中国科学技术大学

专业硕士学位论文

（专业学位类型）



**代码重用攻击防御系统的设计与实现**

作者姓名： 赵鹏辉

专业领域： 软件工程

校内导师： 汪增福 教授

企业导师： 侯锐 研究员

完成时间： 二〇二一年十月二十八日

University of Science and Technology of China

A dissertation for master’s degree

（Professional degree type）



**Design and implementation of code reuse attack defense system**

Zhao Penghui

Software Engineering

Prof. Wang Zengfu

Prof. Hou Rui

October 28, 2021

Author：

Speciality：

Supervisors：

Advisor:

Finished time:

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：\_\_\_\_\_\_\_\_\_\_\_ 签字日期：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

控阅的学位论文在解除后也遵守此规定。

□公开 □控阅（\_\_\_\_年）

作者签名：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 导师签名：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

签字日期：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 签字日期：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# 摘 要

现行计算机编程语言的内存管理存在很多不安全的因素。像C/C++将内存的使用交由程序员处理，这会导致许多与内存相关的安全漏洞，其中控制流劫持是利用内存安全漏洞所实施的一类常见且危害严重的攻击手段。它通过改变程序运行方向而实现攻击，针对这类攻击业界提出了许多相关的防御机制，但是这些防御机制由于自身的局限性和缺点，无法在安全性与性能上达到较好的平衡。如对C/C++语言实现的完全内存安全防御通常会产生较大的性能损耗，而部分地内存防御通常被证明不具有较强的安全保证。

为了解决这个问题一项针对代码重用攻击的防御机制被提出，其中的代码指针完整性（Code Pointer Integrity，CPI）的概念可以在安全性和性能开销之间取得很好的平衡，从而在保证控制流完整性的同时不会造成较大的性能开销。CPI的关键思想是将内存划分为常规区和安全区，并利用安全区来保护与程序控制流相关的数据。在此基础上增加了相关的程序跳转检查，保证了每一个控制流转移的正确性，从而实现对控制流劫持攻击的防御。但是随着相关的深入研究发现，CPI在X86-64架构上利用信息隐藏来隔离安全区的做法会导致较弱的安全性。攻击者可以利用程序中的内存信息泄漏探测到安全区位置，从而容易地对安全区实施攻击。

CPI是一项很有潜力的安全技术，我们在LLVM编译器平台上实现了基于英特尔内存保护扩展（Memory Protection Extensions，MPX）硬件辅助机制的CPI防御模型，同时我们改进安全区隔离方式来防止内存泄漏攻击，对安全区内存页施加一种基于硬件原语的读写保护机制来限制安全区域的读写权限，保证了即使安全区位置泄漏依然可以限制恶意代码的读写，这种强化的安全区隔离方法能够完全阻止对安全区的非法访问。为了验证所提方法的有效性，我们对所提出的相关模型进行了评估和验证。

**关键词：**代码指针完整 控制流劫持 内存保护扩展 信息隐藏

# ABSTRACT

There are many unsafe factors in memory management of current computer programming languages. Leaving the use of memory in the hands of programmers like C/C ++ can lead to many memory-related security vulnerabilities, among which control-flow hijacking is a common and serious attack method to exploit memory security vulnerabilities. It realizes the attack by changing the running direction of the program. Many related defense mechanisms have been put forward in this kind of attack, but these defense mechanisms cannot achieve a good balance between security and performance because of their own limitations and shortcomings. For example, full memory security defense for C/C++ implementations usually results in a high performance loss, while partial memory defense usually proves to be not a strong security guarantee.

To solve this problem, a defense mechanism against code reuse attacks is proposed, in which the concept of Code Pointer Integrity (CPI) can strike a good balance between security and performance costs, so as to ensure the integrity of control flow without causing a large performance cost. The key idea of CPI is to divide memory into regular zones and safe zones, and use the safe zones to protect data related to program control flow. On this basis, related program jump check is added to ensure the correctness of each control flow transfer, so as to realize the defense against control flow hijacking attack. However, further research revealed that CPI's use of information hiding to isolate security zones on x86-64 architecture resulted in weak security. An attacker can exploit memory leaks in a program to detect the location of a security zone, making it easy to attack a security zone.

CPI is a security technology with great potential. We implemented CPI defense model based on Intel MPX hardware assist mechanism on LLVM compiler platform. Meanwhile, we improved security isolation method to prevent memory leak attacks. A read/write protection mechanism based on hardware primitives is applied to the security zone memory pages to limit the read/write permissions of the security zone, ensuring that malicious code can be restricted even if the security zone location is leaked. This enhanced security zone isolation method can completely prevent illegal access to the security zone. In order to verify the

effectiveness of the proposed method, we evaluate and verify the proposed model

**Keywords:** code pointer integrity，control flow hijacking，memory protection extensions，information hiding

目 录

[摘 要 I](#_Toc29816)

[ABSTRACT III](#_Toc6370)

[第1章 绪 论 1](#_Toc29844)

[1.1 研究背景 1](#_Toc9378)

[1.2 国内外发展现状 4](#_Toc12605)

[1.3 本文工作 6](#_Toc17037)

[1.3.1 CPI缺点分析 6](#_Toc229)

[1.3.2 具体工作 8](#_Toc26610)

[1.4 章节内容 9](#_Toc8640)

[第2章 相关技术分析 10](#_Toc26598)

[2.1 LLVM介绍 10](#_Toc14708)

[2.2 CPI原理分析 12](#_Toc7354)

[2.3 英特尔MPX分析 13](#_Toc24394)

[2.4 英特尔MPK的分析 19](#_Toc13796)

[第3章 系统需求分析 21](#_Toc27880)

[3.1 可行性分析 21](#_Toc5715)

[3.1.1 技术可行性 21](#_Toc29150)

[3.1.2 硬件可行性 23](#_Toc20042)

[3.2 功能性分析 23](#_Toc18568)

[3.2.1 功能概述 23](#_Toc21818)

[3.2.2 初始化需求 25](#_Toc32530)

[3.2.3 静态分析需求 25](#_Toc2860)

[3.2.4 动态插桩需求 25](#_Toc7595)

[第4章 系 统 设 计 26](#_Toc1721)

[4.1 设计概述 26](#_Toc23326)

[4.1.1 系统功能模块概述 26](#_Toc7087)

[4.1.2 系统逻辑层次概述 27](#_Toc7023)

[4.2 静态分析模块设计 28](#_Toc10180)

[4.3 动态插桩模块设计 31](#_Toc12339)

[4.3.1 控制流检查 31](#_Toc24080)

[4.3.2 安全区保护锁 33](#_Toc20654)

[第5章 系 统 实 现 35](#_Toc20258)

[5.1 概述 35](#_Toc12302)

[5.2 初始化实现 37](#_Toc6888)

[5.3 静态分析实现 38](#_Toc2606)

[5.4 控制流检查实现 40](#_Toc25293)

[5.5 安全区保护锁实现 45](#_Toc972)

[第6章 系 统 测 试 47](#_Toc1454)

[6.1 测试概述 47](#_Toc24856)

[6.2 测试方案 47](#_Toc15514)

[6.3 正确性分析 48](#_Toc28160)

[6.4 测试数据 49](#_Toc9853)

[第7章 结论与展望 53](#_Toc3500)

[7.1 总结 53](#_Toc540)

[7.2 未来工作 53](#_Toc21251)

[参 考 文 献 54](#_Toc4375)

[致 谢 58](#_Toc21042)

# 第1章 绪 论

## 1.1 研究背景

计算机系统安全是一个日益严重的问题，在利用非内存安全的语言编写（如C/C++）的程序中，允许用户程序完全控制系统内存，这使程序具有很高的运行效率但是也带来许多的内存漏洞问题[1]。内存漏洞通常以时间和空间上的方式出现，空间上的内存漏洞通常是程序运行中从不同于开发人员预期的内存空间中读取和加载数据所导致的，这也被称为缓冲区溢出和内存越界访问漏洞[2,3]。而时间上的漏洞则是程序未在变量对象生命周期使用该对象而造成的内存泄漏（例如悬空指针等）。同时一些内存漏洞还可以用来推测程序的内存布局，例如未初始化的内存变量等[4]，这些违反内存安全的行为可能导致系统崩溃，数据丢失，造成一系列严重的损失[5]。此外这些漏洞还可能被精心设计构成内存攻击，目前控制流劫持[2]是一类较为常见的针对内存漏洞的攻击方法，攻击者通过对程序内存指定数据的覆盖来实现控制流的转移（例如对返回地址），这重写了程序控制流相关的地址，导致程序跳转向非法的内存区域来实现控制流的劫持。攻击者构造特定攻击载体，利用内存中的漏洞改变程序跳转到攻击入口处，继而执行特定的攻击操作达到攻击目的。根据攻击载体的来源，可以将控制流劫持攻击划分为代码注入类攻击和代码重用类攻击两类[6,7,8]。

代码注入攻击是攻击者向程序中注入代码片段，然后通过覆盖栈空间中的返回地址使程序跳转到注入的代码处来实现特定的攻击操作。这是早期的控制流劫持攻击，随着数据执行保护（Data Execution Prevention，DEP）[9]的提出，将进程空间划分为不同的区域，设置每个区域不同的权限，将数据存储区域设置为可写不可执行，代码存储区域设置为可执行但不可写。这使攻击者以溢出缓冲区等形式注入的恶意代码不会再被允许执行，从而抑制了代码注入攻击。DEP是计算机系统较早由比较基础的防御方式，主要针对与代码注入攻击。随着一种新的内存攻击方式代码重用攻击的出现，可以很容易绕开DEP的防御，它不使用注入代码作为攻击载体，而是利用程序中已有的代码片段（称为gadget）来组成攻击载体，这种攻击具有图灵完备性[10]，其中具有代表性的是面向返回的编程攻击（Return Oriented Program，ROP）[10-11]，如图1.1所示ROP的gadget链是以ret结尾的指令片段，ROP扫描程序中的代码找到可以利用的gadget来组成一个完备gadget链，然后通过栈溢出等漏洞覆盖返回地址或者是代码指针等将各个gadget的地址和控制流转移地址相结合，一旦被覆盖的数据产生控制流转移，那么CPU就会按照栈中攻击者存放的gadget地址链进行跳转，不断地从一个gadget返回并跳转到下一个gadget执行，从而完成攻击操作。

另外一种代码重用攻击面向跳转的编程攻击（Jump-oriented Programming，JOP）[12-13]则利用程序中JMP指令为结尾的gadget链来实现。如图1.1所示JOP将所有的gadget地址组织成一张配件地址表（dispatch table），同时通过配件调度（dispather）来实现向每个gadget地址的跳转，dispather类似于eip寄存器可以跳转到指定的地址，同时实现自增操作。每一个gadget最后的jump指令都将跳转到dispather，然后通过dispather继续跳转到下一个gadget的地址来继续gadget链的执行。JOP不依赖与栈空间而是将gadget链保存在任意的可读可写的内存空间中，这使JOP具有很大的灵活性可以很容易地使保护返回地址的防御机制失效。ROP和JOP攻击都需要攻击者预先分析进程的内存布局来找出可以利用的gadget地址，因此当在程序运行开始时利用地址空间随机化会打乱原来确定的gadget链，导致攻击失效。例如地址空间随机化（Address Space Layout Randomization，ASLR）[14-15]等在进程运行前提供防御措施的静态防御技术，可以有效抵御传统的ROP和JOP攻击。

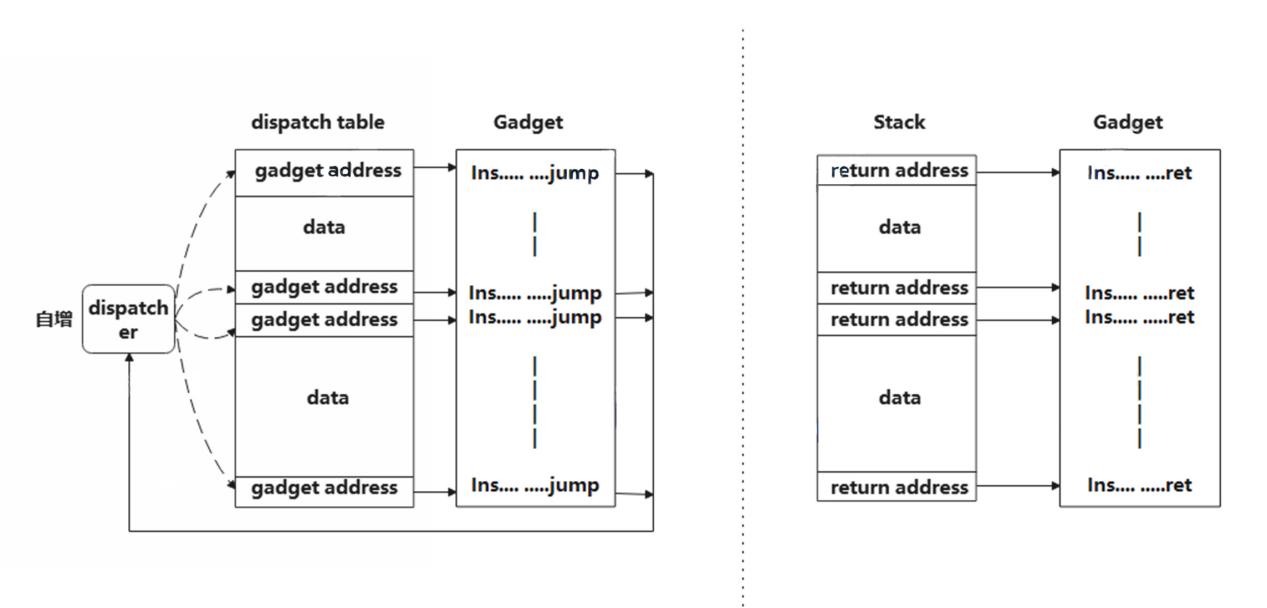


图1.1 JOP和ROP攻击原理（图中左为JOP攻击，右为ROP攻击）

JOP和ROP攻击都是比较普遍的代码控制流劫持攻击，以它们为基础逐渐发展出许多的攻击方式，例如JIT-ROP[16]（Just-in-time ROP）攻击技术，它在运行时动态恢复内存布局，击败了细粒度运行前静态随机化技术。相比于传统的ROP技术。JIT-ROP攻击方法可以绕开运行前的静态随机化防御。JIT-ROP技术在程序运行过程中动态寻找gadget。此外作为JIT-ROP攻击的扩展模式BROP[18]（blind ROP）攻击可以在不获取对方源码的情况下进行远程的攻击。攻击者通过不断地向目标机器发送嗅探信息来获取目标程序运行时的内存布局，从而找到能够作为攻击载体的gadget。这给网络中的计算机系统造成很大的危害。如图1-3所示能够看出一次攻击的具体步骤，完成一次控制流劫持攻击首先需要利用程序中的数据指针漏洞来溢出修改代码指针。根据上述可知这一步可以启动攻击，重定位程序控制流到gadget链的开始位置，然后利用返回地址和jmp地址进行ROP或者是JOP攻击。代码重用攻击是目前较为普遍的攻击方式，从第一次代码重用攻击的出现的开始，已经从早期重定位控制流到一些库函数（例如libc库）[17]执行有限的操作到后来可以跳转任意设计好的内存位置。这主要有两个先决条件（1）对源代码的布局的先验知识以及（2）能够在程序运行中覆盖和控制流转移相关的数据。针对与这两方面防御机制也主要分为两大类：增加代码布局的不可预知和保证程序控制流的完整。

ASLR是基于第一类防御思想的方法，它在程序运行之前随机化程序的代码区，堆栈区，数据区等地址空间，使程序内存重新布局，这否定了进行代码重用攻击的第一个先决条件，程序地址空间的重新布局使原来设置好的gadget的地址不可预测，没有随机化的知识将不会知道新的地址布局。试图发起的重用攻击也会造成程序崩溃。理论上ASLR是有效的防御方法，它具有极高的性能，在现代系统中得到了广泛地使用，但是近些年的研究来看基于随机化的防御方法有一个根本的问题是内存信息的泄漏[16,17,18,19]，ASLR通常不会修改模块中的指令的位置，这就导致随着攻击者不停地嗅探，可以通过程序中的一些漏洞重新推测出整个程序的内存布局，这就使运行前的随机化失去作用。ASLR拥有极为突出的弱点和优点，目前很多研究都是以ASLR为基础做改进。另外一类防御是保证程序的控制流完整性，主要思想是保证程序的每一次间接跳转都有正确的目的地址，其中以控制流完整性（Control-flow Integrity,CFI）[20]思想为代表，CFI首先利用静态分析来画出控制流图（CFG），然后计算出所有的间接转移指令的可能的目标地址范围。为程序中的控制流转移指令添加检查逻辑来判断当程序跳转的时候是否在指定的范围之内。如果不再规定范围则中断程序，这种为每一个控制流相关分配地址范围的方法被称为细粒度的CFI。它保证了每个控制流跳转的正确性，CFI具有强大的安全性，但是纯软件的实现也造成了较高的性能损耗，细粒度的CFI的性能损耗在20%以上[20]这导致其不能很好地应用到实际系统中，同时如果需要保护的代码量巨大，很精确地画出其控制流程图将是一个很大的问题。对于细粒度CFI粗粒度的实现则具有更低的开销，允许控制流跳转到更大的目标范围，或者不要求精确的静态控制流图，不幸的是这些实现普遍具有较弱的安全性也都渐渐被业界证明为无效的防御而逐渐淘汰。

代码指针完整（Code Pointer Integrity，CPI）是一种较新的防御思想，它属于控制流完整性一类的防御。区别于CFI为每个控制流转移设置白名单和ASLR打乱内存布局的方法，其核心思想是使将内存分为安全区域和常规区域。安全区域用来保护敏感数据（例如代码指针，返回地址），常规区域则保存其他程序数据，CPI主要针对代码指针进行保护，这类数据直接影响程序控制流的转移。当攻击者通过修改代码指针可以在程序调用函数时跳转到错误的地方，而CPI则将代码指针这类数据集中放在安全区域，然后利用内存信息隐藏来隔离该区域，同时增加对应的检查机制在每个指针使用时检查正确性，这种防御能够在攻击启动阶段就停止攻击，提供了强大的安全性保证。我们在较新版本的LLVM编译器平台上实现基于CPI思想的防御机制，同时分析了原有CPI实现所具有的缺点，以及可能遭受的攻击方法，提出了相应的解决方案，在原有的实现基础增加了安全区的隔离性。我们将在本章最后一节阐述CPI的不足以及我们的主要工作。

## 1.2 国内外发展现状

在计算机发展的过程中，对内存漏洞的利用产生了许多不同的方法，从最初的代码注入攻击，栈溢出攻击到现在的代码重用攻击，侧信道攻击[19]等，内存漏洞攻击早在70年代[23]就开始使用，并且现在仍然有较大的威胁[24]，完全的内存安全技术如用CETS[25]扩展的SoftBound和利用硬件实现的边界检查机制MPX[21]都具有非常大的开销（通常有2-4倍的速度下降），CCured[26]和Cyclone[27]技术被提出来用于提供指针的空间上的安全性，但是与现有的C代码不兼容。其他的方法像Cling[28]，Memcheck[29]，LBC[30]和AddressSanitizer[31]仅提供临时的指针安全来保护悬空指针（use-after-free指针）除了上述的一些方法，迄今为止，也已经提出了多种许多的防御机制来应对日益增长的控制流劫持攻击的挑战。根据上述所知主要有ASLR随机化技术和CFI控制流完整性技术。

基于控制流完整性的防御代表是CFI，最早由Martin等[20]提出，CFI是具有较强保护性的防御机制，能够抵御动态的JIT-ROP技术，随着对CFI的研究也逐渐演化成细粒度的CFI和粗粒度的CFI。CCFI[32]技术会为程序中每一个影响控制流的对象生成一个识别码，在控制流跳转之前会检查识别码来防止运行时控制流不被修改。KCoFI[33]技术在CFI保证控制流完整的前提下，在操作系统内核添加了对进程的动态监视功能，用来保护一些关键数据。细粒度的CFI会进行较为精细的控制流检查，这会导致较大的性能开销，所以粗粒度的CFI被提出。例如，CCFIR[34]技术会收集所有合法的跳转指令目标地址将其放置在一块独立的内存区中，规定所有的跳转都要通过这块内存区。binCFI[35]技术将间接转移指令进行分类，同一类转移指令归纳到同一目标集合中，不对每一条指令做检查，这提高了检查效率。粗粒度CFI不再对间接跳转地址做细致的区分，这就导致了攻击者可以利用比较特殊的gadget链来实现攻击，文献[36-37]证明了粗粒度的CFI可以被完全地绕开，因此粗粒度的CFI也逐渐被业界所淘汰。

