基于指令校验的软硬件协同代码重用攻击防护方法

吕雅帅
(中国人民解放军装备学院 国防科技重点实验室,北京 101416)

摘 要: 面向 x86 处理器的代码重用攻击难于防护的一个重要原因是,在 x86 程序代码中存在大量合法但非编程者预期要执行的指令. 这些在代码中大量存在的非预期指令可被用于构造实现 CRA 的组件. 先前研究均采用软件方法解决非预期指令问题,运行开销大且应用受限. 本文的主要贡献之一是提出了一种低开销的软硬件协同方法来解决 x86 的非预期指令问题. 实验表明,本文的实现方法仅给应用程序带来了-0.093% ~2.993% 的额外运行开销. 此外,本文还提出采用硬件实现的控制流锁定作为一项补充技术. 通过同时采用两个技术,可以极大降低 x86 平台遭受代码重用攻击的风险.

关键词: 代码重用攻击; 非预期指令; 指令校验
中图分类号: TP303 **文献标识码:** A **文章编号:** 0372-2112 (2016)10-2403-07
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2016.10.018

An Instruction Verification Based Hardware/Software Co-design Approach for Mitigating Code-Reuse Attacks

LÜ Ya-shuai
(*Key Laboratory, Academy of Equipment, Beijing* 101416, *China*)

Abstract: Code-reuse attacks (CRAs) are difficult to detect and defend, especially on widely used x86 processors. One reason is that lots of unintended but legal instructions exist in x86 binary codes. The unintended instructions make the finding of so called gadgets for CRAs is much easier than that of RISC processors. Previous studies rely on software-only means to tackle the unintended instruction problem, which makes their approaches are either very costly or can only be applied under restricted conditions. In this paper, we propose a hardware/software co-design approach to tackle the unintended instruction problem. The proposed mechanism has little performance impact on the examined SPEC CPU 2006 benchmarks. We also propose using hardware control-flow locking as a complementary technique. By using the two techniques together, an attacker will have little chance to carry out CRAs on x86 processors.

Key words: code-reuse attack; unintended instruction; instruction verification

1 引言

长期以来,代码注入攻击是最常用的一种恶意代码攻击手段. 为了阻止代码注入攻击,研究者与业界提出并实现了一种名为 W⊕X 的保护机制. W⊕X 机制限制了操作系统一个存储页的写入和执行是互斥的. W⊕X 目前已被主流的处理器和操作系统如 Windows、Linux 等采用. W⊕X 机制基本杜绝了进行代码注入攻击的可能性. 但近年又出现了一种新的攻击手段可以绕过 W⊕X 防护机制,即代码重用攻击(code-reuse attack, CRA). CRA 利用应用程序自身的代码实现恶意攻击,而不用像代码注入那样对运行中的程序进行修改. CRA 最早源于 return-to-libc 技术. 在 return-to-libc 攻击中,攻击者首先破坏运行栈并将控制流转移至一个现有的 libc 函数,然后通过调用系统函数来加载恶意代码并绕过 W⊕X 防护机制. 在 2007 年的论文[1]中,Shacham 提出了 return-oriented programming (ROP)攻击技术. 与 return-to-libc 相比,ROP 攻击不但同样可以绕过 W⊕X

收稿日期:2015-01-28;修回日期:2016-05-22;责任编辑:蓝红杰
基金项目:国家自然科学基金(No.61202129)

机制,并且被证明是图灵完备的. ROP 利用所谓的一系列组件(gadget,以函数返回指令 ret 为结束的短代码序列)来构造恶意代码. 自从 ROP 出现后,研究者提出了一系列应对 ROP 攻击的方法[2,3]. 然而,一种新的 CRA 方法—jump-oriented programming(JOP)[4,5] 又被开发出来. JOP 不需要以 ret 指令作为组件的结束指令. JOP 不但大大降低了在 CRA 中构造恶意代码的困难程度,而且使得原先应对 ROP 攻击的防护技术都完全失效.

在 x86 平台的二进制代码中寻找 CRA 组件要远比 RISC 处理器平台容易. 这是因为在 x86 二进制代码中存在着大量合法但不是编程者预期要执行的指令(unintended instruction). ROP 和 JOP 攻击往往利用这些非预期指令构造 CRA 所需的组件. 目前缺乏有效应对面向 x86 处理器代码重用攻击的一个重要原因就在于,非预期指令问题没有得到很好地解决. 本文的贡献主要有以下几点:(1)提出了一种软硬件协同实现方法,解决了 x86 处理器的非预期指令问题,该方法只需对处理器和操作系统进行细微修改,且不需要对二进制应用程序本身进行任何修改;(2)进一步提出了利用硬件实现的控制流锁定作为一项补充技术,两者相结合可以极大增加面向 x86 处理器代码重用攻击的难度;(3)上述技术通过用周期精确 CPU 模拟器进行测试,实验结果表明,本文提出的方法对应用程序运行性能的影响非常小.

