













况下,最后一个间接 call 指令将最先被验证,然后再验证前一个间接 call 指令,以此类推,直至验证到第 1 个间接 call 指令.故最终可以验证 Function A 中间接 call 指令的合法性),尽管这种情况可能没有直接通过 Function A 中的 Invalid call 报告代码复用攻击,但是按照该思路继续分析,最终也可以发现 Function A 的 Invalid call.另外,若 LKM A 的 Function D 执行 call \*%rax 时指向 Function B,尽管其指向 Function B 的入口位置,但是由于 Function B 并未在模块 A 的导入表中,故报告 call 指令的非法使用.

因此,call 插桩需要在每个函数的入口位置插入 vmcall(0,a)标签指令以验证 call 指令进入位置的正确性,其中 a 为标签编号.

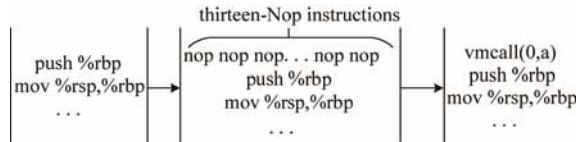


Fig.5 CFI table instruction instrumentation process based on GCC

图 5 基于 GCC 的 CFI 标签指令插桩过程

根据 Intel 开发手册<sup>[37]</sup>,vmcall 指令的操作码为 0F 01 C1,占用 3 个字节,第 1 个输入参数 0 通过寄存器传递,占用 5 个字节,第 2 个输入参数标签 a 也通过寄存器传递,占用 5 个字节,共需要 13 个字节.那么在函数入口插入标签指令 vmcall(0,a)的具体过程如图 5 所示.

根据图 5 可知,标签指令插入过程包括两个阶段,第 1 个阶段在函数入口处插入 13 个 nop 指令,第 2 个阶段重写 13 个 nop 指令为标签指令对

应的二进制码,比如 vmcall(0,4)对应的二进制码为 b8 00 00 00 00 ba 04 00 00 00 0f 01 c1.

第 1 阶段中 nop 指令的插入采用基于重编译 recompiling 的方法.内核编译采用的是 gcc 编译器,本文通过扩展 gcc 实现标签指令的插桩.gcc 编译器内部结构如图 6 所示,整个编译过程包含 3 个重要的转换过程:待编译的源代码→gcc 抽象语法树(abstract syntax tree,简称 AST)→RTL(register transfer language,寄存器转移语言)表达式→汇编代码.为了实现在函数入口处插入 nop 指令,我们选择在第 2 个转换中扩展 gcc 的功能.首先生成 13 个 nop 指令的 RTL 表示,然后将其插入到 RTL 链中.那么,当 RTL 链转换为汇编代码时,就会将 nop 指令插入到内核映像中.

第 2 阶段是重写插入的 nop 指令为标签指令.首先寻找需要重写的 13 个 nop 指令的地址.由于内核中可能存在连续的 nop 指令,所以不能够直接通过查找连续的 13 个 nop 指令来确定需要重写的 nop 指令地址.这里,我们通过反汇编重编译的内核,根据汇编代码获取每个函数的头部位置(节偏移+函数地址偏移)即为需要重写的地址.然后将标签指令对应的二进制代码依次替换 nop 指令即可.为方便标签指令中标签的编号,我们采用顺序编号,即从头扫描内核映像,每扫到 1 个函数编号 1 次,每编号 1 次编号加 1.

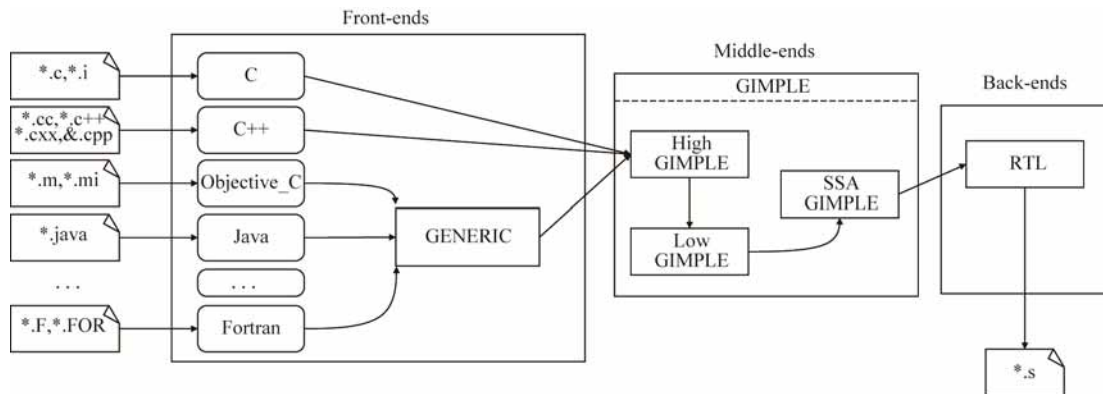


Fig.6 Architecture for GCC

图 6 GCC 编译器结构

### 3.2 ret 指令目标合法性验证和插桩

根据 ret 指令的 CFI 约束,ret 指令执行后控制流转回 caller.ret 指令使用合法性验证使用 vmcall(1,label)标签指令,该指令位于 call 指令之后,如图 7 所示.