基于地址随机化的防御主要是ASLR方法以及细粒度与粗粒度的变体，不同的ASLR技术在程序加载时会随机化堆栈，代码，链接库位置，由于ASLR具有很明显的优点和缺点，目前许多的方法都是对ASLR做改进，Pax ASLR[14]是比较早的随机化技术，它会在可执行程序加载时随机化动态库，堆栈等地址，还能加载代码段到不同的内存位置，这打乱了原来攻击者的gadget地址，使攻击无效。Pax ASLR是粗粒度的随机化技术同时随机熵较低，很容易被信息泄漏或者暴力方法攻破。细粒度的ASLR往往会随机化程序中的对象地址，具有更大的随机熵，这增加了攻击者探测内存布局的难度。上述方法属于静态随机化技术，很容易遭到动态了内存信息泄漏攻击。例如JIT-ROP攻击是一类动态ROP技术，在程序运行时利用内存信息泄露重新推断内存布局，这被证明很容易绕开静态的ASLR。针对这类攻击逐渐产生了一种运行时重随机化的技术。

ASLR-Guard[38]是传统ASLR的改进，它重写二进制文件将代码段与数据段的关联解耦，同时修改链接器增加对程序中可能泄露代码段位置的数据的保护（例如GOT表等）。Giuffrida[39]等通过高效细粒度的随机化技术和运行时重随机化思想来提高操作系统的安全性。Chen[40]等提出的JIT-ASLR技术利用虚拟内存管理在运行时通过修改代码页的页表来实现地址空间的改变。Remix[41]技术是一种基于基本块的随机化技术，利用LLVM编译器在编译程序时重新排列每个函数中的基本块来实现随机化。同时对基本块进行基于时间的重随机化。这些技术都需要修改内核模块来实现功能，兼容性较差。Hawkins[42]等提出的Mixr运行时再随机化系统可以在不依赖源代码以及不修改系统内核和加载器等组件的基础上进行，然而该技术由于在运行时需要反复进行动态链接导致性能开销过大。TASR[43]技术将随机化与潜在的运行时信息泄露相结合，规定成对出现的输入/输出操作为保护操作，如果程序中出现这类操作则触发随机化。运行时的随机化减少了基于定期时间的随机化产生的开销。文献[44]缩小了TASR技术对危险操作的界定范围，规定了只有对内存的存储和读取才触发随机化操作。Morton[45]等提出了当攻击者公开了所谓的“僵尸”gadget时，才会进行随机化防御，这具有极小的性能开销。这些基于风险的随机化操作减少了固定时间随机化所带来不必要的随机化操作，大部分来讲要优于基于时间的随机化防御。

由于程序语言的非安全性，内存漏洞将会一直伴随着计算机的发展，相对应的防御方式也越来越多，同时又促进着攻击人员研究新的攻击方式，这是一个相互羁绊的过程。软件系统的漏洞造成的损失时不可预计的，随着计算机在各个领域的普及，在未来系统的安全性保证将会越来越重要，在致力于安全防护研究的同时，也需要从根源提高程序开发者以及软件使用者的安全和道德意识。

## 1.3 本文工作

内存漏洞依然是C/C++语言程序安全的主要问题，这类语言因为更加偏向底层硬件拥有直接控制内存的能力，所以执行时具有非常高的效率，但是也为此付出巨大的安全代价。现如今代码重用攻击依然是针对内存漏洞的主流攻击方式，针对这类攻击有许多的方案可以在C/C++语言程序中加强内存安全。而我们目的是找到一种能够防御代码重用攻击保证控制流安全的同时又不会带来较高的性能损耗的防御方法部署到C/C++的编译器平台上，以帮助程序在编译时为其提供安全保障。CPI是一种很好的防御思想，根据作者的证明[46]它能够防御住不同的代码重用攻击，同时也具有较低的性能开销。我们根据文献[22]的研究分析了这种防御机制所具有的缺点，以及利用该缺点进行的攻击方式，以此为我们所做具体工作提供理论基础，同时也提出了一系列新的实现改进方案。

### 1.3.1 CPI缺点分析

对于CPI我们首先需要明白，CPI关键思想使用新开辟的内存区存储控制流相关的敏感数据，这将程序中的控制流数据集中到一起存储。如果内存中未提供强大的隔离技术，则很容易对该内存区域进行非法修改[47]，这会导致严重的后果。所以顾名思义只有在与普通内存区域隔离的前提下才能被当作CPI安全区使用，而这也恰好是CPI原有实现的缺点：CPI对安全区内存的隔离性不够。在没有硬件段保护的架构平台上如X86-64，CPI仅使用信息隐藏的技术来实现安全区的隔离（即随机分配安全区内存的首地址）。CPI作者断言[46]任何对48位地址的暴力推测是不可能的事情，这通常会被程序中的其他异常检测机制检测到，但是事实上这种断言是不正确的，文献[22]证明通过内存中的信息泄露可以在不导致程序崩溃的前提下推断出内存的布局。同时CPI作者认为在常规区域没有直接指向安全区域的地址，两者是相互独立的内存区，所以不会通过常规区域的信息暴露安全区域地址，这是一个错误的断言。CPI保护对象范围处于整个内存空间，因为其常规区和安全区地址一一映射，造成了安全区内存会非常地大，CPI的安全区如果需要保护所有的程序空间通常需要占据一半的进程内存。因此这就表示攻击者可以很容易修改其他区域（如栈空间数据指针）定位到CPI安全区之内，而不需要任何直接指向安全区的指针。

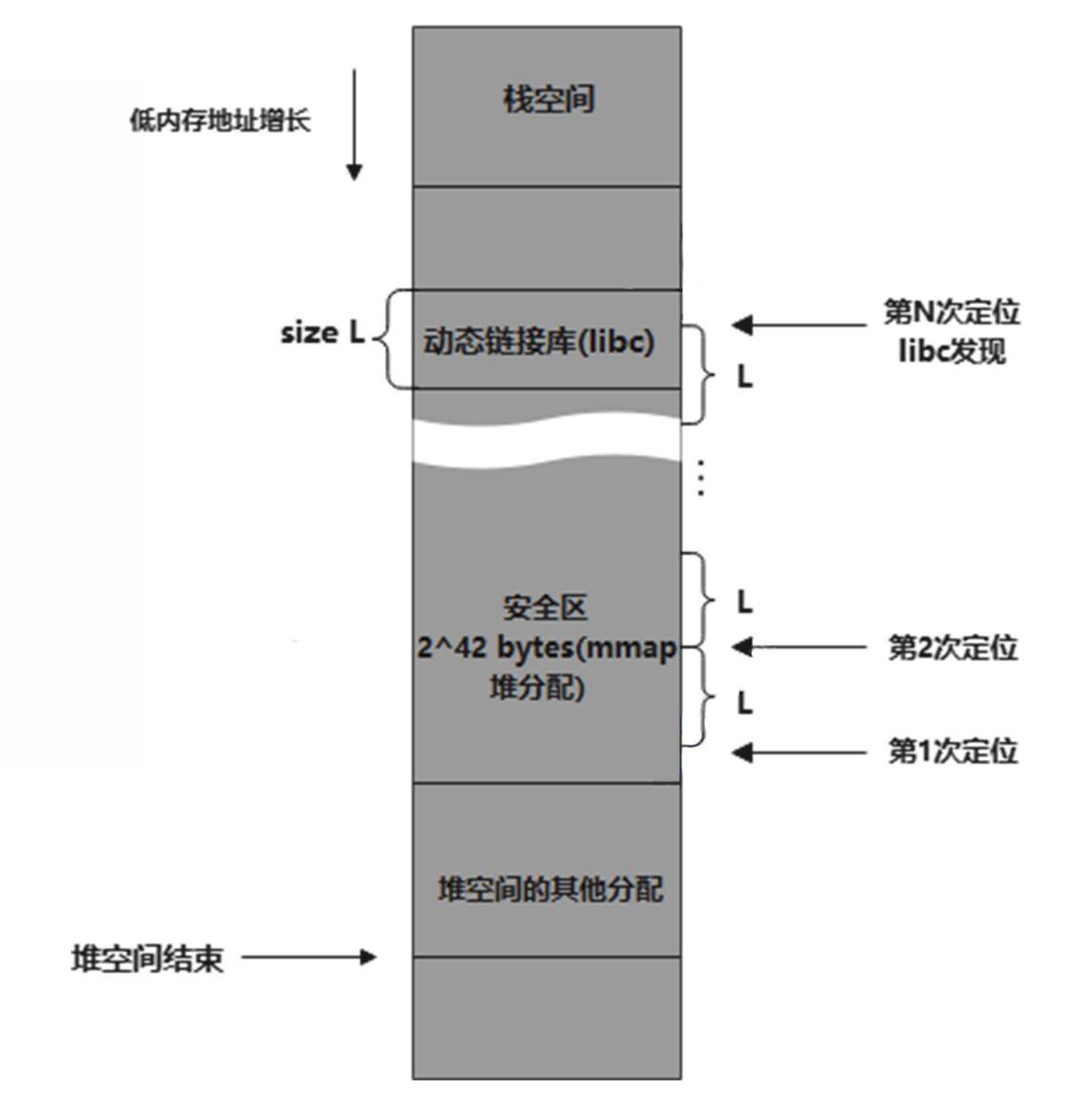


图1.2 安全区攻击流程

另外一方面CPI为了保证较低的性能损失，将保护重点放在代码指针上，这就忽略了其他一些可以利用的内存漏洞（如数据指针等）。根据CPI的威胁模型来看，攻击者可以劫持程序的栈空间，通过数据指针的溢出不断的探测安全区的位置。文献[22]证明即使在有ASLR随机化的保护下，依然可以通过对数据指针的覆盖在不导致程序崩溃的前提下推断出内存的布局，同时提出了一种侧信道攻击方法，通过不断覆盖堆栈上的数据指针随机定位到安全区内的位置来最终推断出安全区域的基地址，继而改变安全区的控制流数据启动ROP攻击。如图1.2是针对CPI安全区攻击的一个简易原型。安全区的内存分配通过mmap调用且总是位于程序连接库（libc）之下，而且安全区具有2^42byte的内存大小。通过对栈空间数据指针的覆盖可以修改内存地址溢出到其他内存区域，继而对指针解引用来确定是否定位到安全区之内，这是由于代码指针相对与整个程序数据只是很少的一部分，所以CPI安全区是稀疏的大部分位置为0，所以可以通过指针解引用的值来区分安全区和链接库。我们在每次试探之后减少链接库的大小L再次试探，这样在经过多次的试探之后，指针最终会定位到链接库之内。通过多次的试探能够推测出链接库的具体位置，在得到链接库的基地址之后通过减去安全区大小可以直接得到安全区基地址。继而对安全区实施攻击以实现控制流劫持。由上可以得出，CPI基于信息隐藏的安全区隔离很容易通过类似的方法遭到攻击，从而将安全区的内存布局暴露在攻击者面前。CPI这种弱内存隔离成为攻击者可以利用的安全漏洞，大大的增加了程序的不安全性，一旦被攻击者利用将导致整个防御机制失效。

### 1.3.2 具体工作

针对上述我们发现对安全区的任意读写是导致攻击实现的必要条件。如果我们限制安全区的非法读写则能够保证即使泄漏安全区的位置也无法进一步改写其中内容，可以杜绝一系列针对安全区的攻击方法。我们希望只有通过特殊的安全区访问指令才能够存取安全区，而不能通过缓冲区溢出或者上述数据指针溢出等方法来修改安全区内容，对该区域的任何非法的访问都会造成程序的崩溃。我们为安全区内存页设置访问权限，规定在程序不访问安全区时设置该区域为不可读写，只有通过我们的安全区访问指令才能够访问安全区。保证除了使用特殊的内存访问指令外不能够对安全区进行访问。阻断了通过非法指令或者其他非常规安全区内存访问的可能性。我们首先移植了原有CPI到新的LLVM平台之上，同时使用英特尔内存保护锁[48]（Memory Protect Keys，MPK）机制来实现我们的安全强化方案。这是一种基于硬件原语实现的内存保护机制，可以设置内存页的读写权限，同时能够高效率的切换权限，不同于DEP保护对内存页的设置，MPK增加了特殊硬件寄存器辅助，可以同时对内存中一组页进行保护。我们在分配安全区的同时为该区域分配一个MPK保护锁用来控制其读写权限，同时修改原有实现增加特定的安全区访问指令提供给用户进行访问安全区，以区别非法的访问。我们在加强安全区保护的同时也改变了原有CPI的整体实现方法。

对于CPI的原有的软件实现方式，同时也具有一些安全区的内存组织问题，像安全区过大，无法动态的回收已经分配的空间，将对安全区的管理交给用户也会造成程序的复杂和一定的安全隐患。在我们的工作中使用硬件辅助的方法实现CPI主要部分。利用了英特尔MPX[21]（MPX是英特尔发布的一个新的ISA扩展）在X86体系结构上增加的一些硬件，我们修改了原有软件实现，将对安全区的读写，检查转换为包含MPX汇编指令的特殊代码。对于常规区域与安全区地址的映射计算，以及安全区的实现与管理，我们将其交给MPX底层指令和操作系统去做，这样保证了向上对用户空间的隔离。我们利用了MPX提供的边界存储表来充当安全区，由于此表是由操作系统管理，操作系统按照需要动态的分配和释放存储表，这提高了内存空间的利用率，节省了内存空间，操作系统也会根据程序中变量的释放来回收多余的内存边界表，提高了安全区的稀疏性，这使针对安全区的探测攻击更加地困难。我们将在后面章节分析MPX边界表带来的优化。对于上述功能与改进我们将在后面章节具体叙述我们的实现模型，以及评估具体的性能损耗。

## 1.4 章节内容

我们在接下来的第二章介绍了具体的实现平台，CPI的思想原理，以及改进所需要的相关机制原理，在第三章我们做了需求分析来阐述我们实现的可行性与功能性需求，第四章是系统的概要设计。第五章介绍了具体的实现，第六章给出具体数据评估了我们的模型。第七章做总结同时提出了对未来的改进意见。

# 第2章 相关技术分析

## 2.1 LLVM介绍

我们在LLVM-10编译器平台实现我们的系统。LLVM以clang为前端解析C/C++代码生成中间语言IR，进一步链接翻译成后端汇编语言。相比于GCC等其他编译器前后段耦合性更低，灵活性更高。中间语言IR是LLVM的设计核心，它是一种通用中间语言，不依赖于具体的后端机器，可支持不同的目标机器语言翻译。LLVM由前端，中间语言，后端三大部分组成。图2.1表示LLVM整体框架以及IR结构关系。

前端：主要是LLVM的子项目Clang组成，源代码在经过词法分析器，语法分析器，语义分析器之后生成中间语言IR。Clang实现了所有与前端相关的步骤。

中间语言IR：LLVM优化器opt可以处理IR程序，来优化代码，同时也可以将多个IR文件链接成一个IR文件继而进行优化。

后端：主要负责代码的生成，他将IR语言转换成特定与目标机器的二进制文件，对程序指令选择，指令调度，寄存器的分配。

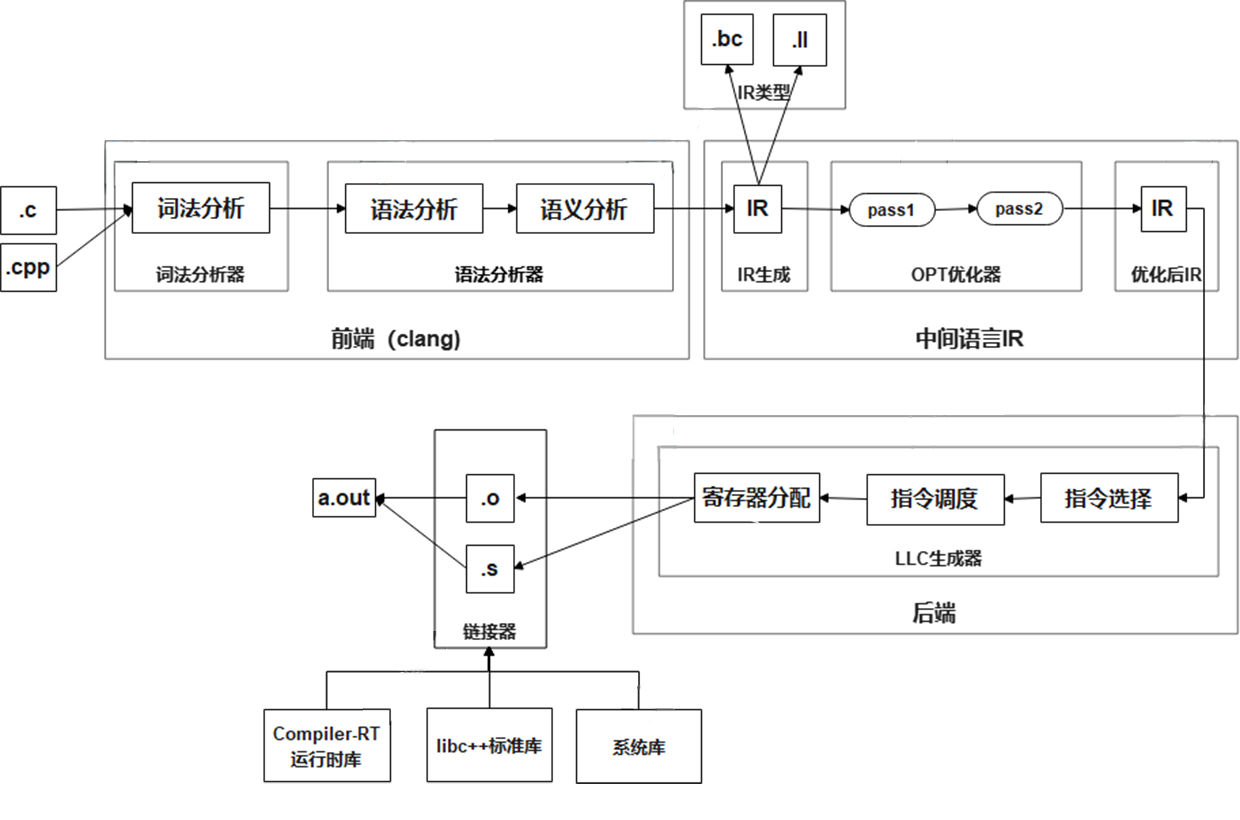


图2.1 LLVM整体框架

IR语言：我们主要工作是对程序中间语言优化，LLVM IR主要有两种表现方式，一种是机器更好理解的bitcode位码的形式以（.bc）结尾，一种是方便开发人员理解的汇编文本形式以（.ll）形式结尾，汇编形式的IR是人类能够理解的中间语言，图2-2展示具体的.ll语言代码。中间语言IR携带了全部的前端语言（C/C++）变量的类型信息，为程序的静态分析提供支持，编译时每个源文件被翻译成一个模板，模板是IR的顶层结构，每个模板中含有一系列函数，每个函数由包含一系列指令的一系列基本块组成，模板还包含用于支持该模板的相关信息，如全局变量，目标机器数据布局，外部函数声明，以及一些用来描述模板指令的元数据等。IR使用静态单赋值（SSA）形式，每个变量有唯一一次赋值操作，不会在之后重新赋值，每次使用一个变量就可以通过LLVM提供的接口找到该变量的定义位置，SSA形式定义了每个变量的使用定义链（use-def）这就为每个变量的数据类型分析提供了支持，它是我们优化代码做静态分析的前提条件。

|  |
| --- |
| 1. ***//int global=0；*** 2. **@global = global i32 0, align 4** 3. **@.str = private unnamed\_addr constant [3 x i8] c"%d\00", align 1** 4. **; Function Attrs: no unwind uwtable** 5. **define i32 @main() #0 {** 6. **%a = alloca i32, align 4** 7. ***//int a=10;*** 8. **store i32 10, i32\* %a, align 4** 9. **%2 = load i32, i32\* %a, align 4** 10. ***//printf("%d",a);*** 11. **%3 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([3 x i8], [3 x i8]\* @.str, i32 0, i32 0), i32 %2)** 12. **ret i32 0** 13. **}** |

