一种基于风险代码抽取的控制流保护方法

李勇钢[1)] 钟叶青[2)] 郑伊健[1)] 林果园[1)] 鲍 宇[1)]

[1)](中国矿业大学计算机学院信息安全系 江苏 徐州 221116)

[2)](香港中文大学(深圳)数据科学学院 广东 深圳 518172)

摘 要 代码复用攻击是控制流安全面临的主要威胁之一.虽然地址分布随机化能够缓解该攻击,但它们很容易被代码探测技术绕过.相比之下,控制流完整性方法具有更好的保护效果.但是,现有的方法要么依赖于源码分析,要么采用无差别跟踪的方式追踪所有的控制流转移.前者无法摆脱对源码的依赖性,后者则会引入巨大的运行时开销.针对上述问题,本文提出一种新的控制流保护方法 MCE(Micro Code Extraction).MCE 的保护目标是源码不可用的闭源对象.与现有的方法相比,MCE 并不会盲目地追踪所有的控制流转移活动.它实时地检测代码探测活动,并仅将被探测的代码作为保护目标.之后,MCE 抽取具有潜在风险的代码片段,以进一步缩小目标对象的大小.最后,所有跳转到风险代码中的控制流都会被追踪和检测,以保护它的合法性.实验和分析表明,MCE 对代码探测和代码复用攻击具有良好的保护效果,并在一般场景下仅对 CPU 引入 2%的开销.

关键词 代码探测;代码复用攻击;控制流劫持;代码抽取;内存访问控制

中图法分类号 TP309 **DOI 号** 10.11897/SP.J.1016.2024.01372

A Control Flow Protection Method Based on Code Extraction

LI Yong-Gang[1)] CHUNG Yeh-Ching[2)] ZHENG Yi-Jian[1)] LIN Guo-Yuan[1)] BAO Yu[1)]

[1)](*Department of Information Security, School of Computer Science, China University of Mining and Technology, Xuzhou, Jiangsu* 221116)

[2)](*School of Data Science, Chinese University of Hong Kong (Shenzhen), Shenzhen, Guangdong* 518172)

Abstract Code reuse attack is one of the main threats to control flow security. Although address space layout randomization can mitigate this attack, it can be bypassed by code probes. In contrast, control flow integrity methods have better protection effects. However, either rely on source code or track all control flows in the entire life cycle of the target process. The former cannot protect the closed source objects, while the latter introduces significant runtime overhead. In response to the above issues, this paper proposes a control flow protection method MCE (Micro Code Extraction). The protection targets of MCE are closed source objects whose source are unavailable. Compared with existing methods, MCE does not blindly track all control flow transfer activities. It detects code probes in real-time and only targets the probed code as a protection target. Afterwards, MCE extracts the code snippets with potential risks to further reduce the size of the target object. Finally, all control flows that jump into the risk code will be tracked and detected. Experiments and analysis have shown that MCE has a good protection effect on code probes and code reuse attacks, and only introduces 2% overhead to the CPU in general scenarios.

Keywords code probes; code reuse attacks; control flow hijacking; code extraction; memory access control

收稿日期:2023-07-05;在线发布日期:2024-03-12.本课题得到“中央高校基本科研业务费专项资金”(2023QN1078)资助.**李勇钢**(通信作者),博士,副教授,中国计算机学会(CCF)会员,主要研究方向为系统安全、系统优化.E-mail:liyg@cumt.edu.cn.**钟叶青**,博士,教授,主要研究领域为系统软件、并行分布式计算.**郑伊健**,本科生,主要研究方向为信息安全.**林果园**,博士,副教授,主要研究方向为操作系统、信息安全.**鲍 宇**,博士,副教授,中国计算机学会(CCF)会员,主要研究方向为智能物联网安全.

1 引　言

合法控制流是确保程序有序运转的基本前提[1].但是,攻击者通过劫持控制流能够破坏代码原有的执行逻辑.代码复用攻击(Code Reuse Attacks,CRAs)是一种典型的控制流劫持技术[2],它不需要向目标对象中注入代码,而是将内存中已有的代码用作攻击载荷,并利用漏洞篡改敏感数据[3],进而将控制流劫持到攻击载荷构成的 gadget 链[4],以实现攻击.ROP(Return-Oriented Programming)[5]和 JOP(Jump-Oriented Programming)[6]是 CRAs 的两种最基本的攻击方法.CRAs 的成功部署需要两个基本条件,特定形式的 gadgets(如 pop %rax; call *%rax)和已知的代码地址.

地址空间分布随机化(Address Space Layout Randomization,ASLR)可隐藏代码地址,致使特定形式的代码片段不能被定位到.ASLR 因其较低的性能开销和良好的保护效果而被广泛采用.

然而,事实表明代码探测技术能够绕过传统的 ASLR.现有的探测技术包括分配探测[7]、进程克隆探测[8]、任意读[9]、地址泄露[10]、任意跳转[11]和侧信道泄露[12]等.这些技术既可以定位到代码段范围,又可以破解随机化后的代码地址,甚至能够得到符合特定代码形式的攻击载荷[13].从攻击原理来看,如果在代码探测之后到 CRAs 部署之前的这段时间内,已被探测的代码地址不再发生变化,那么攻击将能够顺利完成.传统 ASLR 在编译或加载阶段改变代码地址.之后,在进程的全生命周期内,代码地址保持不变.因此,传统 ASLR 无法抵御现有的代码探测.虽然进程运行时的周期性随机化方法能够提升防御效果,但它们对两次随机化间隙内的代码探测与控制流劫持毫无效果.从防御原理来看,在代码探测之后到控制流劫持之前进行随机化能够实现安全防御,这也是新型 ASLR 的发展方向.现有的运行时随机化方法为在合适的位置埋设随机化点,并在随机化后维持原有的代码调用关系,不得不依赖源码分析和编译控制,致使其对闭源对象无效.

与 ASLR 相比,控制流完整性(Control Flow Integrity,CFI)直接将控制流作为保护对象.它将指令路径限制在特定的集合内,进而实现对控制流转移活动的强制约束.任何跳转到集合外的控制流都会被判定为非法的.

但是,如何建立精准的合法路径集合,并筛选出合适的控制流检测点,是所有 CFI 方法面临的关键挑战.对控制流转移指令来说,它在执行时有且仅有一个跳转目标是合法的.因此,理想的路径集合应当在跳转指令执行时,仅包含该指令在当前状态下唯一的合法路径.然而,由于闭源软件的内部逻辑是未知的,其合法路径难以被识别.虽然通过分析静态可执行文件能够构建出控制流图(Control Flow Graph,CFG),但该方法面临着状态爆炸问题.爆炸性的 CFG 会涵盖大量冗余路径,甚至会将危险性路径指示为合法路径.此外,现有 CFI 方法为保证保护的全面性,往往将整个代码段作为保护目标,并无差别地追踪所有控制流转移活动.实际上,控制流劫持仅发生在特定场景中的特定代码中.大范围的无差别跟踪会将大量的合法控制流纳入追踪范围,进而造成不必要的开销.虽然基于编译控制的方法能够在特定的代码位置设定检测点,并以较低的开销取得较好的保护效果,但该方法无法推广到闭源软件中.理想的 CFI 方法应当仅在潜在的攻击场景出现时(具备场景针对性),追踪可能被恶意利用的局部代码(具备目标针对性),这也是 CFI 方法未来的发展趋势.

综合来看,ASLR 和 CFI 两类方法都存在一定的局限性.前者会被代码探测绕过,而后者则无法对闭源软件提供高效强力的控制流保护.针对这些问题,本文提出一种基于代码抽取的控制流保护方法 MCE,用于保护闭源对象的控制流安全.

随着 ASLR(尤其是细粒度的 ASLR)的普及,攻击者在部署 CRAs 之前必须要对目标对象进行代码探测,以获取符合攻击特征的 gadgets 及其地址[13].基于该特征,MCE 实时地感知攻击者对闭源对象的探测活动,以发现被探测的函数(即风险函数).之后,MCE 解析该函数中的风险代码.风险代码是指风险函数中可被用作 gadgets 的代码片段,即,包含间接控制流转移(Indirect Control Transfer,ICT)指令和返回指令的代码块.接下来,风险代码会被从原有空间中抽取出来.最后,所有跳转到风险代码的控制流都会被追踪和检测,以确保其合法性.概括地说,本文的主要贡献如下:

(1)提出一种代码探测感知机制.该机制能够筛选出潜在的攻击场景,使 MCE 具备更强的场景针对性.其通过行为侦测发现代码探测活动,并将被探测的函数标记为风险函数.

(2)提出一种风险代码抽取机制.该机制能够从风险函数中抽取二进制代码,并将其迁移到新的

地址空间中.该机制本质上是在代码探测之后对局部代码进行的运行时随机化,其能够防止被探测代码被直接用作攻击载荷.

(3)建立针对风险代码的控制流保护模型.该模型仅追踪跳转到风险代码中的控制流,使得 MCE 具备更强的目标针对性.其根据控制流攻击特征制定上下文敏感的安全策略,以检测跳入和跳出风险代码的控制流,从而摆脱了对源码的依赖性,并能够根据运行时上下文确定当前控制流路径的合法性.

(4)在 Linux 中实现 MCE 原型.据我们所知,MCE 是首个针对小尺寸风险代码的控制流保护方法,具有良好的场景针对性和目标针对性,从而以较小的开销取得较好的效果.

2 背景知识与相关工作

CRAs 使用系统中已有的代码作为攻击载荷而无需向攻击目标中注入任何代码,因而具备更强的隐蔽性和可操作性.从攻击原理来看,部署 CRAs 需要取得攻击载荷,并破坏原有的执行逻辑.因此,对攻击者隐藏攻击载荷,或对易受攻击的代码进行逻辑约束均可实现防御目标.

从攻击发展趋势来看,CRAs 逐渐与新型的代码探测技术相结合,从而击败了当前的代码隐藏技术,如 ASLR.目前,先进的探测方法既能得到特定的 gadget 形式,如基于任意跳转的探测[11],又能得到具体的地址,如基于侧信道的代码探测[12].在代码探测技术的帮助下,CRAs 不需要任何先验知识即可劫持控制流.因此,目前的 CRAs 能够将攻击目标从有源码对象扩展到闭源对象当中.而现有的控制流检测方法仍难以摆脱对源码的依赖性,致使其无法对闭源对象提供强力保护.尤其是软件供应链上的闭源对象,它们的内部逻辑是未知的,加载与运行时间是随机的,给控制流检测方法带来了严峻的挑战.例如,Linux 中的指令"apt install chrome"在安装浏览器 Chrome 时会自动安装 106 个闭源库,且整个安装过程不会给用户预留安全审计窗口.

实际上,针对闭源软件的 CRAs 和代码探测必然会引发异常行为.因此,通过捕捉并分析代码行为即可实现检测目标.硬件辅助的虚拟化技术为高效快速捕捉闭源软件的异常行为提供了可能.基于该技术,闭源软件的内存访问和执行过程都可被控制.现有的 ASLR 方法和 CFI 方法在防御 CRAs 方面取得了一些成效,下面我们将分别介绍它们.

2.1 基于 ASLR 的控制流保护方法

ASLR 是一种间接控制流保护方法,其防御对象是代码探测.代码探测的本质是代码地址与代码形式的获取过程.理想的 ASLR 需要在代码探测发生后到 CRAs 部署之前的这段时间内完成代码随机化,进而使得攻击者探测到的代码地址不再指向原有的代码块.但是,现有的方法并不能兼顾防御效果和执行效率,导致难以得到推广.本文统计并分析了当前具有代表性的 ASLR 方法,如表 1 所示.它们都具有一定的反代码探测能力,但由于无法适时且高效地确定随机化点和随机化对象,致使它们在效果、效率和部署范围等方面仍存在局限性.

表 1 反代码探测方案

Methods	AS	CM	KM	LiM	LoM	G	InP	RS	O	Ins
Mixr[14]	n	n	n	n	n	U	U	U	1.66x	3.51x
Remix[15]	y	y	n	n	n	B	U	U	2.8%	14.8%
Srandomizer [16]	n	n	n	n	n	S	50	S	2.1%	—
fASLR[17]	y	n	n	n	n	F	U	U	10%	<5%
RCMond[18]	y	n	n	n	n	F	SC	S	5%	—
PointerScope[19]	y	y	n	y	n	F/B	L	S	0.45%	0.01 MB~50 MB
Mardu[20]	y	y	y	n	n	B	E	S	5.5%	>1x
Reranz[21]	n	n	y	n	n	B	SC	S	6%	73 MB
CCR[22]	y	y	n	y	n	B	L	S	0.28%	11.46%
SafeHidden[23]	n	n	n	n	n	S	f/SC	U	2.75%	—
Adelie[24]	y	y	n	n	n	B	C	S	<2%	—
RuntimeASLR[8]	n	n	n	n	n	M	f	S	>217x	—

注:AS:访问源码;CM:编译器修改;KM:内核修改;LiM:链接器修改;LoM:加载器修改;G:粒度;InP:随机化点;RS:随机化范围;U:用户指定;number x:每隔 x 毫秒;SC:系统调用;f:fork();E:指定事件;L:代码加载时;C:编译时;F:函数;B:基本代码块;M:模块;S:整个代码段,整个堆或整个栈;O:性能开销;Ins:增加代码或文件的大小.