根据图 7,Function A 在入口处激活标签 a,之后调用 Function B,状态机从 S<sub>0</sub> 进入 S<sub>1</sub>,执行 vmcall(0,b)指令,激活标签 b.在 ret 指令执行之前,执行指令 vmcall(2,b)取消标签 b 的激活状态.由于 ret 指令可能被攻击者利用,在加入标签指令后,若 ret 指令返回 callq Function B 的下一条指令 vmcall(1,a),则该指令陷入 Hypevisor 后检查标签 a 的激活状态,若其处于激活状态,并且标签存储空间中 a 以下未存在处于激活状态的标签,那么系统运行正常;若 ret 指令返回到其他位置的指令,如 vmcall(1,c),则该指令检查标签 c 的激活状态,由于 c 未被激活,说明没有与之对应的 call 指令,产生异常报告,进入状态 S<sub>4</sub>.此外,如果 retq 指令返回的不是某一个 callq 指令的下一条指令,即没有指向 vmcall(1,x)指令,那么为了检测此类非法 retq 指令情况,我们增加了 vmcall(2,x)指令的功能,即定时器功能.该指令在取消标签属性的同时,启动定时器(定时时间为 retq 指令和 vmcall(1,x)指令所需要的 CPU 时钟周期数).那么在定时器时间耗尽后将运行判断是否执行 vmcall(1,x)指令的功能函数.若在给定时间戳下 vmcall(1,x)指令未得到执行,那么判断 retq 指令执行为非法的,产生异常报告;若 vmcall(1,x)得到执行,那么根据 vmcall(1,x)的功能判断 ret 指令执行的合法性.

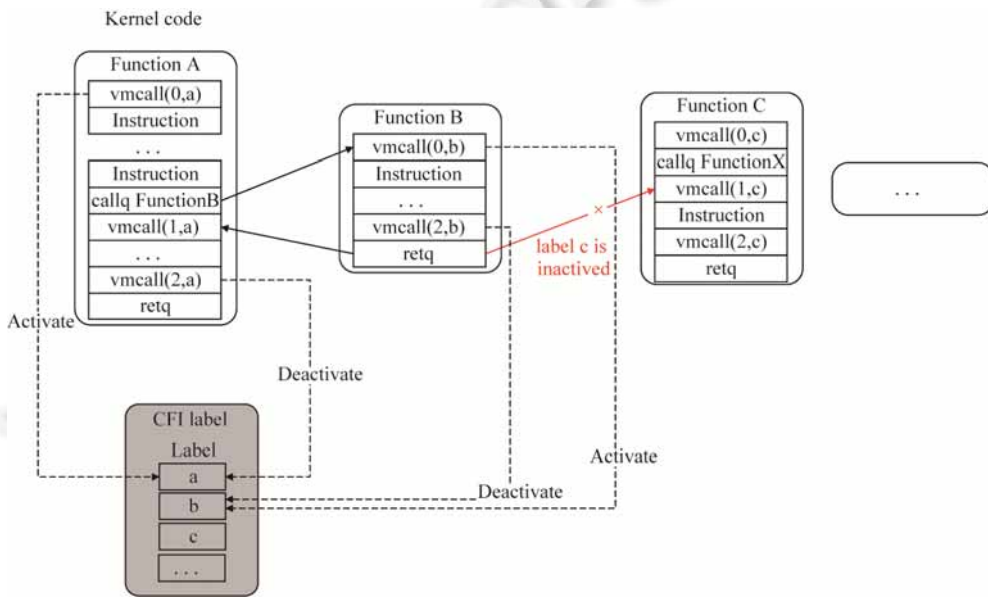


Fig.7 CFI policy for valid function return instruction verification  
图 7 基于 CFI 策略的 ret 指令使用合法性验证

因此,ret 插桩需要在 call 指令后插入 vmcall(1,a)标签指令验证 ret 返回位置的正确性,并在 ret 指令之前插入 vmcall(2,b)指令实现标签状态的处理和定时器功能,其中,a,b 为标签编号.

ret 指令插桩与 call 指令插桩的不同之处在于需要在两个不同的位置处插入 nop 指令,分别在 call 指令的后面和 ret 指令的前面插入 nop 指令.我们选择在 gcc 编译转换过程的第 3 个阶段实现 nop 指令的插入.针对每个 CPU 平台,gcc 有对应的 Machine Description 用于指导指令生成.ret 指令插桩是在 call 指令之后和 ret 指令之前插入 nop 指令,故只需修改 call 指令和 ret 指令的 Machine Description 即可实现 nop 指令的插入.对于第 1 个位置处的插桩,当发现 call 指令时,在生成 call 指令汇编代码的同时,在该汇编代码的后面添加 13 个 nop 指令后再进行下一条指令的汇编代码生成;对于第 2 个位置处的插桩,当发现 ret 指令时,在生成 ret 指令汇编代码时,在该汇编代码前添加 13 个 nop 指令.ret 指令插桩第 2 阶段的处理过程与 call 指令类似,但需要确保同一个函数标签



编号的一致性.