2 相关研究

自从 CRA 出现后,研究者提出了一系列防护技术来应对,但绝大部分针对 ROP 攻击. 文献[6,7]提出用影子栈技术防止攻击者通过破坏运行栈来改变程序的控制流. 文献[2,3]通过监测返回指令的运行状态来探测 CRA 组件,而文献[8,9]则通过监测 call-ret 指令对来检测 ROP 攻击. 另外一些研究通过修改库代码[10,11],消灭那些可能被用作组件的代码片段来降低利用系统库代码进行 CRA 的可能性. 这些研究中所提出的方法主要是应对 ROP 攻击,对 JOP 攻击而言,这些防护技术是无效的.

相对于 ROP 而言,JOP 是较近提出的 CRA 技术,所以针对 JOP 的防护技术比较少. 在已提出的 JOP 防护方法中,文献[12,13]采用的是纯软件方式. 纯软件方式必须依赖动态二进制翻译,这意味着应用程序运行性能会受严重影响. Kayaalp 等在文献[9]中采用的是软硬件相结合的手段. 该方法仅阻止了函数间组件的跳转,但攻击者还是有可能利用一个函数内的组件实现攻击. Kayaalp 等在文献[14]中又提出了一种基于已知 CRA 特征,通过硬件在线识别并阻止 CRA 的方法. 该方法不需要对软件进行修改,但需要用户设定检测阈值来探测 CRA 攻击. 该方法的一个主要缺陷在于要依赖已知的 CRA 特征来检测,对于未知的攻击特征则无法检测.

实现面向 x86 处理器 CRA 攻击的一个重要手段是借助 x86 代码中的非预期指令序列. 在 CRA 被提出之前,有研究者已经意识到非预期指令是 x86 处理器潜在的安全隐患,提出了在应用程序沙箱中对非预期指令进行检测[15]. 这种软件检测非预期指令的方法给应用程序带来的性能损失较大. 与前述研究相比,本文的一个重要贡献是通过软硬件协同手段解决了 x86 处理器的非预期指令这一潜在安全隐患,极大增加了实施 CRA 攻击的难度.

3 方法与实现

本文提出的 CRA 防护方法主要解决两个方面的问题:(1)防止处理器执行非预期指令;(2)防止处理器进行非预期的控制流转移.

3.1 校验非预期指令

像 x86 这样 CISC 处理器的一个重要特点就是指令长度是可变的,每条指令不需要像 RISC 处理器那样对齐到 32 位或 64 位地址边界上,代码段每一个起始地址都有可能是一条新指令的开始. 这造成的一个结果就是一段二进制代码可被解释成多个完全不同的有效指令序列. CRA 正是利用了 x86 处理器的这一特点来构造恶意攻击所需的组件. 图 1 展示了这一现象.

图 1 中反汇编代码片段抽取自 libc-2.12-32 的_IO_vfprintf 函数. 在正常解码的情况下,该片段包含三条指令(图 1 的上半部分所示). 然而,如果从 call 指令的第三个字节开始解码,就是一个完全不同的代码序列(图 1 的下半部分所示). 该序列同样包含三条合法指令,但却不是编程人员想要执行的. 由于最后一条指令恰好为间接跳转指令,因此这三条非预期指令极有可能被用作 CRA 攻击的组件. 值得指出的是,在 x86 平台上从非预期指令中寻找间接跳转指令是非常容易的. 因为字节 FF 正好是间接跳转指令的操作码,而 FF 又恰好是很多指令的操作数. 由此可见,要提升面向 x86 平台的 CRA 防护能力,非预期指令是必须要解决的一个重要问题.

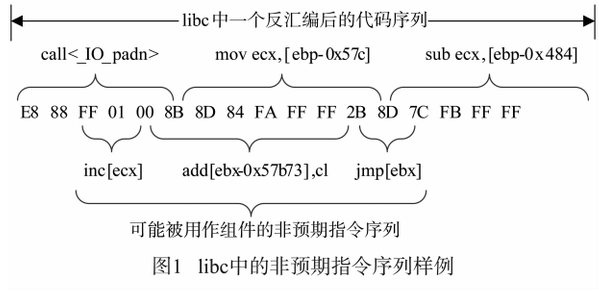

图1 libc中的非预期指令序列样例

3.1.1 PC 校验表

处理器流水线中取指是比较适合进行非预期指令检测的地方. 如何在取指时判断 PC 所指向的指令是否是非预期指令呢? 前面提到过,所有非预期指令都是合法指令,也就是说,仅从指令编码本身是无法作出判断的,只有通过 PC,也就是该指令的地址,才能判断一条指令是否是非预期指令. 那如何判断一个 PC 是否是合法的呢? 本文借助一个合法的 PC 校验列表进行比较.