Mixr[14] 和 Remix[15] 具有目标针对性,但前者因过多的随机化点而影响性能,而后者对源码具有依赖性.周期性随机化 Srandomizer[16] 会给攻击者留下足够大的保护空白期.fASLR[17]、RCMond[18]、PointerScope[19] 和 Mardu[20] 需要源代码提供的高级语义,这使得它们对闭源对象无效.此外,在面对闭

源软件中的代码探测时，绝大多数方法，如 Reranz[21] 和 RuntimeASLR[8]，都无法摆脱目标对象的泛化性问题，进而引发了巨大的内存开销或 CPU 开销. SafeHidden[23] 虽然降低了性能开销，但过于稀疏的随机化点影响了它的防御效果.

从已有的攻击形式来看，局部突发的代码探测导致现有方法的随机化时机与随机化对象都难以被精准地确定，给攻击者留下了明显的攻击窗口和攻击目标. 如果随机化无法在代码探测发生后到 CRAs 部署之前完成，那么随机化将变得毫无意义. 代码运行时的动态随机化是解决该问题的关键，也是随机化方法发展的主要趋势. 但是，现有方法的随机化对象是进程的全部代码，它们需要为整个代码段维持随机化后的调用关系，这对于复杂的闭源软件而言几乎是不可实现的.

本文所提出的 MCE 是一种基于代码探测事件触发的安全方法. 它会将被探测的局部代码迁移到一个新的地址空间中，其本质上也是一种运行时随机化方法. 与现有的随机化方法相比，它仅迁移少量的风险代码，因而具有更强的目标针对性，也避免了随机化之后为整个代码段维持复杂的调用关系. 此外，它仅在代码探测发生场景下迁移代码，因而也具备更强的场景针对性.

2.2　基于 CFI 的控制流保护方法

控制流劫持的本质是打破控制流间的逻辑约束性. 为解决该问题，各类 CFI 方法已被广泛提出. 但是，它们在应用场景、作用对象、执行效率和保护效果方面仍存在一些限制.

为提高分析精度，一些方法不得不依赖源码，如 OS-CFI[25]、CFI-LB[26]、VTI[27] 和 TyPro[28]，这些方法无法对闭源对象提供保护. 基于路径限制的方法需要为目标对象提供精确的指令边界，这会引发控制流约束性问题. 例如，O-CFI[29] 为跳转指令构建了一个目标集合，但集合中的元素不能精准地指向跳转目标，无法提供强力的控制流约束性. 此外，当前的 CFI 方法具有一定的盲目性，它们不加区分地对大量控制流进行追踪、检测和分析，从而引发了目标对象的泛化问题，这也是引发性能问题的关键因素. 例如，C3 引入的运行时开销超过 70%[30]. 相比之下，BBB-CFI 因需扩展硬件而难以推广[31].

现有研究表明，解决控制流的逻辑约束性问题是提升控制流安全性的关键，而建立软硬件协同的方法则是确保执行效率的有效手段[32-37]，这也是 CFI 方法未来的主要发展趋势. 然而，现有的 CFI 方法为构建精准的指令边界并在控制流转移处设定检测点需要源码分析和编译控制，导致它们难以摆脱对源码的依赖性.

与现有方法相比，MCE 基于硬件辅助的虚拟化技术追踪和分析进程，进而可摆脱对源码的依赖性. 从部署原理来看，MCE 会提取二进制码的运行时上下文以得到代码逻辑和数据逻辑，进而可为控制流提供运行时的逻辑约束性.

3　假设与威胁模型

3.1　假设

第一，假设受保护进程的地址空间已经按照函数粒度进行了随机化，攻击者必须探测内存才能找到足够多的 gadgets，这与 Buddy[38] 类似. 第二，假设攻击者可以利用现有的探测技术获取用户空间中的二进制代码信息. 第三，假设攻击者可以利用内存损坏（如溢出漏洞）劫持控制流. 第四，假设攻击者无法修改运行的二进制代码. 第五，假设被攻击的进程可以捕获信号 SIGSEGV 和 SIGILL 之外的所有信号，并对它们进行处理以防止进程崩溃，或可重启因攻击而崩溃的对象.

3.2　威胁模型

攻击向量 1. 内存分配探测[7]. 该技术使用内存分配函数（如 malloc 和 new）来分配特定的区域，返回的结果将显示该区域是否已映射.

攻击向量 2. 进程克隆探测[8]. 该技术使用克隆的子进程来探测父进程的内存布局. 子进程中的非法访问并不会导致父进程崩溃. 因此，攻击者可以在多个子进程中对目标对象进行持续性探测，直到获取到足够的 gadgets 为止.

攻击向量 3. 任意读探测[9]. 该技术可以直接读取目标对象的代码信息.

攻击向量 4. 关键数据泄漏[10]. 该技术读取 PLT（Procedure Linkage Table）中的相对偏移量，以获得 GOT（Global Offset Table）的随机地址，而 GOT 中存储着当前进程所需的所有库函数地址.

攻击向量 5. 任意跳转探测[11]. 该技术通过篡改目标控制数据将控制流重定向到任何位置，并通过执行结果来定位和识别可用的 gadgets.

攻击向量 6. 侧信道泄露[12]. 该技术使用各级页表的缓存命中引发的时间差异性来逐步破解虚拟地址的第 12 位至第 47 位.

4 风险代码抽取系统的总体设计

与现有方法不同的是，MCE 并不会在目标对象的全生命周期内追踪并检测所有的控制流转移活动. 它仅将特定场景中具有潜在风险的微小代码作为保护目标，这能够减少不必要的控制流追踪与分析，进而提升保护效率. 此外，MCE 的保护目标是运行的二进制码，其能够摆脱对源码的依赖性. 为实现上述目标，MCE 需要完成以下几项任务：

第一，在闭源对象运行时，实时地感知代码探测活动. 为获取代码信息，攻击者需要在目标对象运行时探测其代码. 在细粒度随机化场景中，探测者无法通过单次探测或泄露推测出整个代码段的布局. 为得到足够多的 gadgets，攻击者必须对目标对象进行多次探测[39]. 之后，非法控制流会在被探测的代码间流转. 因此，发现探测活动能够识别出潜在的攻击场景，进而缩小目标控制流的追踪范围，以避免不必要的安全检测与分析.

代码探测的本质是攻击者根据直接或间接的内存访问获取代码地址或代码形式. 实际上，合法的内存访问与代码探测之间会存在一定的差异性. 根据探测原理分析，探测活动可能会触发异常信号(向量 5)，访问特定空间(向量 6)，或读取代码(向量 3 和向量 4). 结合上述方法，一些探测技术(如向量 1 和向量 2)还可作为辅助手段实现快速隐蔽的代码探测. 这些差异行为可能会体现在当前的内存访问活动中，也可能会体现在后续的执行过程中. 例如，被合法 debugger 读取的间接控制流转移(ICT)指令，在后续的调用过程中仍会按照合法的代码逻辑跳转到目标代码中，而被攻击者读取的代码则会连接与其不存在调用关系的代码块. MCE 需要实时地感知异常信号、特定的空间访问和代码读取活动，才能发现潜在的代码探测活动.

第二，抽取被探测的函数中的风险代码. 并不是所有被探测的代码都会被攻击者选中. 在现有的 CRAs 攻击模式中，gadgets 中需要包含 ICT 指令或 ret(统称为风险指令). 所以，在被探测的代码中，只有包含风险指令的代码块才是可能被恶意利用的风险代码. 因此，筛选风险代码能够确定潜在的攻击载荷，有助于建立起具有目标针对性的方法.

MCE 会标记风险代码，以捕捉所有跳转到风险代码中的控制流. 然而，被标记的风险代码会破坏原有代码的大小和功能，这会导致执行异常. 为解决该问题，MCE 会将风险代码从原有空间中抽取出来，并将其放置在新的空间中. 新空间由内核函数 do_mmap 生成，其处在与原有空间相距[−4 GB，4 GB]的一个随机位置. 在新空间中，风险代码仍能保持原有的功能，并可将控制流传送到下一个合法位置. 需要说明的是，新空间仍处在当前进程的地址空间范围内. 因此，新旧空间的切换并不需要更新 CR3 寄存器. 此外，由于操作系统采用惰性方式分配物理页，所以并不是所有新空间都会占据物理内存. 实际上，新空间只是占用少量的寻址页表和代码页，并不会占据大量的物理内存.

第三，检测并保护跳转到风险代码中的控制流. 控制流可通过多种方式被传输到风险代码中，包括代码顺序性执行、直接跳转、间接跳转和返回执行. 在代码运行时，跳转源与跳转目标之间是一一对应的. 这种对应关系可通过代码间的逻辑关联性确定. 但是，对于闭源软件中细粒度的风险代码来说，它与其他代码块之间的逻辑关联性是未知的. 因此，难以在风险代码运行前为其唯一地确定跳转源和跳转目标. 为检测跳入和跳出风险代码的控制流，MCE 需要制定上下文敏感的安全策略，以判定跳转源和跳转目标的合法性.

对于合法控制流，MCE 不仅要将跳转到风险代码的控制流传送到被抽取的风险代码中，还要将跳转出风险代码的控制流传送到正确的位置. 为降低性能开销，MCE 还需要避免对合法控制流的重复追踪与检测.

由于仅追踪代码探测场景中已被探测的风险代码，所以 MCE 是一个具有目标针对性和场景针对性的控制流保护方法. 其总体设计如图 1 所示.

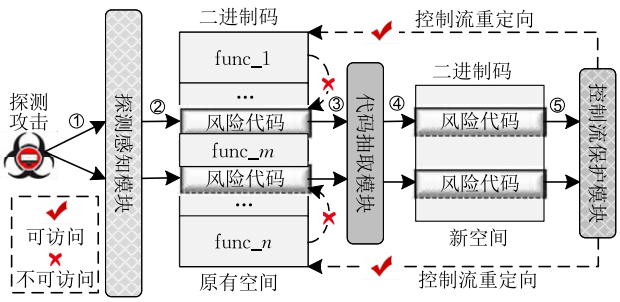


图 1 MCE 的总体设计

MCE 本质上是一个具有安全功能的 hypervisor，旨在利用硬件辅助功能 EPT(Extended Page Table)和 VMX(Virtual Machine Extension)保护控制流安全. 在运行过程中，它会以可加载内核模块(LKM)的形式被加载到内核中.

MCE 包含 3 个关键组件：代码探测感知模块、代码抽取模块和控制流保护模块. 代码探测发生时，代码探测感知模块会定位正在被探测的函数，即风险函数(①②). 之后，代码抽取模块从风险函数中筛选出风险代码，并将其从原有空间迁移到新空间中(③④). 同时，原有空间中的风险代码会被标记. 最后，所有跳转到风险代码的控制流都会被控制流保护模块捕获并分析(⑤). 对于合法的控制流，控制流保护模块会将其重定向到新空间中，以完成原有代码的功能. 在新空间中，控制流保护模块还要在风险代码执行完成后将控制流重定向到原有空间. 需要说明的是，在加载目标进程时，MCE 会利用反汇编工具(如 objdump)解析可执行文件，以得到该进程的汇编代码，用于分析风险代码.

MCE 的保护目标是闭源软件，其内部逻辑是未知的. 闭源软件的逻辑未知性体现为，其指令路径、内存访问和与之相关的特定事件(如调用漏洞函数)等行为都是未知的. 未知性逻辑会阻碍 MCE 识别异常行为. 因此，MCE 需要在闭源软件运行时消除其逻辑未知性. 从系统底层来看，通过监视、追踪和控制闭源软件的执行过程即可获得详细的上下文信息，进而可消除其逻辑未知性. 为了实现该目的，本文利用 VMX root 和 VMX non-root，将原有操作系统(OS)中的运行模式分为 host 和 guest 两种类型. 在正常情况下，OS 运行在 guest 模式中. 当特定事件发生时，运行模式会从 guest 切换到 host，这被称为系统陷入. 结合 EPT 和 VMX，可设定各种系统陷入事件，包括进程切换、断点设置、特定指令的执行(如 int3、vmcall)、中断、单步调试和异常保护等. 此外，还可以通过修改 VMCS(Virtual Machine Control Structure)中的字段来重写 CPU 上下文，进而实现执行过程控制. 例如，通过修改 VMCS 中的 guest rip 可实现控制流重定向. 通过上述设计，MCE 即可实现对受保护进程的行为监视、分析和控制.

5　风险代码抽取系统的实现方法

本节将围绕感知代码探测、抽取风险代码和保护控制流 3 个目标描述 MCE 的实现方法.

5.1　感知代码探测

传统 CFI 方法会在进程全生命周期内追踪并检测所有的控制流转移. 对闭源软件来说，长时间且大范围的控制流追踪与检测会极大地增加实现复杂度，并引发不可忽视的性能开销. 实际上，控制流攻击仅发生在特定的场景中. 在细粒度随机化环境中，代码探测已成为控制流攻击过程中的必要环节. 基于该攻击原理，MCE 通过检测代码探测活动可筛选出潜在的攻击场景.

与合法的内存访问相比，代码探测会表现出特定的行为特征，包括读取代码、触发异常信号和访问特定范围的内存等. 由于代码段是不可写的，探测者只能通过读或执行代码来探测目标对象. 被读取的内容和执行代码产生的结果会直接或间接地向攻击者披露代码地址或代码形式. 因此，通过捕捉探测动作本身或探测产生的结果即可检测到攻击者的探测意图. 本文将从权限感知、空间感知和信号感知 3 个方面建立代码探测感知机制.