图2-2 IR语言（图中load和store语句为内存加载存储语句）

IR元数据：IR元数据是用来描述数据的数据，是模板的一部分，通常被附加到指令和全局变量上用来向优化器和代码生成器提供信息，例如调试信息，模板描述信息，变量类型别名分析等，这帮助我们更好的判断变量的类型，以完成对程序的静态分析。

由图2.1可以看出，优化器opt能够对IR进行优化，它实际上是由许多的pass优化模块组成，每个模块负责一类优化，当程序在IR阶段时，优化器负责遍历程序，将需要的pass优化模块作用在每个IR模板上来完成优化操作，也可以利用llvm-link链接器将不同的IR模块链接成同一个文件，进行链接时优化。我们需要增加自己的优化模块到opt中来完成我们工作，这主要包括两大部分：源程序的静态分析，以及动态的插桩。我们首先需要对源程序进行静态分析这区别于CPI的分析，我们在CPI的基础上增加了敏感数据的定义范围，进一步减少了利用信息泄漏推断安全区位置的风险。在动态方面我们利用MPX来实现敏感数据的存储，加载，边界的检查等插桩操作。

## 2.2 CPI原理分析

CPI静态分析：CPI基于IR语言做变量类型分析来确定要保护的敏感数据范围，规定代码指针，指向代码指针的指针，包含一个或多个敏感类型成员的复合类型的指针（例如struct或array），通用指针（void \*，unsigned char \*指针可能在运行过程中指向敏感数据）为敏感数据，敏感数据的定义具有传递性，如图2.3我们保护结构体中的指针3和4，同时指向结构体的指针1和2也在保护范围。静态分析搜索所有变量的类型，为确定的敏感数据创建相关的元数据，这主要包括指针地址的上下边界，指针的值，将每个元数据存储到指针地址相对应的安全区内，以供加载时做正确性检查。静态分析中常常会产生一类动态数据，这主要是通用指针在运行过程中的转换，这无法根据变量的静态类型来判断。CPI利用程序指令流分析通用指针是否在解引用前指向函数指针，或者在以后被强制转换为函数指针使用。当然静态分析并不是绝对的准确，程序中跨函数之间的类型分析就无法实现。

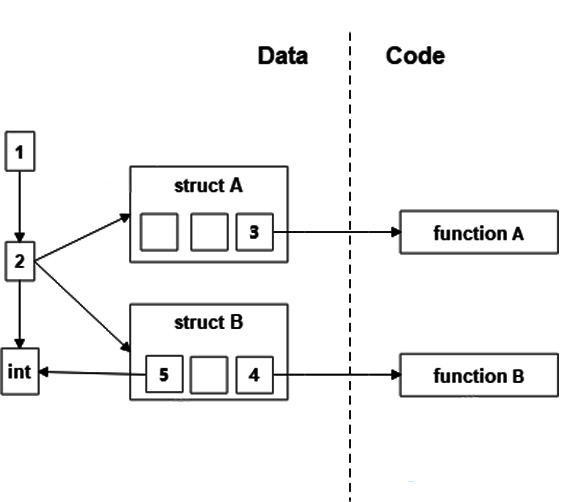


图2.3 指针关系依赖

CPI动态检查：在确定具体敏感数据之后，CPI首先创建该指针的元数据，然后存储元数据到指定的安全区域，CPI预先在内存中分配好一块足够大的内存区域当作安全区，为常规区和安全区的内存地址建立映射关系，如图2.4所示，函数指针在常规区域中的地址通过指定的转换找到对应的安全区地址。CPI修改源程序中每一个敏感数据的存储加载指令为对应的安全区加载存储指令。以将代码指针之类的元数据存储到安全区之中。CPI中栈被单独划分，CPI利用安全栈机制来保护返回地址，分离了栈空间数据和返回地址，保证返回地址不会因为其他数据的溢出而被覆盖，这很好地保护了控制流同时性能损耗接近0[46]。CPI也增加了检查操作，用来从安全区加载存储数据时检查对应代码指针的元数据以及相应地址的正确性。

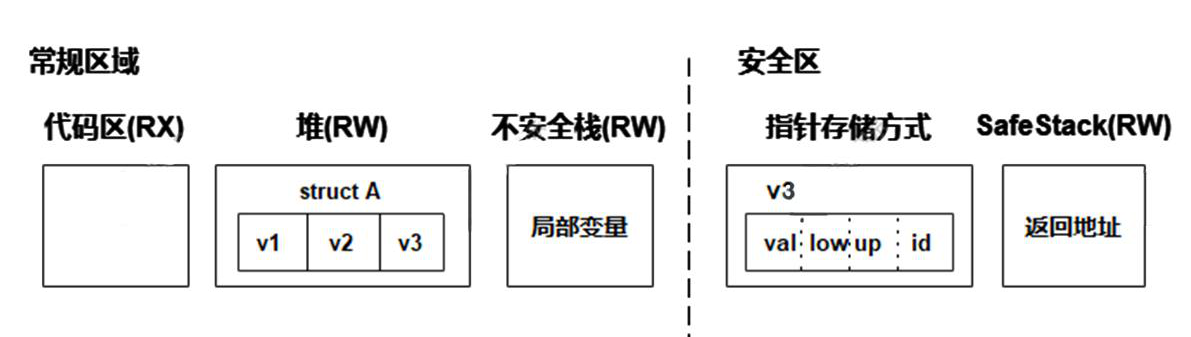


图2.4 CPI内存布局

CPI安全区隔离：安全区存储了大量的函数指针值以及元数据，这需要进行隔离操作以保证安全区不会被常规区域非法访问。在x86-32系统上CPI依赖硬件段保护机制来隔离安全区，但是x86-32系统逐渐被淘汰。在x86-64上，CPI依赖于以下事实：常规区域中永远不会存储指向安全区域的地址。CPI使用段寄存器GS存储安全区域基地址。安全区域被随机化分配，并且通过在常规区域中对象的地址来索引安全区的存储位置，因此在执行过程中常规区域不会有直接指向安全区的指针。同时现代x86-64 CPU的48位地址空间使猜测安全区域地址变得不切实际，大多数的暴力猜测会使程序崩溃。但是根据上述可知，这是薄弱的安全区隔离技术，是CPI所具有的安全隐患，也被证实可以利用内存信息泄漏攻击。

## 2.3 英特尔MPX分析

MPX是集成在英特尔Skylake处理器架构之上利用硬件辅助的边界检查机制，为C/C++语言提供内存安全保证，基于动态的边界检查技术是用来防止内存的边界溢出攻击的普遍方法，但是大多数的基于完全内存安全的边界检查都有50%-150%的性能开销[21]。这限制了它们的使用。MPX通过增加特定的指令和寄存器的为内存边界检查提供帮助使其性能上得到提升。指针的边界检查用来防止指针的内存越界访问，这会导致内存漏洞。如图2.5所示我们能看到一个程序中数组可能引起的访问越界以及需要的边界检查操作。图中能够清晰地看到内存边界检查的作用，这与上文提到的CPI的边界检查原理相同，我们可以看到在代码中一共进行了两次的边界检查，第一次是对数组元素a[i]的检查，通过变量i的访问，每个元素都需要检查上下界来防止变量访问越界，这里的边界值是数组a的低地址和高地址，第二次边界检查是在加载结构体中的len变量时所做的检查，由图2.5可看出在加载a[i]的地址时会对应的将a[i]值的边界加载出来，这个边界是a[i]所指向的结构体元素的上下边界值，当我们访问结构体中的元素len时会对应的检查len地址是否处在上下边界之中。

|  |
| --- |
| 1. **struct obj { char buf[100]; int len }** 2. **obj\* a[10]** 3. ***//创建边界值（指针在内存中8bit）*** 4. **{a,a+79}** 5. **total = 0** 6. **for (i=0; i<M; i++):** 7. **ai = a + i** 8. ***//插入指令检查a[i]的下边界*** 9. **if(ai < a) abort();** 10. ***//插入指令检查a[i]的上边界*** 11. **if(ai+7 >a+79)abort();** 12. ***//加载a[i]值*** 13. **objptr = load ai** 14. ***//插入指令加载a[i]的边界值*** 15. **{low ,up} = load bound** 16. **lenptr = objptr + 100** 17. ***//插入指令检查obj.len的下边界*** 18. **if(low > lenptr)abort();** 19. ***//插入指令检查obj.len的上边界*** 20. **if(up < lenptr+3)abort();** 21. **len = load lenptr** |

图2.5 MPX边界检查原理伪代码

MPX硬件辅助：MPX在x86架构上添加了特殊功能的指令集。每一个操作都有相应的指令实现，同时也增加了寄存器来辅助操作。MPX增加了4个边界寄存器bnd0-bnd3，每个寄存器128位用来存储64位的上下边界值，两个配置寄存器BNDCFGS/BNDCFGU（用户模式和内核模式各一个）和一个状态寄存器BNDSTATUS，配置寄存器用来使能内核和用户模式下对MPX的支持，存放边界目录表的基础地址值（MPX采用二级页表形式来存储边界值）。状态寄存器存储当前的程序引起#BR异常的原因，以及bndstx存储失败时的边界目录条目的入口地址（这个页目录是MPX二级页表的页目录，接下来将详细阐述）。

MPX增加了7个新的指令如表2.1所示，其中bndmk用来创建上下边界值，并存储到128位的bnd边界寄存器中。bndldx和bndstx指令分别用来从边界表中加载边界值到bnd寄存器中和存储bnd寄存器中的边界值到边界表，它们会通过特殊的地址映射方法来计算存储边界值的地址，然后加载/存储上下边界值，每当bndstx存储失败时会将失败原因记录在BNDSTATUS状态寄存器中，同时也会将存储失败的边界目录地址记录在其中这方便在重新分配内存空间时直接使用，下一节将具体介绍地址转换方法。bndcl和bndcu/bndcn指令用来检查边界，判断内存地址是否处在bnd寄存器中的上下边界中，这代替了CPI软件实现的判断语句。bndcl比较源操作数中的地址和bnd寄存器的低64位。而bndcu比较高64位。bndcn是bndcu的补码版本，经过测试比较指令能节省一个指令周期的时间同时减少了判断语句的分支预测压力[21]。每当边界检查错误时状态寄存器BNDSTATUS的低两位会记录引起的错误类型。bndmov指令用来传送内存地址的128位数据到bnd寄存器中或者从寄存器传送数据到内存（注意操作数不能使用普通的x86寄存器）。

表2.1 MPX指令扩展

| 指令 | 描述 |
| --- | --- |
| BNDMK bnd, m64 | 从m64处创建上下边界并且存储到bnd寄存器中 |
| BNDMOV bnd, m64/bnd/m128 | 传送值从边界寄存器bnd到另外的bnd寄存器，128位内存或者64位内存中 |
| BNDMOV m64/bnd/m128, bnd | 传送值从bnd寄存器，128位内存或者64位内存到bnd寄存器中 |
| BNDLDX bnd, mib | 通过地址的转换加载内存中的上下边界值到bnd寄存器中 |
| BNDSTX mib, bnd | 通过地址转换存储bnd中的边界值到内存中 |

续表2.1 MPX指令扩展

|  |  |
| --- | --- |
| 指令 | 描述 |
| BNDCL bnd, r/m64 | 比较m64或者寄存器r中的值和bnd寄存器的下边界，如果小于则产生#BR异常 |
| BNDCU/N bnd, r/m64 | 比较m64的值和bnd的上边界，如果大于则产生#BR异常，BNDCN是BNDCU的补码版本 |

注：表中m64表示8byte的内存，m128表示16byte的内存，mib表示寄存器组合

MPX操作系统：MPX修改内核代码引入了一个新的异常#BR来帮助在程序引起边界错误时做异常处理，同时管理维护了一个二级表结构的边界存储区。这将一系列的复杂操作交给操作系统实现，更加方便同时减少了出错风险。操作系统增加了#BR异常，和其他系统中的异常一样，#BR异常有相应的异常处理程序，有两种情况会引起该异常：(i)bndcl，bndcu和bndcn在比较边界时失败时，(ii)bndstx当向边界表存储边界时，如果边界目录是无效的表明当前地址没有分配会产生#BR异常，从而向系统申请新的空间。当异常恢复时会重新存储边界值。如果是由边界检查错误而引起的#BR异常，处理器会陷入内核，内核会得到错误的边界值和地址并将它们存储在一个名叫siginfo结构中，然后它将SIGSEGV信号连同siginfo中的违规信息一同传递给应用程序，此时用户有一个选择来处理，他可以提供一个特别的信号来恢复程序，也可以选择默认处理策略：崩溃，打印错误，或者是选择忽略它。如果是第二种情况而引起的#BR异常，则由操作系统控制调用具体的异常处理程序来分配新的边界表，然后恢复异常并重新存储边界值。

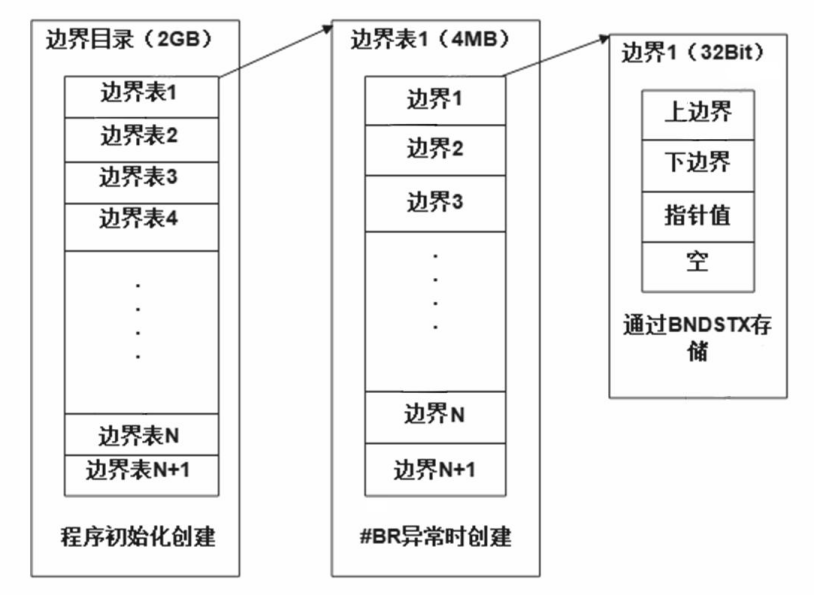


图2.6 MPX边界表组织方式

在内存中MPX提供了4个寄存器用来存储边界值，这在程序中很明显不够存储指针边界值，如果超过4个需要存储边界值的指针就会寄存器溢出，于是MPX引入了额外的内存空间来存储边界值，同时MPX引入了两条指令bndstx和bndldx负责专门存储边界值和加载边界值。从图2.6可以看到二级页表的组织方式，边界目录中每一项存储不同边界表的基地址，当边界目录在初始化创建后所有的边界目录项都被初始化invalid（值二进制位最后一位为0）表示没有边界表分配。

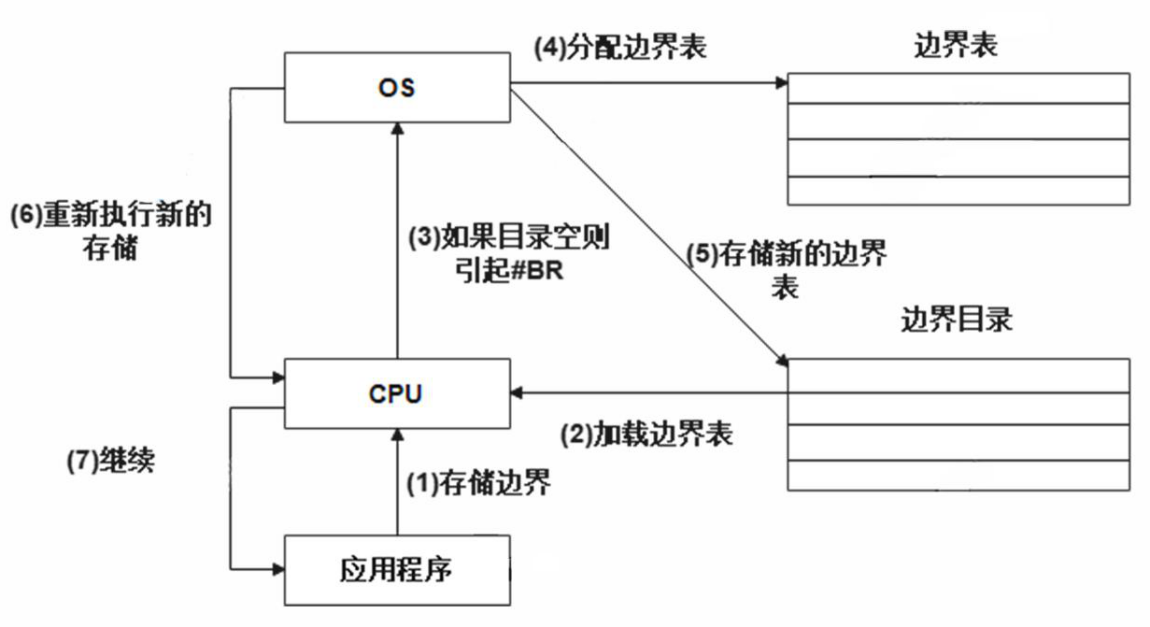


图2.7 边界异常引起与恢复过程

边界表的创建则是在运行过程中动态创建的。当程序中需要利用bndstx存储边界值时会引起#BR异常，系统会分配新的边界表并把基地值存储到对应边界目录项中，同时把目录内容改变为valid（值的二进制位最后一位置为1）。然后会回到存储边界的地方重新执行边界存储。当使用bndstx和bndldx指令时利用了类似操作系统中虚拟页表机制的地址转换方案，每个指针在边界表中有且仅有唯一的条目。请注意MPX的分页只是概念性的；实现方面有很大的不同。首先MMU不参与翻译，所有操作都由CPU自己执行。其次，也是最重要的一点，英特尔MPX没有专用缓存（如TLB缓存），因此它必须与应用程序数据共享普通缓存。在某些情况下，这可能会导致由缓存抖动引起的严重性能下降。如图2.7所示能够看出边界存储异常的产生与恢复过程。

指针边界的存储需要指针地址进行转换找到唯一的边界存储地址，地址的转换是一个多阶段的过程。如图2.8所示这是一个复杂的过程，会造成很高的操作成本，大约需要3次寄存器到寄存器的移动、3次移位和2次内存加载。不过相对于软件的实现，MPX把操作交给CPU去完成在速度上会有很大的提升，具体转换过程如下：

（1）从指针地址的20–47位提取边界目录条目的偏移量，并将其向左移动3位（因为每个边界目录条目都是8Bit大小）。

（2）从BNDCFGx寄存器加载边界目录的基地址。

（3）将基址和偏移量相加，并从结果地址加载边界表的基地址。

（4）CPU从指针地址地第3-19位提取边界表条目的偏移量，并将其向左移动5位（因为所有边界表条目都是32Bit大小）。

（5）将第一阶段加载的边界表基地址和偏移量相加得到值的存放地址。

（6）加载上下边界和指针值。

对MPX的支持需要从硬件，指令集，编译器（我们实现自己的编译器操作），操作系统几方面来增加相应的功能。在使用时首先需要查看是否该架构支持，如果支持则需要通过xsave指令来设置相关的寄存器使能CPU，同时也需要打开内核支持。CPU的支持和操作系统的支持是分开的。根据以上所述内核提供边界表的管理和异常处理，如果不提供内核支持，边界表的管理就需要交给用户来实现，这仍然支持MPX的指令。但是在用户空间管理边界表因为不能对表实现回收操作，会造成极大的空间浪费，同时增加用户程序的复杂性。

