基于配件加权标记的代码重用攻击防御框架

马梦雨[1,2], 陈李维[1,2], 史 岗[1,2], 孟 丹[1,2]

[1]中国科学院信息工程研究所 北京 中国 100093
[2]中国科学院大学 网络空间安全学院 北京 中国 100049

摘要 代码重用攻击(Code Reuse Attack, CRA)目前已经成为主流的攻击方式, 能够对抗多种防御机制, 给计算机安全带来极大的威胁和挑战。本文提出一种基于配件加权标记(Gadget Weighted Tagging, GWT)的 CRAs 防御框架。首先, GWT 找到代码空间中所有可能被 CRAs 利用的配件。其次, GWT 为每个配件附加相应的权值标记, 这些权值可以根据用户需求灵活地配置。最后, GWT 在程序运行时监控配件的权值信息, 从而检测和防御 CRAs。另外, 我们结合粗粒度 CFI 的思想, 进一步提出 GWT+CFI 的设计框架, 相比基础的 GWT, GWT+CFI 能够提高识别配件开端的精确性并减少可用配件的数量。我们基于软件和硬件模拟的方案实现 GWT 和 GWT+CFI 系统, 结果表明其平均性能开销分别为 2.31%和 3.55%, 且 GWT 理论上能够防御大多数 CRAs, 特别是使用自动化工具生成配件链的 CRAs。

关键词 代码重用攻击; 配件加权标记; 控制流完整性
中图法分类号 TP309.2 DOI 号 10.19363/J.cnki.cn10-1380/tn.2018.09.07

A Framework based on Gadget Weighted Tagging (GWT) to Protect Against Code Reuse Attacks

MA Mengyu[1,2], CHEN Liwei[1,2], SHI Gang[1,2], MENG Dan[1,2]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

Abstract Code reuse attacks(CRAs) become the primary attack vector nowadays. CRAs are able to bypass a variety of security mechanisms so that CRAs pose a great challenge in the field of security research. In this paper, we propose Gadget Weighted Tagging(GWT), a flexible framework to protect against CRAs. First, we find all possible gadgets, which can be used in CRAs. Then, we attach weighted tags to these gadgets, and the weighted values are configurable as the need. At last, we monitor the weighted tag information at runtime to detect and prevent CRAs. Furthermore, combining with the rule-based CFI, GWT+CFI can precisely confirm the gadget start and greatly reduce the number of possible gadgets, compared to the baseline GWT. We implement a software and emulation-based hardware framework to support GWT and GWT+CFI. The results show that the average performance overheads of GWT and GWT+CFI are 2.31% and 3.55% respectively, and GWT can defeat the majority of CRAs, especially those generated by automated tools.

Key words code reuse attack; gadget weighted tagging; control flow integrity

1 前言

攻击者利用软件漏洞控制应用程序的行为和逻辑, 进而通过提权威胁计算机系统的安全。传统的代码注入攻击通过外部输入的方式在程序内存中存储恶意代码, 同时利用软件漏洞劫持程序控制流转向恶意代码。但是研究者提出的数据执行保护(Data Execution Prevention, DEP)能够从本质上缓解代码注入攻击, DEP 规定所有可写的内存页不可执行, 因此用户输入的恶意代码仅仅被当作数据, 即使程序指针跳转到恶意代码所在的内存页, 攻击也无法继续执行。随后, 研究者提出了代码重用攻击(Code Reuse

通讯作者: 陈李维, 博士, 助理研究员, Email: chenliwei@iie.ac.cn。
本课题得到国家自然科学基金(No. 61602469), 中国科学院信息工程研究所和信息安全国家重点实验室(No. Y7Z0411105)资助。
收稿日期: 2017-06-19; 修改日期: 2017-11-21; 定稿日期: 2018-08-20

attack, CRA), 它使用程序中原有的指令片段替代外部输入的指令, 从而绕过 DEP 的防御。根据重用指令的不同可分为返回导向编程(Return-oriented Programming, ROP)[1]和跳转导向编程(Jump-oriented Programming, JOP)[2]两种主流攻击方式。如今, CRA 逐渐取代代码注入成为主要的攻击手段。

CRA 的核心思想是重用程序中已经存在的指令, 这些指令能够完成一定的计算或赋值等功能, 然后以 ret 指令或 jmp 指令结尾构成一段指令片段, 称之为配件(Gadget), 多个配件相连构成恶意的 shellcode。攻击者通过软件漏洞劫持程序控制流转向第一个配件的地址, 系统就会自动地执行配件链完成攻击。CRA 被证明是图灵完备的[2], 且能够通过固定的模式形成自动化的攻击工具[3], 因此 CRA 具备容易使用、难以防御、图灵完备的特性。随后, 研究者提出的控制流完整性(Control Flow Integrity, CFI)[4-5]理论上能够完全防御 ROP 和 JOP 两种主流攻击方式, CFI 首先分析程序中的所有合法执行路径并生成控制流图(Control Flow Graph, CFG), 其次在 CFG 路径上添加标记信息, 最后严格按照 CFG 执行程序, 因此攻击者无法随意更改控制流, 也就打断了配件链的执行过程。但是 CFI 有两个主要的缺点, 第一, 较高的复杂性造成近百分之五十的系统开销[4]; 第二, 构建完善且精确的 CFG 几乎不可能实现。

为了解决 CFI 的两个主要问题, 研究者提出了很多实际的解决方案[6-9]。制定更为宽松的 CFI 策略来降低系统的性能开销, 首先, 通过分析普通程序的合法控制流转移, 总结出一些控制流转移规律, 例如 call 指令应该指向一个函数入口的地址等; 其次, 不依赖完整的 CFG, 而是利用这些规律识别目标程序中合法与非法的执行路径。这些宽松的 CFI 策略被称之为粗粒度 CFI, 它并没有构建程序的 CFG, 而是从中抽取控制流转移的规律来制定相应的规则, 虽然降低了系统性能开销, 但其安全保障要低于基于 CFG 的原始 CFI, 最近也有很多学者提出使用特殊配件绕过粗粒度 CFI 的方案[10-13]。

我们分析现有能够被 CRA 利用的配件, 发现配件的几个特征如下: 仅仅占据整个代码空间的很小一部分; 在普通程序中很少出现且不连续; 在 CRA 中连续出现并形成配件链。因此, 我们能够通过监控程序运行时的配件特征来区分普通和恶意程序, 如图 1 所示。

本文提出配件加权标记(Gadget Weighted Tagging, GWT)的思想, 一种灵活地检测并防御 CRAs 的防御机制。首先, 我们根据配件在 CRA 中发挥的作用, 把所有可能的配件分成三种主要类型: 功能配件, NOP 配件和普通代码; 其次, 我们为不同类型的配件分配不同的权值, 并在二进制文件中附加对应的权值标记; 最后, 在程序运行时监控各类配件权值的累加值, 当这个值超过设定的阈值时, GWT 判断为发生 CRA。除此之外, 我们还提出了 GWT 结合粗粒度 CFI 的思想, 因为我们发现粗粒度 CFI 对控制流转移的策略是对 GWT 很好的补充, 采用 GWT+CFI 可以更精准的识别配件并且完善框架的安全策略。

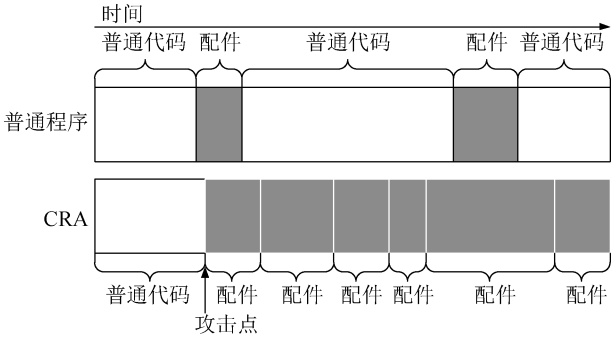

图 1 普通程序和 CRA 的配件分布规律

Figure 1 The Gadget Distribution in Normal Programs and CRAs

GWT 的实现依赖一些重要的参数, 如不同配件类型对应的权值, 用户可根据实际需求对这些参数进行调整并重新配置。同时如果用户需要添加一些新的特殊配件, 也可以在 GWT 中添加相应的配件类型并分配权值。正是因为 GWT 具备灵活的可扩展性, 所以 GWT 能够检测 CRA 的多种利用方式, 而且 GWT 的安全性也能够随着配件类型的不断完善而逐渐加强。当研究者提出一种新的 CRA 方式或配件类型时, 通常会介绍攻击的构建方法或配件的寻找算法, 所以 GWT 可以重用这些方法或算法找到程序中可能存在的配件, 完善其配件类型和安全策略。我们基于软件和硬件模拟的设计框架实现 GWT 以及 GWT+CFI 系统, 结果表明其性能开销分别为 2.31% 和 3.55%, 并且 GWT 理论上能够检测使用自动化工具 Q[3]生成配件链的 CRA, 以及使用特殊配件的新型 CRA。

这篇论文的主要贡献如下:

1) 我们提出一种防御代码重用攻击(Code Reuse Attack, CRA)的新型系统化框架: 配件加权标记(Gadget Weighted Tagging, GWT), 具备灵活可配置、高安全性、低开销、易用的特点。

2) 我们提出 GWT 结合粗粒度 CFI 的防御思想, 相比基础的 GWT, GWT+CFI 寻找配件更加精确。

3) 我们基于软件和硬件模拟的设计框架实现 GWT 和 GWT+CFI 系统, 并且评估了其安全性和性能开销。

本文内容组织如下: 第二章节介绍相关研究背景, 包括 CRA、CFI、合法配件以及威胁模型; 第三和第四章节分别阐述 GWT 以及 GWT+CFI 的基本原理和实现细节; 第五章节为实验评估, 包括 GWT 的安全性分析和性能分析; 第六章节对配件分类和 GWT 框架的灵活性进行讨论; 第七章节介绍防御 CRA 的研究现状; 第八章节对全文进行总结。

