

多变体执行安全防御技术研究综述

姚 东¹, 张 铮¹, 张高斐¹, 刘 浩¹, 潘传幸¹, 鄢江兴²

¹ 数学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 软件和信息系统的快速发展在给人们生活带来诸多便利的同时,也让更多的安全风险来到了我们身边,不法分子可以很方便的利用无处不在的网络和越来越自动化、低门槛的攻击技术去获得非法利益。面对这种现状,传统被动式的安全防御已显得力不从心,更高的防御需求,促进了安全领域不断研究新的主动防御技术。这其中,基于攻击面随机化扰动的移动目标防御技术和基于异构冗余思想的多变体执行架构技术受到了广泛的关注,被认为是有可能改变网络空间游戏规则的安全技术,有望改变攻防双方不平衡的地位。本文对近年来多变体执行架构技术在安全防御方面的研究工作归纳总结,梳理了该方向的关键技术及评价体系。在此基础上,分析了多变体执行架构在安全防御方面的有效性,最后指出多变体执行架构技术当前面临的挑战与未来的研究方向。

关键词 多变体执行; 安全防御

中图分类号 TP309 DOI号 10.19363/J.cnki.cn10-1380/tn.2020.09.06

A Survey on Multi-Variant Execution Security Defense Technology

YAO Dong¹, ZHANG Zheng¹, ZHANG Gaofei¹, LIU Hao¹, PAN Chuanxing¹, WU Jiangxing²

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 The rapid development of software and information systems has brought many conveniences to people's lives, and at the same time, more security risks have come to us. Lawless elements can easily exploit ubiquitous networks and increasingly automated, low-threshold attack techniques to gain illicit benefits. Faced with this situation, the traditional passive security defense has become incapable, and the higher defense demand has promoted the continuous research of new active defense technologies in the security field. Among them, the moving target defense technology based on attack surface randomization perturbation and the multi-variant execution architecture technology based on heterogeneous redundancy idea have received extensive attention, and it is considered to be a security technology that may change the rules of cyberspace games, and is expected to change the offensive and defensive positions. This paper summarizes the research work of multi-variant execution architecture technology in security defense in recent years, and combs the key technology and evaluation system in this direction. On this basis, the effectiveness of multi-variant execution architecture in security defense is analyzed. Finally, we pointed out the current challenges and future research directions of multi-variant implementation architecture technology.

Key words multi-variant execution; security defense

1 前言

1.1 漏洞和软件同质化

在互联网和信息社会高速发展的今天,软件产品越来越丰富,功能越来越复杂,软件代码的数量也随之越来越庞大,因此不可避免地会出现各种各样的漏洞^[1]。

漏洞在小范围内被利用是可控的,但是由于网络的发展和普及使得漏洞可以快速、广泛地传播,而造成不可估量的后果,这其中软件同质化^[2-3]也是使漏洞可以广泛传播的一个因素。

目前大多数信息系统在部署时多采用市面上流行的架构,这些运行机制大致相似的系统在软件的选择和部署上也有很大的相似程序,这使得在一个

通讯作者: 张铮, 博士, 副教授, Email: ponyzhang@126.com。

本课题得到国家重点研发计划网络空间安全专项(No. 2018YFB0804003, No. 2017YFB0803204)资助。

收稿日期: 2019-09-12; 修改日期: 2020-02-01; 定稿日期: 2020-08-21

系统起作用的攻击或病毒可以很容易地被攻击者应用或传播到采用相同环境的系统。软件开发中使用集成第三方库、借鉴开源代码以及继承自身历史代码等代码重用方法也让很多软件具有相似的“基因”。同构的软件、相同的协议和标准虽然能够大大降低软件开发、维护的成本、促进信息的传播,但也大大降低了黑客攻击的成本,进一步为攻击范围的扩散提供了便利。比如,2014 年 BASH 被公布存在远程代码执行漏洞(CVE-2014-6271)^[4],可以通过构造环境变量的值来执行想要执行的攻击代码脚本,会影响到与 BASH 交互的多种应用,主流的 Linux 和 Mac OS X 操作系统平台,包括但不限于 RedHat、CentOS、Ubuntu、Debian、Fedora、Amazon Linux、OS X 10.10 等平台都受到了影响。

为了抵御、缓解或降低各种安全风险及其带来的损失,安全厂商和科研机构研究制定了各种各样的防御方法。如针对缓冲区溢出漏洞,有静态分析的检测和消除方法^[5-8],还有程序变形和动态检测^[9-14]以及通过修改硬件进行防御的方法^[15-18],针对字符串格式化漏洞的防御方法^[19-22]等。在内存中对可执行页面的限制可以阻止大多数代码注入式攻击^[23],动态污点分析通过对信息流的追踪可以识别内存破坏类攻击^[24],控制流完整性检测可以检测到进入了非法执行路径的程序^[25-28]。在这些防御方法中,有一些能够在代价和性能损耗很小的情况下完成对某些类型漏洞的防御,还有一些防御方法只能针对某一种漏洞进行防御。这些方法仍然属于“修补式”的防御,且容易被攻破,比如针对控制流完整性检测的防御就有很多方法可以绕过^[29-32]。另外,由于漏洞的不可避免和软件同质化现象的普遍,使得网络空间总会面临诸如勒索病毒等大范围安全风险的可能,而现有的防御方法还不能很好地解决这个问题。

1.2 基于多样化技术和多变体执行架构的安全防御

许多安全研究者注意到软件同质化为攻击者带来的便利,开始从增加软件多样性的角度去研究缓解攻击、限制攻击传播的有效策略,进而出现了大量的软件多样化技术^[33-39]。多样化技术相当于对以往攻击者利用的攻击面或系统的某些状态、属性进行了一定程度的“加密”,但由于“熵”空间的限制,要破解它们^[40,41]并不是不可能,再加上另辟蹊径的侧信道攻击等手段,使得单纯多样化的使用并不能保证软件个体的安全。如前所述,多样化的使用只是针对软件同质性,在攻击传播速度和规模方面起到了一定的限制作用。

在进一步的研究过程中,研究者们发现多变体执行中蕴含的异构和冗余执行思想不仅可以用来应对同质化带来的威胁传播风险,也可以用来发现安全风险^[42]。因此该技术被安全研究者们广泛借鉴到网络空间安全防御领域中,为网络空间安全防御带来了一种全新的主动防御思路。多变体执行架构使用软件多样化技术构建功能等价的变体集合,运行时对从集合中选择的每个变体都赋予相同的输入,再通过监视器监视每个变体的输出行为并检测它们之间的分歧。多变体执行架构是多样化技术解决安全问题的延续和补充,只要多样化的个体在一定程度上满足攻击面不大范围“相交”就可以作为变体的“源”加入执行框架。多变体执行并不追求用“加密”的方法来保护程序中某些可以被攻击者利用的信息,只要经过多样化处理过的程序变体在漏洞利用方面相互之间出现了空间或时间上的差异就可,这就使得某些“熵”空间不足、单独使用时易受攻击的多样化技术可以在多变体执行框架下、变体之间发挥协同防御的作用,如数据的表示、寄存器的设置、栈的方向,内存的布局等。攻击者使用的某个攻击方法必须同时对多变体执行架构下的所有变体都有效才能达到攻击目的。这样的执行架构极大的增加了黑客的攻击难度,同时又充分利用了软件多样化技术,对软件漏洞实现了真正意义上的主动防御。

1.3 论文结构

本文对近年来多变体执行架构在软件安全防御方向的研究进行了归纳总结。第 1 章主要介绍了由漏洞和软件同质化带来的一系列问题以及基于多样化技术和多变体执行架构的安全防御技术的提出;第 2 章主要梳理了多执行体冗余执行为解决安全问题所做的探索及主要的研究机构;第 3 章从一些主流的多变体执行框架入手,介绍了整体架构演变、变体同步粒度和监控机制和架构关系;第 4 章按多变体执行架构的组成,对其中的关键技术进行总结;第 5 章讨论了目前多变体执行框架的主要评价体系和方法;第 6 章对多变体执行防御技术的有效性进行了分析;第 7 章阐述了多变体执行面临的挑战与未来研究方向;第 8 章进行了全文总结。

2 多执行体冗余执行解决安全问题的探索

多执行体冗余执行其实并不是一个新概念,复制进程的结构很早就被提出并用于程序调试、错误容忍以及改善程序的可靠性,只是后来才被研究者用于解决软件的安全防御问题。由于某个软件的多执行体是基于对同一个软件进行多样化改造而完成

的,所以在软件安全领域常称这些冗余执行体为多变体。多变体运行最早出现在 Knowlton 在 1968 年发表的文章中^[43],被用于检测和定位程序的错误,具体做法是同时执行两个在逻辑上等价的程序,在程序具体的实现上是将代码分成小片段,然后对它们和数据段利用跳转指令进行重新排序,前提是保持程序在语义上与原始程序一致。CPU 在检测模式并行执行两个程序,并验证它们是否在执行语义上等价,变体本身不提供任何安全保证,但可以确保以很高的概率检测到诸如控制转换越界和“野”指针使用类错误。Berger 和 Zorn 后来提出了软件多个复制体进行冗余执行的框架^[44],在每个复制体的堆上使用随机化技术生成不一样的堆对象布局,以一定的概率提供内存安全保证,主要目的还是增强软件的可靠性和可用性,因此当执行框架检测到不一致时仍然允许其中一个变体执行,并假设发生在另一个变体上的不一致行为只是由于内存的某个错误导致的,而不是发生了攻击。该框架只对所复制进程使用的标准 I/O 和用户空间发起的小部分系统调用进行处理,因此如果攻击没有使用标准输出缓存则不会被其监视器发现。Cox 在 2006 年首次提出了一个较为完整的用于软件安全防御的多变体执行架构^[45],原理验证系统中的执行体采用了非重叠内存空间的多样化技术,实现了对依赖于空间信息类攻击的确定性防御。在软件多变体执行架构中,软件变体是以特定的攻击可检测为目的,用特定的变换机制对软件实施“变化”而生成的,Cox 架构中的变体可以针对不同攻击进行修改与扩展,如使用随机化技术对变体的指令、堆布局等进行改造。除变体生成方面的研究以外,针对整个架构,后来的研究主要是在 Cox 提出的基础架构上从三个方面展开的:1)对正常运行状态下变体某些不一致行为的处理;2)对多线程的处理;3)对架构本身安全性和执行性能方面的改进。