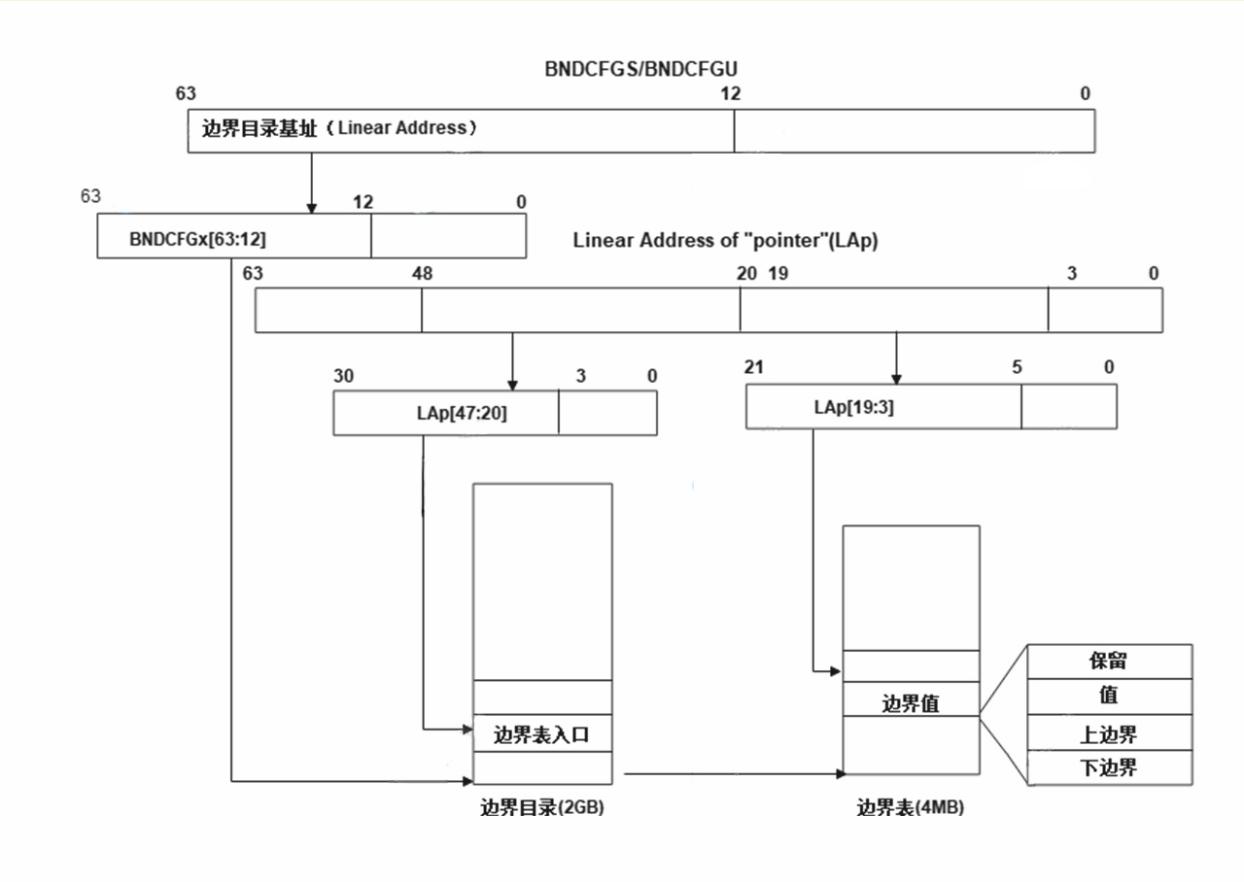


图2.8 地址转换过程

## 2.4 英特尔MPK的分析

内存页的访问权限是处理器和操作系统的重要职责，操作系统通过更改页表目录项和刷新块表TLB来重新加载该页以更改页表的访问权限。除了以上一些处理器也允许操作系统标记不同的页表，来为一组内存页分配相同的权限。MPK是英特尔推出的一种基于硬件原语的内存页保护锁机制，用来控制内存页的访问权限，MPK可以同时保护一组内存页，这和传统的mprotect()不同，通过为不同的页施加同一个保护锁来同时改变页组的读写权限。

MPK利用4个未使用的页表目录位（第32位到第35位）来标记页，这表示进程内存空间能够有16个不同的保护锁来保护页，被施加同一标记的页具有相同的权限，在程序运行过程中可以通过MPK提供的指令改变该页组的读写权限，由于系统中只有受到监督的代码才能够访问和更改页表，所以内核（版本4.6以后）提供了一种新的系统调用pkey\_mprotect()允许用户程序更为内存页设置保护锁。同时提供pkey\_alloc()系统调用提前分配保护锁和pkey\_free()来释放不用的保护锁。在使用保护锁时需要预先分配，然后利用pkey\_mprotect()系统调用将锁关联到对应的页。因为只有16个锁，所以当该锁不在使用时则需要调用pkey\_free()来释放。（注：在内核4.6版本之前，锁的分配释放等通过syscall函数来陷入内核调用相应的系统调用，这具有和上述相同的功能）

PKRU寄存器：MPK提供了一个32位的寄存器PKRU来表示16个保护锁的读写权限，每个锁用寄存器的两位来表示分别是访问禁止位（AD）和写入禁止位（WD），（AD，WD）的组合代表了页面的访问权限，其中（0 ，0）表示可读可写，（0，1）表示只读，（1，0）和（1，1）表示访问禁止。通过MPK提供的指令可以修改寄存器中的值从而设置不同权限。

表2.2 MPK指令扩展

|  |  |
| --- | --- |
| 指令 | 描述 |
| pkey\_alloc() | 分配新的pkey |
| pkey\_free() | 释放pkey |
| pkey\_mprotect() | 将pkey保护锁与页相关联 |
| WRPKRU | 更新访问权限 |
| RDPKRU | 获得当前访问权限 |

MPK指令集：如表2.2所示MPK提供了两条新的指令来管理PKRU。WRPKRU指令用来更新PKRU来设置不同权限，RDPKRU指令用来读取PKRU寄存器的值。WRPKRU使用三个寄存器作为输入，EAX，ECX，EDX寄存器，其中EAX的值用来更新PKRU寄存器，ECX和EDX则用0填充。RDPKRU也利用上述三个寄存器来输入，其中EAX存放PKRU的读取值。

MPK作为硬件保护锁机制相比于传统的页面保护mprotect具有性能上的优势，MPK可以在用户空间仅通过指令WRPKRU指令来切换权限，这需要不到20个指令周期[48]，不需要进行快表TLB刷新和上下文的切换。这在速度上有很大提升。其次MPK可以保护一组具有相同保护锁的页面，能够一次更改16个不同页组的权限这是相当高效的。MPK也是线程局部的，程序中每个线程都拥有一个独立的PKRU寄存器值，因此进程中的线程对同一地址空间的访问权限也有可能不同。

# 第3章 系统需求分析

## 3.1 可行性分析

### 3.1.1 技术可行性

我们从所依赖的平台技术，具体的开发语言，结合与硬件辅助的可行性与优点和硬件上的可行性来分析我们系统实现的可行性。

平台技术可行性：我们实现我们的系统主要基于版本10.0的LLVM编译器架构之上，是由C++语言开发的开源编译器平台。其前端Clang主要针对C/C++语言做编译，LLVM的架构相比于GCC编译器前后段耦合性小，具有灵活的中间语言IR，同时针对IR语言提供了丰富静态分析和动态优化接口。IR语言较为独立我们不需要了解特定的目标机器，可以直接在其上做优化。相比于原本CPI实现在LLVM3平台之上，我们更新了实现的接口代码。LLVM优化器opt可以将我们的安全优化组成pass模块（LLVM中的每一个优化方法都组成一个或多个pass模块）来在程序翻译时施加在代码之上，优化器同时提供给用户接口来使用优化方法，以pass为优化单元的设计极其方便了我们的实现。

MPX实现的可行性：不同防御机制的性能和程序中需要处理的操作的数量有很大的关系。CPI的一个主要优点是它的选择性：被认为是敏感的指针操作的数量只是程序中所有指针操作的一小部分。对于SPECCPU2006基准程序来说CPI静态分析规定的保护范围只有所有指针访问的6.5％[46]。相对于完整的内存安全性，这意味着性能开销减少了16–44倍。MPX是完整的内存安全防御机制，其性能损耗在50%以上，但是我们的工作是基于CPI思想，利用MPX这一系列的硬件辅助代替CPI的软件实现，这表示我们的实现不是完全的内存安全机制，主要内存保护对象是代码指针大大减少了需要保护的数据。根据上文介绍在性能损耗方面主要是边界存储和加载指令bndstx，bndldx所带来的，这两个指令在加载和存储边界时会由复杂的地址转换，这是主要的性能瓶颈。同时我们的实现解决了原来CPI安全区的缺点。下列将详细介绍利用MPX硬件辅助所带来的安全区的优化，以及软件实现所不具备的硬件优点。

安全区的缺点与优化：除了上文所说CPI安全区在安全性上的薄弱缺点，安全区也可能因为过大而造成内存浪费，而且大片的内存更加容易被探测到。CPI以软件的方式实现了和MPX一样的二级表来当作安全区空间，这将边界表交给用户管理，因此缺少了表的动态释放操作，在程序的逐渐运行过程中安全区会根据需求越来越大。在最坏的情况边界表的大小可能是正在保护数据的4倍（每一个代码指针需要32bit存储边界信息），一页的数据可能要求4个边界表界面，所以一个XGB的虚拟空间则需要4\*XGB的安全空间[49]，如果对大型的程序（比如百万行代码）做优化就会导致内存空间的巨大消耗。这是一个不可忽视的问题。MPX的边界存储表由操作系统管理（linux操作系统下），边界目录是在初始化时候创建一次，每个边界表根据程序内存分配需要动态创建，同时操作系统也必须释放这些表。在Linux中，每当内存对象被释放例如调用free()时，或者更准确地说，被取消映射时就会执行这种“垃圾收集”，操作系统遍历对象的元素，并删除所有相应的边界表条目，在内存中总是保存了最小安全区，而且表的动态释放也会使整个安全区的稀疏性提高，这进一步增加了通过栈空间数据指针探测到安全区位置的难度。虽然相比于原来CPI的实现会产生异常处理与边界表释放时候的性能损失，但是将安全区交给操作系统管理会更加的合理。而且操作系统不允许在用户空间修改边界目录中的目录项，这防止在暴露边界目录基地址之后直接修改目录项导致的程序崩溃。

优化安全栈的性能：由图2.4所示安全区的包括安全栈和常规安全区，这两块内存地基地址都需要放在初始化分配之后放在寄存器中，否则会造成性能上的损耗。当通过常规区域映射安全区的时候会加载安全区的首地址然后加上映射偏移量来计算对应位置。这导致在每次存储安全区的时候都需要加载安全区首地址，根据汇编指令的寻址特性，如果首地址被存储在寄存器中则可以通过寄存器间接寻址直接找到偏移地址比如（mov （gs ,rdx,1） , rax gs是x86段寄存器） ，而不需要多出一条内存加载指令来预先从内存区取出首地址。因此当程序中需要频繁的存取安全区的时候会不停地加载安全区首地址，相比较下会有一定性能损失。所以需要尽量将安全栈和常规安全区的首地址放置在寄存器中。x86-64架构中提供了GS段寄存器来供用户程序使用，可以通过系统调用arch\_prctl()来设置其中内容，MPX的使用同时也提供配置寄存器BNDCFGS/U来保存常规安全区的首地址，这不会导致GS寄存器的使用冲突，可以利用GS寄存器来保存安全栈的首地址。我们测试了安全栈的实现使用和不使用该寄存器的性能损耗达到了4%，我们将在测试章节给出具体的数据。

边界异常的恢复：#BR异常的引入为用户提供了在边界检查错误时的选择，在bndcl或者bndcn/bndcu引起边界检查错误时，处理器会陷入内核，内核会得到错误的边界值和地址并将它们存储在siginfo结构中，然后它将SIGSEGV连同siginfo中的违规信息一同传递给应用程序，该异常的引入使我们能够利用异常捕捉程序做相应的操作，而不是像原来一样仅仅简单地接受程序的崩溃。

MPX的向后兼容：MPX的一大优点就是它的兼容性，如果目标平台不支持MPX机制或者没有使能MPX则MPX的指令在执行的时候会被处理器解释为空指令（NOPs），这使在受到MPX保护的代码可以运行在不同平台而不会导致错误，增加了代码的可移植性。

MPK实现的可行性：MPK相比于原来的页保护机制，能够同时保护多个页，因此我们能够为所有的安全区页设置保护锁，而且根据上文所述MPK相比于原来的单页保护具有更快的权限切换速度，这不会给我们的整体性能造成过大的损失，我们将在测试一章给出MPK指令所造成的性能消耗，所以我们完全可以利用MPK实现我们对CPI的改进。

### 3.1.2 硬件可行性

英特尔Skylake处理器架构同时支持MPX和MPK机制，而关于内核的支持，MPX在5.4版本的内核才被移除，在这之前内核一直支持。同时MPK机制第一次被增加到内核中是在3.19版本，在4.9内核版本最终完善，我们的平台架构支持两种机制，选择的内核是4.15版本的ubuntu操作系统，所以在硬件上完全可以实现我们的模型。

综上所述我们的模型实现在技术上以及硬件上都具有可行性，我们所利用硬件辅助机制是架构直接支持的，这避免了使用模拟器来模拟硬件平台造成的未知错误。

## 3.2 功能性分析

### 3.2.1 功能概述

我们的安全系统主要目的是防御针对C/C++语言的代码重用攻击，保证程序跳转的控制流完整性，我们的工作专注于保护代码指针以及其他控制流相关的内存漏洞。根据我们需求的作用时间主要包括三个部分：初始化，静态分析需求和动态插桩需求，如图3.1和3.2所示能看到我们系统的主要用例。

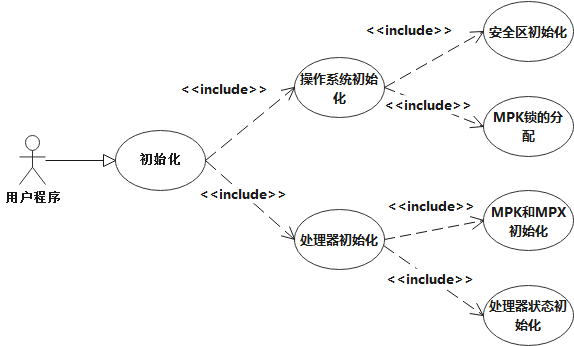


图3.1 初始化用例

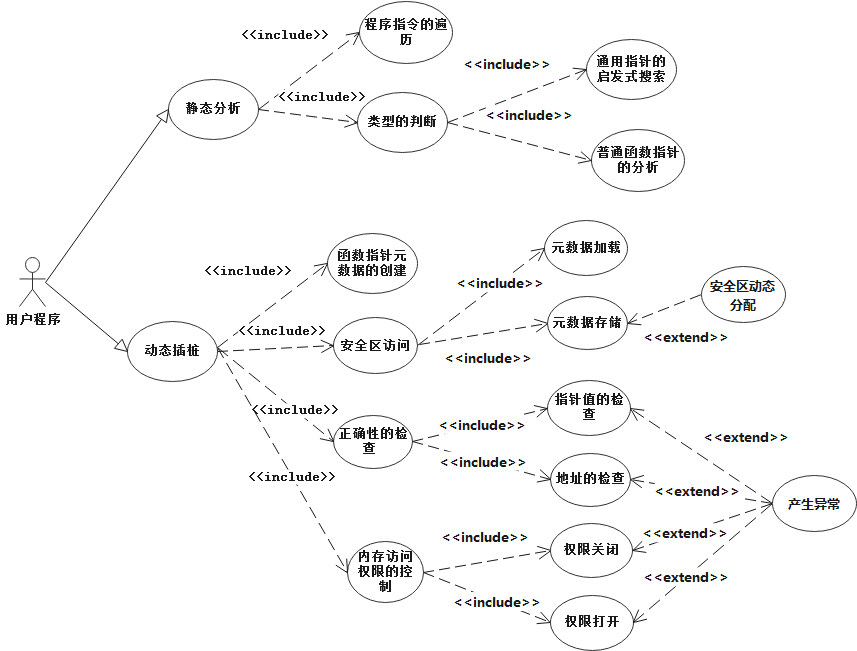


图3.2 静态分析与动态插桩用例

### 3.2.2 初始化需求

如图3.1所示对于初始化需要完成系统实现的准备操作，首先我们需要使能内核以及处理器对硬件辅助的支持。在处理器方面需要设置对应的CPU的扩展寄存器和MPX的配置寄存器等来支持。在操作系统方面，首先需要利用系统调用打开对MPX的内核支持否则系统将不提供边界表的管理和异常处理，其次我们分配边界目录并初始化为空，边界目录存储索引值需要在程序开始前进行分配，边界表则按需要动态分配。这还需要分配安全栈帧用来保护返回地址等。我们也需要向系统申请MPK保护锁，然后关联我们分配的边界目录（区分边界目录和边界表），并且初始化PKRU寄存器来设置初始访问权限。

### 3.2.3 静态分析需求

对代码的静态分析需求，主要是确定程序中需要保护的数据范围，首先能够对用户程序进行指令的遍历，特殊指令的识别以及拆分，以供我们做每条指令的类型判断。系统需要对程序做类型的判断，我们需要找出每一个控制流相关的数据，这包括堆栈空间普通函数指针变量，数据区的函数指针，运行时产生的控制流数据（如返回地址），编译器编译时生成的对象（如C++虚表指针等），一些特殊指针的分析（void \*类型的指针可以通过对指令流的启发式搜索来确定动态类型）。同时我们还需要通过迭代搜索结构体类型数据判断是否包含需要保护的数据。

### 3.2.4 动态插桩需求

在动态插桩阶段，根据静态分析所产生的结果，我们需要改变原来代码的结构，插入相应的指令来实现我们的操作。首先系统需要具有根据函数指针的地址和值创建元数据的功能。同时需要具有访问安全区的功能包括存储和加载元数据从对应安全区位置。对于正确性检查功能需要在指针使用时检查该指针的正确性以及地址范围是否溢出。为了保护安全区，我们增加了相应的内存访问控制功能，包括在访问安全区时打开和访问结束后关闭两部分。系统同时还需要增加异常产生的能力，当正确性检查出错，或者安全区的非法访问时会产生异常并且报告用户。

# 第4章 系 统 设 计

## 4.1 设计概述

### 4.1.1 系统功能模块概述

本章详细的分析设计了主要模块所具有的功能，具体结构组成如图4.1所示，IR语言在经过一系列用户和系统优化之后生成新的IR语言。LLVM pass是专门用来优化的代码模块，pass是一种编译器开发的结构化技术，用于完成编译对象（IR）的分析，优化等功能。系统中pass主要有两类，分析pass（analysis pass）用于静态分析并记录结果以供给其他的优化模块使用，例如别名分析（Alias Anaysis）pass，类型基础分析（Type Base Alias Anaysis，TBAA）pass。转换pass（transform pass）是通过对源代码的插入和删除改变代码结构，例如死循环消除（Delete Dead loops）等pass。我们将优化模块插入在所有系统优化之前，尽可能地利用系统的优化消除我们插入指令的冗余。由图4.1可以看出我们系统包括三大模块，分别是初始化模块，静态分析模块，动态插桩模块。我们将在下一节详细的细分系统的功能。

（1）初始化模块主要用来初始化处理器和操作系统以及为接下来的操作分配内存空间。

（2）静态分析模块主要用来分析整个程序包括（i）堆栈空间的变量分析：主要是分析堆栈空间上的函数指针进行分析（ii）运行时产生的数据分析：主要是对返回地址的处理（iii）编译时产生数据分析：包括提前初始化的全局变量，C++虚表，虚表指针，jump table表等（iv）通用指针的分析：主要是对程序中void \*类型的指针静态上下文分析来确定是否需要保护。

（3）动态插桩模块用来对静态分析的数据进行处理包括（i）元数据的处理：主要是对堆栈，全局空间上的函数指针创建，加载和存储元数据（ii）元数据的传递：主要是对函数指针参数，拷贝函数的处理（iii）正确性检查：主要是对函数指针在使用时的正确性检查，以及加载存储函数指针时对地址的检查（iv）安全区隔离：主要利用利用MPK保护锁保护分配的安全区以此来提供强大的隔离功能。

