软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(12):5921−5939 [doi: 10.13328/j.cnki.jos.006762]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于 PKS 硬件特性的 eBPF 内存隔离机制*

李 浩, 古金宇, 夏虞斌, 臧斌宇, 陈海波

(上海交通大学 软件学院, 上海 200240)

通信作者: 古金宇, E-mail: gujinyu@sjtu.edu.cn

摘 要: Linux 内核中的 eBPF (extended Berkeley packet filter) 机制可以将用户提供的不受信任的程序安全地加载到内核中. 在 eBPF 机制中, 检查器负责检查并保证用户提供的程序不会导致内核崩溃或者恶意地访问内核地址空间. 近年来, eBPF 机制得到了快速发展, 随着加入越来越多的新功能, 其检查器也变得愈发复杂. 观察到复杂的 eBPF 安全检查器存在的两个问题: 一是“假阴性”问题: 检查器复杂的安全检查逻辑中存在诸多漏洞, 而攻击者可以利用这些漏洞设计能够通过检查的恶意 eBPF 程序来攻击内核; 二是“假阳性”问题: 检查器采用静态检查的方式, 由于缺乏运行时信息只能进行保守检查, 可能造成原本安全的程序无法通过检查, 也只能支持很受限的语义, 为 eBPF 程序的开发带来了困难. 通过进一步分析, 发现 eBPF 检查器中的静态模拟执行检查机制代码量大, 复杂度高, 分析保守, 是引起安全漏洞和误报的主要原因. 因此, 提出使用轻量级动态检查的方式取代 eBPF 检查器中的静态模拟执行检查机制, eBPF 检查器中原本由于模拟执行而存在的漏洞与保守检查不复存在, 从而能够消除诸多上述的“假阴性”和“假阳性”问题. 具体来说, 将 eBPF 程序运行在内核态沙箱中, 由沙箱对程序运行时的内存访问进行动态检查, 保证程序无法对内核内存进行非法访问; 为高效实现轻量化的内核态沙箱, 利用新型硬件特性 Intel PKS (protection keys for supervisor) 进行零开销的访存指令检查, 并提出高效的内核与沙箱中 eBPF 程序交互方法. 评测结果表明, 所提方法能够消除内核 eBPF 检查器中的内存安全漏洞 (自 2020 年以来该类型漏洞在 eBPF 检查器的总漏洞中占比超过 60%); 即使在吞吐量较高的网络包处理场景下, 轻量化内核沙箱带来的性能开销低于 3%.

关键词: eBPF; PKS; 内存隔离; 操作系统内核

中图法分类号: TP306

中文引用格式: 李浩, 古金宇, 夏虞斌, 臧斌宇, 陈海波. 基于PKS硬件特性的eBPF内存隔离机制. 软件学报, 2023, 34(12): 5921–5939. http://www.jos.org.cn/1000-9825/6762.htm

英文引用格式: Li H, Gu JY, Xia YB, Zang BY, Chen HB. Memory Isolation Mechanism of eBPF Based on PKS Hardware Feature. Ruan Jian Xue Bao/Journal of Software, 2023, 34(12): 5921–5939 (in Chinese). http://www.jos.org.cn/1000-9825/6762.htm

Memory Isolation Mechanism of eBPF Based on PKS Hardware Feature

LI Hao, GU Jin-Yu, XIA Yu-Bin, ZANG Bin-Yu, CHEN Hai-Bo

(School of Software, Shanghai Jiao Tong University, Shanghai 200240, China)

Abstract: The extended Berkeley packet filter (eBPF) mechanism in the Linux kernel can safely load user-provided untrusted programs into the kernel. In the eBPF mechanism, the verifier checks these programs and ensures that they will not cause the kernel to crash or access the kernel address space maliciously. In recent years, the eBPF mechanism has developed rapidly, and its verifier has become more complex as more and more new features are added. This study observes two problems of the complex eBPF verifier. One is the “false negative” problem: There are many bugs in the complex security check logic of the verifier, and attackers can leverage these bugs to design malicious eBPF programs that can pass the verifier to attack the kernel. The other is the “false positive” problem: Since the verifier

* 基金项目: 国家杰出青年科学基金 (61925206); 华为创新计划
收稿时间: 2022-04-17; 修改时间: 2022-07-18; 采用时间: 2022-08-09; jos 在线出版时间: 2023-02-15
CNKI 网络首发时间: 2023-02-16

adopts the static check method, only conservative checks can be performed due to the lack of runtime information. This may cause the originally safe program to fail the check of the verifier and only support limited semantics, which brings difficulties to the development of eBPF programs. Further analysis shows that the static simulation execution check mechanism in the eBPF verifier features massive codes, high complexity, and conservative analysis, which are the main reasons for security vulnerabilities and false positives. Therefore, this study proposes to replace the static simulation execution check mechanism in the eBPF verifier with a lightweight dynamic check method. The bugs and conservative checks that originally existed in the eBPF verifier due to simulation execution no longer exist, and hence, the above-mentioned "false negative" and "false positive" problems can be eliminated. Specifically, the eBPF program is run in a kernel sandbox, which dynamically checks the memory access of the program in the runtime to prevent it from accessing the kernel memory illegally. For efficient implementation of a lightweight kernel sandbox, the Intel protection keys for supervisor (PKS), a new hardware feature, is used to perform a zero-overhead memory access check, and an efficient interaction method between the kernel and the eBPF program in the sandbox is presented. The evaluation results show that this study can eliminate memory security vulnerabilities of the eBPF verifier (this type of vulnerability has accounted for more than 60% of the total vulnerabilities of the eBPF verifier since 2020). Moreover, in the scenario of high-throughput network packet processing, the performance overhead brought by the lightweight kernel sandbox is lower than 3%.

Key words: extended Berkeley packet filter (eBPF); protection keys for supervisor (PKS); memory isolation; operating system kernel

Extended Berkeley packet filter (eBPF) 机制可以将用户提供的程序直接运行在内核中, 促进了内核可编程和动态可扩展技术的发展. 随着 eBPF 机制被加入到 Linux 内核, 它发展迅速并成为了内核中的重要组成部分. 与修改并重新编译内核或加载内核模块不同, eBPF 可以将用户提供的程序编译成字节码并插入到 Linux 内核中预先配置好的地方 (例如网络包处理、系统调用) 执行, 或者基于 Kprobe 技术将其动态加载到内核中几乎任意函数的执行处[1]. 因此 eBPF 程序可以便利地实现进程跟踪、网络包预处理等功能. 同时 eBPF 使用严格的检查器 (verifier)[1,2] 来保证加载的程序不会对内核进行攻击, 例如导致系统崩溃或者泄露内核数据.

Linux 中的 eBPF 检查器会模拟执行待检测 eBPF 程序的每一条指令, 并追踪每一个变量的取值范围, 以防止 eBPF 程序越界访问内核内存. 由于 eBPF 程序可能包含条件分支, 该检查器需要遍历所有的分支状态, 这使得分析一个非常复杂的程序变得不现实. 为了加速检查, 该检查器还缓存了运行过若干条指令后的程序状态, 并实现了一套等价性检查的算法来进行动态剪枝. 尽管如此, eBPF 机制对待检测程序的复杂度 (例如指令数量) 仍然存在严格的要求.

eBPF 机制中静态安全检查机制的设计存在两个关键问题. 其一是检查器的代码自身存在漏洞, 可能被攻击者利用而造成"假阴性"问题. 在 5.10 版本的 Linux 内核中, eBPF 检查器的代码约为 12 000 行, 逻辑复杂, 规模庞大. 自 2020 年以来, Linux 内核披露的漏洞中涉及 eBPF 检查器的高达 14 个[3], 攻击者可以根据漏洞精心制作能够导致内核崩溃或者窃取内核敏感数据的 eBPF 程序[4], 而这些程序可以通过检查器的检查. 其二是检查器的保守检查带来的"假阳性"问题[2,5]. 一方面为了避免检查器占用过多的 CPU 执行时间, eBPF 在设计之初就仅支持有限的语义, 例如对指令条数有限制, 不支持动态内存分配等; 另一方面检查器采用的模拟执行的方法缺少变量间关系的语义, 因此难以准确追踪变量的取值范围. 这些因素为 eBPF 程序的开发带来了困难, 开发者们往往需要仔细分析, 甚至了解检查器底层原理才能写出高效且被检查器接收的代码.

观察到检查器采用静态模拟执行检查的方式是造成"假阴性"和"假阳性"的主要原因, 本工作提出在内核中为 eBPF 程序构建沙箱作为运行环境, 使用动态检查取代静态模拟执行检查. 运行在内核沙箱中的 eBPF 程序没有权限访问沙箱外的内存空间. 该设计一方面显著简化了检查器的静态检查过程, 进而减小了检查器中出现漏洞的可能性, 而且其运行时保护机制可以防止逃逸检查的恶意 eBPF 程序危害内核, 这缓解了上述的"假阴性"问题. 另一方面, 简化后的检查器允许更多原本安全的代码通过检查, 例如可以将变量间的关系的追踪交由运行时检查; 而且还能提供更多的语义, 例如支持动态内存分配等, 这些则缓解了上述的"假阳性"问题. 但是这种设计面临两个新的挑战: 一是如何高效地构造轻量化沙箱, 在保证其中运行的 eBPF 程序无法访问内核内存的同时仅引入极低的性能开销; 二是如何为沙箱中的 eBPF 程序提供高效而安全的与内核交互的机制.

为了应对上述挑战, 本工作提出利用 Intel 处理器的 protection keys for supervisor (PKS) 硬件特性[6]在内核中

为 eBPF 程序构建沙箱的机制. PKS 可以将内核地址空间以页的粒度划分为独立的内存域, 并可以在不引入额外性能开销的情况下检查程序运行时的每一次内存访问操作是否具有对相应内存域的访问权限. 使用 PKS 特性构建沙箱存在以下 3 方面的挑战: 一是需要准确识别 eBPF 程序能够合法访问的内存, 本文详细分析了 eBPF 程序的内存模型, 并仅将 eBPF 程序需要用到的内存其置于沙箱内; 二是需要保障 PKS 构建的沙箱的安全性, 本文详细分析了如何防止 eBPF 程序通过执行流攻击或者使用特权指令逃逸沙箱; 三是需要保证沙箱的轻量性, 本文测试并分析了进出沙箱对系统性能的影响, 结果表明本机制带来的性能开销极低.

本文工作主要有以下贡献.

- 提出软硬协同的内核态沙箱设计以增强 eBPF 机制的安全性和灵活性.
- 设计与实现面向 eBPF 程序的高效的内核沙箱原型系统.
- 对原型系统进行了功能和性能测试, 评测结果表明本工作能有效缓解 eBPF 检查器的“假阴性”和“假阳性”问题, 并且性能开销极低.