### 3.3 indirect jump 目标合法性验证和插桩

根据 indirect jump 的 CFI 约束, indirect jump 只能跳转到本函数内部或者另外一个函数的入口. 由于 indirect jump 的目标地址在其运行时才确定, 所以为了确定跳转目标的合法性, 我们在 indirect jump 指令之前使用 vmcall(3, address) 指令. 该指令包括两个方面的功能, 一方面验证跳转目标是否处于本函数内, 另一方面验证跳转目标对应的指令是否为 vmcall(0, label) 指令. 若满足两方面中的一个, 则表明 indirect jump 使用合法, 否则, 报告异常, 如图 8 所示.

从图 8 可以看出, vmcall(3, %r10) 指令触发 indirect jump 使用合法性检查, 状态从  $S_0$  进入  $S_3$ , 在  $S_3$ , 该指令判断 indirect jump 的目标地址是否处于自身所在函数内部或者是否是函数入口处. 图 8 中显示, 如果其跳转到 Function B 的入口处则是合法的, 状态从  $S_3$  回到  $S_0$ ; 若是跳转到 Function B 的内部, 则是非法的, 进入状态  $S_4$ , 报告异常状态.

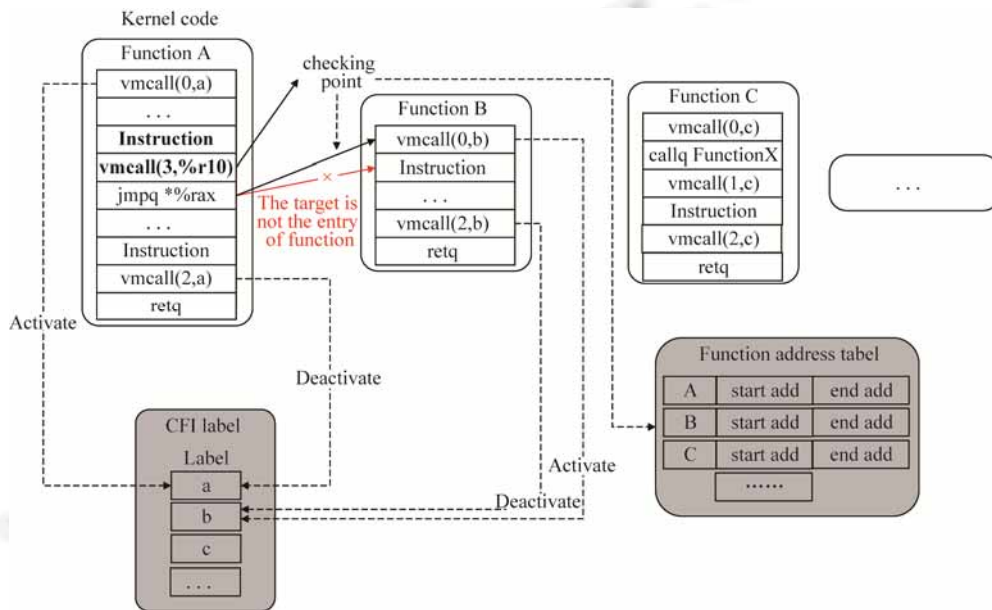


Fig.8 CFI policy for valid indirect jump instruction verification

图 8 基于 CFI 策略的 indirect jump 指令使用合法性验证

由于 vmcall 指令传递参数到 Hypervisor 使用的是寄存器, 并且 indirect jump 指令的操作数也是使用寄存器确定转移目标地址, 所以为避免破坏跳转目标所使用的寄存器, 通过反汇编内核镜像, 分析 indirect jump 指令所使用的寄存器, 最终确定使用 r10 寄存器传递 indirect jump 指令的跳转目标到 Hypervisor 中. 首先将 indirect jump 指令的跳转目标存入 r10 寄存器, 然后将该地址传入 Hypervisor 中进行分析 and 判断. 分析寄存器的同时分析内核各个函数的入口地址和结束地址构建函数地址表, 用于 jump 目标地址合法性判断.