2 背景

2.1 代码重用攻击 CRA

CRA 本质就是在已有的代码空间中寻找具备一定功能的指令片段, 称之为配件(Gadget), 然后串连配件形成图灵完备的配件链执行恶意功能, 比如打开一个非法 Shell。配件通常由几条具备计算功能的普通指令加上一条间接转移指令组成, 普通指令完成恶意功能, 间接转移指令劫持程序控制流并串连多个配件。通过上述分析, CRA 的核心在于配件, 而配件的特征在于结尾的间接跳转指令, 所以本文关注三类通用的间接转移指令: call, return, indirect-jmp。除此之外, 多数 CRAs 中使用的配件还具备以下两点重要特征:

1) 稀疏的配件分布

配件都是以间接转移指令为结尾, 然而间接转移指令仅仅占整个代码空间的很小一部分, 因此 CRA 使用的配件在普通程序中的分布应该是非常稀少的。

2) 短小的指令片段

通常配件中包含的指令条数越多, 完成的功能也就越多, 但同时也不可避免的造成一些副作用, 比如寄存器的依赖冲突将会导致相关数据被覆写。因此攻击者会使用简单且指令条数较少的配件来消除这些副作用[3], 而不是为了完成更多的操作, 选择冗长且复杂的配件。实际上, 在真实的 CRA 中, 配件的指令数大约为 2~6 条[8]。

根据配件中间接转移指令的不同可以把传统 CRA 分为 ROP 攻击和 JOP 攻击。ROP 攻击依赖栈指针的返回机制完成控制流的劫持, 多个配件通过 ret 指令跳转相连。JOP 攻击相比 ROP 要复杂一些, 因为它引入一种新的配件, 称之为调度配件(dispatcher gadget), 所以 JOP 并不依赖于栈的返回机制完成攻击, 而是使用一种调度表(dispatch table)或具有连接各个配件功能的结构保存所有配件的地址, 通过调度配件劫持程序控制流, 然后驱动调度表或同等结构中的各个配件完成攻击[14]。调度配件是构造 JOP 攻击的关键, 它通常包含一些自修改跳转地址的操作, 并且以 indirect-jmp 指令结尾, 相当于程序 eip 指针的作用, 可用过程 1 简单描述调度配件的功能[14]。

过程 1. 调度配件的功能描述

PC <— *f*(*PC*);

GOTO **PC*;

2.2 控制流完整性 CFI

CFI 作为对抗 CRA 的主流防御方法之一, 自然备受研究者关注, 近年来也提出了很多实际的实施方案[5-9, 15]。本文把这些 CFI 方案大体分类两类: 基于 CFG 的 CFI 和基于策略的 CFI。

2.2.1 基于 CFG 的 CFI

基于 CFG 的 CFI, 也称之为细粒度 CFI, 通过静态分析生成 CFG, 并严格按照 CFG 执行程序, 具备很高的安全性, 理论上能够检测任何非法的控制流劫持。但是, 仅仅通过静态分析几乎不可能生成包含所有可达路径的理想 CFG, 且细粒度 CFI 的性能开销很大[4]。正因为上述不足, 近几年一些研究者提出了针对基于保守 CFG 的 CFI 的绕过方法[16, 17], 更有研究者表明即使能够生成理想的 CFG, 也可以通过污染非控制数据使得控制流合法转移, 绕过基于理想 CFG 的 CFI[18]。

2.2.2 基于策略的 CFI

基于策略的 CFI, 也称之为粗粒度 CFI, 它并不依赖静态生成的 CFG, 而是通过制定一些控制流转移的规则来防御多数 CRAs, 即依据普通程序的合法控制流转移规律来制定安全策略, 因此粗粒度 CFI 具备低开销, 易部署的优势。这些规则大体上可分为以下两类:

1) 控制流转移策略: 这些策略主要规定不同间接转移指令的合法目的地址[7, 19]。

RETURN 指令 一条 return 指令应当指向 call 指令的下一条指令(call-preceded 指令)。

CALL 指令 一条 call 指令应当指向一个函数开始执行的第一条指令(函数入口指令)。

Indirect-JUMP 指令 一条 indirect-jmp 指令要么指向所在函数地址范围内的任意一条指令, 那么指向其他函数开始执行的第一条指令(函数入口指令)。

2) 代码长度策略: 这些策略规定单一配件的长度, 以及完成 CRA 使用的配件链长度[8, 9]。

配件长度 考虑单一配件, 除了指令片段应当以间接转移指令为结尾, 还应当满足包含的指令条

数大致在 2~6 的数量范围之内, 更长的指令片段不再认为是配件。

配件链长度 考虑配件链中配件数量的统计规律, 通常完成多数 CRAs 至少需要六个配件, 因此更短的配件链长度不再认为是 CRA。

2.3 合法配件

粗粒度 CFI 通过定义一些规则(如 2.2 节中提到的控制流转移和代码长度策略)来识别程序中哪些代码片段是合法的指令, 哪些是攻击者利用的配件。但是, 攻击者能够找到绕过这些规则的特殊配件(Special Gadgets), 本文称之为合法配件。因此在 CRA 中使用合法配件就可以绕过粗粒度 CFI 的防御, 下面介绍三类合法配件的概念:

1) Call-Preceded 配件

Call-Preceded 指令是指代码空间中 call 指令紧邻的下一条指令[13]。Call-Preceded 配件表示以一条 Call-Preceded 指令开头的配件类型, 因此它能够符合粗粒度 CFI 的控制流转移策略, 作为任何一条 return 指令的目的地址。

2) Entry-Point 配件

Entry-Point 配件是指以函数入口指令为开头的配件类型, 因此它能够符合粗粒度 CFI 的控制流转移策略, 作为任何一条 call 指令或 indirect-jmp 指令的目的地址。

3) Long-NOP 配件

Long-NOP 配件是指包含足够指令数量的配件类型, 且不能和普通配件依赖的数据产生冲突。通常 Long-NOP 配件包含一些如 NOP 等不修改寄存器的指令, 或者包含一些写内存指令(修改无关紧要的内存地址), 这些指令不会影响配件链的正常功能。因此攻击者可以使用 Long-NOP 配件构建更长的指令片段作为攻击配件, 以此绕过粗粒度 CFI 中基于代码长度的策略。

2.4 威胁模型

本文提出的防御机制主要针对于各类 CRAs, 因此我们假设攻击者拥有对数据和堆栈的控制权, 同时能够修改处理器中通用寄存器的值。对于攻击手段, 我们假设系统中已经部署 DEP, 因此攻击者不能够使用代码注入攻击而不得不选择使用代码重用的攻击方式。对于攻击目的, 我们假设主要是为了获得系统权限。另外我们假设攻击者不使用非对齐配件, 所谓非对齐配件, 是由不定长指令集(例如 x86 平台)导致对于相同一段机器码, 从不同位置开始读取, 得到的指令是不一样的。而这种情况很容易被 CFI 的控制流转移策略检测[19], 因此本文不考虑非对齐配件的情况。原因有两点, 第一, GWT 框架要求所有的间接跳转指令之前都接一条预取指令, 增加了攻击者寻找非对齐配件的难度, 如果一个间接跳转指令之前没有预取指令, 或者预取指令包含的信息有误, 我们就可以认为这是一个非对齐配件; 第二, GWT 和粗粒度 CFI 结合使用, 规定了间接跳转指令的目标地址(如函数头, call 指令的下一条指令), 也就避免了非对齐配件地出现。

为了更好地介绍我们提出的防御方法 GWT, 本文提出了以下两种不同的系统环境:

1) 基础环境

基础环境是指系统中不包含任何其他特殊的防御机制, 仅仅部署 DEP 技术。

2) 保护环境

保护环境是指系统中除了部署 DEP, 还添加了如 2.2 节描述的粗粒度 CFI 技术, 对控制流转移进行保护。

3 配件加权标记

本章节包括两个部分, 第一部分在 3.1~3.3 节介绍**基础环境**中配件加权标记(Gadget Weighted tagging, GWT)的整体框架, 主要包含三个步骤: 寻找所有可能被 CRA 利用的配件; 根据权值信息标记第一步找到的配件; 程序运行时监控权值标记的变化情况, 判断是否发生 CRA。

第二部分作为对第一部分的补充, 在 3.4 节介绍**保护环境**中 GWT 结合粗粒度 CFI 的框架(GWT+CFI)。粗粒度 CFI 包含控制流转移策略和代码长度策略, 但是 GWT 中已经考虑了配件的长度和类型, 也就是说 GWT 已经包含了代码长度策略, 所以 GWT 要比单独考虑代码长度策略的防御机制更加精确[8,9], 因此我们添加控制流转移策略作为 GWT 的补充即可。

3.1 配件寻找过程

3.1.1 寻找配件的结尾

因为配件的主要特征是以间接转移指令(return, call, indirect-jmp)为结尾, 所以理论上所有以间接转移指令结尾的代码片段都有可能作为配件。因此, 我们首先搜索程序代码空间中所有的间接转移指令, 并把这些指令定义为配件的结尾。

3.1.2 定义配件类型

著名的 ROP 攻击框架 Q[3]开源了一种支持多种架构的 ROP 编译器, 它能够根据 Q 中定义的配件类型判断一段指令是否属于配件以及配件的具体类型。Q 依据指令片段表示的功能定义不同的配件类型,

包括 NoOp, Jump, MoveReg, LoadConst, Arithmetic, LoadMem, StoreMem, ArithmeticLoad, ArithmeticStore 共九种基本配件类型[3], 并提出了一种算法来识别每个配件的类型, 且每个配件应当仅仅符合定义中的一种功能。

在 GWT 中, 我们定义四种配件类型, 如图 2 所示。其中, 功能配件和 Q 定义的配件类型一致, 但不包括 NoOp, 调度配件(Dispatcher Gadget)和系统调用配件(Syscall Gadget)是功能配件(Functional Gadget)中的两类特殊配件, 并且这两类特殊配件在 CRA 中具备至关重要的功能。下面对具体的配件类型进行介绍:

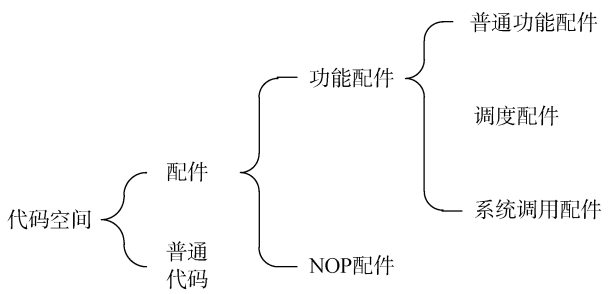

图 2 GWT 中配件类型的分类
Figure 2 The Gadget Types in GWT

1) 功能配件

在 CRA 中, 功能配件是稳定完成某些实际操作且不产生任何副作用的指令片段, 例如算数逻辑运算、分支转移、从内存读取数据等。GWT 中功能配件的定义与 Q 基本一致, 但不包括 NoOp 配件。