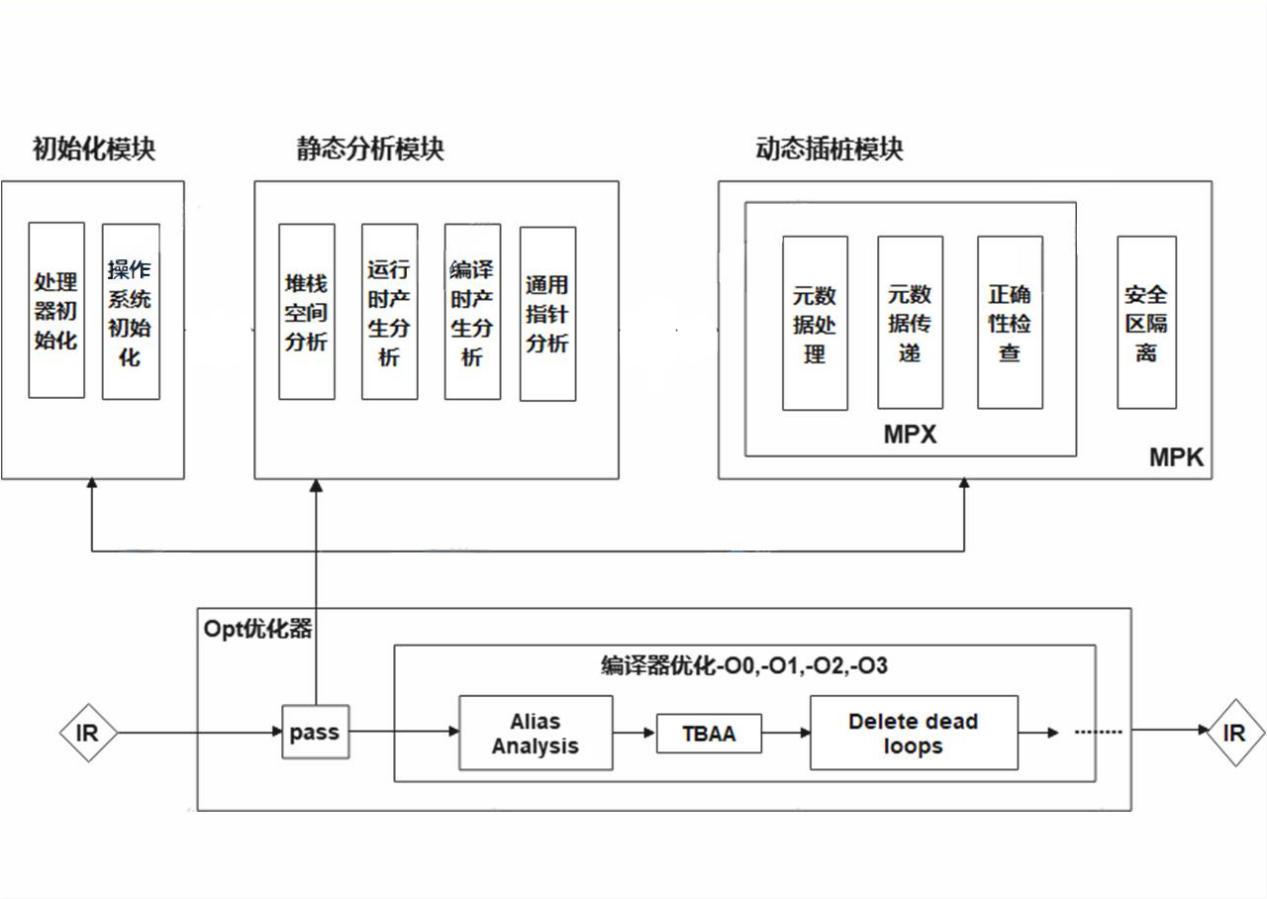


图4.1 系统功能模块结构

### 4.1.2 系统逻辑层次概述

除了在编译器层面的实现，也包括操作系统以及硬件架构的支持。如图4.2所示从逻辑层面上我们的系统涉及到硬件，底层体系结构，操作系统，以及编译器层面。

（1）在硬件层面主要使用MPX以及MPK提供的寄存器组包括PKRU，BNDSTATUS ，BNDCFGS/U，bnd0-3等寄存器来辅助对应功能的实现，帮助我们存储边界元数据以及控制安全区的访问权限。

（2）在底层体系结构层面主要是使用相关的汇编指令来实现具体的操作，包括bndstx，bndldx指令进行存储边界元数据，WRPKRU切换安全区读写权限等。

（3）在操作系统层面主要是对系统的初始化以及边界表的组织与管理，包括保护所的分配，边界表的动态分配与释放，异常的产生的处理等。

（4）在编译器层面主要来实现对程序的优化，包括对整个程序的静态分析，内存布局的改变等，具体通过下层提供的支持来实现。

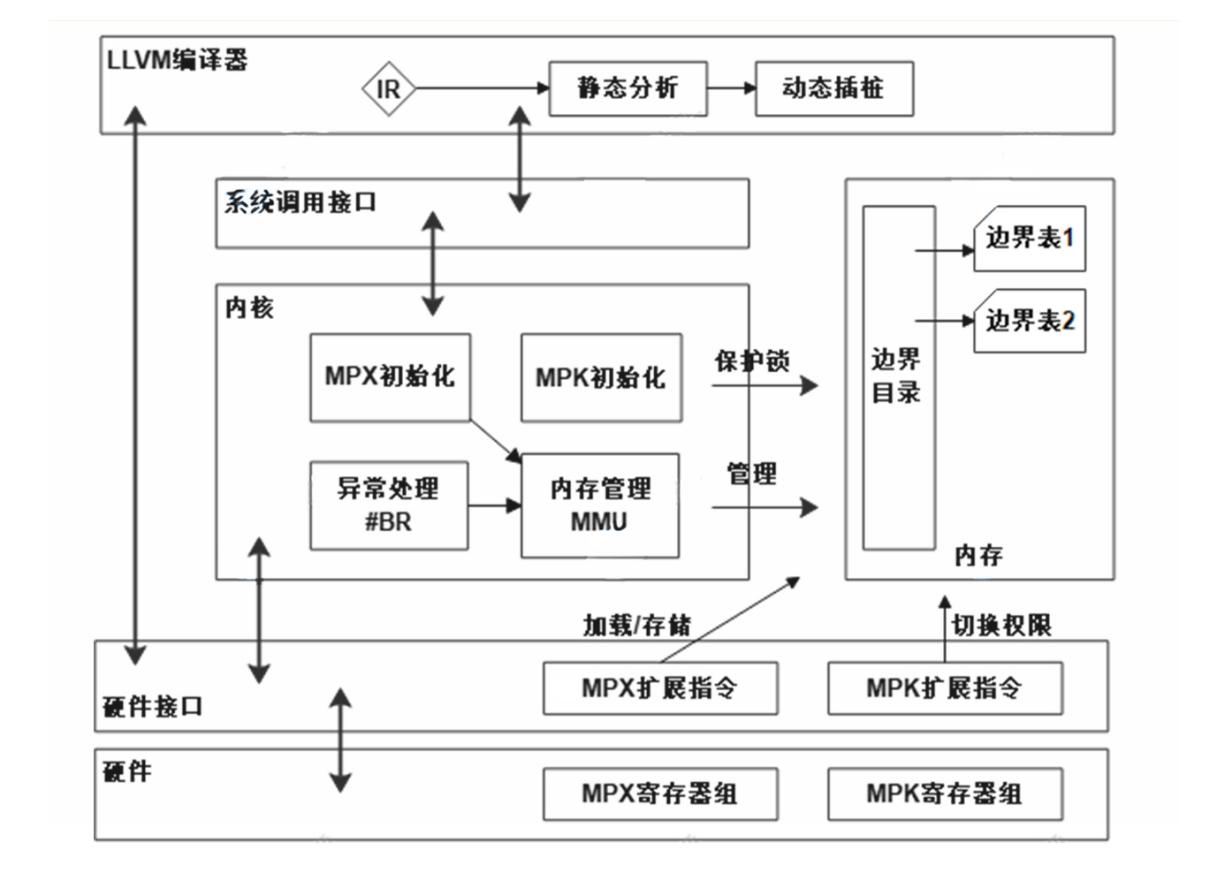


图4.2 系统逻辑结构

## 4.2 静态分析模块设计

我们在CPI的基础上扩大了敏感数据的界定范围，分析了程序中其他可能被利用来推测安全区位置的漏洞（例如C++虚表，jump table等），我们将这些数据归为保护类数据。我们对IR语言做变量类型分析，同时对于动态数据增加指令流地遍历来确定其类型，在IR语言最大的特点是只使用load和store两条指令来访问内存，这就表示我们能够通过搜索load和store指令来做类型判断，不需要处理其他的指令类型。我们的整个静态分析基于load和store指令。如图4.3所示是基本的静态分析处理流程。我们只需要处理进行内存访问的变量，我们保护敏感数组具有传递性，我们同时保护函数指针，指向函数指针的指针，指向包含敏感数据的结构体的指针，函数指针数组。这些变量被分配在不同的内存空间（例如堆栈等）。根据上面对系统模块的描述，我们将细分程序中需要静态分析的具体内容。

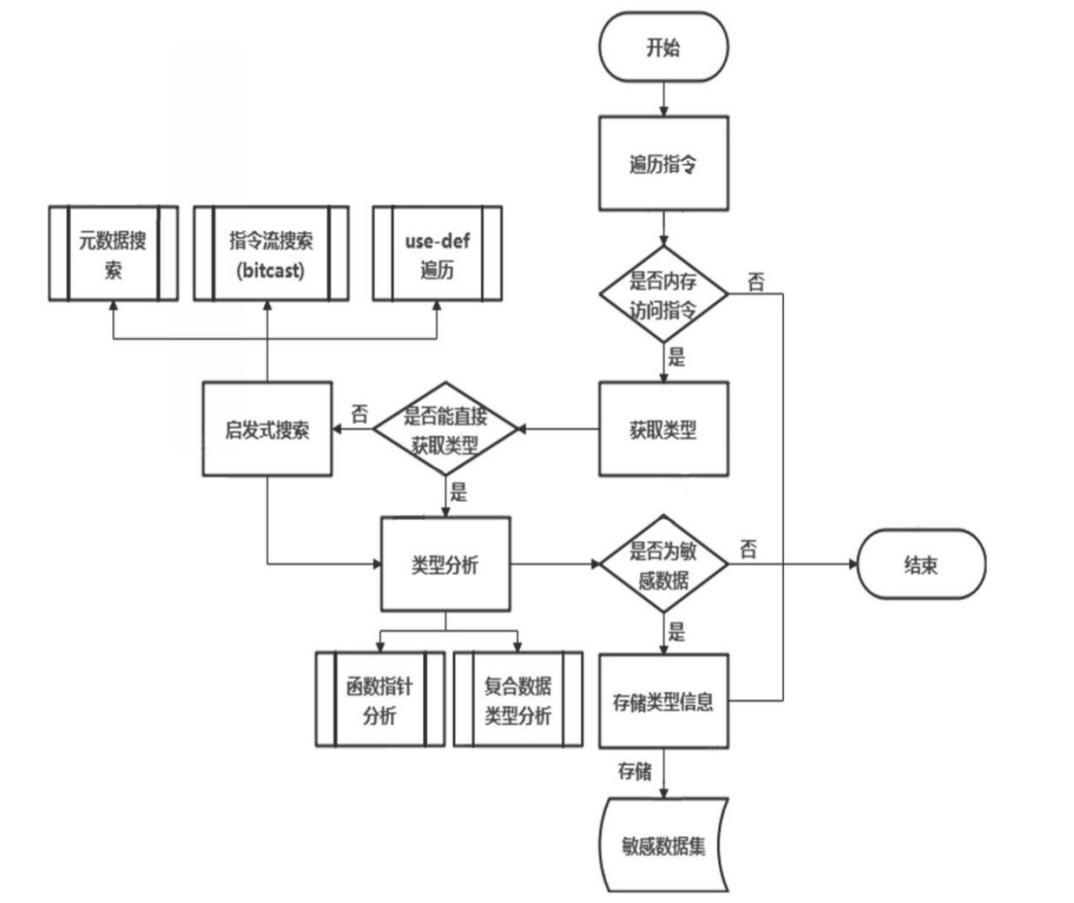


图4.3 静态分析流程

栈上的分配：在IR中通过alloc指令在栈上开辟空间，对变量对象的赋值会调用IR存储语句（store语句），根据上述传递性，我们保护所有相关的数据。

堆上的分配：堆空间的分配和栈空间分配有所不同，堆的分配通常采用动态分配函数例如malloc，realloc，new等，这类函数返回分配的地址通常是void \*类型的（IR中表示为i8 \*类型）。在使用动态内存时会转换为所分配的类型。所以我们不能在动态分配的地方判断返回值类型，需要向后遍历指令流（use-def链）找到使用的地方才能判断是否需要保护，而且可能会有在分配后程序一直以void \*类型的形式使用该动态内存。这种情况会导致无法判断具体类型，但是在我们的测试程序中并未发现类似情况。

static变量/全局变量：对static变量和全局变量的保护也基于传递性，我们保护所有相关变量数据，全局变量这类在程序翻译成IR时就已经有初始值，通常不使用store语句来存储值，但是在使用时会通过load指令从内存中加载，所以我们在main函数的运行之前找到这类数据进行类型分析，同时创建元数据并将预先初始化的值存储到安全区之中。

C++虚表指针：程序中在每个类的构造函数中同样利用store语句将虚表入口指针存储到类对象中，当该对象创建时会调用构造函数而为每个对象关联唯一的虚表。在虚函数解引用的时候会首先加载虚表入口地址继而调用虚函数。C++的虚表入口指针在IR中被翻译成void \*的类型，我们无法判断其类型。但是在IR中编译器会给虚表地址的store语句附加上TBAA（type based alias analysis）元数据用来描述其类型，我们在遍历构造函数的时候通过得到TBAA元数据来判断是否需要保护。

C++虚表：C++虚表存放具体的虚函数地址，同样其在编译时由编译器初始化，在编译器编译的时候会在数据区生成一张虚表来存储虚函数地址，这也是一类需要保护的数据，攻击者可能通过覆盖虚表来改变控制流。我们不希望非法修改虚表以及虚表指针。虚表存放在一片连续的内存区域，我们将整个虚表拷贝到安全区域。

返回地址：在程序运行时函数的调用会产生返回地址，通常攻击者利用缓冲区溢出覆盖返回地址以在返回时跳转到错误的地方，同时代码指针类型的参数溢出到栈上时也有被覆盖的风险。这可以利用安全栈来保护，它将栈上的返回地址和普通数据分别放在不同的区域，这防止了普通数据缓冲区溢出覆盖返回地址（例如利用数组的溢出攻击）。由此也可以看出这不需要元数据的检查操作。安全栈是一个很好的机制在性能以及保护性上相对较优。安全栈的访问不需要运行时的检查和相关的边界元数据，在使用返回地址时只需要从安全栈上返回即可。根据CPI来看只有不到25％的函数需要这种额外的堆栈框架，多数的函数不访问内存，所以栈的创建是根据需要与否，安全栈的性能开销基本为0。我们同样也实现了安全栈机制用来保护函数的返回地址等。

setjump缓冲区/jump table：setjump函数类似于goto语句，当在程序中使用setjump函数时会保存当前程序的上下文到固定缓冲区中，其中就有下一句指令的地址。当在之后的程序中调用longjump函数返回的时候会回到保存的地址处。所以我们需要保护setjump用来保存上下文的缓冲区。jump table用来辅助switch语句优化，当switch的case选项较多时会被编译器翻译成为jump table来优化程序代码，减少了判断语句的生成。跳转表中存放了每个case块的首地址。以上的数据也可能被攻击者利用来改变程序的跳转地址，我们需要对这类数据做保护。

通用指针：通用指针一般指void \* ，unsigned char \*指针，在IR中被翻译成i8 \*，包括上述堆栈分配地址，或者程序中的一些函数调用的参数会先转换为i8 \*的类型来传递。这类指针通常被当作i8 \*类型存储，往往在使用时当作敏感类型使用。也有编译时由编译器产生的通用指针，我们一般通过搜索其指令流来确定真实的类型，也结合该指令上被编译器添加的元数据来判断。结构体有一种类型是opaque类型的指针，这种类型结构体则无法确定其结构体中数据类型，我们观察我们测试程序发现这类数据只存在与C++程序中并且极少，我们将此类数据列入保护范围。

## 4.3 动态插桩模块设计

### 4.3.1 控制流检查

我们利用MPX指令集来实现所有的检查操作，通过在源代码中插入正确的MPX指令来实现功能，主要包括：元数据的创建，元数据的存储，指针地址的正确性检查，指针值的正确性检查，参数的处理（参数地址边界的传递），连续内存操作函数的处理（例如memcpy，memset等），动态分配函数处理，编译时初始化值的处理，图4.4可以看出我们具体的动态检查的流程。我们遍历静态分析得到的数据集合首先判断是否需要创建边界，如果需要就插入创建边界操作，为该变量创建元数据，然后存储边界值。如果不需要创建则加载边界检查控制流的正确性。在存储边界值的时候不管是否需要都应该检查存储地址的正确性。上述流程中对安全区的访问也都需要进行访问权限的切换。

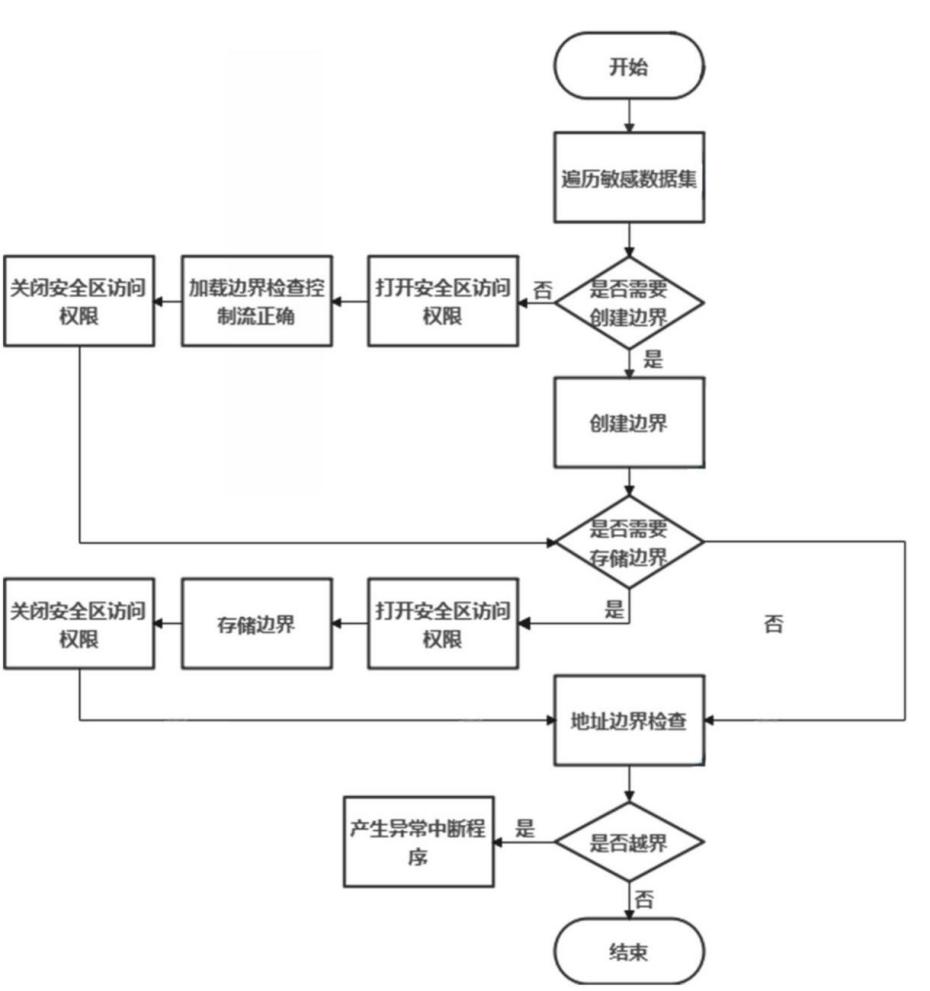


图4.4 动态检查流程

元数据的创建：在程序中为变量分配内存的地方需要创建相应的上下边界值，包括堆栈上的空间分配，全局变量/static变量，这些是程序员开辟内存空间的地方。由第二节分析可知，变量上下边界创建是基于变量所处对象的上下边界，这大致有两种情况，一种是变量是单独的函数指针不处于任何结构体中，变量的下边界就是该变量的地址，上边界就是该变量地址加上变量大小。另一种情况是，变量处在其他结构体之中（结构体，数组，容器，结构体中的数组），因此变量的上下边界就是最外层结构体的上下边界，这也表示结构体的上下边界是所有结构体中敏感数据的上下边界值，我们根据静态分析所确定的类型插入相应的创建指令实现对应元数据的创建。

