CON-MVX：一种基于容器技术的多变体执行系统

刘子敬[1], 张　铮[1], 席睿成[1], 朱鹏喆[1], 邬江兴[2]

[1]数字工程与先进计算国家重点实验室 郑州 中国 450001
[2]国家数字交换系统工程技术研究中心 郑州 中国 450002

摘要　多变体执行是一种网络安全防御技术, 其利用软件多样性生成等价异构的执行体, 将程序输入分发至多个执行体并行执行, 通过监控和比较执行体的状态来达到攻击检测的目的。相较于传统的补丁式被动防御技术, 多变体执行不依赖于具体的攻击威胁特征进行分析, 而是通过构建系统的内生安全能力来对大多数已知、甚至未知的漏洞做出有效防御。近年来, 多变体执行技术在不断改进和完善, 但是误报问题是制约其发展的主要因素。本文针对多变体执行产生误报的原因进行了详细分析, 并在此基础上提出利用容器技术实现多变体执行系统在解决误报问题上的优势。为提升多变体执行技术的可用性, 本文设计并实现了一种基于容器技术的多变体执行系统 CON-MVX, 有效解决传统多变体执行系统的误报问题。CON-MVX 利用多个经过运行时随机化技术构建的异构容器作为执行体, 使用可重构的模块化组件和独立的容器管理工具对容器执行体进行编排管理, 建立进程间监控器 CGMon, 在内核层级实现对多个执行体的输入同步和输出裁决。同时, 为满足与客户端良好交互性, 建立中继端口策略, 保证系统运行状态的正常反馈。实验结果表明, CON-MVX 在保证安全能力的前提下, 能有效降低多变体执行系统的误报率, 在双冗余度执行条件下使用 SPEC CPU 2006 测试集测试时, 系统带来的平均额外性能损耗不超过 15%。

关键词　多变体执行; 容器技术; 软件漏洞; 安全能力
中图法分类号　TP393　DOI 号 10.19363/J.cnki.cn10-1380/tn.2024.03.04

CON-MVX: A Multi-Variant Execution System Based on Container Technology

LIU Zijing[1], ZHANG Zheng[1], XI Ruicheng[1], ZHU Pengzhe[1], WU Jiangxing[2]

[1] State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China
[2] National Digital Switching System Engineering & Technological R&D Center, Zhengzhou 450002, China

Abstract　Multi-variant execution (MVX) is a network security defense technology. It uses software diversity to generate equivalent heterogeneous variants, distributes input to several variants for parallel execution, and achieves the purpose of attack detection by monitoring and comparing the states of variants. Compared with traditional patch-type passive defense technology, MVX does not depend on analysis of specific attack threat characteristics, but can effectively defend against most known or even unknown vulnerabilities by establishing the endogenous security capability of the system. In recent years, the MVX technology has been improved and perfected a lot, but the false positive problem is the main factor restricting its development. We analyze the causes of the false positive problem of MVX technology in detail, on this basis, we propose the advantages of using container technology to build MVX system in solving the false positive problem. To improve the availability of MVX, we design and implement a container-based MVX system which is called CON-MVX to effectively address the false positive problem of traditional MVX systems. CON-MVX uses multiple heterogeneous containers constructed by runtime randomization as variants, uses reconfigurable modular components and independent container management tools to arrange and manage container variants, establishes the cross-process monitor which is called CGMon to synchronize the input and verdict the output of several variants at the kernel level. At the same time, a trunk port policy is established to ensure the normal feedback of the system running state in order to meet the favorable interactivity with the client. The experimental results show that CON-MVX can effectively reduce the false positive rate of the MVX system under the premise of ensuring the security capability, and the average extra performance loss of the system is less than 15% in our test with SPEC CPU 2006 under the dual redundancy execution condition.

Key words　multi-variant execution; container technology; software vulnerability; security capability

通讯作者：张铮, 博士, 副教授, Email: ponyzhang@126.com。
本课题得到国家自然科学基金资金项目(No. 61521003); 国家重点研发计划基金资助项目(No. 2018YFB0804003)资助。
收稿日期: 2022-04-26; 修改日期: 2022-09-30; 定稿日期: 2023-11-01

1 引言

互联网是一个复杂的动态环境[1], 随着信息技术的进步和程序代码量级的提升, 软件产品的功能越来越复杂, 从软件安全与其复杂性关系的角度来看, 不可避免的会产生诸多漏洞, 本地环境下漏洞带来的威胁是可以控制的, 但是在信息高速传播和网络全球化的背景下, 软件产品的供应链不断拉长, 使得漏洞在网络环境中得以迅速传播, 给软件供应链带来安全威胁, 甚至会造成供应链断裂, 带来较为严重的后果。其中, 软件同质化是加剧供应链风险程度的重要因素[2-3], 开源代码框架的使用继承给开发者带来便利的同时, 极大的降低了攻击者利用漏洞进行破坏的门槛, 使得漏洞后门可以被利用到相似环境的目标系统中, 扩大了网络安全的受威胁范围。

为解决软件同质化带来的安全威胁, 研究者从软件多样化的角度对攻击手段进行缓解, 限制攻击的传播途径[4]。但是由于存在软件“熵”空间的限制, 使得仅仅依靠多样化无法实现对软件的全方位保护[5-6]。近年来, 研究者在软件多样化思想的基础上, 提出了一种基于异构冗余思想的多变体执行防御技术, 有效地解决了仅依靠软件多样化带来的局限性, 为网络空间安全提供一种全新的主动防御手段[7]。多变体执行技术使用软件多样化技术构建功能等价、结构互异的执行体集合, 在程序运行时为每个执行体赋予相同的输入, 设置同步分发器保证多个执行体的输入同步, 利用监控器监视每个执行体的输出行为并检测其输出结果的区别, 通过裁决输出结果, 就可以发现程序执行过程中存在的异常行为, 进而达到安全防御的目的[8]。

然而, 现有的多变体执行架构依然存在许多不完善的方面, 裁决误报是导致多变体执行架构可用性降低的主要因素。裁决误报问题产生的主要原因是系统监控器会把不需要参与裁决的变量被动地容纳到裁决过程中, 导致正常的输入在多个执行体并行执行的过程中产生不一致的结果, 进而引起监控器的裁决误判, 将正常的输入判定为攻击异常, 严重影响了多变体执行系统的可用性。

为解决多变体执行技术中存在的误报问题, 本文基于容器技术设计实现一种新型的多变体执行系统 CON-MVX。CON-MVX 利用自定义容器编排工具管理多个异构容器执行体; 利用容器命名空间隔离机制, 实现对多个执行体系统资源的隔离, 为执行体创建独立的运行环境, 避免非必要的裁决因素对系统监控裁决的影响; 通过构建内核层级的监控器, 实现多个相互隔离的执行体之间的资源同步和输出裁决。实验证明, CON-MVX 在保证多变体执行系统安全防御能力的同时, 能够有效降低误报率, 极大地提升了多变体执行技术的可用性。

本文的主要创新点有:

1) 本文首次通过容器技术构建多变体执行系统, 利用容器的资源隔离属性, 降低系统误报率, 提升了多变体执行的可用性;

2) 本文提出多异构容器执行体构建模型, 为多个异构容器创建统一的管理环境, 对攻击者展现为无法预知内部结构的黑盒容器组, 保证多变体执行系统的安全性;

3) 本文基于 Linux 内核加载模块设计实现多变体执行进程间监控器(Container Group Monitor, CGMon)。CGMon 建立在内核态, 能够有效实现对多个命名空间相隔离的执行体系统调用的监控和资源同步。

