计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(11):11–20 [doi: 10.15888/j.cnki.csa.009331]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

RISC-V 架构硬件辅助用户态内存安全防御方案概览①

解　达[1,2], 欧阳慈俨[1,2], 宋　威[1,2]

[1](中国科学院 信息工程研究所 信息安全国家重点实验室, 北京 100195)
[2](中国科学院大学 网络空间安全学院, 北京 101408)
通信作者: 宋　威, E-mail: songwei@iie.ac.cn

摘　要: 传统的用户态内存安全防御机制基于 x86 架构和纯软件方式实现, 实现内存安全保护的运行时开销很高, 难以部署在生产环境中. 近年来, 随着主流商业处理器开始提供硬件安全扩展, 以及 RISC-V 等开源处理器架构的兴起, 内存安全保护方案开始面向 x86-64、ARM、RISC-V 等多种体系架构和硬件辅助实现方式. 我们对 RISC-V 架构上实现的内存安全防御方案进行了讨论, 并对 x86-64、ARM、RISC-V 等处理器架构在安全方案设计上的特点进行了比较. 得益于开放的指令集架构生态, RISC-V 架构的内存安全防御方案相较于其他架构有一些优势. 一些低成本的安全防御技术有望在 RISC-V 架构上实现.

关键词: RISC-V; 内存安全; 硬件安全扩展; 处理器

引用格式: 解达,欧阳慈俨,宋威.RISC-V 架构硬件辅助用户态内存安全防御方案概览.计算机系统应用,2023,32(11):11–20. http://www.c-s-a.org.cn/1003-3254/9331.html

Summary of Hardware-assisted User-mode Memory Safety Defenses on RISC-V Architechture

XIE Da[1,2], OUYANG Ci-Yan[1,2], SONG Wei[1,2]

[1](State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100195, China)
[2](School of Cyber Security, University of Chinese Academy of Sciences, Beijing 101408, China)

Abstract: Traditional x86-based and software-based user-mode memory safety defenses can hardly be deployed in a production-ready environment due to significant runtime overheads. In recent years, as mainstream commercial processors begin to provide hardware security extensions and open-source architectures like RISC-V rise, hardware-assisted memory safety protections have become popular, and their implementations are based on various architectures, such as x86-64, ARM, and RISC-V. This study discusses user-mode memory safety defenses on the RISC-V architecture and compares the features of x86-64, ARM, and RISC-V in the context of security defense design. RISC-V has some advantages over other architectures due to its opening ecosystem, making the implementation of some low-cost and promising defense techniques possible.

Key words: RISC-V; memory safety; hardware security extension; processor

1　引言

使用 C、C++等非内存安全语言编写的用户态程序经常存在内存安全漏洞[1,2], 容易受到内存安全错误的影响, 为程序带来严重的安全隐患. 传统的内存安全

① 基金项目: 国家自然科学基金 (61802402, 62172406); 中国科学院率先行动“百人计划”青年俊才 (C 类)
本文由“RISC-V 技术及生态”专题特约编辑邢明杰高级工程师、宋威副研究员、张科正高级工程师以及易秋萍副教授推荐.
收稿时间: 2023-05-26; 修改时间: 2023-06-27, 2023-07-03; 采用时间: 2023-07-21; csa 在线出版时间: 2023-09-15
CNKI 网络首发时间: 2023-09-18

防御机制如栈金丝雀 (stack canary)[3], SoftBound[4] 等通过纯软件方法提供内存安全性保护. 然而纯软件方法的实现要么防御简单, 容易被攻击者绕过; 要么性能代价过高, 难以在生产环境中部署.

近年来, 随着主流商业处理器开始提供硬件安全扩展, 以及 RISC-V 等开源处理器架构的兴起, 内存安全保护方案的设计开始逐渐从单一的 x86 架构和纯软件实现方式过渡到 x86-64、ARM、RISC-V 等多种架构, 软硬件相结合的实现方式. 由于边界检查和元数据查找等耗时操作可以通过硬件加速实现, 硬件辅助防御方案的性能开销相较于安全性相近的纯软件方案大幅降低. RISC-V 作为新兴开源处理器架构的代表, 和传统的处理器架构相比在内存安全防御方案的设计上存在一些优势:

首先, RISC-V 采用精简指令集架构 (reduced instruction set computer, RISC), 指令之间的耦合程度低, 指令集架构的定义简单清晰. 作为新兴指令集, RISC-V 架构没有历史包袱, 更方便进行安全加固.

其次, RISC-V 指令集架构拥有开放的生态. 相较于闭源的 x86 和 ARM 指令集架构, RISC-V 架构允许设计者对处理器和指令集进行改动, 从而可以更加灵活地进行软硬件工作的划分. 此外, RISC-V 指令集架构采用模块化设计, 除了几个必要的基本模块, 其他模块都是可选的. 模块化设计使得 RISC-V 指令集架构具有很强的可扩展性. 安全方案的设计可以快速迭代. 与此相对, x86 和 ARM 架构的指令集架构生态相对封闭, 不允许私有扩展的公开商业应用. 这也导致上述架构的新安全特性从提出到实现需要较长时间.

最后, 支持 RISC-V 指令集架构的处理器普遍具有开源特性. RISC-V 指令集拥有大量的开源片上系统实现, 降低了设计者修改硬件设计的难度, 设计者搭建实验平台的代价较低, 方便对防御方案进行验证和测试. 与此相对的, 在 x86 和 ARM 指令集架构上设计内存安全防御方案要么直接依赖商业处理器提供的硬件安全特性, 要么需要设计者通过软件模拟的方式实现硬件设计.