由于不同寻址方式需要添加的指令数不一样, 为了方便统一处理, 我们选择了最复杂的 indirect jump 指令, 即操作数构成最复杂的指令. 例如, jmp \*-0x12345678(,%rax,4), 该指令的跳转目标为  $[4*\%rax-0x12345678]$ . 计算该跳转目标需要 2 条指令完成, mov %rax,%rsi; mov -0x12345678(,%rsi,4),%r10. 第 1 条指令占用 3 个字节, 第 2 条指令占用 4 个字节. vmcall(3,%r10) 占用 13 个字节, 故总共需要 20 个字节. 对于不满 20 字节的情况, 如 jmp \*%rax, 计算目标地址只需 1 条指令, mov %rax,%r10, 加上 vmcall 指令字节数共需 16 字节, 剩余 4 个字节保留为 nop 即可, 而且 nop 指令的填充选择后向填充. indirect jump 指令的插桩过程与 ret 指令类似, 这里不再赘述.

在对 3 类指令进行插桩时存在一些异常情况需要进行特殊处理,主要是以汇编形式或内联汇编存在的内核代码.它们不能采用 gcc 扩展方式实现插桩,需要手工修改这部分代码实现间接转移指令的插桩.譬如 entry\_64.S 中的函数 ftrace\_caller,首先我们在 ENTRY(ftrace\_caller)后增加指令 vmcall(0,x),然后在 END(ftrace\_caller)前的 retq 指令之前增加指令 vmcall(2,x),由于该函数内部未包含间接 call 指令和间接 jump 指令,故不需要进行其他处理,其他类似的函数进行类似的修改.那么编译后的内核映像在执行到该函数时就会通过 vmcall 指令陷入 Hypervisor 中进行相应的验证.

#### 4 实验及结果分析

本节从有效性和性能两个方面对 CFI-KCraD 进行测试.有效性测试用于验证 CFI-KCraD 能否实现内核代码复用攻击检测,性能测试用于验证 CFI-KCraD 的效率和开销.实验环境如下:主机 CPU 为 Intel(R) Core(TM) i5-750 @ 2.67GHz,内存大小为 4GB.Hypervisor 采用 Bitvisor<sup>[38]</sup>,其版本为 1.4.0,Guest OS 采用的是 3.2.43-x86\_64 内核的 Ubuntu 12.04.

##### 4.1 有效性测试

为了测试 CFI-KCraD 的有效性,我们选择了文献[6,7,10,12,27,35]实现的 ROP rootkits 作为测试用例,并对其进行修改,使其能够运行于 3.2.43 版本的内核.

文献[27]实现了一种 call-preceded 的 ROP 攻击,可绕过检测 ret 结尾的 gadgets 特征的检测方法.文献[35]实现的是以 ret 指令结尾的 6 个 gadgets 构成的 rootkit,该 rootkit 不实现恶意功能,只是在内存中增加一个字的内容.文献[12]实现的是以 jump 指令结尾的 101 个 gadgets 构成的 rootkit,它实现了进程隐藏.文献[6]面向的是 Windows 系统,实现了以 ret 为结尾的 gadgets 构成的 rootkit,它也实现了进程隐藏.我们对其进行改造使其运行在 Linux 系统.文献[7]提出了一种增强型的代码复用攻击 BIOP,使用 jmp 指令、call 指令结尾的短指令序列构造攻击.文献[10]实现的 rootkit 原理与文献[12]类似,只是实现了不同类型的攻击,它利用 jump 指令结尾的 34 个 gadgets 构成实现系统调用表篡改的 rootkit.测试结果见表 1.表 1 的第 1 列为测试用例名称,第 2 列是测试用例的结构组成,第 3 列表示 CFI-KCraD 能够检测第 1 列所列的测试用例,第 4 列表示测试用例违背何种 CFI 约束规则而被检测.

**Table 1** The detection results for ROP rootkits  
表 1 ROP rootkits 检测结果

ROP rootkits	构成	检测	违背
文献[27]	23 个 call-preceded 构成的 gadgets	√	call
文献[35]	6 个 ret 结尾的 gadgets	√	ret
文献[12]	101 个 jump 结尾的 gadgets	√	indirect jump
文献[6]	85 个 ret 结尾的 gadgets	√	ret
文献[7]	10 个 gadgets	√	call, indirect jump
文献[10]	34 个 jump 结尾的 gadgets	√	indirect jump

从表 1 可知,CFI-KCraD 能够有效检测这 6 类攻击,这表明本文提出的方法是有效的.

为了进一步验证本文方法的有效性,与目前能够用于内核代码复用攻击检测的工具 HDROP<sup>[30]</sup>和文献[20]提出的方法进行对比测试,测试结果见表 2.