一种获取合法 PC 列表的方式是在程序运行前对程序的代码段进行反汇编. 当可执行程序被加载至内存时,操作系统会知道每个代码段在虚存中的起始地址. 从某个代码段的首字节开始,依次对每条指令进行反汇编,从而可以获取每条指令的正确起始地址. 当操作系统采用 Address Space Layout Randomization(ASLR)机制时,由于仅是代码段入口地址随机化,且该入口地址对于操作系统是已知的,所以依然可以从代码段入口对指令进行反汇编. 有了合法 PC 列表后,下一个问题是以何种形式保存这个合法 PC 列表. 如果直接以地址形式保存的话,那么处理器在对一个 PC 进行判断时就要在这个列表中进行查找,这种方式显然效率很低,其搜索时间为 O(log n). 另一种方式是用有效位"1"或"0"来表示某个地址是否是一条合法指令的起始地址,如图 2 所示. 通过用图 2 中有效位的方式来表示合法指令,在对一个 PC 进行判断时仅需 O(1)时间.

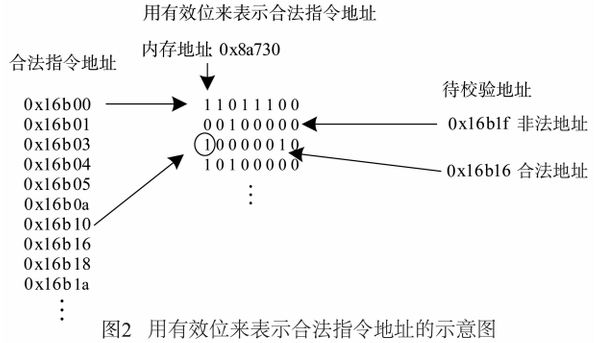

图2　用有效位来表示合法指令地址的示意图

3.1.2 分支地址校验

理论上,要确保只有合法指令才能被执行,就需要对每一条将要被取指的指令地址进行校验. 然而,如果程序的第一条指令指向的是合法地址(即程序没有在一开始就被破坏),那么实际上不需对所有指令地址进行校验,只需校验控制转移指令的目标地址即可. 也就是说,正常指令的取指执行操作不需要被校验过程打断,直至遇到分支转移指令. 所以,非预期指令校验仅需在分支指令的分支目标地址被处理器流水线计算出来后进行. 此外,如果系统采用了 W⊕X 机制的话,那么理论上只需校验间接分支指令即可. 这是因为直接分支指令的地址是写在指令操作数中的,当使用 W⊕X 机制时,恶意代码是无法修改指令操作数的.

图 3 给出了简要的校验流程. 首先,待校验的分支目标地址会与已校验地址缓存中的地址进行比较,如果命中则表明该地址合法. 已校验地址缓存中存放了最近校验成功的分支目标地址. 当没有命中已校验地址缓存时,则需要与存放在内存中的校验数据进行比较. 访问内存中的数据需要首先确定待校验地址属于哪个代码段,这一过程借助代码段查找表完成. 代码段查找表中存放了校验数据在内存中的对应位置,获取校验数据位置之后,便可以按图 2 中的方法来获取待校验地址所对应的校验数据. 下面将分别阐述已校验地址缓存和代码段查找表的设计,以及如何在处理器流水线中实现该校验流程.

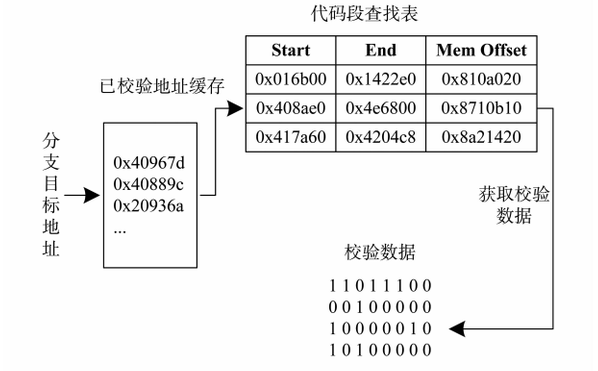

图3　校验流程示意图

已校验地址缓存的结构如图 4 所示,其功能是缓存最近校验通过的分支目标地址,从而减少因校验地址引起的处理器停顿. 该已校验地址缓存的结构与组相连 cache 相似,其存储空间被划分为 N 行和 M 组. 待校验分支目标地址通过一个 hash 函数决定属于哪个组. 每个组包含 N 个校验过的地址项或无效项. 待校验的地址与一个组中的所有地址进行并行比较,以快速决

