中国科学技术大学

专业硕士学位论文

（专业学位类型）



**RISC-V指令集模拟器的设计与实现**

作者姓名： 王昊

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： 二〇二二年一月二十八日

University of Science and Technology of China

A dissertation for master’s degree

（Professional degree type）



**Design and implementation of RISC-V instruction set simulator**

Wang Hao

Software Engineering

Prof. Wang Zengfu

Prof. Hou Rui

January 28, 2022

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 系统开发背景

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。

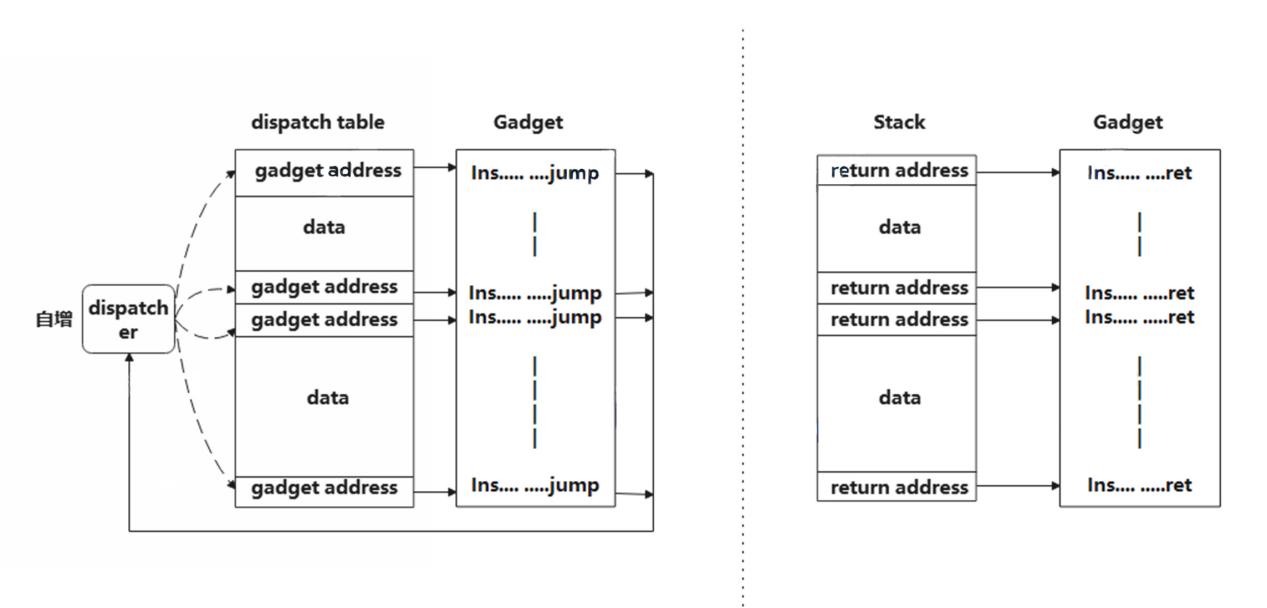


图1.1 JOP和ROP攻击原理（图中左为JOP攻击，右为ROP攻击）

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 本文的主要工作

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 硬件设计与验证中的软件仿真

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。

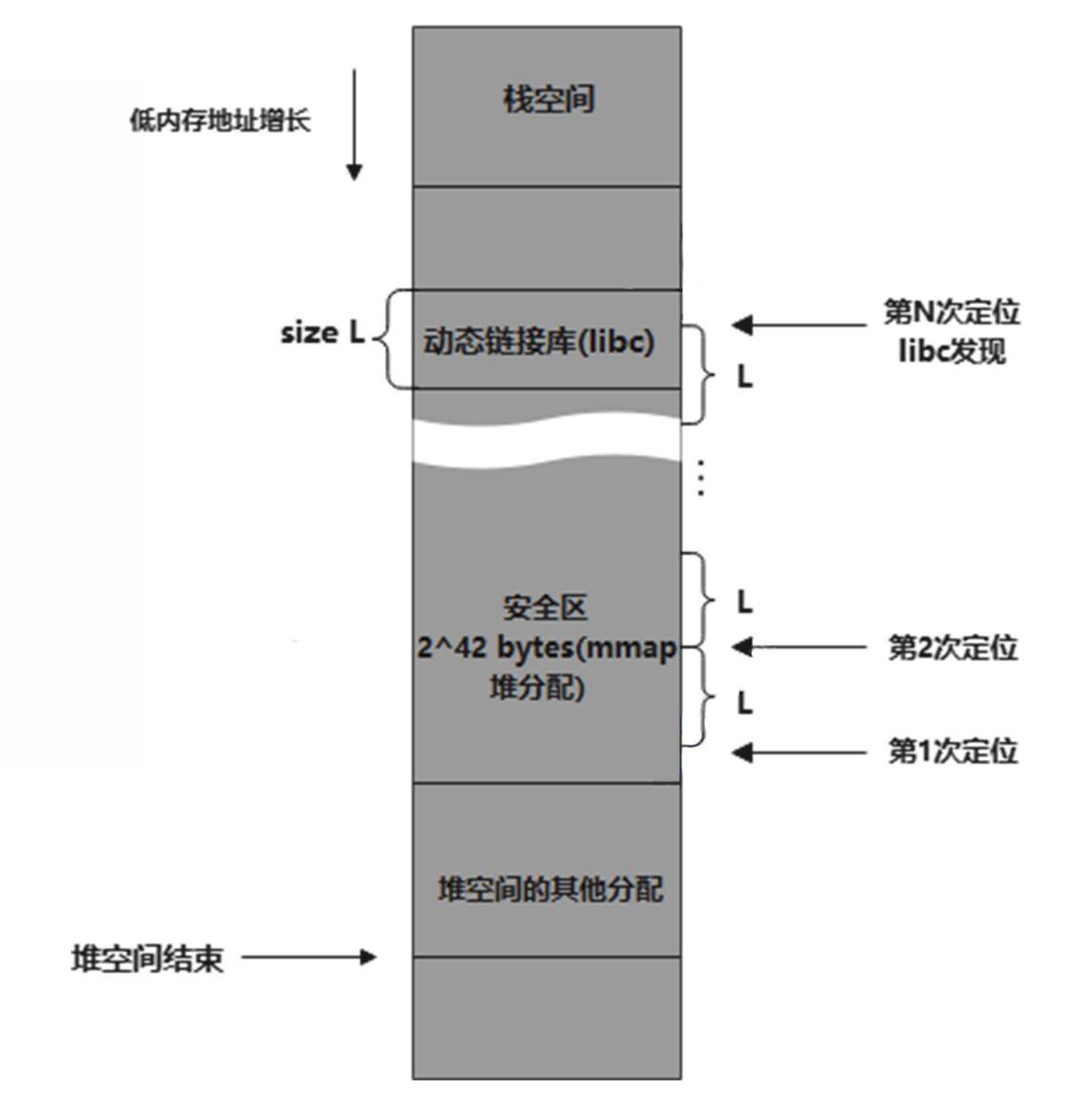


图1.2 安全区攻击流程

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。

# 相关技术分析

由于本系统是基于RISC-V指令集的体系结构模拟器,目标是模拟处理器执行RISC-V汇编指令前后的软硬件行为,故本章首先介绍RISC-V指令集架构相关的内容以及体系结构模拟器的相关技术.