1 背景知识

1.1 eBPF

eBPF 机制的前身是 1992 年提出的一个高效的网络包过滤器 BPF[7], 它可以在网络包被网络栈处理之前检查包的内容, 并决定是否接收该网络包. BPF 机制将程序运行在一个语言虚拟机内, 这保证了程序运行的安全性; 而且它在做决策时避免了网络包的拷贝, 这使得其性能领先于当时的其他技术方案[1]. 此后 BPF 一直被作为网络包过滤的主流技术. 在 2014 年, 为了适应现代硬件, eBPF 作为 BPF 的下一代技术被引入. eBPF 支持了许多新特性, 例如允许使用 64 位寄存器; 将支持的通用寄存器数目从 2 个提升到 10 个; 扩充了指令集, 可以更好地将其映射为目标硬件指令集, 进而提升了程序运行的性能. 它开始被暴露给用户态, 其使用场景也不再局限于网络包过滤, 而是在网络管理、系统性能监测、系统安全等多个领域都取得应用[8−13].

图 1 展示了 eBPF 程序的执行过程. eBPF 机制提供了一套简化的 reduced instruction set computing (RISC) 指令集架构, 用户可以使用 LLVM 提供的 eBPF 后端[14]将 C 语言代码编译为 eBPF 字节码. 在将字节码加载到内核中执行之前, eBPF 依赖检查器来确保加载的程序的安全性. 由于 LLVM 编译器不在可信计算基 (trusted computing base, TCB) 中, 检查器会在 eBPF 字节码层面做检查, 确保 eBPF 程序不会访问内核敏感信息或导致内核崩溃. 成功通过了检查器检查的 eBPF 字节码会被 just in time (JIT) 编译器编译为目标硬件架构的机器码. eBPF 程序在被成功加载到内核之后, 会被特定的事件触发执行. 例如在网络驱动处理网络包、系统调用的开始和结束处都有内核预先设定好的 eBPF 触发点, 当系统执行到这些地方时, 会执行挂载在该处的 eBPF 程序. 此外, eBPF 还可以依靠 Kprobe 技术将程序插入到内核中几乎任何函数所在的位置[1].

为了便于 eBPF 程序的开发, 内核为 eBPF 程序提供了 map 数据结构[15], eBPF 程序可以通过键值对的方式将数据存储到预先定义好的 map 中. 这些 map 可以在不同的 eBPF 程序之间共享, 而且可以被用户态程序访问, 进而可以实现内核态与用户态程序间的数据交换. 此外, 内核还为 eBPF 程序提供了一系列辅助函数, 其中包括对 map 数据结构的访问函数、获取当前运行状态的函数 (例如获取当前 CPU 编号)、对 Linux 网络栈的套接字缓存 (socket buffer) 的处理函数等. 这些辅助函数一般都是向后兼容的, 因此随着内核系统的迭代, eBPF 程序大都可以不做修改地运行.

eBPF 程序只能访问有限的内存区域, 它们包括: (1) 512 B 的栈区域; (2) 程序上下文区域: 根据程序类型不同, eBPF 程序可以访问不同的确定大小的上下文区域, 它们主要被用于从内核向 eBPF 程序传递参数; (3) 网络包区域: 需要处理网络包的 eBPF 程序可以直接访问网络包中的数据; (4) map 区域: 内核提供 map 相关的 API 供程序使用, map 可以和其他 eBPF 程序或者用户态程序共享. 其中网络包区域的大小是静态检查时未知的, 它的起始位置和大小被作为参数储存在程序上下文区域中. eBPF 检查器会通过静态检查追踪每个变量的类型和取值范围, 并判断每次访存是否合法, 并拒绝加载访问了超出上述内存区域的 eBPF 程序.

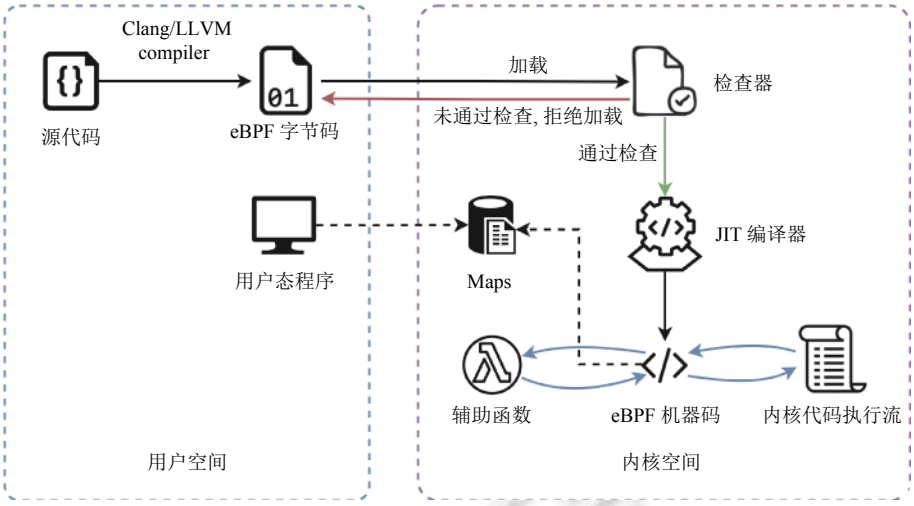

图 1 eBPF 程序执行过程

1.2 PKS

Intel 在 Skylake-SP 处理器上推出了 protection keys for userspace (PKU), 主要被用来实现用户态应用的进程间隔离[6]. PKU 利用 page table entry (PTE) 中的部分保留位标记一个页所属的内存域 (目前的实现使用 4 位作为内存域 ID, 因而能够支持 16 个内存域). PKU 引入了一个新的 32 位寄存器 pkru 来控制当前线程对每个内存域的访问权限. pkru 的每两位对应一个内存域, 分别表示禁止读和禁止写. 每当处理器需要访问用户态的页时, 会根据当前线程 pkru 的值判断是否有访问该页所在域的权限, 如果没有就会触发 PKU 缺页异常. 由于 PKU 可以实现轻量化的内存隔离, 很多研究者将其用于用户态应用程序的进程间隔离[16−19].

但是 PKU 机制对内核页表无效, 为了实现内核地址空间的隔离, Intel 计划在下一代处理器中引入 PKS[6,20]. 与 PKU 类似, PKS 也增加了一个新的 32 位寄存器 pkrs 来控制当前内核线程对内存域的访问权限. 同时, 处理器可以使用 RDMSR/WRMSR 指令对 pkrs 寄存器进行读写操作. PKS 适合被用于对持久化内存的保护, 持久化内存所在的页被隔离在一个域中, 所有对持久化内存的访问需要先执行 WRMSR 指令获取相应的访问权限才能继续进行, 这可以防止内核中的无关线程意外地修改持久化内存中的数据.

与传统的基于 software fault isolation (SFI)[21−23]或者虚拟化[24−26]的内核隔离机制相比, PKS 有许多优势. SFI 一般通过范围检查等软件的方法[27]来避免对敏感数据的读写操作. 但是 SFI 一般需要将敏感数据聚集在一起以减少检查的次数, 这往往需要侵入式地修改待隔离模块的内存布局. 目前在 x86-64 上最高效的 SFI 实现基于 Intel 的 memory protection extensions (MPX)[28,29]特性, 但是其运行时开销不可忽略, 而且需要结合 control flow integrity (CFI) 来保证检查不被绕过. 而基于虚拟化的隔离则会引入特权层, 难以适用于非虚拟化的场景, 而且频繁的特权层切换会严重影响系统性能. PKS 以页的粒度将内存空间分为至多 16 个区域, 这些区域不必连续, 适用场景广泛, 而且运行时的基于硬件的检查几乎不会带来性能损失.

硬件模拟: 由于截止到目前 Intel 尚未发布支持 PKS 特性的商用 CPU, 我们参考之前的工作[30,31], 使用 PKU 来模拟 PKS. 无论运行在用户态还是内核态, PKU 都会对用户可以访问的内存页做权限检查. 因此可以将内核页对应的四级页表中的 User/Kernel 位均设置为 1 (即 User 位), 以允许 PKU 在内核态保护内核页.

2 研究动机: eBPF 检查器存在的问题分析

eBPF 检查器利用一套静态检查机制来判断用户提供的程序是否安全, 这套机制的设计与实现较为复杂. 本节详细分析了 eBPF 检查器的检查过程, 总结出 eBPF 检查器存在的“假阴性”和“假阳性”问题.

2.1 检查器的“假阴性”与“假阳性”问题

用户通过 BPF 系统调用将 eBPF 字节码传入内核, 内核通过严格的检查器检查用户提供的程序是否符合安全要求. 安全要求主要包括两部分: 一是确保程序会终止, 不会出现死循环等会导致处理器无限制等待的情况; 二是 eBPF 程序仅能访问有限的内存, 不会泄露内核中的数据或者恶意访问无效内存地址导致内核崩溃. 具体来说, 检查器会做以下检查.

(1) 环路检查: 根据传入的字节码构建控制流图 (control flow graph, CFG), 通过深度优先算法判断程序是否存在回路 (循环结构). 尽管在 5.3 版本的内核中已经支持了有限次数的循环[32], 但是通用循环结构仍然会阻碍判断程序的可终止性, 因此目前仍不被支持.

(2) 复杂度检查: 统计 eBPF 程序的指令条数、分支条数, 如果超过限制, 就拒绝该程序. 由于 eBPF 检查器后续会通过模拟执行的方法分析每一条指令, 如果指令条数过多或分支条数过多, 会导致分析的复杂度太高, 例如会出现“路径爆炸”的问题.

(3) 访存检查: 采用模拟执行的方式分析每一条指令, 记录每一个变量的类型以及取值范围. 变量的类型可以分为标量和指针, 其中指针又被进一步地分为 26 个小类, 例如指向上下文的指针、指向 map 的指针、指向栈的指针等. 每当分析到内存访问指令时, 会根据访存的变量的类型和范围判断该访存是否合法. 例如一次栈上的访存, 其偏移量超过了 512 B (栈的大小), 就会被判定为非法访存. 此外, 每当分析到辅助函数的时候, 会根据内核提供的辅助函数的参数类型判断调用是否合法. 内核为每一个辅助函数定义了一个说明其参数类型的结构体, 例如查询 map 的函数的第一个参数应该为指向 map 的指针, 如果实际传入的指针的类型与之不符, 检查器就会拒绝该程序.

(4) 其他检查: 例如检查除法操作的被除数是否可能为 0, 程序是否可能会死锁等.

然而 eBPF 检查器面临着“假阴性”与“假阳性”的问题. eBPF 检查器的“假阴性”问题是由于检查器自身存在漏洞而造成的, 攻击者可以利用这些漏洞构造出能够通过检查器检查但是会导致内核崩溃或者修改内核敏感信息的 eBPF 程序. 例如在 2022 年披露的 Linux eBPF 漏洞 CVE-2022-23222[4]中, 检查器错误地允许一类指针执行算术运算, 因此未能正确计算变量的取值范围, 攻击者可以利用该漏洞使得 eBPF 程序访问内核中的任意内存.

eBPF 检查器的“假阳性”问题是由于检查器为降低复杂性采用保守检查原则而导致的, 可能会拒绝不会危害内核的安全程序. 首先, 检查器依赖模拟执行的方式分析 eBPF 字节码中的每一条指令, 当程序规模增大时, 特别是条件分支增多时, 会导致“路径爆炸”的问题, 因此检查器只支持有限规模的 eBPF 程序. 其次, 检查器限制了 eBPF 程序的表达能力. 例如 eBPF 程序无法动态分配内存; 在程序中使用函数一般要强制内联, 否则检查器不能很好地支持跨函数的变量范围追踪. 再次, 检查器过于严格给 eBPF 程序开发带来了不便, 开发者必须得为代码增加许多额外的检查才能使得程序通过检查器. 最后, 检查器为编译期的代码优化带来了障碍, LLVM 编译器不得不采用保守的策略以避免代码优化违反检查器的安全要求[33].

