示每个 Host 时分配给它们处理器的数量, 作为对比, 其他两个工具设置了两组对比实验, 分别是 1*4、1*8、1*16 和 1*2、1*4、1*8, 它们作为计算资源充足和计算资源不足时的对比数据。

表 2 软件处理器数量配置

Table 2 Number of processors software configure

Soft	Core-处理器					
	4		8		16	
Diskaller	4*1		4*2		4*4	
Syzkaller	1*4	1*2	1*8	1*4	1*16	1*8
Triforce	1*4	1*2	1*8	1*4	1*16	1*8

测试过程中, 主要选用执行速率、路径覆盖率和脆弱性数量来作为主要的比较对象。选用这些指标作为比较对象的原因是:

执行速率代表单位时间内执行测试例的数量, 它能反映一个软件整体的性能, 执行速率越快, 相同的时间内可能发现的 BUG 数量越多。对于模糊测试来说, 准确的生成符合语法规则的测试例固然重要, 但是如果因此而花费的开销过大, 可能会得不偿失。

路径覆盖率是度量测试完整性的一个手段, 它可以反映软件的整体执行效果。如果执行过程中路径覆盖率越高, 代表发现的路径越多, 可能发现 BUG 的概率也就越高。

脆弱性数量是一个模糊测试器最重要的指标之一, 也是设计模糊测试器的意义所在。脆弱性的数量越多, 代表漏洞的数量可能越多。

4.2 测试结果分析

首先, 为了验证计算资源确实会影响测试工具的性能, 我们做了如下的对比实验: 通过设置 Syzkaller 的处理器个数分别为 2、4、6、8 时, 对比得到的执行速率、覆盖率(这里用发现新边的数量代表)及路径的数量, 实验结果如图 8。实验表明, 处理器数量越多, Host 所能够运行的 VM 实例越多, 相同时间内所发现的执行速率、覆盖率及路径数量都有一定提升, 说明计算资源对测试工具的整体性能有较大影响。

Diskaller 计算节点 Host 之间通过控制节点交换覆盖率信息, 因此每个计算节点间的路径信息和覆盖率信息都具有高度一致性。图 9 展示了当每个 Host 处理器数量为 2 时, 四个 Host 间路径信息和覆盖率信息的随时间的变化过程, 可以看出: 四个计算节点间的路径信息及覆盖率信息的变化趋势基本是一