2) 调度配件

调度配件是一种特殊的功能配件, 它在 JOP 中扮演重要的角色[14], 包含一些自修改跳转地址的操作, 并且以 indirect-jmp 指令结尾。调度配件的本质就是模拟了一个虚拟的 PC 指针, 驱动 JOP 攻击的可持续过程, 连接一个又一个功能配件。

3) 系统调用配件

系统调用配件是一种以 syscall 指令结尾的特殊功能配件, 攻击者首先污染 rax 填充任意系统调用号, 然后结合系统调用配件实现恶意的系统调用劫持。例如, 为了使 CRA 更加简单, 攻击者通过系统调用配件陷入内核, 强行关闭一部分内存页的不可执行位(使得 DEP 防御失效), 然后向这部分内存注入恶意代码, 最后劫持控制流转向恶意代码完成攻击 [10]。

4) NOP 配件

NOP 配件指那些既不产生功能也不产生副作用的配件类型, 它包括 Q 提出的 NoOp 配件和本文第二章提到的 Long-NOP 配件。为了确保添加 NOP 配件不改变 CRA 配件链的原始语义, NOP 配件通常只使用很少一部分寄存器[13], 这样可以避免和功能配件使用的寄存器产生冲突。攻击者使用 NOP 配件的目的往往是绕过一些防御机制, 例如基于策略的粗粒度 CFI[8-9]。

3.1.3 寻找配件的开端(最大配件长度)

根据配件的组成特点, 很容易找到它的结尾(间接转移指令), 但寻找配件的开端相对困难。根据之前研究者的工作, 往往通过代码的长度去识别配件, 因为配件的长度具有一定的规律, 过长的代码片段会产生副作用, 也有可能会和其他配件相互冲突。通常配件的长度在 2~6 条指令之间[8], 但是不排除攻击者为了躲避一些防御机制, 使用几乎对 CRA 没有副作用的特殊配件(例如 Long-NOP 配件), 将其插入常规配件链中以躲避基于规则的防御。

在 GWT 框架中, 随着配件长度的增加, 包含过多的指令将产生无法避免的副作用, 所以 NOP 配件终将变成普通代码。普通代码就是无法被任何 CRAs 利用的配件, 本文假设所有不是功能配件和 NOP 配件的代码都是普通代码。类似, 若配件包含过多的指令, 功能配件也终将变成 NOP 配件或普通代码。因此, GWT 使用每种配件类型的最大长度来识别配件的开端。下面分析功能配件和 NOP 配件的最大长度:

1) 功能配件的最大长度

寻找到配件的结尾以后, 采用回溯的方法寻找配件的最大长度, 每次向前增加一条指令, 并结合 Q[3]中识别配件的模块(ROP 编译器)判断该指令片段是否属于配件, 属于什么类型的配件。如果属于功能配件, 则继续回溯指令, 直到该指令片段不再属于功能配件为止, 此时前一状态下的指令数就是该功能配件的最大长度, 指令片段的第一条指令就是该功能配件的开端。对于调度配件和系统调用配件, 因为它们属于功能配件的特殊情况, 所以它们最大长度的寻找过程同普通功能配件一致。

2) NOP 配件的最大长度

类似功能配件, 当 Q 判断出该指令片段属于 NOP 配件时, 继续回溯, 直到该指令片段不再能够被 CRA 利用为止(例如普通代码), 此时前一状态下的指令数就是该 NOP 配件的最大长度, 指令片段的第一条指令就是该 NOP 配件的开端。

3.2 标记权值信息

GWT 框架需要为找到的所有配件附加权值标记信息, 权值标记的结构包含三个部分: 配件类型, 功能配件的最大长度, 以及 NOP 配件的最大长度。这里的配件类型除了 3.1.2 节中定义的四种之外,

还包括普通代码, 表示无法被任何 CRAs 利用的配件类型。

其中功能配件最大长度的概念涵盖了特殊配件的情况, 即包括普通的功能配件和调度配件以及系统调用配件的最大长度。随着配件长度的增加, 功能配件可能先变为 NOP 配件再变为普通代码, 所以存在同一条配件结尾指令(间接转移指令)在不同状态下对应两种配件类型的最大长度。因此我们假设如果同一条配件结尾指令对应不同的配件类型, 依据 NOP 配件的最大长度不小于功能配件的最大长度进行进一步判断。

不同的配件类型在 CRA 中扮演着不同的角色, 产生不同的效果, 因此我们依据对系统的威胁程度为不同配件类型分配不同的权值, 权值越高, 则表示对应的配件类型对系统威胁越大, 同时在 CRA 中也越可能被使用。

每一种配件类型的具体权值分配如表 1 所示, 不同配件类型的权值可以由用户根据安全需求自行配置, 而且也可以添加新的配件类型和新的权值。对于 NOP 配件, 因为它对 CRA 并无实质性的帮助, 只是为了稀释配件链的长度而存在, 所以分配给它的权值为 0。对于功能配件, 因为它包含了 CRA 中一系列的具体功能操作, 所以分配给它的权值为 1。对于调度配件, 因为它是 JOP 攻击中的核心配件且管理其他配件的运行, 所以分配给它的权值为 2。对于系统调用配件, 因为它是操作系统内核函数的入口, 并且多数 CRAs 的最终目的是通过系统调用提权或禁用系统安全机制, 它比其他类型的配件都重要, 所以分配给它的权值为 4。对于普通代码, 因为 CRA 中配件链的执行是连续无间断的, 另外本文假设普通代码包含过多实际的功能操作, 对 CRA 具有很强的干扰, 会破坏配件之间的数据传递, 不可能被当作配件使用, 攻击者不可能把普通代码插入到任何的配件链中, 所以当遇到普通代码时, 则可判断此时不会发生恶意攻击, 对权值进行清零操作。

表 1 不同配件类型的权值分配

Table 1 Weighted Scores of Different Gadget Types

配件类型的二进制表示(3 bits)	配件类型	权值分配
000	普通代码	清零
001	功能配件	1
010	调度配件	2
011	系统调用配件	4
100	NOP 配件	0
其他	未定义	-

3.3 监控配件标记

因为 CRA 由一系列连续的配件组成, 且每个配件以一条间接转移指令为结尾, 基于这两点特性, GWT 截取程序运行时每两条间接转移指令之间的代码片段(包含后执行的那条间接转移指令)组成不同长度的可能配件, 本文称之为候选配件。另外我们在静态分析阶段为每条间接转移指令附加了权值标记, 且 GWT 又能够在动态分析阶段识别当前的候选配件, 因此可以通过动静结合的方式判断运行时候选配件的真实类型: 如果当前候选配件的长度大于 NOP 配件的最大长度, 则候选配件可判断为普通代码; 如果当前候选配件的长度小于功能配件的最大长度, 则候选配件可判断为功能配件; 如果以上两种情况都不满足, 则可判断候选配件为 NOP 配件。

通过上述过程, GWT 对当前程序运行状态下的候选配件进行判断, 若识别为普通代码, 则表示当前程序并无 CRA 发生。相反, 若 GWT 识别多个候选配件为功能配件时, 则表示当前程序很有可能发生了 CRA。GWT 通过动态监控当前程序候选配件的真实类型, 依据表 1 中不同配件类型的权值分配, 累加候选配件对应的权值, 最终得出一个数值, GWT 根据这个数值得出发生 CRA 的概率。

3.4 GWT+CFI 框架

3.4.1 GWT+CFI 框架的动机

GWT 的主要不足之处在于无法准确识别配件的开端, 仅仅是通过寻找配件的最大长度来确定, 这并不是十分精确, 因此我们提出 GWT 结合粗粒度 CFI 的思想, GWT+CFI 能够带来以下两点好处:

1) 识别配件开端

攻击者为了能够绕过粗粒度 CFI 的防御, 常见的思路就是使用特殊配件, 例如 Call-Preceded 配件和 Entry-Point 配件。正因为如此, 特殊配件也带来了无法避免的特征, 那就是攻击者必须要求配件的第一条指令, Call-Preceded 配件需要以 Call-Preceded 指令为开端, Entry-Point 配件需要以函数入口指令为开端。因此, 我们能够使用 GWT+CFI 的思想更精确地识别配件的开端。

2) 减少配件数量

除了合法配件, 粗粒度 CFI 能够检测出其他非法的配件, 所以我们能够使用 GWT+CFI 的思想减少能够被 CRA 利用的配件数量, 因此我们能够更加精确定位代码空间中可能被 CRA 利用的配件。

3.4.2 寻找合法配件

GWT 和 GWT+CFI 的最大区别在于寻找配件的过程。GWT+CFI 只需去寻找合法的配件即可, 对于

配件的权值标记和动态监控都和基础的 GWT 框架一致。

GWT+CFI 寻找配件的过程大致如下: 首先, 需要找到所有的间接转移指令, 即配件的结尾; 其次, 需要找到所有合法配件的开端, Call-Preceded 配件的开端就是 Call-Preceded 指令, Entry-Point 配件的开端就是函数的入口指令; 然后, 根据配件的结尾和开端识别所有的合法配件; 最后, 应当从合法配件中选择能够被 CRA 利用的配件, 选择过程和 3.1 节中介绍的基本一致, 区别在于对配件类型的划分。GWT+CFI 包含的配件类型有功能配件、调度配件、系统调用配件、NOP 配件、普通代码, 同时引入了合法配件和非法配件的概念, 如图 3 所示。