## 2.1 RISC-V介绍

第五代精简指令集(RISC-V, Reduced Instruction Set Computer - Five)是由加州大学伯克利分校的David.Paterson教授团队研发的一套指令集架构,其设计初衷是为了支持计算机体系结构的研究和教育,如今RISC-V已经成为行业实施的标准免费开源指令集架构

体系结构模拟器理论基础，硬件仿真器、软件模拟器，模拟开发流程图，解释型模拟spike/Gem5(取值译码执行流程图)、编译型模拟qemu(流程图)，RISC-V简介，相关开源社区工具介绍。

## 2.2 体系结构模拟器

模拟器是体系结构量化分析的重要手段，对架构设计、芯片开发有重要的指导作用．基于模拟器辅助进行集成电路设计可以追溯到1980年代，自此模拟器便一直是处理器设计过程中不可或缺的工具．在芯片开发过程中，体系结构模拟器可以缩短处理器的设计时间，降低开发成本，其具体作用如图１所示：

由图１可知：

1. 在芯片开发早期，基于模拟器可以进行微结构探索和粗粒度微结构定义，此时模拟器的开发抽象层次较高．
2. 随着处理器设计的不断推进和模拟器的不断完善，基于模拟器可以持续对芯片微结构进行评估、修改和取舍．
3. 当模拟器趋于成熟，可以对微结构、多核互联系统、一致性协议等进行详细性能分析，基于分析结果对微结构进行微调．
4. 在对处理器逻辑设计进行验证的阶段，模拟器可以作为参考模型辅助进行验证，可以快速定位逻辑设计错误．
5. 在未流片之前基于模拟器就可以开展系统软件开发和适配工作，这样可以在芯片流片结束后以最快速度启动系统软件．
6. 流片结束后，基于模拟器可以辅助进行芯片硅后验证环境的搭建以及测试用例编写工作．为了保证模拟器可以顺利辅助进行处理器设计，在整个芯片开发过程中，需要持续对模拟器进行校准，通过持续对比模拟器和寄存器传输层(Register-Transfer Level, RTL)之间的差别，可以互相校准并发现模拟器或者 RTL的设计错误.

模拟开发的基本流程如图2.1 所示。 进行模拟，首先通过对实际硬件系统建模来将之具体化。建模与具体化的过程中必须保证所建模型的结构与实际硬件系统相近或一致，以确保所建模型的精确性，只有精确度较高的模型才能真实的模拟出硬件系统的行为，最终获得正确的结果。在对硬件系统的建模过程中，需要考虑所选择的算法是否合适。评价一种算法是否合适的准则在于是否符合模拟的要求和硬件系统的特征。为了保证最终的模拟精度，必须确保所选择算法的精度够高，稳定性够好。选定合适的算法后，进行程序设计，即用程序语言将模型描述出来。待确定程序模型的正确以后，就可以用这个模型来进行模拟实验，得到相应的结果。最后分析模拟结果，结果分析既可以是针对模型本身的数据，对模型本身进行评价或研究；也可以是对模拟的目标系统性能作出评价。

体系结构模拟器的驱动方式主要分为两种：执行驱动(又叫程序驱动)和踪迹驱动,这两种驱动方式对应的指令集模拟策略分别是编译型模拟和解释性模拟.

### 2.2.1 执行驱动模拟

随着体系结构的研究越来越深入，其设计的复杂度越来越高，这给模拟器的性能带来了一定的负面影响。为了提高模拟性能，研究者们对模拟过程进行了改进，引入了编译的思想——采用一次翻译多次执行的方法来缩短模拟过程中指令译码的时间。译码就是将目标指令翻译成宿主机可识别的指令，宿主机通过这些指令来完成对目标机状态的操作。根据译码过程处于编译还是运行时，编译型指令集模拟器又可分为静态编译型指令集模拟器（Static Compiled ISS）和动态编译型指令集模拟器（Dynamic Compiled ISS）。 由 Zhu and Gajski给出的静态编译型指令集模拟器将本处于运行时的指令译码过程转移至编译时，如图 2.5 所示。目标机二进制代码经编译器编译，之后由代码生成器优化生成宿主机的二进制代码，并最终运行于宿主机。

该技术有一个缺点，那就是要求程序代码为静态的，这就使得静态编译技术只能为一小部分数字信号处理器所使用。与典型的 DSP（数字信号处理器）应用不同的是，微处理器通常面向的是完整的操作系统，而操作系统有一个重要的特点——程序代码在运行时是动态的，这就局限了静态编译型模拟器的广泛使用。

动态编译型指令集模拟器的典型代表为 Embra[33]及 Shade[34]，其工作流程如图2.6 所示：

其基本思想是复用译码信息，即在内存中取出一条指令后，判断该条指令是否是第一次执行，若是，那么对其进行译码，并在译码完成后将译码信息保存到Cache 中，然后执行；若不是，则直接在 Cache 中调用该指令的译码信息执行。由于该技术在程序运行时进行指令译码，因此很难进行代码优化。

对于执行驱动模拟，存在两种不同的观点：一种认为执行驱动模拟器输入的是程序的二进制文件（可执行指令），而不是踪迹信息，因此，模拟器的输入集大小只与程序的静态指令数相关，而与动态指令数无关。如图 2.3(a)所示，由模拟器来执行程序的所有可执行指令。还有一种观点认为，执行驱动模拟器的输入信息（程序的可执行指令）分成两部分：一部分由宿主机代为执行，另一部分则由模拟器执行。如图 2.3(b)所示，这类模拟器只执行与所研究问题相关的指令，其它指令则由宿主机代替执行。这样做的优点在于模拟速度较快，但是这种模拟方式要求宿主机与目标机的指令集体系结构一致，这就限制了它的应用范围。

执行驱动模拟器将用高级语言编写的应用程序经编译和链接后得到的二进制文件作为输入信息，在模拟过程中会模拟系统的动态特征，如动态指令的生成和分支预测等，因此更加接近目标系统的真实情况，精确度更高。然而，也正是由于需要模拟目标系统的动态特征，使得执行驱动模拟器的模拟速度较踪迹驱动模拟器更慢。

### 2.2.2 踪迹驱动模拟

长期以来，由于工作原理简单和模拟精度高，解释型指令集模拟器受到了广泛的关注（如 Gem5、SimpleScalar 等）。解释型 ISS 最大的特点在于直接将硬件行为映射到软件[31]，从而模拟出真实的硬件环境。由于其对指令进行逐条翻译，使得指令的执行可以很好的被控制。解释型指令集模拟器的工作流程很简单，通常是取指（fetch）－译码（decode）－执行（execute）的循环，如图 2.4 所示。

1. 取指：从模拟器的内存空间中取出一条指令.
2. 译码：对取出的指令进行翻译，得到指令的操作码、操作数、执行函数等信息.

3) 执行：执行指令所对应的执行函数，并修改相应寄存器的值，包括 PC 寄存器.