还有一些类似的研究使用架构设计多样化的方法提供安全性^[46-48],如使用不同的 Web 服务器软件(如 HTTP servers Apache on Linux and IIS on Windows)提供相同的 web 服务,并对它们的输出或使用预先定义的阈值范围进行比较,以此来发现可能出现的攻击行为。与软件多变体执行架构不同的是设计多样化使用目前已有的同类功能软件搭建异构化的执行架构,而不是基于同一软件经多样化“改造”生成变体,因此常把这种架构称之为基于 COTS(Commercial Off-The-Shelf)的异构体执行架构。设计多样性一般工作在软件栈的高层,它能

通过应用层面的语义检测到攻击行为,而多样化手段生成的执行体变体更多地是在软件栈的低层发现内存破坏类攻击或代码注入攻击。

本文选取的相关文献以多变体执行架构的主要研究机构和大学为主线,如美国的加利福尼亚大学 Michael Franz 教授团队和弗吉尼亚大学 Benjamin Cox 团队为代表的主要研究成果,还有欧洲以比利时根特大学和荷兰阿姆斯特丹大学相关团队为代表的研究成果。另外选取以上重点科研成果为基础展开的扩展和创新研究成果,内容基本涵盖了 2006 年以来该研究方向的主要研究成果,范围涉及多变体执行架构的原理、设计、实现和评估。

3 多变体执行架构的相关研究

多变体执行,顾名思义,是运行一组功能等价、结构各异的同一软件的多个变体。用多变体执行来解决软件安全问题的基本思想是并行运行同一程序的多个功能相同、结构各异的变体,以锁定步骤、同步执行的方式将输入分发给所有变体,由于变体在异构过程中攻击面在空间上发生了变化,而这种变化使得不同变体对正常的输入能够产生一致的输出,但是对攻击等恶意行为会产生不一致的输出,因此通过对变体的输出结果在检查点进行对比、检测就可以发现变体执行过程中出现的异常。多变体执行架构的抽象如图 1 所示。

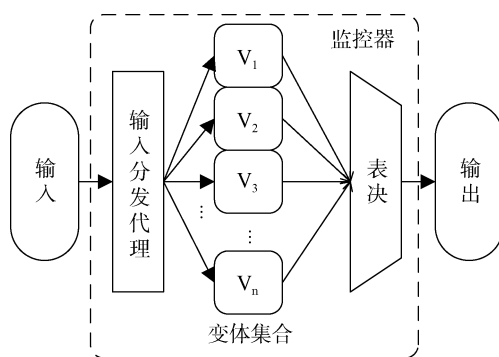


图 1 多变体执行架构

Figure 1 Multi-Variant Execution Architecture

对多变体执行架构的研究主要是围绕着整个架构中各关键组成部分的实现机制和协同工作方式展开的,如变体生成机制、监视器实现机制和表决算法等。主要解决的问题,一是增加安全防御的种类、范围和准确性,二是提高执行效率和架构自身的安全性,寻找执行效率和安全性之间的平衡点。还有一点需要注意的是保证对用户的透明,即任何时刻在用户看来与只有一个执行体工作的情形无异。在整个

架构中, 变体生成、输入和表决技术的改进和创新直接的体现在多变体执行架构的变化上, 因此在总结各组成部分的关键技术之前, 先概括一下近年来多变体执行架构的演变。

3.1 整体架构的发展和演变

Cox 等人^[45]在 2006 年首次针对软件安全问题提出了 N-Variant 执行架构, 如图 2 所示。该架构使用了内存空间的异构和指令集标签化技术来生成变体。每个变体以及负责分发和监视的模块都采用独立进程的方式工作。通过一个原理验证系统作者发现了实现多变体架构时要注意的几个问题: 1) 变体的隔离范围, 增大隔离范围, 如将变体置于不同的主机或不同的进程, 会减少某个变体受到攻击时给其它变体带来的影响, 但同是也会增加变体及分发和监视之间的通信环节, 而这些通信环节又会引入新的攻击面, 也会对系统性能带来一些损失; 2) 监控粒度的选择, 监控粒度过粗会引起误报率的上升, 如 web 服务器响应粒度下的时间戳和 IP 地址, 反之, 监控粒度过细则会对系统性能带来较大的影响。针对原理验证系统中发现的问题作者提出了基于对 Linux 内核进行修改实现的多变体执行架构。在该架构下, 所有变体、分发器和监视器都运行在同一个平台下, 它们之间的隔离是通过独立进程来实现的。监视器的监视粒度设定在系统调用级, 为了提高执行效率, 所有系统调用被分为共享、反射和危险系统调用三大类, 每一类有相应的处理方式。监视器位于进程的内核空间, 这样可以提高多变体架构的执行效率。作为网络空间安全防御领域首次提出的较为完整的多变体执行架构, 该篇文章奠定了该领域研究的基础。同时, 作者也提出了多线程、信号处理等需要进一步研究和解决的问题。

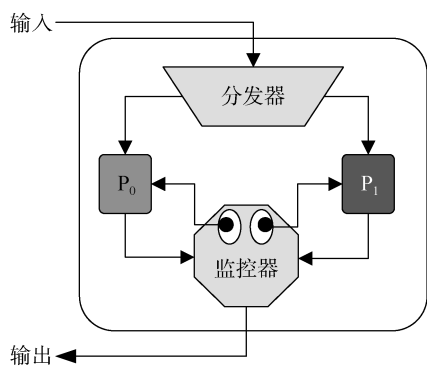


图 2 N-变体系统框架

Figure 2 N-Variant System Framework

同年, Berger 和 Zorn^[44]提出了通过软件多个复制体进行冗余执行的框架—DieHard, 如图 3 所示。

它的变体生成策略是在每个执行体的堆上使用随机化技术生成不一样的堆对象布局。变体和监视器都是以独立进程的方式工作。变体通过管道从 DieHard 主进程接收输入, 然后将输出写入一块由所有变体共享的内存区域, 该区域中每个变体有自己的专属缓冲区, 裁决进程在同步点对所有变体的输出缓存进行比较, 在正确的情况下将结果输出。DieHard 只实现了对 Unix 下使用标准输入、输出的命令的监控, 但证明了该架构能够提供内存安全保证。

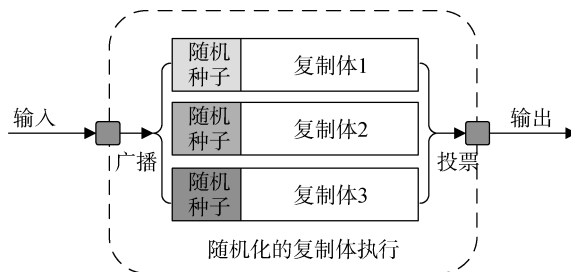


图 3 DieHard 系统架构

Figure 3 DieHard System Architecture

2007 年, Novark 等人^[49]在 DieHard 的基础上进行扩展提出了 Exterminator, 其系统架构如图 4 所示。该架构中使用不同的随机化种子通过对程序的堆进行多样化处理生成变体, 并且通过表决算法可以定位内存发生错误的位置和大小, 同时可以在运行时生成补丁来修正检测到的错误。这个创新首次为多变体执行架构引入了动态反馈的思想。

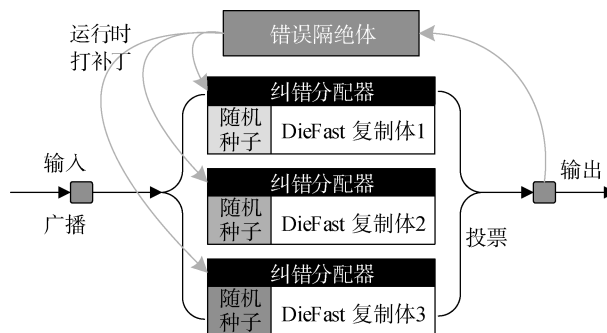


图 4 Exterminator 系统架构

Figure 4 Exterminator System Architecture

同年, Aydan 等人^[50]提出了 TightLip。TightLip 检测和防御恶意行为的关键机制是并行运行要保护的原始进程与其在沙箱中的一个拷贝进程, 且沙箱中的进程只是在原始进程要处理“敏感数据”时才生成并开始与原始进程并行工作, 通过在系统调用处的同步与参数比对来发现是否有“敏感数据”的泄露。对“敏感数据”的定义是通过一个预定义的扫描器来完成查找和标记的。拷贝进程与原始进程

大致是相同的,但传递给拷贝进程的输入是经过多样化处理的。对大多数程序而言,多样化的输入可能会引起程序控制流的不一致,因此在检测时可能会出现误报。该架构引入了沙箱对执行体进行隔离,但沙箱本身可能会引入新的攻击面,且在一定程度上会降低执行体的执行效率。为了克服这个缺点,在具体实现上,它需要对操作系统和文件系统进行部分改动,但这又增加了部署的难度,但是文章提出的从保护敏感数据角度出发来进行安全防御的思路以及它的进程隔离方式后来被很多研究者借鉴。

2009 年, Weatherwax 从数据保护的角度提出了一整套构建多变体执行架构的方法^[51],并分析了构建多变体执行架构过程中易于引入的四类漏洞,同时给出了解决方案。同年, Salamat 等人^[52]提出 Orchestra 架构,如图 5 所示。该架构的设计原则是不增加原有系统的可信基(Trusted Computing Base, TCB),因为随着代码量的增大,错误也更容易发生。Orchestra 的监视器以独立进程的方式完全运行在用户空间,监控机制是使用 ptrace API 来对变体的系统调用进行拦截和比对,整个构架的实现不需要对操作系统进行任何改动。监视器在拦截到系统调用后根据事先对系统调用的分类决定是由自己执行该系统调用还是由变体来执行。该架构首次将监视器与执行体合二为一,让监视器承担了一部分执行体的工作,这可以解决一部分由所有变体执行某系统调用产生的不一致问题,同时还提高整个架构的执行效率。

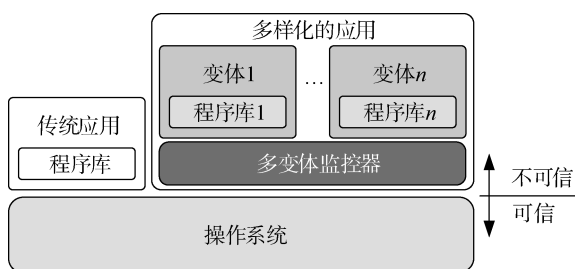


图 5 Orchestra 系统架构

Figure 5 Orchestra System Architecture

2013 年,比利时根特大学的 Volckaert 等人^[53]提出了 GHUMVEE,并于 2016 年在 GHUMVEE 的基础上提出了 ReMon^[54],其架构图如图 6 所示。GHUMVEE 之前提出的执行架构在对输入进行拦截时,都假设所有的输入要么源于系统调用接口,要么可以在系统调用接口可被拦截,GHUMVEE 通过例举 x86 处理器下的某些指令以及 Linux 系统中的虚拟调用否定了这一假设,同时提出了用共享内存解决程序中的“隐含”输入在多变体架构下带来的