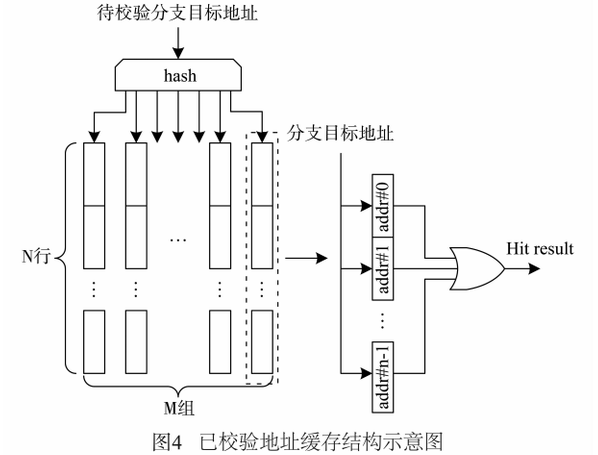

图4　已校验地址缓存结构示意图

定该地址是否命中缓存. 该缓存实现时可采用伪 LRU 算法作为组内校验地址项的替换策略. 显而易见,已校验地址缓存的硬件开销主要依赖于其所需存储空间,本文将在实验部分讨论该问题.

代码段查找表里存放代码段的起始地址和相应校验数据在内存中的存放位置. 代码段查找表中的项与应用程序实际代码段没有必要一一对应. 可以把连续的代码段合并为代码段查找表中的一项,以减少代码段查找表的项数.

如图 5 所示,完整的代码段查找表存放在内存中,并由操作系统进行维护. 像 TLB 一样,在处理器中也有相应的代码段查找表缓存,该缓存存放了最近使用的代码段查找表项. 待校验地址首先访问该缓存,如果命中则从内存相应位置访问校验数据,否则引发处理器中断,操作系统从完整的代码段查找表中寻找相应项,并加载至缓存.

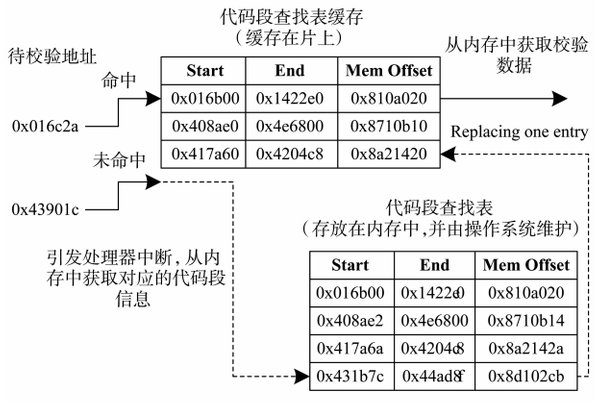

图5 代码段查找表的工作原理

接下来阐述如何将校验过程与乱序发射处理器流水线集成,图 6 给出了集成方案的示意图. 前面提到过,没有必要对每一条取出的指令地址进行校验,而只需校验分支地址即可. 所以可把校验过程从处理器流水线的取指阶段转移到提交阶段. 这主要基于两点考虑:(1)一个分支指令的分支目标只有在经过执行阶段后才能确定;(2)对于乱序发射流水线来说,只有进入了提交阶段的分支指令才是真正要执行的分支指令. 然而,如果在一条分支指令正要提交时对其分支地址进行校验,那么即使能够命中已校验地址缓存,也需要流水线停顿一到两个时钟周期来确定校验结果. 一种较好的处理方法是在分支指令位于 ROB 队列前面但还未提交时进行校验,并用一位来表示是否校验成功.

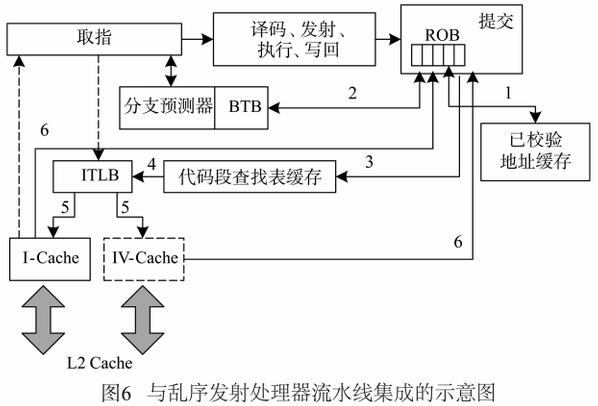

图6 与乱序发射处理器流水线集成的示意图

图 6 同时也给出了在处理器流水线中的具体校验流程. 首先,当分支指令接近 ROB 顶端时,在已校验地址缓存中检查是否存在该分支目标地址. 如果未命中,便查询分支预测器中的分支目标缓存(BTB). 如果 BTB 也没有命中,则需要从内存调取校验数据. 这一过程首先要通过代码段查找表缓存查询校验数据在内存中的位置,然后通过 ITLB 转换成物理地址,从而去内存中获取数据. 接下来的流程有两种选择方案,一种是单独设置一个指令校验 cache(即图中的 IV-Cache)存放校验数据;另一种方案是利用指令 cache 存放校验数据. 利用指令 cache 显然会挤占指令的缓存空间,但实验部分的测试数据表明,利用指令 cache 的方案并不会对应用程序性能造成太大影响.

