软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(1):489−508 [doi: 10.13328/j.cnki.jos.006436]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

进程控制流完整性保护技术综述*

张 正[1], 薛静锋[2], 张静慈[1], 陈 田[1], 谭毓安[1], 李元章[1], 张全新[1]

[1](北京理工大学 计算机学院, 北京 100081)

[2](北京理工大学 软件学院, 北京 100081)

通信作者: 张全新, E-mail: zhangqx@bit.edu.cn

摘 要: 控制流劫持攻击利用程序内存漏洞获取程序的控制权, 进而控制程序执行恶意代码, 对系统安全造成极大的威胁. 为了应对控制流劫持攻击, 研究人员提出了一系列的防御手段. 控制流完整性是一种运行时防御方法, 通过阻止进程控制流的非法转移, 来确保控制流始终处于程序要求的范围之内. 近年来, 越来越多的研究致力于解决控制流完整性的相关问题, 例如提出新的控制流完整性方案、新的控制流完整性方案评估方法等. 首先阐述了控制流完整性的基本原理, 然后对现有控制流完整性方案进行了分类, 并分别进行了分析, 同时介绍了现有针对控制流完整性方案的评估方法与评价指标. 最后, 对控制流完整性的未来工作进行了展望, 以期对未来的控制流完整性研究提供参考.

关键词: 控制流完整性; 控制流劫持; 控制流图; 系统安全

中图法分类号: TP316

中文引用格式: 张正, 薛静锋, 张静慈, 陈田, 谭毓安, 李元章, 张全新. 进程控制流完整性保护技术综述. 软件学报, 2023, 34(1): 489–508. http://www.jos.org.cn/1000-9825/6436.htm

英文引用格式: Zhang Z, Xue JF, Zhang JC, Chen T, Tan YA, Li YZ, Zhang QX. Survey on Control-flow Integrity Techniques. Ruan Jian Xue Bao/Journal of Software, 2023, 34(1): 489–508 (in Chinese). http://www.jos.org.cn/1000-9825/6436.htm

Survey on Control-flow Integrity Techniques

ZHANG Zheng[1], XUE Jing-Feng[2], ZHANG Jing-Ci[1], CHEN Tian[1], TAN Yu-An[1], LI Yuan-Zhang[1], ZHANG Quan-Xin[1]

[1](School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China)

[2](School of Software, Beijing Institute of Technology, Beijing 100081, China)

Abstract: Control-flow hijacking attacks exploit memory corruption vulnerabilities to grab control of the program, and then hijack the program to execute malicious code, which brings a great threat to system security. In order to prevent control-flow hijacking attacks, researchers have presented a series of defense methods. Control-flow integrity is a runtime defense method that prevents illegal transfer of process control-flow to ensure that control-flow is always within the range required by the program. In recent years, more and more research works are devoted to solving related problems of control-flow integrity, such as presenting new control-flow integrity schemes, new control-flow integrity scheme evaluation methods, etc. This study explains the basic principles of control flow integrity, and then classifies existing control flow integrity schemes. The existing evaluation methods and evaluation indicators of the control-flow integrity scheme are introduced at the same time. Finally, the thoughts on potential future work on control-flow integrity is summarized, which, hopefully, will provide an outlook of the research direction in the future.

Key words: control flow integrity; control flow hijacking; control-flow graph; system security

随着计算机技术的不断发展, 各类计算机软件的出现给人们带来极大的便利. 然而由于计算机体系和编程语言存在的固有局限性, 计算机软件不可避免地存在一些安全漏洞. C/C++因其较高的执行效率, 成为很多底层库、

* 基金项目: 国家自然科学基金 (U1936218, 62072037); 之江实验室开放课题 (2020LE0AB02)
收稿时间: 2021-05-02; 修改时间: 2021-06-24; 采用时间: 2021-08-18; jos 在线出版时间: 2022-09-30
CNKI 网络首发时间: 2022-11-16

大型软件实现的首选语言, 其内存管理机制需要开发者自行申请和释放内存, 这为设计和实现高效程序提供了极大自由度, 但同时也为内存安全埋下隐患. 输入数据超过缓冲区可容纳的最大数据量并溢出到临近存储区域便会引发缓冲区溢出漏洞. 缓冲区溢出漏洞是当今危害最为广泛的安全漏洞之一, 且难以彻底消除[1]. 控制流劫持攻击利用了缓冲区溢出等内存漏洞, 通过劫持程序进程的控制流, 使其执行特定的恶意代码, 从而达到攻击的目的.

为了抵御控制流劫持攻击, 一系列应对策略相继被提出, 根据防御引入时间可以分为运行前的静态防御技术和运行时的动态防御技术[2]. 其中运行前静态防御方法包括数据执行保护 (data execution protection, DEP)[3]、地址空间布局随机化 (address space layout randomization, ASLR)[4]、Stack Canary[5]等. 虽然此类技术已被广泛应用并可以有效防御代码注入攻击[6], 但是无法完全防御代码重用攻击 (code-reuse attacks, CRAs). Return-into-libc[7]和返回导向编程 (return oriented programming, ROP)[8]是常见的代码重用攻击, 其中 Return-into-libc 利用目标程序的库函数完成对目标进程的攻击, ROP 则可以从目标程序中选取已有的代码片段 gadget, 串连形成图灵完备的攻击逻辑进而完成攻击, 显示出巨大的威胁性. 由于 ROP 攻击需要获取内存布局信息来获取 gadget 的地址, 以 ASLR 为代表的运行前静态防御方法旨在通过随机化内存布局, 来阻止 ROP 攻击构造攻击逻辑. 然而跳转导向编程 (jump oriented programming, JOP)[9] 、JIT-ROP (just-in-time ROP)[10]等 ROP 的变种攻击方法相继被提出, 不断突破已有的防御技术, 进一步增加了防御控制流劫持攻击的难度.

2005 年, Abadi 等人提出了控制流完整性 (control flow integrity, CFI)[11] 的概念. CFI 是一种运行时动态防御方法, 其目标是确保进程运行时的控制流始终处于要求的范围之内, 并严格执行程序要求的转移. ROP 执行的恶意代码会被 CFI 视为非法控制流转移, 并阻止其进一步运行, 从而起到防御控制流劫持攻击的作用. CFI 自提出以来, 已经经历了十多年的发展, 研究人员在原始方法的基础上提出了多种应用于不同场景的 CFI 改进方案, 包括用于商业软件的 CFI 或者直接集成在编译器、操作系统中的 CFI 等, 满足程序实际运行环境的需求.

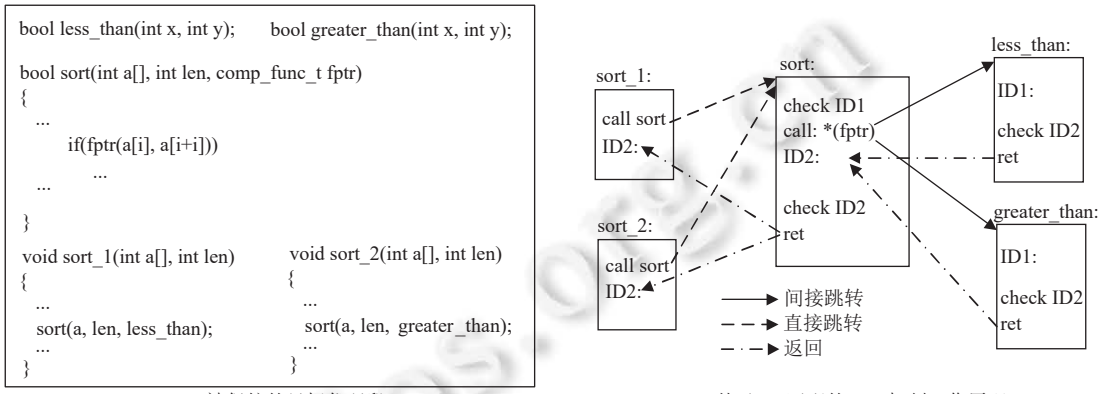
本文对 CFI 方法进行了全面的调查和论述, 第 1 节阐述了 CFI 的基本原理, 从技术原理的角度将现有的 CFI 方法分类为上下文无关的 CFI 和上下文敏感的 CFI; 第 2 节介绍了本文的文献调研方法和调研结果, 并对不同方法进行了归纳总结; 第 3 节将上下文无关的 CFI 分类为基于源码的 CFI、基于二进制的 CFI 和硬件支持的 CFI 并分别阐述了其原理机制; 第 4 节介绍了上下文敏感的 CFI; 第 5 节阐述了 CFI 的评价方法, 并根据评估角度对现有的 CFI 评估研究进行对比、讨论和分析; 最后, 第 6 节对全文进行了总结与展望.

1 CFI 原理概述

CFI 对控制流的保护主要分为两个阶段, 首先是在运行前对程序进行源码分析或二进制分析, 获得程序的控制流图 (control flow graph, CFG), 然后在程序运行时监测程序控制流是否始终处于 CFG 中, 来判断程序是否遭受了攻击. CFG 是一个由基本块和有向边组成的图, 其中每一个基本块代表一段在中间不包含控制流跳转指令和其他控制流跳转指令的目标的连续代码, 每一条有向边则代表程序控制流的一次转移. CFI 机制使用的 CFG 为过程间 CFG, 包含了函数间的调用边 (call edge) 和返回边 (return edge). 图 1(b) 所示的 CFG 由图 1(a) 的原始程序段分析得到, 它反映了原始程序段的控制流, 即程序所有的控制流转移指令以及程序执行过程中所可能跳转的目标地址集合.

控制流的转移以跳转指令为基础. 程序通过跳转指令进入不同分支执行, 而保证跳转指令的目标分支地址始终合法即维护了进程控制流的完整性. 以 x86 平台为例, 跳转指令包括 jmp 指令、call 指令和 ret 指令. 其中 jmp 指令和 call 指令将程序控制流转移到一个新的位置, 称为前向转移; ret 指令将程序的控制流返回到先前的位置, 称为后向转移. 跳转指令根据寻址方式的差异, 可以分为直接和间接两种形式. 其中直接跳转指令根据给定的目标绝对地址或相对偏移量完成跳转, 且地址在运行时无法修改. 因此在 DEP 技术有效的前提下, 只要保证代码的完整性, 直接跳转可以被视为安全指令; 间接跳转指令的目标地址由寄存器给出, 且在程序执行时动态决定, 因此间接跳转指令的目标地址有被攻击者恶意篡改的风险. 因此 CFI 的目标在于确保进程间接跳转指令的目标地址始终合法, 对前向间接转移的保护称为前向边沿保护 (forward-edge protect), 对后向间接转移的保护称为后向边沿保护 (backward-edge protect).

图 1(b) 展示了基于 ID 匹配的 CFI 机制防护控制流劫持攻击的原理. 基于 ID 匹配的 CFI 机制为跳转目标函数分配唯一的 ID, 并在每条控制流转移指令前插入检查代码, 以在跳转前比较预期跳转 ID 与目标函数的 ID 是否一致. 当出现不一致的情况时, 说明进程出现了违背 CFG 的非法转移, 则终止程序的运行. 具体到图 1(a) 所示的代码段, 为了避免由于 ROP 攻击对函数指针 fptr 的目标地址进行恶意修改, ID1 检查用于验证从 fptr 的函数 sort 到目标函数 less_than 和 greater_than 的合法跳转, 任何到其他目的地的非法跳转都不能通过 ID1 检查, 因为那些目标地址没有插入 ID1. 同时, ID2 检查用于验证调用函数对函数调用点的合法返回, 以避免攻击者对堆栈中的返回地址进行恶意的篡改.



(a) 被保护的目标代码段 (b) 基于 ID 匹配的 CFI 机制工作原理

图 1 程序控制流图及 CFI 机制的工作原理图

研究人员在原始 CFI 思想的基础上, 提出了一系列应用于不同场景的 CFI 方案. 现有工业界和学术界提出的 CFI 方案囊括了基于源代码的实现、基于二进制可执行文件的实现、编译器层面、内核层面的实现以及利用 CPU 硬件特性的实现等诸多层面, 其适用范围、防护安全性以及运行开销均有不同. 此外, 为了抵御愈来愈强大的控制流劫持攻击, 研究人员开始考虑将进程执行状态的上下文语义加入到 CFI 方案中, 相对于此前无状态的 CFI 方案, 进一步提高了其防御能力. 尽管 CFI 方案的形式多变, 其目的始终是为了获得更好的防御性能和更低的性能开销.