不一致问题。GHUMVEE 同样使用 ptrace API 在系统调用这个粒度对变体进行监控,但它加入了代理模块对可逃离监控的输入进行了处理,同时还使用“记录+重放”技术提供了对多线程的支持。ReMon 架构上最大的改变在监控机制上,它采取了平衡性能与安全性的思想,将进程内监控与进程外监控相结合(在进程外设置一个单独的监视进程对与安全紧密相关的重点系统调用进行监控,在进程内对其他系统调用进行监控)实现了性能的大幅度提升。

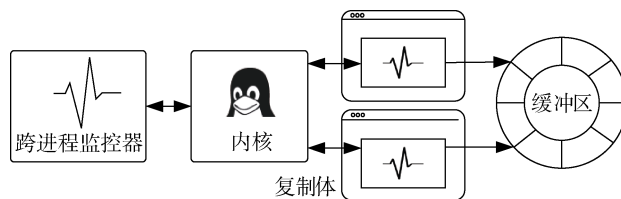


图 6 ReMon 系统架构图

Figure 6 ReMon System Architecture

2015 年, Hosek 等人^[55]提出了 VARAN,不同于以往的多执行体架构,VARAN 的事件流设计基于 Ring Buffer,是完全异步的,不支持同步检测策略,因为 VARAN 关注的是软件的可靠性而不是安全性,类似的架构还有 Tachyon^[56]和 MX^[57]。VARAN 的事件流体系结构与 GHUMVEE 记录和重放系统有着相似之处,其中执行过程被连续记录到日志中,此日志稍后可用于在本地或其他计算机上重新执行应用程序,并可选择在重放期间部署其他检查,可用于安全审核等离线安全应用。VARAN 对多执行体架构最大的贡献在于它的事件流异步共享机制,后续的很多架构在实现时都借鉴了 VARAN 的思想来实现变体之间信息的传递和共享。

2016 年,阿姆斯特丹大学的 Koning 等人^[58]针对现有多变体执行架构中系统调用监控机制运行效率低和变体生成策略僵化等问题提出了可自定义安全策略的多变体执行架构—MvArmor,如图 7 所示。该架构可通过环境感知来生成变体,与之前的架构相比在灵活性、有效性、架构本身的安全性和执行效率上有较大的改进。MvArmor 架构中的所有模块都被置于一个使用 Intel CPU VT-x 硬件加速的虚拟化系统—Dune^[59]中,通过虚拟管理层和对外部系统中的 vDSO 的重写实现了对所有系统调用的有效拦截。MvArmor 将变体分在主变体和从变体,在运行时由安全策略管理模块决定哪些系统调用由主变体执行,哪些所有变体都要执行。变体之间以及变体与监视器之间的通信采用了类似 VARAN 的结构。MvArmor 依靠硬件支持的虚拟化安全地将监视器与不受信任

的变体执行隔离开来,同时以普通用户权限让监视器工作在 libOS 的 Ring0 中,进一步提升了多变体执行架构的运行效率。

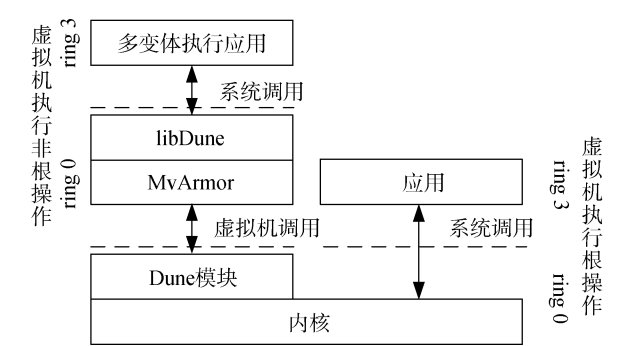


图 7 MvArmor 系统架构图
Figure 7 MvArmor System Architecture

Lu K 在 2018 年提出 BUDDY^[79]的多变体执行架构,首次将同步点设置为 I/O 写操作,在攻击者向外发送数据的时候进行监控和检测,相对于系统调用级的同步,它减少了监视器同步比较的次数,性能损失更小。不足之处是只能防御信息泄露类攻击,不过该方法还可应用于脚本级应用。

2019 年, Voulimeneas A 首次提出了分布式异构平台 DMON^[80]的多变体运行方案,将同样的程序分别在使用 Intel 和 ARM CPU 的平台上编译,并在使用变体技术处理后共同在一个多变体架构下执行。异构平台的方案带来了更好的安全性,但它的实现难度和成本也随之增加,而且需要额外的手段来解决通信成本问题。

同年, Österlund S 提出了操作系统内核多变体执行架构 kMVX^[81],如图 8 所示。它主要是用来防御通过内核漏洞发起的内核信息泄露攻击,该研究为从操作系统级解决软件安全问题提供了思路。

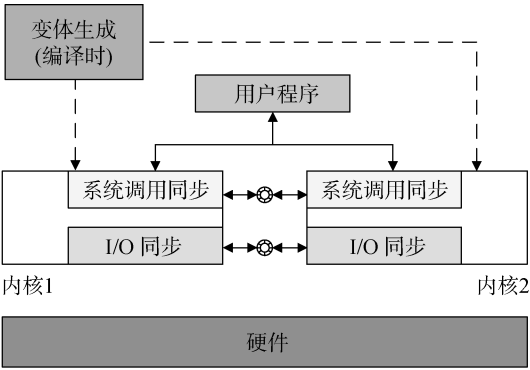


图 8 kMVX 系统架构图
Figure 8 kMVX System Architecture

3.2 变体同步粒度

多变体执行架构对变体实施检查的点一般称为检查点或同步点,它的监控机制可以从不同的粒度上去实现,在目前的研究中有应用级、函数级、代码块级、语句级、系统调用级和指令级。粗粒度的应用级只比较两个变体的最终输出是否一致(如 BUDDY),而到细粒度的指令级,则可能需要借助硬件来确保变体同步执行的指令语义上的等价。不同的监控粒度大多数情况下并不影响对安全威胁的检测,但是它决定了捕获安全威胁的时机。对同一类的监控对象,有的多变体执行架构选择对所有事件进行同步监控,还有的根据预先定义的规则对选择出的部分事件进行同步监控(如 MvArmor 中的策略单元可以自定义要监控的系统调用)。由于监控事件数量的不同,粗粒度通常比细粒度的执行效率更高,但同时也会在一定程度上影响检测的准确率和完整性。之所以有不同实现方式的选择,主要还是与安全等级的需求和系统执行性能有密切关系。表 1 是多变体执行常见架构的实现粒度以及性能和错报率的对比。

表 1 实现粒度总结
Table 1 Implementation Granularity Summary

多变体系统	同步点	延时/性能损失	错报率
N-variant ^[45]	所有系统调用	17.6%~48%(延时)	高
VARAN ^[55]	所有系统调用	14.2%(性能损失)	低
DCL ^[73]	大多数系统调用	6.37%(性能损失)	高
TightLip ^[50]	所有系统调用	5%(性能损失)	高
ShadowExe ^[89]	所有输入	>100%(延时)	高
ReMon ^[54]	部分系统调用	2.4%~34%(性能损失)	低
MvArmor ^[58]	规则定义的系统调用	55%(性能损失)	低
LDX ^[90]	输出	平均 6.08%(性能损失)	高
Detile ^[74]	脚本语言字节码	4%~28.5%(性能损失)	低
BUDDY ^[79]	I/O	4%	低

3.3 监控机制与架构的关系

监控机制的设计是整个多变体架构设计中最重要的一环,是影响整个架构安全性与性能的关键部分。若监控机制的设计偏向安全性,整个架构的执行效率往往会受到较大的影响。反之,执行效率高的监控机制在安全性方面就会有所缺失。监控器在执行监控的同时还要负责多个执行体的同步以及整个架构的控制流与数据转换,最终通过裁决环节发现意图突破多变体执行防御架构的攻击。

监控器可能包含在多个变体进程中,在同步点通过共享内存的通信实现监控和变体之间的同步,这种方式称为进程内监控器(In-Process Monitor, IPM)。在这种监控方式下,变体通常有主、次之分,主变体完成大部分监控器的工作,以及对外输出。监控器还可能是一个单独的进程,通过程序调试相关的 API 对变体进行监控,这种实现方式称为跨进程监控器(Cross-Process Monitor, CPM)。使用 CPM 方式的多变体执行架构中,各变体地位平等,执行相同的工作,由 CPM 选择由哪个变体负责最终的对外输出。CPM 的地址空间与变体地址空间构成进程之间的硬件强制边界,可以避免受攻击的变体直接对监控器产生影响,缺点是交互延迟较大。

监控机制的具体技术详见 4.2 节。

4 多变体执行架构中的关键技术

多变体执行架构中涉及到的关键技术有变体生成技术、监控技术、分发与裁决技术、控制流转移和数据交换技术,这四种技术在架构中环环相扣、相辅相成。变体生成技术是系统运行时利用监控技术发现攻击的基础,而分发与裁决技术则为架构提供功能正常与安全输出的保证,控制流转移和数据交换技术则是多个异构变体执行中的关键环节。

4.1 变体生成技术

从一个软件的生命周期来看,生成它的变体可以在一个软件的开发、编译、链接、安装、加载执行时进行实施。在不同的阶段实施有不同的优势和缺点。开发时生成软件变体,即 N 版本编程^[60-61],目前除了在一些特殊行业(如航空航天)会使用,因其高昂的设计和开发成本在其他行业并不适用。而在软件编译和链接时生成它的多变体本质上是针对编译器进行改造或改变编译参数,它的优点是可以根据便捷的生成同一软件的多个变体,同时可以针对多种硬件平台进行编译;缺点是无法应用于没有源码的程序。在软件安装时使用多样化技术多数是为了在程

序加载时生成变体做准备,一般要用到反汇编技术和静态二进制重写^[28,62]。程序加载时生成变体多数情况下需要结合操作系统的一些随机化技术如 ASLR^[63],它的弹性空间比较大,通过与动态二进制重写技术^[64-65]的结合可以适用于没有源码程序的变体生成。虽然变体的生成方式各有千秋,但总的指导原则是让生成的所有变体对所防御的威胁在攻击面上呈现出最大的无关性(理想状况是攻击面正交)。

从变体生成的时机来看,在多变体执行之前使用各种多样化技术对软件(源码或二进制文件)进行处理预先生成变体可以减少运行时生成变体所需的时间,但它的不确定性也因此大打折扣。