致的, 为了便于观测, 后面的计算中出现的 Diskaller 路径数量、覆盖率信息及执行速率都以各计算节点的平均值代表。

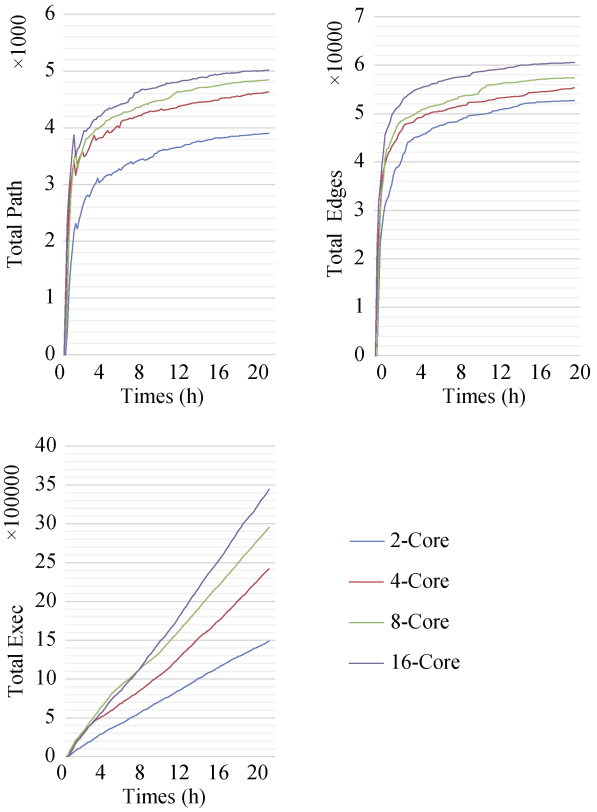


图 8　处理器数量对 Syzkaller 的影响

Figure 8　The influence of the number of processors on Syzkaller

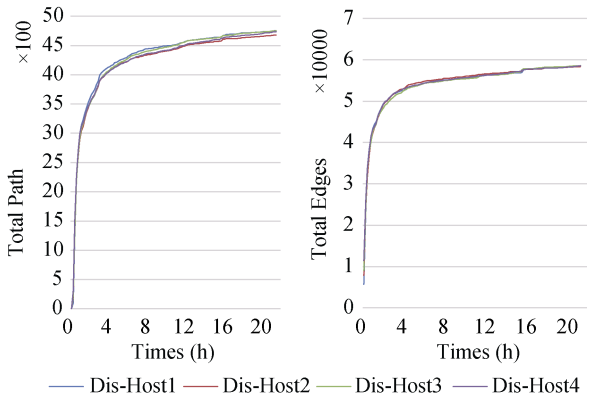


图 9　Diskaller 各 Host 执行趋势

Figure 9　The execution trend of each Host

根据上述设计的三组实验, 我们统计了持续测试 20 小时的结果, 如表 3 所示。从中可以看出, 当处理器数量较小时, Diskaller 性能较 Syzkaller 提升较低; 当处理器数量增大时, Diskaller 的性能较 Syzkaller 性能逐步较大。产生这个现象的主要原因是: 当计算资源较低时, 影响覆盖率和执行速率的主要因素是计算资源, 此时 Diskaller 和 Syzkaller 都面临计算资源低的问题, 因此性能差距不明显; 当计算资源增大, 影响覆盖率和执行速率的因素变为计算资源和计算节点中的数据竞争所带来的性能损耗, 由于 Diskaller 将测试过程分布在不同的计算节点中, 因此其数据竞争较低, 因此性能提升较为明显。为了进一步体现 Diskaller 与 Syzkaller 和 Triforce 的性能差异, 我们以 10 分钟为间隔, 对比 Diskaller、Syzkaller、Triforce 路径数量、覆盖率和测执行速率的走势情况, 测试结果如图 10。

表 3　测试结果对比

Table 3　Comparison of test results

Soft		Core-处理器			
		2	4	8	16
Diskaller	Corpus	—	4647	4773	5425
	Execs	—	4628544	6285626	9738920
	Edges	—	56890	58561	64758
	Crash	—	0	1	2
Syzkaller	Corpus	3907	4630	4810	5020
	Execs	1489782	2421519	2956379	3445985
	Edges	52693	55286	57409	60529
	Crash	0	0	1	1
Triforce	Corpus	921	1062	1189	1247
	Execs	973653	1215761	1524627	1615761
	Edges	10287	12637	14468	15281
	Crash	0	0	0	0

4.2.1　执行速率提升

根据表三, 我们可以计算出在处理器数量分别为 4、8、16 的情况下, Diskaller 四个 Host 的总执行速率分别为 231427、314271、486946 个/h, 单个 Host 执行速率约为 57856、78567、121736 个/h, 在处理器数量为 2、4、8、16 的情况下, Syzkaller 执行速率分别为 74489、121076、147818、172299 个/h, TriforceAFL 执行速率分别为 48682、60788、76231、80788 个/h。由此可以得出, 在资源充足的情况下, Diskaller 总执行速率较 Syzkaller 分别提升 91.1%、112.6%、182.6%; 较 TriforceAFL 分别提升 280%、312.2%、502.7%; 在资源不足的情况下, Diskaller 总执行速率较 Syzkaller 分别提升 210.6%、159.6%、229.4%。这里由于 TriforceAFL 采用 QEMU 动态插桩, 在执行速度上和 Diskaller 具有一定差距。

4.2.2　路径覆盖率提升

覆盖率计算过程中, 我们将边的数量模拟为覆盖率, 因为边的数量可以体现代码的覆盖程度。根据表三, 我们可以得到在处理器数量分别为 4、8、16 时, Diskaller 的覆盖率较 Syzkaller 分别提升 2.91%、2.01%、6.98%; 在模拟计算资源不足的情况下, 覆盖