文章结构安排如下: 第 1 章为全文的引言部分; 第 2 章介绍背景知识和研究现状, 并对误报问题进行分析解决; 第 3 章提出系统威胁模型; 第 4 章提出系统设计方案; 第 5 章阐述系统实现方案; 第 6 章对原型系统进行全面的实验评估; 第 7 章通过分析系统局限性对未来工作进行展望; 第 8 章对全文进行总结。

2 背景及现状分析

2.1 背景知识

控制流劫持是一种常见的攻击手段, 攻击者通常利用缓冲区溢出等类型的软件漏洞, 非法篡改进程中的控制数据, 从而改变进程的控制流程并执行特定恶意代码, 达到攻击目的[9]。控制流劫持攻击的实例包括代码注入、代码重用、面向返回的编程(ROP)、面向跳转的编程(JOP)和面向调用的编程(COP)等。针对控制流劫持的防御措施已经得到了广泛的研究, 其中包括程序运行前静态分析检测[10-12], 数据执行保护[13], 内存地址空间布局随机化[14-17], 控制流完整性等技术[18-20]。虽然这些防御手段能够降低成本和性能开销, 对特定的漏洞实施有效防御, 但是只能防御有限类别的漏洞或特定的攻击方式, 无法高效抑制层出不穷的 0-day 漏洞。

攻击技术和防御技术始终在此消彼长, 不断演进, 防御技术也由传统的被动防御发展到新型的主动防御。多变体执行技术是主动防御在软件层级应用的典型代表[8]。Cox 等人于 2006 年首次提出多变

体执行架构用于解决软件安全问题[7]。作为一种主动防御方式, 多变体执行使用软件多样性技术生成执行体集合, 将程序输入分发至多个功能相同、结构互异的执行体并行执行, 设置裁决点, 通过比较发现执行体状态的不一致来检测攻击。攻击者必须在不触发裁决检测的情况下同时对多变体执行架构中所有变体实施攻击才能达到攻击的目的, 极大增加了攻击的难度。多变体执行技术利用的软件多样性技术, 通过对冗余异构执行体的监控裁决能够有效防御基于控制流劫持的攻击手段, 同时, 较传统防御手段在面临未知威胁时的适用性更强, 更具有普适性[7, 21]。

程序运行时随机化技术为网络空间安全提供了新的技术手段和发展方向, 其在空间维度上产生结构差别的特性能够有效的阻隔攻击链的相关性, 为多变体执行技术中异构执行体的建立提供了有效支撑[8]。

执行体的输出裁决技术是多变体执行技术的核心内容之一, 在进行多变体执行系统监控时采用的方法主要有: 在单一副本程序或从执行体中进行监控, 通过内存缓冲区读取主执行体的输出结果进行裁决判定, 本文称这种方式为进程内监控(In-process monitoring, IPM), IPM 主要通过建立共享内存缓冲区, 向其中写入主执行体的输出数据, 从执行体在锁步执行的前提下, 读取缓冲区数据, 并在系统调用层级进行比较裁决[22]。但是这种监控手段存在一定的安全隐患, 无法确保攻击者是否对从变体共享内存缓冲区进行数据篡改, 监控器的执行容易受到干扰甚至破坏。另一种监控手段是建立独立于执行体进程的监控器进程, 本文称这种方式为进程间监控(Cross-process monitoring, CPM), CPM 的监控的粒度通常在系统调用层级, 通常在用户态采用 ptrace 调试工具对系统调用进行拦截处理, 通过程序在系统调用之前暂停副本, 检查它们的参数是否相等来进行裁决判定。

2.2 研究现状

多变体执行技术虽然具有较传统防御技术更高的适用性, 但是仍然存在一定的局限, 其中, 误报问题是制约多变体执行技术发展的主要瓶颈。在不具备攻击输入的条件下, 多个执行体并行执行时出现不一致的程序状态转换或输出结果, 监控器将正常执行识别为攻击行为, 称为多变体执行过程中的误报问题。近年来, 解决多变体执行的误报问题始终是研究重点。2015 年 Hosek 等人[22]提出的 VARAN 多变体执行系统, 通过减小共享环形缓冲区大小的方式来降低系统调用的判断次数, 以此来解决误报, 但是会给系统性能带来较大影响, 同时, 由于其未能对敏感与非敏感系统调用进行区分, 使得在进行监控器裁决时, 仍然容易出现误报, 给攻击者带来更多的利用机会。2016 年 Volckaert 等人[23]提出的 ReMon 监控器框架, 针对监控器 IP-MON 进行系统调用裁决策略的改造, 但是依然无法避免程序本身与操作系统底层过多的资源交换带来的误报问题。该类基于系统内核的多变体执行架构在处理执行体裁决粒度问题上仍然面临困难, 在进行系统调用处理过程时仍然存在较多的误报, 监控器的执行容易受到多个执行体进程间通信的影响, 导致系统攻击面扩大, 安全性能降低。2020 年, 弗吉尼亚理工大学团队提出分布式的多变体执行框架 HeterSec[24], 通过多个硬件的指令集架构的异构性提高应用程序安全性。HeterSec 使保护进程能够利用多样化指令集作为动态防御的附加层, 通过多个编译器和内核扩展构建起安全执行环境, 主从执行体通过 InfiniBand 连接通信。这样虽然能够降低系统误报率, 但是多个异构芯片之间的通信给性能带来了较大的损失, 而且部署成本较高, 应用场景较为狭窄。2021 年, 潘传幸等人[25]基于拟态防御原理提出面向进程控制流劫持的防御系统 Mimicbox, 能够有效防御基于绝大部分基于已知类型二进制漏洞的控制流劫持攻击, 但是依然存在冗余进程之间内生的不一致属性导致的误报问题。

2.3 误报原因分析

现有的多变体执行技术以进程为冗余执行的主体, 在多个执行体进程并行执行过程中, 将部分进程之间相关联的对象, 如进程内部随机数变量、进程外部由操作系统管理的进程号(PID)和文件描述符等被动地纳入到了多变体执行的裁决过程中, 由这些合法的不一致因素产生的正常不一致输出被监控器误判为异常, 从而产生误报。为更清晰地解释误报问题产生的原因, 本文根据图 1 实例进行分析, 执行体 1 和执行体 2 分别向操作系统请求 PID, 作为 libc 库随机数种子。PID 是单一进程在整个系统空间下的唯一认证, 所以不存在两个相同 PID 的进程。在传统多变体执行系统中, 由于执行体是通过多次 *fork()*调用完成创建的, 不存在父子进程关系, 所以系统 PID 号必然不一致, 导致生成的随机数也不同, 在后续使用随机数, 如 HTTPS 协议交换随机数等应用场景时输出产生不一致, 监控器将正常随机数读取误报为攻击行为。