解释执行的工作流程使得解释型 ISS 设计原理比较简单，易于设计和实现，且灵活性较好，模拟精度高。由于其对指令进行逐条翻译然后执行，使得其可以很容易的实现调试机制。但是也正是由于需要对指令进行逐条翻译，导致模拟器需要在译码阶段花费大量的时间，所以解释型指令集模拟器的模拟速度一般不是很高。

踪迹驱动模拟将每条指令顺序执行所产生的所有信息作为模拟器的输入，从而模拟某种体系结构处理器的功能和性能。其模拟过程如图 2.2 所示。

假设要模拟的目标系统为 T，目标系统上执行的应用程序为 w，则模拟 T 中 w执行的问题可以分解为两个部分：生成踪迹信息以及输入踪迹信息进行模拟。如果把生成踪迹信息的系统命名为 G，那么，踪迹信息就是 G 执行 w 时产生的所有信息。接下来将踪迹信息作为模拟器的输入，在目标系统 T 中模拟执行 w。踪迹驱动的优点在于比较简单，而且可以不关心其它无关部分而只对局部进行详细的模拟。但是这种方式要求生成踪迹的系统 G与模拟的目标系统 T的体系结构相似，否则由 G 生成的踪迹信息不能正确反映 w 在 T 上的实际运行情况，从而导致获得错误的模拟结果。踪迹驱动的另外一个缺点在于模拟器的输入信息是静态的，不能对目标系统的动态特征进行研究。

## 2.3 本章小结

# 系统需求分析

需求分析是软件生产周期中的一个重要环节,本章将采用面向对象分析的方法对体系结构模拟器的需求进行具体分析与建模.明确模拟器所需实现的功能性需求和非功能性需求.

## 3.1 需求导出

近几十年来信息技术飞速发展，对芯片的各种要求越来越高，各种新型处理器不断面世。当今社会是以芯片为基石的信息社会，芯片的开发关系到信息生活和服务的各个方面。芯片的设计需要考虑其是否能够具有足够的市场竞争力，这与支持这款芯片的指令集架构的软件种类数量密切相关。例如，Inter x86 架构的服务器和个人主机，在经过多年的发展，如今在商用和家用领域具有很高的市场占有率，其配套的软件的种类和数量在这么多年发展下，也已经十分丰富[1]。考虑到成本和利润因素，芯片厂家在设计新的芯片时偏向于选用具有成熟生态环境的指令集架构。

中国 CPU 设计行业起步较晚，即使经过几十年的发展，它们仍未占据技术的制高点，自主指令集架构处理器的生态环境还没有完善。国内也还没有相对成熟的指令集架构，很多芯片的设计需要靠国外的技术授权，这不仅极大的提高芯片设计的成本，还会受到授权厂商的制约，市场竞争力不强，而且信息安全也有很大的问题。所有这些都极大地限制了国产自主架构处理器的应用和推广。而且，芯片设计属于核心技术，与国家战略有关，国家和科研机构都迫切需要采用独立自主的指令集架构的处理器，来消除中国在电子信息领域和国防信息安全等方面的问题。

Fpga/流片上的系统软件移植测试：软件交叉编译（vmlinux 5min），粘贴fsbl，vivado平台烧录（15min）/流片验证就是烧写平台（20min），xilinx SoC，启动bbl，内核启动，然后只有单步命令行调试，或者jtag。

软硬件行为的模拟，忽略一些指标如缓存加速等硬件性能指标类的测试。主要关注寄存器状态，内存状态，中断系统工作情况，调试交互信息等。

功能性需求：前端设计(gui、text/寄存器/内存/pc指令流、交互/设置断点/中断下发/内存查询/调试过程、保存快照)；指令集注册/解析(riscv-opcodes)；单条指令step前后的寄存器/内存状态转移(指令对应的功能函数/主要的行为模拟)/即指令的单步执行模拟(这部分是重点，包括寄存器和其他存储部件的模拟)/指令流执行的驱动方式；中断系统模拟(clint时钟/软中断；plic外部中断)；调试模块的设计，主要是在功能函数前后进行断点检查(针对寄存器/内存状态的检查)

非功能性需求：可拓展性，UI易用性，速度，模拟精度，代码膨胀率

在芯片设计及验证的流程中，对于基础系统软件尤其是操作系统，底层驱动等的适配和验证往往是反馈硬件设计缺陷最频繁的部分，这部分的工作不仅是对于前期硬件设计的重要测试，也是后续用户态程序开发的基础。对于系统软件的移植和适配工作，有两种主流方式，一种是在模拟芯片硬件特性的FPGA开发板上仿真，另一种是通过软件模拟。两种方法各有利弊，FPGA开发板更加接近真实硬件环境，能够获取精确的仿真信号，但是速度相对较慢，并且能够提供的调试信息较少。而模拟器环境下的开发，其运行速度接近宿主机，并且调试方便，虽然信号精度与真实硬件有差异，但是能够在测试的前期反馈大部分的缺陷。所以真实的开发和测试流程一般是先使用模拟器验证，再上FPGA平台仿真，这样既能够提高开发效率，又不失精度.

随着RISC-V开源社区的日益壮大，更多的芯片设计厂商选择RISC-V作为其指令集架构，在芯片的验证过程中，软硬件适配工作作为测试的重点，往往需要模拟器环境的支持，当前开源社区的指令集模拟器spike由SiFive公司维护，能够定期地更新RISC-V的最新特权级指令，是各厂商优先考虑使用的基础模拟器。厂商可以根据自身产品特性，拓展spike支持的设备，并在此基础之上进行软件移植工作和前期软硬件设配工作，以此来提高芯片验证与测试工作的效率。因此，模拟器的优化与拓展，应该在硬件层面上与待验证芯片的硬件特性相符合，又要在软件层面上满足调试功能易用性的要求。

RTL级，register transfer level，指的是用寄存器这一级别的描述方式来描述电路的数据流方式，rtl级可以理解为，可以直接给综合工具生成你要的网表的代码。

目前RISC-V开源社区的指令集模拟器是spike，由SiFive公司维护，spike模拟实际代码执⾏过程中的软硬件⾏为，提供指令级别的仿真，本质上就是C++程序模拟每条指令执行过程中的软硬件行为，当前最新版本的spike模拟的设备包括总线，内存，时钟，处理器，和调试模块，这些部分能够使得spike运行一个简单的代理内核，并运行一些简单的RISC-V指令集架构的程序。但是对于真正进行RISC-V芯片设计工作的团队来说，spike模拟的内容还远远不够。真实的设计要复杂得多。

spike模拟器的运行过程包括三个部分，模拟器自身部分，面向开发者(用户)的串口调试部分，以及目标程序部分。目标程序一般指的是移植到RISC-V平台的通用基础软件，如linux内核，处理器所支持外设的驱动程序等，这部分程序涉及到的资源就是模拟器应该支持的部分。因此，本次课题涉及到的需求分为两个部分，一个就是对上述功能模块提供模拟器支持的需求，另一部分是优化串口调试模块的需求。

## 3.2 分析建模

指令集模拟器的主要参与者是进行系统软件开发和移植的程序员,通过对实际芯片开发验证过程的分析和归纳,得出模拟器所需要的主要功能有:

1. 设置模拟器启动配置,包括elf文件路径添加,指令集模块注册,运行模式选择等.
2. 模拟器执行流程控制.包括正常运行模式下的uart串口交互,暂停执行进入调试模式,模拟器重启.
3. 调试功能.在调试模式下,进行断点设置,内存查询,历史指令查询,单步执行等.
4. 模拟外部中断信号发送.