元数据的存储：在程序存储敏感数据的地方，需要存储相应指针的元数据到安全区，系统在初始化时会有安全区的分配，我们的安全区是操作系统维护的二级表结构，每个边界表项32bit。IR中利用store语句来存储对应指针值到内存，我们修改原来的store语句，通过bndstx指令存储敏感指针的元数据。以保证在后续加载使用该指针时能够得到正确的元数据进行检查。还有一类边界的存储需要注意，就是拷贝操作，这类操作通常已经创建了元数据，需要复制源操作数的元数据到目的操作数的安全区中。

指针存储地址和指针值的检查：检查主要包括两部分，一部分时确认加载出来函数指针值不被修改，以保证在指针做解引用时控制流的正确跳转，通过对IR load语句进行修改，在指针从内存中加载出来时我们会利用MPX bndldx指令将安全区保存的元数据也加载出来，然后检查元数据中指针值是否改变，bndldx指令可以实现这一系列的操作，利用bndldx加载元数据时会先对指针值进行比较，如果值没有被改变则才会加载边界值，这也是一个比较好的特性。代替了软件实现的比较，在速度上有一定提升。第二部分是对代码指针地址的边界检查，包括store时和load时的地址检查，检查指针地址是否处在正确的边界范围之内，我们保护所有与代码指针相关的对象。注意检查的原则是每一个地址都有对应的元数据，而在加载和存储时的地址所对应的元数据并不是当前地址关联安全区所存储的数据，如图2.5所示我们能看出地址的检查是基于所处的对象的范围，对当前地址的检查我们需要根据指令流向前寻找其对应上下边界。

参数的处理：在函数调用时，如果参数是敏感数据类型，则需要为参数传递相应的上下边界值到该函数中。LLVM的优化是以函数为单位的静态分析，并不提供函数之间的数据流分析，所以我们不能在函数中确定该函数参数的上下边界。这是存在于调用者的函数体之中。因此需要对参数做单独的处理，在调用函数中判断每个被调用的函数参数的数据类型，为敏感类型参数传递上下边界以供在被调用者的函数体中使用。我们用MPX bnd寄存器传递上下边界值，这样在被调用者的函数中如果使用参数的边界数据就只需要到bnd寄存器中读取。

连续内存操作函数的处理：连续内存操作包括memcopy，memset，memmove等，我们需要在拷贝的同时将对应安全区的元数据也都拷贝。这类函数通常将源地址与目的地址转换为void \*的类型然后进行拷贝，所以这需要分析确定拷贝的真实类型。但是在测试中我们发现有些无法判断其真实类型，所以将这类操作也归为待处理范围，这不会影响整个程序的正确性，但是会在性能上造成一点损失，我们观察测试程序发现这类操作并不多。对连续内存的拷贝需要区分拷贝的每个变量是否是敏感指针（例如结构体中只有是函数指针的变量需要拷贝元数据）。对不能确定类型的操作，我们会拷贝结构体中所有变量对应的元数据。

动态分配函数处理：动态分配除了需要创建边界信息，还需要存储边界信息到对应的安全区之中，动态分配函数一般返回的类型时是通用指针类型。我们需要在静态分析阶段确定分析真实的类型。同时动态分配会在分配空间时进行初始化，我们同样需要为其创建元数据存储到安全区之中。

编译时初始化值的处理：这包括全局变量，static变量，C++虚表，jump table等，这类数据在编译时被初始化，所以在程序的中没有store语句去存储具体值，每个IR模板都会维护该模板的全局变量值，我们通过遍历全局接口取出初始化的值，然后插入对应的bndstx操作存储到安全区，这类数据的处理需要在main函数运行开始之前，所以我们修改了编译器翻译时全局初始化的函数来加入我们的操作。

### 4.3.2 安全区保护锁

MPK保护锁：如图4.2所示我们在原有的实现上添加MPK的保护，这包括保护锁的初始化和安全区权限的检查两部分。

保护锁分配：要利用MPK首先需要初始化，我们利用内核提供的系统调用申请一个MPK锁。在程序开始的时会分配边界目录，我们同时也会关联保护锁到边界目录内存也，设置边界目录为不可访问。而边界表是按照需要分配的，会调用#BR的异常处理程序来完成边界表的分配，我们需要修改内核提供的#BR异常处理程序来加入操作：即在每一次分配边界表的时候都对为边界表界面设置保护锁。我们只需要一个保护锁来为所有安全区内存页提供保护，当程序运行结束后会执行我们的资源回收操作，避免锁的浪费。

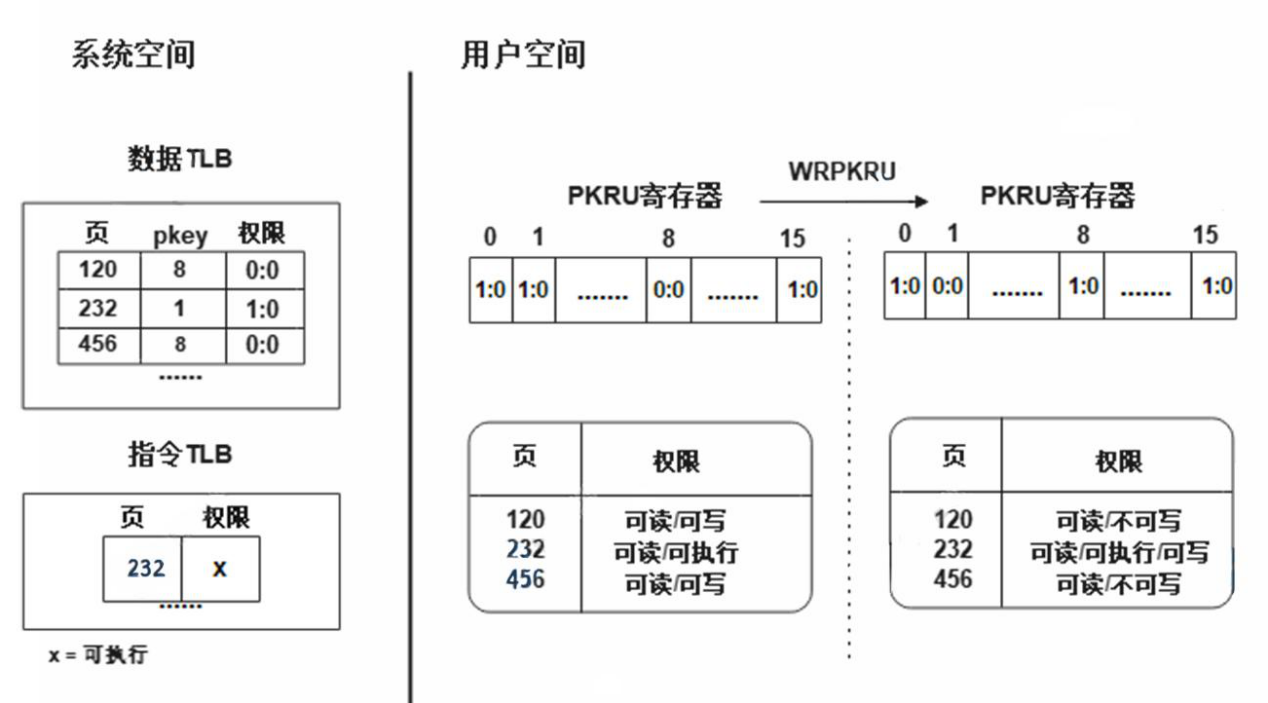


图4.5 MPK内存页权限切换

安全区权限检查：初始化时安全区被设定为不可写，我们需要在访问安全区的时候打开写权限，同时在结束访问时关闭。这利用WRPKRU指令将PKRU寄存器覆盖成不同的值来实现，锁的打开和关闭往往是成对出现的。当不访问安全区时则一直处在不可写的状态，只有利用我们提供的安全区访问指令才能够访问。由上文可知程序中访问安全区的代码只有我们插入的MPX指令。所以根据我们插入的存储加载安全区的代码来增加相应的打开关闭权限操作。我们插入的访问安全区指令只有bndstx和bndldx指令，所以在这两种指令之前我们需要做相应的权限修改操作。而通过非法指令去修改或者溢出攻击则会导致程序崩溃，即使安全区位置泄漏，攻击者也无法修改其中的元数据，如图4.5所示我们能够看到利用WRPKRU指令来操作内存页权限的过程，保护锁8和1分别控制两组页，可以同时修改其权限。由于需要加入保护锁的打开关闭代码这在性能上会造成一定的损失，我们将在测试章节给出具体的性能损失数据。

# 第5章 系 统 实 现

## 5.1 概述

我们实现基于LLVM10编译器架构，如图5.2我们的pass优化模块被注册到优化管道中，通过clang编译命令（-Xclang -load -Xclang）加载我们的优化模块。同时在LLVM compiler-RT（LLVM运行时库）链接库中增加我们库支持。我们将需要插入的指令封装成函数，通过调用的方式插入到程序的指定位置。我们修改compiler-RT增加了我们的函数接口。同时因为向程序中插入频繁的函数调用会产生较大的开销，所以我们将函数以内联的方式优化。这需要将我们提供的函数和源程序编译成bitcode位码，然后在链接时候启用链接时优化（-flto）来实现，如图5.1所示能看出两种不同的优化方式。我们将为上一章的模块实现对应的pass，这主要包括：初始化，静态的分析pass，MPX插桩pass，MPK插桩pass，我们在这一节将介绍主要的代码框架以及实现效果。我们要实现自己的优化pass首先继承系统提供给我们基础的pass类，这些pass类根据对程序不同的需求来继承不同的基类，主要包ModulePass、CallGraphSCCPass、FunctionPass、LoopPass和RegionPass类等。如图5.2所示，我们主要继承于ModulePass基类来实现我们的分析转化功能，图5.3是一个pass的主要框架，可以看出ModulePass以IR模板为输入来完成操作，我们需要实现runOnModule函数来完成具体操作。

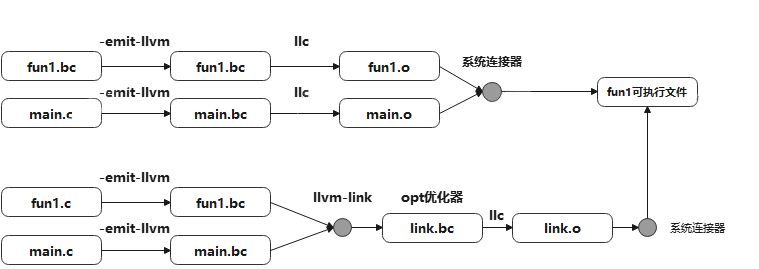


图5.1 pass优化方式

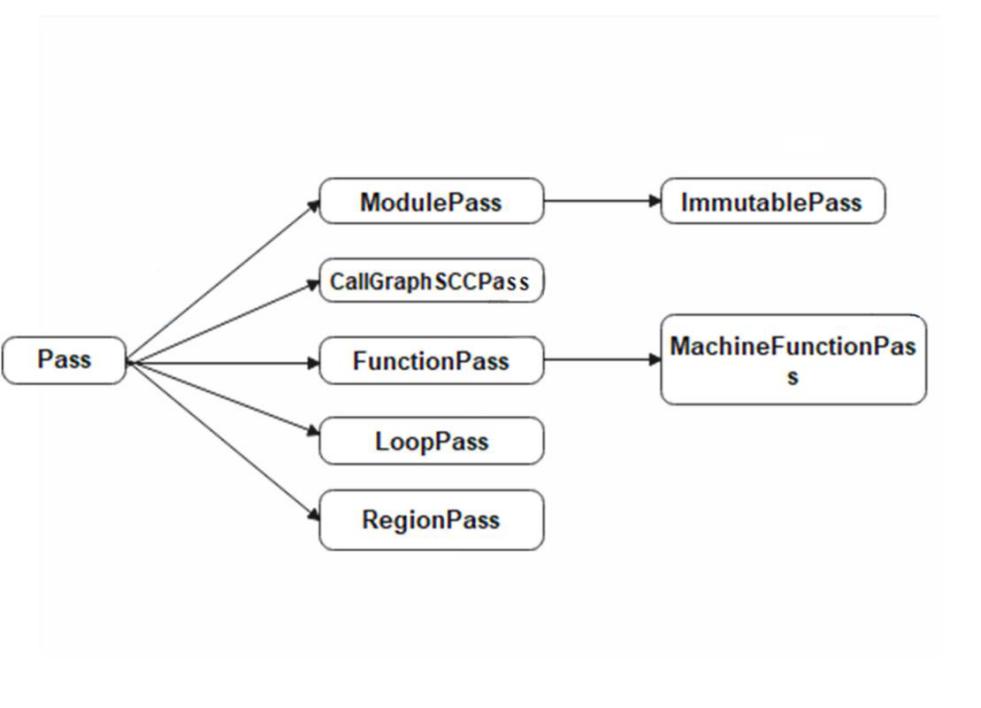


图5.2 pass实现的继承基类

|  |
| --- |
| 1. **namespace {** 2. **struct MyPass: public ModulePass {** 3. **static char ID;** 4. **MyPass() : ModulePass(ID) {}** 5. **bool runOnModule(Module &M) override {** 6. **}** 7. **};** 8. **}** 9. **char MyPass::ID = 0;** 10. ***//Register for opt*** 11. **static RegisterPass<MyPass> X("MyPass", "my pass");** 12. ***//Register for clang ，add函数会将pass添加到系统优化集合中*** 13. **static RegisterStandardPasses Y(PassManagerBuilder::EP\_EarlyAsPossible,** 14. **[](const PassManagerBuilder &Builder, legacy::PassManagerBase &PM) {PM.add(new MyPass());});** |

图5.3 Pass功能的基本实现框架伪代码

优化pass注册：为了能够使用pass需要将我们的pass提前注册到LLVM优化管道中，我们选择在所有的主模板优化之前插入我们的优化，如图5.2所示是主要的注册方法，RegisterPass是pass的静态注册方法，RegisterStandardPasses是pass的动态注册方法。静态的注册表示pass可以用优化器opt的命令去使用我们的pass，pass也以静态的方式存储在系统的优化合集中。动态注册表示我们不需要在程序翻译成IR语言时手动添加优化，程序的编译时我们的pass会以动态库的形式加入到优化集合中。

## 5.2 初始化实现

MPX使能：在使用MPX机制时需要先进行初始化，保证CPU，操作系统（这里是linux系统）支持MPX，我们需要设置指定寄存器以打开对MPX的支持。(i)首先设置CPUID的14位（MPX使能位），这表示了CPU是否支持MPX指令集，如果不设置MPX指令在编译时会被当作空指令（NOPs）。(ii)继而需要设置XCR0寄存器中的关于MPX的位BNDREGS和位BNDCSR，保证在进程做上下文切换时，MPX边界寄存器bnd，配置寄存器，状态寄存器能够保存和恢复（注意这里要同时设置两位来使能）。（iii）设置MPX的配置寄存器最低位，这一位也用来表示操作系统是否支持MPX。从图5.4我们能看到具体寄存器以及需要设置地位。

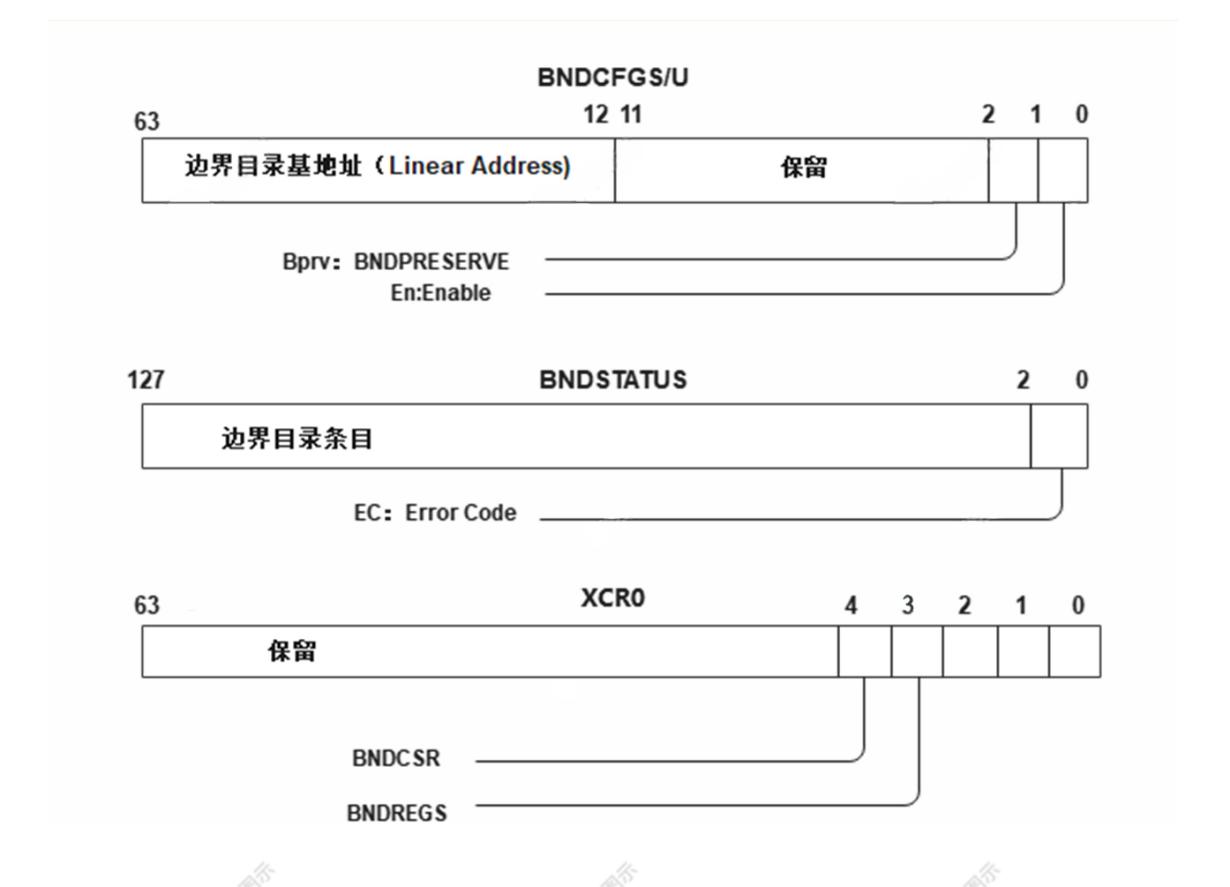


图5.4 初始化寄存器设置

安全区分配：安全区的分配通过系统调用mmap函数随机分配，48位的地址保证了很难暴力猜测到基地址。同时为了使安全区首地址的频繁存取不会导致性能上的过大开销，我们需要将分配的目录表首地址设置到MPX BNDCFGS/U配置寄存器中，该寄存器禁止了非法指令的访问，在以后程序中对该寄存器的访问只有通过bndstx和bndldx两条MPX指令，该寄存器在进程上下文的保存和恢复也都由操作系统来实现。

对于上述MPX寄存器的设置我们使用XRSTOR指令，这个指令用于从指定内存区域恢复CPU寄存器的状态，我们将需要设置的寄存器值设置到指定内存区域，然后利用插入的XRSTOR指令将这些内存的值恢复到上述所介绍的寄存器中以此来使能CPU支持MPX指令，还需要利用prctl()系统调用使能内核的支持，保证MPX二级页表和异常处理能够使用。我们将这一类初始化组成相应的函数加入到LLVM运行库中，同时以函数调用的方式来插入到IR中，在程序main函数调用之前的全局变量初始化阶段运行。利用函数appendToGlobalCtors将我们的初始化实现附加到全局槽中（appendToGlobalCtor是LLVM实现能够设置指定的函数到程序的全局初始化队列中，同时能够指定运行顺序）保证在main函数开始运行之前做好准备工作，同时在程序运行完之后的收尾阶段时回收上述分配的内存空间同样利用appendToGlobalDtor函数插入我们的调用。