根据表 2,CFI-KCraD 能够检测出所有的代码复用攻击,而 HDROP 和文献[20]的方法无法检测文献[7]、文献[12]和文献[10]提出的攻击方法.这是因为,HDROP 通过分支预测异常检测以 ret 结尾的 ROP 攻击,文献[20]则是构建隐秘地址空间存储返回地址,通

**Table 2** Comparison of different detection methods  
表 2 检测方法对比

ROP rootkits	CFI-KCraD	HDROP	文献[20]的方法
文献[27]	√	×	×
文献[35]	√	√	√
文献[12]	√	×	×
文献[6]	√	√	√
文献[7]	√	×	×
文献[10]	√	×	×

过对比 ret 返回地址与存储的返回地址判断是否有 ROP 攻击,它们都是针对 ret 指令进行监控,所以无法检测文献[12]、文献[10]实现的 JOP 攻击和文献[7]实现的 BIOP 攻击.HDROP 和文献[20]的检测方法也无法检测文献[27]实现的 ROP 攻击,因为在文献[27]的攻击方法中,call 和 ret 指令是成对出现的,并且 ret 返回的是 call 指令的下一条指令,并未违背它们采用的检测特征.而 CFI-KCraD 能够检测到此攻击是因为其在执行 call 指令时并未从函数入口处开始执行,违背了 call 指令的使用规则.

为了检验本文方法的误报率,我们使用了一些常用的应用程序进行对比检测.通过运行表 3 所列举的应用程序 50 次,观察每次运行过程中是否会产生误报,记录产生误报的次数,结果见表 3.

由表 3 可知,没有误报产生,这也进一步验证了本文方法的有效性.

Table 3 Results of false positive test

表 3 误报测试结果

Benchmark	版本	命令	误报次数
decompress	1.0.6	tar -zxf inux-source.tar.bz2	0
compression	1.4	tar -zcf linux-source-dir	0
file copy	8.13	cp -r linux-3.2.43 any	0
kernel build	3.2.43	make	0
unixbench	5.1.2	./Run	0
linux boot	Ubuntu 12.04	linux boot	0

## 4.2 性能测试

### 4.2.1 微基准测试

我们采用 lmbench 作为本次实验的微基准测试集.由于 CFI-KCraD 的标签指令分布在内核各个子系统中,

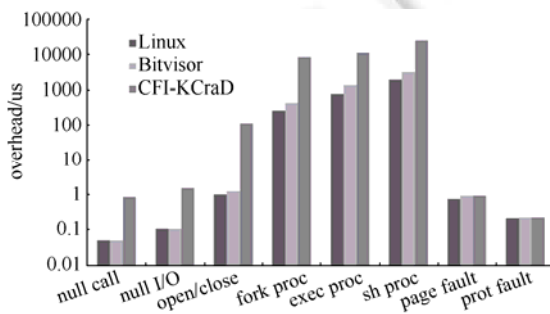


Fig.9 Microbenchmark results of CFI-KCraD

图 9 CFI-KCraD 微基准测试结果

如文件系统、进程管理等,为了充分测试 CFI-KCraD 的性能开销,我们从 lmbench 选择了与系统调用相关、I/O 读写相关等测试项作为基准测试指标.通过分别测试 CFI-KCraD、Bitvisor、裸机三者下的性能开销,并通过对比来分析 CFI-KCraD 所引入的性能开销,测试结果如图 9 所示.

其中,由于不同程序测试的性能开销数据差距较大,故我们对测试数据(纵坐标)进行对数化(基数为 10)处理.根据图 9 的数据可知,Null Call 操作测试取进程号所需时间,Null I/O 操作测试空读写的平均时间,Bitvisor 未对它们进行监控,直接由处理器处理,未带来任何开销,而 CFI-KCraD 在它们执行时因进行系统调用而陷入 Hypervisor 一次,故带来的开销较小;而 open/close 操作是先打开文档后再关闭文档,这一过程中需要调用系统调用以及内存映射等操作,在这些操作发生时陷入 Hypervisor 中,CPU 切换次数增多,故 CFI-KCraD 较 Linux 和 Bitvisor 带来了额外开销.同时,CFI-KCraD 在 fork proc、exec proc 和 sh proc 这 3 项测试指标下的性能远大于其他指标,并且开销比 Bitvisor 大很多.这是因为,这 3 项指标的测试过程涉及到进程管理、文件管理、内存管理等,需要陷入陷入 Hypervisor 多次.prot fault 指的是保护异常带来的开销,CFI-KCraD 和 Bitvisor 一样只是捕获该操作,捕获之后将该操作的处理返还给操作系统内核,故开销较小;page fault 指的是缺页异常带来的开销,由于 CFI-KCraD 并未对其作额外的处理,故与 Bitvisor 的开销差不多.

### 4.2.2 应用程序基准测试

为了进一步评测 CFI-KCraD 的性能,我们选择与表 3 相同的应用程序基准测试集测试 CFI-KCraD 的性能开销.测试结果如图 10 所示.

