doi:10.3772/j.issn.1002-0470.2018.04.003

基于硬件的代码复用攻击防御机制综述①

张 军②[*,**,***] 侯 锐[**] 詹志远[**,***] 张立新[*] 陈明宇[*] 孟 丹[**]

([*]中国科学院计算技术研究所 北京 100190)

([**]中国科学院信息工程研究所 北京 100093)

([***]中国科学院大学 北京 100049)

摘 要 给出了代码复用攻击挟持控制流的过程,介绍了代码复用攻击防御机制研究现状。重点论述了基于硬件的防御机制,包括基于硬件的存储安全、代码指针完整性、攻击特征检查、控制流完整性、数据执行保护与隔离技术。讨论了基于硬件的防御机制存在的问题、优势及硬件与软件的关系。展望了基于硬件的代码复用攻击防御机制的发展方向:基于硬件的代码指针完整性有望成为防止控制流挟持攻击的有效手段,针对非控制数据的攻击与防御有可能成为新的研究热点,支持可配置的硬件防御架构是硬件防御的重要趋势之一。

关键词 控制流挟持,存储错误,代码复用攻击,存储安全,代码指针完整性,控制流完整性(CFI),数据执行保护(DEP),隔离技术

0 引 言

随着信息技术的发展与广泛应用,计算机系统与我们的生产和生活联系越来越紧密,其安全问题越来越受重视。由于软件功能的不断创新和完善,系统软件层的设计复杂度和代码量迅猛提升,可被攻击者利用的代码漏洞也随之增加。2005 年以后,公共漏洞和暴露(common vulnerabilities and exposures,CVE)字典每年统计的漏洞数量在 4000 个以上,2014 年甚至接近 8000 个。存储错误漏洞作为当前主要的攻击向量之一,占总漏洞数量的 20.13%[1]。

存储错误主要包括栈/堆上的缓冲区溢出、整数溢出、格式化字符串漏洞及释放后再利用等[2]。历史上曾发生过多起基于存储错误的漏洞利用,例如 1988 年的网络蠕虫病毒[3]与 2003 年的 SQL Slammer 病毒[4]等。基于存储错误的漏洞利用往往被攻击者用来挟持程序的控制流[2]。控制流挟持攻击可分为代码注入攻击与代码复用攻击(code-reuse attacks,CRA)[2,5]。前者将恶意代码注入到应用的地址空间,然后将程序控制流重定向到注入的恶意代码。数据执行保护(data execution prevention,DEP)将内存中可写的数据标记为可写不可执行(W⊕X),有效防御代码注入攻击[5]。然而,DEP 不能防御 CRA。因此,下文中用 CRA 代表控制流挟持攻击。CRA 利用存储错误漏洞,将控制流重定向到内存中可执行代码,通过一系列现有代码片段串在一起执行达到攻击目的[6-8]。

根据实施 CRA 的不同阶段,CRA 的防御机制分为 6 类:存储安全、代码指针完整性(code pointer integrity, CPI)、地址空间布局随机化(ALSR)、控制流完整性(control-flow integrity, CFI)、数据执行保护(DEP)与隔离技术[2]。这些机制可用软件实现,也可以基于硬件实现。本文中硬件包括处理器本身

① 863 计划(2015AA0153032),国家重点研发计划(2016YFB1000400),中国科学院前沿科学重点研究项目(QYZDB - SSW - JSC010)和国家自然科学基金优秀青年科学基金(61522212)资助项目。

② 男,1985 年生,博士;研究方向:体系结构扩展增强系统安全性;E-mail: zhangjun02@ict.ac.cn

(收稿日期:2017-11-15)

已有的功能(如处理器的性能计数器)与通过扩展处理器体系结构增加的功能。纯软件实现的防御机制性能开销较大,并且软件防御机制本身也可能存在漏洞,容易被攻击者绕过。基于硬件的防御机制性能开销较少,并且硬件处于系统最底层,可隔离高特权级程序的操作。虽然已经有工作对代码复用攻击(CRA)的防御机制进行了总结[9-11],但他们并没有分析基于硬件的防御机制。本文着重介绍基于硬件的 CRA 防御机制。讨论当前基于硬件的防御机制存在的问题与改进方法,并在此基础上探讨基于硬件防御机制的研究方向。

1　代码复用攻击及其防御研究现状

与代码注入攻击不同,代码复用攻击(CRA)将控制流重定向到攻击者选择的可执行代码段。根据复用的代码片段(gadgets)类型的不同,可将 CRA 分为 3 类:面向返回编程(return-oriented programming, ROP)攻击[6],面向跳转编程(jump-oriented programming, JOP)攻击[7],面向伪造对象编程(counterfeit object-oriented programming, COOP)攻击[8]。本节先介绍这三类 CRA 的方式,然后概述当前 CRA 防御机制的研究现状。

1.1　代码复用攻击

1.1.1　面向返回编程(ROP)攻击

返回到系统库函数(ret-to-libc)是最早的 ROP 攻击[12]。这种方法可以绕过数据执行保护(DEP),在不注入恶意代码的情况下,通过修改返回地址,跳转到并执行 libc(包含各种共享函数的 C 语言库)中的某个函数。例如:跳转到库函数 system(),并提供参数"/bin/sh",衍生一个 shell。虽然返回到系统库函数攻击突破了数据执行保护(DEP)的限制,但 libc 中可用的函数功能有限,限制了这种方式的攻击能力[6]。

为了突破返回到系统库函数攻击的限制,Shacham 等人[6]在 2007 年提出 ROP 攻击。这种攻击复用的代码片段以 *ret* 指令或具有类似功能的指令序列(例如:*pop x*; *jmp* ∗*x*)结尾。*ret* 之前指令实现攻击者选择实现的功能,*ret* 指令则将控制流转换到下一个代码片段,这种方式使攻击者复用现有代码构造复杂攻击。

目前 ROP 攻击是漏洞利用的重要技术之一,攻击者可利用 ROP 攻击先关闭 DEP,然后注入并执行恶意代码。图 1 所示为 ROP 攻击的例子。在第 1 步,攻击者将攻击负载载入到应用的堆栈中。负载为虚线框中的内容,包括一定数量的返回地址与相关参数。在第 2 步,攻击者利用存储错误,用代码片段 A 的地址覆盖栈中返回地址。从第 3 步开始,攻击已经挟持控制流,将控制流重定向到代码片段 A。代码片段 A 改变栈指针(在 x86 架构中栈指针为 esp)指向载入的攻击负载。此时,esp 代替 eip 作为程序指针,这为第 4 步。从第 5 步开始,通过执行 *ret* 指令,将控制流从一个代码片段转移到下一个代码片段,本例最后执行一个系统调用。从上例可以看出,ROP 的基本块为不改变 esp 的代码块,条件分支与循环可以通过修改 esp 实现。综合使用算术运算、逻辑运算与条件分支可以实现图灵完备的攻击。