## 5.3 静态分析实现

分析pass用来确定并且记录需要保护的敏感数据，这基于遍历指令为基础，我们继承ModulePass模板，这以IR Module为输入。在IR模板中拥有函数和指令遍历器帮助我们遍历所有指令。对模板的静态分析需要遍历到每条指令，我们搜索模板中的所有store，load指令以及函数调用指令，我们对load，store指令做指令操作数的类型判断。同时收集所有函数调用判断其是否是函数指针调用。具体如图5.5所示这是基础的指令搜索框架。我们找出程序中所有的load，store语句，然后取出其中的类型做具体的分析，如果是函数指针可以直接判断，如果是复合的类型需要遍历该复合类型中的每一个变量类型，同时我们使用迭代搜索复合类型中的复合类型，直到确定该类型中含有代码指针，我们将迭代路径上所有的类型指针加入需要保护范围。如果是通用指针类型我们将从该指令向前搜索找到该指令中操作数的定义，同时也能够向后搜索找到该指令标号的使用者。这是一些程序中通用指针的类型判断的基础，可以通过找到其最初的类型（注意跨函数的分析则不可能）来确定其在程序运行中是否有转换操作。当我们遍历复合类型的数据成员时例如结构体，编译器会为结构体中的成员附加tbaa.struct类型的元数据来描述结构体中各个成员的信息，我们可以通过LLVM接口函数getMetadata()得到指令的元数据进行分析，通过use-def链做启发式的搜索来找到通用指针是否在程序运行中被转换为敏感数据。图5.6能看出具体的use-def搜索方法。其中getOpcode()为LLVM接口用来的到指令的操作数，getOperand()用来的到每条指令的操作数，use\_begin()和use\_end()则是LLVM提供的迭代器用来遍历一个变量的使用集合。有些指令在翻译成IR时会被编译器翻译成聚合操作（会将多个指令组合成一句），我们需要分析每一条指令。

|  |
| --- |
| 1. ***//指令搜索*** 2. ***//Module: IR模板类*** 3. **procedure search(M:Module);** 4. ***//Instruction: IR指令类*** 5. **val inst : Instruction;** 6. ***//Function: IR函数类型*** 7. **val func : Function;** 8. **begin** 9. **//遍历函数** 10. **for(func := M.begin() to M.end()) do** 11. **begin** 12. ***//遍历指令*** 13. **for(inst := func.inst\_begin() to func.inst\_end()) do** 14. **begin** 15. ***//判断指令类型*** 16. **if(inst = Instruction::load or** 17. **inst = Instruction::store or** 18. **inst = Instruction::call or** 19. **inst = Instruction::ret）** 20. **then** 21. ***//做具体类型分析处理，并且存储敏感类型*** 22. **Analysize Real Type;** 23. **end** 24. **end** 25. **end** |

图5.5 静态分析搜索过程伪代码

|  |
| --- |
| 1. ***//use-def遍历*** 2. ***//Value:IR中变量基础类*** 3. **Function getRealType(v:Value) Type;** 4. **val type:Type;** 5. **val op:Operator;** 6. **val use:Use;** 7. **begin** 8. **do{         *//Bitcast:IR强转指令*** 9. **type:=getType(v);** 10. **flag:=call analysis(type);** 11. **if(flag = true) then** 12. **return type;** 13. **}while(v = op and v:= cast<op>(v) and** 14. **v->getOpcode() == Instruction::Bitcast and** 15. **v=v->getOperand())** 16. ***//遍历v的所有使用者*** 17. **for(use = v->use\_begin() to v->use\_end()) do** 18. **type:=getType(use);** 19. ***//分析类型*** 20. **call analysis(type);** 21. **end** 22. **end** |

图5.6 use-def遍历分析伪代码

## 5.4 控制流检查实现

创建上下边界：在程序中只有在申请内存时才需要为该内存创建上下边界，在其他情况都是通过指令流来传递我们的边界值，我们通过搜索指令流可以确定每条指令的操作数所对应的上下边界值（每条指令的每个操作数都各自对应这自己的边界）。IR中栈空间上分配指令是alloc，堆空间的分配指令有new，malloc，realloc等，我们需要以这些指令为插入点，插入我们的指令来创建边界值。如图5.7所示，首先利用SetInsertPoin()函数设置alloc语句为插入点，该函数时LLVM提供接口可以设置IR程序中的插入点，这里设置每个alloc语句之后为插入点，alloc返回分配的地址，我们插入强转语句将其转换为类型void \*（IR中用i8 \*表示）创建下边界，然后利用基地址加上分配大小创建上边界，其中LLVM维护每种类型的size。

|  |
| --- |
| 1. **struct A{** 2. **int a;** 3. **void (\*p)();** 4. **};** 5. ***//%表示临时标号 ，alloc分配结构体*** 6. **%1 = alloc struct(A);** 7. ***//插入bitcast,low为下边界*** 8. **%low = bitcast %1 to int64;** 9. ***//插入add,up为上边界*** 10. **%up = add %low , int64(size-1) ;** 11. ***//插入bndmk创建上下边界*** 12. **bndmk (%low,%up) , bnd0-3** |

图5.7 创建边界操作伪代码

元数据的存储：在程序中只有在store语句时我们才需要存储我们的元数据，元数据的来源有两种，一种是如上述创建的边界值，一种是通过指令流传递的边界例如从程序中加载结构体指针时会对应的加载其中的上下边界，当在之后加载该结构中的函数指针时会利用结构体的上下边界值进行检查。如图5.8所示我们插入相应的MPX指令来保存我们的元数据。bndstx指令传递地址address和值val为目的操作数，bndstx翻译address找到对应安全区的地址然后将bnd0-3寄存器中的边界值和值val存储到其中（注意如果val缺省则存储0），IR中不支持直接插入汇编指令，但是支持插入嵌入式汇编，我们将以嵌入式汇编的形式插入我们的操作。

|  |
| --- |
| 1. **store %val, %address;** 2. **//插入bndstx,将bnd0-3中的边界和val值存储到安全区** 3. **\_\_asm\_\_ (** 4. **bndstx bnd0-3 , (%address , %val));** |

图5.8 元数据存储伪代码

边界的检查和函数指针的检查：边界的检查需要对两个地方检查，分别是load和store语句，这需要在访问内存地址之前检查该地址是否处于正确的范围之内。如图5.9所示，我们需要插入bndcl和bndcu指令来检查上下边界值，同时我们还需要在load指令加载内存变量时，检查变量是否被改变，在load指令之后插入bndldx指令用来在变量加载时同时加载它的上下边界，以供在接下来使用该变量时能够传递边界值。而我们对变量值的检查也用bndldx指令完成，bndldx指令会在加载时比较存储在安全区的指针值和load语句的指针值是否一致，如果一致就会加载边界值到bnd寄存器中，否则加载0边界值到bnd寄存器中。我们由此可以判断变量值是否改变，这只需要一条指令的实现即可省去了软件实现的比较，同时完成函数指针的检查。

|  |
| --- |
| 1. **\_\_asm\_\_(** 2. **//插入bndcl比较下边界** 3. **bndcl %address , bnd0-3** 4. **//插入bndcu比较上边界** 5. **bndcu %address+8 , bnd0-3);** 6. **store %val, %address;** 7. **\_\_asm\_\_(** 8. **bndcl %address , bnd0-3** 9. **bndcu %address+8,bnd0-3;)** 10. **%val = load %address;** 11. **\_\_asm\_\_(** 12. **//插入bndldx传递边界值** 13. **bndldx (%address , %val) , bnd0-3);** |

图5.9 边界检查操作伪代码

安全栈的实现：安全栈的实现主要用来保护运行时栈空间的返回地址，和传参的溢出，这不需要增加对应元数据的检查。我们利用初始化时分配的安全栈空间，将一些常规数据放置在新的栈帧上，注意这里是保留原来的栈为安全栈，因为返回地址是在运行时产生，我们无法在IR层得到，所以不能移动返回地址到其他地方。这里我们删除原来的alloc指令，重新插入分配指令将常规的数据移动到新的栈帧上。最后利用LLVM接口函数replaceAllUsesWith更新原来alloc指令的use-def链（将所有使用alloc指令替换为使用新的分配语句）具体的代码实现如图5.10所示，栈空间是向下生长，所以unsafestack\_address是高地址减去变量大小得出其基地址。同时从图5.11我们可以看出安全栈实现的示意图，在新开辟的栈上我们从安全栈基地值依次存放局部变量，当函数调用时开辟新的栈空间时我们也在安全栈对应位置开辟新的空间。这样不会带来空间上的浪费。安全栈的实现相比于另外一种常用的返回地址保护技术CFI影子栈[50]来说性能上损耗可以忽略不计，影子栈的原理是将返回地址复制一份放在开辟的影子栈空间上，当函数返回时取出影子栈中对应的值做比较，如果不同则崩溃程序，这种机制因为要对所有函数分配栈空间所以相比于安全栈的实现性能上有一定损耗，影子栈不再我们的讨论范围，我们不再这里介绍具体的实现原理，我们实现了影子栈来单独与安全栈作比较，在测试一章我们将给出具体的测试数据。

|  |
| --- |
| 1. **func(){** 2. **//unsafestack\_address安全栈首地址** 3. **%0 = load @unsafestack\_address;** 4. **%1 = bitcast %0 to int64;** 5. **%2 = alloc int(a);** 6. **//size表示a的大小** 7. **%3 = sub %1 , %a.size;** 8. **//转换为类型a** 9. **%4 = bitcast %3 to int;** 10. **%5 = alloc struct(A);** 11. **//size表示A的大小** 12. **%6 = sub %1 , %A.size;** 13. **//转换为类型A** 14. **%7 = bitcast %6 to struct;** 15. **//释放栈** 16. **store %0 , @unsafestack\_address;** 17. **ret** 18. **}** |

图5.10 安全栈实现伪代码

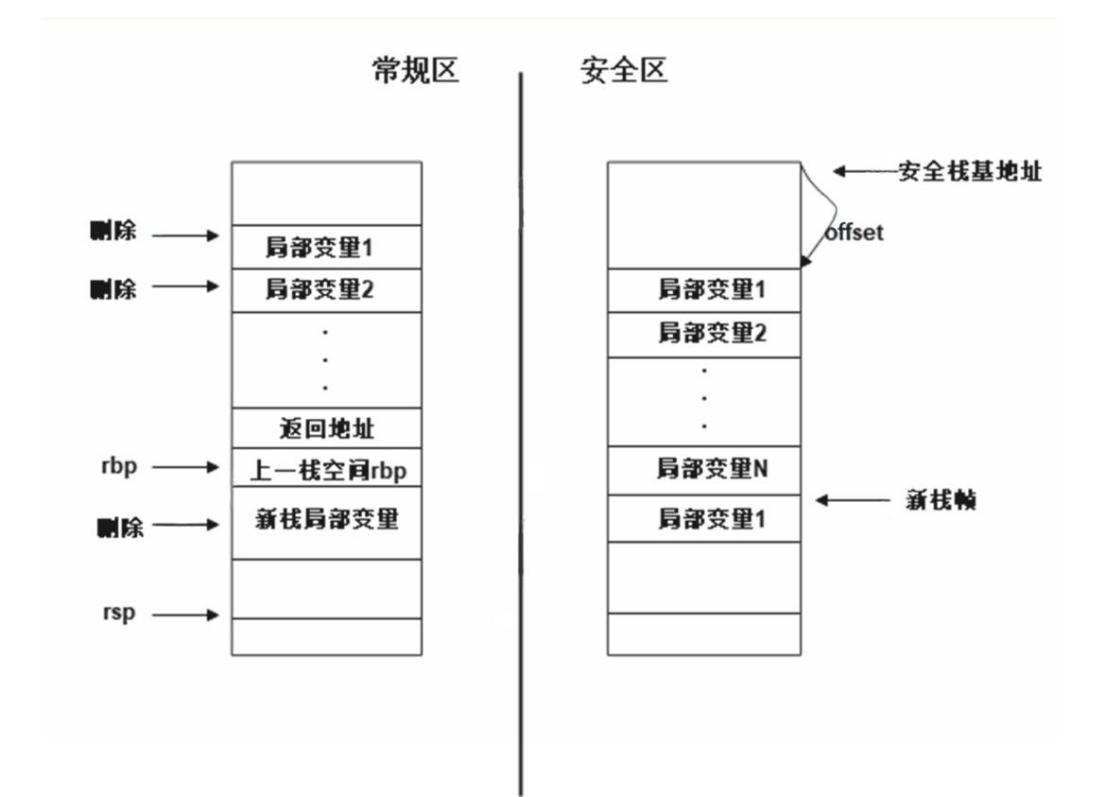


图5.11 安全栈实现原理图

参数边界的处理：当函数调用时需要向敏感类型的参数传递边界值，这需要在每个函数调用的地方插入bndmov指令将对应边界移动到bnd寄存器中，而在该函数中在使用该参数的地方根据第几个参数到对应的寄存器中取边界值，我们只利用bnd1-3这三个寄存器，如果参数多于三个需要存储我们将边界值存储到对应的安全栈之上来保证寄存器的溢出值也得到保护。同时在该函数中使用参数时从对应的安全栈上加载边界值。如图5.12所示可以看出在函数调用前和函数中参数的处理方式，其中创建边界和加载边界均是我们插入的指令集合。

|  |
| --- |
| 1. **%1 = load %address;** 2. **//插入bndldx传递边界值** 3. **{ %low ,%up }=bndldx (%address , %1) , bnd0-3);** 4. **\_\_asm\_\_(** 5. **//插入bndmov为参数传递边界值** 6. **bndmov  (%low,%up) , bnd1);** 7. **%10 = call func (%0 , %1);** 8. **func(%0 ,%1){** 9. **%3 = load %1;** 10. **\_\_asm\_\_(** 11. **//插入bndcl比较下边界** 12. **bndcl  %1 , %bnd1;** 13. **//插入bndcu比较上边界** 14. **bndcu  %1+size , %bnd1;** 15. **);** |

图5.12 参数处理伪代码

连续内存操作和动态分配的处理：我们处理程序中memcpy，memmove，memset，bcopy，bzero函数，程序中拷贝连续的内存时，可能包括敏感数据的拷贝例如拷贝结构体，需要将对应安全区的元数据也进行拷贝，此类函数的参数都是通用指针类型无法直接判断要拷贝的变量类型，我们通过上述的use-def遍历来查找真实的数据类型，如果类型可以确定，我们将遍历所有拷贝的变量，将敏感数据对应元数据进行拷贝。如果不能确定具体类型，为了保证插入代码的正确性，需要将所有的变量都进行处理，由于拷贝的内存是连续地址，所以其安全区也属于地址连续的，我们计算源地址对应安全区位置，插入memcpy函数拷贝安全区的元数据，同时要为目的地址分配对应的边界表，要注意当拷贝的数据不再同一边界表时需要做特殊的处理。动态分配以通用指针的形式返回分配地址，这需要和memcpy做同样的类型判断。如图5.13所示描述了当无法确定拷贝类型时的处理过程。

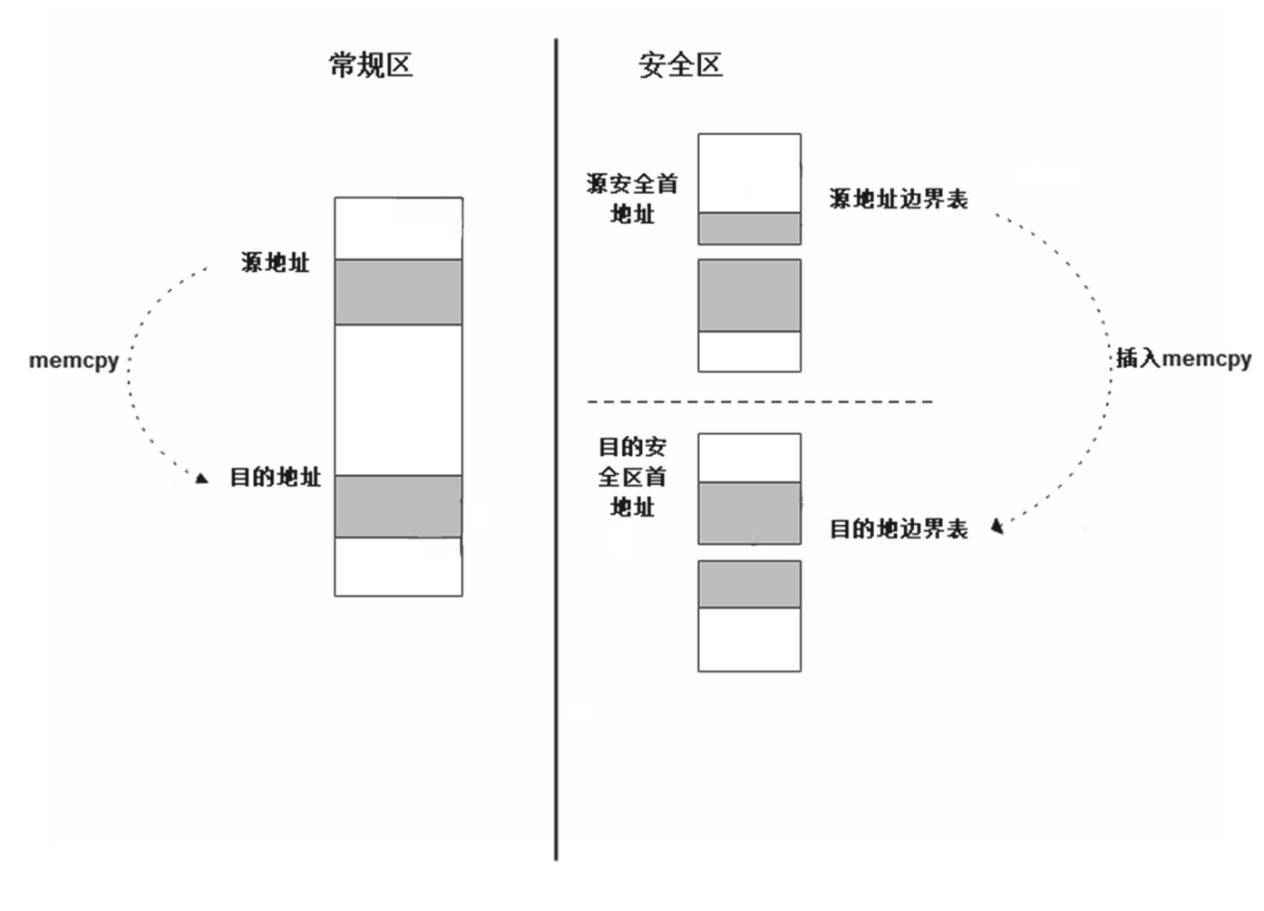


图5.13 连续内存拷贝操作处理方法

## 5.5 安全区保护锁实现

保护锁的初始化：安全区保护锁的实现提供pkey\_alloc()函数调用（系统调用号为330）来首先申请锁，这一步骤只需要在安全区分配代码一起插入到全局初始化队列中即可，但是给安全区内存页上锁则需要修改内核代码。因为每一个表的分配是通过bndstx指令引起的异常来处理的。所以我们需要修改内核中#BR异常处理程序的代码，我们在分配新的边界表的时候通过pkey\_mprotect函数（系统调用号为329）将边界表加入保护页组，我们重新编译安装了内核。

访问权限切换实现：当我们从安全区加载存储数据时，需要打开安全区，这里我们需要根据原来我们插入指令访问安全区的代码来确定我们需要插入的位置，包括安全区的存储加载指令bndstx和bndldx的位置，连续内存拷贝函数的处理位置，动态分配函数的处理位置，以及全局变量等其他位置。如图5.14表示我们开锁和关锁操作的实现。这在任何位置都是成对出现的。

|  |
| --- |
| 1. **\_\_asm\_\_ (** 2. **//清空ecx和edx寄存器** 3. **xor %ecx, %ecx** 4. **xor %edx, %edx** 5. **mov %0, %eax** 6. **//WRPKRU用eax寄存器的值覆盖PKRU寄存器** 7. **WRPKRU** 8. **:** 9. **:"r"(enable\_write));     //enable\_write使能写** 10. **//访问安全区** 11. **bndstx bnd0-3 , (%address , %val));** 12. **\_\_asm\_\_ volatile (** 13. **xor %ecx, %ecx** 14. **xor %edx, %edx** 15. **mov %0, %eax** 16. **WRPKRU\n** 17. **:** 18. **:"r"(diable\_write));      //diable\_write关闭读** |