相对来说,在软件运行时实时生成变体带给攻击者的不确定性更大,另外还可以与操作系统之间紧密配合(如随机化系统调用号,随机化动态加载库的入口点,随机化栈的位置或反转增长方向,随机化堆中对象的布局等),结合安全策略和实时运行环境感知(如 MvArmor 中的变体生成技术)等指导手段生成最符合防御目的的变体,这一点也更加符合主动防御的思想。

目前常见的变体技术有如下一些:

1) 反向栈

反向栈的设计本身在单个变体中无法阻止针对栈的攻击,但是在多变体执行环境中可以让针对栈的攻击在不同的变体中无法改变不同栈中同样偏移位置的数据^[52,82]。

2) 指令集随机化

指对机器指令进行随机化的加密,并在 CPU 要执行指令前进行解密。比如使用简单的 xor 指令对机器指令进行处理。单独使用这种变化可以让外部注入的代码无法执行,但是单独使用时无法防御攻击者对栈或堆上变量的修改,以及对控制流的改变^[36]。

3) 堆布局随机化

它主要用来阻止攻击者针对堆的溢出攻击,主要原理是让程序在堆上的分配操作无法预测,DieHard 中的变体所使用的就是这种技术。

4) 栈基址随机化

这种随机化技术实际上在目前流行的操作系统中都已经实现的技术,但是由于随机化空间熵不足很容易被攻击者绕过。

5) 栈保护

这种保护措施是在栈上的缓冲区与返回地址或指针之间插入一个称为“canary”的监督变量,当攻击者想要通常缓冲区溢出来修改返回地址或指针的值时也会改变“canary”的值,这样只需要在函数返

回处或使用指针之前检查“canary”的值就可以知道是否有攻击行为发生。不过在单独使用时,攻击者可以绕过这种保护措施对返回地址或指针进行修改^[83],在多变体环境中可以为不同的变体设置不同监督值来抵御这种攻击。

6) 系统调用号随机化

这项技术与指令集随机化类似,旨在让注入代码无法正确的发起想要的系统调用,它的不足是系统调用号随机的范围也是有限的,因此可以被暴力破解。

7) 库函数名随机化

某些攻击要直接调用库函数。通常情况下,相似的操作系统在载入共享库时总是会将其映射到相同的虚拟地址,这让攻击者很容易使用。将库中的函数名进行随机化能有效的防止攻击者对库的使用。

具体实现在这里不再赘述,可参阅文献[66]。

4.2 监控技术

多变体执行架构中的监控组件在整个架构中起着举足轻重的作用,虽然通常将其命名为监控器(Monitor),但它的工作除了监控之外经常还要负责变体的同步和调度等工作。

4.2.1 监控器类型

由前文所述的 IPM 和 CPM,加上监控器可能运行在内核空间(Kernel Space, KS)或者用户空间(User Space, US),在目前的架构实现中其类型主要有如图 9 所示的四种:

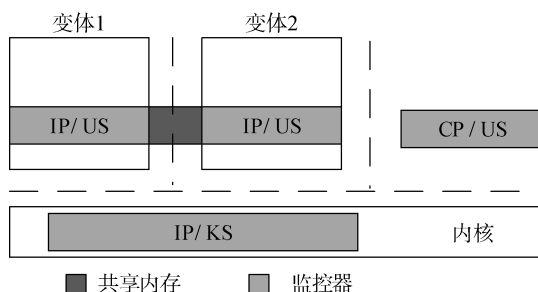


图 9 监控器类型

Figure 9 Type of Monitor

1) IP/US: 监控器与变体处于相同的进程地址空间,并且拥有相同的特权级别,它们之间使用分支转移指令来实现控制权的转移,这一做法很高效,但并不安全,因为一旦攻击者控制了变体,就可以在不触发监控器的情况下发起系统调用。

2) IP/KS: 监控器与变体处于相同的进程地址空间,但监控器拥有比变体更高的特权级,因此变体无法直接与监控器发生控制权的转移,它只能通过

内核的系统调用或陷阱处理器以及 HOOK 来触发,这种方式相对来说比 IP/US 的方式更为安全。

3) CP/US: 监控器与变体位于不同的进程空间,它们的交互要通过操作系统的调试接口,即由监控器来扮演调试者的角色。这种实现方式更加安全,但它的交互成本更大。

4) CP+IP/US+KS: 即 ReMon 的实现方式,它在内核空间有一个类似系统调用代理的组件,这个组件可以根据预先定义好的策略来决定某个系统调用交由 CP 还是 IP 来处理,这种方式可以在安全和高效之间进行更好的选择。

4.2.2 监控器实现方法

目前为止,基于可加载的内核模块(Loadable Kernel Module, LKM)实现的监控器拦截系统调用是最有效的,因为它不需要额外的上下文切换。Cox 等人^[45]在其提出的 N 变体系统中就是基于这种方式来实现的。但这种方法的问题是监控器需要完全运行在内核中,这会增加 TCB 的大小,而且部署不太容易。如果监控器出现任何问题,则会对整个系统有致命的威胁。

另一种方法是基于 ptrace API 实现监控器,主要有 Salamat 等人^[52]实现的 Orchestra 系统、Cavallaro 等人^[69]实现的原型验证系统、Volckaert 等人^[53]实现的 GHUMVEE 系统、Hosek 等人^[57]实现的 Mx 系统、Maurer 等人^[56]实现的 Tachyon 系统等。ptrace 是 UNIX 系统自带的一个系统调用,它能够跟踪进程的执行并让一个进程能够控制另一个进程。虽然使用 ptrace 实现监控器的多变体执行系统易于部署,但是它也在监控器与其要追踪的进程之间引起了过多的上下文切换,在一定程度上影响了多变体系统的执行效率。同时,使用 ptrace 的实现方式由于监控操作之间存在较大的延迟而容易受到 TOCTOU 攻击^[70]的影响。

为了提升多变体执行效率,有研究^[71]提出了基于二进制重写机制实现监控器的方法,通过重写二进制文件替换系统中执行系统调用的指令(x86 上的 \$0x80 和 x86-64 上的 syscall)来构建监控器,相对于 ptrace 的实现方式,它在执行时的系统开销较小。如 Hosek 等人提出的 VARAN 框架^[55]。该方法将监控器放在每个变体内部,能够极大的减少跨进程的上下文切换,但是该策略并不能很好的作为面向安全应用的监控器解决方案,因为攻击者可以很轻松的改变一个进程内的监控器状态,并通过这些来绕过系统调用拦截和差异检测。

以保证安全和提高执行效率为目标,监控器的

研究也在不断的改进和创新。如 Koning 等人^[58]利用 Intel CPU 提供的硬件虚拟化技术设计的 MvArmor 框架, 让监控器能够在进程内监控的同时安全使用所需的内核级特权指令, 但其使用的虚拟层 libOS—Dune^[59]支持维护较少且不是线程安全的, 所以限制了它的推广部署, 但其提出的解决思路很值得研究人员参考借鉴。Volckaert 等人^[54]在之前基于 ptrace 的 GHUMVEE 系统中, 参考 VARAN 而设计了新的多变体监控系统—ReMon。其结合了跨进程监控器和进程内监控器的优点, 在获得较好的安全性的同时也有接近 VARAN 的性能。

监控器中如何针对不同的场景来决定性能损耗与安全防护还需要进一步研究。

4.3 分发与裁决技术

分发技术决定多变体系统能够正确、透明的运行, 而裁决技术则决定系统能否正确发现异常。

多变体系统中的分发模块将被定义的需要由多变体共同处理的输入分发到各个变体, 裁决模块在变体执行前、后对输入及输出按预定规则进行裁决。这两个模块可能出现在多变体执行架构的设计中, 但在实现时通过都集成在监控器中。

大部分多变体执行架构都将系统调用作为检查点, 在执行系统调用时, 由于要保证每个变体中的一致性, 避免多次读写而引起之后的裁决出现假阳性等情况。通常将系统调用按照不同的类型进行分发, 同时将各个变体运行模式设计为 master-slave 或者 leader-follower。目前基本上可以将系统调用类型分为只能执行一次的系统调用(如 I/O 相关)和敏感系统调用(如 exec 相关)。敏感系统调用需要每个变体的执行, 同时监控器对这些系统调用的参数进行裁决, 若执行结果一致则通过裁决, 否则视为攻击。而只能执行一次的系统调用, 通过监控器将这些系统调用分给 master/leader 变体执行, 之后将执行的结果再同步到 slave/follower 变体上, 从而避免了各个变体之间出现异常的分歧。分发技术的设计和实现则与多变体系统的上述归约紧密相关。

裁决技术可根据监控器的类型来进行分类, 主要可分为进程内裁决和跨进程裁决, 同监控器一样, 跨进程的裁决仍然是较为安全的选择, 因为地址边界可以隔离处于异常状态的变体。而在进程内裁决则更容易受到内部进程的影响, 从而被攻击者操纵。

另一个重要的部分就是裁决算法, 目前的系统大都使用最简单的大数裁决, 但后来的研究发现它存在攻击逃逸的风险。因此, 裁决算法的研究越来越

受到研究者的关注, 如何设计能够快速、准确地发现、定位攻击同时避免逃逸漏洞的裁决算法是未来的研究目标。

4.4 控制流转移和数据交换技术

多变体执行架构中另一个重要技术就是变体与监控器之间的控制流转移和数据交换, 以及变体之间的数据交换。

1) 监控器与变体之间的控制流转移

监控器与变体在变体运行过程中触发了同步事件或处理器发生了异常时都需要进行交互, 这种交互意味着在整个执行架构下进行控制流的转移, 在目前的架构设计中, 相对来说 IP/US 模式的交互效率较高, 因为监控器和变体位于同一个进程空间, 但这种方式的安全性也最低。对于独立进程监控器的实现方式, 需要使用系统的 DEBUG 接口来完成控制流的转移, 这种方式的安全性相对比 IP/US 的方式要高, 但同时也带来了较大的性能损失。后来出现的 ReMon 就是为了在架构设计上对安全性和性能做出较好的折衷, 因此选择了 CP+IP/US+KS 的方式。

2) 监控器与变体之间运行时的数据交互

监控器与变体之间不仅有控制流程的转移, 同时还要进行数据的交互, 如变体发生的系统调用号和调用参数等信息。对于 IP/US 模式, 监控器与变体共享相同的虚拟内存空间以及寄存器的内容, 因此在信息交换时不需要发生上下文切换, 是所有架构中性能损失最小的一种方式。IP/KS 模式下, 则需要在特权级发生切换时对寄存器内容进行拷贝, 在切换频繁时会带来不小的性能损失。相对来说, 性能损失最大的还是独立进程监控器(CP)的实现方式。