由于 kernel build(内核编译)和 Unixbench 是综合性应用,既需要 CPU 时间,也需要大量的 I/O 操作,测试过

程中大量使用系统资源,如文件系统、管道和进程等,需要频繁访问磁盘和内存,这些行为调用了内核服务,如系统调用和进程切换等,而 CFI-KCraD 在系统调用处理函数、进程相关函数等入口出口位置因标签指令陷入 Hypervisor 中进行验证,故带来了较大的性能开销.对于 decompress(解压缩)和 compression(压缩)程序,分别引入了 11%和 8%的性能开销,它们属于计算密集型应用,而 CFI-KCraD 基于 Bitvisor 开发,使用了硬件虚拟化技术,计算指令可直接由物理 CPU 运行,故相对内核编译和 Unixbench 开销较小.file copy(文件复制)是 I/O 密集型作业,其中包含大量的读写操作,由于 CFI-KCraD 对读写系统调用函数被调用时陷入 Hypervisor,因此导致了一定的性能开销(41%).Linux boot 可全面测试 CFI-KCraD 的性能,因为系统启动过程中运行大量的应用程序、库并执行系统调用,该项测试引入的开销为 33%.综上测试,CFI-KCraD 的性能开销不超过 60%,虽然开销比 HDROP 的 19%和文献[20]的 10%要高,但 CFI-KCraD 的检出率明显高于其他内核级代码复用攻击检测工具.

由于采用了重编译方法在内核中插入了 nop 指令,使得内核的代码分布发生了变化,内核映像的大小也发生了变化,最终内核映像大小增长约 6%.

### 5 结 论

本文提出了一种基于细粒度 CFI 的内核级代码复用攻击检测方法 CFI-KCraD.该方法基于 CFI 约束规则和状态机模型给出了检测方法的基本思路;根据内核级代码复用攻击误用的间接转移指令类型分别采取不同的验证方法,在编译器辅助下实现 CFI 标签指令的插桩;同时引入 Hypervisor 实现 CFI 策略的验证和分析,一方面使用 Hypervisor 提供的指令 vmcall 作为 CFI 标签指令,不需要增添额外的监控指令,另一方面将检测程序放置于 Hypervisor 中,提高了检测方法的安全性.通过对 CFI-KCraD 的有效性和性能进行测试,测试结果表明,CFI-KCraD 能够有效地检测内核级代码复用攻击,并且引入的性能开销在一个可接受的范围内.目前,插桩过程中未考虑可加载内核模块(loadable kernel module,简称 LKM),而它们的代码也是内核级代码复用攻击利用的对象,下一步研究拟将分析对象扩展到 LKM,在此过程中可能需要解决 LKM 的动态加载位置确定问题.

致谢 在此,向对本文研究工作提供基金支持的单位和评阅本文的审稿专家表示衷心的感谢,向为本文研究工作提供基础和平台的前辈致敬.

### References:

- [1] Andersen S, Abella V. Data execution prevention, changes to functionality in Microsoft Windows XP Service Pack 2, Part 3: memory protection technologies. 2015. <https://technet.microsoft.com/en-us/library/bb457155.aspx>
- [2] Cowan C, Pu C, Maier D, Hintony H, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proc. of the 7th USENIX Security Symp. Berkeley: Usenix Association, 1998. 63–78. <https://www.usenix.org/legacy/publications/library/proceedings/sec98/>
- [3] Tran M, Etheridge M, Bletsch T, Jiang XX, Freeh V, Ning P. On the expressiveness of return-into-libc attacks. In: Proc. of the 14th Int'l Conf. on Recent Advances in Intrusion Detection. Cambridge: Springer-Verlag, 2011. 121–141. [doi: 10.1007/978-3-642-23644-0\_7]
- [4] Shacham H. The geometry of innocent flesh on the bone: Return-Into-Libc without function calls (on the x86). In: Proc. of the 14th ACM Conf. on Computer and Communications Security. New York: ACM Press, 2007. 552–561. [doi: 10.1145/1315245.1315313]
- [5] Buchanan E, Roemer R, Shacham H, Savage S. When good instructions go bad: Generalizing return-oriented programming to RISC. In: Proc. of the 15th ACM Conf. on Computer and Communications Security. New York: ACM Press, 2008. 27–38. [doi: 10.1145/1455770.1455776]