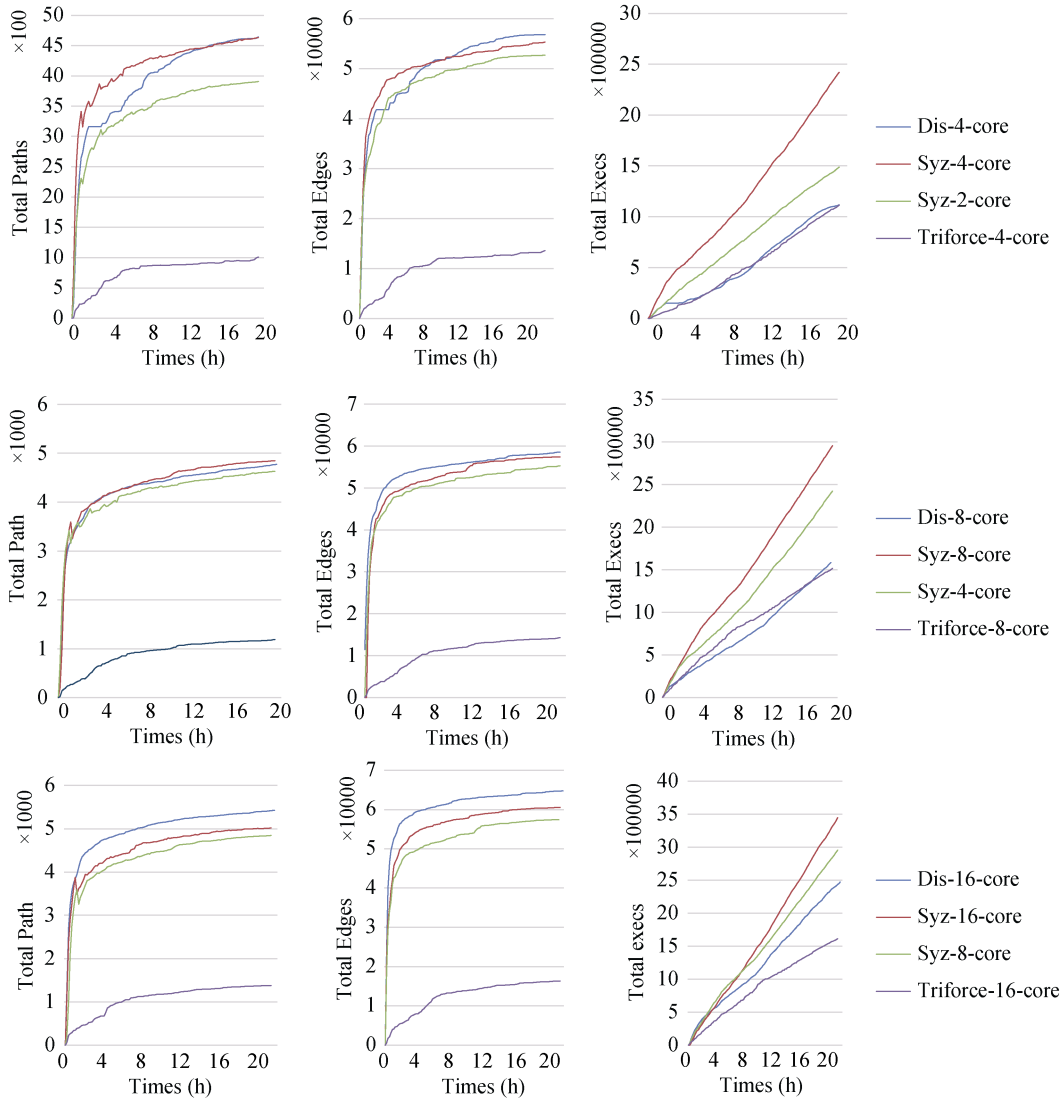


图 10　测试实时效果对比

Figure 10　Efficiency comparsion of test tools

率提升 7.96%、5.92%、12.8%。覆盖率较 TriforceAFL 提升 350%、305%、324%。可以看到, 在处理器数量增多时, Diskaller 覆盖率的提升效果更加明显, 这是因为处理器数量增多时, 每台 Host 启动的虚拟机数量增多, Syzkaller 和 TriforceAFL 内部存在大量的数据交互、数据竞争与互斥锁, 当启动的 vm 实例数量增加时, 互斥锁的存在会降低执行效率。而 Diskaller 将同一系统内的竞争行为分散到多个系统中, 因此对覆盖率的提升有更加明显的效果, 从理论上来说, Diskaller 能够无限的提升计算资源。同 Diskaller 相比, TriforceAFL 在相同的时间内发现的路径数量更少, 因为其输入测试例是随机生成的。