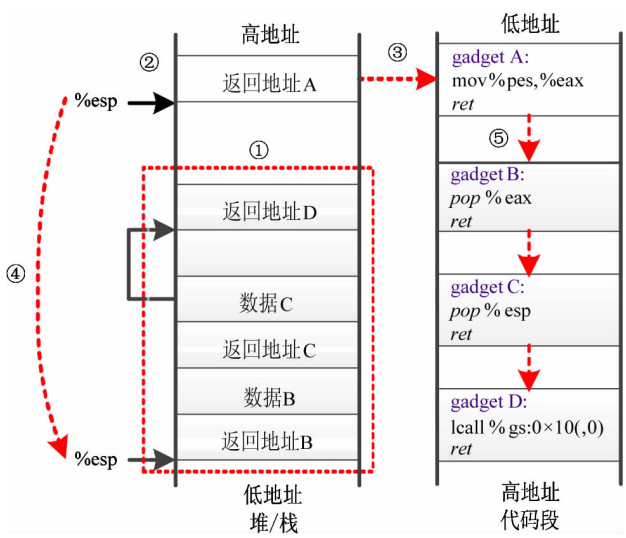

图 1　面向返回编程攻击示例

1.1.2　面向跳转编程(JOP)攻击

因为 ROP 攻击依赖于返回指令,很容易根据这一特征进行检测并防御,例如:检测指令流中频繁使用返回指令,或利用编译器消除可执行文件中的返回指令。为了改变这一不足,Bletsch 等提出 JOP 攻击[7]。与 ROP 攻击类似,JOP 攻击也是将较短的代码片段串连执行实现,只是复用的代码片段由间接

跳转指令 *jmp* 结尾。JOP 攻击利用派发表(dispatch table)保存被复用代码片段地址与参数,利用任意指向派发表的寄存器作为程序计数器,由特定的派发代码片段(dispatcher)挟持控制流。在调用下一个复用代码片段时,派发代码片段设定虚拟程序计数器,跳转到相应代码片段。

图 2 所示为 JOP 攻击示例。与 ROP 攻击类似,JOP 攻击也需将攻击负载载入到内存中,攻击负载为图中所示的派发表,包含可复用代码的起始地址及相关数据。在发起攻击之前,将派发表的起始地址保存到寄存器 edx 中(即虚拟程序计数器),派发代码片段利用一条加法指令驱动控制流。由此可见,JOP 攻击不依赖于程序栈和 esp 引用被复用的代码片段,可以用任意的,甚至不连续的存储区间保存派发表。

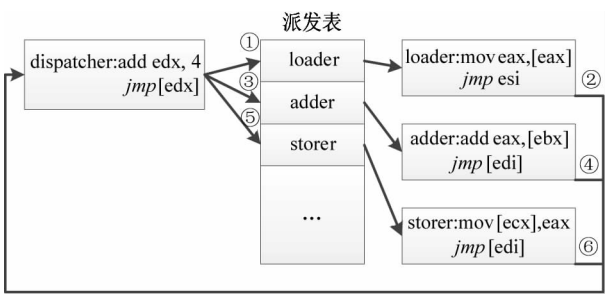

图 2　面向跳转编程攻击示例[7]

1.1.3　面向伪造对象编程(COOP)攻击

COOP 攻击是一种顶级的针对前向控制流完整性(CFI)的攻击方式,目前主要出现在学术界,还没有出现在漏洞利用的工具包中。Schuster 等[8]证明 COOP 攻击可攻破 Win 7 上的 IE10,以及 64 位 Linux 上的 Firefox。Matt 等[13]则用这种攻击方式成功绕过微软执行流保护(CFG)对 Win 10 Edge 的保护。

为了绕过前向 CFI 检查,COOP 攻击复用C++对象中的虚函数。每个虚函数实现一定的功能(例如:实现算术运算,载入值到寄存器等),把这些虚函数组合在一起实现复杂的任务。每个复用的虚函数称为虚函数片段(virtual function gadget, vfgadget),其中最重要的 vfgadget 是主循环函数(ML-G)。主循环函数中包含一个指针数组或链表,可用于将伪造对象组合在一起,然后在主循环函数中按攻击者的意图依次调用 vfgadget。

COOP 攻击的控制流程如图 3 所示。COOP 攻击开始时挟持目标应用中的C++对象,这个对象称为初始化对象。初始化对象的虚指针指向一个包含主循环函数指针的虚表。攻击者通过控制的初始化对象调用一个虚函数启动主循环函数。通过在主循环函数中迭代对象数组或链表(包含伪造对象指针),调用攻击者指定的 vfgadget。由此可见,COOP 攻击也与 ROP 攻击类似,ROP 攻击在控制程序指针后先操作栈指针,然后连续执行一系列的以返回指令结尾的代码片段,而 COOP 在控制程序指针后进入一个循环函数,依靠循环函数依次执行攻击者精心伪造的虚函数。

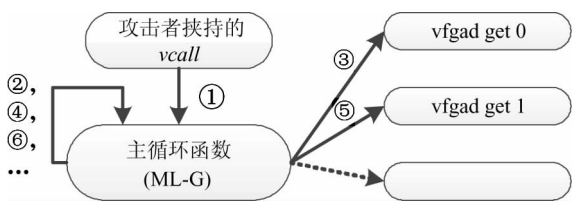

图 3　COOP 攻击控制流程[8]

1.2　防御机制研究现状

基于 Szekeres 等[2]构建的存储错误漏洞利用模型,图 4 描述了实施控制流挟持的步骤及对应的防御机制。

实施控制流挟持攻击的根源在于系统中存在存储错误漏洞。攻击从触发存储错误开始,例如:使一个指针超出边界或悬空。若能彻底消除代码中的存储错误,则能从根源上防止控制流挟持攻击。然而为了追求高效,当前的系统程序均采用C/C++这类低层次语言编写。因为这类语言缺少安全检测,很容易出现存储错误。若使用诸如 Cyclone[14]与 CCured[15]等具有安全检测的语言,则会带来很大性能开销。因此完全消除存储错误不现实,只能采用额外的机制在程序运行时防止基于存储错误的漏洞利用。

在攻击的第 2 步,攻击者利用存储错误将一个指针重写。例如使用缓冲区溢出重写程序栈上保存的返回地址;利用释放后重用将函数指针指向其它

位置。为了保护与控制流相关的指针，Kuznetsov 等[16]提出代码指针完整性(CPI)机制。CPI 通过静态分析识别代码中安全敏感指针，将安全敏感指针保存在安全区域。在这些指针引用时，通过检查相应的元数据判断指针引用的有效性。但 CPI 的安全性依赖于安全区域的安全性，Evans 等[17]与 Göktaş 等[18]通过侧信道攻击，成功找到了 CPI 的安全区域。CCFI[19]通过加密的方式保证敏感指针的完整性，可有效防止控制流挟持攻击，但 CCFI 的性能开销较大。

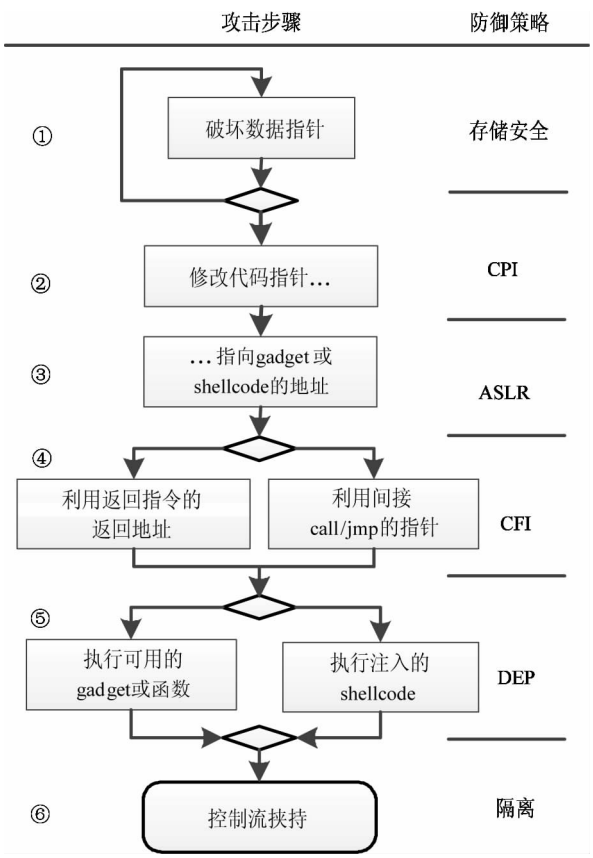

图 4　控制流挟持实施步骤及各步的防御机制[2]