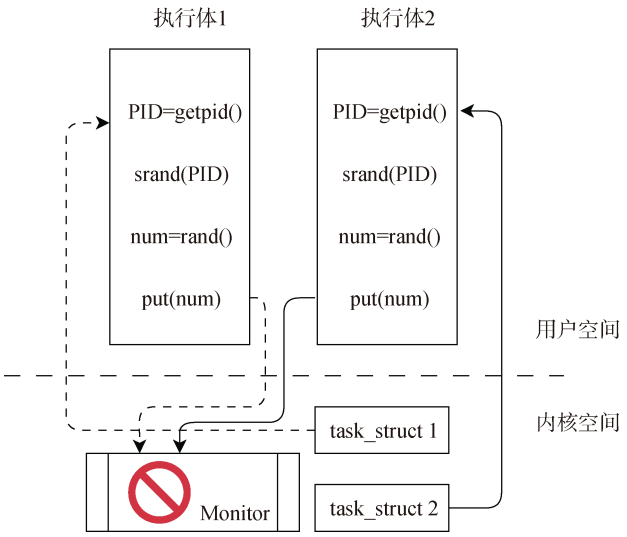


图 1　多变体执行误报问题

Figure 1　Multi-variant execution false positive problem

2.4　误报问题解决方法

解决误报问题的关键在于为执行体创建独立的运行环境, 消除多变体执行中合法的不一致因素给裁决带来的影响。容器的隔离机制可以避免多变体执行过程中进程内外部的不一致因素糅杂到裁决过程中, 有效解决裁决误报问题。隔离机制是通过将运行在容器中的进程包装到命名空间(namespace)中, 限制进程对运行在其他容器和底层主机中进程的可见性来实现的。Namespace 机制通过对系统资源进行封装, 有效隔离容器内进程和系统进程的 PID、Mount、IPC 等资源手段[26-27]。以 PID 为例, 由于 PID 名称空间是分级的[28], 一个进程只能在自己所属命名空间或其子命名空间中看到其他进程。在涉及 PID 参与的裁决过程中, 由于不同命名空间中 PID 允许重复, 所以两个执行体可以拥有相同的 PID, 不会因 PID 不同产生其他不一致因素而造成裁决误报。同理, 其他不一致因素造成的误报问题也可以利用容器得到有效解决。

3　威胁模型

本节通过建立威胁模型对 CON-MVX 的安全能力做出限定。CON-MVX 旨在防御利用内存漏洞实现控制流劫持的攻击手段。本文假定受保护的应用程序存在内存漏洞, 攻击者利用 ROP 等攻击手段, 实现应用程序控制流的异常跳转, 进而控制敏感类系统调用对系统内存进行非法操作, 达到攻击目的。其中, 攻击者能够利用的内存漏洞类型包括缓冲区溢出、整数溢出、UAF 等。本文对威胁模型做出以下限定条件:

(1) 假定攻击者实施攻击时采用远程控制的方式, 无法获得 CON-MVX 本地用户的操作权限, 无法对本地 C/S 模式构成威胁;

(2) 攻击者能够在受保护的程序中找到存在的内存漏洞, 在仅包含单一执行体的条件下, 可以绕过 NX、PIE、Canary 等传统防御措施, 构造有效攻击载荷实施控制流劫持;

(3) 不考虑除内存漏洞外的其他软件漏洞的攻击利用。同时, 将操作系统底层数据接口和硬件设备列入可信计算基, 不考虑基于硬件的攻击手段[29-31]。

4　系统设计

4.1　系统整体架构设计

CON-MVX 系统的整体架构图如图 2 所示, 容器通过本地 C/S 的模式实现创建和运行管理, 利用本地客户端与服务端之间的连接实现控制流的交换和数据流的转移。CON-MVX 通过多个功能组件实现对容器执行体的管理和控制, 建立 mimictlclient 客户端, 制定适用于本客户端的指令标准, 用户利用既定指令通过 gRpc 通信传递至 containerd。containerd 是一个强调简单性、健壮性和可移植性工业级标准的容器运行时, 在容器建立和控制过程中可以跳过对 docker-shim 的调用, 缩短容器执行体实例化过程的调用链[32], 最终通过底层调用 runC 完成多个执行体的实例化。通过运行时随机化技术生成异构执行体, 保证各个执行体在内存空间布局上的异构性。由于 namespace 机制的加入, 使得各个容器执行体在利用 UTS、PID、IPC、Mount 等系统资源的调度手段时实现隔离。系统的监控裁决模块通过 CPM 方式实现, 多个执行体进程通过系统调用接口与 CGMon 进行交互, 数据的输入和输出都需经过监控器的同步和裁决, 保证进程运行过程中的功能等价性。利用内核探针实现敏感系统调用(例如 *write()*)的监视, 利用 Monitor 对输出或写入内容进行裁决判定。针对非敏感类系统调用(例如 *read()*)直接将读取内容通过缓冲区返回执行体, 系统调用的分类极大降低了性能损耗, 同时, 加入缓冲区清洗机制, 对频繁系统上下文切换产生的 TLB 虚拟页缓存进行清洗, 进一步降低由于冗余执行带来的性能损耗。在处理远程网络连接时, 建立中继端口策略, 在 socket 层设置代理进程, 代理进程完成解包之后, 通过中继端口将数据内容拷贝分发给多个容器执行体, 保证输入内容的一致性。

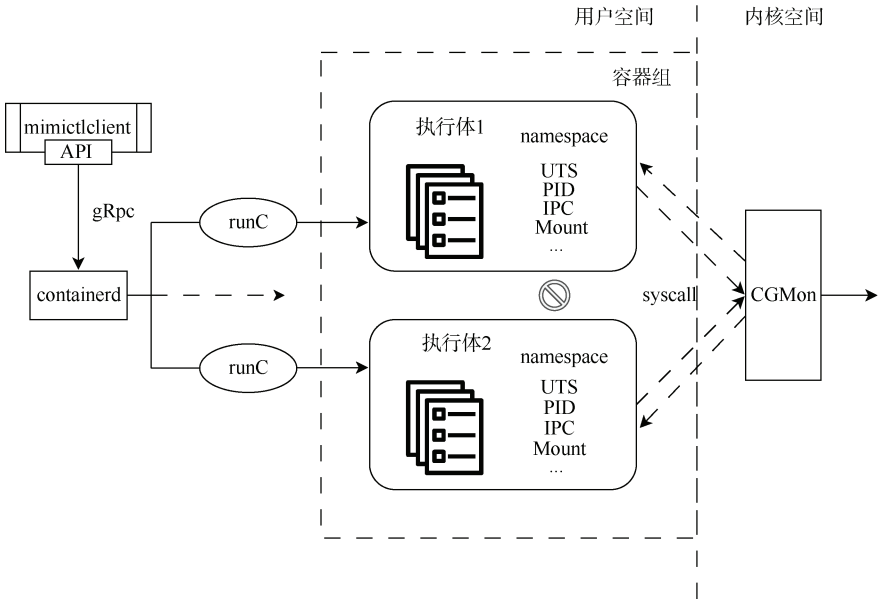


图 2　CON-MVX 系统整体架构

Figure 2　CON-MVX architecture overview

4.2　异构容器执行体生成

整个系统的执行体构件是容器, 由于无法保证在修改并利用镜像文件后生成容器的功能等价性, 本文在容器运行时实现异构执行体的建立。在通过 mimictlclient 实例化多个容器执行体时, 利用相同的镜像文件, 保证各执行体的功能等价性。由于容器与主机操作系统共享内核, 可以利用程序运行时随机化技术实现执行体的异构, 因此, 本文采用程序运行时随机化技术生成异构执行体。传统的防御措施在面临控制流劫持的攻击手段时, 会因为攻击者构建更加完备的攻击载荷而被击破, 比如单字节爆破可以逐个字节泄露利用函数的返回地址, 覆盖 Canary 值绕过检查, 达到攻击利用的目的。但是, 通过构建多个内存布局互异的执行体可以有效的避免该类问题出现。通过地址空间随机化技术, 堆、栈布局随机化技术[33], 数据随机化技术[34]等方式, 异构每个容器执行体的内存布局空间, 实现异构执行体的构建。如图 3 所示, 建立执行体 1、2、3 的异构内存空间, 设置数据段随机秘钥 1、2、3 分别插入多个执行体内存空间, 进一步提升执行体的异构性, 保证系统的安全防御能力。