3.2 控制流校验

本小节介绍如何通过进一步措施来防护 CRA. 前面介绍过,目前 CRA 主要有 ROP 攻击和 JOP 攻击两类. 采用低开销硬件机制如安全调用栈等技术,可以有效地阻止 ROP 攻击. 因此,这里主要讨论如何阻止 JOP 攻击.

当系统使用 W⊕X 机制时,攻击者无法利用直接控制转移指令进行恶意攻击,因为跳转地址直接写在操作数中,攻击者无法进行修改. 实际上,JOP 攻击可利用的也只能是间接跳转指令. 值得指出的是,一个应用程序在实现时,如果严格遵循结构化编程的原则,而不刻意使用 goto 语句或以嵌入汇编方式使用间接跳转指令,那么程序控制流转移的地址实际上在高级语言编译时均是可知的. 因此,可以采用 Bletsch 等人在文献[16]中提出的一种叫控制流锁定(Control-Flow Locking)的技术来校验程序控制流是否在进行正常转移.

控制流锁定的思想类似于多线程同步机制中互斥变量加解锁的概念. Bletsch 等人所提出的控制流锁定技术采用软件手段实现,每条间接跳转指令之前都会加入一个“加锁”代码片段,在跳转的目的地址处加入一个“解锁”代码片段. 显然,如果程序运行过程中每次间接跳转均需要运行“加解锁”代码的话,其开销是非常大的. 本文在这里提出一种硬件实现的方式,不需要加入“加解锁”代码片段. 而是在每个跳转目标指令前

加一个前缀以标明该指令为特殊的“解锁”指令. 对于每条成功提交的间接跳转指令,处理器自动进入一个“跳转加锁”的状态. 若提交的下一条指令是“解锁”指令,则处理器解除“跳转锁定”状态,否则触发处理器异常.

图7 给出了一个示例. 地址 0x451 处间接跳转指令的目的地址是 0x4f0. 处理器完成 jmp rax 指令的提交后会进入“跳转锁定”状态. 地址 0x4f0 处原本应该是一条 ret 指令,但由于它是一个跳转的目标地址,所以在编译器编译源代码时,会将该指令修改为一条带“解锁”前缀的 ret 指令(即 unlock ret). 带有“解锁”前缀的指令除完成该指令自身的功能外,还可解除处理器的“跳转锁定”状态. 如果处理器没有处于“跳转锁定”时执行了有“解锁”前缀的指令,则会忽略“解锁”前缀,仅执行该指令的正常功能. 硬件上实现上述控制流锁定机制十分简单,一是增加一个处理器状态来标识“跳转锁定”状态,二是在指令译码部分增加对“解锁”前缀的译码. 而在软件上,则需编译器在对高级语言进行编译时,将所有跳转目标处的指令修改为带有“解锁”前缀的指令,从而实现程序运行时处理器“跳转锁定”状态的正确解锁.

通过实现非预期指令和控制流的双重校验,可以极大降低 x86 平台代码重用攻击的概率.

```
<ssl3_write>:
   ...
   444:      mov   esi,0x17
   449:      mov   rax,QWORD PTR [rax+0x70]
   44d:      add   rsp,0x18
加锁 451:    jmp   rax
   ...
   4e2:      mov   rbp,QWORD PTR [rsp+0x8]
   4e7:      mov   r12,QWORD PTR [rsp+0x10]
   4ec:      add   rsp,0x18
解锁 4f0:    unlock ret
```

图7　控制流锁定机制示例

4　实验评估

对于本文所提出的方法,我们在 gem5 模拟器上实现并进行性能模拟. 本文实验在一台高性能台式机上完成,处理器为 3.4GHz Core i7-4770,内存 16GB,操作系统为 CentOS 6.4 x86-64. 本文从 SPECCPU 2006 测试集中选取了 21 个程序进行测试评估. 对于每一个应用,测试运行 100 亿条指令或直至其运行完毕.

首先,对已校验地址缓存的大小进行评估. 设计已校验地址缓存的目的是提供快速校验地址查询以减少处理器停顿. 为了能够进行低延迟并行比较,组内的项数不能设置太多. 在本文工作中,每组设置 4 项,并用伪 LRU 算法作为替换策略. 而对于 hash 函数,本文工作使用地址的低位作为组索引,既易于实现同时也有很高的命中率. 有了这些假定后,现在所要关注的是需要设置多少组才能提供足够高的命中率.