如图 2 所示, 本文根据是否结合进程上下文语义, 将现有 CFI 分为上下文无关的 CFI 和上下文敏感的 CFI, 根据实现层面的不同, 上下文无关的 CFI 又可分为基于源码的 CFI 方案、基于二进制程序的 CFI 方案和硬件支持的 CFI 方案. 上下文敏感的 CFI 方案利用了进程执行时函数调用过程的上下文语义信息, 进一步保证进程跳转过程符合控制流图的执行逻辑, 需要借助处理器硬件机制实现对进程控制流的高效监控, 以减少运行开销.

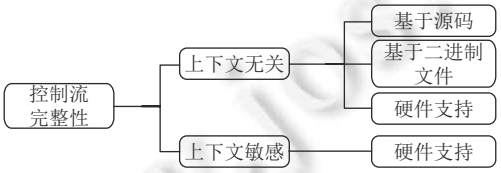


图 2 CFI 机制的分类

2 研究概况

本节介绍了本文的文献调研方法和调研结果. 目前 Burow 等人[12]、de Clercq 等人[13]、武成岗等人[14]、王丰峰等人[2]对已有的控制流保护方面的研究成果从不同方面进行了系统的总结. Burow 等人[12]对近年来已有的 CFI 方案整理和分析, 重点从 CFI 方案的准确性、安全性和性能进行了细致的评估和总结. de Clercq 等人[13]总结了近年来

的 21 个基于硬件支持的 CFI 方案, 并对这些方案的安全性、局限性、硬件成本、性能和实用性进行了评估与比较. 武成岗等人[14]以时间顺序介绍了控制流完整性技术的发展历程, 同时介绍了针对 CFI 的攻击手段. 王丰峰等人[2]系统地总结了进程控制流劫持攻击及其防御技术的相关研究, 首先介绍了进程控制流劫持攻击技术的研究现状, 然后将控制流劫持防御技术分为运行前静态防御技术和运行时动态防御技术分别进行介绍. 现有综述性文献未对 CFI 方案进行明确的分类, 且均未对 CFI 方案的评价方法及指标进行统一的归纳. 针对这一问题, 本文对近年来已有的各类 CFI 方案进行了充分的调研, 并从实现原理角度对 CFI 技术进行了系统的分类汇总. 在评估各类 CFI 方案优势与不足的同时, 进一步对 CFI 技术原理的发展过程与趋势进行了分析; 此外, 本文还关注针对 CFI 方案的评价方法, 对相关评测基准套件、现有评价指标以及评估方法类文献进行了归纳与总结.

为了对本文所研究的问题进行系统的梳理和分析, 首先以“Control Flow Integrity”“Control-Flow Attacks”“Control Flow Hijacking”等设为主要搜索关键词, 在国内外重要的学术搜索引擎 (例如谷歌学术搜索、CNKI 等) 中检索相关论文; 然后, 筛选出与综述问题相关的论文; 接着, 通过已搜索到的论文获取更多的相关文献, 方法包括查阅论文的引用和被引用, 以及论文作者已发表论文的列表; 同时参考已有综述性文献的调研情况, 确定了最终的文献列表. 其中包括提出 CFI 方案的论文 60 篇、相关评测基准套件或方法 17 个、其他与综述研究问题直接相关的论文 30 篇.

图 3 描述了不同 CFI 方案的论文随时间线的增长图. 从图中可以看出, 自 CFI 于 2005 年提出以来, 基于源码的 CFI 方案最早被提出且逐年稳步增加; 随着计算机硬件的迭代更新, 利用硬件特性支持的 CFI 技术逐步代替纯软件的实现成为发展的趋势. 此类硬件具有分支转移的追踪记录功能, 能够使得 CFI 摆脱对二进制改写技术的依赖, 尽可能地保持被保护代码的完整性和透明性; 同时, 此类硬件也提供了丰富的进程上下文信息, 使得上下文敏感的 CFI 的实际应用成为了可能. 到目前为止, 各方向论文的分布情况为: 基于源码的 CFI 方案有 20 个; 基于二进制程序的 CFI 方案有 13 个; 硬件支持的 CFI 方案有 22 个; 上下文敏感的 CFI 方案有 5 个. 虽然上下文敏感的 CFI 方案目前较少, 但具有较大的发展潜力, 是未来的研究趋势.

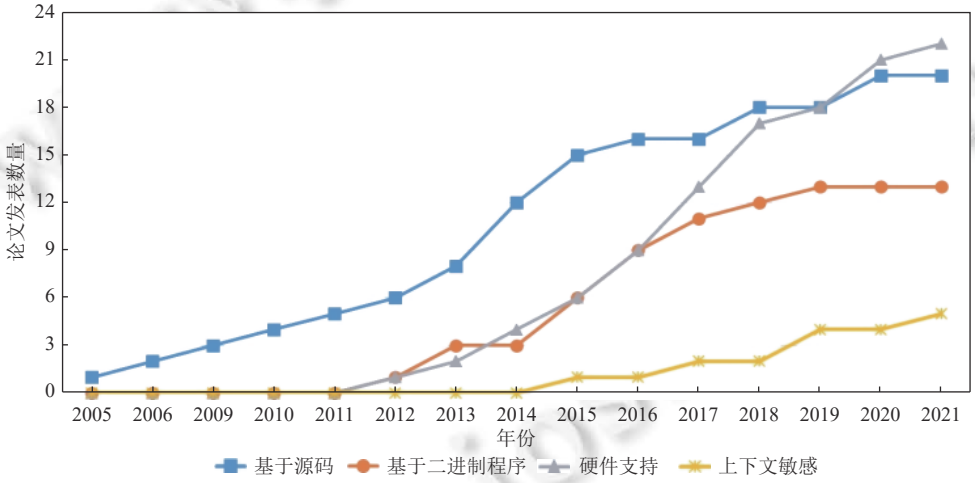


图 3 论文数量增长趋势

表 1 列举了近年来关于控制流完整性的研究汇总, 概括地描述了 CFI 方案的类别及其技术特点. 类别分为上下文无关的 CFI 和上下文敏感的 CFI. 上下文无关的 CFI 又可进一步细分为基于源码的 CFI、基于二进制程序的 CFI 和基于硬件的 CFI; 技术特点概括了不同 CFI 方案实现控制流完整性保护所使用的技术原理. 后文表 1 同时列出了 CFI 方案的名称、发表年份和在本文中的引用编号.

3 上下文无关的 CFI

上下文无关的 CFI 不将进程执行的历史信息作为控制流转移合法性的参考依据, 只需保证控制流遵循预先确

定的 CFG 的路径. 根据 CFG 获取方式的不同, 上下文无关的 CFI 可以进一步细分为基于源码的 CFI、基于二进制程序的 CFI 和硬件支持的 CFI. 基于源码的 CFI 可以通过被保护程序的源码确定每个间接转移指令的合法目标地址, 从而确定 CFG 中每个基本块之间是否存在可以转移的有向边, 因此可以构建精准的 CFG; 不依赖源代码的 CFI 只使用二进制文件进行分析, 所得到的间接转移指令的目标地址集合与实际合法的目标地址集合存在差异, 这往往需要通过分析被保护程序所依赖的库、匹配函数调用时的参数数量[15]等方式来进一步筛选, 从而尽可能地优化 CFG.

表 1 CFI 方案总结

文献	年份	类别	技术特点
HyperSafe[16]	2010	基于源码	内存页面锁定+间接跳转目标白名单
MIP[17]	2013	基于源码	代码分块+边界检查
MCFI[18]	2014	基于源码	代码分块+唯一ID匹配
Forward-CFI[19]	2014	基于源码	间接跳转目标白名单, 只保护前向边沿
KCoFI[20]	2014	基于源码	唯一ID匹配结合影子堆栈保护返回地址
RockJIT[21]	2014	基于源码	代码分块+唯一ID匹配
JITScope[22]	2015	基于源码	唯一ID匹配结合影子堆栈保护返回地址
πCFI[23]	2015	基于源码	将特定输入对应的CFG边沿逐一加入全局CFG中
Fine-CFI[24]	2018	基于源码	函数指针分析
IBV-CFI[25]	2020	基于源码	为所有间接转移指令生成独立的位向量, 并在位向量中存储每个间接转移指令的有效目标集
MoCFI[26]	2012	基于二进制程序	间接跳转目标白名单+影子堆栈
CCFIR[27]	2013	基于二进制程序	使用Springboard段验证间接跳转目标并转发
BinCFI[28]	2013	基于二进制程序	间接跳转目标白名单
BinCC[29]	2015	基于二进制程序	代码分块+边界检查
O-CFI[30]	2015	基于二进制程序	细粒度的代码随机化和粗粒度的控制流保护相结合
Lockdown[31]	2015	基于二进制程序	代码分块+影子堆栈
TypeArmor[32]	2016	基于二进制程序	构造控制流不变式, 只保护前向边沿
τCFI[33]	2018	基于二进制程序	函数参数类型和参数数量匹配验证
RECFISH[34]	2019	基于二进制程序	标签匹配保护前向边沿+影子堆栈保护返回地址
CFIMon[35]	2012	硬件支持	BTS辅助收集进程的间接跳转目标地址
kBouncer[36]	2013	硬件支持	LBR辅助收集进程的间接跳转目标地址
ROPecker[37]	2014	硬件支持	LBR辅助收集间接跳转目标地址+滑动窗口机制
CFIGuard[38]	2015	硬件支持	LBR和PMU组合实现控制流保护
PT-CFI[39]	2017	硬件支持	PT辅助收集间接跳转目标地址并构成图结构+影子堆栈
μCFI[40]	2018	硬件支持	PT辅助收集历史跳转目标, 确保间接跳转目标的唯一性
HCIC[41]	2018	硬件支持	PUF辅助的线性加密/解密+加密的汉明距离匹配
PARTS[42]	2019	硬件支持	基于指针验证(PA)机制保护指针完整性
ABCFI[43]	2020	硬件支持	将标签构造为地址低位
SPECCFI[44]	2020	硬件支持	将CFI原则嵌入分支预测决策中
FastCFI[45]	2021	硬件支持	由FPGA实现并对ARM平台上运行程序进行监测
PathArmor[15]	2015	上下文敏感	LBR辅助实现运行时执行路径监控
PITTYPAT[46]	2017	上下文敏感	PT辅助重建执行路径, 实现运行时执行路径监控
OS-CFI[47]	2019	上下文敏感	将间接转移指令调用的代码指针的起源作为上下文
CFI-LB[48]	2019	上下文敏感	每个间接跳转分支具有自适应上下文敏感性
BCI-CFI[49]	2021	上下文敏感	将分支相关关系作为上下文信息

Abadi 等人提出了第一个上下文无关的 CFI, 并给出了具体实施方案[50]. 原始 CFI 基于唯一 ID 匹配的原理实现控制流完整性保护, 如图 4 所示为其基本框架. 原始 CFI 在每一条间接转移指令前插入检查代码, 其中包括间接 call 指令、间接 jmp 指令和 ret 指令在内的所有间接转移指令都需确保合法后才能够继续执行, 合法性判断所依据的 CFG 由被保护程序的静态二进制分析得到. 当间接转移指令被检测为合法跳转时, 程序可以继续执行, 否则将被 CFI 机制视为非法跳转并终止被保护进程. 为了确保函数调用能够返回到最近的调用点, 原始 CFI 应用了影子堆栈[51]来保护存有返回地址的堆栈, 防止其在运行时的被篡改. 影子堆栈常用来保护函数的返回地址, 每当程序调用一个函数时, 都会将返回地址复制在专用的影子堆栈上, 函数返回时, 将程序要使用的返回地址与存储在影子堆栈中的地址进行比较, 仅当二者一致时, 才认为返回地址没有遭到恶意的篡改. 原始 CFI 针对每一个控制流转移均进行检查, 可以最大化系统的安全性. 但是由于 ID 的创建、查询、比较和存储会导致性能大幅下降, 以至于难以在实际生产中部署. 为了减少检查的开销, 后续出现了放松检查条件的粗粒度 CFI, 通过降低防御安全性的方式来换取性能的提升.