5.1.1　权限感知

现有的 XOM 方法[40-42]将代码页设置为不可读，以阻止代码泄露. 但是，应用代码并不是一次性被完全加载到内存中. 为区分代码页和数据页，现有方法需要追踪每一个代码页的分配. 由于代码页无法被直接映射为不可读，现有方法还需要捕捉每一个代码页访问，以筛选出代码可读活动. 这样的设计会引入不可忽视的开销. 例如，Heisenbyte 会对 perlbench 引入超过 60%的开销[43].

为解决这些问题，本文提出一种新的内存管理方法，即内存分离机制，如图 2 所示. 该机制的基本功能是基于虚拟化原理对内存权限实现分类管理. 即利用 EPT 技术提前将用户代码所使用的内存页设定

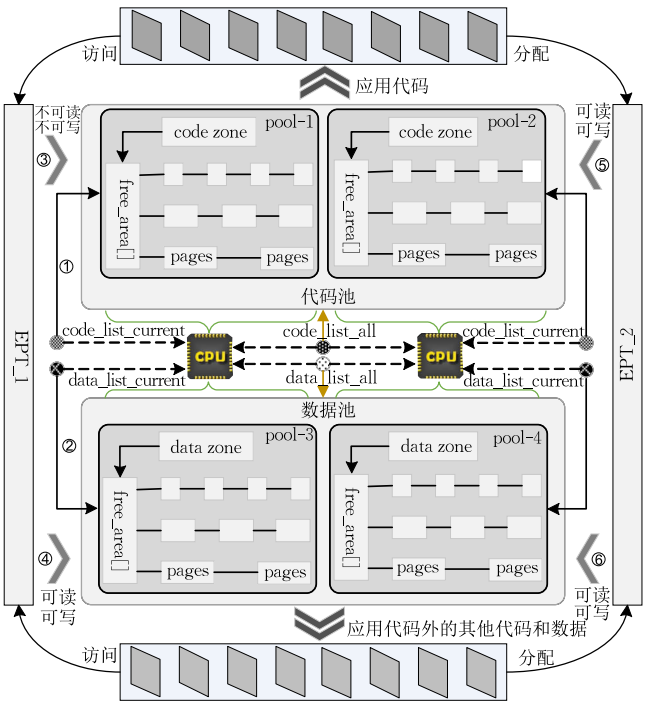


图 2　内存分离机制

为不可读,进而避免频繁地追踪和识别页分配操作.

内存分离机制通过改造原有的 buddy system 将整个物理空间分为两部分,即代码池和数据池.代码池和数据池会被分别设定为不可读的和可读的.所有应用代码所需的内存页都来自于代码池,而其他代码和数据(如数据和内核代码等)都来自于数据池.需要说明的是,内核启动的早期阶段并不使用 buddy system 管理内存,而是使用 memblock 或 bootmem.之后,buddy system 才会进行初始化,并接管内存管理任务.由于我们修改了 buddy system,新 buddy system 在初始化过程中会将所有已分配的物理页(包括早期加载的核心代码)归类到数据池中.而在内核启动之后,来自于数据池的内核代码(如系统调用服务代码)是常驻内存的,并由 buddy system 统一管理.概括地说,该机制可避免追踪代码页分配和代码页访问,也无需部署复杂的代码权限动态调整策略.

在 Linux 中,进程的代码页分配是由 buddy system 完成的.物理内存通过 zone list、zone 和 page 被组织在一起.在 NUMA 架构中,buddy system 对每个 CPU 都使用两个 zone list 管理所有的 zone.第一个 zone list 用于管理与当前 CPU 直连的 zone,第二个 zone list 用于管理所有的 zone.当代码出现缺页异常时,buddy system 会从 zone list(优先选择第一个 zone list)中的某个 zone 里面选取特定数量的物理页分配给进程代码段.

内存分离机制结合 EPT 技术改写原有的 buddy system.在改写后的 buddy system 中,每个 CPU 所能使用的 zone list 数量增加一倍.Code_list_all 和 data_list_all 分别指向代码池和数据池.此外,每个 CPU 都有一个 code_list 和一个 data_list,它们分别指向与当前 CPU 直连的代码 zone 和数据 zone,如图 2 中的①②所示.

在实际中,内存页无法通过 EPT 被同时设定为可写和不可读,否则会引发 EPT 配置异常.在新 buddy system 中,代码池已被设定为了不可读(如图 2 中的③所示),这导致其不可能是可写的.然而,代码加载过程要求代码页必须是可写的,否则将触发权限异常.为解决该矛盾,代码页在代码加载过程中必须是可读和可写的,在加载完成后则变为可读和不可写的.

在内存分离机制中,全部物理页都由 EPT_1 和 EPT_2 轮流管理.前者将代码池设定为不可读和不可写的,而后者则将所有物理页都设定为可读和可写的.VMX 指令 vmfunc 会被添加到函数 filemap_fault(该函数将代码从 ELF 文件加载到内存中)的头部和尾部.缺页异常发生后,vmfunc 会将 EPT_1 切换到 EPT_2.此时,所有页都是可读可写的(如图 2 中的⑤⑥所示),代码页可完成正常加载.当 filemap_fault 返回时,vmfunc 再将 EPT_2 切换回 EPT_1.此时,代码页已经加载到内存中,并且代码页是不可读取和不可写入的.

此外,为保证被换入和换出的代码页保持不可读状态,do_swap_page 会被监视.在 do_swap_page 返回之前,被分配的代码页会被检测是否已被设定为不可读.若否,该页面会被重新设置为不可读的.所以,被交换的代码页仍在 MCE 的保护范围内.同时,MCE 还通过 Intel VT-d 创建了一组重映射表,用于屏蔽代码池中的物理内存.因此,探测者无法通过 DMA(直接内存访问)读取用户代码.

基于上述设计,任何读取代码活动都会因触发权限异常而被捕捉到.之后,当前代码页的读权限会被开启.同时,单步调试模式也会被启动.每一条读指令结束后,当前代码页会被再次切换为不可读的.因此,所有的读取活动都能获取到代码信息,这能够保证合法的代码读取请求.而所有被读取的代码都能够被捕捉到.被读取的代码所在的函数将会被认定为风险函数.综合来看,攻击向量 3 和攻击向量 4 都会被内存分离机制检测到.

5.1.2 空间感知

除了直接读取代码之外,攻击者还可以间接得到代码的空间范围或准确地址.内存分配探测(向量 1)[7]通过内存分配器的返回结果能探测已映射空间,进而定位被隐藏的内存区域.

与之相比,侧信道泄露(向量 6)[12]利用 TLB 命中与非命中之间的时间差能够准确地推测出代码的第 12 位至第 47 位.为观测 TLB 命中状况,攻击者利用 flush+reload[44]、evict+time[45]或 prime+probe[46]反复驱逐或填充 TLB.之后,代码地址的第 15 位至第 20 位、第 24 位至第 29 位、第 33 位至第 38 位和第 42 位至第 47 位可根据访问 TLB 的时间差异性得到.为破解剩余的第 12 位至第 14 位、第 21 位至第 23 位、第 30 位至第 32 位和第 39 位至第 41 位,与目标代码相距 $n\times4$ KB、$n\times2$ MB、$n\times1$ GB 和 $n\times512$ GB($n<8$)的内存会分别被访问.如果这些区域中存在未分配的区域,攻击者需要先分配相应的空

间，再对其发起访问. 对于已分配的区域，攻击者则直接访问该区域.

为感知向量 1 和向量 6，MCE 建立空间感应机制，如图 3 所示. 该机制本质上是通过建立大量的感应段，以将真实代码段隐藏在感应段中. 因此，代码探测过程中产生的越界访问或对特定空间的访问，都可能落在感应段中，进而被捕捉到.

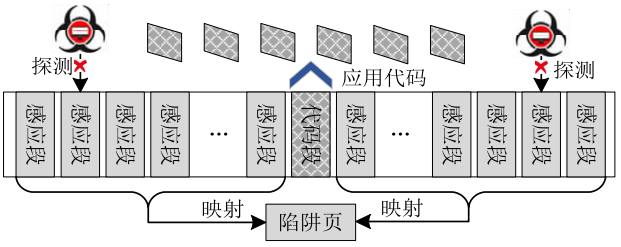


图 3　空间感应机制

在 Linux 中，用户的地址空间多达 128 TB. 实际上，仅有很小的一部分被使用. 剩余的绝大部分空间未被映射. MCE 在目标进程被创建时为其分配远大于/多于原有空间的地址空间. 额外分配的虚拟地址空间(感应段)与原有代码段大小一致，其数量在 $10^3 \sim 10^6$ 之间. 具体数量和段间偏移量都是随机的. 所有感应段都会通过 EPT 被映射到同一个不可读、不可写和不可执行的页中(即陷阱页). 需要说明的是，绝大部分感应段被强制共用同一组寻址页表. 因此，它们并不会占据过多的页表. 此外，与原有代码段相距 $n\times 4$ KB、$n\times 2$ MB、$n\times 1$ GB 和 $n\times 512$ GB ($n<8$)的区域若未被映射，它们也会被映射到陷阱页中.

在空间感应机制的保护下，向量 1 探测到的映射区域非常多，它并不能从中筛选出真实的代码段. 实际上，该设计并不能直接感知到向量 1，但它可使向量 1 失去探测意义. 而向量 6 在访问特定范围的内存时会因访问到陷阱页而被 MCE 捕捉到. 之后，MCE 按照相应的偏移量即可找到正在被探测的代码. 该代码所属的函数会被标记为风险函数.

5.1.3　信号感知

除上述探测方法之外，攻击者还可以利用任意跳转(攻击向量 5)引发的异常信号来筛选出符合特定形式的代码段. 在细粒度的 ASLR 中，向量 5 会大概率地跳转到未映射的区域或非法指令中. 前者会触发信号 SIGSGEV，后者会触发信号 SIGILL. 通过感知异常信号即可发现潜在的代码探测活动.

为捕捉信号 SIGSGEV 和 SIGILL，系统调用 signal 和 sigaction 会被拦截. 此外，如果向量 5 跳转到陷阱页也会因触发权限异常而被捕捉到. 当检测到信号 SIGSGEV 或 SIGILL 时，触发信号的用户函数会被标记为风险函数；跳转到陷阱页的主调函数也会被标记为风险函数.

综合来看，代码探测感知机制能够发现潜在的代码探测活动. 但是，直接阻止当前的可疑活动并不能实现强力的控制流保护. 具体原因如下：

第一，直接阻止当前活动可能会导致执行冲突. 因为当前活动只是可疑活动，其是否属于真正的代码探测需要进一步验证. 例如，调试器和探测者都可能读取代码，但读取活动本身并没有区别.

第二，直接阻止当前探测活动并不能阻止未知的恶意载荷在下一轮攻击中继续探测代码或劫持控制流. 因为，一些恶意载荷无需代码探测即可被用作探测工具或 gadgets，例如，对于已知的栈溢出漏洞，攻击者无需任何代码探测，只要设计好输入参数即可劫持返回控制流. 此外，在检测到当前活动之前，一些代码探测可能已经执行了多次. 例如，任意跳转在触发异常信号之前可能已经执行了多次，并且已经找到了包含 ret 的代码块. 无论是固有的恶意载荷还是已被泄露而未被发现的 gadgets，安全方法都无法通过阻止当前探测活动而消除它们的危险性. 攻击者使用这些代码块仍可进行后续的代码探测或控制流劫持.

考虑到上述问题，本文在检测到潜在的代码探测时并不阻止它们，而是继续追踪其控制流.

5.2　抽取风险代码

实际上，即使是在潜在的攻击场景中，也不是所有的代码都可被用作 gadgets. 一般来讲，攻击者会选择包含风险指令的代码块作为 gadgets. 基于该特征，MCE 从风险函数中选取包含风险指令的微小代码块作为保护目标(即风险代码)，进而减少不必要的控制流追踪与检测.

MCE 选择的风险代码至少包含 1 个入口点和 1 个出口点. 出口点是风险指令，而入口点则是与出口点距离最近的一条控制流转移指令的下一条指令. 当检测到代码探测时，MCE 会根据预先得到的汇编码从风险函数中筛选出所有的风险代码.

对于风险代码，MCE 需要捕捉传入其中的控制流，以检测其合法性. 同时，MCE 还要保证原有的风险代码能够被合法地调用，并确保代码功能的不变性. 为实现上述目标，MCE 建立了风险代码抽取机制，如图 4 所示.

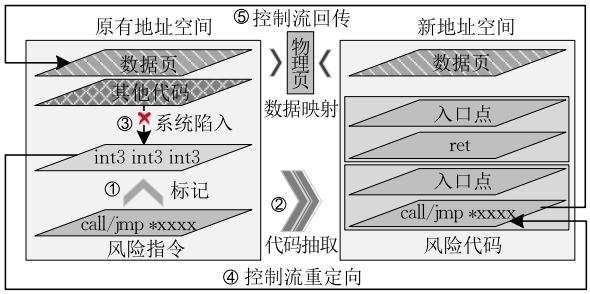


图 4 代码抽取机制

该机制本质上是将可被用作攻击载荷且已被探测的代码块从原有空间迁移到新的空间中. 通过代码标记,所有跳转到原有空间的控制流都能够被实时地捕捉到. 之后,被探测过的代码会成为 MCE 的追踪和检测目标,这能够及时发现非法控制流.