综上所述可以得出如表1.1所示的用户需求描述表.

|  |  |  |
| --- | --- | --- |
| 名称 | 参与者 | 说明 |
| 模拟器配置并启动 | 系统软件开发/移植程序员 | 设置模拟器启动参数并运行 |
| 切换至调试模式 | 系统软件开发/移植程序员 | 模拟器从运行模式切换为调试模式 |
| 切换至运行模式 | 系统软件开发/移植程序员 | 模拟器从调试模式切换为运行模式 |
| 重启模拟器 | 系统软件开发/移植程序员 | 重新加载当前配置项并运行 |
| 断点设置 | 系统软件开发/移植程序员 | 调试模式下进行断点添加/移除 |
| 内存查询 | 系统软件开发/移植程序员 | 调试模式下对虚拟地址/物理地址内容查询 |
| 中断信号发送 | 系统软件开发/移植程序员 | 点击外部中断源按钮,发送对应的外部中断到plic |

根据用例描述表可以得出系统软件开发/移植程序员的用例图如图1.1所示

下面分别对系统软件开发/移植程序员的五个主要用例进行详细描述.

|  |  |
| --- | --- |
| 用例名称 | 模拟器配置并启动 |
| 用例描述 | 设置模拟器启动参数并运行 |
| 触发条件 | 勾选模拟器配置选项,输入elf文件路径 |
| 后置条件 | 模拟器解析配置参数,启动程序 |
| 基本事件流 | 1. 输入elf文件路径 2. 选择启动模式是否为调试模式 3. 其他参数勾选,包括核心数,模拟外设路径等 |
| 异常事件流 | 配置参数错误,启动失败 |

|  |  |
| --- | --- |
| 用例名称 | 切换至调试模式 |
| 用例描述 | 模拟器从运行模式切换为调试模式 |
| 触发条件 | 点击run/stop按键 |
| 后置条件 | 模拟器进入调试模式/模拟器进入运行模式 |
| 基本事件流 | 在运行模式下点击run/stop按键 |
| 异常事件流 | 运行模式下点击run按键/调试模式下点击halt按键 |

|  |  |
| --- | --- |
| 用例名称 | 断点设置 |
| 用例描述 | 调试模式下进行断点添加/移除 |
| 触发条件 | 在调试窗口勾选断点类型,输入断点条件,点击”应用” |
| 后置条件 | 点击run进入运行模式,模拟器运行至断点条件触发调试中断,进入调试模式 |
| 基本事件流 | 1. 调试窗口添加/移除断点 2. 程序运行,触发断点 3. 进入调试模式,打印断点信息 |
| 异常事件流 | 断点信息填写错误导致无效断点条件 |

|  |  |
| --- | --- |
| 用例名称 | 内存查询 |
| 用例描述 | 调试模式下对虚拟地址/物理地址内容查询 |
| 触发条件 | 查询窗口输入内存地址,点击查询 |
| 后置条件 | 输出内存对应地址内容 |
| 基本事件流 | 选择地址类型为虚拟地址/物理地址,虚拟地址需要指定核心,输入16进制地址,点击查询 |
| 异常事件流 | 输入无效地址导致访存失败 |

|  |  |
| --- | --- |
| 用例名称 | mailbox中断信号发送 |
| 用例描述 | 点击外部中断源按钮,发送对应的外部中断到plic |
| 触发条件 | 点击对应中断源的中断信号发送按键 |
| 后置条件 | 模拟器响应中断,执行中断处理程序 |
| 基本事件流 | 1. 点击mailbox中断源发送中断信号 2. 弹出mailbox消息窗口,填写消息,点击发送 3. 系统接受外部中断,执行自定义的中断处理函数,显示在mailbox窗口 |
| 异常事件流 | mailbox中断发送后系统无响应 |

### 3.2.1 串口调试模块

spike本身具备单步调试的功能，但是在模拟真实硬件行为的复杂场景下，命令行式的单步调试显然难以使用，根据以往的调试经验，人性化的UI设计需要涉及到处理器状态寄存器，当前特权级模式，程序运行窗口等信息，还需要设置对应多核的寄存器触发条件，实现mailbox中断主动触发，保存快照方便复现bug等功能。

这部分的功能需求主要有:

1. 对于多核心独立csr寄存器触发条件的支持
2. 对内存/指令格式等触发条件的支持
3. 能够查询历史指令序列执行情况
4. 实时监测状态寄存器，当前特权级模式
5. 主动发送mailbox中断信号

### 3.2.2 平台级中断控制器PLIC

RISC-V核心(hart)包含local中断源和global中断源。只有global中断源可以被PLIC Core响应，通常为I/O设备，PLIC负责将中断源(global interrupt sources)链接到中断对象(interrupt targets)，也就是处理器核心。spike自身不支持PLIC设备，所以对于外部中断的支持只能通过轮询来实现，效率相当低下，并且不符合实际处理器的特性。因此，首先对spike的拓展便是PLIC的支持，对于uart，spi等串口控制器，可以在后续通过设备树直接挂载为PLIC的中断源。多个外设作为独立的中断源，通过PLIC Core仲裁，将外部中断信号传递给相应的核心。PLIC中断控制流程如图1所示。

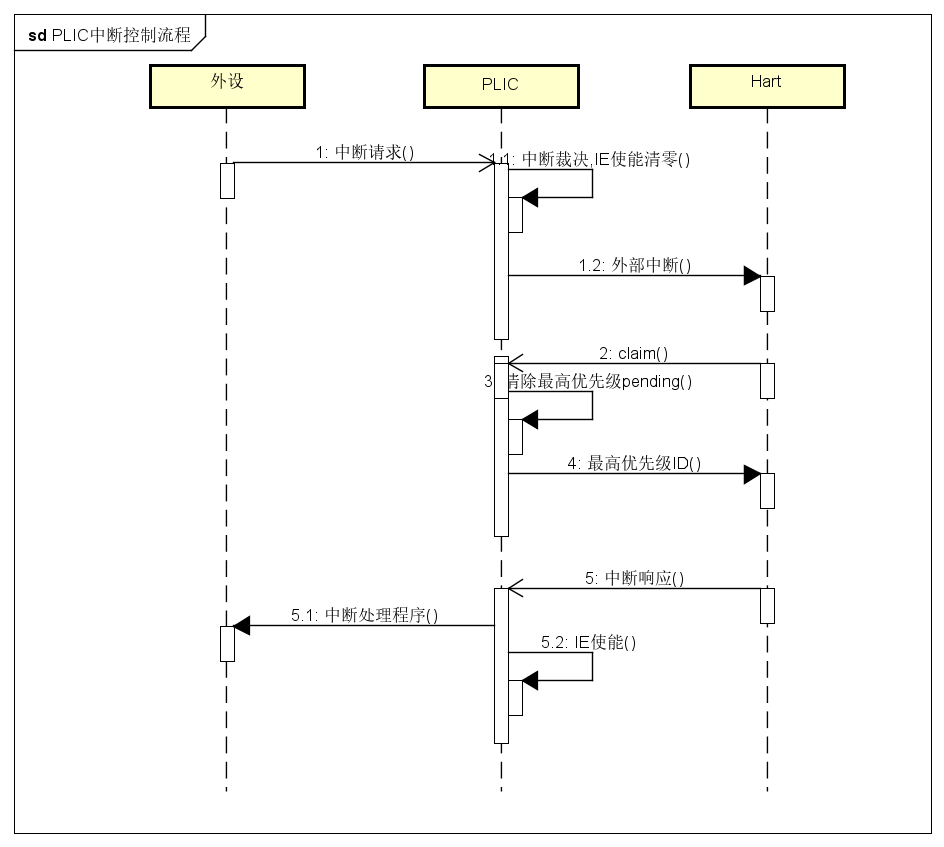


图1 PLIC中断控制流程

对PLIC设备模块的需求是，模拟器启动linux内核，并且uart以中断号2成功挂载到PLIC，通过查看/proc/interrupts进行验证，具体的设计见系统概要设计部分。