3) 监控器与监控器之间的数据交换方式

对 IP/US 的实现方式, 它的监控器位于每个变体进程当中, 在监控器拦截了系统调用之后, 所有变体的监控器之间要交换信息来确认是否变体都处于同步状态(即以同样的参数发起了相同的系统调用), 它们之间的信息传递方式通常是建立一个所有变体都共享的内存区域, VARAN 中提出的 ring buffer 就是建立这种共享内存的一种实现方式, 后来的研究中大量的借鉴了这种方式。

实际上, 在构建多变体执行架构的过程中, 无论采用哪种具体方式都或多或少的引入了额外的信息交换通道, 而这些新引入的信息通道都有可能成为攻击者新的攻击目标。在攻击者对目标的运行架构不了解的情况下, 由于多变体执行架构对外是透明的, 即在用户看来与使用单执行体的情形类似。而一旦攻击者有了对目标执行架构的先验知识, 那么

架构在设计和实现上的缺陷以及新引入的信道都会成为攻击者新的攻击目标。目前已经有研究表明使用侧信道的方式可以对多变体架构中的通信环节实施攻击, 并成功的窃取系统中的数据。

另外, 监控器同时还要负责处理一些多变体执

行环境中常见的非攻击或恶意行为引起的非一致问题, 具体请参考文献[53]。

4.5 多变体执行架构总结

表 2 是对常见的多变体执行架构在关键技术等方面的总结。

表 2 多变体执行架构总结

Table 2 Multi-variant execution framework summary

多变体执行架构	变体生成技术	检查点	监控器实现	监控器类型	测试集
N-variant ^[45]	地址空间分区, 指令集标签化	系统调用	内核	IP/KS	Apache
DieHard ^[44]	堆布局随机化	I/O	-	CP/US	SPEC CPU2000, Squid Web 缓存服务器
Cavallaro ^[72]	非重叠地址空间	系统调用	ptrace	CP/US	thttpd
Exterminator ^[49]	堆布局随机化	I/O	-	CP/US	SPEC CPU2000, Squid Web 缓存服务器
Orchestra ^[52]	改变栈的生长方向	系统调用	ptrace	CP/US	SPEC CPU2000, Apache, Snort, Find, Tar, md5deep
Tachyon ^[56]	多个程序修正	系统调用	ptrace	CP/US	cURL, mplayer, php5, ncompress, htget, gs, glftpd, socat, corehttp, compress, primegaps, mencoder, lighttpd, thttpd
GHUMVEE ^[53]	完全支持 ASLR	系统调用	ptrace	CP/US	SPE CPU2006, gcalctool, kcalc, LibreOffice
Mx ^[57]	多个程序修正	系统调用	ptrace	CP/US	SPEC CPU2006, GNU Coreutils, Redis, Lighttpd
DCL ^[73]	不相交的代码布局	系统调用	ptrace	CP/US	SPEC CPU2006, gcalctool, kcalc, LibreOffice, MPlayer, CVE-2013-2028, CVE-2010-4221, CVE-2012-4409, CVE-2014-0749
VARAN ^[55]	多程序修正版本	系统调用	二进制重写	IP/US	SPEC CPU2000, SPEC CPU2006, Apache, Lighttpd, thttpd, Nginx, Redis, Beanstalkd, Memcached, Microbenchmarks
ReMon ^[54]	完全支持 ASLR, 不相交的代码布局	系统调用	ptrace+内核	CP+IP/US+KS	SPEC CPU2006, SPECint 2006, SPECfp 2006, PARSEC, SPLASH, Phoronix, Apache, thttpd, Lighttpd, Beanstalkd, Memcached, Nginx, Redis
MvArmor ^[58]	不重叠的地址空间, 不重叠的堆分配偏移	系统调用	Dune	IP/US	SPEC CPU2006, Microbenchmarks, Nginx, Lighttpd, Bind, Beanstalkd, CVE-2004-0488, CVE-2014-0160, CVE-2014-0195
Detile ^[74]	重新随机化地址空间布局	字节码	解释器	CP/US	Internet Explorer, Firefox, CVE-2011-1346, CVE-2014-6355, CVE-2014-0322, CVE-2015-0061
BUDDY ^[75]	分区地址随机化, 堆栈随机填充	I/O write	内核	IP/KS	SPEC CPU2006, Apache, Nginx, OpenSSL, PHP, Lighttpd, Orzhhttpd, Microbenchmarks

5 主要评价体系

现有文献对多变体执行技术的评价体系主要包括三个方面: 功能性、安全性和性能。

5.1 功能性评价

对多变体系统的功能性评估主要有两个目的: 1) 检验该多变体系统的功能完备性; 2) 验证该多变体系统不会引起假阳性错误。

目前检验功能完备性主要是从交互服务类应用入手, 如 Web 服务程序。评估时在多变体系统中运行多种服务器程序, 包括: Apache、Nginx、Lighttpd、Orzhhttpd 等等, 检验这些程序是否能够在多变体系统中正常运行, 以及是否能正常的与用户进行交互。检查的另一个主要目的是看在多变体系统中运行的服务器程序是否对外透明, 即与单一服务器程序表现一致。之后, 从多方面检查该多变体系统的兼容性,

验证该多变体系统不会在正常的操作下被检测到不一致行为,引起监控器的误判。同时还要验证该系统能否对防御目标进行有效防御。

5.2 安全性评价

安全性对于多变体系统是最重要的特性。对多变体系统的安全性评估可以从两个方面入手: 1) 基于现有漏洞对运行在多变体系统中的应用进行相应类型的攻击验证; 2) 研究者构造漏洞对运行在多变体系统中的应用进行攻击验证; 3) 检验分发机制、通信机制和表决算法本身是否存在可被攻击者利用的设计缺陷和漏洞。

目前研究者在验证其系统安全性时, 会首先建立威胁模型, 针对不同的威胁模型, 设计攻击向量, 一般从权威的漏洞数据库, 比如 CVE (Common Vulnerabilities and Exposures) 中获取相应应用的漏洞, 或者自己对应用构造相应漏洞。之后对应用发起如缓冲区溢出、信息泄露、代码复用、远程代码执行等攻击。如若在单变体系统或独立运行时能够攻击成功, 则证明攻击有效; 然后在多变体系统中运行应用, 若应用仍能在发起攻击的情况下正常运行, 则证明防护有效, 即验证了多变体系统的安全性。

在多变体架构设计和实现中引入的缺陷和漏洞不属于多变体架构安全防护理论本身的问题, 但设计和实现的不当也会给攻击者留下新的入侵路径, 因此在安全评估时除了对理论安全有效性进行评估以外, 还要留意架构设计和实现中的缺陷。

5.3 性能评价

多变体系统对于性能的要求也是不可或缺的, 一味追求在时间和空间上全面强化的内存安全措施会带来很大的性能开销^[76-77], 这会使多变体执行很难实际部署应用。目前研究者对性能进行评估时, 主要从 CPU 密集型、I/O 密集型以及系统调用周期三方面入手。

对 CPU 密集型的基准测试主要是考察该系统的计算能力。目前的研究者一般使用 SPEC CPU 2000 或者 SPEC CPU 2006 来进行测试, 也会考察 CPU 在极限状态下和一般状态下多变体系统的计算能力影响。

对 I/O 密集型的基准测试主要考察该系统的输入输出能力, 是否有较大的延迟, 能否提供正常的服务。目前来说, 主要使用一些例如 Apache、Nginx、Lighttpd 等 Web 容器承载网站, 并使用一些 Web 性能测试工具, 例如 ApacheBench、WebBench, 通过控制工作进程数、并发连接数、访问页面大小等因素, 来考察请求往返时间、数据传输率等性能指标, 从而进行评估 I/O 负载。

而对系统调用周期的微基准测试则是为了测

试监控器对系统调用拦截处理的影响, 来评估负载的影响大小, 一般选取几个常用的系统调用来进行测试。

6 多变体防御有效性分析

6.1 原理分析

该部分从宏观的多变体系统来分析其在防御攻击时的有效性。从多变体系统的组成来看, 其拥有一组内部结构不同、功能等价的变体, 可称为功能等价的异构变体, 同时该组变体为冗余执行, 则多变体系统也可称之为一个异构冗余系统。

若一个多变体系统能够正常运行, 且能在异常发生时发现攻击, 则需满足以下几个前提: ①变体 $V_1 \sim V_i$ 都具有独立完成给定任务的能力; ②变体 V_i 能正确完成任务是大概率事件; ③变体 V_i 之间不存在任何协同或协作关系; ④不排除多模大数表决结果是错误的可能。由此作出以下形式化推论。

假设 IPO 系统中存在变体集合 $V = \{V_i\}_{i=1}^n$ 和输入矢量集合 $I = \{I_j\}_{j=1}^m$ 变体 V_i 正确地响应输入矢量 I_j 时, 得到的输出矢量记为 R_{ij} , 其中 R_{ij} 是正确且唯一的输出矢量; 当变体 V_i 错误地响应输入矢量 I_j 时, 得到的输出矢量是集合 W_{ij} 中的某个特定输出矢量, 其中 W_{ij} 是变体 V_i 可能产生的错误输出矢量集合。故由图 10 可知, 针对输入矢量集合 I , IPO 系统将对应得到输出矢量集合 $O = \{R_{ij} \cup W_{ij}\}_{i=1, j=1}^{n, m}$ 。

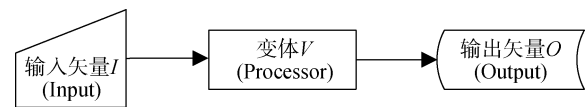


图 10 IPO 系统
Figure 10 IPO System

根据前提条件①可知, 任意两个变体 V_i 和 V_k 针对特定输入矢量 I_j 产生错误输出矢量的情形是相互独立的。进一步, 由前提条件③可知, 变体 V_i 和 V_k 产生相同错误输出矢量的情形也是相互独立的。即可描述为, 对于任意 $i, k = 1, 2, \dots, n$, 且 $i \neq k$, 令错误输出矢量 $\delta \in W_{ij} \cap W_{kj}$, 都满足等式(1):

$$P\{V_i(I_j) = \delta, V_k(I_j) = \delta\} = P\{V_i(I_j) = \delta\} \cdot P\{V_k(I_j) = \delta\} \quad (1)$$

根据前提条件②可知, “错误完成任务是小概率事件”, 因此给定概率限定值: $\alpha (0 < \alpha \leq 0.05)$ ^[78], 则对于任意变体 $V_i, i = 1, 2, \dots, n$, 给定的输入矢量

$I_j, j = 1, 2, \dots, m$, 都满足下列等式:

$$P\{V_i(I_j) \in W_{ij}\} < \alpha,$$

$$P\{V_i(I_j) \in W_{ij}\} > 0, i = 1, 2, \dots, n, j = 1, 2, \dots, m$$

且有等式(2):