2.2 观察: 检查器的访存检查是造成“假阴性”和“假阳性”问题的主要原因

通过分析 Linux 内核中的检查器代码, 本工作观察到主要是第 2.1 节中所述的第 3 步访存检查导致了检查器的“假阴性”和“假阳性”问题. 首先, 本工作分析了自 2020 年以来内核中披露的关于检查器的 14 个 CVE, 发现其中 9 个与检查器的访存检查相关, 例如 CVE-2021-45402[34]和 CVE-2021-3490[35]都是在处理算术运算时没有正确更新某些变量的范围, 导致检查器可能会接收一些内存访问越界的 eBPF 程序. 剩余的 5 个 CVE 中有 4 个 CVE 与暂态执行攻击相关, 1 个与检查器本身实现出错可能导致内核崩溃有关, 与检查器的其他 3 项检查无关. 检查器的访存检查存在较多漏洞主要由于其设计与实现的复杂性最高, 这也体现在其代码规模上. 例如在 Linux 5.10 版本的内核中, 检查器的代码共有 12 031 行, 其中第 3 步检查相关的代码有 8 082 行, 占比为 67.2%. 这部分代码规模庞大, 而且缺乏形式化的正确性证明, 因此成为了漏洞重灾区, 攻击者可以利用漏洞绕过第 3 步的访存检查, 访问原本不允许 eBPF 程序访问的内存, 进而篡改内核敏感数据或者导致内核崩溃.

其次, 同样是第 3 步的访存检查限制了 eBPF 程序的表达能力, 使得检查器更容易拒绝用户提供的原本正确

的代码. 在第 3 步中, 检查器通过深度优先算法遍历每一条执行路径, 同时缓存部分语句执行后的系统状态, 当第 2 次执行到相同语句时会将当前状态与缓存的状态比较, 如果状态相同, 就证明此后的执行流已经被检查过了, 因此可以进行剪枝操作, 继续遍历下一条执行路径. 尽管经过剪枝操作, 整个检查过程依旧非常耗时, 阻碍了检查器分析更加复杂的程序. 因此 eBPF 程序开发者不得不尽可能精简程序以符合该要求. 而且这一部分的检查过于严格, 检查器采用一种保守的检查策略, 亦即不允许所有“可能出错”的程序运行, 而实际上由于检查器缺乏程序的运行时语义, 往往不能做出正确的判断. 此时需要开发者提供更多的手动检查语句, 这一方面增加了开发者开发 eBPF 程序的难度, 另一方面也给检查器的检查工作带来额外的负担.

3 基于 PKS 构建 eBPF 程序的沙箱运行环境

基于上述观察, 本工作摒弃 eBPF 检查器中的静态访存检查, 采用动态访存检查方式, 即在 eBPF 程序运行时对访存指令的目标地址进行检查. 许多现有的动态访存检查方法依赖 SFI[22,36,37], SFI 可以在内存访问处插入检查代码以确保访存不会越界. 由于 eBPF 程序允许访问的内存较为分散, 内存插桩检查会引入过高的负载. 另外一些工作[19,38,39]将隔离的内存空间置于独立的页表中, 并利用 memory management unit (MMU) 硬件来检查内存访问以避免软件插桩带来的高开销. 但是频繁的页表切换也会带来较大的性能开销, 例如使用多个页表隔离 NGINX 服务器可能会带来 65% 的系统负载[19]. 新的硬件特性 PKS 支持高效的内核态内存隔离, 本工作利用该硬件为 eBPF 程序构建内核态轻量级沙箱. 如图 2 所示, 内核沙箱中的 eBPF 程序以隔离态线程的形式运行, 其权限由内核在沙箱出入口处设置. eBPF 程序具有访问沙箱中的隔离内存的权限, 而当访问内核外的内存时会触发 PKS 缺页异常, 内核将终止 eBPF 程序的运行, 并将系统恢复到 eBPF 运行前的状态. eBPF 程序仅能调用内核预设的辅助函数, eBPF 机制的检查器和 JIT 编译器保证沙箱中的 eBPF 程序不会调用内核中的其他函数.

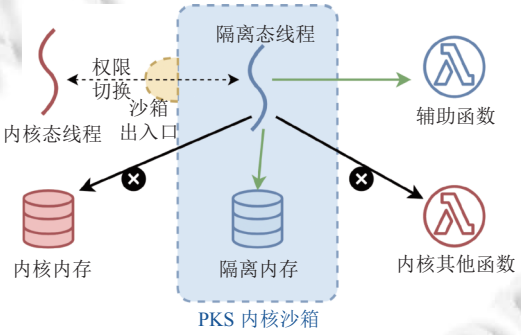

图 2 PKS 内核沙箱示意图

本文提出的动态检查机制有以下优势: 一是可以显著减少检查器的工作, 减小了 TCB 的代码量; 二是可以有效防御恶意 eBPF 程序攻击内核, 恶意 eBPF 程序运行时无法非法访存; 三是允许开发者编写更加灵活的代码, 而不需要手动增加不必要的检查语句; 四是允许编译器更加激进地优化代码, 而不需要过度考虑检查器规定的安全语义. 但是这带来了新的挑战: PKS 机制仅提供了内存隔离的功能, 如何将其用于为沙箱中的 eBPF 程序提供高效而安全的与内核交互的机制.

系统设计架构图如后文图 3 所示, eBPF 程序运行所需的栈区域、上下文区域、map 区域以及数据包区域都被置于内核沙箱中. 本节将从一个在沙箱中运行的 eBPF 程序的角度, 介绍程序加载时如何创建并初始化沙箱 (第 3.1 节), 如何向沙箱中的 eBPF 程序传递参数 (第 3.2 节), 如何在沙箱中支持 eBPF 程序的运行时环境 (第 3.3 节), 最后分析该设计是否满足功能性和安全性需求 (第 3.4 节).

3.1 基于内核沙箱的 eBPF 程序加载

当用户通过 BPF 系统调用加载一个 eBPF 程序时, 内核会为其分配一个新的沙箱. 如图 4 所示, 每个沙箱有一

个唯一的沙箱 ID (例如沙箱 1 的 ID 为 2), 沙箱中的内存页对应的 PTE 中的内存域的值为沙箱 ID. 所有内存页默认处于 0 号内存域中, 内核可以通过修改内存页对应的内存域 ID 的值将其置于特定沙箱中. eBPF 程序运行时能访问的内存空间包括栈区域、上下文区域、map 区域和数据包区域. 其中 map 区域和数据包区域所在的内存页在 eBPF 程序运行前是已知的, 因此可以在初始化的时候将它们置于沙箱中. 而上下文区域和栈区域则是运行时动态确定的, 本文将分别在第 3.2 节和第 3.3 节中介绍相关内容.

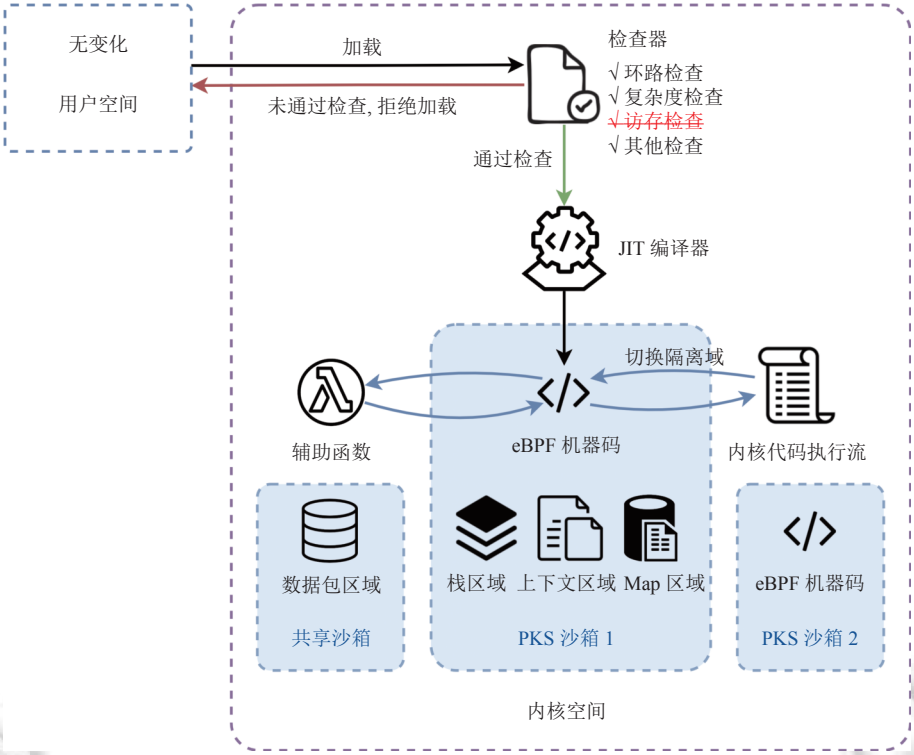

图 3 基于 PKS 的 eBPF 增强系统架构图

(1) map 区域. eBPF 程序若要使用 map, 需要预先定义 map 的类型、键和值的大小、储存的键值对的最大个数等. 在加载 map 的时候, 首先需要使用 BPF 系统调用在内核空间内创建该 map 数据结构, 并返回一个匿名的文件描述符, 此后该 eBPF 程序使用该文件描述符访问对应的 map. 为了保证沙箱中的 eBPF 程序能够正常访问 map, 本设计保证在创建 map 数据结构的时候将分配的内存页置于内核沙箱内. 由于 PKS 机制以页的粒度保护内存区域, 因此本设计修改了每一种 map 类型的内存分配函数, 使得其分配的内存以页的粒度对齐.

此外, map 可以在不同 eBPF 程序间共享, 为了支持这一特性, 本设计引入了共享沙箱. 共享沙箱不用于执行 eBPF 程序, 而是仅用来标记共享内存. 如图 4 所示, ID 为 3 的沙箱是一个共享沙箱, 其中储存的是沙箱 1 和沙箱 2 共享的 map 区域的内存页. 沙箱 1 和沙箱 2 中的 eBPF 程序在执行时均可以访问沙箱 3 中的内存页.