在攻击的第 3 步，攻击者需要将代码指针指向特定的代码片段或 shellcode。针对这一步，可以采用地址空间布局随机化(ASLR)[20]的方法防御。ASLR 采用随机化的方法使程序以不同的形态(每次执行随机改变代码位置)出现，使攻击者难于确定可复用代码的位置，从而提高攻击难度。但通过信息泄漏攻击仍能绕过 ASLR 的防护[21,22]。

在攻击的第 4 步，攻击者使用重写的返回地址或函数指针将控制流转移到攻击者指定的位置。CFI[23]被认为是检测异常控制流转移的通用方法，这种方法保证程序的控制流沿着应用的控制流图(control-flow graph，CFG)确定的路径执行。LLVM 编译器从 2014 年开始支持 CFI[24]。微软从 2014 年开始，在 Windows 中增加 CFI 的保护 CFG[25]。CFG 在间接函数调用前检查目标地址是否是进程位图上的合法地址。攻击者很快利用 CFG 的实现漏洞绕过其保护。这些方法存在两个问题：(1) 因为 C/C++程序中普遍存在指针类型转换，很难得到准确的 CFG[26]；(2) 利用软件方式实现的 CFI 本身存在漏洞，容易被攻击者绕过[13,27]。

在攻击的第 5 步，按攻击者的意图执行恶意注入的代码或复用已有的代码。DEP 能有效防止注入代码的执行，但不能防止执行复用的代码。若攻击者成功实现了前五步，他已经成功挟持了控制流。在第六步，可通过隔离(isolation)限制攻击者的能力。例如使用软件错误隔离(software fault isolation，SFI)[28]限制带有攻击的代码影响其它地址空间。Intel 的 SGX 则防止攻击者破坏安全敏感地址空间的存储[29]。

针对每步的防御机制可用纯软件实现，也可以基于硬件实现。表 1 中对典型的基于软件的防御机制与基于硬件的防御机制进行比较，其中基于纯软件的防御机制用灰色标记。从表中可以看出，基于纯软件的防御机制性能开销较大，有些方法仍能被攻击者绕过。基于硬件的防御机制不仅有效防止 CRA，并且带来性能开销较小。

表 1　典型基于软件与硬件防御机制比较

防御策略	实现方法	性能开销	防御效果
存储安全	SoftBound[30]	67%	完全防止存储错误
	HardBound[31]	5%	完全防止存储错误
CPI	CPI[16]	8.4%	受信息泄漏攻击
	HDFI[32]	<2%	有效防止 CRA
CFI	CFG[25]	--	被攻击者绕过
	Abadi 等[33]	16%	有效防止 CRA
	Sullivan 等[34]	1.75%	有效防止 CRA

2　基于硬件的防御机制

本节详细介绍图4所示各攻击阶段基于硬件的CRA防御机制。

2.1　基于硬件的存储安全

可以通过多种方式实现存储安全,包括纯静态分析,扩展C/C++语言,运行时检查等[2]。基于硬件的存储安全机制采用运行时检查的实现方式。硬件实现方式的原理与纯软件实现的机制相同[30,31,35-37]。二者都通过检查指针引用的正确性发现存储错误。为了检查指针引用的正确性,二者为指针绑定一个元数据(meta-data),元数据中包括指针的边界与标记等。边界信息用于检查空间上(如缓冲区溢出)的存储错误,标记用于检查时间上(如释放后利用)的存储错误。硬件对存储安全检查的支持有效降低性能开销[30,37]。

存储安全检查的关键问题是如何存放元数据。目前,元数据存储方式有两种:(1)与指针一起存储,称为胖指针;(2)元数据存储在与指针相对应的区域,通过查找表的方式索引,称为分离存储。相对分离存储方式,胖指针能更快找到元数据,引入的性能开销较小。但胖指针存储方式改变了存储的布局,与现有程序不兼容。为了兼顾兼容性与性能,HardBound[31]与Watchdog[36]等将元数据存储在与指针对应的位置,通过增加缓存的方式降低性能开销。在2013年,Intel将存储安全保护增加到指令集架构中,即内存保护扩展(memory protection extension, MPX)[38,39]。

本文以MPX为例介绍基于硬件的存储安全的实现方法。MPX的元数据为指针的上下边界,因此只能检测空间上的存储错误。为了降低性能开销,MPX增加了7条新指令和一组128位的边界寄存器(Intel Skylake架构有4个寄存器)。寄存器的低64位存下边界,高64位存上边界。新的指令实现边界创建,验证指针边界,边界的载入与存储,边界传递等功能。进程的边界信息保存在边界表中,边界表的组织结构类似于页的组织结构,由操作系统管理。虽然MPX有硬件支持,但其平均性能开销为50%,在实际应用中仍不能使用。

2.2　基于硬件的代码指针完整性

MPX性能开销大的原因在于所有基于指针操作都增加边界检查。Kuznetsov等[16]用形式化方法证明,只需保证安全敏感指针的完整性就可以防止CRA,这种方法称为CPI。安全敏感指针类型包括:指向函数的指针,指向安全敏感类型的指针,指向包含一个或多个安全敏感类成员的结构体,或指向前几种类型的无类型指针(*void)。它们在SPEC 2006中识别的敏感指针只占总指针数的6.5%,这个结果表明CPI可以大大减少存储安全检查的性能开销。然而,软件方法实现的CPI通过信息隐藏保证指针元数据的安全,攻击者仍可以绕过基于信息隐藏的防护[16,17]。因此需要额外机制保证指针元数据的安全。

目前硬件机制只检查间接跳转目标地址与返回地址完整。Cryptographic CFI (CCFI)[19]通过AES加密算法保证指针元数据的安全,保护的指针只限于函数指针,返回地址,虚表(vtable)指针与虚函数指针。CCFI的元数据为指针的信息认证码(MAC)。MAC是AES加密计算得到的,可以保存到任何位置。因为MAC中包含了指针类型、指针地址及指针值三部分内容,可以有效防止控制流挟持攻击。CCFI利用Intel处理器的加密指令(AES.NI)加速MAC计算速度,但平均性能开销仍超过50%。另一方面,CCFI在程序开始时随机产生密钥,在进程上下文切换时需操作系统管理密钥。

为了将密钥管理与操作系统隔离,RAGuard[40]设计了基于物理不可克隆函数(PUF)的密钥管理装置。在进程创建或重新开始运行时,将进程特征信息(如进程号)作为PUF的输入,以PUF的输出作为计算MAC的密钥。RAGuard将系统的可信计算基础限定在系统处理器。

在2016年,ARM在ARMv8.3-A指令集架构中增加了指针认证(pointer authentication, PA)指令[41],高通发布了PA的白皮书[42]。与CCFI和RAGuard类似,PA用加密算法计算MAC(pointer authentication code, PAC)认证指针。比较巧妙的是,PA利用64位架构下用到的实际地址少于64位