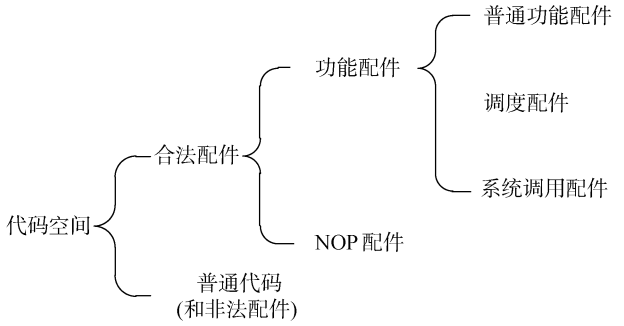

图 3 GWT+CFI 中配件类型的分类
Figure 3 The Gadget Types in GWT+CFI

需要注意的是间接 jmp 指令在相同的函数内部能够任意位置跳转, 且并不违背粗粒度 CFI 中关于控制流转移的策略。攻击者能够利用这一点在函数内部找到除 Call-Preceded 或 Entry-Point 配件之外的合法配件, 因此对于包含 jmp 指令函数内部的配件寻找过程应当按照 GWT 框架中的算法, 而不是 GWT+CFI 框架中合法配件的寻找算法。

4 实现细节

4.1 配件寻找过程

本节内容主要介绍 GWT 和 GWT+CFI 的配件寻找过程, 此过程能够找到并确定所有可能配件的类型和长度, 分为四步, 可用过程 2 描述如下:

过程 2. 配件寻找过程

步骤一: 利用二进制分析工具 IDA pro 编写脚本找到程序中所有的间接转移指令(配件的结尾), 另外在 GWT+CFI 框架中, 还要找到所有的 Call-Preceded 指令和函数入口指令(配件的开端)。

FOR EACH(间接转移指令){

步骤二: 通过回溯的思想筛选出每一个可能的配件, 直到找到功能配件的最大长度为止。

步骤三: 对步骤二中得到的每一个可能配件, 利用 Q[3]进一步得到配件的类型。

步骤四: 获取 NOP 配件的最大长度, 这个值不能够小于功能配件的最大长度。

}

END EACH

我们基于 Q[3]提出的算法判断每个配件的类型。对于功能配件, 它往往包含一系列具有目的性的操作, 并且应当只完成 Q 中定义的一种操作, 例如 MoveReg, 算术逻辑运算等。识别功能函数的算法细节可以参考文献[3], 核心思想是提取指令片段执行前后的状态变化(相关寄存器和内存值的变化), 用先决条件公式判断这些变化是否满足某个配件类型的特征条件, 从而得出功能配件的具体类型。但是 Q 中的原始算法仅仅能够用来寻找以 ret 指令结尾的 ROP 配件(Q 开源的 ROP 编译器框架更新了对 indirect-jmp、indirect-call, 和 syscall 的支持, 通过间接跳转指令之前的代码片段判断该配件的类型), 因此我们需要添加新的特征模块去寻找 GWT 中定义的合法配件。

如果一个配件被判断是功能配件, 我们还需要进一步识别它是否属于特殊的功能配件。如果它以 syscall 指令结尾, 则属于系统调用配件。如果它以 indirect-jmp 指令结尾, 且配件中包含修改 indirect-jmp 目标寄存器的指令操作, 则属于调度配件。其他情况则属于普通的功能配件。

如果一个配件不属于功能配件, 则它可能属于 NOP 配件或者普通代码。为了不改变 CRA 中配件链的原始语义, NOP 配件应该尽可能少的包含操作寄存器的指令。例如, 对于超长的配件, 如果该配件内部操作简单, 则该配件仍然会被当作 NOP 配件, 如果该配件的操作很复杂, 对寄存器的操作很多, 则认为该配件无法被攻击者利用, 所以被认为是普通代码。因此我们定义一个重要的变量 *MaxRegMod*, 表示 NOP 配件中修改寄存器的最大个数。如果一个配件修改寄存器的个数不超过我们定义的 *MaxRegMod*, 则这个配件就属于 NOP 配件, 否则不属于。需要注意的是, 在 GWT 中功能配件的优先级要高于 NOP 配件, 即如果同一个配件同时属于两种类型, 那么应当首先考虑它属于功能配件的情况。另外 NOP 配件和普通代码的区分是比较模糊的, 使用 *MaxRegMod* 也只是一种折中的办法, 并不能彻底将这两类配件完全分开。但是对于当前实际的攻击, 我们认为 GWT 已经基本满足需求, 并且优于其他基于配件长度的检测[8-9]方法。

GWT 寻找功能配件和 NOP 配件的方法不是一

成不变的, 而是随着配件分类地不断完善, 寻找算法也会不断优化。例如, 文献[21]提出了一种基于函数的 CRA, 使用整个函数作为一个配件, 这种思想能够绕过许多防御机制, 包括基于 CFG 的理想 CFI[18]。因此, 我们能够依据攻击者使用新型配件的定义在 GWT 中构建对应的配件寻找算法, 以此来可能防御基于函数的 CRAs。

图 4 表示 GWT 在程序中寻找配件的整个过程。第一, 找到一个间接转移指令, 例如 ret 指令; 第二, 判断配件的类型, 例如配件 1 包含两条指令(push 和 ret 指令), 因此它属于一个普通功能配件(StoreMem), 在添加标记信息时则注明为功能配件。同样, 配件 2 也是一个功能配件(ArithmeticStore)。配件 3 和配件 2 相比多了一条 pop 指令, 但这条指令会改变当前函数的栈布局, 对 CRA 产生一定的副作用, 因此配件 3 不属于功能配件。但是, 配件 3 修改寄存器的数量为 2, 如果我们设定变量 *MaxRegMod* 的值为 6, 根据前两段的描述, 则它属于 NOP 配件, 此时功能配件的最大长度就是 3(配件 2 包含的指令数)。随着更多指令的回溯, 我们假设配件 4 和配件 5 修改寄存器的数量分别为 6 和 7, 变量 *MaxRegMod* 的值依旧为 6, 则配件 4 属于 NOP 配件, 它的最大长度就是配件 4 包含的指令数, 而配件 5 因为修改寄存的数量大于我们设定的 *MaxRedMod*, 所以它不属于任何配件类型, 可判断为普通代码。

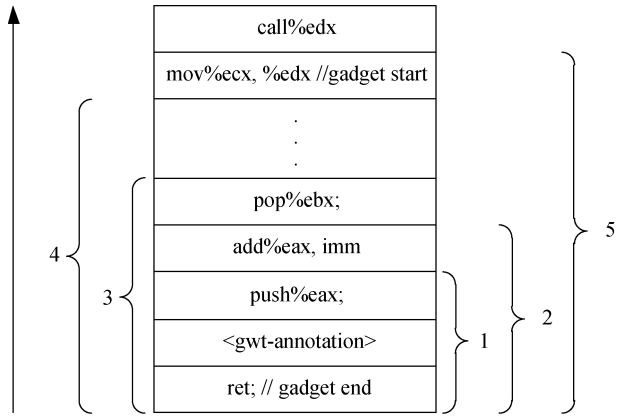

图 4 配件寻找示例

Figure 4 An Example of Gadget Search

在 GWT+CFI 框架中, 除了考虑配件的结尾, 我们还应该找到配件的开端(例如图 4 中的 mov 指令)。因此, 在 GWT+CFI 中, 只有配件 5 可能属于某种类型配件。如果按照前面的假设, 配件 5 不属于功能配件也不是 NOP 配件, 仅仅为普通代码, 则对应的功能配件和 NOP 配件的最大长度就是无效的。

4.2 程序标记结构

权值标记的结构如图 5 所示, 使用 32 位结构, 包含 3 个参数: 配件类型、功能配件的最大长度、NOP 配件的最大长度。其中, 配件类型占用 3 个比特, 具体表示的含义和表 1 一致, 功能配件的最大长度占用 14 个比特, NOP 配件的最大长度占用剩下 15 个比特。

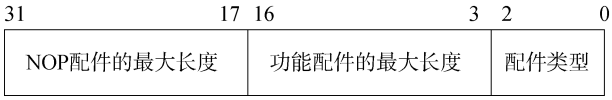

图 5 权值标记的结构示意图

Figure 5 The Structure of Weighted Tagging for Gadget End

我们利用图 5 中的标记结构, 在可执行程序的二进制文件中添加每个配件的标记信息。如图 4 所示, 在每个间接转移指令前添加一个 32 位的权值标记, 包含以该间接转移指令结尾的所有可能配件类型和相应配件类型的最大长度。为了保证二进制文件的兼容性, 标记结构以一条预取指令为开端, 预取指令之后是加权配件的信息, 这和文献[4,19]中的方法类似。因为我们假设系统支持 DEP 技术, 所以攻击者无法篡改权值标记的结构, 同时预取指令和标记信息相邻, 所以攻击者也无法伪造虚假的权值标记结构。

4.3 CRA 检测算法

在程序运行时, 每执行一条指令, 将配件的长度加一, 当执行一条间接跳转指令时, 读取对应的配件标记信息, 包括配件的类型和最大长度, 判断当前候选配件的真实类型。另外, 我们定义四个中间变量来计算 CRA 发生的概率, 分别是当前配件的类型(Current Gadget Type, CGT), 当前配件的长度(Current Gadget Length, CGL), 当前配件的真实类型(Real Gadget Type, RGT), 以及 CRA 发生的概率因子(CRA Occurrence Index, COI)。

运行时配件的真实类型 RGT 需要依据 CGT 和 CGL 来做出判断。有关 RGT 的实现过程由算法 1 描述如下, *MaxFunc* 表示功能配件的最大长度, *MaxNOP* 表示 NOP 配件的最大长度。

算法 1. RGT 的判断

输入: CGT, CGL, *MaxFunc, MaxNOP*

输出: RGT

```
IF (CGT = = Normal Code){
      RGT = Normal Code;
}
ELSEIF (CGT = = NOP Gadget){
```

```
        IF (CGL <= MaxNOP){
            RGT = NOP Gadget;
        }
        ELSE{
            RGT = Normal Code;
        }
    }
    //CGT=Functional/Dispatcher/Syscall Gadget
    ELSE{
        IF (CGL <= MaxFunc){
            RGT = CGT;
        }
        ELSEIF (CGL <=MaxNOP){
            RGT = NOP gadget;
        }
        ELSE{
            RGT = Normal Code;
        }
    }
```

我们使用 COI 表示 CRA 发生的概率, COI 越大意味着发生 CRA 的概率就越大。COI 的值由 RGT 和配件对应的权值决定, 详细的过程如算法 2 描述如下。除此之外, 我们定义变量 *MaxCOI*, 表示 COI 的最大值, 也是预先设定的阈值, 如果COI的值大于 *MaxCOI*, 则我们认为发生了 CRA, 否则程序没有异常行为。