### 3.2.3 uart16550串口控制器

spike提供HTIF模拟串口通信控制器, HTIF 是伯克利处理器的非标准工具，因此没有文档。随着 RISC-V 平台规范的发布和内核的更新为自托管，HTIF很快就会消失。HTIF 是主机/目标接口，它通过riscv-fesvr与目标设计（Sodor）通信。riscv-fesvr 通过 HTIF mem 端口将二进制文件加载到 Sodor 内存中，然后通过状态寄存器告知内核。程序完成后，Sodor 告诉 riscv-fesvr，它通过重置主机 CSR 完成交互，模拟结束。

在bootloader加载内核之前,串口通信控制器就必须能够工作,提供裸机的交互功能,是前期进行串口调试的重要工具,能够结合spike的单步调试功能完成前期的大部分调试工作,其实现的优先级很高。

总体来说,HTIF提供了串口通信的功能,但是其本质上还是需要主机的轮询查询,效率很低,考虑到实际硬件设计中必不可少的串口通信控制模块(uart16550a芯片),需要为spike添加uart模拟,通过外部中断的方式实现串口通信.也可以在模拟器上完成对uart的前期测试,包括波特率的设置,传输模式的选择等等.

### 3.2.4 mailbox核间通信模块

由于本次设计涉及到的芯片是一款主动安全处理器芯片,通过和其他厂商计算核心的互联发挥作用,核间的mailbox通信部分至关重要. 区别于ipi处理器内部中断方式, mailbox是一种框架，通过消息队列和中断驱动信号处理多处理器间的通讯.

对mailbox的模拟,需要满足高并发的异步通信需求,鉴于无法真实模拟与另一个计算核心的通信, 需要对模拟器核心间通信添加mailbox支持, 来模拟真实的交互环境.

## 3.3 非功能性需求

该指令集模拟器的非功能性需求有:

1. 准确性: 体系结构模拟器的首要需求就是准确性,只有准确模拟出真实硬件的行为,才能在模拟器上进行后续的软件开发和移植工作.由于本模拟器的模拟精度在指令级别,不涉及到流水线,乱序执行,分支预测等更细粒度的模拟,因此要求模拟器要和真实硬件在寄存器级别完全一致.
2. 可靠性: 可靠性要求模拟器要能够在使用过程中持续稳定运行,不会因为宿主机上程序的设计缺陷导致模拟过程发生崩溃.如果遇到异常情况,模拟器需要能够在不修改启动配置的情况下重启成功,且模拟过程是可复现的.
3. 实时性: 作为一个基于指令集翻译的体系结构模拟器,虽然本身的设计初衷不是为了测试CPU性能,但是模拟器运行速度
4. 友好性:

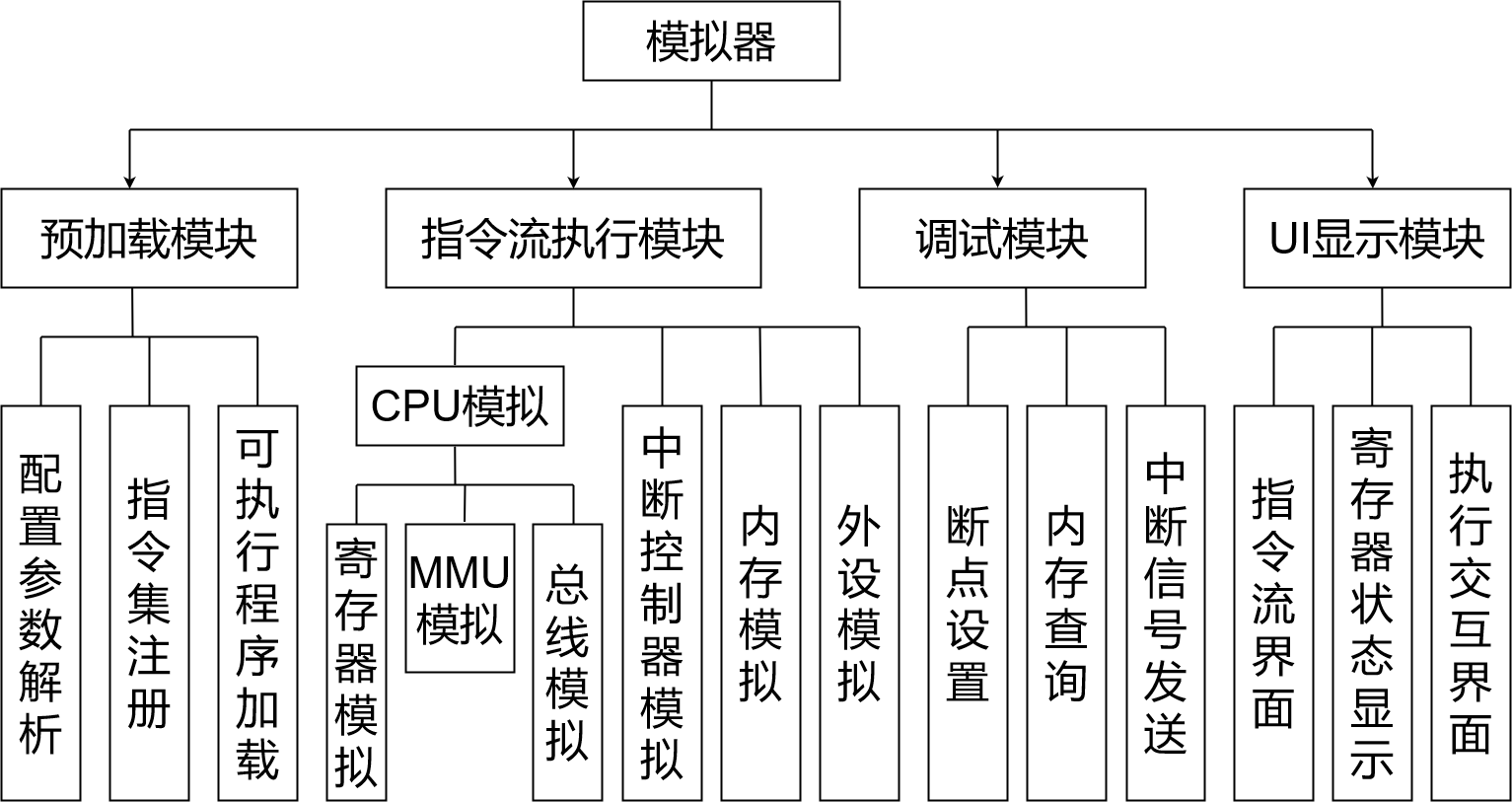
# 第4章 系统概要设计

## 4.1 系统概述

本系统是针对RISC-V芯片开发团队在系统软件开发和移植过程中使用的体系结构模拟器.将编译好的RISC-V架构可执行代码加载到模拟器上运行,观察执行结果,进行系统软件的调试,一方面对系统软件的开发和移植工作做了前期的基本测试,另一方面也有助于发现硬件实现可能存在的缺陷.