控制流可以通过直接跳转、间接跳转和返回跳转等多种方式被传送到风险代码中. 为捕捉跳转到风险代码的控制流,风险指令会被标记为 int3. 因此,风险指令的执行会因触发系统陷入而被捕捉到. 对于首次跳转到风险代码的合法控制流,MCE 会通过修改 VMCS 中的字段 guest rip 将其重定向到新空间中,从而实现对原有代码的调用. 需要说明的是,新空间中所有的数据虚拟页都会被映射到原有的数据页中. 所以,风险代码通过相对地址仍能访问到正确的数据.

在新空间中,风险代码中的控制流会通过风险指令将控制流传出. 所有风险指令的跳转目标都是指向原有空间的绝对地址. 因此,它们可以直接跳转到原有空间中. 代码抽取后的处理方法如图 5 所示. 通过标记和重写被探测代码,所有跳转到被探测代码的控制流都能被捕捉到. 同时,利用重构的跳转指令可以使控制流在原有空间和新空间之间自由切换,进而保证合法控制流的正常转移. 需要注意的是,新空间中的 call *xx 在执行时会将返回地址填入栈中. 所以,返回地址仍指向新空间,而返回控制流也会再次跳转到新空间中. 为将返回控制流重定向到原有地址空间,MCE 在新空间中的 call *xx 后添加指令 jmp address,从而将返回控制流传送到原有风险指令的下一条指令,进而确保原有代码的正常执行.

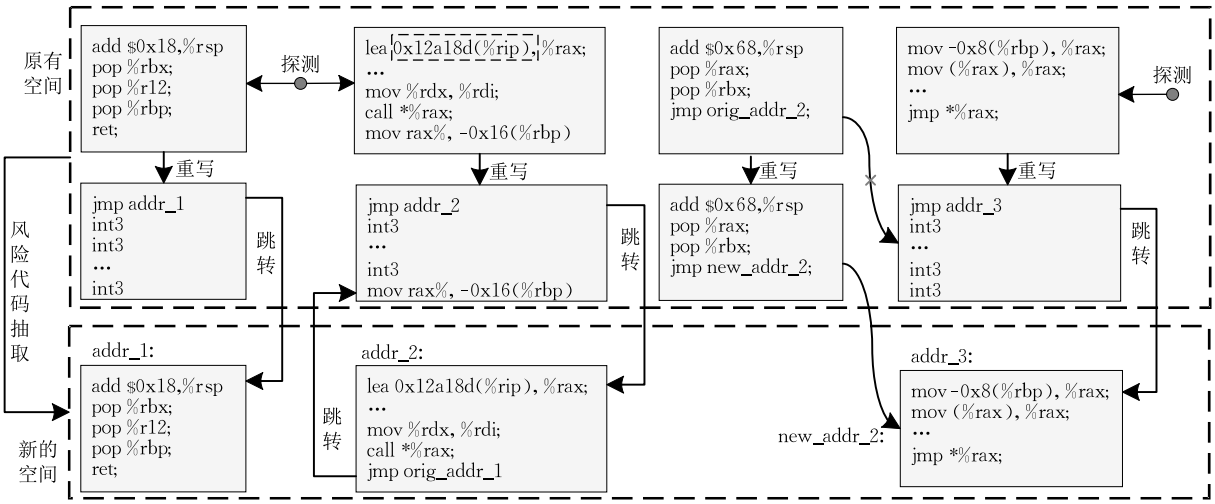


图 5 代码抽取后的处理方法

MCE 通过代码抽取机制可以仅追踪和分析与风险指令相关的执行过程,进而减少不必要的控制流追踪与检测. 之后,利用 LBR(Last Branch Record)寄存器组即可得到调用风险代码的主调指令,从而解析出闭源软件在此刻的代码逻辑. 该逻辑是判定闭源软件当前控制流合法性的关键.

5.3 保护控制流

MCE 需要检测所有跳转到风险代码的控制流以确保其合法性. 但是,闭源软件的内部逻辑是透明的,致使难以为其建立完整且精准的控制流图.

控制流攻击的本质就是破坏代码之间的逻辑关联性. 其最常用的方法是破坏那些能够决定控制流走向的控制数据,如返回地址. 异常的逻辑关联性会体现在代码逻辑和数据逻辑两个方面. 例如,被篡改的函数指针不再指向函数头部,而是指向风险代码,这破坏了数据逻辑;被篡改的返回地址可以使控制流不经过 PLT 直接跳转到共享库中,这破坏了代码逻辑. 逻辑关联性并不会因为源码的缺失而被掩盖,它们被包含在控制流的上下文中. MCE 建立控制流逆向追溯机制用于发现与风险代码相关的上下文,如图 6

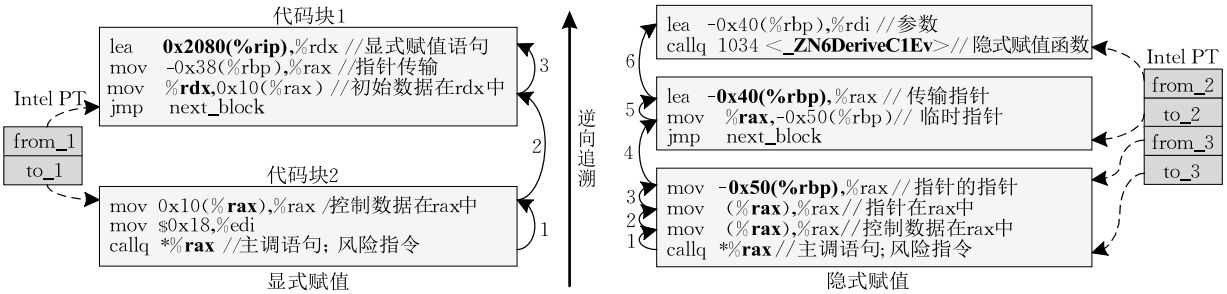


图 6　逆向追溯机制

所示. 该机制使用 Intel PT 技术记录控制流的历史路径. 当风险指令被调用时，通过解析已被执行的代码块即可得到风险代码的上下文，包括控制数据的赋值、传输和暂存等.

上下文中的控制数据可能是被显式赋值的，也可能是被隐式赋值的. 显式赋值是指原始控制数据显式地出现在代码段中，如 jmp 0x40080 中的 0x40080. 隐式赋值是指原始控制数据并不出现在代码段中，如存储在栈中的返回地址.

被显式赋值的控制数据是由赋值语句决定的，如图 6 中的“lea 0x2080(%rip)，%rdx”. 该类控制数据是已知且不可篡改的. 与之相关的控制流能够被劫持的原因是，控制数据会在传输过程中被暂存在可被篡改的内存中，如图 6 中的“mov %rdx，0x10(%rax)”. 被隐式赋值的控制数据是由内存拷贝函数/编译器/链接器/系统决定的，如图 6 中的“call 1034〈_ZN6DeviceCIEv〉”. 此类控制数据可能会被存储在可写内存中(如堆)，也可能被存储在不可写内存中(如 jump table). 对于不可写内存中的控制数据，攻击者可通过篡改数据引用以劫持控制流；对可写内存中的控制数据则直接使用溢出攻击即可实现控制流劫持. 例如，图 6 中的 mov %rax，－0x50(%rbp)将指针放在栈中，致使其存在被篡改的可能.

所有被劫持的控制流都会表现出异常的逻辑关联性. 为判定控制流的合法性，MCE 围绕逻辑关联性制定了上下文敏感的安全策略，包括基于代码逻辑的安全策略和基于数据逻辑的安全策略.

对于将控制流传入原有空间中风险代码的主调指令和将控制流传出新空间的风险指令，MCE 基于代码逻辑制定的安全策略如下：

(1) 如果主调指令是 call/jmp address，当前控制流是合法的. 捕捉到该指令之后，address 会被修改，使其指向新空间中的入口点. 因此，当该主调指令再次调用风险代码时，合法控制流会被直接传送到新空间中的风险代码中，进而可以避免不必要的系统陷入和控制流检测.

(2) 如果主调指令是 call *xx，控制流必须跳转到风险函数头部.

(3) 如果主调指令是 jmp *xx，它必须要跳转到风险代码中的操作码头部.

(4) 在同一段连续映射空间中，除了函数 longjmp 和 PLT 条目外，主调指令 jmp *xx 仅能跳转到当前函数的内部.

(5) 如果主调指令是 ret，必须存在与之成对的 call xx/*xx，且 ret 必须要在 xx 所指向的函数内. 如果风险指令是 ret，它的跳转目标的上一条指令也要是 call xx/*xx.

(6) 任何风险指令都不能不通过 PLT 而直接将控制流传送到共享库中.

(7) 若风险指令在被调用时被用作主调指令，即当前风险指令调用其他风险指令，其需要遵守(1)～(6).

对于跳转到原有空间中风险代码的主调指令来说，它所依赖的控制数据需要符合数据逻辑. MCE 围绕该控制数据制定的安全策略如下：

(8) 对于被显式赋值的控制数据，MCE 会根据原有空间与新空间之间的偏移量对其进行修正. 如果主调指令的跳转目标源于该数据，控制流会直接被传送到新空间，而不会触发系统陷入. 如果主调指令使用被篡改的临时数据将控制流重定向到被探测过的风险代码中，那么系统陷入将再次被触发. 此时的控制流就是非法的.

(9) 如果被隐式赋值的控制数据指向风险代码，且被存储在不可写内存中，如 Vtable 或 jump table，MCE 会根据原有空间与新空间之间的偏移量对其进行修正，使其指向新空间中风险代码的入口点. 之后，依赖该数据的主调指令会将控制流直接传送到新空间中，从而避免系统陷入与控制流检测. 如果该指令再次触发系统陷入，则说明对原始控制数据的引用已被篡改.

(10)如果被隐式赋值的控制数据指向风险代码且在可写内存中,它也会被修改为指向新空间的值.若该值不再发生变化,那么后续使用该数据的控制流将不再引发系统陷入.如果系统陷入再次发生,我们将再次利用安全策略判定当前控制流的合法性.

(11)对于同一主调指令或风险指令在不同时刻所使用的解引用指针,它们应当具有相同的指向性特征.例如,指向 jump table 的解引用指针在下一轮调用中依然指向不可写的 jump table,而不会指向可写的堆或栈;指向 Vtable 头部的指针在下一轮调用中依然指向 Vtable 头部,而不会指向其他位置.

经过安全检测后,合法控制流会被重定向到正确的位置.对于非法控制流,MCE 则在 host 模式下向当前进程中注入常规保护异常.此外,如果发现非法控制流,MCE 会通过 Intel PT 记录的路径逆向追溯已被执行的代码块,并利用安全策略判定历史控制流的合法性,直到得到合法的控制流为止.在该过程中,所有被非法调用的代码都会被认定为风险代码,并被迁移到新空间中.该方法有助于发现更多已被攻击者选中但未被发现的 gadgets.

6 评 估

本文所有实验均在 Dell T640 服务器中进行,其包含 2 颗银牌 4210R 的 CPU(主频 2.4GHz,单 CPU 含 10 核)、64 GB 内存、512 GB 固态硬盘和 4TB 机械硬盘.OS 为 64 位 Ubuntu20.04,内核 5.15.

6.1 安全效果评估

6.1.1 针对代码探测的安全评估

MCE 能够实时地检测到代码探测活动,并对被探测的代码进行控制流保护.为验证 MCE 对代码探测的感知能力,本文在不同的应用中部署攻击向量 1,即内存分配探测[7].在进程创建时,首先捕捉 main 函数的执行.之后,通过修改 VMCS 中的 guest rip 将控制流重定向到攻击模块.该模块遵循内存分配探测[7]中所采用的方法,利用内存分配函数(如 malloc)探测进程地址空间,探测结果如表 2 所示.

表 2 不同陷阱空间数量下 MCE 对攻击向量 1 的防御

APPs	0		100		1000		10 000		100 000	
	probes	traps	probes	traps	probes	traps	probes	traps	probes	traps
400perlbench	60	0	749	51	5297	413	89 956	5016	930 147	41 239
401bzip2	101	0	1125	63	8460	622	40 048	3127	745 938	50 127
403gcc	61	0	966	39	10 001	536	60 014	4386	427 583	30 458
410bwaves	44	0	681	46	7215	399	95 671	6248	1002 561	55 489
416gamess	69	0	1214	38	4679	406	70 783	4449	612 375	36 951
429mcf	77	0	559	62	6628	534	99 446	5146	778 564	40 020
433milc	59	0	781	53	4683	387	64 458	4554	399 842	29 998
434zeusmp	48	0	883	49	11 245	624	93 567	5328	789 951	60 429
400perlbench	60	0	749	51	5297	413	89 956	5016	930 147	41 239
401bzip2	101	0	1125	63	8460	622	40 048	3127	745 938	50 127

probes 表示在得到真实代码段时已进行的代码探测次数;traps 表示整个过程得到的陷阱空间数量.结果显示,随着陷阱数量的增多,向量 1 需要更多次的探测才能得到真实的代码段,同时也会探测到更多的陷阱空间.例如,当 gcc 中的陷阱数量达到 1000 时,攻击者需要 1 万次探测才能得到 1 个真实的代码段,而真实的代码段与 500 多个陷阱空间混淆在一起.从攻击者角度来看,他并知道哪一个是真实的代码段,哪一个是陷阱空间.在实际中,攻击者会结合内存扫描工具读取探测到的映射区域,以筛选出攻击载荷.在 MCE 的保护下,所有的陷阱空间都被映射到不可访问的陷阱页中,致使对陷阱空间的扫描活动会被捕捉并阻止.因此,MCE 能够抵御向量 1 的探测.