图 8 给出了校验所有控制转移指令时,组数目与缓存命中率间关系,图 9 则给出了仅校验间接跳转指令时,组数目与缓存命中率间关系. 从两个图的数据可以看出,这些应用程序可以大致分为两类:计算密集型和控制密集型. 计算密集型应用在组数目非常小时依然可以获得较高的命中率. 如果把标准设定为 64 组命中率在 90% 以上,那么对于图 8 来说,计算密集型应用包括 bzip2、mcf、milc、zeusmp、cactusADM、leslie3d、namd、soplex、calculix、hmmer、GemsFDTD、libquantum、h264ref 和 lbm. 由于图 9 中的实验仅校验间接跳转指令,所以地址冲突要小于图 8,并可以看到,对于同样的组数和应用,图 9 的命中率要高于图 8. 此外,从两个图的数据可以得出的一个重要结论是,当组数高于 128 时,命中率变化趋于平稳. 在图 8 中,平均命中率在组数为 128、256 和 512 时,分别是 94.68%、97.35% 和 98.84%;在图 9 中,平均命中率在组数为 128、256 和 512 时,分别是 99.11%、99.61% 和 99.70%. 基于这样的观察,本文选择 128 作为下面实验中已校验地址缓存的组数,那么缓存的总项数为 $128 \times 4 = 512$.

接下来使用三种不同的配置进行进一步的性能评估. 第一种配置是校验所有控制转移指令并使用系统自带分支预测器作为后备已校验地址缓存(记作 BP&AC);第二种配置是仅校验间接跳转指令并使用系统自带分支预测器作为后备已校验地址缓存(记作 BP&IC);第三种是仅校验间接跳转指令但不使用系统

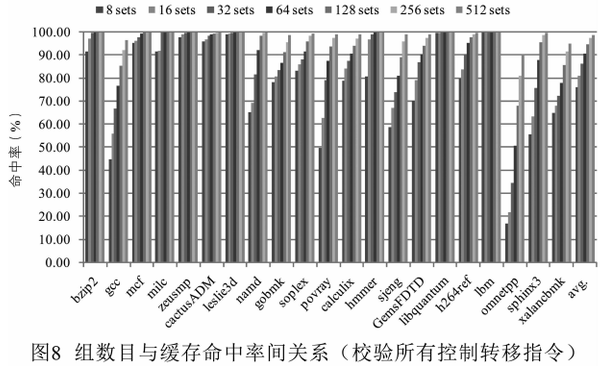

图8　组数目与缓存命中率间关系（校验所有控制转移指令）

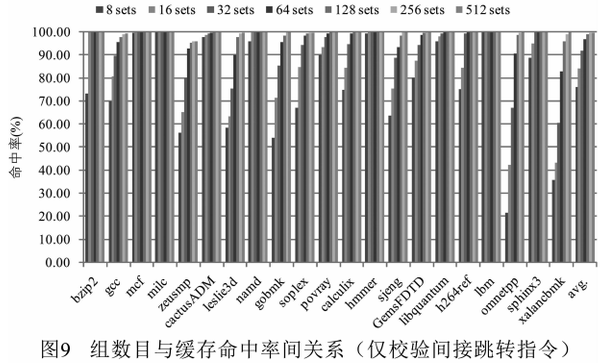

图9　组数目与缓存命中率间关系（仅校验间接跳转指令）

自带分支预测器作为后备已校验地址缓存(记作 IC).图 10 展示了基准配置与上述三种配置下的指令吞吐率(IPC)对比.这里将 x86 指令分解后的微操作记为一条指令.为了能更清楚地进行对比,图 11 ~ 图 13 展示了上述三种配置相对基准配置的性能损失.