指令集模拟(riscv-opcodes)，取值译码执行过程；寄存器/存储器设计；中断系统设计；前端设计。整体各模块层次图。

## 4.2 系统静态结构



## 4.3 系统动态结构

### 4.3.1 设备模拟

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。

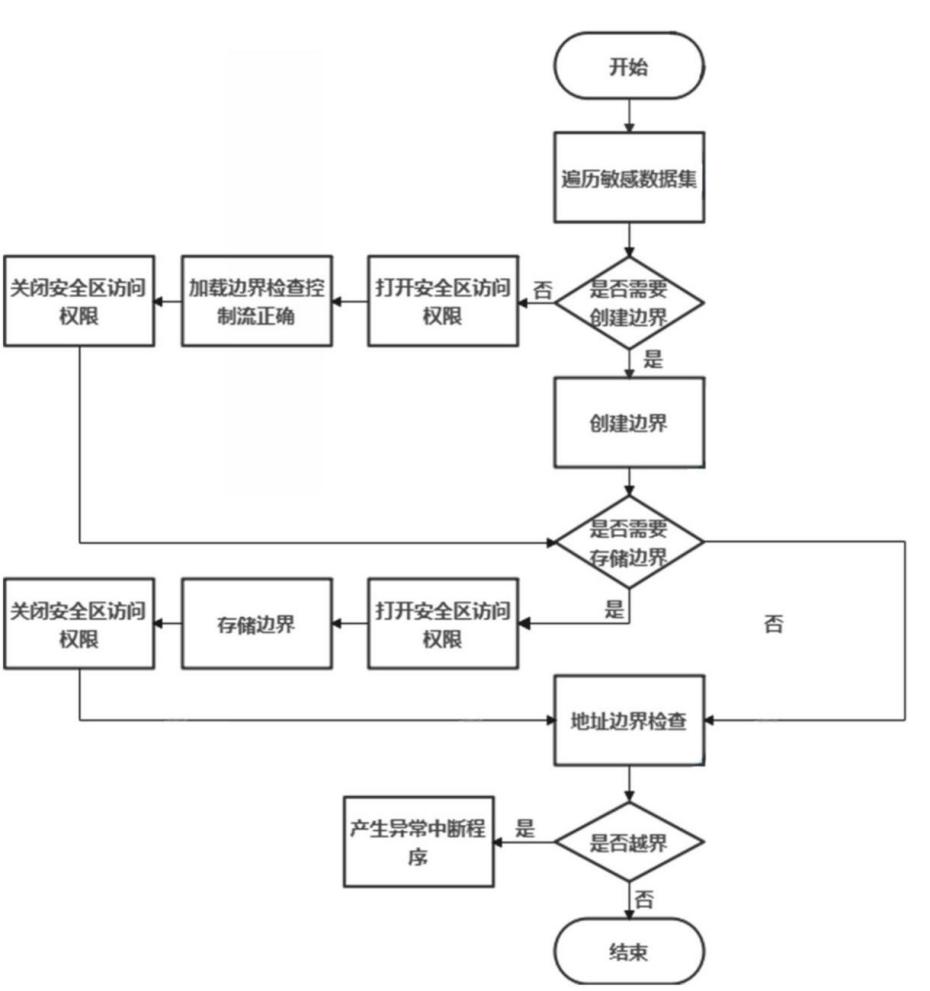


图4.4 动态检查流程

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 内存模拟

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。

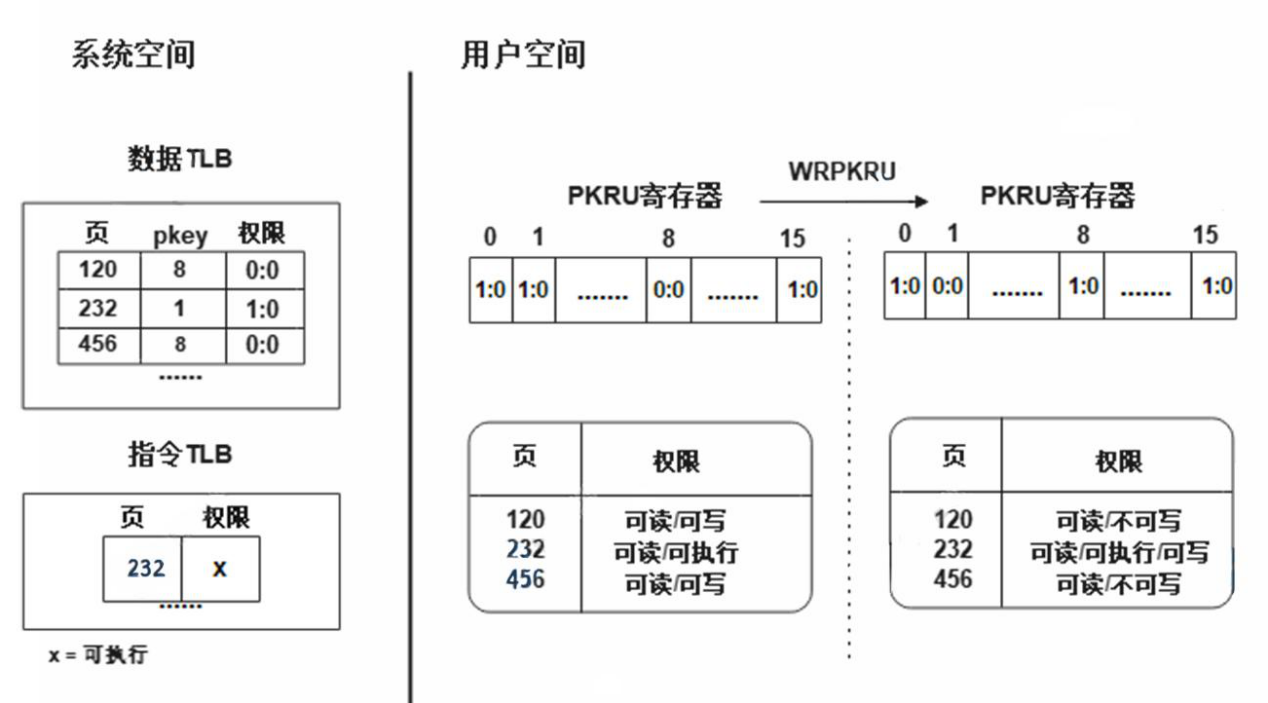


图4.5 MPK内存页权限切换

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。