为测量 MCE 对向量 5(任意跳转)[11]的捕捉效果,本文通过修改 Redis 中函数的返回地址触发任意跳转,如图 7 所示.在测试过程中,每隔 10 s 钟 Redis

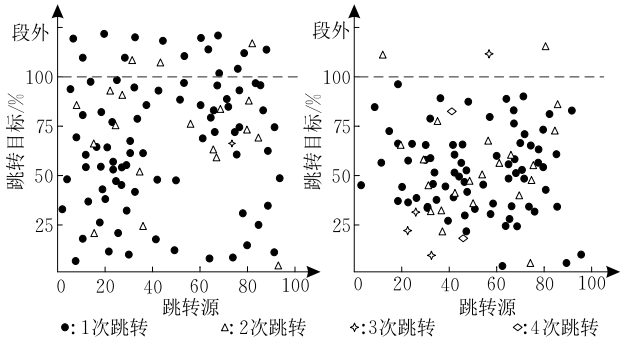


图 7 MCE 运行时 Redis 任意跳转的次数(100 次测量)(横坐标表示跳转的起始位置(相对代码段头部),纵坐标表示指令被捕捉到时的位置.左侧图代表代码段大小已知条件下的测试,右侧图代表代码段首尾地址已知条件下的测试)

的代码段就会被 EPT 设置为不可执行. 此时，当前函数栈帧中的返回地址会被修改，以实现控制流的任意跳转. 结果显示，任意跳转至多进行 4 次就会因跳转到陷阱空间或触发异常信号(SIGSEGV 或 SIGILL)而被 MCE 捕捉到.

在代码段大小已知的条件下，攻击者可估算出返回地址的低位地址范围. 例如，如果代码段的大小为 1 MB，那么攻击者只需要篡改返回地址的低 20 位即可降低控制流跳转到未映射区域的概率. 在代码段首尾地址都已知的条件下，攻击者可以直接确定正确的跳转范围，进而避免段外访问. 然而无论是在何种条件下，MCE 都能捕捉到向量 5. 实际上，向量 5 至多只能准确地控制第一次非法跳转的目的地，如被栈溢出篡改的返回地址. 而首次非法跳转后被执行的代码是不可预测的，导致后续的跳转目标是不可控的. 因此，非法指令与段外访问也是不可避免的. 例如，如果返回控制流被重定向到操作数 c3(ret)，且此时栈底数据指向未映射区，那么信号 SIGSEGV 就会被触发. 所以，在真实的攻击场景中，向量 5 的攻击目标只能是那些支持崩溃后自动重启的应用(如浏览器)，从而可实现持续性探测. 在 MCE 的保护下，超过 70% 的任意跳转在首次跳转之后和第二次跳转之前就会因触发异常信号或访问陷阱空间而被捕捉到. 即使攻击者足够幸运，他也会在第 5 次跳转之前被 MCE 发现.

向量 3(任意读)[9]和向量 4(数据泄露)[10]本质上都属于代码读取. 为模拟基于任意读的代码探测，我们使用一个可加载内核模块(Loadable Kernel Module，LKM)读取 Apache 地址空间中[task_struct→code_start，task_struct→code_end]范围内的代码. 结果发现，每一次读代码活动都会触发 EPT 异常，并被 MCE 捕捉到.

对于向量 6(侧信道泄露)[12]，攻击者在探测虚拟地址的第 15 位至第 20 位、第 24 位至第 29 位、第 33 位至第 38 位和第 42 位至第 47 位时，MCE 并不能发现异常. 然而，攻击者在探测第 30 位至第 32 位时会因访问陷阱空间而被捕捉到. 对于向量 2(进程克隆)[8]，被克隆的子进程会继承父进程不可读的代码段和所有的陷阱空间. 因此，对子进程的探测虽然不会触发父进程崩溃，但仍能够被 MCE 捕捉到.

综合来看，MCE 的安全机制能够对不同的代码探测做出响应，如表 3 所示. 由于不依赖于源码，MCE 适用于防护未知来源的闭源软件. 尤其是对于不具备安全知识的用户来说，MCE 能够对其自由安装的应用实现运行时保护，并不需要用户专门预留安全检测窗口.

表 3　MCE 对不同攻击向量的响应机制

攻击向量	安全机制		
	权限感知	空间感知	信号感知
向量 1	√	√	—
向量 2	√	√	√
向量 3	√	—	—
向量 4	√	—	—
向量 5	√	—	√
向量 6	√	√	—

然而，如果攻击者不对目标进程进行代码探测即可展开攻击，那么 MCE 将会被绕过. 针对该问题，MCE 在实际应用中会与 ASLR(尤其是细粒度的 ASLR)配合部署. 因为在 ASLR 的环境中，代码探测已逐渐成为部署 CRAs 的首要步骤[13]，这为触发 MCE 提供了必要条件. 此外，如果攻击者采用的探测技术能够规避 MCE 的检测，那么 MCE 也会被绕过. 实际上，除了表 3 中提到的 6 种攻击向量之外，其它需要读取代码、访问特定空间或触发异常信号的位置探测技术都能被 MCE 感知到. 但是，对于采用新技术的代码探测，我们需要分析其攻击原理并建立新的检测模型，否则 MCE 仍存在被绕过的可能.

6.1.2　针对非法控制流的安全评估

被探测的代码会被标记为风险代码，之后跳转到该代码的控制流需要经过安全检测. 为验证安全策略的有效性，本文使用 ropper① 扫描不同应用中的 gadgets，并分析 MCE 对 gadget 链的检测效果，如表 4 所示. 结果显示，ROP 攻击和 JOP 攻击分别在连接第 3 个和第 4 个 gadget 时就会被 100% 检测到.

在现有的攻击形式当中，最常见的风险代码形式有三类，分别是“pop xx，ret”、“mov xx register/pointer、call *register/*pointer”和“mov xx register/pointer、jmp *register/*pointer”. 包含这些风险指令的代码块可分别衍生出 ROP gadget 和 JOP gadget. 攻击者通过溢出漏洞篡改返回地址或函数指针即可改变风险指令的跳转目标. 然而，攻击者在连接不同的 gadgets 的过程中会违背 MCE 提出的安全策略. 以 perlbench 中的 ROP 和 JOP 为例，它们在连接 gadgets 时违背了代码逻辑如图 8 所示.

图 8　控制流跳转违背的安全策略

① https://github.com/sashs/Ropper

表 4 MCE 对 CRAs 的防御(chain-x:可由 x 个 gadgets 构成的 gadget chain)

APPs	size/KB	rop gadgets	chain-2		chain-3		jop gadgets	chain-2		chain-3		chain-4	
			数量	检测率/%	数量	检测率/%		数量	检测率/%	数量	检测率/%	数量	检测率/%
Apache	696	6788	2.3E+07	97.9	5.2E+10	100	1844	1.7E+06	94.3	1.0E+09	99.1	4.8E+11	100
Sqlite	36	216	2.3E+04	99.4	1.7E+06	100	58	1.7E+03	88.7	3.1E+04	97.8	4.2E+05	100
redis-server	14704	26328	3.5E+08	99.6	3.0E+12	100	7656	2.9E+07	95.2	7.5E+10	100.0	1.4E+14	100
redis-cli	7468	6990	2.4E+07	96.7	5.7E+10	100	3306	5.5E+06	93.8	6.0E+09	96.4	5.0E+12	100
ngnix	4392	7788	3.0E+07	99.9	7.9E+10	100	1968	1.9E+06	95.7	1.3E+09	98.7	6.2E+11	100
libc-2.31.so	1984	31562	5.0E+08	95.4	5.2E+12	100	8339	3.5E+07	96.3	9.7E+10	100.0	2.0E+14	100
400perlbench	3876	30889	4.8E+08	100.0	4.9E+12	100	3109	4.8E+06	89.9	5.0E+09	95.7	3.9E+12	100
401bzip2	196	1535	1.2E+06	99.9	6.0E+08	100	547	1.5E+05	86.4	2.7E+07	96.3	3.7E+09	100
403gcc	12108	77290	3.0E+09	98.7	7.7E+13	100	10260	5.3E+07	90.2	1.8E+11	98.3	4.6E+14	100
410bwaves	100	475	1.1E+05	99.5	1.8E+07	100	49	1.2E+03	87.6	1.8E+04	98.1	2.1E+05	100
416gamess	3204	9491	4.5E+07	99.5	1.4E+11	100	968	4.7E+05	92.6	1.5E+08	99.7	3.6E+10	100
429mcf	92	714	2.5E+05	99.8	6.0E+07	100	50	1.2E+03	90.5	2.0E+04	99.9	2.3E+05	100
433milc	648	4168	8.7E+06	99.6	1.2E+10	100	147	1.1E+04	86.5	5.2E+05	100.0	1.9E+07	100
434zeusmp	832	3819	7.3E+06	97.8	9.3E+09	100	371	6.9E+04	91.7	8.4E+06	100.0	7.8E+08	100
435gromacs	3192	18105	1.6E+08	99.8	9.9E+11	100	1627	1.3E+06	95.6	7.2E+08	100.0	2.9E+11	100
436cactus	3080	19008	1.8E+08	99.6	1.1E+12	100	2310	2.7E+06	96.7	2.1E+09	99.9	1.2E+12	100
437leslie3d	264	1169	6.8E+05	99.1	2.7E+08	100	72	2.6E+03	93.8	6.0E+04	100.0	1.0E+06	100
444namd	768	19664	1.9E+08	99.3	1.3E+12	100	397	7.9E+04	89.4	1.0E+07	98.8	1.0E+09	100
447dealⅡ	6792	20089	2.0E+08	96.9	1.4E+12	100	8209	3.4E+07	92.5	9.2E+10	97.1	1.9E+14	100
450soplex	272	1580	1.2E+06	98.7	6.6E+08	100	733	2.7E+05	94.8	6.5E+07	100.0	1.2E+10	100
453povray	3696	26354	3.5E+08	99.2	3.1E+12	100	5082	1.3E+07	88.7	2.2E+10	96.2	2.8E+13	100
454calculix	5836	26919	3.6E+08	95.4	3.3E+12	100	3002	4.5E+06	95.6	4.5E+09	99.1	3.4E+12	100
456hmmer	1080	8777	3.9E+07	99.4	1.1E+11	100	552	1.5E+05	97.8	2.8E+07	100.0	3.8E+09	100
458sjeng	348	1981	2.0E+06	100.0	1.3E+09	100	149	1.1E+04	90.3	5.4E+05	100.0	2.0E+07	100

代码探测发生后,被探测函数中的风险指令会被抽取出来.之后,所有跳转到其中的控制流都要受到检测.在图 8 所示的攻击场景中,ROP 利用栈溢出漏洞篡改返回地址,使得 ret 能够将控制流传送到 pop %r14.由于 pop %r14 的上一条指令并不是 call xx,这违背了安全策略(5).而图中的 JOP 通过篡改 rip+0x12a18d 的内存使控制流跳转到一个函数内部,这违背了安全策略(9).

MCE 本质上是一种上下文敏感的控制流检测方法.它通过捕捉并分析运行时上下文能够准确地获取到控制流转移指令的执行逻辑,因而可摆脱对源码的依赖性.但是,由于动态上下文存在丢失的可能,MCE 无法 100%检测到首次跳转的非法指令.幸运的是,在已知的 CRAs 当中,攻击者均需要多个攻击载荷才能完成特定的任务.所以,MCE 对现有的 CRAs 及其变种仍具有良好的防御效果.

MCE 仅将少量的风险代码作为追踪和检测目标,进而可降低性能开销.被 MCE 选中的风险代码包括 ICT 指令和返回指令,它们的首地址均指向指令的操作码.然而新型的控制流攻击不仅能利用常规的风险代码,还能利用代码的非对齐特征将特定形式的操作数用作非法代码.例如,如果控制流跳转到指令"pushq 0xc3"(机器码 68 c3 00 00 00)头部的下一个字节,那么原有的操作数 c3 就会被转化为风险代码 ret.对于此类被非法转化而来的风险代码,MCE 是不具有防御效果的.实际上,MCE 可通过扩大风险代码的认定范围解决此类问题.例如,所有被探测的代码所在的函数都被认定为风险代码,那么所有的操作数就都会被纳入追踪和检测范围,进而即可发现跳入其中的非法控制流.但是,此类方法必然会扩大控制流的追踪范围,从而引发更大的开销.如何解决开销问题是此类方法面临的重要挑战.

6.1.3 MCE 在真实攻击案例中的表现

为观察 MCE 在真实攻击案例中的表现我们基于漏洞 CVE-2013-2028,在 Nginx-1.4.0 中实现了"Blind Rop"(BROP)攻击[11].同时,我们观察 MCE 对该攻击的响应状况.

BROP 是一种实用的攻击技术,它是代码探测技术和代码复用攻击技术的结合体.BROP 既不依赖源码,也不需要解析静态可执行文件.它通过远程代码探测即可得到可被用作 gadgets 的攻击载荷.

在攻击实验中,我们首先破解 Nginx 栈上的 canary.通过栈溢出方式,我们逐字节地覆盖 canary 值.错误的数值将导致进程崩溃并重启.至多 256 次尝试之后,即可得到正确的 canary 值.由于 canary 破解过程不涉及任何代码探测活动,所以 MCE 并不能发现当前的异常活动.