算法 2. COI 的判断

输入: RGT, *MaxCOI*, RGT 对应的权值

输出: COI

```
COI = 0;                    //initialization
FOR EACH (间接转移指令){
    IF (COI <= MaxCOI){
        IF (RGT = = Normal Code){
            COI = 0;
        }
        ELSE{
            COI = COI + RGT 相应权值;
        }
    }
    //COI > MaxCOI
    ELSE {
        exit();     // CRA 发生
    }
}
```

4.4 硬件架构

关于 GWT 和 GWT+CFI 的硬件执行架构如图 6 所示, 包含三个重要的硬件模块: 控制流监控器(Control-flow Monitor, CFM), 权值标记配置(Weighted Tag Configuration, WTC), 以及 CRA 检测模块(CRA Detection Module, CDM)。另外, 我们假设基于策略的粗粒度 CFI 已经集成在这个系统环境中。

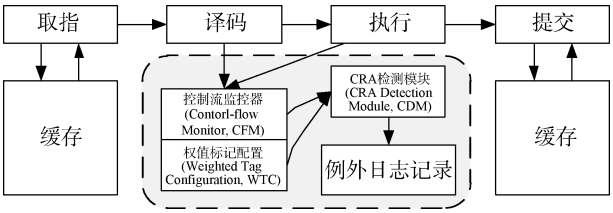

图 6 GWT 和 GWT+CFI 的硬件执行架构

Figure 6 The Hardware Implementation of GWT and GWT+CFI

CFM 是监控配件权值信息的主要模块, 它记录了程序运行时系统执行的指令数(CGL)和当前配件的类型(CGT)。当一条间接转移指令被执行时, CFM 根据算法 1 判断 RGT 的内容, 把 RGT 作为输出结果的同时输入 CDM。CDM 根据算法 2 计算 COI 的值, 并比较 COI 和阈值 *MaxCOI* 的大小关系, 如果 COI 的值大于 *MaxCOI*, 则判定为发生了 CRA, 发送一个例外并记录攻击日志。

WTC 主要负责存贮 CFM 和 CDM 使用的相关参数, 例如 *MaxCOI* 和表 1 中不同配件类型对应的权值。由于这些参数具有灵活可配置的特性, 因此用户能够根据自身需求修改这些参数, 例如, 若用户要提高系统的安全性, 他们可以降低阈值 *MaxCOI*, 或者提高配件的权值。相反, 若用户要缓解 GWT 的限制, 他们可以增加阈值 *MaxCOI*, 或者减少配件的权值。当然, 攻击者无法直接篡改存贮在 WTC 中的参数, 因为我们规定参数的初始值设定之后无法在 GWT 运行时更改。

5 实验评估

5.1 安全性分析

因为 GWT 的硬件架构非常简单, 仅仅需要记录配件的信息和计算 COI 的值。因此, 为了简化分析, 我们利用著名的动态插桩工具 Pin[22]来分析 GWT 框架的安全性。Pin 是动态程序分析工具, 能够对每条指令的执行进行监控, 通过编写 PinTools 来模拟 GWT 硬件执行的功能。关于测试样例, 我们选择 Ubuntu 系统目录/bin 和/usr/bin/下被用户广泛使用的程序。为了维护 GWT 框架的安全性和稳定性, 我们设置重要参数 *MaxRegMod* 的值为 6, *MaxCOI* 的值为 8, 不同配件类型对应的权值和表 1 一致。

因为 GWT 和 GWT+CFI 框架的主要不同在于配件的寻找过程, 而两者对于 CRA 的检测过程是一致的, 因此我们主要关注 GWT 框架的安全性分析, 而 GWT+CFI 和它是一致的。

5.1.1 配件寻找过程

在 GWT 框架中, 每个测试程序 40KB 的代码空间中平均存在 1072 个可能属于配件的代码片段(候选配件)。进一步分析不同配件类型的数量关系得出, 功能配件、调度配件、系统调用配件, 以及 NOP 配件的平均数量分别为 452、7、1.5, 以及 612, 如图 7 所示。另外, 每个程序中包含的所有候选配件的平均长度为 9.9, 其中功能配件(包括调度配件和系统调用配件)的平均长度为 5.3, NOP 配件的平均长度为 13.3, 如图 8 所示。

在 GWT+CFI 框架中, 每个测试程序 40KB 的代码空间中, 合法配件的平均数量是 471 个, 相比较 GWT 基础框架中候选配件的数量, 合法配件的数量大幅减少, 尤其是合法的功能配件。其中, 功能配件、调度配件、系统调用配件, 以及 NOP 配件的平均数量分别为 45、0.9、0.15, 以及 425, 如图 7 所示。相反, 合法配件的平均长度为 13.5, 要大于 GWT 中配件的平均长度。其中功能配件和 NOP 配件的平均长度分别为 7.7 和 14.1, 如图 8 所示。

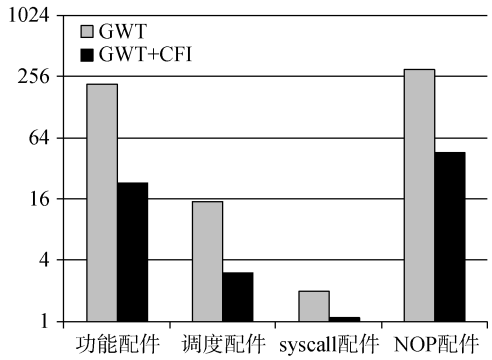

图 7 GWT 和 GWT+CFI 中不同配件的平均数量
Figure 7 The Average Number of Different Gadgets in GWT and GWT+CFI

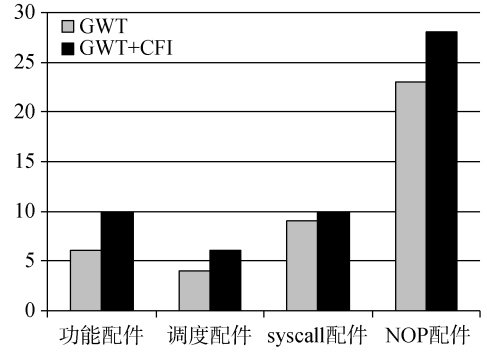

图 8 GWT 和 GWT+CFI 中不同配件的平均长度
Figure 8 The Average Lengths of Different Gadgets in GWT and GWT+CFI

对于 GWT 中的重要参数 *MaxRegMod*, 即配件修改寄存器的最大数量, 同时也是划分 NOP 配件和普通代码的边界阈值。显而易见, 如果增加 *MaxRegMod* 的值, 候选 NOP 配件的最大长度也会随之增加, 造成更多的普通代码转变为 NOP 配件, 因此会一定程度上增加 GWT 的误报率。我们改变参数 *MaxRegMod* 的初始化值, 重复前文过程 2 的配件寻找过程, 对测试集合中的程序进行静态分析, 统计不同 *MaxRegMod* 值对应的 NOP 配件的平均长度。实验表明, 随着 *MaxRegMod* 的增加, GWT 和 GWT+CFI 中 NOP 配件平均长度的变化情况如图 9 所示。尽管处理器中有很多寄存器, 但多数都有特定的用途, 仅仅只有 8 个通用目标寄存器(32 位系统)常常被使用, 因此当 *MaxRegMod* 的值大于 8 时, NOP 配件的平均长度几乎不再改变, 这和图 9 显示的实验结果一致。另外由于 NOP 配件对 CRA 的贡献不是很大, 也就是说 NOP 配件并不具备实际的功能价值, 所以我们分配给它的权值为 0, 因此适当的增加 *MaxRegMod* 的值并不会影响 GWT 对 CRA 的检测效果。

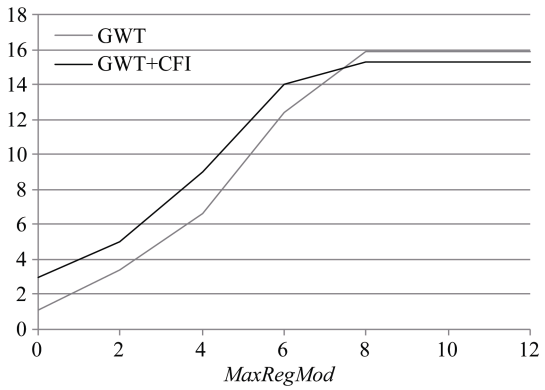

图 9 GWT 和 GWT+CFI 中 NOP 配件的平均长度
Figure 9 The Average Lengths of NOP Gadgets in GWT and GWT+CFI

5.1.2 实际的攻击

首先, 我们使用自动化工具 Q[3]生成实际可被攻击者利用的配件链, 然后针对已知的漏洞程序构造 CRA 来检测 GWT 框架的安全性。因为我们已经提前标记了目标程序中所有的可能配件, 所以 GWT 能够轻易检测出由连续配件组成的 shellcode, 并记录配件链中配件的权值信息。对于配件权值, 因为 Q 主要生成以 ret 指令结尾的配件, 且配件类型都属于 GWT 中的普通功能配件, 所以分配的权值均为 1。针对构造的攻击样本和 Linux 系统目录 /bin 和 /usr/bin/ 下一百多个常用程序进行实验, GWT 能够成功检测所有由 Q 生成的恶意配件链。

其次, 我们使用手动构造一些特殊的 CRAs 来试

图绕过 GWT 的防御, 以此检测 GWT 的安全性。在 GWT 框架中, 识别功能配件的标准非常严格, 而识别 NOP 配件的标准则相对宽松, 因此我们精心选择一些特殊的 NOP 配件(替代功能配件)构造 CRA。但是这些 NOP 配件可能会包含一些额外的操作, 因此构造一个长的 NOP 配件链去做一些复杂的操作是非常困难的, 原因在于 NOP 配件可能会产生副作用。一个实际可行的攻击方法是通过构建简单短小的配件链去关闭 DEP, 然后通过注入恶意代码的方式完成复杂的攻击目的。