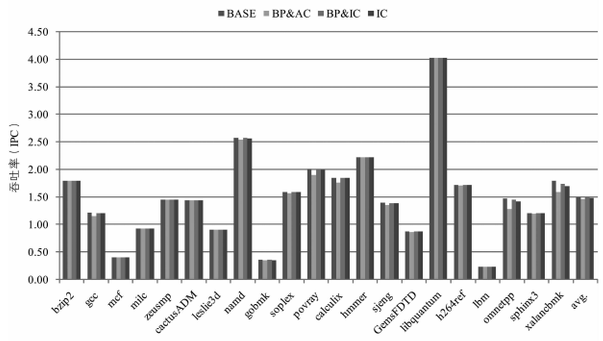

图10 三种不同配置与基准配置吞吐率对比

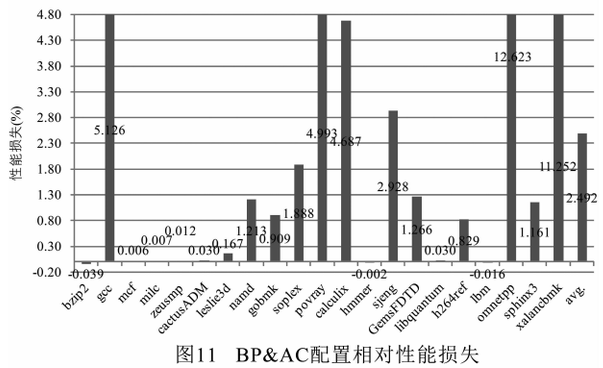

图11 BP&AC配置相对性能损失

从图 10 可以看到,无论对于哪一种配置,本文所提出校验机制对于应用程序的性能影响均是比较小的.对于 BP&AC 配置来说,平均性能损失为 2.492%,其中性能损失最高为 12.623%(omnetpp),最低为-0.039%(bzip2,负值意味着有一定性能提升),且有 11 个应用性能损失低于 1%.正如可以预见的那样,BP&IC 配置性能损失最低,在-0.093% 至 2.993% 之间,平均为 0.284%.其中仅有两个应用性能损失超过 1%,18 个应用性能损失低于 0.3%.在没有自带分支预测器的支持下,IC 配置性能损失略高,但优于 BP&AC 配置,其性能损失在 0.043% 至 5.247% 之间,平均为 0.616%.

如果将图 10 ~ 图 13 的性能测试数据与图 8 和图 9 的已校验地址缓存命中率数据进行对比可以发现,命中率通常与性能损失成反比.举例来说,omnetpp 在所有应用中命中率是最低的,其性能损失在所有三种配置中也是最高的.观察性能数据可以发现一个有意思的现象是,有些应用实际上性能还有略微提升.经过进一步分析发现,首先,这些应用的已校验地址缓存命中率都比较高;其次,2 级 cache 命中率比基准配置均有提升.这意味着处理器在从内存中取校验数据时,对 2 级 cache 行为造成

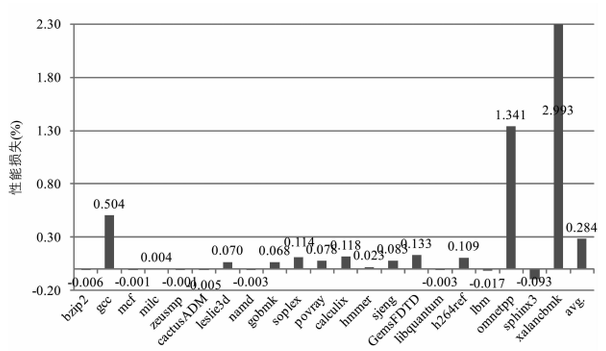

图12 BP&IC配置相对性能损失

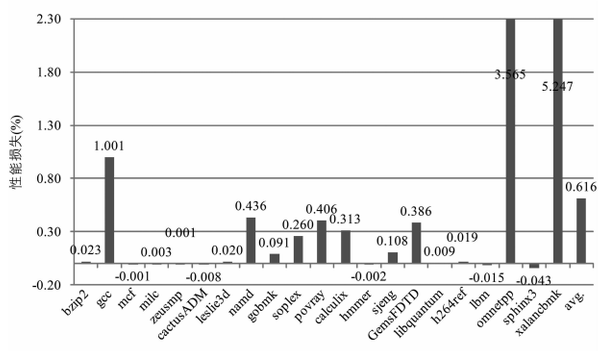

图13 IC配置相对性能损失

了影响,从而使得这些应用有一定的性能提升.由于 W⊕X 保护机制已广泛应用于当前的计算机系统,所以仅校验间接跳转指令是足够安全的.图 12 的测试数据表明,仅校验间接跳转指令对应用程序的性能影响是非常小的.即使校验所有控制指令,与 BP&IC 配置相比,IC 配置也仅带来 0.332% 的平均性能损失.总体而言,BP&IC 配置的性能损失在-0.093% ~2.993% 之间.

表 1 建立校验数据所需开销

应用程序	时间(秒)	应用程序	时间(秒)
astar	0.004882	bwaves	0.014530
bzip2	0.009874	cactusADM	0.064493
calculix	0.163895	dealII	0.308739
gamess	0.757682	gcc	0.266470
GemsFDTD	0.035370	gobmk	0.078845
gromacs	0.091436	h264ref	0.049466
hmmer	0.031631	lbm	0.005658
leslie3d	0.017685	libquantum	0.004892
mcf	0.005363	milc	0.015106
namd	0.030756	omnetpp	0.051266
perlbench	0.098552	povray	0.088732
sjeng	0.015126	soplex	0.037534
sphinx	0.022001	tonto	0.390543
Xalan	0.318079	zeusmp	0.034434

需要指出的是,除了应用程序正常运行时校验过程所带来的开销外,还有在程序运行前对程序反汇编建立校验数据所带来的开销. 表 1 给出了应用程序反汇编并建立校验数据所需开销. 实验数据在前述 3.4GHz Core i7-4770 平台上获取的. 在所有 28 个应用中,仅有 6 个应用耗时超过了 0.1 秒. 由于构建校验数据只在程序运行前进行,所以可以认作是程序的启动开销,并不影响程序运行时的性能.