的特点,将指针的 PAC 保存在指针值的高位。这样可以在指针写入内存之前为每个需要保护的指针插入 PAC,并且不增加存储开销,也不需要像 MPX 那样由操作系统管理元数据。另一方面,PAC 与指针值一起存在内存,需要在提交写命令前完成 PAC 计算。为了满足这一要求,高通采用轻量级的加密算法 QARMA[43]。

2.3 基于硬件的攻击特征检查(粗粒度 CFI)

由于实现完整的 CFI 性能开销较大,早期的 CRA 防御机制主要基于攻击特征的检查,因此可以看成粗粒度的 CFI。攻击特征包括两个方面:(1)实施 CRA 时指令序列的特征。这种方法主要针对 ROP 攻击,ROP 攻击总是执行一系列以 *ret* 结尾的指令序列,并且指令序列中指令数目较少;(2)程序正常执行的安全规则。这些规则包括:(Ⅰ)返回指令的目标地址必须是相应函数调用指令保存的返回地址;(Ⅱ)间接跳转分支指令的目标地址必须为函数的入口地址,或在相同函数内部;(Ⅲ)间接函数调用指令的目标地址必须为函数的入口地址。这些目标地址可以通过静态分析二进制文件得到。

除了定义攻击特征,基于硬件的攻击特征检查还需完成两个工作:运行时信息收集与异常判断。第一项工作可以通过处理器现有的硬件装置在程序执行时实时收集,如 LBR (last branch record)与 BTS (branch trace store)。除了使用处理器已有的硬件装置外,还可以通过扩展处理器的流水线结构实现。当用已有硬件装置收集信息时,需要扩展操作系统内核,增加异常检测的功能。当通过扩展处理器流水线结构实现时,则可用硬件直接检测异常。

kBouncer[44]利用 LBR 收集系统调用前的 16 条返回指令,由内核中的模块根据收集的返回指令序列判断是否存在 ROP 攻击。为了减少性能开销,kBouncer 只检查可能产生危害的系统调用。与 kBouncer 根据 ROP 攻击时指令序列的特征检测攻击不同,CFIMon[45]根据程序正常执行的安全规则检测攻击。因此,kBouncer 只能检测 ROP 攻击,而 CFIMon 能够检测所有类型的 CRA。CFIMon 利用 BTS 收集所有的控制流转移事件(函数调用,函数返回与各类间接分支跳转)。在程序运行时,由 BST 收集控制流信息并写入到内存中。BST 快满时,由操作系统将控制流信息拷贝到用户空间,由用户空间的监视器判断是否存在攻击,应用这种批处理的方式可降低检测的性能开销。Kayaalp 等[46]通过扩展处理器流水线结构实时检测控制流转移是否符合程序正常执行的规则。这种方法不需要静态分析控制流转换的目标地址,只需要在可执行文件中标记出函数的开始与结束的位置。

虽然基于攻击特征检查提高了 CRA 的难度,但这种机制只能实现粗粒度的 CFI,攻击者仍能绕过这些机制的保护并实现完备的 CRA[47,48]。因此,现在的研究主要集中在实现细粒度的 CFI。

2.4 基于硬件的细粒度 CFI

CFI 由 Abadi 等[23,33]在 2005 年提出。实现 CFI 包括两个阶段:(1)在程序运行之前通过静态分析程序的源代码或二进制文件构建程序的 CFG;(2)在运行时检查控制流转移是否在 CFG 的边上。运行时 CFI 实现方法如图 5 所示。图中由间接函数调用 *call*(或间接分支跳转指令 *jmp*)指令引起的控制流称为前向控制流,由 *ret* 指令引起的控制流称为后向控制流。对于前向控制流,根据 CFG,为每个目标地址前插入标记(即图中 *prefetchnta* 指令),在控制流转移前插入检查标记的指令(即图中 *cmp* 指令)。若 *cmp* 指令中的标记与 *prefetchnta* 指令预取的标记一致,则控制流转换是合法的,否则产生错误。因为一个程序可能被很多位置调用,如果与前向 CFI 一样用标记检查,则这个程序有很多合法的返回位置,并且 *ret* 指令前也会增加很多 *cmp* 指令。因此通过用影子栈(shadow stack)记录函数调用的顺序保证后向 CFI。

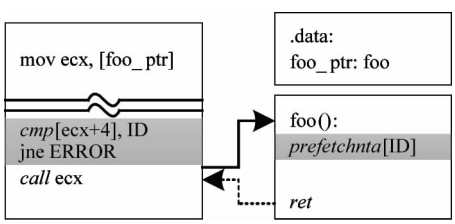

图 5 CFI 实现示例

图 5 中的例子采用二进插桩的方式实施 CFI 检查,这种方式引起 15% 的性能开销[33]。为了降低

性能开销,Budiu 等[49]在体系结构中增加 CFI 检查的指令。他们为每个分支目标之前插入一个标记检查指令(*cfilabel* L),每个间接跳转指令用相应的 CFI 指令代替(*jmpc* reg, L)。当执行 CFI 的跳转指令时,将指令包含的标记设置到 CFI 寄存器中。在这条指令执行完后,处理器的状态转变为只能执行 *cfilabel* 指令的状态。此时只有执行的 *cfilabel* 的标记与 CFI 寄存器中保存的标记相匹配时才能回到正常执行状态。CFI 检查指令对实现前向 CFI 非常有效,在 2016 年,Intel 在体系结构中增加了 CFI 检查的指令,并增加了实现后向 CFI 的影子栈,这一技术称为控制流强制技术(control-flow enforcement technology,CET)[50]。

目前基于硬件的 CFI 面临两个问题:(1) CFI 的安全性依赖于所构建 CFG 的准确性,但由于 C/C++代码中有各种类型转换,构建精确的 CFG 并不容易[26]。另外,即使能够构建最精确的 CFG,攻击者仍能在通过 CFI 检查的情况下挟持控制流[51,52];(2) 影子栈被认为是实现后向 CFI 的关键部件,但用软件方法实现的影子栈需要额外的存储安全保护,并且性能开销较大(大约 10%)[53]。硬件方法实现的影子栈的容量有限,在影子栈溢出和上下文切换时需要操作系统处理。另外不管是软件还是硬件实现的影子栈都需要处理 setjmp/longjmp 这类特殊情况,这些特殊情况处理过程中容易产生误判[40,54]。

以上几种方法的安全性均依赖于 CFG 的准确性。为了解决这一问题,研究者通过收集程序运行时的 trace 得到程序的合法控制流。Shi 等[55]通过扩展处理器的流水线结构检测控制流挟持攻击。他们用分支指令地址(BPC),分支目标地址(TPC)和执行路径(EP)标记控制流(BPC||TPC||EP),通过训练收集控制流信息。这种方法是对处理器分支预测单元的扩展,并没有带来较大的硬件开销。Liu 等[56]与 Ge 等[57]用 IPT (Intel processor trace)收集运行时控制流信息。通过将运行时控制流信息与离线训练得到的 CFG 比较检测出异常控制流。这两种方法的信息收集与检测均需操作系统参与,因此只能检测用户空间的异常控制流。

2.5　数据执行保护(DEP)