4.2.3　脆弱性数量提升

通过持续运行 Diskaller 共计 72 小时, 我们发现了多处已被发现的内核漏洞和少量之前未被发现的内核漏洞, 其中通过 KASAN[29]发现的多个 net/ipv4 模块的 use-after-free 漏洞[30-31]已被厂商修复, 并发现两个此前未被公开的漏洞[32-33]。通过人工分类排除重复的崩溃, 共发现堆溢出漏洞 3 个, 空指针异常 1 个, 栈溢出 1 个, 系统死锁 1 个, 统计结果见图 11。

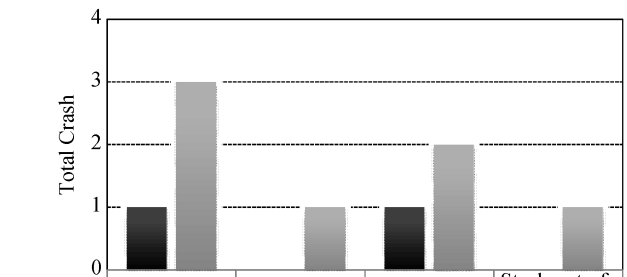


图 11　漏洞数量统计

Figure 11　Distribution of the vulnerabilities

5 结论

本文结合模糊测试技术, 提出一种基于覆盖率制导的并行操作系统内核漏洞挖掘方法, 并用这种方法和 Syzkaller 进行性能对比, 实验结果表明这种方法比现有漏洞挖掘工具速率更快, 效率更高, 覆盖率更好, 自动化更强的优势, 具有一定的实用性与有效性。当并行规模的范围扩大时, 通过分布式漏洞挖掘的方法依旧存在不足, 主要问题在于计算节点增多时, Control 主机同时处理返回的数据存在性能损耗, 所以在下一步研究中, 将重点考虑多主目标的数据交互的性能优化。

参考文献

[1] “Microsoft Security Bulletin,” Microsoft, https://docs.microsoft.com/zh-cn/security-updates/Securitybulletins/2017/ms17-010, Oct. 2017.

[2] “Linux Kernel Development,” Jonathan Corbet and Greg KroahHartman, http://go.linuxfoundation.org/linux-dev-report-2016, Oct. 2016.

[3] Nigel Hinds, Paul Larson, Hubertus Franke, and Marty Ridgeway, “Using Code Coverage Tools in the Linux Kernel,” in *ACM SIGSOFT Software Engineering Notes(ACM’04),* pp. 70-81, 2014.

[4] Ian Beer, “Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” in *Proceedings of the USENIX Symposium on Operating System Design and Implementation(USS’05)*, pp.29-46, 2008.

[5] Gerhard Bohm and Günter Zech, “Introduction to Statistics and Data Analysis for Physicists,” Chicago: American Academic Press, Mar. 2004.

[6] Wang Xi, Chen H, Jia Z, et al, “Improving integer security for systems with KINT,” in *Proc. of the USENIX Conference on Operating Systems Design and Implementation (USS’10),* pp. 163-177, 2012.

[7] Pengfei Wang, Jens Krinke, Kai Lu and Gen Li, “How Double-Fetch Situations turn into Double-Fetch Vulunerabilities: A Study of Double Fetches in the Linux Kernel,” in *Proc. of the 26th USENIX Security Symposium(USS’26),* pp. 102-119, Aug. 2017.

[8] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backs and Taesoo Kim, “Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels,” in *Proc. of IEEE Symposium on Security and Privacy(S&P),* pp: 270-287, 2018.

[9] Chipounov V, Kuznetsiv V, Candea G, “S2E, A Platform for In-Vivo Multi-Path Analysis of Software Systems,” in *ACM SIGARCH Computer Architecture News*, pp. 265-278, 2011.

[10] Matthew J, Asim Kadav, and Michael M, “Symdrive: Testing Drivers without Devices,” in *Proc. of the 10*th *USENIX Symposium on Operating Systems Design and Implementation*, pp. 102-116, Oct. 2012.

[11] James C. King, “Symbolic Execution and Program Testing,” in *Communications of the ACM*, pp. 385-394, Jul. 1976.

[12] Richard McNally, Ken Yiu, and Duncan Grove, “Fuzzing: The State of the Art,” Australia: DSTO Defence Science Organisation, Jan. 2012.