配件类型对应的权值和变量 *MaxCOI* 是 CRA 检测模块的两个关键参数, 更大的权值分配和更小的 *MaxCOI* 能够提供更高的安全性, 但同时也会造成系统高的误报率, 因此我们应当选择合适的参数值来平衡安全性和稳定性的关系。我们改变参数 *MaxCOI* 的初始化值, 重复 PinTools 的插桩模块, 依据第 4.3 节 CRA 的检测算法对测试集合中的程序进行动态分析, 统计不同 *MaxCOI* 值对应的实验数据。如图 10 和图 11 描述了随着 *MaxCOI* 值的变化, GWT 中 CRA 检测模块查全率和误报率的变化情况。*MaxCOI* 的值变大, GWT 判定发生 CRA 则需要更多的配件, 一方面, 因为由 Q 编译器生成的配件链通常包含 20~40 个配件, 所以我们设置 *MaxCOI* 的值小于 20 才能检测由 Q 编译器生成的大部分配件链; 另一方面, 随着 *MaxCOI* 的减小, GWT 可能会误判普通程序发生了 CRA, 例如一个普通程序陷入一个系统调用, 这个系统调用可能被识别为系统调用配件, 如果我们设置 *MaxCOI* 的值小于 4, GWT 则判定这个普通程序陷入系统调用的行为是恶意的 CRA。

5.1.3 检测不同的 CRAs

我们提出的 GWT 框架理想情况下能够检测多种不同的 CRAs, 包括: 普通的 ROP 攻击[1], 普通的 JOP 攻击[14], 特殊配件构造的各种 CRAs[13,16], 基于

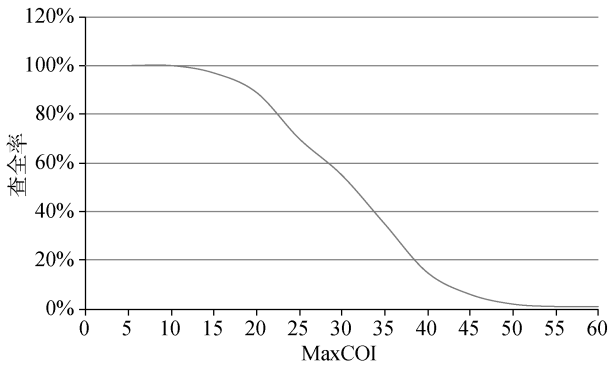

图 10 GWT 中 CRA 检测模块的查全率

Figure 10 The Recall Rate of CRA Detection Module in GWT

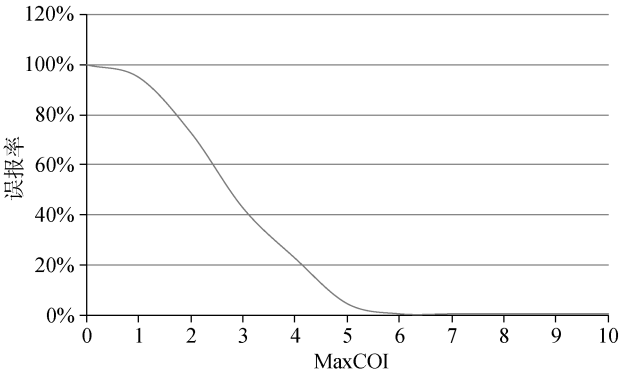

图 11 GWT 中 CRA 检测模块的误报率

Figure 11 The False Positive Rate of CRA Detection Module in GWT

函数的 CRA[21], 非控制数据攻击[18]。本文在理论的支持下探讨 GWT 对多种 CRAs 的防御效果, 对前三种 CRAs 进行真实的实现和测试, 未来工作中会进一步寻找新型 CRAs(基于函数的 CRA 和非控制数据攻击)的特征, 测试 GWT 的防御效果和性能损耗:

1) 普通的 ROP 攻击

普通的 ROP 攻击是指以 ret 指令为结尾的配件构成的 CRA, 例如攻击者使用自动化工具 Q 生成 ROP 配件链, 因为我们已经标记了这些配件, 所以一旦 ROP 配件的权值总和大于 *MaxCOI*, GWT 就能够轻松检测。

2) 普通的 JOP 攻击

普通的 JOP 攻击是指使用 indirect-jmp 指令为结尾的配件构成的 CRA, 它包含了普通的功能配件和调度配件, 而调度配件在 GWT 中会被分配很高的权值, 因此相比较 ROP 攻击, GWT 更容易检测和防御 JOP 攻击。

3) 特殊配件的 CRAs

NOP 配件 通常 NOP 配件被用来绕过基于策略的粗粒度 CFI[13,20]。在 GWT 中, 我们识别 NOP 配件, 并分配给它的权值为 0, 一方面, 在配件链插入 NOP 配件并不会减少功能配件的数量, GWT 能够检测插入任何数量 NOP 配件的 CRA。另一方面, GWT 无法检测全部由 NOP 配件组成的 CRA, 但是全部使用 NOP 配件几乎不可能构造出复杂的攻击过程, 也无法被自动化的工具生成。因此, GWT 能够检测使用 NOP 配件来稀释功能配件链的 CRA, 同时不会受到 NOP 配件的误导。

系统调用配件 系统调用配件主要被 CRA 用来减少配件链的长度。如果我们设置 *MaxCOI* 的值为 8, 系统调用配件的权值为 4, GWT 能够检测出执行系统调用之前包含不少于 4 条功能配件的 CRA。当

然我们也可以减小 *MaxCOI* 的值并增大系统调用的权值来检测包含更少配件数量的 CRA。

其他特殊配件 如果攻击者提出了一些新的配件类型来绕过现有的防御机制, 我们可以在 GWT 中添加这些特殊的配件并分配它们合适的权值, 例如文献[16]提出了一种新的配件类型, 它能够绕过基于不精确 CFG 的细粒度 CFI, 称之为间接函数调用的参数污染配件(Argument Corruptible Indirect Call Sites, ACICS), 包含一个间接调用指令和一个目标函数, 且目标函数能够在符合 CFG 的情况下执行远程代码[16]。我们可以根据文献中关于 ACICS 配件的描述, 在目标程序中找到并标记这类配件, 这样 GWT 就能够检测并防御使用 ACICS 配件的 CRA。

4) 基于函数的 CRA

基于函数的 CRA 使用整个函数作为配件[18,21], 它能够绕过大多数的 CFI 方案。如图 12 所示, 基于函数的 CRA 也需要调度配件来管理程序中能够被利用的函数配件, 如果我们能够找到并标记这些调度配件, GWT 就能够检测基于函数的 CRA。不过要准确区分普通程序和基于函数的 CRA 在调用函数时的不同特征, 正如图 12 所示, 如果程序仅仅调用单一目标函数, 则它属于普通程序, 若连续调用许多不同函数, 则它可能属于 CRA, 例如我们应当标记图 12 中的 *test()*函数为普通代码而不是调度配件。

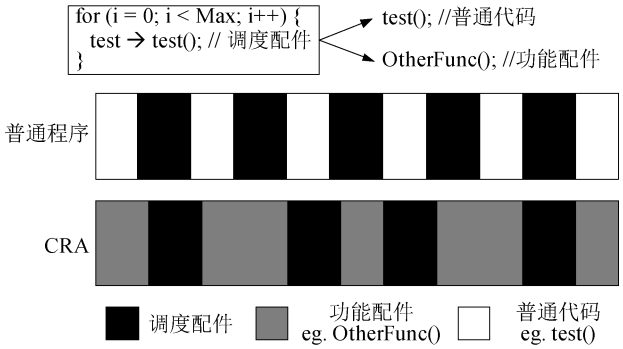

图 12 基于函数的 CRA 和普通程序的对比

Figure 12 The Comparison of Function-based CRAs and Normal Programs

5) 非控制数据攻击

文献[18]提出了一种控制流弯曲的新型 CRA, 它结合了非控制数据攻击的思想, 通过中转函数绕过基于精确 CFG 的细粒度 CFI。但是在 GWT 框架中, 我们可以标记所有能够被控制流弯曲利用的漏洞函数, 这些函数中存在可被利用的功能配件, 例如 *printf(), fputs()*。如果这些库函数连续执行的数量超过了 *MaxCOI*, GWT 就能够检测控制流弯曲攻击。相反, 普通程序通常很少连续多次执行这些库函数, 另外, 控制流弯曲的攻击也需要调度配件来管理这些中转函数, 所以我们标记和监控调度配件理论上也可以成功防御控制流弯曲攻击, 方法类似基于函数的 CRA。

5.2 性能分析

我们在配置为 CPU Intel x/E7-4820, 频率 2.5GHz,内存类型 DDR3-1600, 大小 8G, 共享缓存 16MB 的服务器上安装 Ubuntu 16.04, 系统内核版本 4.4(x86)。我们通过运行硬件模拟器 SimpleScalar, 实现了 GWT 和 GWT+CFI 的硬件架构, 如图 6 所示。

因为 GWT 和 GWT+CFI 框架的主要不同在于配件的寻找过程, 而两者对于 CRA 的检测过程是一致的, 因此它们在 SimpleScalar 中的功能实现可由过程 3 描述如下, 共添加代码 310 余行。另外, 关于处理器参数的具体配置细节总结于表 2。我们从标准的 SPEC CPU2006[23]中选取性能测试套件, 这些标准测试程序均通过 GCC 编译器进行 O3 级优化编译。

表 2 处理器配置信息

Table 2 The Configuration of Processor

体系结构	参数配置
处理器	四发射, 乱序执行
L1 I-Cache	64KB, 四通道, 32B lines
L1 D-Cache	64KB, 四通道, 32B lines
L1 Latency	2 cycels
L2 Cache	1MB,四通道, 32B lines
L2 Latency	20 cycles
主存	4GB, 256 cycles latency

过程 3. GWT 和 GWT+CFI 在 SimpleScalar 中的逻辑实现

```
sim-safe.c main():
    INIT WTC()
    MD_FETCH_INST()
    IF (Indirect-Branch):
        THEN:
            INPUT static_result
            IF (EXIST):
                THEN:
                    EXECUTE CFM()
                ELSE:
                    CONTINUE
            FI
            EXECUTE CDM()
            IF (CRA):
                THEN:
                    Exception()
```

```
                ELSE:
                    CONTINUE
            FI
        ELSE:
            CONTINUE
    FI
    MD_SET_OPCODE()
```