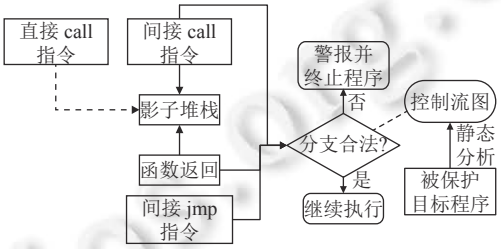


图 4 原始 CFI 的基本框架

3.1 基于源码的 CFI

基于源码的 CFI 利用程序源代码获取程序间接转移的详细信息, 从而构建精准的 CFG. 此类方案在实际部署时, 或直接对源代码进行修改, 或将控制流保护逻辑集成在编译器或者操作系统内核中, 从而在程序运行前完成攻击检测与防御代码的安插. 相较于基于二进制程序的 CFI 方案, 可以执行更多的优化从而获得更好的性能.

Wang 等人[16]提出了一种保护虚拟机管理程序的细粒度 CFI 保护机制 HyperSafe, 该方案主要针对 Type-I 虚拟机管理程序提供运行周期内的控制流保护. HyperSafe 通过两种关键技术实现其目标: 第 1 种技术可锁定受写保护的内存页面, 在页表更新之前将其设置为只读并检测更新是否安全, 防止内存页在运行时被恶意代码篡改, 从而有效地保护了虚拟机监控程序的代码完整性. 第 2 种技术将间接转移的所有目标地址集中在一张目标表中, 并使用指针索引替换实际的目标地址, 从而将控制流的跳转限制在目标表中存在的地址范围内. 第 1 种技术通过直接修改虚拟机管理程序的源码来实现; 第 2 种技术利用编译器 LLVM[52]中实现的数据结构分析功能和手动分析相结合的方法来获取间接跳转目标的地址. 该文献使用 HyperSafe 实现了对两个虚拟机管理程序 BitVisor 和 Xen 的保护, 从而验证了该方案的有效性.

Niu 等人[18]提出一种模块化支持的 CFI 方案 MCFI, 该方法支持对模块进行独立的控制流安全检测, 并将各个模块以静态或动态方式链接起来, 组合生成新的 CFG. MCFI 使用 ID 表管理各个模块, 并使用锁机制实现事务来维护 ID 表的一致性, 进一步实现模块的多线程同时访问. MCFI 将类型信息附加在各个模块中, 并使用函数指针的类型与函数匹配来生成精确的 CFG, 同时需要对源代码进行适当地修改. 各个模块类型信息的生成则依赖于可拓展的编译工具 LLVM.

Tice 等人[19]提出了第一个集成到编译器中的 CFI 前向边沿保护方案, 并在 GCC 和 LLVM 两种编译器上进行了实现. 作者针对 GCC 编译器提出了虚函数表验证 (vtable verification, VTV), 防止攻击者通过伪造的虚函数表劫持虚拟调用进行攻击. VTV 构造了有效虚函数表指针集合, 在执行调用之前, 验证用于虚拟调用的虚函数表指针的有效性, 对于不在集合中的虚函数表指针, VTV 将终止程序的运行. 针对 LLVM 编译器, 作者提出间接函数调用检查 (indirect function-call checks, IFCC), 可以确保间接调用的目标地址均为合法地址. IFCC 为间接调用目标生成

跳转表并强制所有间接调用通过跳转表执行, 大大减少了间接跳转目标被攻击的可能性. 文章还设计了 FSan, 这是一个可选的间接调用检查工具. FSan 被集成在 LLVM 中, 具有对类型信息的完全访问权限, 用来识别可能导致安全问题的违反 CFI 的行为. 然而, Conti 等人[53]指出 IFCC 只保护了 CFG 的前向边沿, 使得其后向边沿易受栈溢出攻击的劫持.

上述方法的关注重点以应用层的 ROP 攻击为主, 针对内核层级的攻击检测方法相对较少. Criswell 等人[20]提出了第一个内核级的 CFI 保护方案 KCoFI, 用于保护操作系统内核免受控制流劫持攻击. KCoFI 基于安全虚拟体系结构 (secure virtual architecture, SVA)[54,55]实现, SVA 在硬件和操作系统之间插入了基于编译器的虚拟机, 所有软件都被编译为 SVA 提供的虚拟指令集. KCoFI 将基于标签的原始 CFI 部署在该虚拟机中, 并在代码从虚拟指令集转换为处理器的原生指令集时进行安全检测, 实现对操作系统内核数据的保护. 实验结果表明, KCoFI 可以阻止内核中检测到的所有 gadgets 的使用, 证明了该方法的有效性. 然而, 将 CFI 机制放置于内核中的风险在于, 一旦内核遭受攻击, 则检测机制会被攻击者禁用从而失效. 为了解决这个问题, 陈志锋等人[56]利用虚拟机架构实现了内核级 ROP 攻击检测方法 CFI-KCraD, 该方法基于标签验证的思想确保控制流转移的合法性, 并通过将检测程序放置在虚拟机监控器中, 提高了 CFI 机制的安全性.

Niu 等人[21]提出了一种保护即时编译器的 CFI 方案 RockJIT, 该方案使用即时编译器的源代码构建细粒度的控制流图, 并在生成新代码时动态更新控制流策略, 以防御代码注入攻击和 JIT spraying[57]等控制流劫持攻击. RockJIT 基于 MCFI 构建, 将编译器新生成的代码看作新的模块, 并将其 CFG 与现有代码的 CFG 链接在一起. 基于即时编译器的 CFI 方案还有基于 NaCL[58]实现的 NaCL-JIT[59]以及 JITScope[22]等, 也实现了较好的防御性能.

传统的 CFI 技术从程序中静态提取 CFG, 并通过执行该 CFG 实现进程控制流的监测. 静态生成的 CFG 包括所有可能输入的所有边沿, 但对于具体的输入, CFG 可能保护许多不必要的边沿. 针对这一问题, Niu 等人[23]提出 πCFI, 该方案根据每个具体输入计算 CFG. πCFI 首先通过静态分析得到的全局 CFG, 并在运行时向一个空 CFG 中逐一添加特定输入对应的 CFG 边沿, 且添加条件为特定输入对应的 CFG 边沿包含在全局 CFG 之中. 实验结果表明, πCFI 可以有效地减少不必要的间接分支数量, 同时降低开销.

此外一些研究提出针对 IRM (inlined reference monitors) 的控制流防护. IRM 是一种底层安全机制, 该机制将监视器代码内联到汇编等低级代码, 以便在执行不安全操作之前对其进行检查. IRM 可以依靠 CFI 或者 SFI (software fault isolation)[60]来实现. 其中 XFI[61]和 MIP[17]等研究提出了针对监控器程序的 CFI 方案, 从而实现了低层级的 IRM.

基于源代码的 CFI 多用于偏底层的控制流保护, 此类方法在内核层面实现相应接口的扩展并通过编译器为程序增加防御功能, 避免了开发人员手动添加防御代码, 为软件开发提供了便利. 然而此类方法在不提供源码或调试信息的应用程序中难以实现. 实际上, 商业软件往往以二进制可执行文件的形式发布, 用户在大多数情况下不能获取软件的源码, 这使得更多的 CFI 技术倾向于直接基于二进制文件或借助处理器硬件特性实现控制流保护.

3.2 基于二进制程序的 CFI

许多 CFI 解决方案要求源码或调试信息才可正常使用, 在现实应用中往往难以得到满足, 这使得许多商业软件无法得到有效的保护. 因此, 基于二进制程序的 CFI 方案得到越来越多科研人员的关注. 基于二进制程序的 CFI 方案可以部署在没有源码的二进制程序中, 因此应用更加广泛. 此类方案通常会使用一些方法从二进制文件中生成控制流图, 设计 CFI 方案, 并使用二进制重写等技术将方案部署在原程序中. 由于无法使用源码来构建更加精准的控制流图, 所以获取更准确地间接跳转目标集合, 提高生成控制流图的精度成为研究人员的研究重点之一.

Zhang 等人[27]提出了 CCFIR, 该方法收集了间接转移指令的所有合法目标地址, 在进程内存中设置一个 Springboard 段, 并将所有合法目标地址放置在该内存区域中. 进程执行过程中的所有间接跳转都需经过 Springboard 段的转发, 而 Springboard 段会对间接跳转指令的目标地址进行验证, 如果存在非法跳转地址, 则说明进程受到恶意的控制流攻击. CCFIR 使用二进制重写技术实现间接跳转目标地址向 Springboard 段的重定向, 因此仅需要重定位表的信息即可应用. CCFIR 将跳转目标函数进行了分组, 并且以组为单位分配 ID, 用于区分间接调用指令和函数返回指令的目标. 相较于原始 CFI 细粒度的检查方式, 分组的思想通过降低防御的粒度减小了运行

开销. 此外, Springboard 段的使用也加大了空间需求.

与 CCFIR 类似, BinCFI[28]也是一种面向二进制程序的 CFI 解决方案, 该方法使用反汇编改写间接跳转指令, 生成新的代码段, 在不改变原有代码的基础上, 实现原程序的控制流完整性增强. BinCFI 维护了两个地址转换哈希表, 分别维护 ret 指令和间接 jmp/call 指令的合法目标集合, 用于原代码段到新代码段地址的转换. 实验表明, 该方法可以应用于大型二进制程序中, 具有较好的性能.

Wang 等人[29]指出了 CCFIR 和 BinCFI 存在的不足以及可能被攻击的路径, 并提出了一种更细粒度的面向二进制程序的 CFI 保护方案 BinCC. 该方法首先对二进制代码进行静态分析, 并使用 Super-CFG 构造算法构造互斥的代码块, 进一步将间接转移指令分为代码块间转移或代码块内转移. 代码块的内部构造有向图, 从而保证代码块间的互斥, 实现更细粒度的约束. 在 CFI 约束规则的实施方面, BinCC 在 BinCFI 的基础上进行拓展, 本质上也是使用了二进制重写的方法.

O-CFI[30]将细粒度的代码随机化和粗粒度的控制流保护相结合, 在使用查询表检验间接转移目标地址合法性的同时, 对进程内存布局进行动态随机化处理, 降低内存漏洞导致的内存信息泄露风险. Lockdown[31]使用了影子堆栈来保护返回地址, 从而实现细粒度的控制流保护; 与其他使用静态二进制重写的 CFI 解决方案不同, 该方法使用动态二进制翻译对所有间接转移进行安全检查, 在运行时动态调整控制流图的大小. TypeArmor[32]部署了面向目标和面向调用点的控制流不变式, 实现了更加严格的控制流劫持攻击的检测, 并成功抵御了伪造面向对象编程 (counterfeit object-oriented programming, COOP)[62]. Grossklags 等人[33]针对 BinCFI 和 TypeArmor 等方法只对前向边沿进行保护, 并假设已使用影子堆栈等技术保护后向边沿的不足, 提出了一种基于函数参数类型和参数数量匹配验证的 CFI 方案 τCFI, 对程序的前向边沿和后向边沿均进行了有效的保护. Davi 等人[26]提出了第一个针对智能手机平台的通用 CFI 框架 MoCFI, 并在 iOS 上成功部署. 实验结果表明, MoCFI 可以成功抵御常见的控制流劫持攻击, 且不会引起明显的性能开销.

随着商用软件对版权保护及数据安全等需求的日益增加, 在二进制程序的基础上增加控制流防护措施变得尤为重要. 通过静态分析和二进制重写的方法实现 CFI 机制需要修改程序原本的二进制文件, 实际应用时面临较大的兼容性问题. 随着历史分支记录等机制的出现, 使用了类似硬件机制的 CFI 技术逐步摆脱了对二进制重写技术的依赖, 提高了防御的兼容性. 因此, 在仅有被保护软件二进制文件的情况下, 硬件支持的 CFI 逐步取代纯软件的实现方法.

3.3 硬件支持的 CFI

为了进一步提高 CFI 的安全性和性能, 一些研究着眼于从硬件机制中寻求可用的支持. 目前商用处理器中基本都集成了性能监控单元 (performance monitoring units, PMUs), 该功能最初用于监控应用程序的运行从而优化系统性能. 为了分析程序的控制流转移情况, 进一步提高 PMUs 的精确性, 大部分商用处理器还集成了分支追踪功能, 例如英特尔处理器的分支追踪存储 (branch tracing store, BTS) 和最近分支记录 (last branch recording, LBR) 等. 基于底层硬件功能进行程序控制流的监控, 检测控制流劫持攻击, 可以有效地降低 CFI 方案的运行开销.