[13] Dave Aitel, “An Introduction to SPIKE”. the Fuzzer Creation Kit. L.A, Black Hat, 2003.

[14] “Mangleme, an automated broken HTML generator and browser tester,” Michal Zalewski, http://www.securityfocus.com/archive/1-/378632, Dec. 2009.

[15] “American fuzzy lop: A novel type of compile-time instrumentation fuzzer,” Michal Zalewski, http://lcamtuf.coredump.cx/afl, May. 2015.

[16] “Honggfuzz: A security oriented, feedback-driven, evolutionary, easy-to-use fuzzer,” Google Group, http://honggfuzz.com, Jan. 2015.

[17] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz, “Comparing Operating Systems Using Robenchmarks,” in *Proc. of the Symposium on Reliable Distributed Systems*, pp. 317-332, Jan. 1997.

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

[19] “IOCTL fuzzer,” Dmytro Oleksiuk, https://www.Dytro.com/cr4sh/ioctlfuzzer.pdf, Nov. 2011.

[20] “Syzkaller: the next generation of kernel fuzzer,” Dmitry Vyukov, https://www.slideshare.net/Dmitry/syzkaller-the-next-gen-kernel-fuzz-er, May. 2017.

[21] “Project Triforce: Run AFL on Everything,” ncc group, http://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything, Jun. 2016.

[22] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz, “kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels,” in *Proc. of the USENIX Security Symposium(USS’26)*, pp. 391-402, Aug. 2017.

[23] HyungSeok Han, and Sang Kil Cha. “IMF: Inffered Model-based Fuzzer,” in *The ACM Conference on Computer and Communication Security(CCS’02)*, pp. 275-289, Nov. 2017.

[24] “Windows Kernel-Fuzzing,” Oracle, http://gsec.hitb.org/meterials/sg2016/fuzzing_the_windows_kernel.pdf, Oct. 2016.

[25] Fabrice Bellard, “QEMU, a Fast and Portable Dynamic Translator,” in *Proc. of the ACM on USENIX Annual Technical Conference,* pp. 27-43, Dec. 2005.

[26] “kcov: code coverage for fuzzing,” The Linux Kernel, http://www.kern el.org/doc/html/v4.13/dev-tools/kcov.html, 2016.

[27] Andrew Tanenba. “Distributed Systems Principles and Paradigm,” Chicago: Harper Collins Publishers, 2006.

[28] Andew Birrell and Bruce Nelson, “Implementing Remote Procedure Calls,” in *ACM Transactions on Inteligent System and Technology(ACM)*, pp. 37-49, 1983.

[29] “The Kernel Address Sanitizer(KASAN),” Linux Kernel Group, https://www.kernel.org/doc/html/v4.11/dev-tools/kasan.html, 2014.

[30] Patchwork. Fix use-after-free of ldt_struct. https://patchwor. kernel-.org/patch/9921341/, Oct. 2017.

[31] Andrey Konovalo. “Use After Free In Addr_grec”. https://groups.google.com/forum/#!topic/syzkaller/dlHu8uuZWfg, May. 2017.

[32] “slab out-of-bounds write in kernel/trace/trace_events_filter.c,”

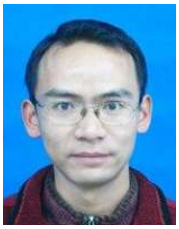
icytxw,htttp://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-12714, 2018.

[33] "Integer Overflow in kernel/time/posix-timers.c," icytxw, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-12896.

涂序文 于 2016 年在江南大学计算机科学与技术专业获得学士学位。现在江南大学计算机科学与技术专业攻读硕士学位。研究主要领域为系统与软件安全方向, 研究兴趣包括: Linux 软件漏洞挖掘, 操作系统内核漏洞挖掘等。Email: icytxw@gmail.com

甘水滔 博士, 主要研究领域为网络安全, 系统安全软件理论及安全性分析。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。现任江南大学副教授。主要研究方向为网络仿真、网络安全、网络虚拟化等。曾作为主要研究人参与完成国家 973、国家 863 等重点项目。Email: wangxf@jiangnan.edu.cn



陈爱国 博士, 主要研究方向为模式识别, 机器学习, 人工智能和漏洞挖掘。Email: chenag@jiangnan.edu.cn