从 Window XP 与 Windows Server 2003 开始,操作系统中增加了 DEP[58]这一系统级的存储保护特征。DEP 在页表项中增加了不可执行位(NX 位),使系统可以将一个或多个页标记为不可执行,这项技术能够帮助防止恶意代码注入攻击。为支持 DEP,目前所有处理器架构都在页表项中增加了不可执行位。在硬件的支持下,DEP 的保护几乎不增加任何性能开销。但不可执行位的设置是由操作系统管理,攻击者能利用 CRA 将 EDP 保护功能关闭,如表 2 所示为利用 CRA 绕过 DEP 保护的例子。

表 2　利用 CRA 绕过 DEP 保护的例子

CVE ID	软件	漏洞
CVE-2013-3897[59]	Microsoft IE	释放后使用
CVE-2015-3113[60]	Adobe Flash	堆缓冲区溢出
CVE-2016-9888[61]	Mozilla Firefox	释放后使用

2.6　隔离技术

2.1~2.5 节所述的防御机制对静态产生的代码很有效,但不能有效防止针对动态代码的攻击,或带来较大性能开销。基于隔离技术能有效保证动态代码的正确执行[62]。另一方面,虽然隔离技术不能直接阻止 CRA,但能有效降低 CRA 成功后对系统的危害。例如:现在的处理器架构与软件层基于包含的权限实现,当攻击者挟持了高权限层的软件时,可以毫无限制地操作或获取其管理的低权限层的代码和数据。基于硬件的隔离技术可以防止被攻破的特权层随意获取用户(用户虚拟机或进程)的信息,因此隔离技术可以看成 CRA 的最后防线。基于硬件的隔离技术分为两种:利用存储加密实现的隔离与基于软件不可访问硬件实现的隔离。

2.6.1　存储加密隔离

当数据以明文形式存放在内存中时,攻击者可能利用漏洞或用物理方法泄露内存中的数据。存储加密可以被用来防止内存中数据被非法访问。XOM[63],AEGIS[64]与 SecureBlue[65]等利用存储加密的方式隔离不同的应用。这些机制在处理器内部共享的资源(如通用寄存器,TLB 或 cache 行)增加标记,用于标记当前活跃的应用。应用的标记与应用

存储加密密钥对应,这样就可以防止受保护应用的数据被其它应用获得。为了减少存储加密对系统性能的影响,在总线与内存之间增加硬件加密引擎。

加密方法保证了存储数据的保密性,但直接加密方式并不能保证存储数据的完整性,因为存储中的数据有可以被欺骗攻击、重组与拼结攻击或重放攻击修改。为了防止这些攻击,麻省理工大学的 Gassend 等[66]提出用 Merkle 树验证存储数据的完整性。Merkle 树是一个 MAC 值的层次树,树的叶子节点对应的是存储块,根节点总是保存在芯片内部,只能被处理器更新。Merkle 树的结构如图 6 所示。利用 Merkle 树验证是一个迭代的过程,当数据从外部存储器取回后,计算其 MAC,与保存的数据块的 MAC 比较,若匹配,则取其兄弟节点的 MAC,计算一个新的 MAC 值,再与其父节点比较,依此类推,直到与根节点比较,中间若有不匹配,则存储完整性验证失败。

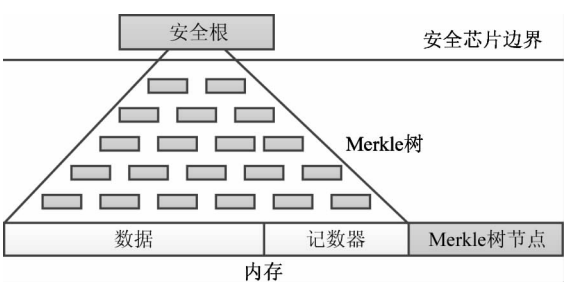

图 6 Merkle 树的结构

目前 Intel 与 AMD 均在存储控制器中增加了硬件加密引擎。Intel 的加密引擎称为 MEE(memory encryption engine)[67],是 SGX(software guard extension)[29]技术的一部分。AMD 的硬件加密引擎可以提供安全存储加密(secure memory encryption, SME)与安全加密虚拟化(secure encrypted virtualization, SMV)两种方式[68]。前者可以实现全内存加密,或由操作系统在页表项中指定对某个页面加密;后者通过加密方式将不同特权级隔离,即使是最高特权级的 Hypervisor 也不能访问用户加密的信息。

2.6.2 硬件辅助的隔离

在基于包含的权限系统架构中,高权限层的软件(Hypervisor 或操作系统)既负责资源(主要指存储)的分配,同时又负责资源的隔离。例如:Hypervisor 决定将某个页分配给哪个虚拟机,同时为每个虚拟机维护一个嵌套页表(nested page table)。为了将高权限软件的资源分配与资源隔离的功能分离,H-SVM[69],HyperWall[70],NIMP[71]和 SGX[29]等利用只有硬件才能访问的数据结构保存与系统安全相关的信息,对高权限层软件的操作加以验证。这样既提高了系统的安全性,同时保留了系统软件资源分配的灵活性。另一方面,硬件辅助的隔离可以为安全敏感的应用提供安全的执行环境,Frassetto 等[62]利用 SGX 在一个隔离的执行环境执行即时编译器,在保证即时编译器正确执行的基础上,将动态产生的代码放到攻击者无法找到的位置执行。

3 分析与讨论

本节首先比较各阶段基于硬件的防御机制,分析当前基于硬件防御机制存在的问题与解决思路。然后总结基于硬件机制的优点及其与软件的关系。

3.1 各阶段防御机制比较

表 3 对各阶段基于硬件防御机制进行比较,其中用灰色标记的实现方法已增加到商用处理器体系结构中。首先从防止 CRA 的有效性对各阶段方法进行比较。基于存储安全与 CPI 的机制可以防止所有控制流挟持攻击。基于攻击特征的防御方法是实现粗粒度的 CFI,已经证明 CRA 可绕过这些机制的保护[47,48]。CFI 的安全性依赖于 CFG 的精确度,但因为C/C++代码中存在各种类型转换,构建完全精确的 CFG 是非常困难的[26]。不可执行栈只能防止代码注入攻击,不能防止 CRA。基于隔离的机制不能阻止控制流挟持攻击,主要为了限制攻击者挟持控制流后对系统的伤害,与其它防御机制是互补的关系。因此,只有基于存储安全与基于 CPI 的实现方法能够阻止现有控制流挟持攻击。从性能上比较,基于 CPI 的实现方法更优于其它的基于存储安全的实现方法。实际上基于 CPI 的方法可以看成基于存储安全实现方法的特殊情况:CPI 只保护与控制流相关的指针完整性,可在很大程度上减少性能开销。