### 4.3.3 缓存模拟

### 4.3.4

## 4.4 中断模块设计

### 4.4.1 CLINT

### 4.4.2 PLIC

## 4.5 调试模块及前端设计

# 第5章 系统详细设计与实现

## 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。

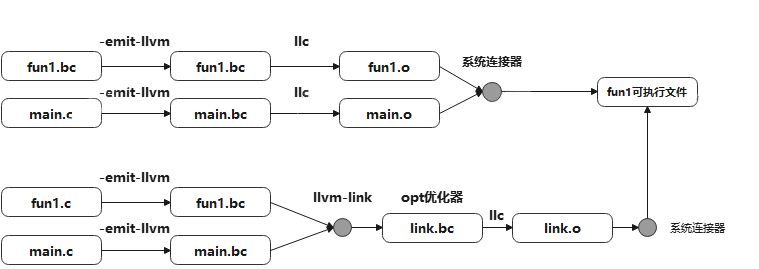


图5.1 pass优化方式

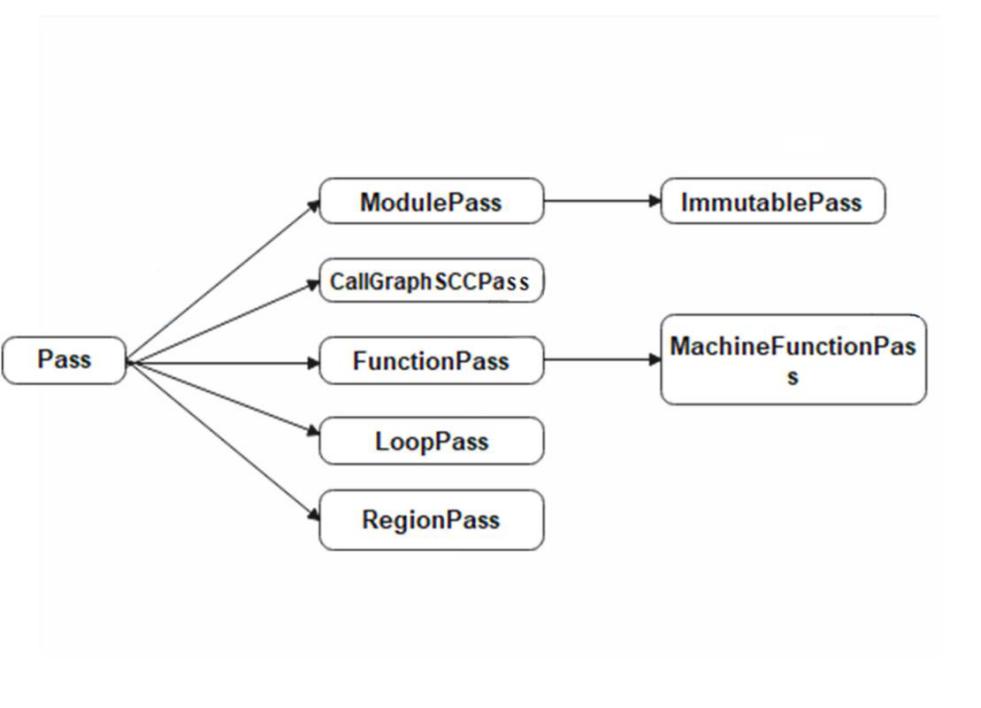


图5.2 pass实现的继承基类

|  |
| --- |
| 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。

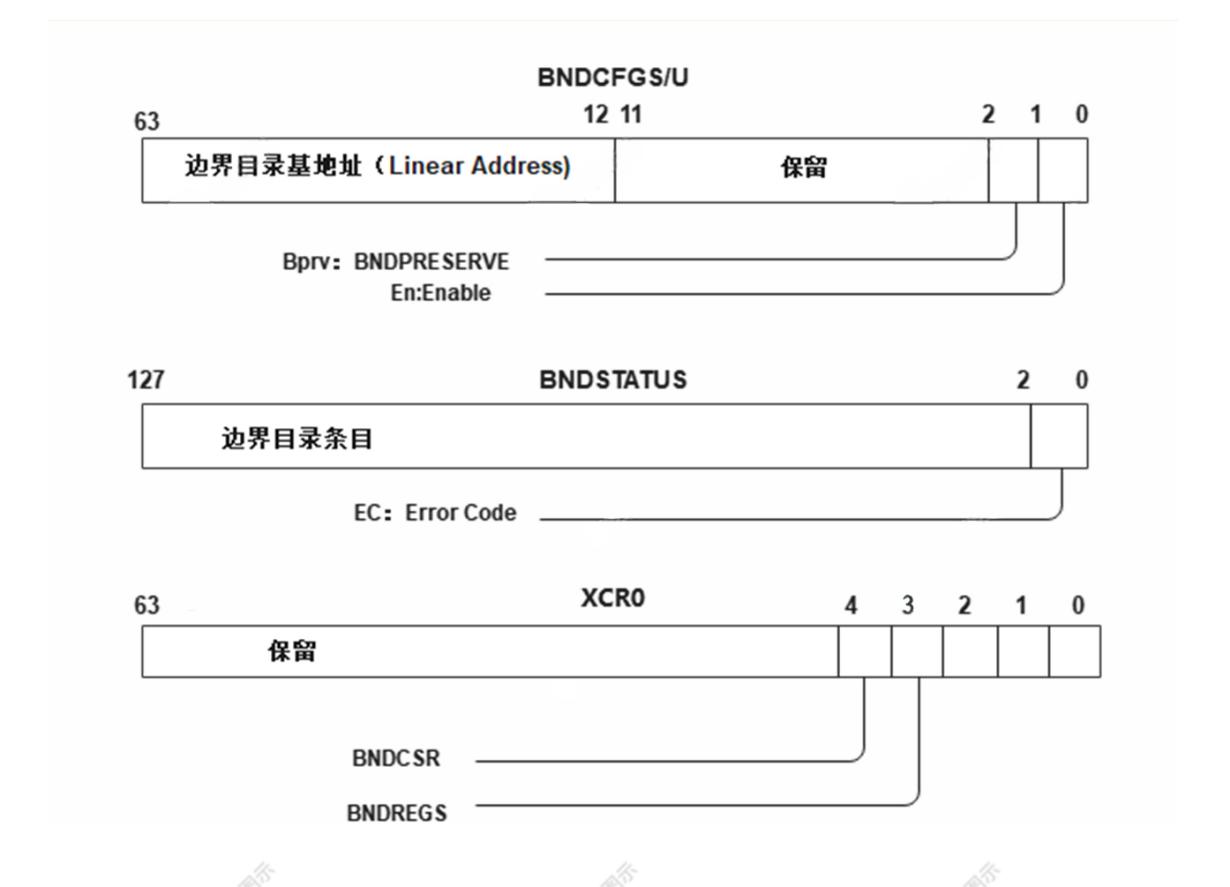


图5.4 初始化寄存器设置

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