Xia 等人[35]提出了 CFIMon, 该方法使用了处理器中的 BTS 机制即时地监控和分析控制流的运行状态. BTS 可以收集程序运行时所有的跳转指令信息并存入特定的缓冲区. 当历史指令即将装满 BTS 的缓冲区, 或进程尝试进行敏感的系统调用时, CFIMon 将启动指令合法性验证. 验证过程主要是判断缓冲区内所有的历史间接转移指令是否均存在于合法跳转目标地址集合中, 若存在非法跳转则判定当前进程受到攻击. 合法跳转目标集合由被保护程序在运行前通过静态分析得到. CFIMon 利用 BTS 辅助获取进程在运行时的跳转目标地址, 是第一个不需要二进制重写技术实现的 CFI 方案, 同时也无需提供源码和编译信息.

图 5 展示了硬件支持的 CFI 的典型工作原理, BTS、LBR 等分支追踪寄存器收集程序运行时所有的跳转指令信息并存入特定的缓冲区, 当进程进行系统调用时, 将触发 CFI 机制的间接转移合法性检测功能. 跳转信息收集模块获取缓冲区中的进程跳转指令并传递给跳转合法性检测模块, 跳转合法性检测模块根据合法跳转目标集合对进程的历史跳转指令进行合法性判断. 在此期间 CFI 获取进程控制流并执行检测过程, 如果间接跳转指令判断为合法指令, 则将控制流返还给被保护进程, 否则将终止程序.

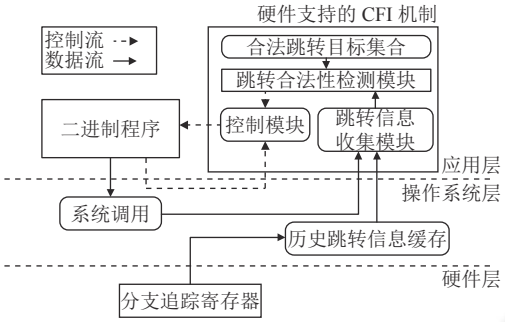


图 5 硬件支持的 CFI 机制的工作原理图

Pappas 等人[36]提出了 kBouncer, 该方案利用 LBR 获取进程最近 16 次的跳转指令信息, 并使用类似 CFIMon 的方式, 在进程试图进行敏感系统调用时进行跳转目标合法性检查. 作者对 IE9、Windows Media Player 和 Adobe Reader 进行了验证实验, 证明了该方法可以有效抵御 ROP 攻击. Cheng 等人[37]指出 kBouncer 完全依赖 LBR 记录的局限性以及使用二进制重写技术存在破坏安全机制兼容性的风险, 并提出了一种不依赖源码和二进制重写技术的 CFI 方案 ROPecker, 该方案也使用 LBR 监控程序的控制流, 进而检测并防御 ROP 攻击. ROPecker 对 ROP 攻击的检测触发条件进行了创新, 提出了一种滑动窗口机制. 该机制将受保护应用程序的最近访问代码包含在滑动窗口内部, 并且设置为可执行, 而将滑动窗口外部的应用程序代码设置为不可执行. 当进程试图执行敏感系统调用或者控制流试图跳出滑动窗口范围时都将会触发 ROP 攻击检测机制. 滑动窗口机制利用了应用程序代码的时间和空间局部性, 具有高效性和较高的准确性. 实验结果证明, ROPecker 可以有效地抵御 ROP 攻击. 上述 3 种方法均是在运行时检测执行流中的长 gadgets-chain 来实现 ROP 攻击的检测, 然而对短 gadgets-chain 攻击的检测与规避效果有待提高. 李威威等人[63]提出 MIBChecker, 将敏感系统调用作为检测触发条件. 由于短 gadgets-chain 攻击的实施者往往通过构造系统调用参数, 进行敏感系统调用从而达到攻击的目的, 包括 mprotect、execve、mmap 等, 因此 MIBChecker 检测系统将每次检测到攻击时的系统状态记录下来, 在执行系统调用时判断当前系统状态是否和检测到攻击时的系统状态一致, 如果一致则判定为遭受到了 ROP 攻击.

Gu 等人[39]提出了一种基于硬件实现的后向边沿保护机制 PT-CFI, 该方案利用硬件功能 Intel processor trace (PT) 提供进程控制流信息.PT 实时收集控制流信息并以数据包的形式发送至内存缓冲区, 当间接转移发生时将触发硬件功能生成 Target IP (TIP) 数据包, 其中包含了间接转移的目标地址. PT-CFI 使用实时接收到的 TIP 包构建 TIP 图结构, 其中图的顶点表示 TIP 包的唯一索引, 而只有两个顺序执行的间接转移的 TIP 包之间才有边相连. 当一系列的间接转移发生时, PT-CFI 将 PT 收集到的 TIP 包序列化并与 TIP 图进行匹配, 如果匹配成功则判定为合法跳转; 如果匹配失败且 TIP 包中的指令类型是 ret 时, 将构造影子堆栈来验证返回地址是否合法, 如果合法则作为新节点添加入 TIP 图中, 否则判定为非法跳转, 终止进程. PT-CFI 实现了动态执行时的后向边沿防御, 无需使用静态二进制重写技术, 但是该方法未对 call/jmp 类型的 TIP 包进行相应的处理, 因此无法保护控制流的前向边沿. 针对现有基于纯软件实现的内核层 CFI 方案需要对内核源码进行分析和修改的问题, 王心然等人[64]提出使用 PT 结合虚拟化技术实现对内核的控制流保护, 通过硬件机制避免了修改内核源码以及重新编译内核, 提高了内核控制流保护的透明性.

一些方案将密码学应用于控制流的保护, 通过对间接转移的返回地址或函数指针进行加密, 阻止攻击者对其进行篡改. Qiu 等人[65]提出了一种基于物理不可克隆函数 (physical unclonable function, PUF) 的线性加密体系结构 (linear encryption architecture, LEA) 以防御控制流劫持攻击. LEA 是一个附加硬件模块, 由加解密单元 (encryption-decryption unit, EDU), 两个寄存器 (KEY_CFI 和 LEN_CFI) 和 PUF 模块组成. 受保护计算机与 EDU 相连并进行通讯, 寄存器和 PUF 在内部与 EDU 一起使用. PUF 模块用于生成加密/解密密钥, 并存储在 KEY_CFI 中, 所用密钥的长度由间接转移指令加密的要求决定, 并存储在 LEN_CFI 中. LEA 在运行时对返回地址和目标地址的第一条指令的某些字节进行加密和解密, 具体操作是将数据与 PUF 生成的密钥进行异或操作. 由于 PUF 产生随机的, 硬件唯一且不可克隆的密钥, 大大提高了被保护程序的安全性, 提高了攻击的成本. 然而, PUF 硬件可能遭

到选择明文攻击 (chosen plaintext attack, CPA), 攻击者可以通过内存泄漏或调试来推断 PUF 密钥. CCFI[66]和 LEA-AES[67]使用高级加密标准 (advanced encryption standard, AES) 代替 PUF 消除这一漏洞, HCIC[41]使用 4 个寄存器 KEY_1, KEY_2, KEY_LEN_1, KEY_LEN_2 存储两组密钥, 在将返回地址存储在内存结构中之前, 计算调用函数的返回地址与 PUF 响应之间的加密的汉明距离 (encrypted Hamming distance, EHD). 然后, 当执行 ret 指令时, 将在运行时计算 EHD. 最后, 将预先计算的 EHD 与运行时计算出的 EHD 进行比较, 以验证 EHD 是否匹配. 如果攻击者修改了堆栈中的返回地址, 则 EHD 将不匹配, 从而阻止攻击者利用内存漏洞获得 PUF 密钥.

此外, 还有其他 CFI 方案也基于此类硬件机制得到较好的防御效果[68−70]. CFIGuard[38]基于 LBR 和 PMU 的组合实现了针对代码复用攻击的 CFI 方案; TSX-based CFI[71]使用英特尔的事务同步扩展 (transactional synchronization extensions, TSX)[72]机制, 将控制流转换映射到事务中; Ge 等人[73]基于 PT 设计了 GRIFFIN, 支持多种类型的 CFI 方案, 可以根据需求实现安全性与性能之间的灵活调节; μCFI[40]提出了 CFI 的唯一代码目标属性, 即规定对于任何间接转移指令的每次调用, 都有且只有一个允许的目标. μCFI 利用 PT 机制对受保护程序严格执行这一属性, 得到较好的防御效果.

基于 LBR、PT 等硬件实现的 CFI 方案可以极大攻击检测的效率, 解决纯软件 CFI 方案高开销的问题, 但是此类防御需要相应硬件的支持, 没有对底层处理器架构进行更改. 不过随着 ROP 等控制流劫持攻击的流行, 包括 Intel 在内的处理器供应商开始逐步将安全原语集成到处理器设计中, 以有效应对特定的攻击. 例如最近加入到 ARMv8-A 架构的新指令指针验证 (pointer authentication, PA) 可以保护指针完整性, 这种直接将安全原语集成在底层处理器架构中的方式可以减小 CFI 方案对特定硬件的依赖, 将是硬件辅助的 CFI 方案的发展趋势.

4 上下文敏感的 CFI

随着控制流劫持攻击的迭代与发展, 粗粒度的 CFI 已经无法抵御最新的 ROP 攻击[74,75], 此外, 细粒度的 CFI 也已被证明存在被绕过的可能性[76]. 上下文敏感的控制流完整性 (context-sensitive CFI, CCFI) 是有望解决这一问题的方法. CCFI 利用进程执行的历史信息作为跳转合法性的参考依据之一, 并结合 CFG 对进程的执行状态进行验证, 提高 CFI 的防御能力.

4.1 控制流弯曲

上下文无关的 CFI 往往只关注间接跳转目标地址的合法性, 而不关注跳转顺序的合法性, 粗粒度的 CFI 对间接转移目标地址类别的划分则更为宽松, 对同一目标集合中的目标地址的跳转顺序不做区分, 这便给了攻击者可乘之机. Göktas 等人[77]在 Overcoming CFI 攻击方法中构造了两种 gadget, 绕过了粗粒度 CFI 的检查机制. 这两种 gadget 虽然满足 CFI 机制定义的合法控制流转移要求, 但并不符合进程真实的运行情况, 其本质原因就在于粗粒度 CFI 无法对同一目标集合中的跳转顺序进行进一步区分. Burow 等人[12]将这种 CFI 方案中最小可区分的一组间接转移目标定义为等价类 (equivalence class, EC). 并指出 CFI 方案应该寻求能够减小平均和最大 EC 规模的 CFI 方案来提高安全性, 这是因为较大规模的 EC 包含更多的间接转移目标, 因此更容易受到攻击.

Carlini 等人[78]将这类针对等价类的攻击方式归纳为控制流弯曲 (control-flow bending, CFB), CFB 定义为每次控制流传输都在有效 CFG 范围内的非控制数据攻击, 允许攻击者改变应用进程的控制流, 但不会导致控制流偏离 CFG. 图 6 展示了控制流弯曲的典型示例, 函数 A 和函数 C 都包含对函数 B 的调用, 在正常执行情况下, 函数 A 首先在边沿 1 上执行对函数 B 的调用, 函数 B 返回边沿 2; 函数 C 在边沿 3 上执行对函数 B 的调用, 函数 B 返回边沿 4. 控制流弯曲攻击可以使函数 A 调用 B 后沿边沿 4 返回, 或者使函数 C 调用 B 后沿边沿 2 返回. 这种攻击往往是合乎 CFI 规则的, 即使细粒度的 CFI 也难以发现这种攻击方式.

作者通过实验成功攻击了具有较高平均间接目标减少量 (average indirect target reduction, AIR)[28]和较少 gadget 数量的 CFI 方案, 证明了现有评估指标无法较好评估 CFI 安全性. 作者指出 CFI 方案部署影子堆栈的必要性, 进一步通过对 6 个软件的攻击实验, 评估了细粒度 CFI 方案的防御效果. 实验结果表明, 部署了细粒度 CFI 的 6 个软件中有 5 个依然受到了非控制数据攻击, 这种攻击只进行不影响进程控制流的数据篡改, 使得修改后的间接跳转目标仍在 CFI 方案定义的 EC 中, 因此不会被检测到, 这进一步证明了减小 EC 最大规模对保护控制流安全

的重要性. CCFI 利用历史执行路径或调用点的特异性等上下文信息对间接跳转目标地址进行进一步的细分, 可以有效减小 EC 的规模, 这给防御控制流弯曲提供了新的思路.