表 3　各阶段基于硬件防御机制比较

防御策略	实现方法	硬件的功能	扩展处理器架构?	防止所有 CRA?	TCB 包含系统软件?	平均性能开销
存储安全	MPX	减小性能开销	增加新指令	是	是:OS 管理边界表	50%
CPI	CCFI	加密算法加速	否	是	是:OS 管理密钥	52%
	RAGuard	加密算法加速, 收集微架构信息	修改流水线	否:不能防止 JOP 与 COOP	否	1.86%
	PA	加密算法加速	增加新指令	是	是:OS 管理密钥	--
攻击特征	kBouncer, CFIMon	收集微架构信息	否	否:实现粗粒度 CFI, 不能完全防止 CRA	是:运行时控制流检查	4% 6.1%
	BR	收集微架构信息	修改流水线		是:OS 处理异常	2%
CFI	FlowGuard, GRIFFIN	收集微架构信息	否	否:依赖于收集运行时控制流信息的准确性	是:运行时控制流检查	11.9%（int） 3.79%
	SmashGuard	减小性能开销	修改流水线	否:不能防止 JOP 与 COOP	是:OS 负责影子栈的溢出与上下文切换	1.8%
	CET	减小性能开销	增加新指令	否:依赖于 CFG 的准确性	是:OS 管理影子栈页面属性,处理违反 CFI 的异常	--
不可执行栈	DEP	减小性能开销	修改流水线	否:不能防止 CRA	是:不可执行位的设置是由操作系统管理	--
隔离技术	H-SVM, HyperWall, NIMP	硬件辅助隔离 收集微架构信息	修改流水线	否:隔离技术不直接阻止 CRA。但可提供安全执行环境,保障安全敏感应用正常执行[62];限制攻击者挟持控制流后对系统的伤害	否	-- -- 0.6%
	SME/SMV	加密算法加速	修改存储控制器		否	--
	SGX	加密算法加速, 硬件辅助隔离	增加新指令 修改流水线 修改存储控制器		是:开发者定义的被保护的代码	--

虽然基于 CPI 的方法最有希望防御所有控制流挟持攻击,但目前基于 CPI 的硬件防御机制还存在不足。CCFI[19]利用 Intel 的 AES. NI 减少计算 MAC 的性能开销,但是其平均性能开销仍大于 50%。另外 CCFI 依靠 OS 管理密钥。与 CCFI 相比,RAGuard[40]将系统 TCB 限定在处理器硬件,但这种机制只保证了后向 CFI,还需要增加对前向控制流的保护。PA[42]利用64 位架构下用到的实际地址少于 64 位的特点,巧妙地将 MAC 保存在指针值的高位,这样既减少了存储 MAC 的空间与读写 MAC 带来的性能开销,也不需要操作系统管理 MAC。但 PA 按指针类型管理密钥,它有可能受到重放攻击威胁。

3.2　硬件的功能

从表 3 中可以看出,硬件在防御 CRA 主要实现以下 4 种功能:

(1) 减小性能开销:与硬件相比,软件方法更灵活。许多防御机制最初是由软件实现,但因性能开销较高而不能被接受。为了降低性能开销,通过扩展处理器体系结构(修改流水线或增加指令),在硬件上支持相关安全机制。例如:存储安全的边界检查机制,最初为 CCured 与 Cyclone 等纯软件方法,为了降低性能开销,HardBound 通过扩展指令支持指针的边界检查。Intel 在这些研究的基础上,将存储保护增加到指令集架构中,即 MPX。

(2) 微架构信息收集:商用主流处理器提供了 LBR, BTS 与 IPT 等硬件装置收集微架构信息,包括:分支指令及其目标指令的地址、读/写内存的次数、cache 的命中次数、系统调用序列等。这种方式不需要修改处理器流水线,但用户态到内核态的切换会带来性能开销,需要研究减少性能开销的手段。若采用扩展处理器流水线结构的方式,则能以较少的代价收集丰富的信息,但需要修改处理器硬件,即使被芯片制造商接受也需经过较长的时间才能商用,并且往往只对某一种攻击有效。

(3) 加速加密算法:加密算法是一种有效的隔离手段。在保护重要信息的同时,又能充分利用软件方法的灵活性。若采用软件方法实现加密算法,则会带来较大的性能开销。因此,内置加密引擎越来越受青睐。目前 Intel, AMD 与 ARM 均在存储控制器中增加加密引擎。

(4) 隔离执行环境:从本质上看,处理器的硬件比软件的漏洞要少,实施硬件攻击的难度与成本比软件攻击大很多。另外,硬件处于系统最底层,可不受软件影响。因此,基于硬件或硬件辅助/增强的安全机制比纯软件实现的安全机制更安全。例如,攻击者利用微软 CFG 实现方法的漏洞很快绕过其保护[13,27]。SGX 等基于硬件的隔离执行环境将系统可信计算基础缩小,即使特权级代码存在安全问题,也能保障被保护代码的安全性。

3.3 硬件与软件的关系

上述的基于硬件的防御机制并不是完全由硬件实现。表 4 列出了 4 种商用基于硬件安全机制中软件实现功能。MPX[38] 与 CET[50] 通过新指令降低防御控制流挟持的开销。MPX 需要软件为其管理边界表,CET 需要软件为其管理影子栈的页面,并且二者都需要软件为其处理异常。对于 TrustZone 与 SGX,软件基于硬件机制创建可信执行环境。由此可见,只有将硬件与软件相互配合,才能以较小的性能与成本代价构建安全的系统架构。因此,需要从系统架构设计整体考虑安全机制的实现,合理划分硬件与软件的功能。

表 4 商用基于硬件安全机制中软件实现功能

基于硬件安全机制	软件实现功能
MPX	边界表的页面管理及违反边界条件的异常处理
CET	影子栈页面属性管理及违反 CFI 的异常处理
TrustZone	监视器:安全操作系统与普通操作系统的切换 TEE:安全操作系统及其上运行的可信应用
SGX	不可信运行时系统:由 Encalve 外应用调用的库函数,如 Enclave 的创建与销毁等 可信库:与 Enclave 静态链接的库,包括可信运行时系统、可信服务库与可信平台服务函数等 SGX 驱动:Enclave 的页分配与载入等

4 未来的研究方向

可以预见,基于硬件的安全防御将是趋势,基于以上章节的总结与分析,本研究认为未来基于硬件的防御机制研究分为以下 3 个方面:

(1) 基于硬件的代码指针完整性:通过对现有基于硬件的 CRA 防御方法比较,本文认为基于硬件的代码指针完整性有望以较小性能开销完全防止 CRA。本文实现基于硬件实现代码指针完整性的思路如下两方面:定义验证指针完整性的元数据,元数据的内容要能够代表指针在空间与时间上的特征,并且这些内容能够被硬件获得;设计基于元数据的代码指针完整性的验证方法。二者要与现有的体系结构兼容,不会带来较大性能开销。

(2) 针对非控制数据攻击的防御:虽然针对控制流挟持的防御机制还有不足,但已经大大提高了控制流挟持攻击的难度,使攻击者开始研究针对非控制数据的攻击。与控制流挟持攻击修改代码指针不同,针对非控制数据的攻击利用存储错误修改数据。例如:通过修改关键函数(如 C 函数库中的 execve())的参数执行任意程序。攻击者还可以通过重写与系统安全相关的配置,将安全检查关闭。目前的研究重点是保护代码指针,针对非控制数据攻击的防御研究较少。本文认为针对非控制数据的攻击与防御有可能成为新的研究热点。

(3) 可配置的硬件防御架构:基于存储错误的漏洞利用与防御是从不停歇的战争,当阻止一类攻击后,又会发展出新的攻击方式。目前除了基于隔离的防御机制外,大部分基于硬件的防御机制大多只能阻止一种攻击。另一方面,基于硬件的机制不如基于软件方法灵活。软件方法若出现漏洞,可以通过打补丁解决。若硬件机制出现问题,则需要修改处理器架构。因此,基于硬件的安全机制需要经过长时间的研究才会被接受。为了防御多种攻击,并缩短基于硬件防御机制的实现周期,本文认为很有必要开展可配置的硬件防御机制的研究。