针对输入矢量 I_j , 假设变体集合 V 中 V_1, V_2, \dots, V_n 产生的错误输出矢量集合 $W_{1j}, W_{2j}, \dots, W_{nj}$, 存在共同的错误输出矢量集 $\omega_j, j = 1, 2, \dots, m$, 存在 ω_j 使得:

$$\omega_j = \bigcap_{i=1}^n W_{ij} \quad (2)$$

错误输出矢量集合 W_{ij} 中输出矢量的个数记为 $\text{card}(W_{ij})$, 且 $\text{card}(W_{ij}) \neq 0$, 集合 ω_j 中输出矢量的个数记为 $\text{card}(\omega_j)$, 且 $\text{card}(\omega_j) \leq \text{card}(W_{ij})$, τ_j 为集合 ω_j 中的元素。假设对于任意一个变体 $V_i \in V$, 在响应输入矢量 I_j 时, 错误输出矢量集合 W_{ij} 中每个元素出现的概率是相等的。根据公式(1)和公式(2), 可得多变体系统在某一时刻被攻击成功的概率为:

$$\begin{aligned} P_j &= \sum_{\tau_j \in \omega_j} P\{E_1(I_j) = \tau_j, \dots, E_n(I_j) = \tau_j\} \\ &= \sum_{\tau_j \in \omega_j} \prod_{i=1}^n P\{E_i(I_j) = \tau_j\} \\ &= \sum_{\tau_j \in \omega_j} \prod_{i=1}^n \frac{P\{E_i(I_j) \in W_{ij}\}}{\text{card}(W_{ij})} \\ &= \prod_{i=1}^n \frac{P\{E_i(I_j) \in W_{ij}\}}{\text{card}(W_{ij})} \cdot \text{card}(\omega_j) \\ &= \prod_{i=1}^n P\{E_i(I_j) \in W_{ij}\} \cdot \frac{\text{card}(\omega_j)}{\text{card}(W_{ij})} \\ &< \prod_{i=1}^n \frac{\text{card}(\omega_j)}{\text{card}(W_{ij})} \cdot \alpha^n \end{aligned} \quad (3)$$

其中, 由于变体可能产生的错误输出矢量的种

类存在不可预知性, 且变体之间存在异构性, 当错误输出矢量集合 W_{ij} 越大, 所有变体均产生错误输出矢量且全部一致的概率 P_j 就越小, 因此, 系统中极少发生变体共模逃逸的事件。而针对 IPO 系统的真实使用场景, 假设某个用户操作可表示为具有 s 步骤的输入序列 L , L 由输入矢量集合 I 中的有限种输入矢量组成, 记为 $L = (I_{l_1}, I_{l_2}, \dots, I_{l_s})$ 。根据公式(3), 该用户操作使得变体发生共模逃逸的概率 P_L 为:

$$P_L = \prod_{t=1}^s P_{l_t} = \prod_{t=1}^s \prod_{i=1}^n P\{V_i(I_{l_t}) \in W_{il_t}\} \cdot \frac{\text{card}(\omega_{l_t})}{\text{card}(W_{il_t})} < (\max\{P_{l_1}, P_{l_2}, \dots, P_{l_s}\})^s$$

由此可见, 多变体系统在攻击防护方面, 当变体个数越多, 系统被攻击成功的概率越小, 但这是在不考虑性能影响的情况下的理想状态。

6.2 攻击分析

变体技术在单独使用时对某些攻击有针对性的防御效果, 不过也较容易被攻击者破解。但在多变体架构下为攻击者带来的攻击难度超出了这些技术的线性组合(见 6.1 原理分析)。然而, 有些时候并不是使用了更多的变体技术就能带来更好的防御效果。表 3 列出了每项变体技术对不同攻击的防御能力。

多变体执行架构通过并行运行变体实例加大了攻击者成功实施攻击的难度, 攻击者必须去构造可同时在多变体执行架构下对所有变体都有效的攻击负载。使用了不同变体生成技术的变体是不是组合起来会带来更大的安全增益? 是不是变体数量越多越好? 答案是不一定。

从理论上来说, 如果单个变体被成功攻击的概率为 P , 则在多变体执行架构下 N 个变体就可以将这一概率减少到 P^N , 但事实上可能并非总是如此, 而且有时候在组合使用多种随机化技术的变体时可能还来负面的效果。

表 3 变体技术防御能力

Table 3 Defensive Ability of Variant Technology

变体技术 \ 漏洞	反向栈	指令随机化	堆布局随机化	栈基址随机化	栈保护	系统调用号随机化	库函数名随机化
栈溢出漏洞	√	√	×	×	√	√	√
Return-to-lib	√	×	√	×	×	×	√
堆溢出漏洞	×	√	√	×	×	√	√
格式化字符串漏洞	×	√	×	×	×	√	√
整数溢出漏洞	×	×	×	×	×	×	×
野指针和悬空指针	×	√	×	×	×	√	√
双重释放	×	√	√	×	×	√	√

如图 11 所示, 某个程序存在栈溢出漏洞, 攻击者可以越过变量 A 的边界去覆盖栈上的返回地址 ret 。当使用多变体执行架构来运行该程序时, 假设有两个变体(变体 1 和变体 2), 则攻击者要成功实施攻击必须构造一个有效负载同时在两个变体中同时覆盖返回地址, 并将自己想要执行的代码地址($Fun()$ 的地址)写入返回地址处。如果变体同时使用了代码空间布局随机化技术生成, 则攻击者想要执行的代码片段在不同的变体中会处于不同的地址, 此时攻击者在使用溢出漏洞构造输入来覆盖栈上的返回地址时同时只能在一个变体中成功。

然而, 如果并行运行的两个变体是变体 1 和变体 3, 因为变量 A 距离返回地址的偏移在两个变体中不同, 这就给了攻击者突破多变体安全防御的可能, 因为可以构造出同时在这两个变体中都成功将返回地址覆盖为 $Fun()$ 地址的攻击负载。

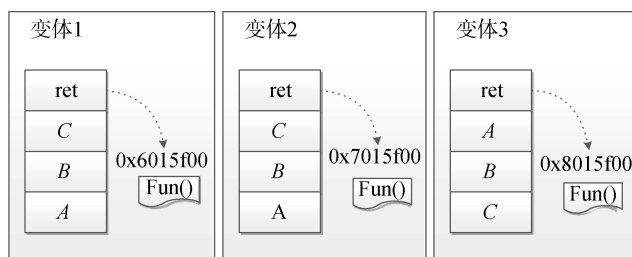


图 11 攻击示例

Figure 11 Attack Example

Davis 对多变体执行环境中类似可能受到的攻击进行了初步地分析和举例^[84], 说明了在构造多变体执行架构时, 对变体技术的不当使用会给攻击者带来可攻击的条件, 如何在多变体执行架构下有效的组合使用变体生成技术还有待进一步研究。但从目前的研究中至少可以总结出以下需要注意的地方:

1) 由于操作系统的执行保护措施使得攻击者很难使用注入代码的方式结合漏洞来轻易实施攻击, 取而代之的是使用程序代码空间中已有的可以完成特定功能的代码片段, 因此在变体生成时使用代码空间布局随机化是必要的;

2) 变体之间最好不要共享内存空间, 即在内存空间布局上应该严格分离, 使得不同变体之间没有共享的绝对地址;

3) 在使用栈和堆空间布局随机化时要加入监督变量进行访问时检查。

其他的攻击风险就是前面提到的控制流转换和数据交换过程中所引入的新的攻击面, 以及决策算法中可能存在的逃逸。

7 多变体执行技术面临的挑战与未来研究方向

多变体执行安全防御技术在经过多年的发展之后, 已经在软件防御方面积累了丰富的经验, 但是基于现有方法所开发出来的多变体系统还不是很完善, 很难在安全性和系统开销方面取得较好的平衡, 而且有一些关键性的问题没有很好的解决, 这为其运用在实际环境中产生了障碍; 此外伴随着近些年各种平台(如移动平台、虚拟化平台、云平台等)的蓬勃发展、软件领域发展趋势的变化(多线程编程)等, 也为多变体系统提出了新的挑战。本节对这些问题进行梳理, 并对未来研究方向进行展望。

7.1 多变体执行当前面临的挑战

1) 多变体执行架构还不完善。多变体执行架构自提出以来并没有发生太多的改变, 对于一套完整的安全防御体系来说, 特别是主动防御, 除了检测和拦截以外, 还应该具有反馈和处理的环节, 这样整个架构就能根据实际工作场景不断自我调整和进化。在当前的基本架构中, 如何根据检测结果来对威胁变体进行反馈调整, 如替换、清洗策略, 变体自身变化机制调整策略等都需要进一步研究。

另外, 目前多变体执行架构对未知威胁的拦截只限于发现和阻止, 在构架中加入对它们进行分析处理的模块不仅能让架构本身更加完善, 也能将得到的“知识”用于与其他防御措施的联动。

2) 监控技术还有遗留问题。大部分多变体系统在发现攻击之后会将受攻击变体停止运行, 下次发现攻击时再重复这一过程。应该设计一种具有“记忆”的监控技术, 这样可以加快裁决, 进一步提高整个架构的执行效率, 另外, 监控的“记忆”在遏制攻击后还能以友好的方式恢复变体状态, 并对用户无感。

3) 不确定及不一致问题。这类问题一部分来自于多线程程序以及子进程和线程的调度问题, 另一部分来自异步信号、文件描述符、进程 ID、时间和随机数等会引起不确定及不一致的方面, 这些都会引起系统裁决时的误报。此问题在多变体执行系统中是一个难点。

4) 未决问题。多变体系统目前不能防御侧信道攻击。同时, 随着目前移动平台、虚拟化平台、云平台的发展, 探究将多变体执行架构在各个平台的应用也是该研究需要扩展的方面。

7.2 拟态安全防御

网络空间拟态防御(Cyber Mimic Defense, CMD)

以动态异构冗余(Dynamic Heterogeneous Redundancy, DHR)形态的广义鲁棒控制架构 DHR 为基础, 在“去协同化”条件下, 对 DHR 的多模裁决、策略调度、负反馈控制、多维动态重构以及相关的输入与输出代理等环节施以拟态伪装策略, 能够一体化的提供高可靠、高可用、高可信的鲁棒性服务和点面结合的融合式防御功能, 并可自然地接纳已有或未来的安全技术获得指数量级的防御增益, 其模型如图 12 所示。