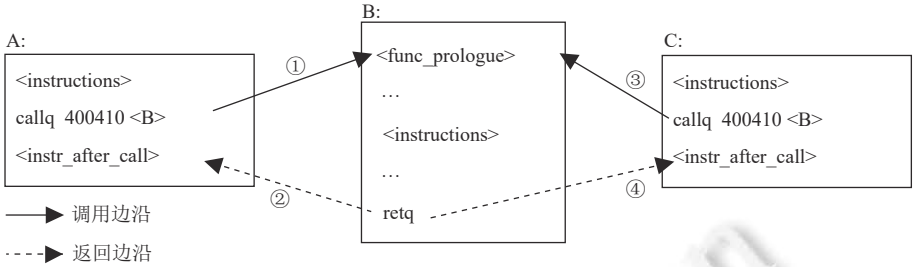


图 6 控制流弯曲举例[78]

4.2 硬件支持的 CCFI

CCFI 在提出时被认为不切实际而未引起人们关注, 因为其实现需要监控进程的执行历史路径或其他信息, 这会带来较大的运行开销. 随着计算机硬件的迭代发展, 商用处理器中集成了越来越多的进程执行信息获取机制, 例如 Intel 处理器的 LBR 和 PT 等, 这些硬件功能提供了有关直接和间接分支的丰富上下文信息, 从而使 CCFI 的实际应用成为了可能.

van de Veen 等人[15]指出实际应用 CCFI 所面临的 3 大挑战, 即高效的路径监控、路径分析和路径验证, 并利用硬件 LBR 机制实现了第一个可以实际部署的二进制级别的 CCFI 解决方案 PathArmor. PathArmor 由内核模块、路径分析模块和动态检测模块组成, 其中内核模块通过 LBR 对多线程程序中的各个线程的间接跳转进行缓存记录, 实现高效的路径监控, 同时在安全性敏感的系统调用之后将会触发路径验证步骤, 并使用路径缓存存储先前验证过路径的哈希值, 来提高验证效率; 路径分析模块从内核模块获取路径信息并且执行静态分析, 重建目标程序的 CFG, 通过一定范围内的上下文敏感静态分析得到执行路径, 确保可扩展的按需路径分析, 有效地降低了运行开销, 消除路径爆炸问题, 路径验证过程则通过对 CFG 执行深度优先搜索来寻找具有与 LBR 记录相同顺序的边的路径; 动态检测模块的主要功能是收集库文件和目标程序的地址偏移量并传递给路径分析模块, 重写目标进程的所有库函数, 插入代码片段确保库函数在执行之前首先向内核模块的 LBR 接口发送禁用请求, 并在执行结束后重新启用. 此外, 动态检测模块设置了一个与内核模块的通信通道来启用路径监控功能. PathArmor 解决了 CCFI 所有的基本挑战, 并且经实验证明具有比以往上下文无关的 CFI 方案更好的性能.

PITTYPAT[46]通过维护一个影子执行/分析过程来强制执行路径敏感的 CFI. 该影子执行/分析过程仅检查与控制相关的数据, 同时与受监视的进程同时运行. PITTYPAT 由驱动模块和分析模块两部分组成, 基于英特尔的 PT 硬件机制实现. 当程序执行时, 驱动模块从 PT 的数据包中获取控制流转移的目标地址, 分析模块根据这些目标地址重建执行路径, 并执行路径敏感点分析. 一旦受监视进程试图将控制权转移到非法分支, 将引发错误.

由于 LBR、PT 等硬件机制只能在内核模式下访问, 因此基于此类机制的 CFI 方案需要对内核进行更改, 这不仅增加了设计的复杂性, 也增加了性能开销. 针对这一问题, Khandaker 等人[48]提出了调用点敏感的 CFI 方案 CFI-LB, 该方法使用间接调用点作为上下文, 且每个调用点具有自适应上下文敏感性, 允许确定自己的敏感度级别来平衡安全性和性能, 并通过静态和动态分析等方法生成多尺度的 CFG, 并将其应用于线上和离线控制转移验证, 相较于基于执行路径的 CFI 方案开销更小. 此外, 为了避免受到其他良性但易受攻击的线程的操纵, CFI-LB 使用 Intel 处理器的 TSX 机制实现了原子性.

由于一个间接转移指令对应的传入执行路径数量有限, PathArmor、PITTYPAT 等路径敏感的 CFI 方案难以分解大规模的 EC. 针对这一问题, Khandaker 等人[47]提出了起源敏感的 CFI (origin-sensitive CFI, OS-CFI), 将间接控制转移指令调用的代码指针的起源作为上下文, 并以此约束间接转移的目标, 相较于路径敏感的实现方案, 可以将 EC 细分至更小的规模. 对于虚拟调用, OS-CFI 存储了以对象指针和对象的创建位置作为键值对的元数据, 因此可以很容易地检测到任何破坏对象指针的 COOP 攻击, 从而实现对象类型完整性[79]. 此外, OS-CFI 使用 Intel 处

理器的 MPX 机制存储和检索指针源, 使用 TSX 机制来实现原子性. 实验结果表明, OS-CFI 可以将 SPEC CPU2006 基准测试中的最大 EC 的规模从 168 减小到 2.

CCFI 利用进程的上下文语义信息, 可以有效减小 EC 的大小, 理论上增强了 CFI 的防御能力, 而在实际应用中仍然面临一些问题. 安全性方面, 如何在有限的执行路径中找到合适的上下文, 从而有效地减小 EC 的尺寸, 是 CCFI 方案需要解决的关键问题. 性能方面, 现有的 CCFI 方案均使用了 LBR、PT 等硬件机制实现上下文信息的获取, 然而相较于上下文无关的 CFI 方案, CCFI 的运行开销依然较大, 执行效率需要进一步提高. 从整个 CFI 技术发展的过程来看, 为了达到更好的安全性和性能, 未来 CCFI 技术的发展离不开底层处理器架构的支持.

5 CFI 的评价方法

为了对比和评价不同 CFI 机制的优劣, 研究人员也从不同角度提出了评价方法. 当前研究界主要关注 CFI 机制两个方面的指标: CFI 机制的安全性指标和性能指标. 其中安全性指标衡量了 CFI 方案对控制流劫持攻击的防御性能, 而性能指标衡量了 CFI 方案的运行开销. 除此之外, 还有一些针对 CFI 方案的兼容性等其他方面的度量研究. 本节对现有 CFI 方案的评价研究进行了详细的介绍, 表 2 列举了现有的评价研究及其评价角度.

表 2 CFI 评价方法总结

文献	发表年份	评价方法	评价角度
AIR[28]	2013	$AIR=\frac{1}{n}\sum_{j=1}^{n}\left(1-\frac{\lvert T_j\rvert}{S}\right)$	安全性
AIA[80]	2016	$AIA=\frac{1}{n}\sum_{i=1}^{n}\lvert T_i\rvert$	安全性
QS_{CFI}[12]	2017	$QS_{\mathrm{CFI}}=EC\times\frac{1}{LC}$	安全性
QS_{CFI}'[48]	2019	$QS_{\mathrm{CFI}}'=AVG_{EC}\times LC$	安全性
CONFIRM[81]	2019	由包含多种代码特性和编码习惯的CFI测试程序组成的测试套件进行评估	适用性、兼容性
LLVM-CFI[82]	2019	使用静态源代码分析框架精确地建模CFI策略, 并使用统一的方法评估	CFI方案的安全性等多角度的评估
CScan/CBench[83]	2020	CFI实际边界测量工具CScan和CFI有效性测试套件CBench相结合进行评估	衡量CFI方案的实际安全与理论安全之间的差距

5.1 CFI 的性能评价方法

CFI 是一种运行时防御手段, 对控制流的转移进行实时的监控, 其运行开销往往不可忽略. 性能较差的 CFI 方案会影响被保护程序的正常运行, 无法部署在实际应用中, 因此, CFI 方案的性能受到研究人员的广泛关注. 当前科研人员通常使用 SPEC CPU benchmark 评估 CFI 方案的运行开销. SPEC CPU benchmark 是标准性能评估机构 SPEC 推出的行业标准化的 CPU 测试基准套件, 专门用于评估 CPU 的性能, 被广泛应用于工业界和学术界. 为了保持基准数据的真实性、公平性和相关性, SPEC CPU 从实际应用程序中提取基准, 而不是使用人工合成的方式制作. SPEC 先后推出了 SPEC CPU2000[84]、SPEC CPU2006[85]以及 SPEC CPU2017[86]版本, 不断增加测试基准程序数量、提高易用性. 最新的 SPEC CPU2017 分为 4 个套件, 包括 43 个基准测试, 并对 CPU 的整点运算性能和浮点运算性能分别进行测试.

本文统计的 CFI 方案中, 超过 2/3 的方案使用 SPEC CPU 基准来评估性能开销. 其余方案没有使用 SPEC CPU 基准, 这是由于这些方案主要针对特定的应用场景, 包括用于虚拟机的控制流保护[16,87]、用于手机的控制流保护[26,88]、用于嵌入式系统的控制流保护[34,89−94]以及操作系统的控制流保护[95], 这些方案采用了特定领域的专用测试套件[96−99]. 此外, 一些 CFI 方案中也使用 SPEC CPU 与其他测试套件[100−102]结合来评估性能开销. 表 3 列举了现有 CFI 机制文献中所报告的性能开销及其所使用的测试套件, “—”表示没有对应的实验结果.

表 3 CFI 机制的运行开销

方法	测试套件	平均开销(x86)	最大开销(x86)	平均开销(x86-64)	最大开销(x86-64)
Origin CFI[11]	SPEC CPU2000[84]	15	46	—	—
CCFIR[27]	SPEC CPU2000[84]	3.6	8.6	—	—
O-CFI[30]	SPEC CPU2000[84]	4.7	11	—	—
BinCFI[28]	SPEC CPU2006[85]	8.54	45	—	—
Lockdown[31]	SPEC CPU2006[85]	19.09	150	—	—
μCFI[40]	SPEC CPU2006[85]	—	—	7.88	49.67
MCFI[18]	SPEC CPU2006[85]	5	10	5	12
ROPecker[37]	SPEC CPU2006[85]	2.6	15	—	—
πCFI[23]	SPEC CPU2006[85]	—	—	3.2	11.4
PT-CFI[39]	SPEC CPU2006[85]	20	65	—	—
PathArmor[15]	SPEC CPU2006[85]	—	—	8.5	—
TypeArmor[32]	SPEC CPU2006[85]	—	—	2.4	9.8
PITTYPAT[46]	SPEC CPU2006[85]	—	—	12.73	47.3
OS-CFI[47]	SPEC CPU2006[85]			7.6	15
CFI-LB[48]	SPEC CPU2006[85]	—	—	4.8	8.4
BCI-CFI[49]	SPEC CPU2006[85]	19.67	31.2	—	—
ABCFI[43]	SPEC CPU2006[85]	—	—	0.55	7
IBV-CFI[25]	SPEC CPU2017[86]	—	—	1.42	6.64
kBouncer[36]	Wine[96]	—	—	1	4
HCIC[41]	RIPE[103]	0.95	1.57	0.95	1.57
CFIGuard[38]	RIPE[103]	2.9	5.6	—	—
CFIMon[35]	Exim[104]Memcached[105]	6.1	8.4	—	—

5.2 CFI 的安全性评价方法

安全性是衡量 CFI 防御效果的最重要指标之一, 学界也针对 CFI 的安全性评价提出了多种度量方法, 包括评价指标、评估框架及评估套件. 然而, 由于各 CFI 方案的实现原理、应用场景差异较大, 到目前为止, 尚未形成统一的度量标准.

Zhang 等人[28]提出使用 *AIR* 评估 CFI 方案的安全性. *AIR* 量化了使用 CFI 方案减少的间接控制流 (indirect control flow, ICF) 目标的比例, 定义如下:

$$AIR = \frac{1}{n}\sum_{j=1}^{n}\left(1-\frac{|T_j|}{S}\right) \tag{1}$$

其中, $i_1,\ldots,i_n$ 是程序中所有的 ICF 传输, S 是一个未被 CFI 保护的程序中所有的 ICF 目标, T_j 是使用 CFI 方案之后剩余的 ICF 目标集合, 则 AIR 定义为 n 个 ICF 传输中间接转移目标减少的百分比的平均值. 则 AIR 的值越大, 说明对应 CFI 方案的安全性越好.

Ge 等人[80]指出 *AIR* 不能直观地描述 CFI 方案与 CFG 的近似程度, 并提出平均间接目标允许量 (average indirect target allowed, *AIA*), 定义如下:

$$AIA = \frac{1}{n}\sum_{i=1}^{n}|T_i| \tag{2}$$

其中, $i_1,\ldots,i_n$ 是程序中所有的 ICF 传输, T_i 分别是其允许传输的目标集合. *AIA* 可以衡量 CFI 方案的严格程度, 然而只能用于比较同一程序上的 CFI 方案, 局限性较大.