4.3　执行体输出裁决

CPM 较 IPM 的安全性更高, 但是由于频繁的上下文切换以及由此产生的过多的 TLB 虚拟页缓存给性能带来一定的影响。由于容器 namespace 机制给进程提供的隔离环境, 使得采用 IPM 方式进行监控变得困难, 这样做不仅违背了隔离执行体环境建立的初衷, 还会给执行体运行时带来更多的不安全因素。因此本文拟采用 CPM 的方式在内核空间设计实现监控器组件。本文设计容器组监控器 CGMon 组件如图 4 所示, 实现对容器组内多个容器执行体的监控。在 CGMon 监控组件中, 建立 Syscall_SYN_Handler 处理装置, 用于对系统调用进行分类并实现同步处理。

为控制对所有系统调用进行监控带来的性能损耗, 本文将系统调用分为敏感类和非敏感类。敏感类主要包括修改文件状态, 例如 write()等写入类操作; 改变进程状态, 通过系统调用创建或终止进程, 例如 exec()、fork()、exit()等; 以及其他可能会对容器外系统造成破坏的系统调用, 例如一些关于内存映

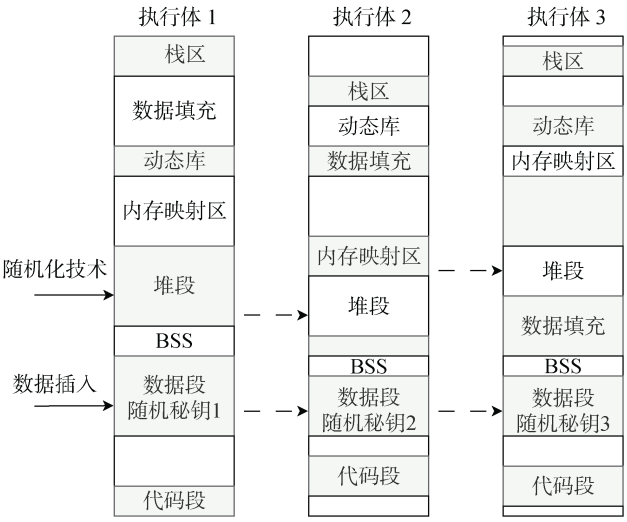


图 3　异构容器执行体内存空间布局

Figure 3　Heterogeneous variant memory space layout

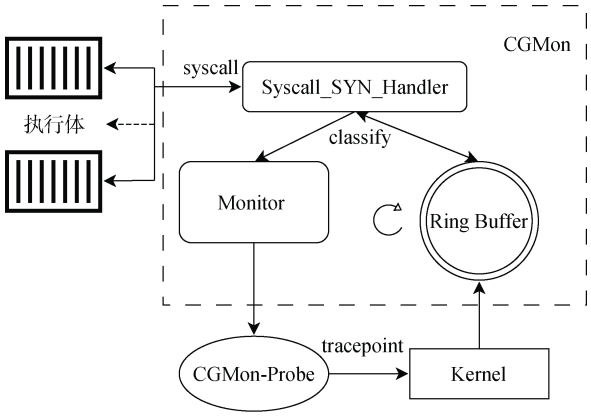


图 4　系统监控部件

Figure 4　System monitoring components

射类的系统调用 mmap(), 由于容器内独立的 IPC, 使得不同容器之间的无法通过命名共享内存块、信号量以及消息队列等实现共享内存访问, mmap()系统调用可能会改变容器进程内存空间, 造成攻击逃逸, 因此列为敏感系统调用, 同时一些对底层操作系统状态可能会产生改变的系统调用都要列入敏感类的系统调用。非敏感类的系统调用包括监视进程内部属性的系统调用, 由于不同执行体处于不同 namespace, 无法对其他执行体造成干扰, 所以无需关心容器执行体内进程对自身状态的监视; 将涉及读取操作的系统调用设置到非敏感调用中, 例如 read()。系统调用的分类如表 1 所示。在 Syscall_SYN_Handler 组件中设置白名单机制, 将非敏感类系统调用设置在信任列表, 实现对敏感类和非敏感类系统调用的分类处理。本文提出的监控手段本质上是 CPM, 但是通过在 Ring Buffer 中加入缓存刷新机制, 能够一定程度上降低系统的整体性能开销。

表 1　系统调用分类

Table 1　System calls classification

敏感类系统调用	非敏感类系统调用
*write,*writev,*pwrite64,*pwritev,*sendto,*sendmsg,*sendmmsg,*sendfile,*epoll_ctl,*setsockopt,*mmap,*shutdown,*exec,*fork,*exit……	*read,*readv,*pread64,*preadv,*select,*poll,*epoll_wait,*recfrom,*recvmsg,*recvmmsg,*getsockname,*getpeername,*getsockopt,*gettimeofday,*clock_gettime,*time,*getpid,*gettid,*getpgrp,*getppid,*getgid,*getegid,*getuid,*geteuid,*getcwd,*getpriority,*getrusage,*times,*capget,*getitimer,*sysinfo,*uname,*sched_yield,*nanosleep……

5　系统实现

本系统基于 Linux 3.10.0-1160.36.2.el7.x86_64 版本内核实现, 采用 containerd-shim-runc-v2 标准, 调用此版本下 runC 的 API 进行创建容器 ID、bundle 目录等操作; 利用 v1.5.4 版本的 containerd 来管理镜像和容器; 对开源的 containerd cli 客户端 nerdctl(v 0.17.1)[35]进行改造, 建立一套适配本系统的指令标准, nerdctl 是 containerd 的非核心子项目, 在进行代码改造时可以对 docker 容器的相关操作进行继承。

5.1　容器执行体生成

在初始化阶段, 首先创建容器 rootfs 文件系统, 保证组内每个容器使用同一个 rootfs 文件系统。rootfs 是一个容器启动时其内部进程可见的文件系统, 在容器内用户视角下修改文件时, 一般会采用写时复制机制, 保证组内容器在同一个 rootfs 下是实现后续输出结果裁决裁定和异常结果处理的必要条件[36]。在容器组内创建异构容器, 组内的每个容器属于不同的命名空间, 创建容器时, 不会使用 containerd 直接去操作容器, 而是创建一个 containerd-shim 的进程去操作容器。containerd-shim 是容器进程的父进程, 可以收集容器执行体状态, 维持 stdin 等 fd 打开工作。避免了容器进程与 containerd 耦合的问题。containerd-shim 和 containerd 不要求是父子进程关系, containerd 退出或重启, shim 会指定进程到类似 systemd 的 1 号进程上。containerd-shim 通过调用 runC 实现 namespace、cgroups, 挂载 rootfs 等 OCI 规范操作。

在容器组内容器正常创建时, 采用随机字符串 UUID 命名的方式设置容器组 ID 和容器 ID, 对其唯一性进行标识, 同时建立组 ID 和容器 ID 之间的映射关系。在完成容器组创建并正常运行组内容器时, 将该对应关系进行保存, 存储位置为自定义路径, 存储形式为 csv 文件。

5.2　网络 I/O 数据处理

容器组对用户呈现为单一容器进程, 在处理远程访问连接时, 需对外暴露统一端口, 实现方式为建立执行体中继端口策略, 利用 TCP 代理进程通过中继端口完成到各执行体数据的转发。

5.2.1　中继端口策略

中继端口为容器组对外暴露的服务端口, 所谓“中继”主要体现在以下两个方面:

(1) 与外界客户端交互, 将客户端请求数据发送给容器组;

(2) 与容器交互, 将容器组响应数据发送给客户端。

中继端口在容器组创建时额外分配, 分配遵循如下原则:

(1) 每个容器组每一组端口配置对应一个中继服务端口, 例如创建容器组时配置-p 5000:5000 和-p 6000:6000 两组端口, 则需要额外分配两个中继端口;

(2) 中继端口必须分配未被占用的主机端口。

5.2.2 代理进程实现

代理进程建立在 socket 层, 在客户端与 nerdctl0 网桥建立连接完成认证之后, 通过 TCP 代理进程将数据包拷贝后分发给每个容器执行体。代理进程实现一对多端口映射到多个容器执行体, 对外呈现唯一接入的服务端口。代理进程的实现基于 libevent 库, 版本号为 libevent-2.1.12-stable。TCP 代理进程的实现采用多线程的处理方式, 实现方式如图 5 所示。

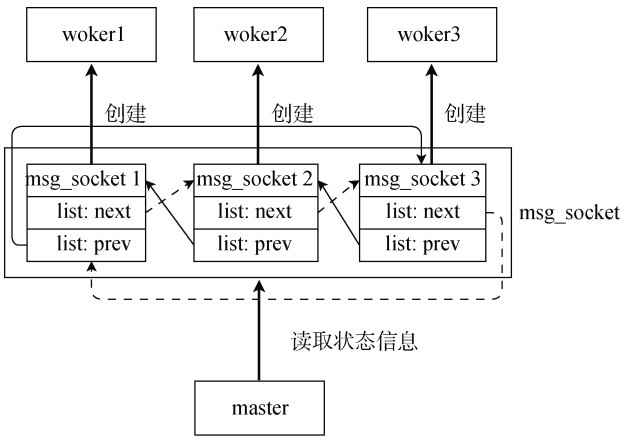


图 5 多线程代理进程结构

Figure 5 Multithreaded agent process structure

下面对数据结构和线程的功能进行解释说明:

msg_socket 链表: 内核的 list_head 链表, 链表中的每一个结点对应一组等价容器, 存储网络 I/O 通信的 socket 结构, 容器映射端口等信息。链表的维护采用内核 RCU 机制[37], 提升并发操作的性能。

master 线程: 作为守护线程, 负责不断查询内核 msg_socket 链表, 若结点存在且不存在响应的 worker 线程, 则创建一个 worker 服务线程。

worker 线程: 负责每一组等价容器与客户端的通信网络 I/O 数据的分发和结果反馈。

代理进程运行过程中, 通过申请 libevent 事件基础结构体并绑定 TCP 连接建立函数创建事件处理对象, 同时, 设置回调函数来根据执行体个数创建分发时需要建立的 TCP 连接个数, 只有从传输对象读取到足够多的数据后才会调用回调函数。通过调用 libevent 库函数实现对 TCP 数据的解析, 获得协议包数据。再利用接口函数将 TCP 缓存中的数据写入事件处理对象的写缓冲区中, 最后通过建立的多个 TCP 连接完成多个容器执行体的数据传输。

5.3 系统调用监控与资源同步

CGMon 中主要通过 Syscall_SYN_Handler 组件实现系统资源的同步处理。利用环形缓冲区 Ring Buffer 实现资源存储和消息传递, Ring Buffer 提供了可靠的消息传递特性, 同时为数据读取提供更高效的解决手段, Ring Buffer 中存储的是非敏感类系统调用读取的内容和敏感类系统调用经过裁决之后的返回值。

在涉及敏感类系统调用时, 同步执行点设置在 Monitor 装置裁决完成之后, 若通过裁决, 则调用 CGMon-Probe 探针, 利用内核组件 tracepoint 捕获在内核层面的事件, 完成单次执行体的系统调用, 之后返回正常值到环形缓冲区 Ring Buffer 中, 经过 Syscall_SYN_Handler 组件的同步处理后, 再将执行结果返回执行体, 有效避免资源竞争的产生; 若未通过裁决, 则无法正常返回, 系统检测到攻击行为, 并在用户终端回显异常信息。涉及到非敏感类的系统调用, 则直接将系统调用返回数据写入到 Ring Buffer 中, 经过 Syscall_SYN_Handler 组件同步读取, 正常返回执行体, 无需对每个执行体进行多次重复读取操作。

Syscall_SYN_Handler 组件的同步处理利用时间片轮询机制实现, 实现如图 6 所示。当执行非敏感类系统调用时, 例如图中的 read, 多个执行体读取的内容经过 Ring Buffer 后直接返回给各个执行体而不需要进行阻塞处理, 保证数据一致性; 当执行敏感类系统调用时, 例如图中的 write, 若通过裁决, 则将需要写入文件的内容存入 Ring Buffer, 在完成时间片阻塞后, 正常执行单次写入, 执行结果正常返回给各执行体, 保证后续调用的正常实施; 若未通裁决, 则丢弃 write 产生的时间片。

6 系统测试

本节针对 CON-MVX 系统降低误报率方面进行对比实验, 同时对整体的安全性和性能进行测试评估。本文采用的测试环境配置如表 2 所示。本文默认将 CON-MVX 作为独立的执行系统进行测试, 对比测试的多变体执行系统和基准测试程序均在物理机(非容器环境)下执行。

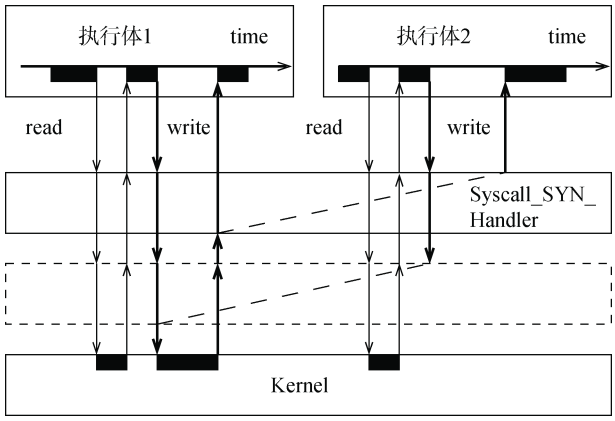


图 6　同步处理机制

Figure 6　Synchronous processing mechanism

表 2　测试环境配置

Table 2　Test environment configuration

项目	参数
操作系统	CentOS Linux release 7.9.2009 (Core)
Linux 内核版本	3.10.0-1160.36.2.el7.x86_64
CPU	Intel Silver 4210 CPU ×2 2.20GHz
内存	16GB×2 DDR4 2666

6.1　对比分析

本文选择 ReMon、Mimicbox 与 CON-MVX 进行对比实验, 分别测试其在强认证状态下的网关服务程序中所产生的误报率, 以此来验证本系统在多变体执行过程中对误报问题的优化解决。