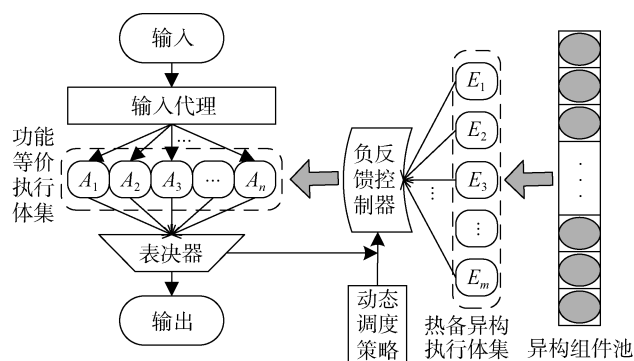


图 12 DHR 模型

Figure 12 DHR Model

拟态防御思想比多变体防御思想更广泛。从多变体的定义可以看出, 变体通常是针对一个应用程序或实体而言的, 它的变化主要是针对执行环境的变化, 是将移动目标防御的思想与冗余执行相结合。而拟态思想在构造冗余执行体时强调的是执行体本身的“变化”, 包括实现方式和算法等, 突出由此带来的内生安全特性。其次是动态, 拟态防御思想中的动态体现在执行体调度、表决和恢复等多个环节, 相对来说, 多变体执行中的动态体现在静态构建执行体的过程中, 在运行时很少有体现。最后是冗余, 虽然两者都有冗余的执行体, 但多变体环境中的冗余执行体是为了反映出攻击行为在改变软件执行流程时的不同, 为了提高执行效率, 在很多架构实现中只有一个变体体负责全部的执行工作, 而拟态架构中的冗余执行体多数情况下是要同时执行相同的任务, 这也是拟态防御思想构建的执行环境同时具有高可用和高可靠的原因。就使用范围来说, 目前拟态思想应用的领域不仅包括软件安全防御, 还包括操作系统^[85]、整个应用服务栈^[86]、云和科学计算^[87]等领域, 以及硬件级的应用^[88]。

拟态防御的动态和反馈机制使得拟态构造的防御系统具有内生的测不准效应, 这也是与多变体执行不同的地方。多变体执行通过多样化技术构造了

变体, 但在实际的运行中缺乏动态的变化, 若没有动态反馈机制, 一旦攻击者成功利用了某个漏洞并绕过了监控机制, 则攻击者可以一直利用此漏洞, 造成不可估量的损失。加入动态及反馈机制之后, 变体会因“势”而“变”, 系统内部在不同时刻呈现出一种测不准效应, 即使某时刻出现小概率的逃逸事件, 也不会让攻击者连续逃逸。

同时, 拟态防御在裁决机制上更加完善。其裁决机制采用可以动态迭代的裁决算法集合, 除了包含大数裁决外还可包含最大近似裁决、权重裁决、掩码裁决、基于历史信息的大数裁决、拜占庭投票裁决等多种裁决算法。加之基于反馈控制环路的后向迭代验证机制, 可将攻击持续逃逸概率最小化。

7.3 未来研究

多变体执行在信息系统安全防御领域已经积累了不少的经验, 但仍存在一些问题使其未大范围实际应用。作为结构更加完善的拟态防御架构为“网络空间再平衡战略”提供了解决方案, 能够在信息系统安全方面解决未知漏洞攻击和安全威胁大范围、快速传播等问题。目前虽然有拟态构造 Web 服务器、拟态构造路由器、拟态构造域名服务器等系列产品设备, 但在拟态软件构造方面还缺乏经验, 将多变体执行安全技术的经验与拟态防御思想相结合来实现软件安全防御, 使目前的多变体执行架构变成具有动态异构冗余结构的拟态构造, 则带来的安全增益不可估量。

8 结语

多变体执行技术通过多样化技术、冗余的架构让针对同一软件的同一漏洞在多个执行体上表现不一致在软件运行时检测攻击, 从而达到软件安全防御的目的, 该思想对安全技术的发展有重要的启发意义。为了这项技术能在实际中广泛应用, 还有一些关键问题有待解决。本文从对多执行体的冗余执行来解决安全问题的探索开始, 概括近年来多变体执行架构的演变发展, 梳理该架构中的关键技术, 并对比关键技术中不同实现方法的优缺点, 指出在工程实现以及评估时需要注意的关键问题, 并列举了当前多变体执行架构技术面临的挑战, 以及未来可能的研究方向。

致谢 本文工作受到国家重点研发计划网络空间安全专项(No.2018YFB0804003, No.2017YFB0803204)资助。

参考文献

- [1] Nie C J, Zhao X F, Chen K, et al. An Software Vulnerability Number Prediction Model Based on Micro-Parameters[J]. *Journal of Computer Research and Development*, 2011, 48(7): 1279-1287.
(聂楚江, 赵险峰, 陈恺, 等. 一种微观漏洞数量预测模型[J]. 计算机研究与发展, 2011, 48(7): 1279-1287.)
- [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] CVE-2014-6271. Common Vulnerabilities and Exposures, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271>, 2014.
- [5] J. Viega, J.-T. Bloch, Y. Kohn, et al. ITS4: A Static Vulnerability Scanner for C and C++ Code[C]. *16th Annual Computer Security Applications Conference (ACSAC'00)*, 2000: 257-267.
- [6] D. A. Wagner, J. S. Foster, E. A. Brewer, et al. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities[C]. *Symposium on Network and Distributed System Security(NDSS'00)*, 2000: 200-202.
- [7] D. Larochelle, D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities[C]. *10th USENIX Security Symposium(SEC'01)*, 2001:158.
- [8] Dor N, Rodeh M, Sagiv M. CSSV: Towards a Realistic Tool for Statically Detecting all Buffer Overflows in C[C]. *the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003: 155-167.
- [9] C. Cowan, C. Pu, D. Maier, et al. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks[C]. *USENIX Security Symposium*, 1998: 63-78.
- [10] A. Baratloo, N. Singh, T. K. Tsai. Transparent Run-Time Defense against Stack Smashing Attacks[C]. *USENIX Annual Technical Conference*, 2000: 251-262.
- [11] E. Haugh, M. Bishop. Testing C Programs for Buffer Overflow Vulnerabilities[C]. *Symposium on Network and Distributed System Security*, 2003:256-261.
- [12] M. Prasad, T.-c. Chiueh. A Binary Rewriting Defense against Stack based Buffer Overflow Attacks[C]. *USENIX Annual Technical Conference*, 2003: 211-224.
- [13] O. Ruwase, M. S. Lam. A Practical Dynamic Buffer Overflow Detector[C]. *Symposium on Network and Distributed System Security*, 2004: 159-169.
- [14] Sidirolglou S, Giovanidis G, Keromytis A D. A Dynamic Mechanism for Recovering from Buffer Overflow Attacks[M]. *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005: 1-15.
- [15] Kuperman B A, Brodley C E, Ozdoganoglu H, et al. Detection and Prevention of Stack Buffer Overflow Attacks[J]. *Communications of the ACM*, 2005, 48(11): 50-56.
- [16] Lee R B, Karig D K, McGregor J P, et al. Enlisting Hardware Architecture to Thwart Malicious Code Injection[M]. *Security in Pervasive Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004: 237-252.
- [17] J. P. McGregor, D. K. Karig, Z. Shi, et al. A Processor Architecture Defense against Buffer Overflow Attacks[C]. *International Conference on Information Technology: Research and Education*, 2003: 243-250.
- [18] N. Tuck, B. Calder, G. Varghese. Hardware and Binary Modification Support for Code Pointer Protection from Buffer Overflow[C]. *the 37th annual IEEE/ACM International Symposium on Microarchitecture*, 2004: 209-220.
- [19] U. Shankar, K. Talwar, J. S. Foster, et al. Detecting Format String Vulnerabilities with Type Qualifiers[C]. *USENIX Security Symposium*, 2001: 201-220.
- [20] C. Cowan, M. Barringer, S. Beattie, et al. FormatGuard: Automatic Protection From printf Format String Vulnerabilities[C]. *USENIX Security Symposium*, 2001:124-131.
- [21] T. Tsai, N. Singh. Libsafe 2.0: Detection of Format String Vulnerability Exploits[OB], white paper, Avaya Labs, 2001.
- [22] Ringenburt M F, Grossman D. Preventing Format-string Attacks via Automatic and Efficient Dynamic Checking[C]. *the 12th ACM conference on Computer and communications security*, 2005: 354-363.
- [23] Data Execution Prevention, Microsoft, <https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention>, May. 2018.
- [24] J. Newsome, D. X. Song. Dynamic Taint Analysis for Automatic Detection[C]. *Symposium on Network and Distributed System Security*, 2005: 3-4.
- [25] 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.
- [26] V. Mohan, P. Larsen, S. Brunthaler, et al. Opaque Control-Flow Integrity[C]. *Symposium on Network and Distributed System Security*, 2015: 27-30.
- [27] C. Zhang, T. Wei, Z. Chen, et al. Practical Control Flow Integrity & Randomization for Binary Executables[C]. *Symposium on Security and Privacy*, 2013: 559-573.
- [28] M. Zhang, R. Sekar. Control Flow Integrity for COTS Binaries[C]. *Presented as part of the 22nd USENIX Security Symposium*, 2013: 337-352.