5　结束语

本文主要提出了一套解决 x86 平台代码重用攻击的硬件实现方法. 该方法主要由两个校验机制组成:一是对非预期指令的校验,二是对程序控制流的校验. 相对于先前的研究工作,本文的贡献主要在对非预期指令的校验上. 本文通过实验数据对所提出的方法进行了评估,实验结果表明,对于 BP&IC 配置实现来说,应用程序的性能损失在-0.093% ~ 2.993%,性能影响很小.

参考文献

[1] Shacham H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)[A]. ACM Conference on Computer and Communications Security[C]. New York:ACM Press,2007. 552 - 561.

[2] Davi L. Dynamic integrity measurement and attestation:towards defense against return-oriented programming attacks [A]. ACM Workshop on Scalable Trusted Computing [C]. New York:ACM Press,2009. 49 - 54.

[3] Chen P. Drop:Detecting return-oriented programming malicious code[A]. 5th International Conference on Information Systems Security [C]. Berlin, Heidelberg: Springer-Verlag,2009. 163 - 177.

[4] Bletsch T. Jump-oriented programming: a new class of code-reuse attack[A]. ACM Symposium on InformAtion, Computer and Communications Security[C]. New York: ACM Press,2011. 30 - 40.

[5] Checkoway S. Return-oriented programming without returns [A]. ACM Conference on Computer and Communications Security[C]. New York:ACM Press,2010. 559 - 572.

[6] Davi L. ROPdefender:A Detection Tool to Defend Against Return-Oriented Programming Attacks[A]. ACM Symposium on InformAtion,Computer and Communications Security[C]. New York:ACM Press,2011. 40 - 51.

[7] Francillon A. Defending embedded systems against control flow attacks[A]. Proceedings of the first ACM workshop on Secure execution of untrusted code[C]. New York: ACM Press,2011. 19 - 26.

[8] Chen P. Efficient Detection of the Return-Oriented Programming Malicious Code[A]. Information Systems Security:6th International Conference,ICISS 2010[C]. Berlin, Heidelberg:Springer-Verlag,2010. 140 - 155.

[9] Kayaalp M. Branch regulation: Low-overhead protection from code reuse attacks[A]. 39th Annual International Symposium on Computer Architecture[C]. New York: ACM Press,2012. 94 - 105.

[10] Hiser J. ILR:Where'd My Gadgets Go? [A]. 2012 IEEE Symposium on Security and Privacy[C]. San Francisco, CA:IEEE,2012. 571 - 585.

[11] Pappas V. Smashing the Gadgets:Hindering Return-Oriented Programming Using In-place Code Randomization [A]. 2012 IEEE Symposium on Security and Privacy [C]. San Francisco,CA:IEEE,2012. 601 - 615.

[12] Huang Z. A dynamic detection method against ROP and JOP[A]. 2012 International Conference on Systems and Informatics (ICSAI)[C]. Yantai, China: IEEE, 2012. 1072 - 1077.

[13] Jacobson E. Detecting code reuse attacks with a model of conformant program execution[A]. In Proceedings of the 6th International Symposium on Engineering Secure Software and Systems (ESSoS)[C]. Berlin, Heidelberg: Springer-Verlag,2014. 1 - 18.

[14] Kayaalp M. SCRAP:Architecture for signature-based protection from code reuse attacks[A]. 19th International Symposium on High Performance Computer Architecture (HPCA2013)[C]. Shenzhen, China: IEEE, 2013. 258 - 269.

[15] Yee B. Native Client:A sandbox for portable, untrusted x86 native code[A]. Proceedings of the 2009 30th IEEE Symposium on Security and Privacy[C]. New York: ACM Press,2009. 79 - 93.

[16] Bletsch T. Mitigating code-reuse attacks with control-flow locking[A]. 27th Annual Computer Security Applications Conference (ACSAC)[C]. New York: ACM Press, 2011. 353 - 362.

作者简介

吕雅帅　男,1981 年生于河北邯郸. 2009 年在国防科学技术大学取得计算机科学与技术博士学位. 目前为中国人民解放军装备学院国防科技重点实验室助理研究员. 研究方向为处理器体系结构.

E-mail:freelancer_lys@163.com