之后，我们寻找“stop gadgets”的代码块和“trap gadgets”. 前者可直接或间接地通知攻击者该段代码已执行，如具备网络发送功能的代码；后者则会引发进程崩溃. 通过栈溢出漏洞，我们即可在栈上部署上述两种 gadgets，进而筛选出符合特定形式的 gadgets（如“pop rsi；pop rdi；ret”）. 实验结果显示，攻击者在执行“stop gadgets”时，信号 SIGSEGV 和 SIGILL 能够被 MCE 捕捉到. 之后，触发信号的代码，以及进行任意跳转的 ret 所在的代码块都会被检测和捕捉. 使用被篡改地址的 ret 会被从原有空间中被抽取到新空间中. 之后所有流向该代码块的控制流都会被检测. 我们观察到，非法控制流会违背基于代码逻辑的安全策略(5). 因此，攻击者将无法进行后续的代码探测活动.

需要说明的是，在攻击过程中信号 SIGSEGV（段错误）和信号 SIGILL（非法指令）难以避免. 在我们的实验中，攻击者至少要经过 356 次的探测，才能得到符合“pop rbx；pop rbp；pop r12；pop r13；pop r14；pop r15；ret”形式的 gadgets. 在这期间，信号 SIGSEGV 和信号 SIGILL 会被触发 122 次. 所以，此类探测几乎难以逃脱被捕捉和检测.

退一步讲，即使攻击者能够避免代码探测过程被 MCE 发现，他在后续攻击环节中使用函数 write 转储二进制码时也会被捕捉到. 因为，代码页是不可读的. 被读取的代码中包含的风险代码会被提取到新空间，所有对它们的调用都会被检测.

然而，MCE 是存在误判和漏判可能的. 开发者在调试运行中的应用时需要读取内存中的二进制码. 此时，合法的代码读取活动会被判定为代码探测，进而引发误判现象. 之后，所有被读取的代码中包含的风险代码都会被提取出来. 虽然代码仍可被正常调用，但这会引发不必要的性能开销.

对于“{if (a) call func-1；else call func-2；}”，攻击者可通过篡改栈上的变量 a 来劫持控制流. 由于该代码块中并不包含风险代码，即使这样的代码已被探测，它也不会被 MCE 提取出来. 因此，这样的攻击会被忽略，进而引发攻击漏判. 幸运的是，此类攻击方法所劫持的控制流只能被传送到固定的位置，而不是 CRAs 要求的任意位置. 所以，与之相关的函数一般不会被 CRAs 用作攻击载荷.

6.2　性能评估

SpecCPU2006 被用来测量 MCE 对 CPU 引入的开销. 在一般场景中，良性应用并不会探测进程代码. 因此，无法观察代码探测活动发生后 MCE 对 SpecCPU2006 的影响. 为解决该问题，利用一个 LKM 去扫描目标进程的二进制代码，以触发 MCE 的安全响应. 被探测的进程代码量从 0 到 100% 递增，并且所有结果都以原生 OS 中的 CPU 表现为基准进行了归一化. 所有结果都是在探测过程中测得的运行时开销，如图 9 所示. 结果显示，在一般场景中（即无代码探测发生的情况下），MCE 对 CPU 引入的平均开销约为 2%. 当仅有 5% 的代码被探测时，平均开销为 4.5%；当探测比例达到 100% 时，运行时开销将达到 55%.

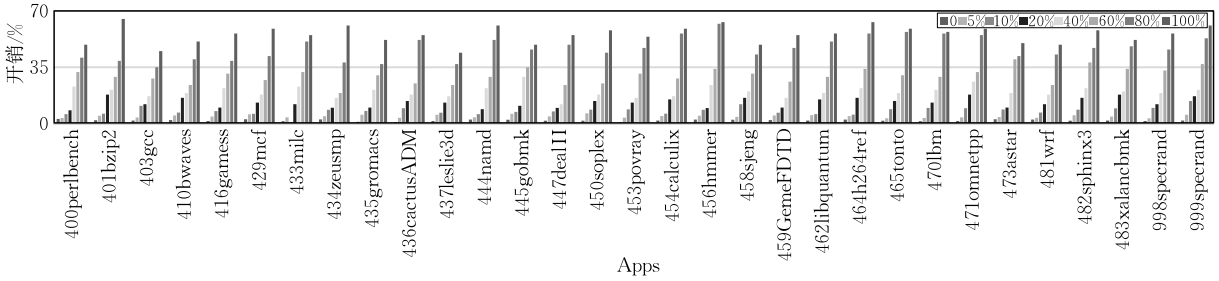


图 9　不同探测比例下 MCE 引入的运行时 CPU 开销

代码探测发生后，MCE 需要将被探测的代码迁移到新的空间中. 迁移过程中，受保护的进程会被挂起，进而影响其执行速度. 之后，所有首次跳转到被探测代码中的控制流都需要被检测和分析. 被探测的代码越多，需要追踪的控制流也就越多. 因此，MCE 对受保护对象引入的开销会随着探测比例的增大而逐渐增大.

通过实验我们发现，MCE 对于少量的代码探测并不会引入明显的开销，而对需要大量探测代码的场景则会引入不可忽视的开销. 在目前已知的代码探测场景中，只有基于代码读取的探测技术会在大量读取代码的情况下严重影响 MCE 的运行效率. 其他探测技术，如任意跳转，在进行大量探测之前就会被 MCE 捕捉并阻止. 我们之所以会保持代码的可读性是考虑到特殊用户（如系统调试员）存在代码读取需求. 如果不考虑此类用户，我们完全可以关闭进程代码的读权限. 在该场景下，攻击者并不能连续性读取大量的代码，他在首次读取代码时就会被阻止. 因此，MCE 无需追踪大量的控制流，进而可降低该场景下的开销.

Lmbench 被用来测量 MCE 在一般场景中引入

的系统延时和本地通信带宽损耗，所有结果都以原生 OS 为基准进行了归一化，如图 10 所示. 各项延时测试的平均值约为 2.4%. 其中，系统延时平均增加 2.2%，本地通信带宽平均降低 3.6%.

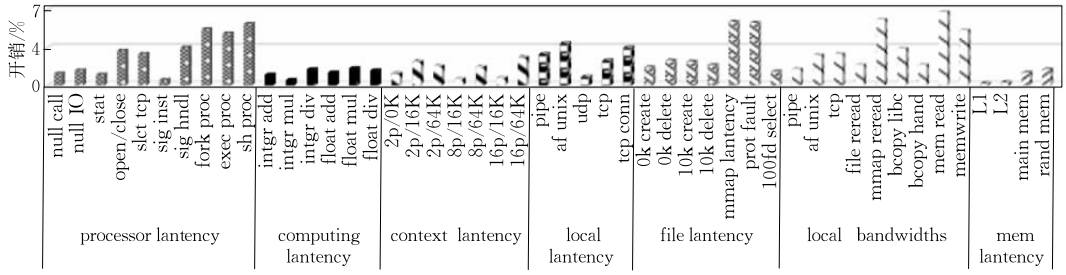


图 10 MCE 引入的系统延时

IOMeter 被用来测量 MCE 在一般场景中引入的 I/O 开销，所有结果都以原生 OS 中的 I/O 表现为基准进行了归一化，如图 11 所示. 结果显示，在一般场景中 MCE 对 I/O 的影响范围为 1.5%～8.1%.

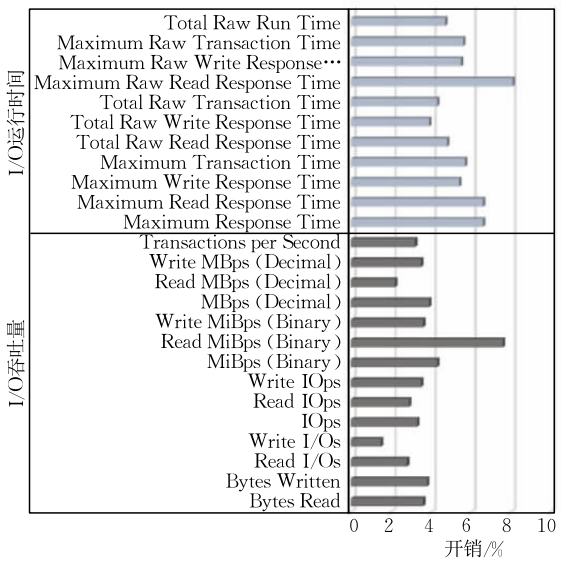


图 11 MCE 引入的 I/O 开销

Lmbench 和 IOMeter 的测试说明，MCE 在一般场景中也会引入性能开销. 其根本原因是，MCE 在一般场景中无法避免系统陷入事件. 系统陷入事件可以分为无条件陷入和条件陷入. 在 host 中，指令 cpuid、gettsec、invd、xsetbv 以及除 vmfunc 之外的所有 VMX 指令都会引发无条件陷入. 测试发现，这些指令在不同应用程序中的执行频率差异较大，这是引发不同开销的主要因素. 系统陷入后，当前进程会被挂起. MCE 会在处理完特定的陷入事件后才会恢复进程执行，这降低了进程的执行速度.

实际上，无论是代码探测还是控制流劫持，它们仅在极少数的特定场景中才会出现. 这就要求安全方法在一般场景中不能引入过高的开销，否则将会对受保护进程的全生命周期产生不可忽视的影响. 而 MCE 在一般场景中引入的开销并不明显，因此具备较高的实用性.

除无条件陷入事件之外，其他陷入事件均是由 MCE 设置的特定事件触发的，包括代码探测、控制流追踪和控制流检测. 控制流追踪与检测是引发性能开销的主要因素. 但是，这种开销并不是一成不变的. 它会随着进程的执行而逐渐降低，这是由 MCE 的部署原理决定的. 为验证这一结论，使用 web 应用测试 MCE 在代码探测发生后的一段时间内引入的开销变化，如图 12 所示.

在实验过程中，web 应用不间断地传输数据. 初始时，扫描工具仅读取函数的一个字节以触发控制流追踪与检测. 结果显示，MCE 引发的运行时开销会随着代码探测比例的增高而增高. 之后，web 应用继续传输数据. 在探测后的第 1、10 和 30 min 时，再次检测 MCE 对 web 应用引入的开销. 结果显示，随着时间推移，MCE 引入的开销逐渐减少. 因为随着进程的不断执行，越来越多的合法指令和显式赋值语句被发现. 与之相关的控制流会被直接重定向到新空间中，避免了不必要的控制流追踪与检测. 因此，MCE 引入的开销会逐渐降低.

由于 MCE 对受保护对象引入的开销会随着进程的执行而逐渐降低，所以更适合被用于保护常驻内存类的应用，如网络服务器和内存数据库. 相反地，对于在操作系统中运行时间很短的应用而言，MCE 的实用性会有所降低. 因为代码探测发生后，MCE 在发现足够多的合法控制流之前，此类应用就已经结束了，导致 MCE 来不及降低其性能开销.

为测量新 buddy system 对内存系统的影响，本节结合压力工具 stress 和内存测试工具 mbw① 测试不同内存压力下的内存系统开销，如图 13 所示.

① https://github.com/raas/mbw

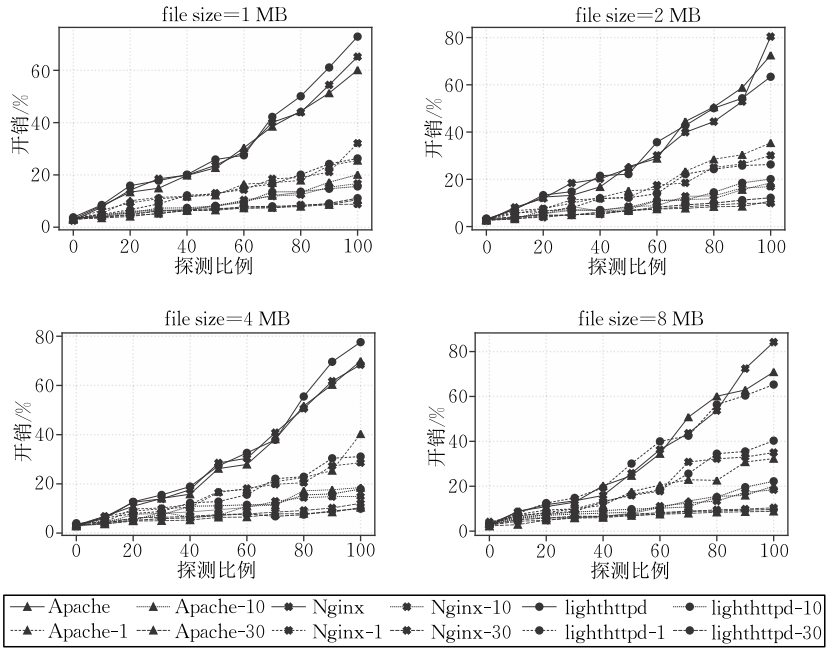
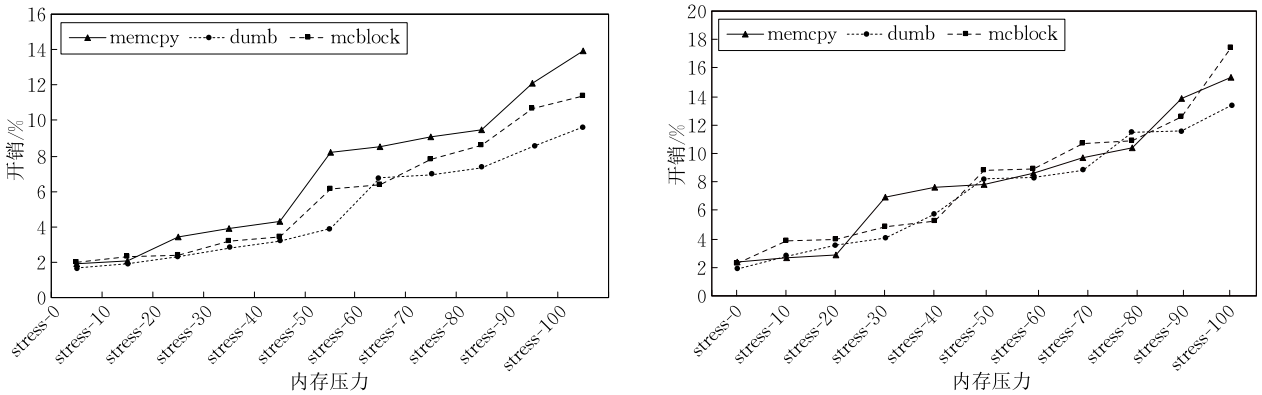


图 12　MCE 引入的性能损耗随时间的变化趋势(Web-name-x：x 分钟后对该应用进行的性能测试)



(a) 内存块大小为256 MB　　(b)内存块大小为10 GB

图 13　Mbw 测试的内存开销

结果显示，当内存压力超过 80%且大量分配内存时，内存系统将会受到明显的影响. 因为新的 buddy system 会将原有的物理内存分为两部分，这导致在内存紧张且需要分配大块内存时，内存系统需要频繁地回收或换出已被分配的物理页. 此外，MCE 还要监视物理页的换入换出过程，以防止被换入的代码页被放置在可读物理页中，这也会对内存分配系统产生影响. 相比之下，在内存低压力场景中，MCE 对内存系统的影响小于 5%.