(2) 数据包区域. 为了避免网络数据包拷贝带来的开销, eBPF 程序可以直接访问网络包的数据. Linux 内核提供了对网络包处理的统一接口: 套接字缓存 (skb). 网络包即为 skb 结构体中 data 与 data_end 之间的内存空间. 网络包数据被所有具有网络包访问权限的 eBPF 程序共享, 为了支持这一特性, 本设计提出将所有的网络包置于一个共享沙箱中. 如图 4 所示, ID 为 1 的沙箱是一个数据包共享沙箱, 拥有数据包访问权限的 eBPF 程序在沙箱中执行时会获得该共享沙箱的读写权限. 本设计在内核中维护了一个用于储存数据包的缓冲区, 并在系统启动时将其置于 ID 为 1 的共享沙箱内. 同时本设计提供了一个轻量级的内存管理库用于管理缓冲区内存空间, 并修改内核中数据包的分配函数以保证所有的数据包分配在该缓冲区中.

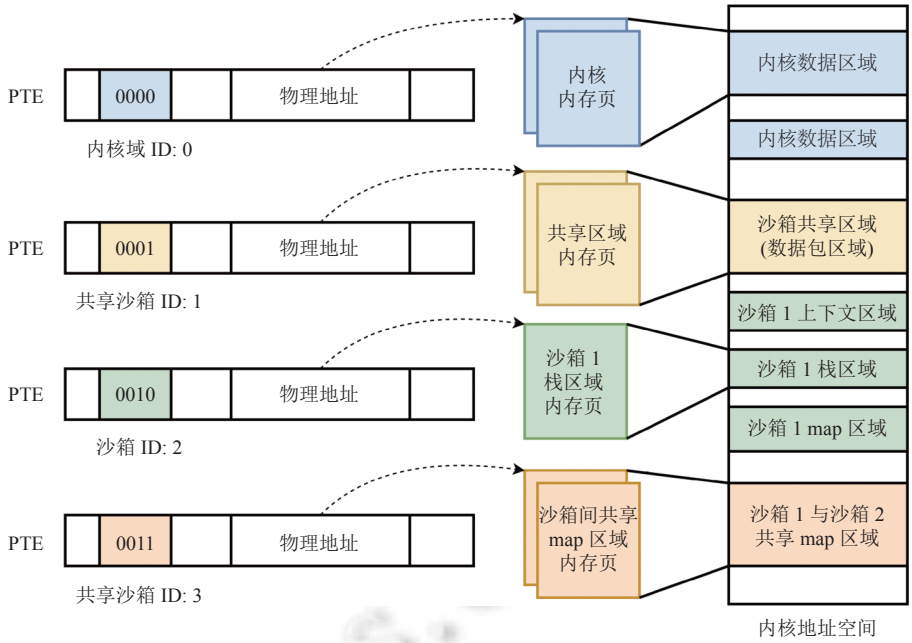

图 4 内核地址空间布局

3.2 内核沙箱中 eBPF 程序的上下文处理

Linux 内核中预先定义了一系列 eBPF 函数的入口函数, 内核线程仅能通过这些入口函数进入沙箱中执行 eBPF 机器码. 图 5 中展示了一个典型的入口函数: dispatcher. 它接收 3 个参数, 其中 ctx 是指向上下文对象的指针, 用于向 eBPF 程序传递参数; insnsi 是 eBPF 程序的字节码数组, 仅用于解释执行, 而主流的运行方式是通过 JIT 编译为机器码执行, 因此该参数可以忽略; bpf_func 指向 JIT 编译出的 eBPF 机器码的地址. 该入口函数的主要功能是以 ctx 为参数执行 eBPF 机器码. 如图 5 所示, 本设计在执行 eBPF 机器码的前后通过设置 pkrs 的寄存器来进出内核沙箱.

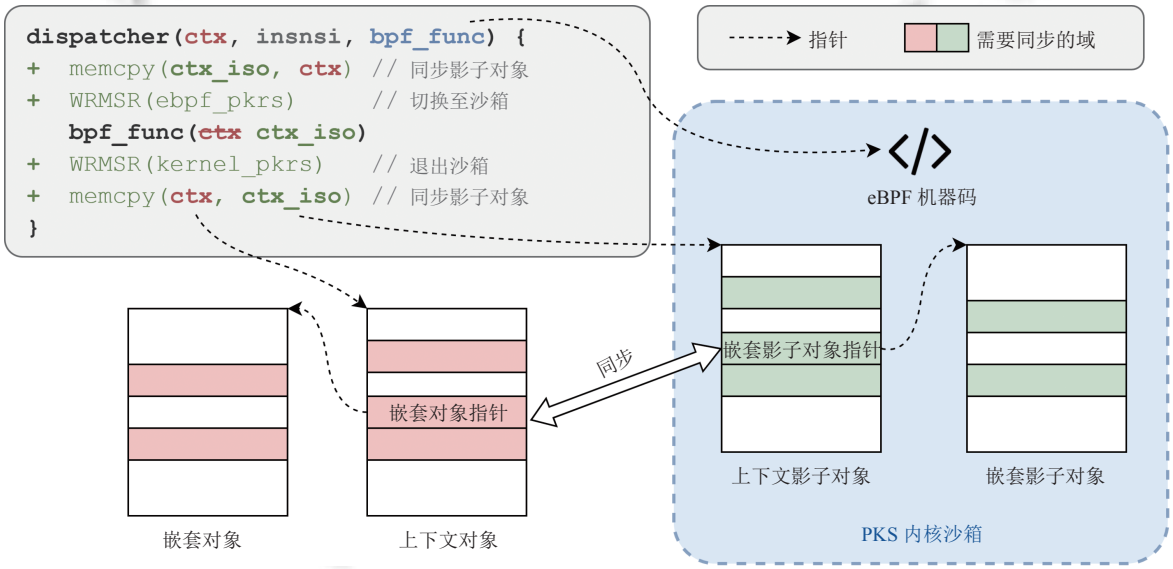

图 5 上下文区域处理示意图

内核沙箱中的 eBPF 程序没有上下文对象的访问权限, 本文提出影子对象机制解决该问题. 影子对象即为内核数据结构在沙箱内的一份拷贝, 它和原数据结构在内容上保持一致. 本文在入口函数处为上下文对象构建一个影子对象拷贝, 并使用指向影子对象的指针 (ctx_iso) 替换原指针 (ctx) 作为参数传给 eBPF 机器码, 因此得以对 eBPF 程序透明地支持上下文区域的隔离. 当 eBPF 函数执行完毕后, 影子对象的值可能被 eBPF 程序修改, 此时再将影子对象的值拷贝回原对象. 不同类型的 eBPF 程序的上下文结构体不同, 表 1 中列出了一些常见类型的 eBPF 程序以及它们对应的上下文结构体, 本设计基于不同类型的 eBPF 程序在相应的入口函数处定义相应类型的上下文影子对象.

表 1 常见类型的 eBPF 程序的上下文结构体

程序类型	结构体类型	说明
Socket Filter	__sk_buff	__sk_buff实际映射sk_buff, 储存网络包元数据
Kprobe	pt_regs	储存寄存器状态
Tracepoint	种类众多*	储存注入代码处上下文信息
XDP	xdp_md	xdp_md实际映射xdp_buff, 储存网络包元数据
Perf Event	bpf_perf_event_data	储存寄存器状态和其他perf元数据
Raw Tracepoint	bpf_raw_tracepoint_args	储存寄存器状态和系统调用序号

注: * 根据注入代码的位置不同, 上下文结构体也不相同, 本文选择所有结构体中最大的作为影子对象的大小

影子对象的同步机制存在以下两个问题. 一是资源浪费的问题: eBPF 程序可能仅会访问影子对象中的少部分域, 而将原对象全部拷贝到影子对象中会浪费大量的处理器资源. 例如 cilium 项目[8]内置的 eBPF 程序文件 bpf_host.c 仅用到了上下文结构体的 42 个域中的 5 个. 二是数据结构映射的问题: 部分上下文结构体是 eBPF 程序独有的, 没有与其直接对应的内核数据结构. 如表 1 中列出的 Socket Filter 类型的 eBPF 程序的上下文结构体类型是__sk_buff, 但是内核中处理网络包的数据结构是套接字缓存 (sk_buff). __sk_buff 是 sk_buff 的镜像对象 (mirror object)[40], 仅包含数据包长度、数据包类型、传输协议等需要暴露给 eBPF 程序的 sk_buff 中的数据. 用户提交的 eBPF 字节码仅能操作__sk_buff 结构体, 在字节码通过检查器检查后, 再由 JIT 编译器将其中对__sk_buff 的访问转化为对 sk_buff 的访问. 使用镜像对象有效地限制了 eBPF 程序访问的内存范围, 即仅允许访问 sk_buff 中的部分数据. 然而动态隔离基于机器码, 缺失了__sk_buff 的语义, 如果将 sk_buff 作为上下文对象进行整体拷贝, eBPF 程序可能会访问到 sk_buff 中不允许被访问的内容.

本工作提出用时拷贝的机制解决上述问题. 如图 5 所示, 上下文对象中仅有部分域 (红色阴影标记的域) 需要被同步到影子对象中. 本设计仅将 eBPF 程序需要用到的域同步到影子对象, 因此解决了资源浪费的问题. 对于__sk_buff 等镜像对象, 本设计仅同步它们与原对象存在映射的域, 因此实现了细粒度的对象隔离. 例如 eBPF 程序无法访问 sk_buff 中的套接字对象 (sock), 因为在同步时该域没有被拷贝到影子对象中; 同时 eBPF 程序尽管能够修改影子对象中 sock 的值, 但是在 eBPF 程序执行完成后该域不会被同步回原对象, 因此也不会对系统状态造成影响.

为了准确识别需要同步的域, 本设计向开发者提供一个可选的接口, 允许其提供待拷贝的域的列表, 因此本设计可以在不影响程序安全性的前提下使得 eBPF 程序获得最优的性能. 更进一步地, 开发者可以指定每一个待拷贝域的读写属性, 如果某个域被指定为只读, 在 eBPF 程序执行完毕后无需将其同步回原对象.

在构建影子对象的时候还需要考虑嵌套数据结构的问题. 例如__sk_buff 结构体包含了一个指向 bpf_flow_keys 结构体的指针, 仅拷贝该指针会导致 eBPF 程序没有对嵌套结构体的访问权限. 如图 5 所示, 本设计在创建一个包含指针的结构体的影子对象的时候, 还会为所有的嵌套结构体创建影子对象 (嵌套影子对象). 与此同时, 上下文影子对象中的指针会被更新为指向嵌套影子对象. 本机制同样可以处理两层及以上嵌套的情形. 但是我们观察到, 在常见类型的 eBPF 程序中几乎不存在多次嵌套的情况, 而且仅有很少一部分程序会真正访问嵌套对象, 例如 bpf_flow_keys 仅在 Flow Dissection 类型的 eBPF 程序中被使用. 因此在大多数场景下, 系统并不会引入嵌套数据

结构带来的额外开销.

为了避免动态分配影子对象带来的性能损失, 本设计在加载 eBPF 程序时预先为影子对象分配一段区域, 并将其置于沙箱内. 这块区域被初始化为一个影子对象的数组, 其大小为 CPU 核心的数目. 当某个 CPU 核心执行 eBPF 程序时, 首先获取其 CPU 序号, 以其为索引访问保留的影子对象数组中对应的影子对象. 当前线程会原子性地获取该对象的所有权, 并在执行完毕后将所有权释放, 因此避免了耗时的动态内存分配.