网关程序在网络服务认证部署过程中一旦受到攻击会对整个网络环境产生严重影响, 同时其执行过程涉及到较多操作系统的进程管理操作, 因此, 本文基于实际应用过程中某电网网关服务端认证程序对三个系统进行对比实验测试, 实验结果对网关程序的安全部署具有重要意义。该网关服务端程序基于开源 OpenVPN 和 OpenSSL 定制开发, 其中 OpenSSL 软件包通过调用改造函数替换 OpenSSL 所提供的随机数生成方式, 该函数在/dev/random 设备接口的基础上将多项系统状态作为随机数生成的熵源, 其中包括由于进程间通信产生的不一致变量, 复杂的随机数熵源给用户认证带来更高的可靠性。将网关服务端程序部署于普通多变体执行架构时, 由于两个网关执行体进程在 TLS 双向认证过程中基于非一致状态生成不同的随机数, 进而加密生成不同的预主密钥输出, 极易在裁决过程中被监控器监测为非一致输出而产生误报。而通过将网关程序部署到 CON-MVX 中时, 命名空间对系统资源的隔离, 可以有效降低由于进程间通信带来的误报。将网关程序分别部署到三个测试系统中, 通过进行 1000 次访问认证, 对比三个系统的误报率, 实验结果如表 3 所示。通过对比实验结果分析, CON-MVX 可以有效降低多变体执行的误报率。同时, 网关程序中的响应时延是评估性能开销的重要标准, 通过测试平均认证响应时间, 对比三个系统的性能开销, 测试结果显示, CON-MVX 系统并未带来较高的访问延迟。

表 3　对比测试结果

Table 3　Contrast test results

测试程序	访问次数	平均响应时间 (ms)	误报率(%)
Mimicbox	1000	153.5	13.6
ReMon	1000	138.1	9.1
CON-MVX	1000	143.7	0.9

6.2　安全性分析

本节首先对 CON-MVX 多变体执行系统的防御原理进行分析。以利用缓冲区溢出漏洞为例对 ROP 攻击进行分析, 攻击者通过扫描动态链接库和可执行文件提取可利用指令片段(gadget), gadget 以 ret 指令结尾, 攻击者通过 gadget 拼接利用实现程序异常跳转, 进而达到攻击目的。如图 7 所示, 内存对象 A 存在栈溢出漏洞, 攻击目标是覆盖程序的函数 A 返回地址为 B, 进而控制程序的执行流程, 完成控制流劫持。攻击者在针对单一执行体进行 ROP 攻击时, 构造攻击载荷为“A+B+0x6007c00”覆盖正常返回地址, 进而跳转攻击利用函数 B, 即使开启 ASLR 机制, 由于各个模块只是基地址被随机化, 模块内部各对象之间的相对偏移地址未发生改变, 因此, 攻击者仍然可以通过构造有效攻击载荷实施控制流劫持。然而, 相同的有效攻击载荷只能覆盖相同返回地址到两个执行体中, 使得攻击者无法同时篡改两个执行

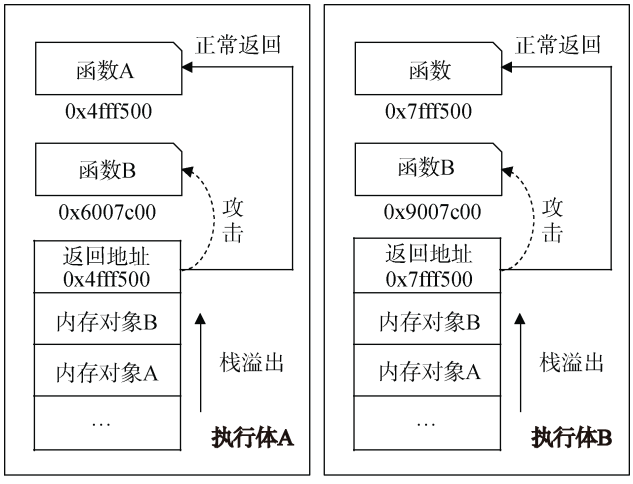


图 7　多变体执行防御原理分析

Figure 7　MVX defense principle analysis

体的返回地址为函数 B, 一旦某一执行体发生异常跳转, 会造成监控器 CGMon 裁决不一致, 检测到攻击行为。

继续通过实验验证系统的安全性, 本文选取 6 个开源软件中的 CVE 漏洞对系统安全性进行测试, 样本均通过 ExploitDB 平台公开获取, 漏洞类型包括栈溢出、整数溢出、UAF、堆溢出等, 测试结果如表 4 所示。

表 4　安全测试结果

Table 4　Safety test results

CVE 编号	漏洞类型	受威胁软件版本	测试版本	是否成功防御
CVE-2013-2028	栈溢出	Nginx1.3.9-1.4.0	1.3.9	是
CVE-2017-13089	整数溢出	Wget < 1.19.2	1.19.0	是
CVE-2017-16943	UAF	Exim 4.88-4.89	4.88	是
CVE-2021-23017	堆溢出	Nginx0.6.18-1.20.0	1.20.0	是
CVE-2021-3711	堆溢出	OpenSSL1.1.1-1.1.1k	1.1.1	是
CVE-2021-44790	栈溢出	Apache HTTP Server <= 2.4.51	2.4.17	是

以 Nginx 服务器存在的栈溢出漏洞(CVE-2013-2028)为例进行分析。动态调试过程中发现可以利用 HTTP 行解析函数实现整数溢出, 通过对描述 HTTP 消息实体传输长度的成员变量进行恶意赋值, 在程序进行赋值判断时, 会接收到超过缓冲区大小的数据, 进而实现缓冲区溢出, 获得程序的返回地址。然后根据获得的返回地址计算 Nginx 二进制文件的基地址, 并使用Nginx二进制文件构建一个ROP链, 得到的ROP程序执行使得攻击者获得远程shell执行权限, 达到系统攻击的目的。由于受漏洞影响版本的限制, 本文通过 GCC 4.8 编译的 Nginx 来进行测试, 测试版本为 1.39。测试过程中, 采用双冗余度执行, 构建有效攻击载荷发起控制流劫持攻击, 在攻击实施过程中, 两个执行体分别跳转到不同的返回地址执行, 由于内存地址空间布局的随机化, 使得该攻击载荷无法实现两个执行体内程序同时跳转到相同的返回地址, 异常跳转利用带来的系统调用结果内容无法通过 CGMon 裁决, 进而检测到攻击行为, 完成测试。实验结果表明该攻击样本失效, 即通过 CON-MVX 有效阻断控制流劫持攻击实施。

通过实验结果分析可知, 本系统能够有效防御基于控制流劫持的攻击手段, 且容器技术的加入并未从原理上降低原多变体执行技术的安全性。

6.3　性能分析

本文采用 SPEC CPU 2006 基准测试集评估系统正常运行时的 CPU 性能开销。将测试程序的运行时间作为性能损耗的评估标准, 把在非容器环境中正常执行的测试程序作为基准, 其运行时间作为基准运行时间。在容器执行体个数即冗余度为 $N(N\in 1,2,3,4,5)$的条件下, 分别测试其运行时间, 该时间与基准运行时间的比值表示为系统的相对性能损耗。其中, N=1 表示仅设置单一执行体, 即测试使用相同容器环境执行测试程序的情况下带来的性能损耗。在测试之前对测试程序进行编译优化, 在编译选项中添加包含-fno循环优化标志和-O2级别的编译优化选项。在编译完成之后, 将各测试程序进行封装, 并在 CON-MVX 系统环境下执行。同时, 为避免单一样本检测的偶然性, 本文分三次取样测试执行时间, 取平均值作为最终实验结果, 性能损耗的测试结果如图 8 所示。实验结果显示, 单独使用容器作为程序执行环境几乎不会带来额外性能损耗, 平均性能损耗为 2.51%。在双冗余度条件下, 即 N=2 时, 所有整型测试样本的额外性能损耗未超过 15%, 平均额外性能损耗为 8.59%。在高冗余度情况下, 当 N=5 时, 系统的平均损耗为 40.12%。其中, 在 N=2 时, 性能损耗最高的测试样本为471.omnetpp 14.82%, 同样性能损耗较为严重的测试样本为 429.mcf(14.57%), 470.lbm (13.69%)。经调查分析, 上述三个测试样本为访存密集型程序。在测试过程中, 执行样本 471.omnetpp 需要最大约 1.9GB 的内存空间, 访存次数频繁, 由此带来较为严重的性能损耗。Jaleel[38]对 SPEC CPU 2006 测试集的 cache 敏感性测试研究可以有效支撑本文的上述分析。