AIR 指标在 CFI 方案的安全性评价方面得到了广泛的应用, 但是也遭到了一些研究人员的质疑[19,77], 一方面, 由于 *AIR* 公式中的 S 远大于 T_j, 使得绝大部分 CFI 都可以达到 99% 的 *AIR* 值, 这表明 *AIR* 不能较好地评估不同

CFI 方案之间安全性的差异; 另一方面, 即使间接转移目标大量减少, 进程依然可能遭到类似 CFB 的攻击方式, 体现了 *AIR* 指标的局限性.

Burow 等人[12]提出了一个等式用于量化 CFI 方案的安全性, 等式的形式如下:

$$QS_{\mathrm{CFI}} = EC \times \frac{1}{LC} \tag{3}$$

其中, EC 是等价类的总数量, LC 是等价类的最大规模, QS_{CFI} 为安全性的量化数值. 等价类的数量增加以及最大等价类规模的减小会使得安全性数值增大. 这是由于等价类的数量增加, 意味着每个等价类的规模减小, 从而为攻击者提供更少的攻击面; 控制 LC 则是为了控制离群值, 使得等价类整体规模趋于小型化. 等价类规模被证明与 CFI 方案的安全性密切相关, 也使得减少等价类最大规模成为科研人员提高 CFI 防御性能的目标之一.

Khandaker 等人[48]指出公式 (3) 不适用于对 CCFI 方案进行评估, 这是由于不同 CCFI 方案可能使用不同的上下文, 从而导致 CFG 中 EC 的数量不同. 此外, EC 的数量可以成倍增加, 而 LC 的变化速度要慢得多. 为了解决这个问题, Khandaker 等人提出了一个改进的等式, 并建议用该等式评估所有 CFI 方案的安全性, 等式的形式如下:

$$QS_{\mathrm{CFI}}' = AVG_{\mathrm{EC}} \times LC \tag{4}$$

其中, AVG_{EC} 是所有 EC 的平均规模, LC 代表等价类的最大规模, QS_{CFI}' 为安全性的量化数值, 且 QS_{CFI}' 越大, 安全性越差.

不同 CFI 方案实现的底层硬件、操作系统以及基准选择的不同, 给统一评估 CFI 方案的安全性带来了较大的难度, 使用单一的评价指标难以全面评估 CFI 的防御性能. 针对这一问题, Muntean 等人[82]提出一个静态源代码分析框架 LLVM-CFI, 在同一环境对现有 CFI 策略进行建模和模拟. 作者提出了 4 个基于 LLVM-CFI 的新指标, 进行针对 CFI 策略的多方面分析, 从而实现了 CFI 方案在相同运行环境的统一评估.

RIPE 套件[103]也被许多研究人员广泛用于评估 CFI 的安全性和精度. RIPE 套件包含 850 个缓冲区溢出攻击形式, 包括针对堆栈、数据段的代码注入攻击、Return-into-libc 攻击、ROP 攻击等. 它旨在提供一种标准的方法来量化一般防御机制的安全覆盖范围.

5.3 CFI 的其他评价方法

除了性能和安全性之外, 一些研究人员着眼于研究如何评价 CFI 方案的实际应用效果. Farkhani 等人[106]从安全性和实用性两个角度评估了基于运行时类型检查 (runtime type checking, RTC) 的 CFI 机制的有效性. 该方案构建了一种特殊的 ROP 攻击 Typed ROP (TROP), 这种 ROP 攻击需要保证篡改的函数指针与函数类型一致, 因此可以绕过 RTC. 作者统计了多个应用程序中可能出现的无效目标函数和无效间接调用的数量, 发现仅在 Nginx 中就有 3 512 个指针指向 319 个无效目标函数, 这是由于这些目标函数共享相同的类型签名. 这表明虽然直接满足 TROP 攻击条件的情况较少, 但是攻击者可以通过调用这些具有相同签名的无效目标函数来完成 TROP 攻击. 研究结果表明, 虽然 RTC 是一种实用的防御手段, 但它本身不足以防御控制流劫持攻击.

CFI 机制部署到实际应用程序时, 会出现一些兼容性问题, 由于不兼容而无法得到 CFI 保护的应用程序面临极大的安全风险. Xu 等人[81]提出了一个测试套件 CONFIRM, 主要用于评估 CFI 方案的可用性和兼容性. CONFIRM 提供了 24 种针对各种 CFI 相关代码功能和编码习惯的测试, 并对 12 种开源 CFI 方案进行了评估. 实验结果表明, CFI 理论的应用存在很大的兼容性问题, 现有 CFI 方案对一部分代码或软件无法完全支持, 以至于性能弱化而无法防御普遍的控制流劫持攻击. 最先进的 CFI 解决方案与用于保护大型商业软件系统所需的 CFI 相关代码功能的兼容性只有约 53%. 多线程、自定义内存管理以及与各种形式的运行时代码生成相关的兼容性和安全性限制是部署 CFI 技术的最大障碍.

Li 等人[83]提出了一种测量 CFI 实际安全性与理论安全性差距的解决方案, 该方案使用一个轻量级的通用工具 CScan 来精确测量 CFI 的间接转移指令在运行时的有效跳转目标, 并将其定义为 CFI 的实际边界, 其中包含了 CFI 声明的理论边界和非预期的跳转目标; 为了进一步评估非预期的跳转目标是否真实, 作者提出了由 23 个易受攻击的 C/C++程序组成的测试套件 CBench, CBench 首先应用目标 CFI 机制保护这些易受攻击的程序, 并使用设计好的典型攻击来验证目标 CFI 机制的有效性. 作者使用 CScan 和 CBench 评估了 12 种开源 CFI 机制, 并指出其中 10 种机制未达到其声称的理论安全效果.

6 总结与展望

为了抵御控制流劫持攻击, 保护系统安全, 针对 CFI 的研究受到越来越多科研人员的关注. 本文提供了针对 CFI 的综述. 具体来说, 本文将现有 CFI 方法分为上下文无关的 CFI 和上下文敏感的 CFI, 并从实现方案和评估方法两方面对该领域进行了详细全面的综述.

尽管目前已经有许多研究围绕 CFI 进行展开, 但目前针对该领域的研究还不够完善, 很多方面都具有较大的改进空间, 研究人员也正在积极地探索与改进. 在这里, 本章对仍然存在的挑战进行列举, 以期为未来的研究方向提供参考, 从而进一步提高对控制流劫持攻击的防御能力.

(1) CFI 方案的安全性仍存在较大隐患. 粗粒度的 CFI 虽然运行开销较小, 但是已被证明难以抵御高级别的 ROP 攻击. CFB 则提出了 EC 内可能存在的攻击行为, 证实了控制流可以在 EC 内部被攻击而不被检测到, 这使得具有较大 EC 规模的细粒度 CFI 亦无法幸免. 因此, 通过结合进程执行上下文有效减小最大 EC 规模的低开销 CCFI, 是一个可能的探索方向.

(2) 尽管 CFI 已成功应用于一些大型应用程序, 然而 CFI 理论与实际应用之间仍然存在着较大的差距. 这通常是由许多基于源代码实现的 CFI 算法需要整个软件生态系统的完整源代码, 以便正确地分析应用程序控制流所导致. 此外, 商用软件范例所共有的复杂控制流, 例如 GUI 交互, 事件驱动等也为 CFI 方案的应用带来巨大的挑战[107]. 因此, 如何提高 CFI 方案与被保护二进制程序的兼容性是一项尚待解决的问题.

(3) 目前仍缺少一种针对 CFI 方案的系统评价方法. 一方面, 不同 CFI 方案实现的底层硬件设备, 操作系统以及基准测试的选择方面均有差异, 这给不同方案之间的评估造成较大的阻碍; 另一方面, 学界对如何评估 CFI 方案的安全性尚未达成一致, 缺少一个科学的评价体系来定量地评估 CFI 方案的防御能力. 因此, 亟需一套统一的、系统的评估方法来对 CFI 方案进行全面而客观的评价.

References:

[1] Situ LY, Wang LZ, Li XD, Liu Y. Buffer overflow detection techniques and tools based on application perspective. Ruan Jian Xue Bao/Journal of Software, 2019, 30(6): 1721–1741 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5491.htm [doi: 10.13328/j.cnki.jos.005491]

[2] Wang FF, Zhang T, Xu WG, Sun M. Overview of control-flow hijacking attack and defense techniques for process. Chinese Journal of Network and Information Security, 2019, 5(6): 10–20 (in Chinese with English abstract). [doi: 10.11959/j.issn.2096-109x.2019058]

[3] Microsoft. Data Execution Prevention (DEP). 2006. https://web.archive.org/web/20140911011045/http://support.microsoft.com/kb/875352/en-us

[4] Team P. PaX address space layout randomization (ASLR). 2003. http://pax.grsecurity.net/docs/aslr.txt

[5] 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 Conf. on USENIX Security Symp. San Antonio: ACM, 1998. 5. [doi: 10.5555/1267549.1267554]

[6] Ray D, Ligatti J. Defining code-injection attacks. ACM SIGPLAN Notices, 2012, 47(1): 179–190. [doi: 10.1145/2103621.2103678]

[7] 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. Alexandria: Association for Computing Machinery, 2007. 552–561. [doi: 10.1145/1315245.1315313]

[8] Roemer R, Buchanan E, Shacham H, Savage S. Return-oriented programming: Systems, languages, and applications. ACM Trans. on Information and System Security, 2012, 15(1): 2. [doi: 10.1145/2133375.2133377]

[9] Bletsch T, Jiang XX, Freeh VW, Liang ZK. Jump-oriented programming: A new class of code-reuse attack. In: Proc. of the 6th ACM Symp. on Information, Computer and Communications Security. Hong Kong: Association for Computing Machinery, 2011. 30–40. [doi: 10.1145/1966913.1966919]

[10] Snow KZ, Monrose F, Davi L, Dmitrienko A, Liebchen C, Sadeghi AR. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: Proc. of the 2013 IEEE Symp. on Security and Privacy. Berkeley: IEEE, 2013. 574–588. [doi: 10.1109/SP.2013.45]

[11] Abadi M, Budiu M, Erlingsson Ú, Ligatti J. Control-flow integrity. In: Proc. of the 12th ACM Conf. on Computer and Communications Security. Alexandria: ACM, 2005. 340–353. [doi: 10.1145/1102120.1102165]

[12] Burow N, Carr SA, Nash J, Larsen P, Franz M, Brunthaler S, Payer M. Control-flow integrity: Precision, security, and performance.

ACM Computing Surveys, 2018, 50(1): 16. [doi: 10.1145/3054924]

[13] de Clercq R, Verbauwhede I. A survey of hardware-based control flow integrity (CFI). arXiv:1706.07257, 2017.

[14] Wu CG, Li JJ. Evolution of control flow integrity. China Education Network, 2016, (4): 52−55 (in Chinese with English abstract). [doi: 10.3969/j.issn.1672-9781.2016.04.031]

[15] van der Veen V, Andriesse D, Göktaş E, Gras B, Sambuc L, Slowinska A, Bos H, Giuffrida C. Practical context-sensitive CFI. In: Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security. Denver: ACM, 2015. 927–940. [doi: 10.1145/2810103.2813673]

[16] Wang Z, Jiang XX. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In: Proc. of the 2010 IEEE Symp. on Security and Privacy. Oakland: IEEE, 2010. 380–395. [doi: 10.1109/SP.2010.30]

[17] Niu B, Tan G. Monitor integrity protection with space efficiency and separate compilation. In: Proc. of the 2013 ACM SIGSAC Conf. on Computer & Communications Security. Berlin: ACM, 2013. 199–210. [doi: 10.1145/2508859.2516649]

[18] Niu B, Tan G. Modular control-flow integrity. In: Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM, 2014. 577–587. [doi: 10.1145/2594291.2594295]

[19] Tice C, Roeder T, Collingbourne P, Checkoway S, Erlingsson Ú, Lozano L, Pike G. Enforcing forward-edge control-flow integrity in GCC & LLVM. In: Proc. of the 23rd USENIX Security Symp. San Diego: USENIX, 2014. 941–955.