3.3 内核沙箱中 eBPF 程序的运行时环境

沙箱中运行的 eBPF 程序需要访问栈区域, 也可能调用一些辅助函数, 本节将分别介绍.

(1) 栈区域. eBPF 程序拥有 512 B 的栈区域用于储存局部变量和函数调用的栈帧. 因为 eBPF 检查器的访存检查会检查栈访问是否超过了 512 B 的界限, 因此执行 eBPF 程序时可以直接沿用内核栈而无需换栈. 但是本工作使用 PKS 动态检查替换了静态访存检查, 这带来了新的问题: 如果内核栈不位于沙箱中, eBPF 程序就失去了对栈区域的访问权限; 而如果将内核栈置于沙箱中, eBPF 程序就拥有了访问栈上其他函数栈帧的权限, 进而可能危害系统安全, 例如恶意的 eBPF 程序可以通过修改其他栈帧的返回地址跳转到内核的任意位置执行.

本文提出影子栈机制解决该问题. 影子栈是位于内核沙箱中的一块内存区域, eBPF 程序执行的时候会将栈指针切换到影子栈, 待执行完毕后再切换回内核栈. 与管理影子对象类似, 为了避免运行时分配影子栈带来性能开销, 本设计在创建 eBPF 沙箱的时候预先在沙箱中分配影子栈数组, 并在程序执行时根据当前的 CPU 序号直接读取影子栈的地址.

(2) 辅助函数. 内核中定义了一系列供 eBPF 程序调用的辅助函数, 例如可以输出调试信息、获取系统运行时间、与 map 交互等. 但是内核沙箱中的 eBPF 程序不能直接调用辅助函数, 一方面辅助函数可能会访问内核数据结构, 而沙箱中运行的线程没有访问权限; 另一方面, 内核不应该直接访问沙箱中的影子对象, 因为这些对象储存的信息可能不完整, 而且直接操作这些对象还可能导致影子对象与内核对象不同步.

本文提出跳板函数机制来解决辅助函数调用的问题. 如图 6 所示, eBPF 程序只能通过本机制提供的跳板函数 (*_callgate) 调用辅助函数. 如果辅助函数需要访问沙箱外的内核数据结构, 跳板函数在调用辅助函数前会更新 pkrs 的值临时退出沙箱, 并在调用后重新进入沙箱. 为了避免内核直接访问影子对象, 跳板函数会在调用辅助函数前将影子对象同步到内核对象, 并将内核对象作为参数传递给辅助函数 (例如图 6 中的②).

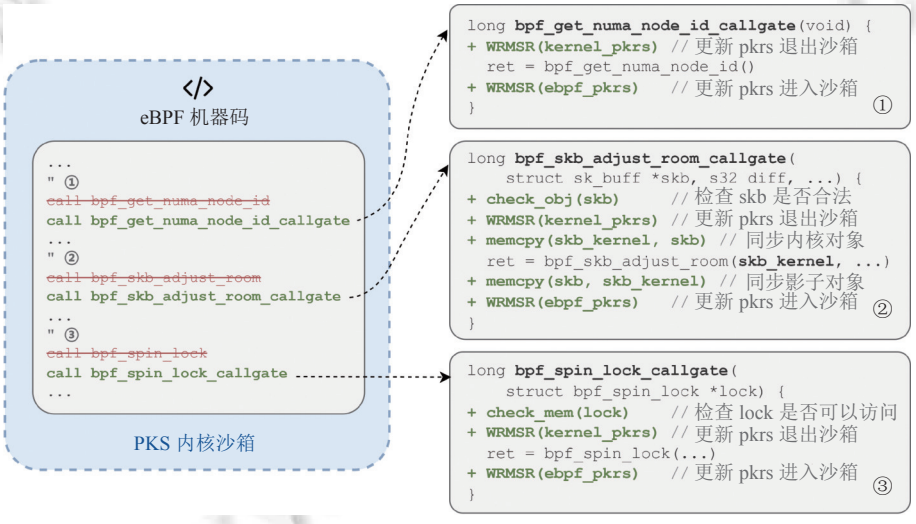

图 6 辅助函数处理示意图

但是这种设计会带来潜在的安全问题, 因为恶意的 eBPF 程序可能会传递错误的参数来攻击内核, 例如 Linux 5.1 版本内核开始支持的 bpf_spin_lock 辅助函数可以将传入的指针指向的内存修改为 1. 如果没有检查器在模拟

执行时的安全检查, 这个辅助函数可以被攻击者用来修改内核任意处的内存.

为了解决该问题, 本工作分析了 Linux 5.10 版本的内核支持的全部 155 个辅助函数, 它们属于以下情形的一类或几类.

(1) 无参数, 用于获取系统信息, 例如 bpf_get_numa_node_id.

(2) 与 map 交互, 其中一个参数类型为 bpf_map 的指针, 例如 bpf_map_lookup_elem.

(3) 其中一个参数为上下文结构体的指针, 例如 bpf_skb_adjust_room.

(4) 其中一个参数是 eBPF 程序可以访问的指针, 例如 bpf_trace_printk、bpf_spin_lock.

(5) bpf_tail_call, 用于调用另一个 eBPF 程序.

(6) bpf_kprobe_read 等本身允许访问任意内核内存的函数.

图 6 中的①展示了第 1 种情形, 调用此类辅助函数不会对系统造成危害, 因此只需更新 pkrs 的值以临时退出沙箱. 针对第 2 种和第 3 种情形, 跳板函数负责检查相应参数的指针是否合法. 例如图 6 中的②展示了第 3 种情形, skb 是指向上下文对象的指针, 跳板函数通过比较 skb 与 eBPF 入口函数记录的上下文影子对象的地址是否相同来判断 skb 是否被恶意篡改. 对于第 4 种情形, 本设计会尝试在内核沙箱中预读取该指针指向的内存, 因此图 6 中的③展示的利用 bpf_spin_lock 传入非法指针的攻击方式会在预读取时触发 PKS 缺页异常. 对于第 5 种情形, tail_call 的处理类似于函数调用, 且调用的函数必须预先指定并通过检查, 因此不会对系统造成危害. 第 6 种情形中涉及的函数仅允许管理员加载的高权限 eBPF 程序调用, 此类程序本身具有访问沙箱外内存的权限, 因此不在本机制的考虑范围之内. 此外, 一些不会访问内核数据结构的辅助函数 (例如仅访问沙箱内的 map 的辅助函数) 无需切换 pkrs 的值来临时退出沙箱.

得益于避免了繁琐的静态检查, 本机制可以更方便地添加新的辅助函数. 例如 eBPF 程序中不支持动态内存分配, 这在原 eBPF 机制中难以得到支持, 因为静态分析时难以获取需要分配的空间大小, 因此难以追踪对分配空间的内存访问是否越界. 而使用动态检查的方法可以解决这个问题: 在 eBPF 程序加载时保留一部分空间, 并将其置于内核沙箱中, 新添加的辅助函数可以直接在沙箱内管理这部分内存空间.

此外, 内核沙箱中的 eBPF 程序在运行时需要全局描述符表 (global descriptor table, GDT)、局部描述符表 (local descriptor table, LDT)、中断描述符表 (interrupt descriptor table, IDT) 等内核数据结构的访问权限, 因此在分配这些数据结构的时候将它们置于一个只读的隔离内存域中, 始终允许 eBPF 程序读取.

3.4 功能性与安全性讨论

本节讨论基于 PKS 动态隔离的 eBPF 系统所面临的功能性和安全性的问题.

(1) PKS 支持的域的数目有限. PKS 仅支持至多 16 个内存域 (0–15 号域), 其中 0 号域是所有内核代码和数据结构所在的默认的域, 1 号域被用作储存 GDT 等只读内核数据结构, 2 号域被用来储存网络包的数据, 因此 eBPF 程序至多可以使用剩余的内存域构建 13 个内核沙箱. 为了支持运行更多的 eBPF 程序, 本设计允许在一个内核沙箱中运行多个 eBPF 程序. 系统中的 eBPF 程序往往由少数用户加载, 例如 Kubernetes 管理员使用 cilium[4]管理集群网络, 可能会加载一系列 eBPF 程序. 本设计提出可以将同一个用户加载的不同 eBPF 程序运行在同一个内核沙箱内, 保证它们不能攻击内核, 但是不保证它们彼此之间的隔离性. 同时, 此设计允许这些程序之间方便地共享 map 等内存区域而不必引入新的共享沙箱. 此外, 一些研究者提出了软件或者硬件的方法突破 PKU 内存域的数目限制[41−43], 这些工作也可以被借鉴用于拓展 PKS 的内存域数目.

(2) PKS 缺页异常处理. 当恶意 eBPF 程序访问到内核沙箱外的内存时, 会触发 PKS 缺页异常. 为了避免内核崩溃, 缺页异常的处理函数中会终止 eBPF 程序的执行, 将 eBPF 程序从内核中卸载, 并将执行流重置到 eBPF 入口函数返回的地址继续执行. 由于 eBPF 程序本身不会修改内核的数据结构, 也不会对系统的全局状态做出修改, 因此只需要执行卸载 eBPF 程序的相关代码 (例如释放 eBPF 程序占用的内存等), 而无需进行复杂的状态保存与恢复.

(3) 执行流安全性分析. 检查器的 CFG 检查和受限的指令集保证了 eBPF 无法在运行时逃逸沙箱. 在加载 eBPF 程序时, 检查器的 CFG 检查保证了 eBPF 程序的跳转目标地址在 eBPF 程序内, 而且仅能跳转到程序中指令

的开始处, 因此攻击者无法劫持 eBPF 程序的控制流以执行任意代码.JIT 编译器仅能生成类型有限的字节码, 其中不包含 WRMSR 等特权指令, 因此被隔离的 eBPF 程序无法通过执行特权指令获取自身沙箱外的内存访问权限. 此外, 检查器的 CFG 检查保证了 eBPF 程序的可终止性.

4 系统评估与测试

我们在 Linux 5.10 版本的内核上实现了基于 PKS (PKU 模拟) 动态检查隔离 eBPF 程序的原型系统, 并开展了一系列测试. 为了说明本文提出的动态隔离机制的有效性, 我们将其与原始 eBPF 检查器的静态检查机制在安全性、易用性和性能 3 个维度进行对比, 对比结果表明本机制在几乎不引入开销的情况下提升了 eBPF 机制的安全性与易用性. 本节将分别从这 3 个维度回答以下问题.