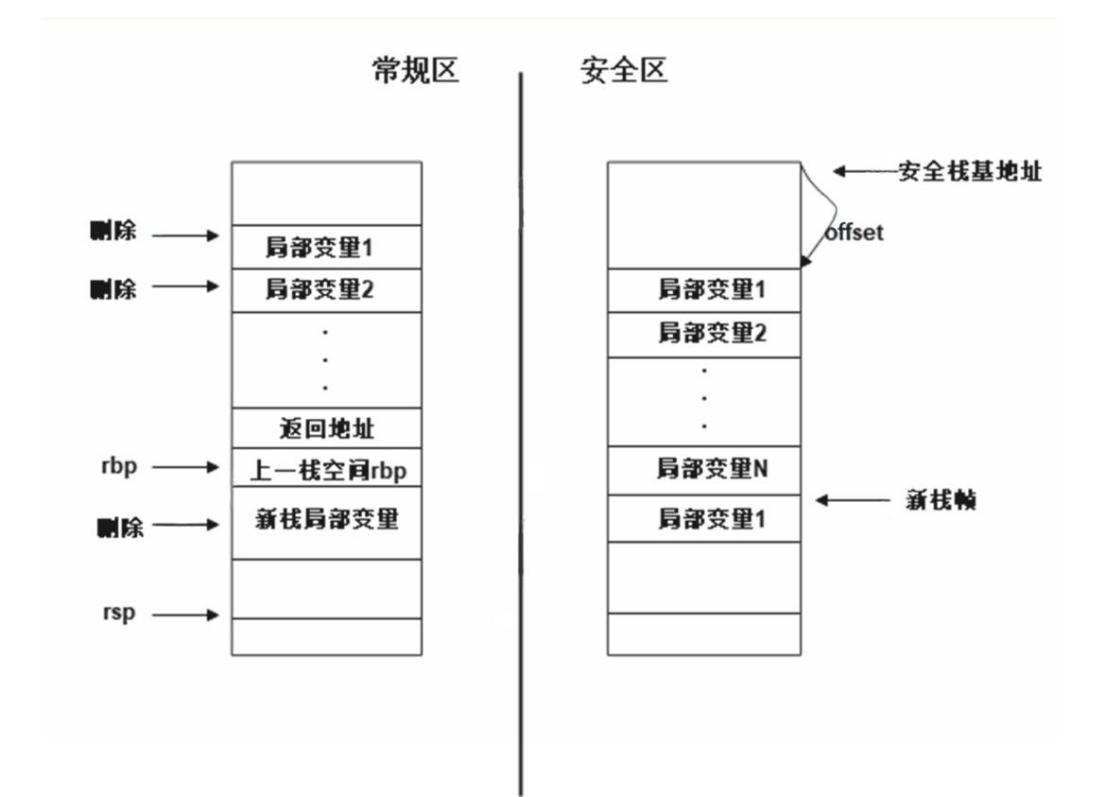


图5.11 安全栈实现原理图

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。

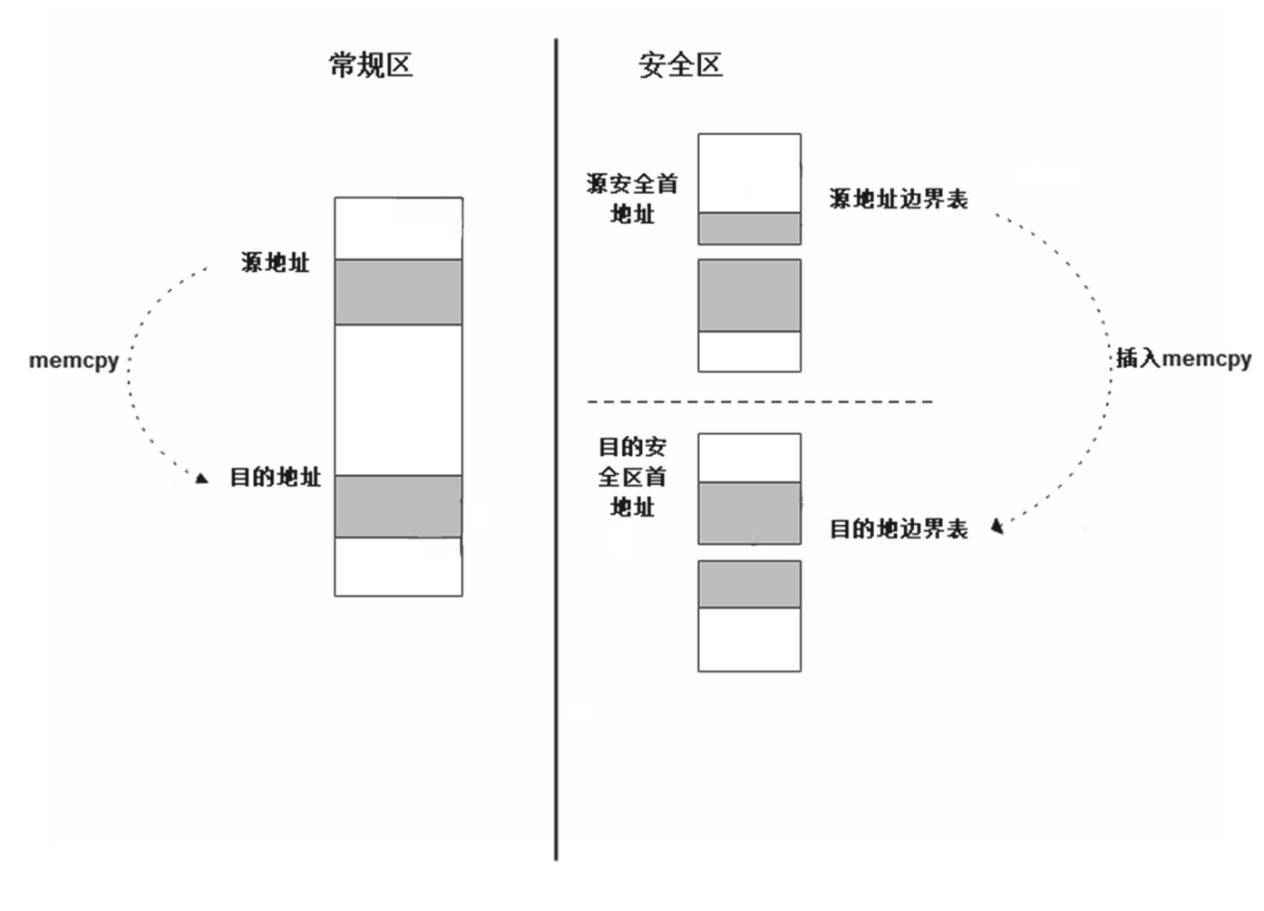


图5.13 连续内存拷贝操作处理方法

## 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。

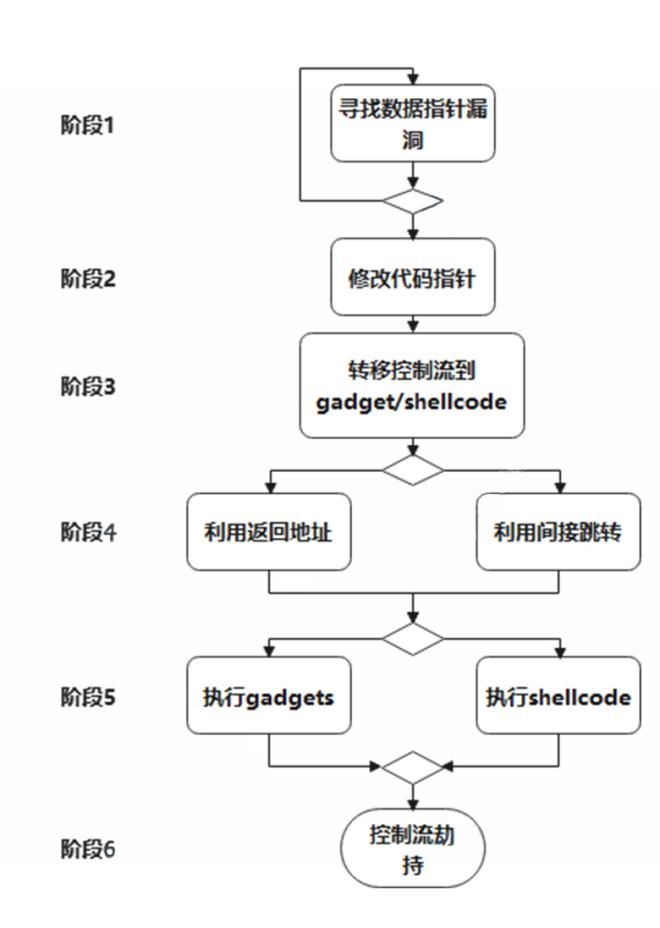


图6.1 代码重用攻击流程

## 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" \t "https://schlr.cnki.net/en/Detail/index/journal/_blank)[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