[20] Criswell J, Dautenhahn N, Adve V. KCoFI: Complete control-flow integrity for commodity operating system kernels. In: Proc. of the 2014 IEEE Symp. on Security and Privacy. Berkeley: IEEE, 2014. 292–307. [doi: 10.1109/SP.2014.26]

[21] Niu B, Tan G. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In: Proc. of the 2014 ACM SIGSAC Conf. on Computer and Communications Security. Scottsdale: ACM, 2014. 1317–1328. [doi: 10.1145/2660267.2660281]

[22] Zhang C, Niknami M, Chen KZ, Song CY, Chen ZF, Song D. JITScope: Protecting Web users from control-flow hijacking attacks. In: Proc. of the 2015 IEEE Conf. on Computer Communications (INFOCOM). Hong Kong: IEEE, 2015. 567 –575. [doi: 10.1109/INFOCOM.2015.7218424]

[23] Niu B, Tan G. Per-input control-flow integrity. In: Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security. Denver: ACM, 2015. 914–926. [doi: 10.1145/2810103.2813644]

[24] Li JK, Tong XM, Zhang FW, Ma JF. Fine-CFI: Fine-grained control-flow integrity for operating system kernels. IEEE Trans. on Information Forensics and Security, 2018, 13(6): 1535–1550. [doi: 10.1109/TIFS.2018.2797932]

[25] Jang H, Park MC, Lee DH. IBV-CFI: Efficient fine-grained control-flow integrity preserving CFG precision. Computers & Security, 2020, 94: 101828.

[26] Davi L, Dmitrienko A, Egele M, Fischer T, Holz T, Hund R, Nürnberger S, Sadeghi AR. MoCFI: A framework to mitigate control-flow attacks on smartphones. In: Proc. of the NDSS Symp. 2012. NDSS, 2012. 27–40.

[27] Zhang C, Wei T, Chen ZF, Duan L, Szekeres L, McCamant S, Song D, Zou W. Practical control flow integrity and randomization for binary executables. In: Proc. of the 2013 IEEE Symp. on Security and Privacy. Berkeley: IEEE, 2013. 559–573. [doi: 10.1109/SP.2013.44]

[28] Zhang MW, Sekar R. Control flow integrity for COTS binaries. In: Proc. of the 22nd USENIX Security Symp. Washington: USENIX, 2013. 337–352.

[29] Wang MH, Yin H, Bhaskar AV, Su PR, Feng DG. Binary code continent: Finer-grained control flow integrity for stripped binaries. In: Proc. of the 31st Annual Computer Security Applications Conf. Los Angeles: ACM, 2015. 331–340. [doi: 10.1145/2818000.2818017]

[30] Mohan V, Larsen P, Brunthaler S, Hamlen KW, Franz M. Opaque control-flow integrity. In: Proc. of the NDSS Symp. 2015. San Dicgo: NDSS, 2015. 27–30.

[31] Payer M, Barresi A, Gross TR. Fine-grained control-flow integrity through binary hardening. In: Proc. of the 12th Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment. Milan: Springer, 2015. 144–164.

[32] van der Veen V, Göktas E, Contag M, Pawoloski A, Chen X, Rawat S, Bos H, Holz T, Athanasopoulos E, Giuffrida C. A tough call: Mitigating advanced code-reuse attacks at the binary level. In: Proc. of the 2016 IEEE Symp. on Security and Privacy (SP). San Jose: IEEE, 2016. 934–953. [doi: 10.1109/SP.2016.60]

[33] Muntean P, Fischer M, Tan G, Lin ZQ, Grossklags J, Eckert C. τCFI: Type-assisted control flow integrity for x86-64 binaries. In: Proc. of the 21st Int'l Symp. on Research in Attacks, Intrusions, and Defenses. Heraklion: Springer, 2018. 423–444. [doi: 10.1007/978-3-030-00470-5_20]

[34] Walls RJ, Brown NF, Le Baron T, Shue CA, Okhravi H, Ward BC. Control-flow integrity for real-time embedded systems. In: Proc. of the 31st Euromicro Conf. on Real-time Systems (ECRTS 2019). Dagstuhl: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019. 2. [doi: 10.4230/LIPIcs.ECRTS.2019.2]

[35] Xia YB, Liu YT, Chen HB, Zang BY. CFIMon: Detecting violation of control flow integrity using performance counters. In: Proc. of

the IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN 2012). Boston: IEEE, 2012. 1–12. [doi: 10.1109/DSN.2012.6263958]

[36] Pappas V, Polychronakis M, Keromytis AD. Transparent ROP exploit mitigation using indirect branch tracing. In: Proc. of the 22nd USENIX Security Symp. Washington: USENIX, 2013. 447–462.

[37] Cheng YQ, Zhou ZW, Yu M, Ding XH, Deng RH. ROPecker: A generic and practical approach for defending against ROP attack. In: Proc. of the 2014 NDSS Symp. San Diego: NDSS, 2014. [doi: 10.14722/ndss.2014.23156]

[38] Yuan PH, Zeng QK, Ding XH. Hardware-assisted fine-grained code-reuse attack detection. In: Proc. of the 18th Int'l Symp. on Research in Attacks, Intrusions, and Defenses. Kyoto: Springer, 2015. 66–85. [doi: 10.1007/978-3-319-26362-5_4]

[39] Gu YF, Zhao QC, Zhang YQ, Lin ZQ. PT-CFI: Transparent backward-edge control flow violation detection using intel processor trace. In: Proc. of the 7th ACM on Conf. on Data and Application Security and Privacy. Scottsdale: ACM, 2017. 173–184. [doi: 10.1145/3029806.3029830]

[40] Hu H, Qian CX, Yagemann C, Chung SPH, Harris WR, Kim T, Lee W. Enforcing unique code target property for control-flow integrity. In: Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. Toronto: ACM, 2018. 1470–1486. [doi: 10.1145/3243734.3243797]

[41] Zhang JL, Qi BH, Qin Z, Qu G. HCIC: Hardware-assisted control-flow integrity checking. IEEE Internet of Things Journal, 2019, 6(1): 458–471. [doi: 10.1109/JIOT.2018.2866164]

[42] Liljestrand H, Nyman T, Wang K, Perez CC, Ekberg JE, Asokan N. PAC it up: Towards pointer integrity using ARM pointer authentication. In: Proc. of the 28th USENIX Security Symp. Santa Clara: USENIX Association, 2019. 177–194.

[43] Li JF, Chen LW, Shi G, Chen K, Meng D. ABCFI: Fast and lightweight fine-grained hardware-assisted control-flow integrity. IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems, 2020, 39(11): 3165–3176. [doi: 10.1109/TCAD.2020.3012640]

[44] Koruyeh EM, Shirazi SHA, Khasawneh KN, Song CY, Abu-Ghazaleh N. SpecCFI: Mitigating spectre attacks using CFI informed speculation. In: Proc. of the 2020 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2020. 39–53. [doi: 10.1109/SP40000.2020.00033]

[45] Feng L, Huang J, Hu J, Reddy A. FastCFI: Real-time control-flow integrity using FPGA without code instrumentation. ACM Trans. on Design Automation of Electronic Systems, 2021, 26(5): 39. [doi: 10.1145/3458471]

[46] Ding R, Qian CX, Song CY, Harris B, Kim T, Lee W. Efficient protection of path-sensitive control security. In: Proc. of the 26th USENIX Conf. on Security Symp. Vancouver: ACM, 2017. 131–148. [doi: 10.5555/3241189.3241201]

[47] Khandaker MR, Liu WQ, Naser A, Wang Z, Yang J. Origin-sensitive control flow integrity. In: Proc. of the 28th USENIX Security Symp. Santa Clara: USENIX, 2019. 195–211.

[48] Khandaker M, Naser A, Liu WQ, Wang Z, Zhou YJ, Cheng YQ. Adaptive call-site sensitive control flow integrity. In: Proc. of the 2019 IEEE European Symp. on Security and Privacy (EuroS&P). Stockholm: IEEE, 2019. 95–110. [doi: 10.1109/EuroSP.2019.00017]

[49] Wang Y, Li QB, Chen ZF, Zhang P, Zhang GM, Shi ZH. BCI-CFI: A context-sensitive control-flow integrity method based on branch correlation integrity. Information and Software Technology, 2021, 136: 106572. [doi: 10.1016/j.infsof.2021.106572]

[50] Abadi M, Budiu M, Erlingsson Ú, Ligatti J. Control-flow integrity principles, implementations, and applications. ACM Trans. on Information and System Security, 2009, 13(1): 4. [doi: 10.1145/1609956.1609960]

[51] Chiueh TC, Hsu FH. RAD: A compile-time solution to buffer overflow attacks. In: Proc. of the 21st Int'l Conf. on Distributed Computing Systems. Mesa: IEEE, 2001. 409–417. [doi: 10.1109/ICDSC.2001.918971]

[52] The LLVM compiler infrastructure. 2010. http://llvm.org

[53] Conti M, Crane S, Davi L, Franz M, Larsen P, Negro M, Liebchen C, Qunaibit M, Sadeghi AR. Losing control: On the effectiveness of control-flow integrity under stack attacks. In: Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security. Denver: ACM, 2015. 952–963. [doi: 10.1145/2810103.2813671]

[54] Criswell J, Geoffray N, Adve VS. Memory safety for low-level software/hardware interactions. In: Proc. of the 18th USENIX Security Symp. Montreal: USENIX, 2009. 83–100.

[55] Criswell J, Lenharth A, Dhurjati D, Adve V. Secure virtual architecture: A safe execution environment for commodity operating systems. In: Proc. of the 21st ACM SIGOPS Symp. on Operating Systems Principles. Washington: ACM, 2007. 351–366. [doi: 10.1145/1294261.1294295]

[56] Chen ZF, Li QB, Zhang P, Wang Y. Kernel code reuse attack detection technique for linux. Ruan Jian Xue Bao/Journal of Software, 2017, 28(7): 1732–1745 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5058.htm [doi: 10.13328/j.cnki.jos.005058]

[57] Blazakis D. Interpreter exploitation. In: Proc. of the 4th USENIX Workshop on Offensive Technologies (WOOT 2010). Washington: USENIX, 2010.

[58] Yee B, Sehr D, Dardyk G, Chen JB, Muth R, Ormandy T, Okasaka S, Narula N, Fullagar N. Native client: A sandbox for portable, untrusted x86 native code. In: Proc. of the 30th IEEE Symp. on Security and Privacy. Oakland: IEEE, 2009. 79–93. [doi: 10.1109/SP.2009.25]

[59] Ansel J, Marchenko P, Erlingsson U, Taylor E, Chen B, Schuff DL, Sehr D, Biffle CL, Yee B. Language-independent sandboxing of just-in-time compilation and self-modifying code. In: Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation. San Jose: ACM, 2011. 355–366. [doi: 10.1145/1993498.1993540]

[60] Wahbe R, Lucco S, Anderson TE, Graham SL. Efficient software-based fault isolation. In: Proc. of the 14th ACM Symp. on Operating Systems Principles. Asheville: ACM, 1994. 203–216. [doi: 10.1145/168619.168635]

[61] Erlingsson Ú, Abadi M, Vrable M, Budiu M, Necula GC. XFI: Software guards for system address spaces. In: Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation. USENIX, 2006. 75–88.