- 本工作提出的机制能否解决 eBPF 检查器的“假阴性”导致的安全性问题? (第 4.1 节)
- 本工作提出的机制能否解决 eBPF 检查器的“假阳性”导致的易用性问题? (第 4.2 节)
- 本工作提出的机制对系统性能有何影响? (第 4.3 节)

4.1 安全性分析

由于 eBPF 检查器存在“假阴性”问题, 攻击者有机会精心构造不安全的程序, 它们能够通过检查, 但可以对内核发起攻击. 本工作利用 PKS 硬件特性对 eBPF 运行时内存访问进行检查, 因此能够避免这类程序攻击. 基于 PKS 的动态检查方法可以简化大部分 eBPF 检查器的代码. 在我们实现的原型系统中, eBPF 检查器的代码共 12 031 行, 在删减模拟执行相关的代码后, 检查器的代码只剩下 3 949 行, 简化了 67.2% 的代码. 我们分析了 2020 年以来内核中与 eBPF 检查器相关的 14 个 CVE, 其中有 9 个漏洞相关代码都在被删减的代码中. 我们使用动态检查的方式可以有效地防御这些 CVE. 本节将从一个典型的 CVE 入手, 详细地分析本文提出的新设计如何进行防御.

CVE-2022-23 222[4]影响了 5.8 及更高版本的内核, 攻击者可以利用该漏洞加载一个恶意的 eBPF 程序达到修改任意内核内存的目的. 通过将进程描述符 (task_struct 结构体) 中的 uid、gid 等域置为 0, 攻击者可以进一步获取管理员权限. 该 CVE 出现的根本原因是 eBPF 检查器存在漏洞, 在处理一些特殊的指针类型的时候 (*_OR_NULL) 没有限制其算术运算, 攻击者可以利用这一点绕过检查器的检查以修改内核内存.

代码 1 列出了利用该漏洞攻击内核的 eBPF 程序的伪代码. 第 1 行是一个辅助函数调用, 被用于向内核申请保留一定大小的环状缓存区, 其中的第 2 个参数是保留的缓冲区的大小, 其返回值的类型被检查器识别为指向内存的可能为空的指针 (PTR_TO_MEM_OR_NULL). 在调用该辅助函数时将第 2 个参数设置为最大的 64 位整数, 因此该函数的实际返回值为空指针. 执行完第 2 行的赋值操作后, r1 的类型也被识别为 PTR_TO_MEM_OR_NULL, 其实际值也为空指针. 在 eBPF 的设计中, 指针是不能直接进行算术运算的, 但是由于检查器存在漏洞, 未能拒绝第 3 行的算术运算, 此时 r1 的实际值为 x, 其类型仍旧为指针. 在第 4 行的判断后, 检查器将认为第 5 行所在的分支中的 r0 是空指针, 由于检查器认为 r0 和 r1 指向同一个对象, 此时检查器会错误地认为 r1 也为 0, 而实际上 r1 可以是用户自定义的任意值 x. 第 5 行中的 r10 为栈顶指针, 检查器认为该语句仅访问了栈区域, 但是这条指令使得程序拥有了访问内核中任意内存的能力.

代码 1. 利用 CVE-2022-23222 漏洞攻击内核的恶意 eBPF 程序伪代码.

```
1.  r0 = bpf_ringbuf_reserve(ptr, u64_max, 0);
2.  r1 = r0;
3.  r1 = r1 + x;
4.  if r0 为空 then
5.      *(r10 + r1) = y
6.  end if
```

检查器中还存在很多类似的漏洞, 例如在 CVE-2021-45402[34]和 CVE-2021-3490[35]中, 检查器在模拟执行

eBPF 程序时没有正确地更新变量的取值范围, 恶意的 eBPF 程序可能会访问非法内存; 在 CVE-2021-20268[44]中, 计算 32 位整数的乘法时可能会导致整数溢出, 当攻击者调用内存分配函数时, 检查器无法正确追踪分配的空间大小, 恶意的 eBPF 程序可以利用该漏洞访问非法内存导致内核崩溃.

导致这些漏洞的直接原因各不相同, 但是其根本原因都是检查器在模拟检查时忽略了一些特殊情况, 导致恶意 eBPF 可以绕过检查器的检查. 我们观察到这些漏洞的最终结果总是使得 eBPF 程序拥有了访问 eBPF 程序之外的内存空间的权限, 进一步导致内核敏感数据被泄露、内核崩溃或者用户程序的权限被非法提升.

而本文提出的基于 PKS 的动态检查机制可以很好地防御此类攻击, 尽管没有检查器的检查, 但是 PKS 机制限制了 eBPF 程序能够访问的内存地址范围, 当恶意 eBPF 程序尝试访问内核沙箱外的内存时, 该动态检查机制就会触发一个 PKS 缺页异常阻止其访问.

4.2 易用性分析

由于 eBPF 检查器过于严格, 开发者不得不从 eBPF 检查器的角度审视提交的代码, 这为开发工作带来了很大的负担. 在很多时候尽管提交的代码符合 eBPF 程序的安全规范 (不会出现死循环或者访问到超出 eBPF 内存模型的内存空间), eBPF 检查器仍然有可能拒绝该程序. 本节从真实使用场景出发, 列举了一些影响 eBPF 机制易用性的因素, 并说明本文提出的新设计可以显著降低开发 eBPF 程序的难度.

(1) 函数需要强制内联. eBPF 检查器在模拟执行时对每一个函数的分析相对独立, 因此函数参数的追踪信息在调用过程中无法被保留. 在 eBPF 程序的开发中通用的做法是将所有函数强制内联到主函数中. 许多开发者没有注意到这一点, 导致程序无法通过检查. 这种“错误”往往也很隐蔽, 因为 eBPF 检查器的报错信息往往与之无关. 代码 2 列出了一个无法通过检查器的代码的例子.

代码 2. 函数需要强制内联的例子.

```
1.  static int inline foo_internal(struct __sk_buff* skb, struct cb_space* cb) {
2.      //... Use skb and cb
3.  }
4.  SEC("classifier/foo")
5.  int foo(struct __sk_buff* skb) {
6.      return foo_internal(skb, (struct cb_space*)&(skb->cb));
7.  }
```

代码 2 展示的场景较为普遍. 第 1 行的 inline 关键字只是给编译器一个内联的提示, 而当该函数复杂度越来越高时, 编译器可能会出于优化代码的目的取消内联, 因此应当将 inline 改为 always_inline 强制编译器使用内联以避免这种情况的发生. 在上面这个例子中, cb 在 foo_internal 中被识别为指向上下文的指针, 但是检查器无法获取它与 skb 之间的关系, 因此在访问 cb 时会报错. 而在本文提出的新设计中, 检查器对上下文指针的检查被替换成了 PKS 动态检查, 因此上述代码可以正常运行.

(2) 无法追踪变量间关系. eBPF 还存在一个显著的问题, 即无法准确追踪变量之间的关系. 代码 3 中列出了一个访问栈区域的例子, 该程序无法通过 eBPF 检查器的检查.

代码 3. 检查器无法追踪变量关系的例子.

```
1.  r1 = r0;
2.  if (r0 > 512) return;
3.  *(u8 *)(r10 – r1) = 0; // r10 是栈的基地址
```

在代码 3 中, 初始时检查器记录的 r0 和 r1 的取值范围都是 [0, u64_max]. 第 1 行赋值语句结束后两个寄存器的取值范围相同, 但是检查器未能将这个信息用于此后的分析. 第 2 行条件分支语句执行结束后 r0 与 r1 的取值

范围都为 [0, 512), 但是检查器未能正确分析出 r1 的取值范围. 第 3 行语句原本应是访问栈上的合法内存, 但是由于检查器认为 r1 的取值范围为 [0, u64_max], 因此该程序会被检查器拒绝. 与之类似的还有一些变种情况, 例如将 r1 是一个指向上下文区域的指针, 在 r1 上执行自增操作会导致检查器失去对 r1 类型的追踪, 此后便无法使用 r1 访问上下文区域 (尽管做了完整的范围检查).

这些检查出现“假阳性”的根源都在于 eBPF 检查器的静态检查机制无法理解较为复杂的程序语义, 导致无法正确判断程序的安全性. 而在使用本文提出的基于 PKS 的动态检查机制的系统中, 仅需保证程序执行时不访问 eBPF 内存模型区域以外的内存即可, 上述代码可以正常运行.

(3) 阻碍编译器进行代码优化. 检查器制定了许多安全规则以简化检查的逻辑, 违反这些规则无法通过检查器的检查, 但是并不代表程序是不安全的, . 为了生成符合检查器安全规则的代码, LLVM 编译器不得不在代码优化上做出妥协. 代码 4 列出的是 eBPF 检查器的栈访问对齐的规则阻碍编译器进行代码优化的例子.

代码 4. eBPF 检查器的安全检查阻碍编译器进行代码优化的例子.

```
1.  // 优化前
2.  *(u8 *)(r0 + off) = 0;
3.  *(u8 *)(r0 + off + 1) = 0;
4.  // 期望优化后
5.  *(u16 *)(r0 + off) = 0;
```

检查器的安全规则要求栈上的访存需要以访存的粒度对齐. 在代码 4 中, 如果 r0+off 是偶数, 那么可以将两次访存优化为一次访存; 而如果是奇数, 那么这个优化就可能会导致无法通过检查器的检查. 因此 LLVM 编译器选择保守的优化策略, 不对此类代码进行优化. 出现此类问题的根源在于代码优化与代码安全检查耦合了, 本文提出的基于 PKS 的机制能够将运行前的优化和运行时的检查解耦, 这给予了 LLVM 等编译器更多的优化空间.

4.3 性能测试

为了准确评估本文提出的方案的性能开销, 我们开展了微基准测试和真实应用场景测试. 其中微基准测试的目标是测试动态检测机制引入的开销. 尽管 PKS 硬件特性保证在内存检查时不引入额外开销, 但是在内核沙箱的出入口仍需要切换内存域, 此外还需要设置影子栈以及配置影子对象, 微基准测试旨在测试这些部分的耗时. 真实场景测试则针对一些流行项目中的 eBPF 程序, 分析并测试这套新机制给整个系统带来的负载.

测试环境: 我们在搭载 Intel i7-10700 CPU 的机器上开展测试, 并将 CPU 的频率锁定为 2.0 GHz 以减小实验数据波动. 机器上运行内核版本为 5.10 的 Ubuntu 18.04LTS 操作系统. 由于 Intel 暂未发布支持 PKS 的处理器, 我们在实现该机制的时候使用 PKU 模拟 PKS. PKU 机制通过用户态指令 WRPKRU 来实现快速的内存域权限切换, 其指令开销约为 28 个时钟周期. 而 PKS 机制通过特权级指令 WRMSR 来实现内存域切换, 其指令开销暂不可知. 根据 WRMSR 指令写的特殊模块寄存器 (model-specific register, MSR) 的不同, 该指令的开销也不尽相同, 但测得其开销一般少于 100 个时钟周期. 我们将内存域切换指令换为等待 80 个时钟周期.

4.3.1 微基准测试

我们利用 Linux 内核提供的 map_perf_test 程序进行微基准测试. 该程序被用于向指定类型的 eBPF map 中依次执行一次更新、查找和删除操作, 并测试其吞吐量, 表 2 展示了其中两个有代表性的数据. hash map 是 eBPF 程序中常用的 map 类型, 可以快速地查找元素; percpu hash map 则是为每个 CPU 维护了一个 hash map, 可以避免多个 CPU 间竞争资源.