为进一步确保性能测试的完备性, 继续针对 I/O 密集型程序进行测试。本文选取服务器应用程序 tinyhttpd, 基于 webbench 测试集对本系统进行时延和吞吐量的测试, 测试访问次数分别为 500 和 1000, 执行体个数设置为 2, 将在非容器环境下单独执行 tinyhttpd 作为基准, 测试结果如表 5 所示。测试结果表示, 在低饱和状态, 即访问次数设置为 500 次时, CON-MVX 系统对时延带来的损耗为 13.67%, 对吞吐量带来的损耗为 27.73%。在较高饱和状态下, 即

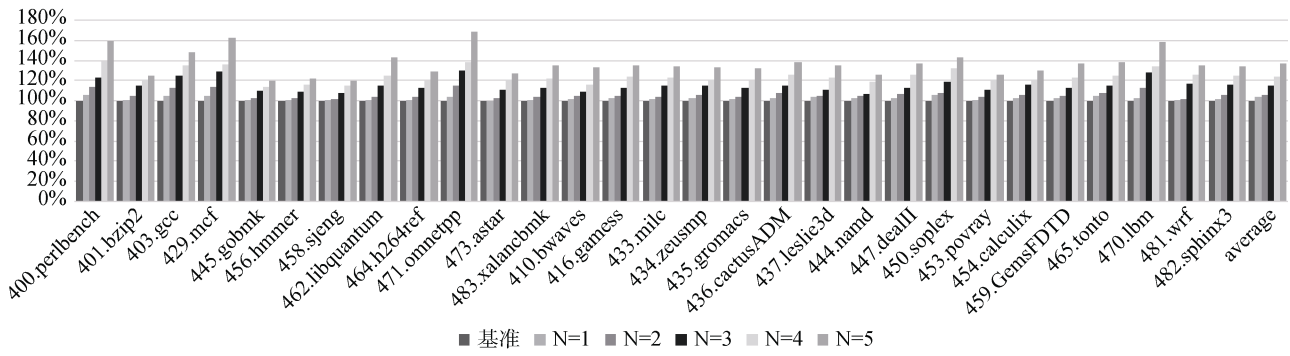


图 8　SPEC CPU 2006 性能测试结果

Figure 8　SPEC CPU 2006 performance test results

表 5　双冗余度条件下 tinyhttpd 性能测试结果

Table 5　Tinyhttpd performance test results under dual redundancy

	服务访问次数	时延(ms)	吞吐量(bytes/s)
基准	500	53.4	25496624
	1000	183.2	26672498
CON-MVX	500	60.7	18426125
	1000	212.4	18184162

访问次数设置为 1000 次时, CON-MVX 带来 15.94%的时延影响, 带来 31.82%的吞吐量损耗。

综上分析, 在处理 CPU 密集型程序时, CON-MVX 在高冗余度条件下, 仍然能保证可控范围内的性能损耗, 因为计算密集型程序的大多数系统调用不需要通过监控器裁决, 可以直接通过缓冲区 Ring buffer 完成同步合路操作, 外加缓冲区刷新机制, 能够较好地降低性能损耗。但是在处理 I/O 密集型程序时, 由于较多的文件读写、I/O 操作和内存处理操作, 需要对敏感类系统调用进行多次裁决比较, 监控器处于较高的工作负载下, 所以测试结果表示在双冗余度条件下, 处于较高饱和状态的服务器进程受到超过 30%的吞吐量性能损耗。

7　工作展望

CON-MVX 利用容器技术实现多变体执行系统, 对原有多变体执行系统在容易产生误报问题方面进行了优化, 但是容器技术的使用可能会带来系统攻击面的增加, 例如容器逃逸等问题, 对系统安全性在全方位的提升仍然是今后研究的重点内容; 在处理 I/O 密集型应用程序时, CON-MVX 系统的性能损耗较为严重, 可能无法有效应用到高并发和大流量处理场景, 通过优化监控器, 对整体系统的同步裁决机制进行优化, 或者通过改变同步策略, 或许会给系统带来更高的性能提升, 针对性能的优化也是今后研究的重要内容之一; 多变体执行技术中的效费比, 即冗余度提升可能带来的安全增益和相应的性能损耗之间的关系仍然需要在未来的工作中完善补充; 同时, 基于容器的多变体执行可以与拟态防御思想[39]相结合来实现软件安全防御, 将拟态防御的动态异构冗余构造引入到容器执行过程中, 在异构冗余的基础之上进行裁决, 将拟态防御发展到云端, 可以有效拓展拟态防御的业务场景。

8　总结

本文提出了一种建立在容器基础上、支持多个容器运行的多变体执行系统 CON-MVX。CON-MVX 创建和部署完整的多变体执行环境, 在统一 OCI 标准下, 利用开源可重用的模块化组件和独立的容器化管理工具, 在完成自定义改造后允许这些组件通过接口的形式相互调用和配合, 支撑整个系统的有效执行。经过实验验证, 本系统在安全性上能够有效防御基于控制流劫持的攻击手段, 多冗余度条件下性能损耗低, 有效降低多变体执行系统的误报率, 提升多变体执行技术的可用性。

致　谢　在此, 向评阅本文的审稿专家表示衷心的感谢, 向对本文工作给与支持和指导的同行, 尤其是课题组的老师表示感谢。

参考文献

[1] Adeyinka O. Internet Attack Methods and Internet Security Technology[C]. *2008 Second Asia International Conference on Modelling & Simulation*, 2008: 77-82.

[2] Zhang Y G, Vin H, Alvisi L, et al. Heterogeneous Networking: A New Survivability Paradigm[C]. *The 2001 workshop on New security paradigms*, 2001: 33-39.

[3] Stamp M. Risks of Monoculture[J]. *Communications of the ACM*, 2004, 47(3): 120.

[4] Wang Z P, Hu H C, Zhang C H. On Achieving SDN Controller Diversity for Improved Network Security Using Coloring Algorithm[C]. *2017 3rd IEEE International Conference on Computer and Communications*, 2018: 1270-1275.

[5] Shacham H, Page M, Pfaff B, et al. On the Effectiveness of Address-Space Randomization[C]. *The 11th ACM conference on Computer and communications security*, 2004: 298-307.

[6] Sovarel A N, Evans D, Paul N. Where's the FEEB? the Effectiveness of Instruction Set Randomization[C]. *The 14th conference on USENIX Security Symposium - Volume 14*, 2005: 10.

[7] Cox B, Evans D, Filipi A, et al. N-Variant Systems: A Secretless Framework for Security through Diversity[C]. *The 15th conference on USENIX Security Symposium - Volume 15*, 2006: 105-120.

[8] Yao D, Zhang Z, Zhang G F, et al. A Survey on Multi-Variant Execution Security Defense Technology[J]. *Journal of Cyber Security*, 2020, 5(5): 77-94.
(姚东, 张铮, 张高斐, 等. 多变体执行安全防御技术研究综述[J]. *信息安全学报*, 2020, 5(5): 77-94.)

[9] Zhang C. Research on software security protection technology against control flow hijacking attack[D]. Beijing: Peking University, 2013.
(张超. 针对控制流劫持攻击的软件安全防护技术研究[D]. 北京: 北京大学, 2013.)