[62] Schuster F, Tendyck T, Liebchen C, Davi L, Sadeghi AR, Holz T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In: Proc. of the 2015 IEEE Symp. on Security and Privacy. San Jose: IEEE, 2015. 745–762. [doi: 10.1109/SP.2015.51]

[63] Li WW, Ma Y, Wang JJ, Gao WY, Yang QS, Li MS. ROP attack detection approach based on hardware branch information. Ruan Jian Xue Bao/Journal of Software, 2020, 31(11): 3588–3602 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5829.htm [doi: 10.13328/j.cnki.jos.005829]

[64] Wang XR, Liu YT, Chen HB. Transparent protection of kernel module against ROP with intel processor trace. Ruan Jian Xue Bao/Journal of Software, 2018, 29(5): 1333−1347 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5496.htm [doi: 10.13328/j.cnki.jos.005496]

[65] Qiu PF, Lyu YQ, Zhai D, Wang DS, Zhang JL, Wang XW, Qu G. Physical unclonable functions-based linear encryption against code reuse attacks. In: Proc. of the 53rd ACM/EDAC/IEEE Design Automation Conf. (DAC). Austin: IEEE, 2016. 75. [doi: 10.1145/2897937.2898061]

[66] Mashtizadeh AJ, Bittau A, Boneh D, Mazières D. CCFI: Cryptographically enforced control flow integrity. In: Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security. Denver: ACM, 2015. 941–951. [doi: 10.1145/2810103.2813676]

[67] Qiu PF, Lyu YQ, Zhang JL, Wang DS, Qu G. Control flow integrity based on lightweight encryption architecture. IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems, 2018, 37(7): 1358–1369. [doi: 10.1109/TCAD.2017.2748000]

[68] 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 ACM/EDAC/IEEE Design Automation Conf. (DAC). San Francisco: IEEE, 2014. 1–6. [doi: 10.1109/DAC.2014.6881460]

[69] Davi L, Hanreich M, Paul D, Sadeghi AR, Koeberl P, Sullivan D, Arias O, Jin Y. HAFIX: Hardware-assisted flow integrity extension. In: Proc. of the 52nd ACM/EDAC/IEEE Design Automation Conf. (DAC). San Francisco: IEEE, 2015. 1–6. [doi: 10.1145/2744769.2744847]

[70] Christoulakis N, Christou G, Athanasopoulos E, Ioannidis S. HCFI: Hardware-enforced control-flow integrity. In: Proc. of the 6th ACM Conf. on Data and Application Security and Privacy. New Orleans: ACM, 2016. 38–49. [doi: 10.1145/2857705.2857722]

[71] Muench M, Pagani F, Shoshitaishvili Y, Kruegel C, Vigna G, Balzarotti D. Taming transactions: Towards hardware-assisted control flow integrity using transactional memory. In: Proc. of the 19th Int'l Symp. on Research in Attacks, Intrusions, and Defenses. Paris: Springer, 2016. 24–48. [doi: 10.1007/978-3-319-45719-2_2]

[72] INTEL. Intel® 64 and ia-32 architectures software developer's manual. Volume 3B: System Programming Guide, Part 2, 2021.

[73] Ge XY, Cui WD, Jaeger T. GRIFFIN: Guarding control flows using intel processor trace. ACM SIGPLAN Notices, 2017, 52(4): 585–598. [doi: 10.1145/3093336.3037716]

[74] Carlini N, Wagner D. ROP is still dangerous: Breaking modern defenses. In: Proc. of the 23rd USENIX Security Symp. San Diego: USENIX, 2014. 385–399.

[75] Davi L, Sadeghi AR, Lehmann D, Monrose F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In: Proc. of the 23rd USENIX Conf. on Security Symp. San Diego: ACM, 2014. 401–416. [doi: 10.5555/2671225.267125]

[76] Evans I, Long F, Otgonbaatar U, Shrobe H, Rinard M, Okhravi H, Sidiroglou-Douskos S. Control Jujutsu: On the weaknesses of fine-grained control flow integrity. In: Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security. Denver: ACM, 2015. 901–913. [doi: 10.1145/2810103.2813646]

[77] Göktas E, Athanasopoulos E, Bos H, Portokalidis G. Out of control: Overcoming control-flow integrity. In: Proc. of the 2014 IEEE Symp. on Security and Privacy. Berkeley: IEEE, 2014. 575–589. [doi: 10.1109/SP.2014.43]

[78] Carlini N, Barresi A, Payer M, Wagner D, Gross TR. Control-flow bending: On the effectiveness of control-flow integrity. In: Proc. of

the 24th USENIX Security Symp. Washington: USENIX, 2015. 161–176.

[79] Burow N, McKee D, Carr SA, Payer M. CFIXX: Object type integrity for C++ virtual dispatch. In: Proc. of the Symp. on Network and Distributed System Security (NDSS). San Diego: NDSS, 2018. [doi: 10.14722/ndss.2018.23279]

[80] Ge XY, Talele N, Payer M, Jaeger T. Fine-grained control-flow integrity for kernel software. In: Proc. of the 2016 IEEE European Symp. on Security and Privacy (EuroS&P). Saarbruecken: IEEE, 2016. 179–194. [doi: 10.1109/EuroSP.2016.24]

[81] Xu XY, Ghaffarinia M, Wang WH, Hamlen KW, Lin ZQ. CONFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In: Proc. of the 28th USENIX Security Symp. Santa Clara: USENIX, 2019. 1805–1821.

[82] Muntean P, Neumayer M, Lin ZQ, Tan G, Grossklags J, Eckert C. Analyzing control flow integrity with LLVM-CFI. In: Proc. of the 35th Annual Computer Security Applications Conf. San Juan: ACM, 2019. 584–597. [doi: 10.1145/3359789.3359806]

[83] Li Y, Wang MZ, Zhang C, Chen XM, Yang ST, Liu Y. Finding cracks in shields: On the security of control flow integrity mechanisms. In: Proc. of the 2020 ACM SIGSAC Conf. on Computer and Communications Security. ACM, 2020. 1821–1835. [doi: 10.1145/3372297.3417867]

[84] Henning JL. SPEC CPU2000: Measuring CPU performance in the new millennium. Computer, 2000, 33(7): 28–35. [doi: 10.1109/2.869367]

[85] Henning JL. SPEC CPU2006 benchmark descriptions. ACM SIGARCH Computer Architecture News, 2006, 34(4): 1–17. [doi: 10.1145/1186736.1186737]

[86] Bucek J, Lange KD, Kistowski JV. SPEC CPU2017: Next-generation compute benchmark. In: Proc. of the 2018 ACM/SPEC Int'l Conf. on Performance Engineering. Berlin: ACM, 2018. 41–42. [doi: 10.1145/3185768.3185771]

[87] Kwon D, Seo J, Baek S, Kim G, Ahn S, Paek Y. VM-CFI: Control-flow integrity for virtual machine kernel using Intel PT. In: Proc. of the 18th Int'l Conf. on Computational Science and Its Applications. Melbourne: Springer, 2018. 127–137. [doi: 10.1007/978-3-319-95174-4_10]

[88] Pewny J, Holz T. Control-flow restrictor: Compiler-based CFI for iOS. In: Proc. of the 29th Annual Computer Security Applications Conf. Louisiana, 2013. 309–318. [doi: 10.1145/2523649.2523674]

[89] Abbasi A, Holz T, Zambon E, Etalle S. ECFI: Asynchronous control flow integrity for programmable logic controllers. In: Proc. of the 33rd Annual Computer Security Applications Conf. Orlando: ACM, 2017. 437–448. [doi: 10.1145/3134600.3134618]

[90] Abera T, Asokan N, Davi L, Ekberg JE, Nyman T, Paverd A, Sadeghi AR, Tsudik G. C-FLAT: Control-flow attestation for embedded systems software. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. Vienna: ACM, 2016. 743–754. [doi: 10.1145/2976749.2978358]

[91] Adepu S, Brasser F, Garcia L, Rodler M, Davi L, Sadeghi AR, Zonouz S. Control behavior integrity for distributed cyber-physical systems. In: Proc. of the 11th ACM/IEEE Int'l Conf. on Cyber-physical Systems (ICCPS). Sydney: IEEE, 2020. 30–40. [doi: 10.1109/ICCPS48487.2020.00011]

[92] Das S, Zhang W, Liu Y. A fine-grained control flow integrity approach against runtime memory attacks for embedded systems. IEEE Trans. on Very Large Scale Integration (VLSI) Systems, 2016, 24(11): 3193–3207. [doi: 10.1109/TVLSI.2016.2548561]

[93] Werner M, Unterluggauer T, Schaffenrath D, Mangard S. Sponge-based control-flow protection for iot devices. In: Proc. of the 2018 IEEE European Symp. on Security and Privacy (EuroS&P). London: IEEE, 2018. 214–226. [doi: 10.1109/EuroSP.2018.00023]

[94] Nyman T, Ekberg JE, Davi L, Asokan N. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In: Proc. of the 20th Int'l Symp. on Research in Attacks, Intrusions, and Defenses. Atlanta: Springer, 2017. 259–284. [doi: 10.1007/978-3-319-66332-6_12]

[95] Kemerlis VP, Portokalidis G, Keromytis AD. kGuard: Lightweight kernel protection against return-to-user attacks. In: Proc. of the 21st USENIX Conf. on Security Symp. Bellevue: ACM, 2012. 39. [doi: 10.5555/2362793.2362832]

[96] The Wine Committee. Wine. 2021. http://www.winehq.org

[97] de Melo AC. Performance counters on linux. In: Proc. of the Linux Plumbers Conf. Portland, 2009.

[98] Coker R. Disk performance benchmark tool–Bonnie. 2016. https://www.coker.com.au/bonnie++

[99] Pozo R, Miller B. SciMark 2.0. 2016. http://math.nist.gov/scimark2

[100] Albayraktaroglu K, Jaleel A, Wu X, Franklin M, Jacob B, Tseng CW, Yeung D. BioBench: A benchmark suite of bioinformatics applications. In: Proc. of the IEEE Int'l Symp. on Performance Analysis of Systems and Software. Austin: IEEE, 2005. 2–9. [doi: 10.1109/ISPASS.2005.1430554]

[101] Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB. MiBench: A free, commercially representative embedded benchmark suite. In: Proc. of the 4th Annual IEEE Int'l Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538). Austin: IEEE, 2001. 3–14. [doi: 10.1109/WWC.2001.990739]

[102] McCalpin JD. STREAM benchmark. 1995. http://www.cs.virginia.edu/stream/ref.html
[103] Wilander J, Nikiforakis N, Younan Y, Kamkar M, Joosen W. RIPE: Runtime intrusion prevention evaluator. In: Proc. of the 27th Annual Computer Security Applications Conf. 2011. 41–50. [doi: 10.1145/2076732.2076739]
[104] Cambridge. Exim. 2021. http://www.exim.org
[105] Fitzpatrick B. Distributed caching with memcached. Linux Journal, 2004: 124.
[106] Farkhani RM, Jafari S, Arshad S, Robertson W, Kirda E, Okhravi H. On the effectiveness of type-based control flow integrity. In: Proc. of the 34th Annual Computer Security Applications Conf. San Juan: ACM, 2018. 28–39. [doi: 10.1145/3274694.3274739]
[107] Wang WH, Xu XY, Hamlen KW. Object flow integrity. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. Dallas: ACM, 2017. 1909–1924. [doi: 10.1145/3133956.3133986]

附中文参考文献:

[1] 司徒凌云, 王林章, 李宣东, 刘杨. 基于应用视角的缓冲区溢出检测技术与工具. 软件学报, 2019, 30(6): 1721–1741. http://www.jos.org.cn/1000-9825/5491.htm [doi: 10.13328/j.cnki.jos.005491]
[2] 王丰峰, 张涛, 徐伟光, 孙蒙. 进程控制流劫持攻击与防御技术综述. 网络与信息安全学报, 2019, 5(6): 10–20. [doi: 10.11959/j.issn.2096-109x.2019058]
[14] 武成岗, 李建军. 控制流完整性的发展历程. 中国教育网络, 2016, (4): 52–55. [doi: 10.3969/j.issn.1672-9781.2016.04.031]
[56] 陈志锋, 李清宝, 张平, 王烨. 面向Linux的内核级代码复用攻击检测技术. 软件学报, 2017, 28(7): 1732–1745. http://www.jos.org.cn/1000-9825/5058.htm [doi: 10.13328/j.cnki.jos.005058]
[63] 李威威, 马越, 王俊杰, 高伟毅, 杨秋松, 李明树. 基于硬件分支信息的ROP攻击检测方法. 软件学报, 2020, 31(11): 3588–3602. http://www.jos.org.cn/1000-9825/5829.htm [doi: 10.13328/j.cnki.jos.005829]
[64] 王心然, 刘宇涛, 陈海波. 基于IPT硬件的内核模块ROP透明保护机制. 软件学报, 2018, 29(5): 1333–1347. http://www.jos.org.cn/1000-9825/5496.htm [doi: 10.13328/j.cnki.jos.005496]

张正(1993－), 男, 博士生, 主要研究领域为系统安全.

谭毓安(1972－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为 Android 安全, 深度学习及对抗, 物联网与嵌入式系统, 数据存储安全.

薛静锋(1975－), 男, 博士, 教授, 博士生导师, 主要研究领域为网络安全, 数据安全, 软件安全, 软件测试.

李元章(1978－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为信息安全, 嵌入式技术.

张静慈(1991－), 女, 博士生, 主要研究领域为网络安全.

张全新(1974－), 男, 博士, 副教授, 主要研究领域为深度学习及其对抗技术, 计算机视觉安全, 信息安全.

陈田(1995－), 男, 博士生, 主要研究领域为系统安全.