- [29] Evans I, Long F, Otgonbaatar U, et al. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity[C]. *the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015: 901-913.
- [30] E. Göktas, E. Athanasopoulos, H. Bos, et al. Out Of Control: Overcoming Control-Flow Integrity[C]. *Symposium on Security and Privacy*, 2014: 575-589.
- [31] Conti M, Crane S, Davi L, et al. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks[C]. *the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015: 952-963.
- [32] Payer M. HexPADS: A Platform to Detect "Stealth" Attacks[M]. *Lecture Notes in Computer Science*. Cham: Springer International Publishing, 2016: 138-154.
- [33] S. Bhatkar, D. C. DuVarney, R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits[C]. *USENIX Security Symposium*, 2003: 291-301.
- [34] S. Forrest, A. Somayaji, D. H. Ackley. Building Diverse Computer Systems[C]. *The Sixth Workshop on Hot Topics in Operating Systems*, 1997: 67-72.
- [35] J. Xu, Z. Kalbarczyk, R. K. Iyer. Transparent Runtime Randomization for Security[C]. *22nd International Symposium on Reliable Distributed System*, 2003: 260-269.
- [36] Kc G S, Keromytis A D, Prevelakis V. Countering Code-injection Attacks with Instruction-set Randomization[C]. *the 10th ACM conference on Computer and communication security*, 2003: 272-280.
- [37] M. Chew, D. Song. Mitigating Buffer Overflows by Operating System Randomization[DB]. 2002. https://xueshu.baidu.com/usercenter/paper/show?paperid=3e2337dd800260571b06bceae7d979f1&site=xueshu_se.
- [38] Lvin V B, Novark G, Berger E D, et al. Archipelago: Trading Address Space for Reliability and Security[C]. *the 13th international conference on Architectural support for programming languages and operating systems*, 2008: 115-124.
- [39] C. Giuffrida, A. Kuijsten, A. S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization[C]. *21st USENIX Security Symposium*, 2012: 475-490.
- [40] 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.
- [41] A. N. Sovarel, D. Evans, N. Paul. Where's the FEEB? The Effectiveness of Instruction Set Randomization[C]. *the 14th conference on USENIX Security Symposium*, 2005: 268-271.
- [42] M. K. Joseph. Architectural issues in fault-tolerant, secure computing systems[PD]. University of California at Los Angeles, 1988.
- [43] Knowlton K C. A Combination Hardware-Software Debugging System[J]. *IEEE Transactions on Computers*, 1968, C-17(1): 84-86.
- [44] Berger E D, Zorn B G. DieHard: Probabilistic Memory Safety for Unsafe Languages[C]. *the 2006 ACM SIGPLAN conference on Programming language design and implementation*, 2006: 158-168.
- [45] B. Cox, D. Evans, A. Filipi, et al. N-Variant Systems: A Secretless Framework for Security through Diversity[C]. *USENIX Security Symposium*, 2006: 105-120.
- [46] J. Reynolds, J. Just, E. Lawson, et al. The Design and Implementation of an Intrusion Tolerant System[C]. *International Conference on Dependable Systems and Networks*, 2002: 285-290.
- [47] Totel E, Majorczyk F, Mé L. COTS Diversity Based Intrusion Detection and Application to Web Servers[M]. *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006: 43-62.
- [48] Gao D B, Reiter M K, Song D. Behavioral Distance for Intrusion Detection[M]. *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006: 63-81.
- [49] G. Novark, E. D. Berger, B. G. Zorn. Exterminator: Automatically Correcting Memory Errors with High Probability[J]. *Communications of the ACM*, 2008, 51(12): 87-95.
- [50] A. R. Yumerefendi, B. Mickle, L. P. Cox. TightLip: Keeping Applications from Spilling the Beans[C]. *the 4th USENIX Symposium on Networked Systems Design & Implementation*, 2007: 25-31.
- [51] E. Weatherwax, J. Knight, A. Nguyen-Tuong. A Model of Secretless Security in N-Variant Systems[C]. *Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, 2009: 25-31.
- [52] Salamat B, Jackson T, Gal A, et al. Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-space[C]. *the fourth ACM european conference on Computer systems*, 2009: 33-46.
- [53] Volckaert S, de Sutter B, de Baets T, et al. GHUMVEE: Efficient, Effective, and Flexible Replication[M]. *Foundations and Practice of Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013: 261-277.
- [54] S. Volckaert, B. Coppens, A. Voulimeas, et al. Secure and Efficient Application Monitoring and Replication[C]. *2016 USENIX Annual Technical Conference*, 2016: 167-179.
- [55] P. Hosek, C. Cadar. VARAN the Unbelievable: An Efficient N-version Execution Framework[C]. *ACM Special Interest Group on Programming Languages*, 2015: 339-353.
- [56] M. Maurer, D. Brumley. TACHYON: Tandem Execution for Efficient Live Patch Testing[C]. *21st USENIX Security Symposium*, 2012: 617-630.

- [57] P. Hosek, C. Cadar. Safe Software Updates via Multi-version Execution[C]. *the 2013 International Conference on Software Engineering*, 2013: 612-621.
- [58] K. Koning, H. Bos, C. Giuffrida. Secure and Efficient Multi-variant Execution Using Hardware-assisted Process Virtualization[C]. *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016: 431-442.
- [59] A. Belay, A. Bittau, A. Mashtizadeh, et al. Dune: Safe User-level Access to Privileged CPU Features[C]. *10th USENIX Symposium on Operating Systems Design and Implementation*, 2012: 335-348.
- [60] A. Avizienis, L. Chen. On the Implementation of N-version Programming for Software Fault-Tolerance During Program Execution[C]. *International Computer Software and Applications Conference*, 1977: 149-155.
- [61] L. Chen, A. Avizienis. N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation[C]. *8th International Symposium on Fault-Tolerant Computing*, 1978:25-26.
- [62] Wartell R, Mohan V, Hamlen K W, et al. Securing Untrusted Code via Compiler-agnostic Binary Rewriting[C]. *the 28th Annual Computer Security Applications Conference*, 2012: 299-308.
- [63] PAX Address Space Layout Randomization. PaX Team, <https://pax.grsecurity.net/docs/aslr.txt>, Mar. 2003.
- [64] Luk C K, Cohn R, Muth R, et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation[C]. *the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005: 190-200.
- [65] Nethercote N, Seward J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation[C]. *the 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007: 89-100.
- [66] P. Larsen, A. Homescu, S. Brunthaler, et al. SoK: Automated Software Diversity[C]. *2014 IEEE Symposium on Security and Privacy*, 2014: 276-291.
- [67] C. Ko, G. Fink, K. Levitt. Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring[C]. *Tenth Annual Computer Security Applications Conference*, 1994: 134-144.
- [68] Parampalli C, Sekar R, Johnson R. A Practical Mimicry Attack Against Powerful System-call Monitors[C]. *the 2008 ACM symposium on Information, computer and communications security*, 2008: 156-167.
- [69] D. Bruschi, L. Cavallaro, A. Lanzi. Diversified Process Replicæ for Defeating Memory Error Exploits[C]. *IEEE International Performance, Computing, and Communications Conference*, 2007: 434-441.
- [70] N. Provos. Improving Host Security with System Call Policies[C]. *the 12th conference on USENIX Security Symposium*, 2003: 257-272.
- [71] Bruening D, Zhao Q, Amarasinghe S. Transparent Dynamic Instrumentation[J]. *ACM SIGPLAN Notices*, 2012, 47(7): 133-144.
- [72] L. Cavallaro. Comprehensive Memory Error Protection via Diversity and Taint-Tracking[PD]. *Universita Degli Studi Di Milano*, 2007.
- [73] Volckaert S, Coppens B, de Sutter Member B. Cloning your Gadgets: Complete ROP Attack Immunity with Multi-Variant Execution[J]. *IEEE Transactions on Dependable and Secure Computing*, 2016, 13(4): 437-450.
- [74] Gawlik R, Koppe P, Kollenda B, et al. Detile: Fine-Grained Information Leak Detection in Script Engines[M]. *Detection of Intrusions and Malware, and Vulnerability Assessment*. Cham: Springer International Publishing, 2016: 322-342.
- [75] K. Lu. Securing Software Systems by Preventing Information Leaks[PD]. *Georgia Institute of Technology*, 2017.
- [76] S. Nagarakatte, J. Zhao, M. M. Martin, et al. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C[J]. *ACM Sigplan Notices*, 2009, 44(6): 245-258.
- [77] S. Nagarakatte, J. Zhao, M. M. Martin, et al. CETS: Compiler-Enforced Temporal Safety for C[J]. *ACM Sigplan Notices*, 2010, 45(8): 31-40.
- [78] 魏宗舒, 等. 概率论与数理统计教程(第2版)第二版, 魏宗舒等, 编高教社高等教育出版社[M]. 北京: 高等教育出版社, 2008.
- [79] Lu K J, Xu M, Song C Y, et al. Stopping Memory Disclosures via Diversification and Replicated Execution[J]. *IEEE Transactions on Dependable and Secure Computing*, 2018: 1.
- [80] Voulimeneas A, Song D, Parzefall F, et al. DMON: A Distributed Heterogeneous N-Variant System[EB/OL]. 2019: arXiv:1903.03643[cs.CR]. <https://arxiv.org/abs/1903.03643>.
- [81] Österlund S, Koning K, Olivier P, et al. KMOVX: Detecting Kernel Information Leaks with Multi-variant Execution[C]. *the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019: 23-31.
- [82] 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.
- [83] Bulba, Kil3r. Bypassing StackGuard and StackShield[DB]. 2000, https://xueshu.baidu.com/usercenter/paper/show?paperid=725d6ccfb189bb020166821cafa69467&site=xueshu_se.
- [84] Davis B, Larsen P, Volckaert S, et al. Composition Challenges for Automated Software Diversity[DB]. 2016, https://www.researchgate.net/publication/320434333_Composition_Challenges_for_Automated_Software_Diversity.
- [85] Qi C, Wu J X, Cheng G Z, et al. An Aware-scheduling Security Architecture with Priority-equal Multi-controller for SDN[J].

China Communications, 2017, 14(9): 144-154.

- [86] Tong Q, Zhang Z, Zhang W H, et al. Design and Implementation of Mimic Defense Web Server[J]. *Journal of Software*, 2017, 28(4): 883-897.

(仝青, 张铮, 张为华, 等. 拟态防御 Web 服务器设计与实现[J]. 软件学报, 2017, 28(4): 883-897.)

- [87] Wang Y W, Wu J X, Guo Y F, et al. Scientific Workflow Execution System Based on Mimic Defense in the Cloud Environment[J]. *Frontiers of Information Technology & Electronic Engineering*, 2018, 19(12): 1522-1536.

- [88] Ma H L, Yi P, Jiang Y M, et al. Dynamic Heterogeneous Redun-

dancy Based Router Architecture with Mimic Defenses[J]. *Journal of Cyber Security*, 2017, 2(1): 29-42.

(马海龙, 伊鹏, 江逸茗, 等. 基于动态异构冗余机制的路由器拟态防御体系结构[J]. 信息安全学报, 2017, 2(1): 29-42.)

- [89] Capizzi R, Longo A, Venkatakrishnan V N, et al. Preventing information leaks through shadow executions[C]. *the 2008 Annual Computer Security Applications Conference*, 2008: 102-110.
- [90] Kwon Y, Kim D, Sumner W N, et al. LDX: Causality Inference by Lightweight Dual Execution[C]. *the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016: 125-131.



姚东 于 2013 年在信息工程大学计算机科学与技术专业获得硕士学位, 现在信息工程大学计算机科学与技术专业攻读博士学位, 研究领域为网络安全。Email: dojn_dd@163.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: 主动防御技术、高性能计算。Email: ponyzhang@126.com



张高斐 于 2018 年在郑州轻工业大学网络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。研究领域为网络安全。研究兴趣包括: 主动防御技术、网络体系结构。Email: gaofei_zhang@163.com



刘浩 于 2017 年在合肥工业大学数学与应用数学专业获得学士学位。现在信息工程大学网络空间安全专业攻读硕士学位。研究领域为网络空间安全。研究兴趣包括: 拟态主动防御、多样化编译。Email: six_paper@163.com



潘传幸 于 2018 年在山东农业大学计算机科学与技术专业获学士学位。现在信息工程大学攻读硕士学位。研究领域为拟态防御。研究兴趣包括: 主动防御, 云计算, 物联网等。Email: chuanxing_pan@163.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。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