基于所有的测试套件, 比较程序在不同架构(基本架构和粗粒度CFI/GWT架构)上的运行时间, 从而得到性能开销的具体数据。我们比较了 GWT, 粗粒度 CFI, 以及 GWT+CFI 的性能开销, 如图 13 所示。因为 GWT 已经考虑了代码长度策略, 所以粗粒度 CFI 只需要添加三条控制流转移策略即可。实验结果表明 GWT 的平均性能开销只有 2.31%, 最大不超过 6%。GWT+CFI 的平均性能开销为 3.55%, 最大不超过 9%, 因为 GWT+CFI 不仅仅包含了 GWT 的性能开销, 而且包括粗粒度 CFI 的开销。由于 GWT 和粗粒度 CFI 的实现具有可以复用的部分, 例如 GWT 和粗粒度 CFI 都需要监控和追踪间接跳转指令的运行, 因此 GWT+CFI 的性能损耗要小于单独 GWT 的性能损耗加上单独粗粒度 CFI 的性能损耗。

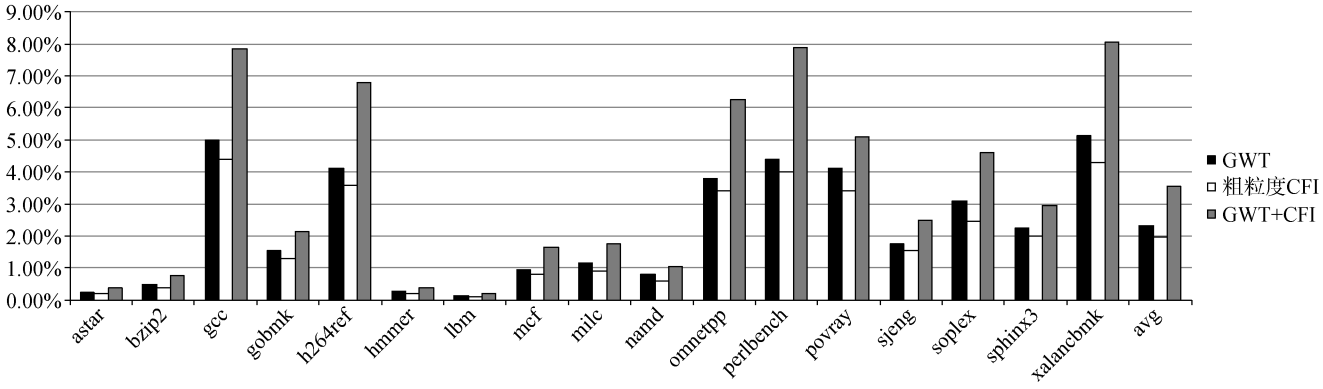

图 13　不同方法的性能开销(归一化为基础环境)

Figure 13　Performance Overheads of Different Methods(Normalized to the Baseline System)

6　讨论

6.1　配件分类

本文把配件主要分为三种类型: 普通代码(无法被 CRA 利用的代码片段)、NOP 配件, 以及功能配件(包括调度配件和系统调用配件)。CRA 使用功能配件完成稳定且有用的操作, 使用 NOP 配件来绕过防御机制。配件的分类虽然简单明了, 但准确识别这些配件类型并不容易。

我们使用基于 Q 的配件寻找算法来识别功能配件, 但Q识别配件的规则过于严格, 这使得攻击者人工找到一些额外的功能配件成为可能。另外我们通过设置变量 *MaxRegMod* 来识别NOP配件, 但是攻击者可能会构造出只需要修改少量寄存器的特殊 NOP 配件来绕过 GWT。因此精确识别各种配件类型并不是很容易, 仅仅使用 Q 或 *MaxRegMod* 还无法做到精确识别各种配件类型。

我们未来的工作会寻找更准确识别功能配件、NOP 配件, 以及普通代码的方法。例如我们可用把 NOP 配件再进行细分, 分为短 NOP 配件, 中等 NOP 配件和长 NOP 配件。其中短 NOP 配件几乎没有额外的副作用, 不会影响功能配件的操作, 而长 NOP 配件可能会带来更多的副作用, 很难被 CRA 利用, 因此可以分别分配短 NOP 配件, 中等 NOP 配件和长 NOP 配件的权值为 0.5, 0 和–0.5, 通过对配件权值的调整增加 GWT 的安全性。

6.2　灵活性分析

GWT 具有良好的灵活性和稳定性, 因此未来我们可以对该框架进一步优化和完善。

1) 可扩展的配件寻找模块

随着 CRA 技术的不断发展, 新的攻击方法和新的配件被研究者提出。例如, 最早的 return-to-libc 首先被提出, 接着新的方法 ROP[1], JOP[2]和基于函数的 CRA[21]相继被提出。配件也由基于 ret 指令的传统配件发展为基于 jmp 指令, Call-Preceded 指令和基于函数的新型配件。

因为每提出一种新的 CRA 方法或配件类型都需要详细描述如何使用或构建它们, 所以我们能够根据描述的细节在 GWT 中完善配件的寻找算法, 因此 GWT 能够在程序运行时监控新的配件并检测新的 CRA。本文已经在 GWT 和 GWT+CFI 中添加了基于 ret 和 jmp 指令的配件, 以及合法配件的寻找算法。未来我们可以添加更多的配件(例如基于函数的配件)寻找算法以提高 GWT 的安全性。

如今攻击者广泛使用自动化工具去寻找配件, 因此在 GWT 中也能够重用自动化工具帮助我们寻找配件, 例如本文就是使用 Q 的自动化框架去识别功能配件。同时所有被自动化工具生成的配件都可以通过 GWT 进行标记, 也就是说, 我们的方法理论上能够防御所有使用自动化工具生成配件链的 CRAs。

2) 可配置的初始化参数

GWT 使用到的参数例如 *MaxCOI*, *MaxRegMod* 以及配件对应的权值, 都是可以重新配置的, 因此用户能够根据实际需求修改这些参数。参数配置对 GWT 的影响极大, 寻找合适的参数需要大量的实验数据, 而且根据现实环境的不同, 攻击特征也会改变, 参数也应该随之调整。例如本文根据 GWT 的基本原理、前人工作和实际经验, 给出了参数配置的参考值, 并通过实验验证了参数配置的实际效果, 探讨了参数配置对漏报率和误报率的影响。另外, 如果我们发现新的配件类型, 可以在 GWT 中添加这些新配件, 并分配新的权值, 因此 GWT 能够检测多种 CRAs 使用的不同配件, 通过添加更多的配件类型可以提高 GWT 的安全性。例如本文在 Q 的基础上添加了调度配件和系统调用配件, 这是两种特殊的新型功能配件, 并为这两类配件分配新的权值, GWT 就能够检测和防御相应的 CRA。

7 相关工作

总结相关研究工作, 防御 CRA 的方法主要有两种[42]: 基于随机化思想的防御机制和基于控制流的指令检查机制。

基于随机化的防御思想通过随机化程序的代码布局, 使得攻击者无法找到配件, 也无法连接配件。地址空间布局重排(Address Space Layout Permutation, ASLP)[24]在函数级随机化代码布局, 指令布局随机化(Instruction Layout Randomization, ISR)[25]在指令级随机化代码布局。但这些基于随机化思想的防御方案都受到内存泄露攻击的威胁[26], 例如 BlindROP[27]利用内存泄露绕过细粒度的代码随机化。随后, 也有文献[28-29]进一步提高随机化的熵值和粒度来抵御内存泄露攻击。

Abadi 等人[4, 15]提出基于控制流完整性的防御思想, 即 CFI。原始 CFI 具有较大的性能开销, 因此研究者提出牺牲部分安全性来换取良好性能的实际方案, 主要分为两类: 基于策略的 CFI 和基于 CFG 的 CFI。

相比原始 CFI, 基于策略的 CFI 并没有根据 CFG 验证控制流的完整性。文献[6-7]通过二进制重写技术标记所有间接转移指令的可能目标地址, 其实这些可能的目标地址就是本文介绍的函数入口指令或 call-preceded 指令, 控制流转移只能跳转到这些可能的目标地址, 否则中断程序执行。kBouncer[9]和 ROPecker[8]利用 x86 处理器中最近分支记录(Last Branch Record, LBR)寄存器检测程序中异常的控制流转移, 但是这些方案都仅仅根据代码片段的长度和控制流转移策略去识别配件, 在 GWT 中, 我们同时使用配件类型, 转移策略, 以及代码长度来识别配件。基于配件长度的检测[8- 9]方法的最大缺点是安全性不够, 配件长度和配件链的长度特征过于简单, 攻击者能够比较容易地寻找特定配件来绕过防御。GWT 增加配件类型和配件内容作为配件的判断依据, 利用已有工具(如 Q)来判断是否为功能配件, 利用 *MaxRegMod* 来区分 NOP 配件和普通代码, 比上述粗粒度 CFI 识别配件更加精确, 安全性更高。例如, 一个配件包含多条 NOP 指令、一条运算指令和一条间接跳转指令, 如果 NOP 指令的数量较多, 则基于配件长度的检测方法认为这不是一个配件。而 GWT 根据指令的具体内容, 能够判断这是一个功能配件。粗粒度 CFI 的方案虽然容易实施, 但也容易绕过。先前的一些研究[10-13]通过使用合法配件的 CRA 对抗粗粒度 CFI, 但是在 GWT 中, 我们已经标记了这些合法配件, 例如 call-preceded 配件, entry-point 配件和 NOP 配件, 所以 GWT 能够检测利用合法配件的攻击手段。

细粒度 CFI 则强调 CFG 的重要性, 基于编译器的 CFI 方案[30-33]能够解决间接控制转移的问题, 因为有源码信息的支持, 所以能够生成精确的 CFGs。SafeDispatch[34]通过插桩所有虚函数, 严格检查调用目标是否合法, 以此防御虚函数调用的劫持攻击[21]。前向(Forward-edge)CFI[35]不仅仅保护虚函数, 它通过函数指针分析维护了一个可信代码指针的列表, 如果间接转移目标不在这个表中, 则中断程序执行。

模块化的 CFI 方案[36-37]则关注 CFG 的一部分。CCFI[38]利用消息认证编码(Message Authentication Codes, MACs)加密控制流相关的返回地址, 函数指针和虚表指针。Per-InputCFI[39]根据每次具体的输入内容来完善 CFG。上下文敏感的 CFI 方案[40]提出在二进制级别通过前向和后向的静态分析加强上下文敏感的 CFI 策略。CFIMon[41]提出基于硬件支持的细粒度 CFI 方案, 增加性能的同时也引出了高的检测延迟。