5 结 论

本文介绍了代码复用攻击(CRA)挟持控制流的机制,同时介绍了当前 CRA 防御机制的研究现状,详细介绍了基于硬件的 CRA 防御机制。通过对现有基于硬件机制的分析与讨论,得到如下结论:(1) 基于硬件的代码指针完整性能在较小性能开销

下全面防止控制流挟持攻击;(2) 相对于软件实现方法,硬件机制有明显优势,例如降低实现开销,收集微架构信息,加速加密算法,从底层提供隔离;(3) 基于硬件的防御机制是软件与硬件相结合的系统,硬件从体系结构上支持安全,而软件则保留了系统操作的灵活性。当前基于硬件防御机制大多只能防御一种攻击,并且需要较长时间才能被接受,本文认为研究可配置的硬件防御架构很有必要。

参考文献

[1] CVE Details. Vulnerabilities by type[EB/OL]. http://www.cvedetails.com/vulnerabilities-by-types.php: CVE Details, 2017

[2] Szekeres L, Payer M, Wei T, et al. SoK: Eternal war in memory. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy, San Francisco, USA, 2013. 48-62

[3] Boettger L. The morris worm: how it affected computer security and lessons learned by it[EB/OL]. http://people.cs.vt.edu/~kafura/cs6204/Readings/Context-Problems/MorrisWorm.htm: VirginiaTech, 2000

[4] NC State University. What is the slammer worm/SQL worm/sapphire worm? [EB/OL]. https://ethics.csc.ncsu.edu/abuse/wvt/Slammer/study.php: NC State University, 2001

[5] Davi L. Code-reuse attacks and defenses:[Ph. D dissertation][D]. Duisburg: Technische Universitat Darmstadt, 2015. 7-26

[6] Shacham H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)[C]. In: Proceedings of ACM Conference on Computer and Communications Security, Alexandria, USA, 2007. 552-561

[7] Bletsch T, Jiang X, Freeh V, et al. Jump-oriented programming: A new class of code-reuse attack[C]. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, Chicago, USA, 2011. 30-40

[8] Schuster F, Tendyck T, Liebchen C, et al. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications[C]. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy, Sanjose, USA, 2015. 745-762

[9] 柳童, 史岗, 孟丹. 代码重用攻击与防御机制综述. 信息安全学报, 2016, 1(2): 15-27

[10] 马梦雨, 陈李维, 孟丹. 内存数据污染攻击和防御综述. 信息安全学报, 2017, 2(4): 82-98

[11] 陈小全, 薛锐. 程序漏洞: 原因、利用与缓解. 信息安全学报, 2017, 2(4): 41-56

[12] Wikipedia. Return-to-libc attack[EB/OL]. https://en.wikipedia.org/wiki/Return-to-libc_attack: Wikipedia, 2009

[13] Spisak M. Disarming control flow guard using advanced code reuse attacks[EB/OL]. https://www.endgame.com/blog/technical-blog/disarming-control-flow-guard-using-advanced-code-reuse-attacks: ENDGAME, 2017

[14] Jim T, Morrisett J G, Grossman D, et al. Cyclone: a safe dialect of C[C]. In: Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference, Philadelphia, USA, 2002. 275-288

[15] Necula G C, McPeak S, Weimer W. CCured: Type-safe retrofitting of legacy code. *SIGPLAN Not*, 2002, 37(1): 128-139

[16] Kuznetsov V, Szekeres L, Payer M, et al. Code-pointer integrity[C]. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, Broomfield, USA, 2014. 147-163

[17] Evans I, Fingeret S, Gonzalez J, et al. Missing the point (er): On the effectiveness of code pointer integrity[C]. In: Proceedings of 2015 IEEE Symposium on Security and Privacy, San Jose, USA, 2015. 781-796

[18] Göktas E, Gawlik R, Kollenda B, et al. Undermining information hiding (and what to do about it)[C]. In: Proceedings of the 25th USENIX Security Symposium, Austin, USA 2016. 105-119

[19] Mashtizadeh A J, Bittau A, Boneh D, et al. CCFI: Cryptographically enforced control flow integrity[C]. In: Proceedings of the 22nd ACM Conference on Computer and Communications Security, Denver, US, 2015. 941-951

[20] Wikipedia. Address space layout randomization[EB/OL]. https://en.wikipedia.org/wiki/Address_space_layout_randomization: Wikipedia, 2003

[21] Rudd R, Skowyra R, Bigelow D, et al. Address oblivious code reuse: on the effectiveness of leakage resilient diversity[C]. In: Proceedings of the 2017 Network and Distributed System Security Symposium, San Diego, USA, 2017

[22] Gras B, Razavi K, Bosman E, et al. ASLR on the line: Practical cache attacks on the MMU[C]. In: Proceed-

ings of the 2017 Network and Distributed System Security Symposium, San Diego, USA, 2017

[23] Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity principles, implementations, and applications[J]. *ACM Trans Inf Syst Secur*, 2009, 13(1): 4:1-4:40

[24] Tice C, Roeder T, Collingbourne P, et al. Enforcing forward-edge control-flow integrity in GCC & LLVM[C]. In: Proceedings of the 23rd USENIX Security Symposium, San Diego, USA, 2014. 941-955

[25] Microsoft. Control flow guard[EB/OL]. https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx: Microsoft, 2014

[26] Evans I, Long F, Otgonbaatar U, et al. Control jujutsu: on the weaknesses of fine-grained control flow integrity [C]. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, USA, 2015. 901-913

[27] FREEBUF. Windows10 系统的控制流防护机制初窥[EB/OL]. http://www.freebuf.com/articles/system/58138.html: FREEBUF, 2015

[28] Castro M, Costa M, Harris T. Securing software by enforcing data-flow integrity[C]. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, Seattle, USA, 2006. 147-160

[29] Costan V, Devadas S. Intel SGX explained[EB/OL]. https://eprint.iacr.org/2016/086: Cryptology ePrint Archive, 2016

[30] Nagarakatte S, Zhao J Z, Martin M M K, et al. SoftBound: Highly compatible and complete spatial memory safety for C[C]. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, Ireland, 2009. 245-258

[31] Devietti J, Blundell C, Martin M M K, et al. Hardbound: Architectural support for spatial safety of the C programming language[C]. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, Seattle, USA, 2008. 103-114

[32] Song C, Moon H, Alam M, et al. HDFI: Hardware-assisted data-flow isolation[C]. In: Proceeding of the 2016 IEEE Symposium on Security and Privacy, San Jose, USA, 2016. 1-17

[33] Abadi M, Budiu M, Erlingsson ú, et al. Control-flow integrity[C]. In: Proceedings of the 12th ACM Conference on Computer and Communications Security, Alexandria, USA, 2005. 340-353

[34] Sullivan D, Arias O, Davi L, et al. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity[C]. In: Proceeding of the 53nd ACM Design Automation Conference, Austin, USA, 2016. 163:1-163:6

[35] Chuang W, Narayanasamy, Calder B. Accelerating meta data checks for software correctness and security[J]. *Journal of Instruction-Level Parallelism*, 2007, 9(9): 1-26

[36] Nagarakatte S, Martin M M K, Zdancewic S. Watchdog: Hardware for safe and secure manual memory management and full memory safety[C]. In: Proceedings of the 39th Annual International Symposium on Computer Architecture, Portland, USA, 2012. 189-200

[37] Nagarakatte S, Martin M M K, Zdancewic S. WatchdogLite: Hardware-accelerated compiler-based pointer checking[C]. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, 2014. 175-184