需要说明的是，MCE 并不会占据过多的内存. 虽然 MCE 要为目标对象建立大量的感应段，但这些感应段所使用的寻址页表会被映射到同一组物理页中. 因此，多段地址空间并不会占据大量的内存. 此外，代码探测发生后，MCE 只会将风险代码迁移到新空间中. 风险代码并不会占据过多的内存. 因为过大的代码块会影响攻击者预设的内存或寄存器，进而导致攻击失败. 一般来讲，风险代码的大小只有十几到几十个字节. 例如，在 gcc 中，能被 JOP 利用的风险代码的常见形式是“mov xx register，call *register”，由此衍生出的 gadgets 均小于 50 字节. 实际上，MCE 最大的内存开销来源于 EPT. 为寻址 64 GB 物理内存，两套完整的 EPT(EPT_1 和 EPT_2)共占据约 256 MB 内存.

6.3　与已有方法的比较

为与现有的方法进行综合比较，本文对包括 MCE 在内的 12 种方法进行了综合评估，如图 14 所示. 评估参数中除 RP(文中报告的性能)之外，其余 3 个参数的定义如式(1)～(3)，它们都是基于安全方法的部署原理进行定性分析的结果.

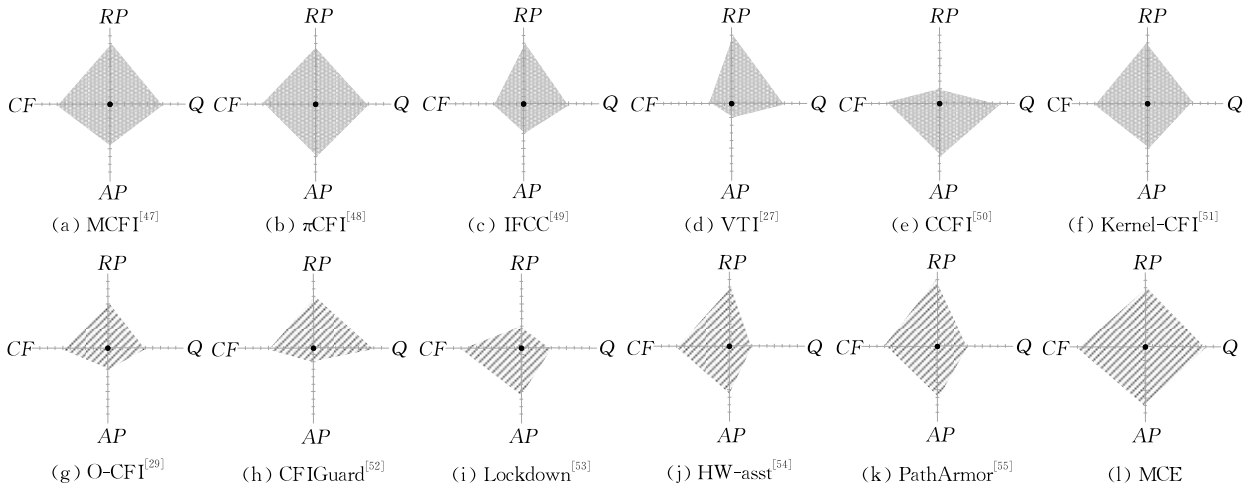


图 14 与现有方法的对比(*RP*:论文中报告的性能;*CF*:可追踪的控制流类型;*AP*:分析精度;*Q*:保护质量.第 1 行的方法需要分析源码;第 2 行中的方法只需要分析二进制码.安全方法的综合表现与图形面积呈正相关)

$$CF=\sum_{k=1}^{N}P_{\text{app}}^{k}\times I_{\text{app}}^{k} \tag{1}$$

$$AP=\sum_{m=1}^{M}P_{\text{attack}}^{m}\times\left(\frac{F_{I_{\text{attack}}^{m}}}{F_{I_{\text{all}}^{m}}}+\frac{B_{I_{\text{attack}}^{m}}}{B_{I_{\text{all}}^{m}}}\right)\Big/2 \tag{2}$$

$$Q=\sum_{n=1}^{tar\_num}P_{\text{mal}}^{n}\times\frac{1}{L_n} \tag{3}$$

CF 是指安全方法能够追踪到的风险代码所引发的控制流的种类. P_{app}^{k} 表示安全方法能够跟踪到第 k 种风险代码的概率. I_{app}^{k} 表示第 k 种风险代码在真实攻击中所占的百分比. 理想的安全方法能够追踪到每一种风险代码,因此其每一个 P_{app}^{k} 都是 1. 相反地,如果一种安全方法无法追踪某一种风险代码,而此类代码在真实中出现的概率又很高,那么它的 *CF* 将受到严重影响. 为简化评估过程,我们将 ROP 所使用的返回指令占所有风险代码的比例设定为 0.5,即 I_{app}^{1} 为 0.5;JOP 所使用的间接跳转指令(如 jmp *pointer)占所有风险代码的比例也为 0.5,即 $\sum_{k=2}^{N}I_{\text{app}}^{k}=0.5$. 例如,VTI[27] 无法追踪 ROP 攻击所使用的 ret($P_{\text{app}}^{0}=0$),致使其 *CF* 必然小于 0.5;相比之下,MCE 会被探测的所有风险代码都迁移到新空间中,因而可捕捉到所有流向风险代码的控制流,因而具有更好的 *CF* 表现.

AP 是指安全方法对风险代码引发的控制流的分析精度. P_{attack}^{m} 表示第 m 种攻击出现且被检测到的概率. 本文将攻击场景分为三种,分别是针对有源应用代码的攻击,针对库代码的攻击和针对闭源应用的攻击. P_{attack}^{m} 的大小取决于安全方法的部署原则. 例如,对于 MCE 来说,三种攻击场景出现并被检测到的概率均为 1/3. $F_{I_{\text{attack}}^{m}}$ 和 $B_{I_{\text{attack}}^{m}}$ 分别表示在第 m 种攻击场景中被追踪的非法前向跳转指令(如 call *)的数量和非法后向跳转指令(如 ret)的数量. $F_{I_{\text{all}}^{m}}$ 和 $B_{I_{\text{all}}^{m}}$ 分别表示在第 m 类攻击场景中所有前向跳转指令的数量和所有后向跳转指令的数量. 理想的安全方法能够在攻击出现时,准确地追踪每一次非法控制流转移活动. 相反地,对攻击场景的漏判和对非法指令识别精度的不足都将直接影响到 *AP*. 例如,IFCC[49] 无法被用于无源码场景中(即 $P_{\text{attack}}^{3}=0$),也无法追踪跳转到库中的控制流(即 $P_{\text{attack}}^{2}=0$),而 O-CFI[29] 会追踪所有的控制流转移活动(包括大量的合法控制流),导致其引入过大的 $F_{I_{\text{all}}^{m}}$ 和 $B_{I_{\text{all}}^{m}}$,因此上述两种方法的 *AP* 表现都会受到严重影响. 与现有方法相比,MCE 能够通过代码事件筛选出可能的攻击载荷并将其作为受保护对象,进而可减少不必要的控制流追踪,因而具有更好的 *AP*.

Q 是指安全方法对控制流的保护质量. 影响 *Q* 的因素主要包括三个方面:攻击原理、防御原理和攻击对象的特征. 本文围绕这三个方面对 *Q* 进行了定性分析. *tar_num* 表示风险指令的总数,它是由攻击对象决定的. L_n 表示第 n 条风险指令被允许跳转到的目标的数量. 在实际中,每一条控制流转移指令在被调用时有且仅有一个目标是合法的. 即理想的 L_n 是 1. 依赖控制流图的方法往往会因为模糊的指令边界导致 *Q* 受到影响. 例如,πCFI[48] 为控制流转移指令构建的跳转目标会随着程序执行而逐渐扩增(即 L_n 从 1 开始逐渐增大),进而导致 *Q* 逐渐减小. P_{mal}^{n} 表示第 n 条风险指令被恶意使用的概率. 它是由攻击原理和具体的攻击场景决定的. P_{mal}^{n} 越大,第 n 条指令的安全风险就越大. 例如,BROP[11] 攻击

中被探测的代码“pop rsi;pop rdi;ret”对应的 P^n_{mal} 为 1. 安全方法对高风险指令的保护力度越大就越能提高 Q，这是符合防御原理的. 通过分析风险代码的代码逻辑和数据逻辑，MCE 能够在跳转指令执行时缩小合法目标的范围，甚至确定出唯一的合法目标，并通过代码探测事件筛选出风险代码. 因此，与同类不依赖源码的方法相比，MCE 具有较好的 Q.

综合来看，MCE 的整体表现较好. 首先，MCE 摆脱了对源码的依赖性，这使其可保护闭源对象. 再者，它可将所有用户代码页都预先设定为不可读，避免了频繁的代码页分配/访问追踪和耗时的读权限动态调整. 最后，MCE 通过感知代码探测活动来发现潜在的攻击场景，并仅将该场景中已被探测代码（而不是整个代码段）作为追踪目标，进而具备了场景针对性和目标针对性. 该设计可减少不必要的控制流追踪与检测，从而降低性能开销.

7　结　论

针对闭源软件中的控制流安全问题，本文提出一种基于风险代码抽取的控制流保护方法 MCE. 该方法通过建立代码探测感知机制，实时地检测潜在的代码探测活动. 之后，具有探测风险的代码会被标记. 同时，风险代码会被抽取并迁移到新的地址空间中. 接下来，所有跳转到风险代码中的控制流都会被捕捉到. 在目标对象运行过程中，MCE 通过分析二进制码的代码逻辑和数据逻辑建立了上下文敏感的安全策略. 所有被捕捉到的控制流都要经过安全策略的验证. 此外，符合安全策略的控制流能够避免被再次追踪和检测，进而降低性能开销. 实验和分析表明，MCE 对 CRAs 具有较好的保护效果，并仅对 CPU 引入约 2%的开销.

但是，MCE 仍存在一些局限性. 第一，MCE 仅能部署到开源的 Linux 中. 第二，MCE 仅支持 x86 架构的处理器，这导致其无法部署在主流的嵌入式系统（运行在单片机、DSP 或 ARM 中）中. 第三，MCE 仅对应用代码有效，对内核代码和共享库代码无效. 第四，由于 MCE 是基于代码探测展开防御工作的，若攻击者不经过代码探测即可部署 CRAs，MCE 将会被绕过. 幸运的是，随着 ASLR（尤其是细粒度的 ASLR）的普及，已知的 CRAs 均无法基于静态代码内容直接展开攻击. 我们认为，代码探测已逐渐成为部署 CRAs 的必要环节. 基于代码探测事件触发的防御方法因具备更强的场景针对性和目标针对性会成为未来安全方法发展的主要方向. 然而，如何准确识别各类代码探测行为是此类方法面临的主要挑战. 第五，MCE 仅能部署到支持硬件辅助虚拟化的处理器环境中；若在云环境下，则需开启嵌套虚拟化功能，否则将无法部署. 围绕着上述局限性，我们在未来会继续改进 MCE，以使其能够部署到云环境中和 ARM 处理器上，同时建立更具针对性的安全策略，以提高其应用范围和安全能力.