细粒度 CFI 的安全性就在于如何构造出精确且

完善的 CFG, 但是几乎不可能生成包含所有有效控制流路径的理想 CFG。文献[17]提出由于上述细粒度 CFI 的缺陷, 导致基于 CFG 的防御机制出现漏洞。Control Jujustu[16]提出的新型 CRA 攻击便是利用细粒度 CFI 无法生成理想 CFG 的缺陷, 构造基于中转函数的 ACICS 配件, 在不违背 CFG 的前提下完成 CRA。然而, 如果把 ACICS 配件添加到 GWT 框架中, 我们的方法理论上能够防御这类攻击。控制流弯曲[18]表明理想的 CFI 方案也存在一些问题, 利用内存泄露漏洞和非控制数据攻击思想就能够合法调用标准库中的函数, 完成图灵完备的攻击。如果把这种基于函数的特殊配件类型添加到 GWT 框架中, 我们的方法理论上存在检测控制流弯曲攻击的可能。

8 总结

为了防御各式各样的 CRAs, 我们提出 GWT 的方法, 它是一种灵活, 低开销且安全的防御框架。GWT 通过静态分析搜索并标记程序中存在的所有可用配件, 然后在程序运行时动态监控配件的权值标记, 以此计算 CRA 发生的概率。另外我们还提出 GWT 结合粗粒度 CFI 的方法, 相比较基础的 GWT 而言, GWT+CFI 能够更精确发现代码空间中存在的可用配件。我们基于软件和硬件模拟的设计框架来实现 GWT 以及 GWT+CFI 系统, 结果表明其平均性能开销分别为 2.31%和 3.55%。GWT 理论上能够抵御使用各种配件的 CRAs, 特别是使用自动化工具生成配件链的 CRA。另外, GWT 能够添加更多的特殊配件类型, 设计更好的配件识别算法, 来进一步完善和优化该防御框架。

GWT 识别功能配件、NOP 配件和普通代码的方法并不是完美的, 因此攻击者可能会精心构造出被 GWT 识别为 NOP 配件但实际具有一定功能的特殊配件, 或者识别为普通代码但实际属于 NOP 配件的情况, 从而绕过 GWT 的防御。因此, 我们未来会提高 GWT 识配件类型的精准性, 并且完善其它特殊配件(例如基于函数的配件)的寻找算法, 从这两个方面进一步提高 GWT 的安全性。此外, 本文在攻击威胁中存在过多的假设, 主要是因为基于粗粒度 CFI 的思想而不再考虑攻击者使用非对齐配件的情况, 因此导致了完备性方面的不足。未来工作中, 我们会测试非对齐配件对 GWT 的威胁, 完善配件的寻找和识别算法。

致 谢 在此向对本文工作提出指导的各位同门以及提出建议的评审专家表示感谢。

参考文献

[1] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," *in Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, pp. 552-561, 2007.

[2] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented Programming Without Returns," *in Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pp. 559-572, 2010.

[3] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit Hardening Made Easy," *in Proceedings of the 20th USENIX Conference on Security (Usenix Security'11)*, pp. 25-40, 2011.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," *in Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, pp. 340-353, 2005.

[5] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," *in Proceedings of the 51st Design Automation Conference (DAC'14)*, pp. 1-6, June 2014.

[6] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical Control Flow Integrity and Randomization for Binary Executables," *in Proceedings of the 34th IEEE Symposium on Security and Privacy (SP'13*), pp. 559-573, May 2013.

[7] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," *in Proceedings of the 22nd USENIX Conference on Security (Usenix Security'13)*, pp. 337-352, 2013.

[8] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. DENG, Huijie, "ROPecker: A Generic and Practical Approach For Defending Against ROP Attack," *in Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, Feb. 2014.

[9] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing," *in Proceedings of the 22Nd USENIX Conference on Security (Usenix Security'13)*, pp. 447-462, 2013.

[10] E. G¨oktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of Control: Overcoming Control-Flow Integrity," *in Proceedings of the 35th IEEE Symposium on Security and Privacy (SP'14)*, pp. 575-589, 2014.

[11] N. Carlini and D. Wagner, "ROP is Still Dangerous: Breaking Modern Defenses," *in Proceedings of the 23rd USENIX Conference on Security Symposium (Usenix Security'14)*, pp. 385-399, 2014.

[12] E. G¨oktas,, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size Does Matter: Why Using Gadget-chain Length to Prevent Code-reuse Attacks is Hard," *in Proceedings of the 23rd USENIX Conference on Security Symposium (Usenix Security'14)*, pp. 417-432, 2014.

[13] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the Gadgets: On the Ineffectiveness of Coarse-grained Control-flow Integrity Protection," *in Proceedings of the 23rd USENIX Conference on Security Symposium (Usenix Security'14)*, pp. 401-416, 2014.

[14] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented Programming: A New Class of Code-reuse Attack," *in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS'11*), pp. 30-40, 2011.

[15] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow Integrity Principles, Implementations, and Applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 4:1-4:40, Nov. 2009.

[16] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity," *in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15),* pp. 901-913, 2015.

[17] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing Control: On the Effectiveness of Control-Flow Integrity Under Stack Attacks," *in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS15)*, pp. 952-963, 2015.

[18] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control -Flow Bending: On the Effectiveness of Control-Flow Integrity," *in Proceedings of the 24th USENIX Conference on Security (Usenix Security'15)*, 2015.

[19] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," *in Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*, pp. 94-105, June 2012.

[20] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "SCRAP: Architecture for Signature-based Protection from Code Reuse Attacks," *in Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*, pp. 258-269, 2013.

[21] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Diffculty of Preventing Code Reuse Attacks in C++ applications," *in Proceedings of the IEEE Symposium on Security and Privacy (SP'15)*, 2015.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05),* pp. 190-200, 2005.

[23] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News,* vol. 34, no. 4, pp. 1-17, Sep. 2006.

[24] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software," *in Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pp. 339-348, Dec 2006.

[25] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd My Gadgets Go?" *in Proceedings of the IEEE Symposium on Security and Privacy (SP'12)*, pp. 571-585, May 2012.

[26] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. R. Sadeghi, "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization," *in Proceedings of the IEEE Symposium on Security and Privacy (SP'13)*, pp. 574-588, May 2013.

[27] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazi`eres, and D. Boneh, "Hacking Blind," *in Proceedings of the IEEE Symposium on Security and Privacy (SP'14)*, pp. 227-242, 2014.

[28] S. Crane, C. Liebchen, A. Homescu, and L. Davi, "Readactor: Practical Code Randomization Resilient to Memory Disclosure," *in Proceedings of the IEEE Symposium on Security and Privacy (SP'15),* pp. 763-780, 2015.

[29] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, "It's a TRaP: Table Randomization and Protection Against Function-Reuse Attacks," *in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, pp. 243-255, 2015.

[30] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating Code-reuse Attacks with Control-flow Locking," *in Proceedings of the 27th Annual Compute Security Applications Conference (ACSAC'11)*, pp. 353-362, 2011.

[31] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," *in Proceedings of the 35th IEEE Symposium on Security and Privacy (SP'14)*, pp. 292-307, May 2014.

[32] Z. Wang and X. Jiang, "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity," *in Proceedings of the 31st IEEE Symposium on Security and Privacy (SP'10)*, pp. 380-395, May 2010.

[33] B. Zeng, G. Tan, and U. Erlingsson, "Strato: A Retargetable Framework for Low-level Inlined-reference Monitors," *in Proceedings of the 22nd USENIX Conference on Security (Usenix Security'13)*, pp. 369-382, 2013.

[34] D. Jang, Z. Tatlock, and S. Lerner, "SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks," *in Proceedings of the Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[35] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, lfar Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC and LLVM," *Proceedings of the Usenix Security Symposium*, 2014.

[36] B. Niu and G. Tan, "Modular Control-flow Integrity," *in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, pp. 577-587, 2014.

[37] ——, "RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity," *in Proceedings of the ACM Sigsac Conference on Computer and Communications Security (CCS'14)*, pp. 1317-1328, 2014.

[38] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazi`eres, "CCFI: Cryptographically Enforced Control Flow Integrity," *in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, pp. 941-951, 2015.

[39] B. Niu and G. Tan, "Per-Input Control-Flow Integrity," *in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, pp. 914-926, 2015.

[40] V. van der Veen, D. Andriesse, E. G¨oktas,, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI," *in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, pp. 927-940, 2015.

[41] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," *in Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*, pp. 1-12, June 2012.

[42] L. Tong, S. Gang, and M. Dan, "A Survey of Code Reuse Attack and Defense Mechanisms," *Journal of Cyber Security*, vol(2), pp. 15-27, 2016.
(柳童, 史岗, 孟丹, "代码重用攻击与防御机制综述", 信息安全学报, 第 2 期, 15-27, 2016)

马梦雨 于 2014 年在西南民族大学计算机科学与技术学院网络工程专业获得学士学位。现在中国科学院信息工程研究所系统结构专业攻读博士学位。研究领域为计算机系统安全。研究兴趣包括: 内存安全, 软件行为分析等。Email: mamengyu@iie.ac.cn

陈李维 于 2014 年在中国科学院计算技术研究所计算机体系结构专业获得博士学位。现任中国科学院信息安全研究所第五研究室助理研究员。研究领域为计算机系统安全。研究兴趣包括: 信息安全, 计算机架构, 视频解码和 VLSI 设计。Email: chenliwei@iie.ac.cn

史岗 于 2004 年在中国科学院计算技术研究所计算机体系结构专业获得博士学位。现任中国科学院信息工程研究所第五研究室高级工程师。研究领域为计算机系统安全。研究兴趣包括: 计算机架构, 计算机芯片安全等。Email: shigang@iie.ac.cn

孟丹 于 1995 年在哈尔滨工业大学获得计算机体系结构专业博士学位。现任中国科学院信息工程研究所所长。研究领域包括计算机系统安全, 大数据与云计算等。研究兴趣包括计算机系统安全, 云计算安全等。Email: mengdan@iie.ac.cn