图5.14 安全区访问权限切换伪代码

# 第6章 系 统 测 试

## 6.1 测试概述

为了评估我们模型的正确性以及性能，我们利用SPEC CPU 2006基准测试集来测试我们的模型性能损耗，其中抛开我们增加的页保护机制我们的模型实现具有12%左右的性能损耗，而施加保护锁的模型则具有19%左右的性能损耗，意味着我们的改进带来7%左右的性能损耗。在SPEC CPU 2006测试程序中通常C++相比于C程序会有更大的性能开销。因为往往有更多的代码指针访问特别是虚函数的调用。我们也测试了单独安全栈实现的性能开销在1-2%之内，同时给出影子栈的性能对比。

## 6.2 测试方案

我们使用SPEC CPU 2006作为我们的测试集，这是一整套的标准测试计算机系统性能的套件，我们利用其中的12个整数类型和6个浮点类型的C/C++基准程序，我们将对这18个基准程序施加我们的优化，我们配置测试脚本利用clang将基准程序编译成bitcode代码链接我们提供的函数库。我们以原有基准程序的CPU基准率（SPEC程序在测试完成后会产生测试报告，其中基准率用来表示CPU的性能）为参照，除以我们添加优化后的基准率来得出相比于原来的性能我们的模型实现造成了多少百分比的下降，我们的测试数据中GEOMEAN表示所有基准程序的几何平均。表6.1描述了我们测试的具体环境。

表6.1 测试环境描述表

|  |  |
| --- | --- |
| 工具 | 说明 |
| 操作系统 | Ubuntu 16.04.1 |
| 处理器类型 | Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz |
| 内核 | Linux 4.15.0-66-generic |
| 处理器架构 | x86-64 |
| 内存总量 | 128G |

## 6.3 正确性分析

一个完整的代码重用攻击总是程序中存在的任意的漏洞开始，如图6.1所示攻击者首先通过对程序的先验知识来获取内存中可能存在的漏洞作为攻击启动的起点（如函数指针），然后通过这些漏洞跳转到指定的gadget链头部开始攻击，攻击者可以利用面向返回地址ROP或者面向间接跳转JOP来完成具体攻击，从而达到劫持控制流的目的。由图可知启动攻击的必要条件是首先转移程序到攻击载体（gadget)入口处，这需要利用内存漏洞来修改函数指针，而CPI的防御思想则是保护所有这类数据，这就表示可以在攻击的启动阶段就停止该过程。这被证明能够有效的防御控制流劫持，同时对于CPI思想的正确性作者也进行了形式化的证明来验证其提供的防御保证，CPI能够很好的保证函数指针不被修改但同时也暴漏了我们前文所介绍的缺点即安全区的较弱的隔离性问题，攻击者完全可以通过对内存布局的探测来发现安全区的位置，从而启动攻击。我们的工作是基于CPI的思想同时提供强大的安全区隔离机制，这表示攻击者不能通过原有的内存漏洞影响函数指针，同时也不能通过直接修改安全区内存来进行攻击。理论上来说我们的工作在原有基础上增强了安全属性。证明我们模型能够有效的防御住代码重用攻击。

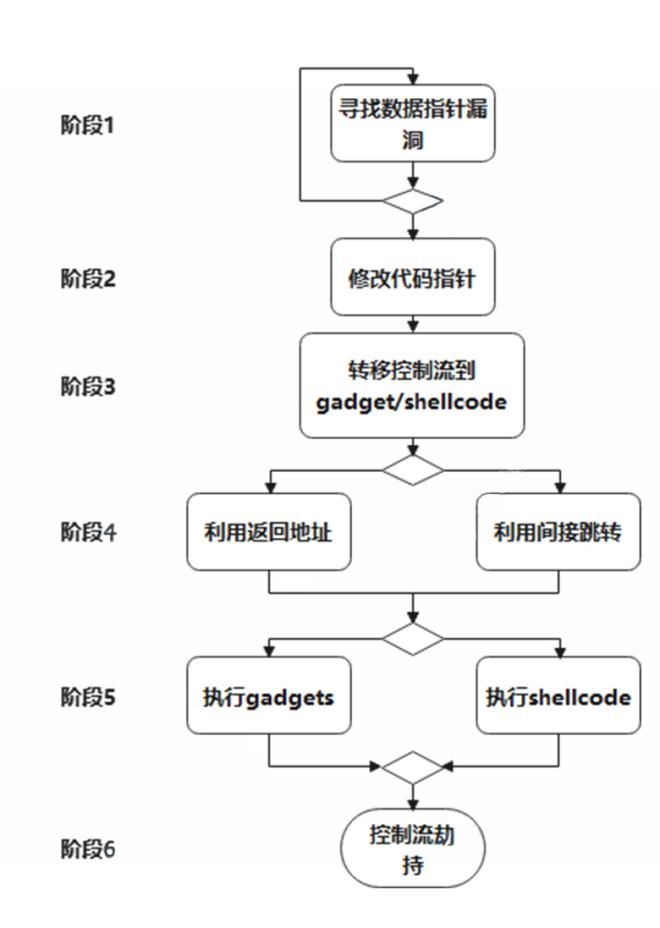


图6.1 代码重用攻击流程

## 6.4 测试数据

由表6.3可以看出不施加安全区保护的模型性能损耗大约在%12左右。从表中我们可以看出perlbench,gcc,omnetpp,xalancbmk基准程序带来较大的损耗，我们统计了各个基准程序中需要处理的动态加载存储指令，其中上述4个程序中拥有较多的针对代码指针的操作，因此bndstx和bndldx指令的增加会导致性能上的下降。如表6.4所示我们测试了程序中各个部分处理所导致的性能上的下降，这包括bndstx和bndldx指令导致的损耗，连续内存拷贝函数导致的损耗，全局变量处理导致的损耗三个主要部分。发现主要的性能瓶颈是bndstx和bndldx指令，这两条指令需要进行复杂的地址转换操作和内存的存储加载操作。这导致了6.4%的下降。而剩下的5%左右的损耗则是连续内存拷贝处理所带来的。我们也利用软件的方式实现了bndstx和bndldx指令，同时测试了bndstx，bndldx指令和软件方式实现上速度的差别，发现硬件上的实现相比于软件有将近2被的速度提升如表6.2所示。由此可以知道，MPX模型相比于原来CPI的实现理论上更快，但是因为安全区的管理是由操作系统实现，这就意味着异常的处理以及边界表的分配和回收时会产生额外的系统调用损耗。

由表6.3可知我们对安全区施加保护锁机制的性能损失大约在19%左右，由此能看出MPK指令带来了7%的性能损耗。我们对安全区增加的保护锁机制能够完全的限制对安全区的非法访问，这就导致通过内存泄漏来推断安全区的攻击无效，我们不需要专门的信息隐藏技术来防止安全区被找到。对比CFI一类的防御机制。在同样SPEC CPU 2006的测试下CFI的平均开销为21%，对于弱保护性的粗粒度CFI虽然有较低的开销，但是不能提供足够的保护性。对于MCFI[51]有5-6％的开销（但是不具有堆栈指针完整性）。CCFIR[34]的开销约为3.6％，binCFI[35]测试的SPEC CPU2006在8.54%左右，WIT[52]是一种基于源的保护机制，可同时执行CFI和写完整性保护，开销为10％。这些防御方法执行了较弱的CFI属性，根据目前的研究来看存在针对这类防御的成功攻击[36,37,53]。相比较来看，我们实现的模型具有强大的安全保证。

**表6.2 指令运行时间**

|  |  |  |  |
| --- | --- | --- | --- |
| 指令 次 | 100000000 | 1000000000 | 10000000000 |
| bndldx and bndstx | 0.667172 s | 6.24614 s | 61.713107 s |
| software | 2.045451 s | 20.498706 s | 205.871169 s |

注：software：bndldx和bndstx指令对应软件实现，次：运行的指令次数。s:秒。

表6.3 MPX模型和MPK模型性能数据

注：表中baseline为不加优化的CPU基准，都为1。另外两列分别为对应模型实现在性能上相比baseline多出的消耗

表6.4 bndldx和bndstx指令所造成的性能损失

注：bndldx and bndstx表示这两种指令所造成的性能消耗，MPX模型表示所有处理所造成的开销

3.安全栈性能测试：我们实现安全栈机制加入到我们的模型之中，用来保护返回地址等，如表6.5所示我们单独测试了的安全栈性能开销发现在1-2%之间。安全栈提供了很高的安全保证，如今部署的堆栈Cookie的开销高达5％，并且提供的保护性远远低于安全栈实现，我们实现了影子栈来对比发现影子栈的实现有4%的消耗。与影子堆栈相比，除了性能上安全堆栈还具有以下优点：

安全栈除了保护返回地址而且大多数局部变量也受到保护，而影子栈只保护返回地址。安全栈可以单独使用，而且支持异常和信号处理等。

表6.5 影子栈的性能开销对比

注：ShadowStack表示影子栈的实现性能开销，safestack表示安全栈实现的性能开销

表6.6 安全栈性能开销对比

注：其中GS表示使用GS寄存器的安全栈，而No GS则表示不使用

同时我们观察了程序的汇编语言发现，如果安全栈的栈基地址被当作全局变量存储在内存中则每回加载的时候需要多一条mov指令加载基地址，这导致了3%左右的下降，如表6.6所示这证明我们需要利用GS寄存器来存储基地址。

# 第7章 结论与展望

## 7.1 总结

控制流劫持是一类普遍且能够造成严重危害的攻击，而代码重用则是进行控制流劫持的主流攻击方法，本文介绍了面向跳转和面向返回地址的两种基本代攻击方法以及它们的衍生。详细论述了攻击的流程以及实现的条件，同时也分析了不同防御方法的原理和优缺点。利用了一种较为创新的防御思想CPI来实现我们的防御模型，这是一种较好代码重用攻击防御方法，主要的原理是保证程序中代码指针的完全性。我们也分析了原有实现的薄弱性，增加了新的内存隔离方法，在原有基础上使程序得到更加强大的安全保证。本文详细地介绍了攻击模型的原理，实现所依赖的技术，分析设计了系统的结构，并提出了具体的解决方案。同时也论证了模型的有效性，测试了具体的性能损耗。

## 7.2 未来工作

我们将进一步完善我们的模型，找出更加精确的静态分析算法和性能更低的插桩实现。我们现有实现上静态分析不够精准，我们希望进一步结合程序之间的数据流分析方法来做更加精确的静态分析。又因为软件种类的繁多，其中包含各种内存漏洞，我们将进一步测试更多的程序来完善我们的模型。对于性能损耗上我们希望实现不同的安全区组织方法来对比这之间的性能差异，寻找出更加高效且节省内存的方法。

# 参 考 文 献

1. Szekeres L , Payer M , Tao W , et al. SoK: Eternal War in Memory[C]// IEEE Symposium on Security & Privacy. IEEE Computer Society, 2013.
2. 邵思豪, 高庆, 马森,等. 缓冲区溢出漏洞分析技术研究进展[J]. 软件学报, 2018, 029 (005):1177-1198.
3. 王清. 0day安全:软件漏洞分析技术(第2版)[J]. 信息安全与通信保密, 2013(11):130-131.
4. Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016.
5. VAN DER V EEN , V., D UTT S HARMA , N., C AVALLARO , L., AND BOS , H. Memory errors: The past, the present, and the future. In Proceedings of the 15th International Symposium on Research in Attacks, Intrusions and Defenses, 2012.
6. 张超. 针对控制流劫持攻击的软件安全防护技术研究[D]. 北京:北京大学, 2013.
7. 柳童,史岗,孟丹.代码重用攻击与防御机制综述[J].信息安全学报, 2016,1(02):15-27.
8. 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](https://kns.cnki.net/kcms/detail/detail.aspx?filename=WXAQ201906002&dbcode=CJFD&dbname=CJFDTEMP)[J].网络与信息安全学 报, 2019(06).
9. Microsoft. A Detailed Description of the Data Execution Prevention (DEP) Feature in Windows XP Service Pack 2,Windows XP Tablet PC Edition 2005, and Windows Server 2003.<https://support.microsoft.com/en-us/kb/875352>.
10. 黄志军,郑滔.基于Return-Oriented Programming的程序攻击与防护[J].计算机科学, 2012,39(S1).
11. Kornau T. Return oriented programming for the ARM architecture[D]. Master’s thesis, Ruhr-Universität Bochum, 2010.
12. T. Bletsch, X. Jiang, V.Freh, and Z. Liang. Jump Oriented Programming:A New Class of Code-Reuse[C]. In Proceedings of the 2016 the 6th ACM Symposium on Information, Computer and Communications Security(ASIACCS ’11), 2011:30-40.
13. 邢骁. 自动化构造Jump-Oriented Programming攻击[D].南京大学, 2012.
14. PAX Team. “Address Space Layout Randomization” <http://pax.grsecurity.net/docs/aslr.txt>, 2003.
15. 林键,郭玉东,周少皇,蔄羽佳.基于基址重定位的地址随机化技术[J].信息工程大学学报, 2016,17(06):748-753.
16. SNOW K Z, MONROSE F, DAVI L, et al. Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization[C]. IEEE Symposium on Security and Privacy, 2013:574-588.
17. BITTAU A, BELAY A, MASHTIZADEH A, et al. Hacking blind[C]. IEEE Symposium on Security and Privacy, 2014:227-242.
18. J. Seibert, H. Okhravi, and E. Soderstrom. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code[C]. In Proceedings of the 21st ACM Confer- ence on Computer and Communications Security (CCS), 2014.
19. R. Strackx, Y. Younan, P. Philippaerts, F. Piessens,S. Lachmund, and T. Walter. Breaking the memory secrecy assumption[C]. In Proceedings of the Second European Workshop on System Security (EUROSEC '09), 2009.
20. M. Abadi, M. Budiu, J. Ligatti, and U. Erlingsson. Control-Flow Integrity[C]. in Proc the 12th ACM Conference on Computer and Communications Security(CCS’05), 2005:340-353.
21. Oleksenko O , Kuvaiskii D , Bhatotia P , et al. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack[J]. In Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2018, 2(2):1-30.
22. Evans I , Fingeret S , Gonzalez J , et al. Missing the Point(er): On the Effectiveness of Code Pointer Integrity[C]. IEEE Symposium on Security and Privacy, 2015.
23. J. P. Anderson. Computer security technology planning study[N]. volume 2. Technical report, DTIC Document,1972.
24. X. Chen, D. Caselden, and M. Scott. New zero-day exploit[J]. targeting internet explorer versions 9 through 11 identified in targeted attacks, 2014.
25. S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: compiler enforced temporal safety for c. In ACM Sigplan Notices, volume 45, pages 31–40. ACM, 2010.
26. G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type safe retrofitting of legacy code[J]. ACM SIGPLAN Notices, 2002,37(1):128-139.
27. Dan G , Hicks M , Jim T , et al. Cyclone: A Type-Safe Dialect of C[J]. C-C++ Users Journal, 2005, 23(1):6,8,10,12-13.
28. P. Akritidis. Cling: A memory allocator to mitigate dangling pointers[C]. In USENIX Security Symposium, 2010:177-192.
29. N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation[J]. In ACM Sigplan Notices, volume 42, ACM, 2007:89-100.
30. N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking[C]. In IEEE/ACM Symp. on Code Generation and Optimization, 2012.
31. K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker[C]. In USENIX Annual Technical Conference, 2012:309-318.
32. MASHTIZADEH A J, BITTAU A, MAZIERES D, et al. Cryptographically enforced control flow integrity[J]. arXiv preprint arXiv, 2014:1408-1451.
33. CRISWELL J, DAUTENHAHN N, ADVE V. KCoFI: complete control-flow integrity for commodity operating system kernels[C]. IEEE Symposium on Security and Privacy, 2014:292-307.
34. ZHANG C, WEI T, CHEN Z, et al. Practical control flow integrity and randomization for binary executables[J]. IEEE Symposium on Security & Privacy, 2013:559-573.
35. ZHANG M, SEKAR R. Control Flow Integrity for COTS Binaries[C]. Presented as part of the 22nd Security Symposium, 2013:337-352.
36. L. Davi, A. Sadeghi, D. Lehmann. and F. Monrose, Stitching the Gadgets On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection[C]. In Proceedings of the 23rd USENIX Security Symposium.(Usenix’14), 2014:401-416.
37. E. Athanasopoulos.H. Bos,G. Portokalidis, and E. Goktas. Out of Control Overcoming Control-Flow Integrity[C]. IEEE Symposium on Security and Privacy.(SP’14),2014:575-589.
38. Lu K , Song C , Lee B , et al. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks[C]. In Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS '15). ACM, 2015.
39. GIUFFRIDA C, KUIJSTEN A, TANENBAUM A S. Enhanced operating system security through efficient and fine-grained address space randomization[C]. Presented as Part of the 21st USENIX Security Symposium (USENIX Security 12), 2012:475-490.
40. Chen X , Xue R , Wu C . Timely address space rerandomization for resisting code reuse attacks[J]. Concurrency and Computation, 2017, 29(16):1-14.
41. CHEN Y, WANG Z, WHALLEY D, et al. Remix: on-demand live randomization[C]. The sixth ACM Conference on Data and Application Security and Privacy, 2016: 50-61.
42. HAWKINS W, NGUYEN-TUONG A, HISER J D, et al. Mixr: flexible runtime rerandomization for binaries[C]. The 2017 Workshop on Moving Target Defense. 2017:27-37.
43. BIGELOW D, HOBSON T,RUDD R, et al. Timely rerandomization for mitigating memory disclosures[C]. ACM Sigsac Conference on Computer & Communications Security, 2015.
44. 雷啸. 内存信息泄露的运行中随机化防御方法的研究与改进[D]. 南京: 南京大学, 2017.
45. MORTON M, KOO H, LI F, et al. Defeating zombie gadgets by re-randomizing code upon disclosure[C]. International Symposium on Engineering Secure Software and Systems, 2017:143-160.
46. V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity[C]. In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, 2014:6-8.
47. Hedayati M , Gravani S , Johnson E , et al. Hodor: intra-process isolation for high-throughput data plane libraries[C]. In the Proceedings of the 2019 USENIX Annual Technical Conference, 2019:10-12.
48. Park S , Lee S , Xu W , et al. libmpk: Software Abstraction for Intel Memory Protection Keys [J]. 2018.
49. Linux. The Linux Kernel Archives[OL]. https://www.kernel.org/doc/html/v5.4/x86/intel\_mpx. [html](https://www.kernel.org/doc/html/v5.4/x86/intel_mpx.html).
50. N. Burow, X. Zhang and M. Payer, SoK: Shining Light on Shadow Stacks[J], 2019 IEEE Symposium on Security and Privacy (SP), 2019:985-999.
51. Ben Niu and Gang Tan. Modular control-flow integrity[C]. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14), 2014:577-587.
52. P. Akritidis, C. Cadar, C. Raiciu, M. Costa and M. Castro, Preventing Memory Error Exploits with WIT.[C]. IEEE Symposium on Security and Privacy (sp 2008), 2008:263-277.
53. N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses[C]. In USENIX Security Symposium, 2014:20-24.

# 致 谢

忙碌的读研生活就要结束了，而我也将再一次经历别离，两年时光里我遇到了许许多多博学可爱的人，在这邻近毕业之际我也只能通过这短短的几句话来表达我对身边可敬之人的感谢

首先我要感谢我的论文指导老师汪老师，感谢老师在论文完成过程中的给予的帮助。其次我要感谢实习单位的领路人应师兄，师兄在我实习的过程中给予了极大的帮助，教授了我许多的知识，在我们平常的学习，论文的撰写都给予了许多宝贵的意见。我也感谢科大给了我学习的机会，在以后的生活中我会一直会以是科大学子而感到自豪，最后感谢各位百忙之中参加评审答辩的老师。

而今我坐在这窗台前，迎着清晨第一缕阳光写下这最后一句话，再见了我18年的学生生涯。

2021年10月28日