表 2 微基准测试下不同种类 map 的吞吐量 (Mops/s)

map类型	无动态隔离机制	动态隔离机制
hash map	5.34	3.64
percpu hash map	5.21	3.61

由于测试时 CPU 的频率被限定为 2 GHz, 因此可以计算出两种情况下 eBPF 程序的执行时间. 以 hash map 为例, 无动态隔离的时候该 eBPF 程序的执行需要 379 个时钟周期, 而动态隔离的时候 eBPF 程序的执行周期为 549 个时钟周期, 因此动态隔离带来的开销约为 170 个时钟周期.

动态隔离带来的开销可以被进一步地划分为切换 pkrs 权限的开销、设置影子栈的开销、设置并拷贝影子对象的开销, 以及辅助函数的开销. 其中切换 pkrs 权限的开销占了绝大部分 (eBPF 程序的入口和出口处分别执行一次 WRMSR, 共 160 个时钟周期). 由于影子栈与影子对象均在 eBPF 程序加载的时候分配, 其地址被储存在 eBPF 程序的内核表示的结构体中, 可以直接获取. 初始化影子对象的时候还需要将部分域从原对象中拷贝到影子对象. 这部分也只涉及少量内存访问, 开销也很低. 此外, eBPF 用到的 map 在沙箱内部分配, 调用操作 map 的辅助函数也无需出入沙箱. 综上所述, 微基准测试结果表明执行 PKS 动态检测机制的开销约为 170 个时钟周期, 其中的绝大部分来源于 eBPF 函数的入口和出口处的内存域切换.

4.3.2 真实应用场景测试

我们首先测试了动态检查机制对真实应用的影响. memcached 作为一个通用的分布式内存对象缓存系统被广泛应用于网络应用中. 我们在待测试的机器上部署了 memcached 作为后端, 在另一台机器上生成 YCSB[45]负载 (YCSB-A–YCSB-F) 并向测试机通过网络发起请求. 我们在测试机上加载了一个 XDP 类型的 eBPF 程序做端口转发, 将网络请求转发到 memcached 监听的端口上. 我们分别测试了在使用动态隔离机制前后系统的吞吐量, 结果如图 7 所示 (因为结果相似, 我们略过了 3 个负载测试). 测试结果表明, 动态隔离机制对系统吞吐量的影响几乎可以忽略不计. 这一方面是因为 PKS 硬件提供了几乎无额外开销的访存检查能力; 另一方面是因为本文提出的设计将进出一次沙箱的开销降低到了不到 200 个时钟周期, 这与系统中其他任务的耗时相比可以忽略不计.

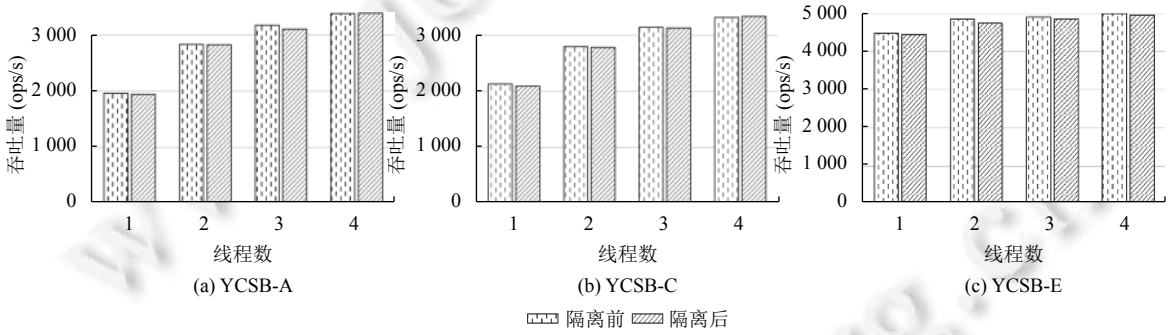

图 7 YCSB 基准测试

进一步地, 我们分别针对 eBPF 程序的 3 种典型应用场景 (网络包处理、内核代码追踪和内核安全监测) 进行测试和分析, 更具一般性地评估本设计在真实应用场景中为系统带来的负载.

(1) 网络包处理场景. 我们利用 Cilium 项目[8]来分析网络包处理场景. Cilium 是一个基于 eBPF 的高性能容器网络领域的开源项目, 主要面向 Kubernetes 等容器编排场景提供应用程序间的网络连接与负载均衡. 第 4.3.1 节中的微基准测试的结果表明, 执行一个 eBPF 程序的额外开销约为 170 个时钟周期, 这与一个数据包在内核中的生命周期相比可以忽略不计. 从整个系统的角度, 我们考虑在性能峰值的最坏情况下, 假设 eBPF 程序处理包的速率为每秒处理 500k 个包, 假设 CPU 的频率为 2.8 GHz, 执行一次 eBPF 程序带来 170 个时钟周期的开销, 可以计算出此时整个系统的性能会下降约 3%.

(2) 内核代码追踪场景. 在该场景下, 用户一般利用 bpftrace[10]等工具来分析程序的调用链以及系统的运行状态. 用户往往更关注于 eBPF 机制提供的功能特性而不是性能, 而且 eBPF 程序被执行的频率要远小于网络包处理的场景, 本机制带来的开销几乎可以忽略不计.

(3) 内核安全监测场景. eBPF 机制可以被用来保护内核安全, 例如在 Falco 项目[11]中, 可以将用户自定义的代码注入到系统调用的开始处, 并检查调用的参数, 以阻止不被允许的系统调用. 与内核代码追踪场景类似, 该场景

中 eBPF 程序执行的频率一般也很低, 本机制也不会给整个系统带来可观测到的性能开销.

综合以上分析, 基于 PKS 的动态检查机制在 eBPF 的典型应用场景中性能表现良好, 能够支持较高的吞吐量的网络包处理场景.

5 相关工作

为了解决检查器的“假阴性”和“假阳性”的问题, 现有工作[2,5,46]往往通过优化静态检查的方式来增强 eBPF 系统的功能.

Gershuni 等人的工作[2]实现了一个基于抽象解释 (abstract interpretation) 方法的 eBPF 解释器 PREVAIL, 该工具可以将 eBPF 二进制文件翻译成一种基于执行流图的语言, 进而可以被基于抽象解释的工具读取并分析. 该方案与 Linux 内核中的解释器相比, 能够正确识别更多的 eBPF 程序, 而且支持程序中的循环结构. 然而 PREVAIL 的检测速度比 Linux 内核中的检查器要慢, 而且不支持 map 嵌套、网络包重分配等功能. 此外, PREVAIL 和当前内核 eBPF 实现不兼容, 它仅实现了检查器, 而 Linux 内核中的检查器除了做基本的检查, 还会改写部分 eBPF 代码, 与后续的 JIT 编译过程紧耦合; 它还依赖抽象解释系统 Crab[47], 这都使得它难以被融入内核系统. PREVAIL 仍属于静态检查, 因此也存在潜在的“假阴性”的问题.

Mahadevan 等人在 2021 年提出的 PRSafe 系统[46]引入了一个非图灵完备的领域特定语言 (domain-specific language), 该语言基于原始递归函数 (primitive recursive functions) 的内存和输入特性确保所有的计算都是可终止的. 该工作旨在设计一个对用户更友好的可以在数学上被验证的 eBPF 编译器, 进而替代现有的 eBPF 检查器. 但是该工作还处于一个较为初级的阶段, 略过了一些关键问题, 例如它依赖 Z3 SMT solver 进行安全检查, 但是如何提供与 eBPF 检查器相同的功能被当作了未来的工作.

Nelson 等人在 2021 年提出的 ExoBPF 系统[5]也聚焦于检查器的假阴性与假阳性问题. ExoBPF 需要开发者自行提供 eBPF 程序的正确性证明, 因此得以将整个 eBPF 检查器移出内核. ExoBPF 计划支持不同种类的检查器, 包括使用 SAT solving 的方式分析大型的程序. 目前, ExoBPF 支持使用 lean theorem prover[48]证明程序的安全性, 开发者只需提供证明的过程, ExoBPF 即可从中解析出 eBPF 程序. 这套方案的缺点在于给开发者带来了较大的证明负担, 因为开发者需要了解形式化证明才能很好地使用这套机制.

为了在内核中构建沙箱, 现有工作[31,49]主要是针对驱动或者内核栈等数据结构设计的, 均无法用于运行 eBPF 程序. Narayanan 等人在 2020 年提出的 LVD[49]支持在虚拟化环境中将驱动程序运行在内核沙箱中. 该工作提出将待隔离驱动所在的物理内存置于独立的 extended page table (EPT) 中, 当执行到驱动代码时切换至该 EPT, 以防止恶意驱动随意访问内核内存. 该工作需要将内核运行在虚拟化环境中, 这限制了其使用场景. Gravani 等人在 2021 年提出的 IskiOS[31]支持在内核中利用 PKU 模拟 PKS 构建内核沙箱, 为了防御代码复用攻击, 该工作将内核的代码段置于不可读不可写的沙箱中, 并将内核影子栈置于只读的沙箱中.

尽管本文主要利用 Intel 处理器提供的 PKS 硬件特性设计 eBPF 内存隔离机制, 但是该机制可以被推广到其他主流处理器硬件上. 例如 ARM 在 ARMV8 和 AArch32 中引入了内存域 (memory domain) 机制[50], 也可以将地址空间划分为多个相互隔离的域; IBM Power 架构[51]处理器采用 5 位的内存域标识符, 可以将内核地址空间划分位 32 个相互隔离的域. 这些机制均可被用于构建可供 eBPF 程序运行的内核沙箱.

6 结束语

本文分析了 eBPF 检查器存在的“假阴性”与“假阳性”问题, 提出了一种基于新型 PKS 硬件特性的 eBPF 内存隔离机制. 本文提出的新方案在一方面减少了 eBPF 检查器的代码量, 可以在运行过程中拦截恶意的 eBPF 程序访存, 解决了“假阴性”问题; 在另一方面将检查器的部分工作移至运行时检查, 因此允许更灵活的语义, 减轻了 eBPF 开发者的负担, 并给 eBPF 编译器提供了更激进优化的可能性. 同时, 性能测试与分析表明本技术在真实系统上带来的开销可以忽略不计.

References:

[1] Calavera D, Fontana L. Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking. Sebastopol: O'Reilly Media, 2019.

[2] Gershuni E, Amit N, Gurfinkel A, Narodytska N, Navas JA, Rinetzky N, Ryzhyk L, Sagiv M. Simple and precise static analysis of untrusted Linux kernel extensions. In: Proc. of the 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Phoenix: ACM, 2019. 1069–1084. [doi: 10.1145/3314221.3314590]

[3] CVE: Search Result. 2022. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=BPF

[4] CVE-2022–23222. 2022. https://nvd.nist.gov/vuln/detail/CVE-2022-23222

[5] Nelson L, Wang X, Torlak E. A proof-carrying approach to building correct and flexible in-kernel verifiers. Technical Report, Linux Plumbers Conf. 2021.