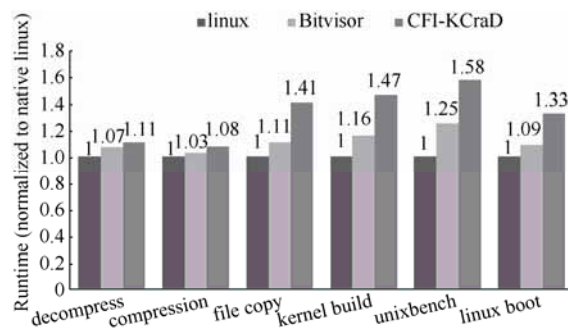


Fig.10 Application benchmark performance test results  
图 10 应用基准程序性能测试结果

- [6] Hund R, Holz T, Freiling FC. Return-Oriented Rootkits: Bypassing kernel code integrity protection mechanisms. In: Proc. of the 18th USENIX Security Symp. Berkeley: Usenix Association, 2009. 383–398. <https://www.usenix.org/legacy/publications/library/proceedings/sec98/>
- [7] Xing X, Chen P, Ding WB, Mao B, Xie L. BIOP: Automatic construction of enhanced ROP attack. Chinese Journal of Computers, 2014,37(5):1111–1123 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2014.01111]
- [8] Checkoway S, Davi L, Dmitrienko A, Sadeghi AR, Shacham H, Winandy M. Return-Oriented programming without returns. In: Proc. of the 17th ACM Conf. on Computer and Communications Security. New York: ACM Press, 2010. 559–572. [doi: 10.1145/1866307.1866370]
- [9] Bletsch T, Jiang X, Freeh VW, Liang Z. Jump-Oriented programming: A new class of code-reuse attack. In: Proc. of the 6th ACM Symp. on Information, Computer and Communications Security. New York: ACM Press, 2011. 30–40. [doi: 10.1145/1966913.1966919]
- [10] Li ZY, Mao B, Xie L. Construction method of Rootkit based on JOP. Computer Science, 2011,38(10A):44–49 (in Chinese with English abstract).
- [11] Bosman E, Bos H. Framing signals—A return to portable exploits. In: Proc. of the 35th IEEE Symp. on Security and Privacy. Washington: IEEE CS Press, 2014. 243–258. [doi: 10.1109/SP.2014.23]
- [12] Chen P. Research on the attack and defense techniques of code reuse [Ph.D. Thesis]. Nanjing University, 2012 (in Chinese with English abstract).
- [13] PaX Team. Address space layout randomization (ASLR). 2015. <http://pax.grsecurity.net/docs/aslr.txt>
- [14] Shacham H, Page M, Pfaff B, Goh EJ, Modadugu N, Boneh D. On the effectiveness of address-space randomization. In: Proc. of the 11th ACM Conf on Computer and Communications Security. New York: ACM Press, 2004. 298–307. [doi: 10.1145/1030083.1030124]
- [15] Seibert J, Okhravi H, Söderström E. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In: Proc. of the 21st ACM SIGSAC Conf. on Computer and Communications Security. New York: ACM Press, 2014. 54–65. [doi: 10.1145/2660267.2660309]
- [16] Pappas V, Polychronakis M, Keromytis AD. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In: Proc. of the 33rd IEEE Symp. on Security and Privacy. Washington: IEEE CS Press, 2012. 601–615. [doi: 10.1109/SP.2012.41]
- [17] Snow KZ, Monroe F, Davi L, Dmitrienko A, Lieben C, Sadeghi AR. Just-in-Time code reuse: On the effectiveness of fine-grained address space layout randomization. In: Proc. of the 34th IEEE Symp. on Security and Privacy. Washington: IEEE CS Press, 2013. 574–588. [doi: 10.1109/SP.2013.45]
- [18] Bletsch T, Jiang X, Freeh V. Mitigating code-reuse attacks with control-flow locking. In: Proc. of the 27th Annual Computer Security Applications Conf. New York: ACM Press, 2011. 353–362. [doi: 10.1145/2076732.2076783]
- [19] Li J, Wang Z, Jiang X, Grace M, Bahram S. Defeating return-oriented rootkits with return-less kernels. In: Proc. of the 5th European Conf. on Computer systems. New York: ACM Press, 2010. 195–208. [doi:10.1145/1755913.1755934]
- [20] Tian S, He YP, Ding BZ. Prevent kernel return-oriented programming attacks using hardware virtualization. In: Proc. of the 8th Int'l Conf. on Information Security Practice and Experience. Cambridge: Springer-Verlag, 2012. 289–300. [doi: 10.1007/978-3-642-29101-2\_20]
- [21] Onarlioglu K, Bilge L, Lanzi A, Balzarotti D, Kirda E. G-Free: Defeating return-oriented programming through gadget-less binaries. In: Proc. of the 26th Annual Computer Security Applications Conf. New York: ACM Press, 2010. 49–58. [doi: 10.1145/1920261.1920269]
- [22] Davi L, Sadeghi A R, Winandy M. ROPdefender: A detection tool to defend against return-oriented programming attacks. In: Proc. of the 6th ACM Symp. on Information, Computer and Communications Security. New York: ACM Press, 2011. 40–51. [doi: 10.1145/1966913.1966920]
- [23] Chen P, Xiao H, Shen X, Yin X, Mao B, Xie L. DROP: Detecting return-oriented programming malicious code. In: Proc. of the 5th Int'l Conf. on Information Systems Security Information Systems Security. Cambridge: Springer-Verlag, 2009. 163–177. [doi :10.1007/978-3-642-10772-6\_13]
- [24] Fratric I. Runtime prevention of return-oriented programming attacks. 2015. <https://github.com/ivanfratric/ropguard>
- [25] Pappas V, Polychronakis M, Keromytis AD. Transparent ROP exploit mitigation using indirect branch tracing. In: Proc. of the 22nd USENIX Security Symp. Berkeley: Usenix Association, 2013. 447–462.
- [26] Cheng Y, Zhou Z, Yu M, Ding X, Deng RH. ROPecker: A generic and practical approach for defending against ROP attacks. In: Proc. of the 21st Network and Distributed System Security (NDSS) Symp. San Diego: The Internet Society, 2014. 1–14. <http://www.internetsociety.org/events/ndss-symposium-2014>