[10] Viega J, Bloch J T, Kohno Y, et al. ITS4: A Static Vulnerability Scanner for C and C Code[C]. *Proceedings 16th Annual Computer Security Applications Conference*, 2002: 257-267.

[11] Cowan C, Barringer M, Beattie S, et al. FormatGuard: Automatic Protection from Printf Format String Vulnerabilities[C]. *The 10th conference on USENIX Security Symposium - Volume 10*, 2001: 15.

[12] Padmanabhuni B M, Kuan Tan H B. Predicting Buffer Overflow Vulnerabilities through Mining Light-Weight Static Code Attributes[C]. *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, 2014: 317-322.

[13] Andersen S, Abella V. Data execution prevention, changes to functionality in Microsoft Windows XP Service Pack 2, Part 3: memory protection technologies. 2015. https://technet.microsoft.com/en-us/library/bb457155.aspx Bhatkar S, DuVarney D C, Sekar R. Address Obfuscation: An Efficient Approach to Combat a Board Range of Memory Error Exploits[C]. *The 12th conference on USENIX Security Symposium - Volume 12*, 2003: 8.

[14] Bhatkar S, Sekar R, DuVarney D C. Efficient Techniques for Comprehensive Protection from Memory Error Exploits[C]. *The 14th conference on USENIX Security Symposium - Volume 14*, 2005: 17.

[15] Xu H Z, Chapin S J. Improving Address Space Randomization with a Dynamic Offset Randomization Technique[C]. *The 2006 ACM symposium on Applied computing*, 2006: 384-391.

[16] Xu H, Chapin S. Address-Space Layout Randomization Using Code Islands[J]. *J Comput Secur*, 2009, 17: 331-362.

[17] Abadi M, Budiu M, Erlingsson U, et al. A theory of secure control flow[C]. *International Conference on Formal Engineering Methods*, 2005: 111-124.

[18] Abadi M, Budiu M H, Erlingsson Ú, et al. Control-Flow Integrity Principles, Implementations, and Applications[J]. *ACM Transactions on Information and System Security*, 2009, 13(1): 1-40.

[19] Walls R J, Brown N F, Le Baron T, et al. Control-flow integrity for real-time embedded systems[C]. *31st Euromicro Conference on Real-Time Systems. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik*, 2019, 133(2):1-24.

[20] Volckaert S, De Sutter B, De Baets T, et al. GHUMVEE: Efficient, Effective, and Flexible Replication[C]. *The 5th international conference on Foundations and Practice of Security*, 2012: 261-277.

[21] Hosek P, Cadar C. Varan the unbelievable: An efficient n-version execution framework[J]. *ACM SIGARCH Computer Architecture News*, 2015, 43(1): 339-353.

[22] Volckaert S, Coppens B, Voulimeneas A, et al. Secure and Efficient Application Monitoring and Replication[C]. *The 2016 USENIX Conference on Usenix Annual Technical Conference*, 2016: 167-179.

[23] Wang X, Yeoh S M, Lyerly R, et al. A Framework for Software Diversification with ISA Heterogeneity[C]. *23rd International Symposium on Research in Attacks, Intrusions and Defenses*, 2020: 427-442.

[24] Pan C X, Zhang Z, Ma B L, et al. Method Against Process Control-Flow Hijacking Based on Mimic Defense[J]. *Journal on Communications*, 2021, 42(1): 37-47.
(潘传幸, 张铮, 马博林, 等. 面向进程控制流劫持攻击的拟态防御方法[J]. *通信学报*, 2021, 42(1): 37-47.)

[25] Rad B B, Bhatti H J, Ahmadi M. An introduction to docker and analysis of its performance[J]. *International Journal of Computer Science and Network Security*, 2017, 17(3): 228.

[26] Casalicchio E. Container orchestration: a survey[J]. *Systems Modeling: Methodologies and Tools*, 2019: 221-235.

[27] Rosen R. Resource management: Linux kernel namespaces and cgroups[J]. *Haifux*, 2013, 186: 70.

[28] Kocher P, Horn J, Fogh A, et al. Spectre Attacks: Exploiting Speculative Execution[C]. *2019 IEEE Symposium on Security and Privacy*, 2019: 1-19.

[29] Lipp M, Schwarz M, Gruss D, et al. Meltdown: Reading kernel memory from user space[C]. *27th USENIX Security Symposium*, 2018: 973-990.

[30] Seaborn M, Dullien T. Exploiting the DRAM rowhammer bug to gain kernel privileges[J]. *Black Hat*, 2015, 15: 71.

[31] Espe L, Jindal A, Podolskiy V, et al. Performance Evaluation of Container Runtimes[C]. *CLOSER*, 2020: 273-281.

[32] B. Salamat, A. Gal, M. Franz. Reverse stack execution in a multi-variant execution environment[C]. *In Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, 2008: 258-261.

[33] Bhatkar S, Sekar R. Data Space Randomization[C]. *The 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008: 1-22.

[34] Akihiro Suda, et al. containERD CTL - Docker-compatible CLI for containerd, with support for Compose, Rootless, eStargz, OCIcrypt, IPFS. *https://github.com/containerd/nerdctl*. 03 Mar 2022.

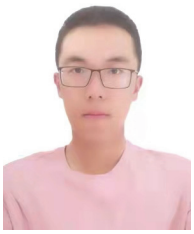
[35] Nadgowda S, Suneja S, Kanso A. Comparing Scaling Methods for Linux Containers[C]. *2017 IEEE International Conference on*

Cloud Engineering, 2017: 266-272.

[36] McKenney P E, Boyd-Wickizer S, Walpole J. RCU usage in the linux kernel: One decade later[J]. *Technical report*, 2013, 12(3): 1-12.

[37] Jaleel A. Memory characterization of workloads using instrumentation-driven simulation–a pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites[J]. *Intel Corporation, VSSAD*, 2010: 1-12.

[38] Wu J X. Research on Cyber Mimic Defense[J]. *Journal of Cyber Security*, 2016, 1(4): 1-10.
(邬江兴. 网络空间拟态防御研究[J]. 信息安全学报, 2016, 1(4): 1-10.)



刘子敬 于 2020 年在信息工程大学计算机科学与技术专业获得学士学位。现在信息工程大学计算机科学与技术专业攻读硕士学位。研究领域为主动防御。研究兴趣包括: 先进计算、内生安全。Email: lzj_arise@outlook.com

朱鹏喆 于 2020 年在电子科技大学软件工程专业获得学士学位。现在信息工程大学计算机科学与技术专业攻读硕士学位。研究领域为主动防御。研究兴趣包括:内生安全、移动目标防御。Email: 1196207482@qq.com

邬江兴 现任国家数字交换系统工程技术研究中心主任, 教授, 博导, 中国工程院院士。研究领域为信息通信网络、网络安全。研究兴趣包括: 主动防御、交换技术与宽带信息网络、高效能计算。Email: 17034203@qq.com

张铮 于 2006 年在信息工程大学计算机科学与技术专业获得博士学位。现任数学工程与先进计算国家重点实验室副教授。研究领域为网络安全、先进计算。研究兴趣包括:主动防御、高性能计算。Email: ponyzhang@126.com

席睿成 于 2019 年在郑州轻工业大学物联网工程专业获得学士学位。现在信息工程大学计算机科学与技术专业攻读硕士学位。研究领域为网络空间安全。研究兴趣包括: 主动防御、软件安全。Email: 370722031@qq.com