[6] Guide P. Intel® 64 and IA-32 architectures software developer's manual. Volume 3B: System programming Guide, Part. 2011. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[7] McCanne S, Jacobson V. The BSD packet filter: A new architecture for user-level packet capture. In: Proc. of the 1993 USENIX Winter Conf. San Diego: ACM, 1993. 2.

[8] Cilium: Linux Native, API-Aware Networking and Security for Containers. 2022. https://cilium.io/

[9] BCC: IO Visor Project. 2022. https://www.iovisor.org/technology/bcc

[10] bpftrace: High-level tracing language for Linux systems. 2022. https://bpftrace.org/

[11] The Falco Project. 2022. https://falco.org/

[12] The Katran Project. 2022. https://github.com/facebookincubator/katran

[13] A seccomp overview. 2022. https://lwn.net/Articles/656307/

[14] The extended Berkeley Packet Filter (eBPF) backend. 2022. http://llvm.org/docs/CodeGenerator.html#the-extended-berkeley-packet-filter-ebpf-backend

[15] eBPF maps. 2022. https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps.html

[16] Burow N, Zhang XP, Payer M. SoK: Shining light on shadow stacks. In: Proc. of the 2019 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2019. 985–999. [doi: 10.1109/SP.2019.00076]

[17] Hedayati M, Gravani S, Johnson E, Criswell J, Scott ML, Shen K, Marty M. Hodor: Intra-process isolation for high-throughput data plane libraries. In: Proc. of the 2019 USENIX Conf. on USENIX Annual Technical Conf. Renton: ACM, 2019. 489–504.

[18] Kjellqvist C, Hedayati M, Scott ML. Safe, fast sharing of memcached as a protected library. In: Proc. of the 49th Int'l Conf. on Parallel Processing-ICPP. Edmonton: ACM, 2020. 1–8. [doi: 10.1145/3404397.3404443]

[19] Vahldiek-Oberwagner A, Elnikety E, Duarte NO, Sammler M, Druschel P, Garg D. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In: Proc. of the 28th USENIX Conf. Security Symp. Santa Clara: ACM, 2019. 1221–1238.

[20] PKS: Add Protection Keys Supervisor (PKS) support. 2022. https://lwn.net/Articles/826091/

[21] Sehr D, Muth R, Biffle C, Khimenko V, Pasko E, Schimpf K, Yee B, Chen B. Adapting software fault isolation to contemporary CPU architectures. In: Proc. of the 19th USENIX Conf. on Security. Washington: ACM, 2010. 1.

[22] Wahbe R, Lucco S, Anderson TE, Graham SL. Efficient software-based fault isolation. In: Proc. of the 14th ACM Symp. on Operating Systems Principles. Asheville: ACM, 1993. 203–216. [doi: 10.1145/168619.168635]

[23] Zhang YF, Huang C, Ou JS, Tang EY, Chen X. Research on reliability and correctness assurance methods and techniques for device drivers. Ruan Jian Xue Bao/Journal of Software, 2015, 26(2): 239−253 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4778.htm [doi: 10.13328/j.cnki.jos.004778]

[24] Sharif MI, Lee W, Cui WD, Lanzi A. Secure in-VM monitoring using hardware virtualization. In: Proc. of the 16th ACM Conf. on Computer and Communications Security. Chicago: ACM, 2009. 477–487. [doi: 10.1145/1653662.1653720]

[25] Zhong BN, Deng L, Zeng QK. Kernel-level multi-domain isolation model based on hardware virtualization. Ruan Jian Xue Bao/Journal of Software, 2022, 33(2): 473–497 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6211.htm [doi: 10.13328/j.cnki.jos.006211]

[26] Yu J, Huang H, Zhu Y, Xu FY. DBox: High-performance secure boxes for various device drivers of monolithic kernels. Chinese Journal of Computers, 2020, 43(4): 724–739 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2020.00724]

[27] Criswell J, Dautenhahn N, Adve V. KCoFI: Complete control-flow integrity for commodity operating system kernels. In: Proc. of the 2014 IEEE Symp. on Security and Privacy. Berkeley: IEEE, 2014. 292–307. [doi: 10.1109/SP.2014.26]

[28] Dong XW, Shen ZJ, Criswell J, Cox AL, Dwarkadas S. Shielding software from privileged side-channel attacks. In: Proc. of the 27th USENIX Conf. on Security Symp. Baltimore: ACM, 2018. 1441–1458. [doi: 10.5555/3277203.3277311]

[29] Pomonis M, Petsios T, Keromytis AD, Polychronakis M, Kemerlis VP. Kernel protection against just-in-time code reuse. ACM Trans. on Privacy and Security, 2019, 22(1): 5. [doi: 10.1145/3277592]

[30] Gu JY, Wu XY, Li WT, Liu N, Mi ZY, Xia YB, Chen HB. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In: Proc. of the 2020 USENIX Annual Technical Conf. ACM, 2020. 401–417.

[31] Gravani S, Hedayati M, Criswell J, Scott ML. Fast Intra-kernel Isolation and Security with IskiOS. In: Proc. of the 24th Int'l Symp. on Research in Attacks, Intrusions and Defenses. San Sebastian: ACM, 2021. 119–134. [doi: 10.1145/3471621.3471849]

[32] Bounded loops in BPF for the 5.3 kernel. 2019. https://lwn.net/Articles/794934/

[33] Xu QW, Wong MD, Wagle T, Narayana S, Sivaraman A. Synthesizing safe and efficient kernel extensions for packet processing. In: Proc. of the 2021 ACM SIGCOMM Conf. ACM, 2021. 50–64. [doi: 10.1145/3452296.3472929]

[34] CVE-2021–45402. 2022. https://nvd.nist.gov/vuln/detail/CVE-2021-45402

[35] CVE-2021–3490. 2022. https://nvd.nist.gov/vuln/detail/CVE-2021-3490

[36] Yee B, Sehr D, Dardyk G, Chen JB, Muth R, Ormandy T, Okasaka S, Narula N, Fullagar N. Native client: A sandbox for portable, untrusted x86 native code. In: Proc. of the 30th IEEE Symp. on Security and Privacy. Oakland: IEEE, 2009. 79–93. [doi: 10.1109/SP.2009.25]

[37] Koning K, Chen X, Bos H, Giuffrida C, Athanasopoulos E. No need to hide: Protecting safe regions on commodity hardware. In: Proc. of the 20th European Conf. on Computer Systems. Belgrade: ACM, 2017. 437–452. [doi: 10.1145/3064176.3064217]

[38] Hsu TCH, Hoffman K, Eugster P, Payer M. Enforcing least privilege memory views for multithreaded applications. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. Vienna: ACM, 2016. 393–405. [doi: 10.1145/2976749.2978327]

[39] Litton J, Vahldiek-Oberwagner A, Elnikety E, Garg D, Bhattacharjee B, Druschel P. Light-weight contexts: An OS abstraction for safety and performance. In: Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation. Savannah: ACM, 2016. 49–64.

[40] BPF: Allow extended BPF programs access skb fields. 2015. https://lwn.net/Articles/636647/

[41] Park S, Lee S, Xu W, Moon H, Kim T. libmpk: Software abstraction for intel memory protection keys (intel MPK). In: Proc. of the 2019 USENIX Annual Technical Conf. Renton: USENIX Association, 2019. 241–254.

[42] Xu YC, Ye CC, Solihin Y, Shen XP. Hardware-based domain virtualization for intra-process isolation of persistent memory objects. In: Proc. of the 47th ACM/IEEE Annual Int'l Symp. on Computer Architecture. Valencia: IEEE, 2020. 680–692. [doi: 10.1109/ISCA45697.2020.00062]

[43] Schrammel D, Weiser S, Steinegger S, Schwarzl M, Schwarz M, Mangard S, Gruss D. Donky: Domain keys—Efficient in-process isolation for RISC-V and x86. In: Proc. of the 29th USENIX Conf. on Security Symp. ACM, 2020. 1677–1694.

[44] CVE-2021–20268. 2022. https://nvd.nist.gov/vuln/detail/CVE-2021-20268

[45] Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In: Proc. of the 1st ACM Symp. on Cloud Computing. Indianapolis: ACM, 2010. 143–154. [doi: 10.1145/1807128.1807152]

[46] Mahadevan SV, Takano Y, Miyaji A. PRSafe: Primitive recursive function based domain specific language using LLVM. In: Proc. of the 2021 Int'l Conf. on Electronics, Information, and Communication. Jeju: IEEE, 2021. 1–4. [doi: 10.1109/ICEIC51217.2021.9369763]

[47] Gurfinkel A, Kahsai T, Komuravelli A, Navas JA. The SeaHorn verification framework. In: Proc. of the 27th Int'l Conf. on Computer Aided Verification. San Francisco: Springer, 2015. 343–361. [doi: 10.1007/978-3-319-21690-4_20]

[48] de Moura L, Kong S, Avigad J, van Doorn F, von Raumer J. The Lean theorem prover (system description). In: Proc. of the 25th Int'l Conf. on Automated Deduction. Berlin: Springer, 2015. 378–388. [doi: 10.1007/978-3-319-21401-6_26]

[49] Narayanan V, Huang YZ, Tan G, Jaeger T, Burtsev A. Lightweight kernel isolation with virtualization and VM functions. In: Proc. of the 16th ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments. Lausanne: ACM, 2020. 157–171. [doi: 10.1145/3381052.3381328]

[50] ARM developer suite developer guide. 2001. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0056d/BABBJAED.html

[51] RISC-V. ISA specification. 2022. https://riscv.org/specifications/

附中文参考文献:

[23] 张一帆, 黄超, 欧建生, 汤恩义, 陈鑫. 设备驱动程序可靠性和正确性保障方法与技术研究进展. 软件学报, 2015, 26(2): 239−253. http://www.jos.org.cn/1000-9825/4778.htm [doi: 10.13328/j.cnki.jos.004778]

[25] 钟炳南, 邓良, 曾庆凯. 基于硬件虚拟化的内核同层多域隔离模型. 软件学报, 2022, 33(2): 473–497. http://www.jos.org.cn/1000-9825/6211.htm [doi: 10.13328/j.cnki.jos.006211]

[26] 余劲, 黄皓, 诸渝, 许封元. DBox: 宏内核下各种设备驱动程序的高性能安全盒. 计算机学报, 2020, 43(4): 724–739. [doi: 10.11897/SP.J.1016.2020.00724]

李浩(1999－), 男, 硕士生, 主要研究领域为操作系统架构与安全.

古金宇(1994－), 男, 博士, 助理研究员, CCF 专业会员, 主要研究领域为操作系统, 系统安全.

夏虞斌(1982－), 男, 博士, 副教授, 博士生导师, CCF 高级会员, 主要研究领域为计算机体系结构, 操作系统, 虚拟化, 系统安全.

臧斌宇(1965－), 男, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为操作系统, 计算机体系结构.

陈海波(1982－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为操作系统, 并行与分布式系统, 虚拟化, 系统安全.