- [27] Carlini N, Wagner D. Rop is still dangerous: Breaking modern defenses. In: Proc. of the 23rd USENIX Security Symp. Berkeley: Usenix, 2014. 385–399. <https://www.usenix.org/conference/usenixsecurity14/>
- [28] Göktaş E, Athanasopoulos E, Polychronakis M, Bos H, Portokalidis G. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In: Proc. of the 23rd USENIX Security Symp. Berkeley: Usenix Association, 2014. 417–432. <https://www.usenix.org/conference/usenixsecurity14/>
- [29] Davi L, Sadeghi A.R, Lehmann D, Monroe F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In: Proc. of the 23rd USENIX Security Symp. Berkeley: Usenix Association, 2014. 401–416. <https://www.usenix.org/conference/usenixsecurity14/>
- [30] Goktas E, Athanasopoulos E, Bos H, Portokalidis G. Out of control: Overcoming control-flow integrity. In: Proc. of the 35th IEEE Symp. on Security and Privacy. Washington: IEEE CS Press, 2014. 575–589. [doi: 10.1109/SP.2014.43]
- [31] Criswell J, Dautenhahn N, Adve V. KCoFI: Complete control-flow integrity for commodity operating system kernels. In: Proc. of the 35th IEEE Symp. on Security and Privacy. Washington: IEEE CS Press, 2014. 14–29. [doi: 10.1109/SP.2014.26]
- [32] Kayaalp M, Schmitt T, Nomani J, Ponomarev DV, Ghazaleh NA. Signature-Based protection from code reuse attacks. IEEE Trans. on Computers, 2015,64(2):533–546. [doi: 10.1109/TC.2013.230]
- [33] Davi L, Koeberl P, Sadeghi AR. Hardware-Assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In: Proc. of the 51st Annual Design Automation Conf. on Design Automation Conf. New York: ACM Press, 2014. 1–6. [doi: 10.1145/2593069.2596656]
- [34] Lucas DM, Hanreich DP, Sadeghi AR, Koeberl P, Sullivan D, Arias O, Jin Y. HAFIX: Hardware-Assisted flow integrity extension. In: Proc. of the 52nd Annual Design Automation Conf. on Design Automation Conf. New York: ACM Press, 2015. [doi: 10.1145/2744769.2744847]
- [35] Zhou HW, Wu X, Shi WC, Yuan JH, Liang B. HDROP: Detecting ROP attacks using performance monitoring counters. In: Proc. of the Int'l Conf. on Information Security Practice and Experience. Cambridge: Springer Int'l Publishing, 2014. 172–186. [doi: 10.1007/978-3-319-06320-1\_14]
- [36] Kim J, Eom YI. Fast and space-efficient defense against jump-oriented programming attacks. In: Proc. of the 2015 Int'l Conf. on Big Data and Smart Computing. Washington: IEEE, 2015. 7–10. [doi: 10.1109/35021BIGCOMP.2015.7072839]
- [37] Guide P. Intel® 64 and IA-32 architectures software developer's manual. 2015. <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325462.pdf>
- [38] Shinagawa T, Eiraku H, Tanimoto K, Omote K, Hasegawa S, Horie T, Hirano M, Kourai K, Oyama Y, Kawai E, Kono K, Chiba S, Shinjo Y, Kato K. Bitvisor: A thin hypervisor for enforcing I/O device security. In: Proc. of the 2009 ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments. New York: ACM Press, 2009. 121–130. [doi: 10.1145/1508293.1508311]

## 附中文参考文献:

- [7] 邢骁,陈平,丁文彪,茅兵,谢立. BIOP:自动构造增强型 ROP 攻击. 计算机学报, 2014, 37(5):1111–1123. [doi: 10.3724/SP.J.1016.2014.01111]
- [10] 李正玉,茅兵,谢立. 一种基于 JOP 的 rootkit 构造方法. 计算机科学, 2011, 38(10A):44–49.
- [12] 陈平. 代码复用攻击与防御技术研究[博士学位论文]. 南京: 南京大学, 2012.



陈志锋(1986 - ),男,福建漳州人,博士,讲师,主要研究领域为信息安全,可信计算.



李清宝(1967 - ),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为信息安全,可信计算.



张平(1969 - ),女,博士,副教授,CCF 专业会员,主要研究领域为并行识别,并行编译,信息安全.



王焯(1993 - ),男,硕士,主要研究领域为网络信息安全.