[38] arXiv. Intel MPX explained: An empirical study of Intel MPX and software-based bounds checking approaches [EB/OL]. https://arxiv.org/abs/1702.00719v1: arXiv, 2017

[39] Intel. Introduction to Intel memory protection extensions [EB/OL]. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions: Intel, 2013

[40] Zhang J, Hou R, Fan, J F, et al. RAGuard: A hardware based mechanism for backward-edge control-flow integrity [C]. In: Proceedings of the Computing Frontiers Conference, Siena, Italy, 2017. 27-34

[41] ARM. Armv8-A architecture - 2016 additions[EB/OL]. https://community.arm.com/processors/b/blog/posts/armv8-a-architecture-2016-additions: ARM, 2016

[42] Qualcomm Technologies, Inc. Pointer authentication on ARMv8.3[EB/OL]. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf: Qualcomm Technologies, Inc., 2017

[43] Avanzi R. The QARMA blockcipher family: almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency S-Boxes [J]. *IACR Cryptology ePrint Archive*, 2016, 2016: 444

[44] Pappas, V, Polychronakis M, Keromytis A D. Transparent ROP exploit mitigation using indirect branch tracing [C]. In: Proceedings of the 22nd USENIX Conference

on Security, Washington, USA, 2013. 447-462

[45] Xia Y B, Liu Y T, Chen H B, et al. CFIMon: Detecting violation of control flow integrity using performance counters[C]. In: Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Boston, USA, 2012. 1-12

[46] Kayaalp M, Ozsoy M, Abu-Ghazaleh N, et al. Branch regulation: Low-overhead protection from code reuse attacks[C]. In: Proceedings of the 39th Annual International Symposium on Computer Architecture, Portland, USA, 2012. 94-105

[47] Carlini N, Wagner D. ROP is still dangerous: Breaking modern defenses[C]. In: Proceedings of the 23rd USENIX Conference on Security Symposium, San Diego, USA, 2014. 385-399

[48] Göktaş E, Athanasopoulos E, Polychronakis M, et al. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard[C]. In: Proceedings of the 23rd USENIX Conference on Security Symposium, San Diego, USA, 2014. 417-432

[49] Budiu M, Erlingsson U, Abadi M. Architectural support for software-based protection[C]. In: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Pendability, San Jose, USA, 2006. 42-51

[50] Intel. Control-flow enforcement technology preview[EB/OL]. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf: Intel, 2017

[51] Conti M, Crane S, Davi L, et al. Losing control: On the effectiveness of control-flow integrity under stack attacks [C]. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, USA, 2015. 952-963

[52] Carlini N, Barresi A, Payer M, et al. Control-flow Bending: On the effectiveness of control-flow integrity[C]. In: Proceedings of the 24th USENIX Conference on Security Symposium, Washington, D.C., USA, 2015. 161-176

[53] Dang T H Y, Maniatis P, Wagner D. The performance cost of shadow stacks and stack canaries[C]. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, Singapore, 2015. 555-566

[54] Ozdoganoglu H, Vijaykumar T N, Carla E K., et al. SmashGuard: A hardware solution to prevent security attacks on the function return address. *IEEE Transactions on Computer*, 2006, 55(10): 1271-1285

[55] Shi Y X, Lee, G. Augmenting branch predictor to secure program execution[C]. In: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Toulouse, Frence, 2007. 10-19

[56] Liu Y T, Shi P, Wang X, et al. Transparent and efficient CFI enforcement with Intel processor trace[C]. In: Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture, Austin, USA, 2017. 529-540

[57] Ge, X Y, Cui W D, Jaeger T. GRIFFIN: Guarding control flows using Intel processor trace[C]. In: Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems, Xi'an, China, 2017. 585-598

[58] Wikipedia. Executable space protection[EB/OL]. https://en.wikipedia.org/wiki/Executable_space_protection#Windows: Wikipedia, 2017

[59] CVE Details. CVE-2013-3897[EB/OL]. http://www.cvedetails.com/cve/CVE-2013-3897/: CVE Details, 2013

[60] CVE Details. CVE-2015-3113[EB/OL]. http://www.cvedetails.com/cve/CVE-2015-3113/: CVE Details, 2015

[61] CVE Details. CVE-2016-9888[EB/OL]. http://www.cvedetails.com/cve-details.php?t=1&cve_id=CVE-2016-9888: CVE Details, 2016

[62] Frassetto T, Gens D, Liebchen C, et al. JITGuard: Harding just-in-time compilers with SGX[C]. In: Proceedings of the 24nd ACM SIGSAC Conference on Computer and Communications Security, Dallas, USA, 2017

[63] Lie D, Thekkath C, Mitchell M, et al. Architectural support for copy and tamper resistant software[J]. *SIGPLAN Not*, 2000, 35(11): 168-177

[64] Suh G E, Clarke D, Gassend B, et al. AEGIS: Architecture for tamper-evident and tamper-resistant processing [C]. In: Proceedings of the 17th Annual International Conference on Supercomputing, San Francisco, USA, 2003. 160-171

[65] Boivie R. SecureBlue + +: CPU support for secure execution[EB/OL]. http://domino. research. ibm. com/library/cyberdig. nsf/papers/E605BDC5439097F085257A13004D25CA: IBM Research, 2012

[66] Gassend B, Suh G E, Clarke D, et al. Caches and hash trees for efficient memory integrity verification[C]. In: Proceedings of the 9th International Symposium on High-Performance Computer Architecture, Anaheim, USA, 2003. 295-306

[67] Gueron S. A memory encryption engine suitable for general purpose processors[EB/OL]. https://eprint. iacr. org/2016/204. pdf: IACR, 2016

[68] Kaplan D, Powell J, Woller T. AMD memory encryption [EB/OL]. http://developer. amd. com/wordpress/media/2013/12/AMD _ Memory _ Encryption _ Whitepaper _ v7-Public. pdf: AMD, 2016

[69] Jin S, Ahn J, Cha S, et al. Architectural support for secure virtualization under a vulnerable hypervisor[C]. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, Porto Alegre, Brazil, 2011. 272-283

[70] Szefer J, Lee R B. Architectural support for hypervisor-secure virtualization[C]. In: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, London, England, 2012. 437-450

[71] Elwell J, Riley R, Abu-Ghazaleh N, et al. A non-inclusive memory permissions architecture for protection against cross-layer attacks[C]. In: Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture, Orlando, USA, 2014. 201-212

A survey of hardware mechanisms against code-reuse attacks

Zhang Jun[*,**,***], Hou Rui[**], Zhan Zhiyuan[**], Zhang Lixin[*], Chen Mingyu[*]

([*] Institue of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([**] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093)

([***] University of Chinese Academy of Sciences, Beijing 100049)

Abstract

The process of control-flow hijacking in code-reuse attacks is described, and the current research status of the code reuse attack defense mechanism is introduced. The main problems of the hardware based defense mechanisms are discussed, including memory safety, code pointer integrity, malware detection based on attacks' features, control flow integrity, data execution prevention, and isolation technology. The disadvantages and advantages of hardware defense mechanisms, and the relationship between hardware and software are also analyzed. The future research directions of hardware based security mechanisms are predicted: hardware based code pointer integrity is the most promising technology to prevent all control-flow hijacks; non-control data attacks and their defense will replace code-reuse attacks as a new research topic. The research also points out the necessity of researching configurable hardware based defense architectures.

Key words: control-flow hijacking, memory errors, code reuse attacks, memory safety, code pointer integrity, control-flow integrity (CFI), data execution prevention (DEP), isolation technology