参　考　文　献

[1] Li Y G, Lin G Y, Chung Y C, et al. MagBox: Keep the risk functions running safely in a magic box. Future Generation Computer Systems, 2023, 140(3): 282-298

[2] Fu A, Ding W, Kuang B, et al. FH-CFI: Fine-grained hardware-assisted control flow integrity for ARM-based IoT devices. Computers & Security, 2022, 116(5): 102666

[3] Wang W, Hu G, Xu X, et al. CRAlert: Hardware-assisted code reuse attack detection. IEEE Transactions on Circuits and Systems II: Express Briefs, 2021, 69(3): 1607-1611

[4] Fu J, Lin Y, Zhang X. Code reuse attack mitigation based on function randomization without symbol table//Proceedings of the IEEE Trustcom/BigDataSE/ISPA. Tianjin, China, 2016: 394-401

[5] Zhang Gui-Min, Li Qing-Bao, Zhang Ping, Cheng San-Jun. Defending code reuse attacks based on running characteristics monitoring. Journal of Software, 2019, 30(11): 3518-3534 (in Chinese)
(张贵民，李清宝，张平，程三军. 基于运行特征监控的代码复用攻击防御. 软件学报，2019，30(11)：3518-3534)

[6] Guo Y, Chen L, Shi G. Function-oriented programming: A new class of code reuse attack in C applications//Proceedings of the IEEE Conference on Communications and Network Security (CNS). Beijing, China, 2018: 1-9

[7] Oikonomopoulos A, Athanasopoulos E, Bos H, et al. Poking holes in information hiding//Proceedings of the USENIX Security Symposium. Austin, USA, 2016: 121-138

[8] Lu K, Lee W, Nürnberger S, et al. How to make ASLR win the clone wars: Runtime re-randomization//Proceedings of the Network and Distributed System Security (NDSS) Symposium. San Diego, USA, 2016

[9] Snow K Z, Monrose F, Davi L, et al. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization//Proceedings of the IEEE Symposium on Security and Privacy. Berkeley, USA, 2013: 574-588

[10] Hu H, Shinde S, Adrian S, et al. Data-oriented programming: On the expressiveness of non-control data attacks//Proceedings of the IEEE Symposium on Security and Privacy (SP). San Jose, USA, 2016: 969-986

[11] Bittau A, Belay A, Mashtizadeh A, et al. Hacking blind//Proceedings of the IEEE Symposium on Security and Privacy. Berkeley, USA, 2014: 227-242

[12] Gras B, Razavi K, Bosman E, et al. ASLR on the line: Practical cache attacks on the MMU//Proceedings of the Network and Distributed System Security (NDSS) Symposium. San Diego, USA, 2017, 17: 26

[13] Li Y G, Chung Y C, Xing J, et al. MProbe: Make the code probing meaningless//Proceedings of the 38th Annual Computer Security Applications Conference. Austin, USA, 2022: 214-226

[14] Hawkins W, Nguyen-Tuong A, Hiser J D, et al. Mixr: Flexible runtime rerandomization for binaries//Proceedings of the 2017 Workshop on Moving Target Defense. Dallas, USA, 2017: 27-37

[15] Chen Y, Wang Z, Whalley D, et al. Remix: On-demand live randomization//Proceedings of the 6th ACM Conference on Data and Application Security and Privacy. New Orleans, USA, 2016: 50-61

[16] Yun J, Park K W, Koo D, et al. Lightweight and seamless memory randomization for mission-critical services in a cloud platform. Energies, 2020, 13(6): 1332

[17] Shao X, Luo L, Ling Z, et al. fASLR: Function-based ASLR for resource-constrained IoT systems//Proceedings of the 27th European Symposium on Research in Computer Security. Copenhagen, Denmark, 2022: 531-548

[18] Zhang Gui-Min, Li Qing-Bao, Zeng Guang-Yu, et al. Defensing code reuse attacks using live code randomization. Journal of Software, 2019, 30(9): 2772-2790(in Chinese)
(张贵民, 李清宝, 曾光裕等. 运行时代码随机化防御代码复用攻击. 软件学报, 2019, 30(9): 2772-2790)

[19] Xie M, Lin Y, Luo C, et al. PointerScope: Understanding pointer patching for code randomization. IEEE Transactions on Dependable and Secure Computing, 2022, 20(4): 3019-3036

[20] Jelesnianski C, Yom J, Min C, et al. Mardu: Efficient and scalable code re-randomization//Proceedings of the 13th ACM International Systems and Storage Conference. Haifa, Israel, 2020: 49-60

[21] Wang Z, Wu C, Li J, et al. Reranz: A light-weight virtual machine to mitigate memory disclosure attacks//Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. Xi'an, China, 2017: 143-156

[22] Koo H, Chen Y, Lu L, et al. Compiler-assisted code randomization//Proceedings of the IEEE Symposium on Security and Privacy (SP). San Francisco, USA, 2018: 461-477

[23] Wang Z, Wu C, Zhang Y, et al. SafeHidden: An efficient and secure information hiding technique using re-randomization//Proceedings of the USENIX Security Symposium. Santa Clara, USA, 2019: 1239-1256

[24] Nikolaev R, Nadeem H, Stone C, et al. Adelie: Continuous address space layout re-randomization for Linux drivers//Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Switzerland, 2022: 483-498

[25] Khandaker M, Liu W, Naser A, et al. Origin-sensitive control flow integrity//Proceedings of the USENIX Security Symposium. Santa Clara, USA, 2019: 195-211

[26] Khandaker M, Naser A, Liu W, et al. Adaptive call-site sensitive control flow integrity//Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P). Stockholm, Sweden, 2019: 95-110

[27] Bounov D, Kici R G, Lerner S. Protecting C++ dynamic dispatch through VTable interleaving//Proceedings of the Network and Distributed System Security (NDSS) Symposium. San Diego, USA, 2016: 100-113

[28] Bauer M, Grishchenko I, Rossow C. TyPro: Forward CFI for C-style indirect function calls using type propagation//Proceedings of the 38th Annual Computer Security Applications Conference. Austin, USA, 2022: 346-360

[29] Mohan V, Larsen P, et al. Opaque control-flow integrity//Proceedings of the Network and Distributed System Security Symposium. San Diego, USA, 2015

[30] Lin Y, Cheng X, Gao D. Control-flow carrying code//Proceedings of the ACM Asia Conference on Computer and Communications Security. Auckland, New Zealand, 2019: 3-14

[31] He W, Das S, et al. BBB-CFI: Lightweight CFI approach against code-reuse attacks using basic block information. ACM Transactions on Embedded Computing Systems, 2020, 19(1): 1-22

[32] DeLozier C, Lakshminarayanan K, Pokam G, et al. Hurdle: Securing jump instructions against code reuse attacks//Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems. Virtual Event, USA, 2020: 653-666

[33] Li J, Chen L, Shi G, et al. ABCFI: Fast and lightweight fine-grained hardware-assisted control-flow integrity. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2020, 39(11): 3165-3176

[34] Li Ya-Wei, Zhang Long-Bing, Zhang Fu-Xin, Wang Jian. A security protection mechanism on program runtime based on software and hardware cooperation. Chinese Journal of Computers, 2023, 46(1): 180-201(in Chinese)
(李亚伟, 章隆兵, 张福新, 王剑. 基于软硬协同的程序运行时安全保护机制. 计算机学报, 2023, 46(1): 180-201)

[35] Yoo S, Park J, Kim S, et al. In-kernel control-flow integrity on commodity OSes using ARM pointer authentication//Proceedings of the 31st USENIX Security Symposium (USENIX Security 22). Boston, USA, 2022: 89-106

[36] Zhang Zheng, Xue Jing-Feng, Zhang Jing-Ci, et al. Survey on control-flow integrity techniques. Journal of Software, 2023, 34(1): 489-508(in Chinese)

(张正，薛静锋，张静慈等. 进程控制流完整性保护技术综述. 软件学报，2023，34(1)：489-508)

[37] Li Wei-Wei, Ma Yue, Wang Jun-Jie, et al. ROP attack detection approach based on hardware branch information. Journal of Software, 2020, 31(11): 3588-3602(in Chinese)
(李威威，马越，王俊杰等. 基于硬件分支信息的 ROP 攻击检测方法. 软件学报，2020，31(11)：3588-3602)

[38] Lu K, Xu M, Song C, et al. Stopping memory disclosures via diversification and replicated execution. IEEE Transactions on Dependable and Secure Computing, 2018, 18(1): 160-173

[39] Schuster F, Tendyck T, Liebchen C, et al. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications//Proceedings of the IEEE Symposium on Security and Privacy. San Jose, USA, 2015

[40] Jason G, William E, Peng N. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities//Proceedings of the 5th ACM Conference on Data and Application Security and Privacy(CODASPY). San Antonio, USA, 2015: 325-336

[41] Crane S, Liebchen C, Homescu A, et al. Readactor: Practical code randomization resilient to memory disclosure//Proceedings of the IEEE Symposium on Security and Privacy. San Jose, USA, 2015: 763-780

[42] Werner J, Baltas G, Dallara R, et al. No-execute-after-read: Preventing code disclosure in commodity software//Proceedings of the 11th ACM Asia Conference on Computer and Communications Security(ASIACCS). Singapore, 2015

[43] Tang A, Sethumadhavan S, Stolfo S. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads//Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. Denver, USA, 2015: 256-267

[44] Yarom Y, Falkner K. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack//Proceedings of the 23rd USENIX Security Symposium. San Diego, USA, 2014: 719-732

[45] Song W, Liu P. Dynamically finding minimal eviction sets can be quicker than you think for side-channel attacks against the LLC//Proceedings of the International Symposium Research in Attacks, Intrusions and Defense(RAID). Beijing, China, 2019: 427-442

[46] Liu Fang-Fei, et al. Last-level cache side-channel attacks are practical//Proceedings of the IEEE Symposium on Security and Privacy. San Jose, USA, 2015: 80-93

[47] Niu B, Tan G. Modular control-flow integrity//Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. Edinburgh, UK, 2014: 577-587

[48] Niu B, Tan G. Per-input control-flow integrity//Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. Denver, USA, 2015: 914-926

[49] Caroline Tice, Tom Roeder, et al. Enforcing forward-edge control-flow integrity in GCC&LLVM//Proceedings of the 23rd USENIX Conference on Security Symposium. San Diego, USA, 2014: 941-955

[50] Mashtizadeh A J, Bittau A, et al. CCFI: Cryptographically enforced control flow integrity//Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. Denver, USA, 2015: 941-951

[51] Ge X, Talele N, Payer M, et al. Fine-grained control-flow integrity for kernel software//Proceedings of the IEEE European Symposium on Security & Privacy. Saarbrücken, Germany, 2016: 179-194

[52] Yuan P, Zeng Q, Ding X. Hardware-assisted fine-grained code-reuse attack detection//Proceedings of the International Symposium on Recent Advances in Intrusion Detection. Kyoto, Japan, 2015: 66-85

[53] Payer M, Barresi A, et al. Fine-grained control-flow integrity through binary hardening//Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Milan, Italy, 2015: 144-164

[54] Davi L, Koeberl P, Sadeghi A R. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation//Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC). San Francisco, USA, 2014: 1-6

[55] Van der Veen V, Andriesse D, et al. Practical context-sensitive CFI//Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. Denver, USA, 2015: 927-940

LI Yong-Gang, Ph. D., associate professor. His research interests include system security and system optimization.

CHUNG Yeh-Ching, Ph. D., professor. His research interests include parallel and distributed processing and system software.

ZHENG Yi-Jian, B. S. candidate. His research interest focuses on information security.

LIN Guo-Yuan, Ph. D., associate professor. He has long been engaged in the research of information security, and operating system.

BAO Yu, Ph. D., associate professor. His research interest is AIoT security.

Background

This paper aims to mitigate the control flow security issues in the closed source software, and its defense goal is the code reuse attacks (CRAs). With the popularization of ASLR (especially the fine-grained ASLR), the code probe has gradually become the first step in deploying CRAs. Overall, the entire deployment of one CRA can be divided into two steps: code probe and gadget connection.

Faced with the challenges brought by CRAs, existing methods have established methods such as re-randomization and CFI. Re-randomization performs another randomization on the binary code during the execution of the target process to invalidate the code address information obtained by the attacker. However, this method faces the problem of determining the randomization point and selecting randomization objects. Randomization is meaningful only when a code probe events occurs and before a CRA is deployed. Otherwise, the attacker can still use the probed code snippets as gadgets. Moreover, most existing methods randomize the entire code segment of a process, which requires addressing complex call relationships between different functions or code blocks. To keep the code logic unchanged before and after randomization, many methods have to rely on source code analysis, which leads to their invalidity with closed source objects.

The existing CFI methods rely on high-precision CFGs. However, due to the unknown logic of the closed source object, it is not possible to establish a high-precision CFG for it. Although a CFG can be built by the static analysis of binary code, this method faces the problem of state explosion. The set of explosive states contains the execution paths that do not exist in real scenes, making it difficult to filter out legitimate behavioral characteristics. In addition, most existing CFI methods track all control flow transfers throughout the entire lifecycle of the target process, which is a key factor that incurs overhead.

In previous research, we have established a mechanism for detecting code probes. Based on this mechanism, a framework is built to locate risk functions, which has been published in ACSAC-2022[13]. On this basis, this paper proposes a new method called MCE. MCE can filter out the potential attack scenarios and extract risk code snippets from them by sensing code probes. Then, the risk code will be migrated to a new address space. Next, all control flows that jump into the risk code will be captured and analyzed. In fact, MCE is a security method triggered by code probe events. It migrates risk code from the original space to a new space if there is a code probe in the target process, which is similar to the ASLR method. In addition, MCE utilizes context-sensitive security policies to detect the legitimacy of control flows, which is similar to the CFI method. Therefore, MCE is a hybrid method that combines ASLR and CFI. Compared with existing methods, MCE has stronger target specificity and scenario specificity. Therefore, it can achieve better protection effects with less overhead. To the best of our knowledge, MCE is the first risk code protection method based on code probe detection. It can provide strong control flow protection for the closed source software. This work is supported by "the Fundamental Research Funds for the Central Universities" (No. 2023QN1078).