得益于上述优点, 设计者在 RISC-V 架构上进行安全方案设计时, 具备了同时协调软件设计和指令集架构设计的能力. 设计能力的变化也为内存安全防御方案的设计带来了新的机遇. 本文将对这些软硬件协同的内存安全防御方案进行讨论. 接下来的部分组织如下: 第 2 节介绍内存安全问题, 以及内存安全性最重要的两个方面, 空间安全性和时间安全性. 第 3 节介绍内存安全防御机制的分类, 以及硬件支持为防御机制设计带来的变化. 第 4 节介绍适用于 RISC-V 架构的内存安全防御方案. 第 5 节讨论指令集架构对内存安全防御方案设计的影响, 并对 x86 和 ARM 指令集架构下的方案进行讨论. 第 6 节对硬件辅助内存安全防御方案设计的发展方向进行了讨论. 第 7 节给出结论.

2 内存安全的讨论范围

本文讨论的防御方案主要面向使用硬件辅助实现用户态内存安全保护的场景, 目的是通过硬件加速以较低开销防止用户态应用程序内存漏洞的利用. 场景假设可以被利用的内存漏洞广泛存在于现有用户态软件中, 内存安全问题所讨论的攻击一般在用户态地址空间对受害者进程进行, 其目的是获得对部分内存地址的访问和修改的能力. 由于内存安全防御方案不能对微体系架构提供保护, 一般不考虑攻击使用瞬态执行攻击的情形. 基于隐私计算技术的内存安全保护、针对操作系统内核的内存安全保护等防御方案使用的威胁模型和传统用户态模型不同, 不在本文所讨论的范围.

内存安全性指用户程序中的指针仅访问预期要访问的内容, 并且访问的内容是有效的[5]. 它包含两个属性: 1) 预期性: 内存对象指针访问的内容不超过对象的地址空间, 表现为内存的空间安全性; 2) 有效性: 内存对象指针的访问处于对象的生命周期内, 表现为内存的时间安全性.

违反内存安全性的行为会导致内存安全性错误, 主要包括两类: 1) 空间安全性错误: 内存对象指针访问的内容不在预期对象地址空间的范围内. 典型的例子是缓冲区溢出; 2) 时间安全性错误: 内存对象指针在对象的生命周期外访问对象. 典型的例子是释放后使用错误 (use-after-free, UAF)[6]. 攻击者可以利用上述两种错误获得对部分内存地址的读写能力. 具体表现为内存泄漏和内存篡改. 两种能力作为基础可以构建更加复杂和高级的攻击. 如信息泄露[7]、特权级提升[8]、代码注入[9]、控制流劫持[10,11] 等.

3 安全防御机制

内存安全防御方案侧重于在程序运行时为程序提供内存安全保护. 传统的内存安全防御方案实现完整的时间和空间安全性保护需要较高的运行时开销, 因

此主要针对内存安全性的特定方面, 或是攻击者利用内存安全漏洞攻击路径的特定步骤进行防御. 近年来由于硬件支持加入了内存安全防御方案的设计, 针对特定方面或特定攻击步骤设计的安全防御方案的开销进一步降低. 同时, 设计者能够以较低的运行时开销实现相对完整的时间和空间安全性保护. 部分方案的性能开销[12,13] 甚至能够与特定目的的保护方案相媲美.

一些常见的内存安全防御方案防御类型如表 1 所示. 空间安全性保护限制对象指针访问的内容在对象的地址空间范围内. 一般通过对指针进行边界检查实现保护. 时间安全性保护限制对象指针访问的内容在对象的生命周期内. 可以通过延迟重用[14]、锁-秘钥机制[15] 等方式实现. 空间和时间安全性保护分别针对空间和时间安全性错误提供完整的防御; 下列防御方案则将保护目标集中于内存安全性的特定侧面, 针对攻击者利用时间空间安全性错误执行的特定攻击步骤展开防御: 指针完整性保护通过数据隔离或机密等手段阻止攻击者读取或篡改指针的值. 可以细分为代码指针完整性保护[16] 和数据流完整性保护[17]. 指针完整性是内存安全性的重要组成部分. 随机化保护通过在程序运行时随机化代码与数据的地址, 阻止攻击者获得目标对象地址. 权限保护通过设置页面的读写执行等权限对页面内对象提供保护. 地址空间随机化[18] 和数据执行保护[19] 是以低运行时开销实现有效防御的经典方案.

传统方法倾向于使用纯软件方式实现内存安全防御, 主要面向 x86 架构, 解决的主要问题是如何以相对较低的性能代价换取较高的安全性. 传统方法[4,15,20] 依赖于编译器以及二进制分析和标注实现, 因此性能开销普遍较高.

近几年内存安全防御方案[12,21–23] 开始倾向于使用软硬件结合的方法, 依赖商业芯片提供的硬件内存安全性扩展或修改开源硬件, 在 x86-64、ARM、RISC-V 等多个指令集架构上实现. 使用硬件降低了防御方案的运行时开销, 但同时防御方案对硬件特性的依赖导致了兼容性问题. 此外, 硬件的计算资源有限, 限制了防御方案能够提供的安全性强度. 新方法的主要问题是如何在实现和软件方案相近强度的安全性防护的同时, 降低运行时开销, 同时尽可能保证和传统代码的兼容性.

表 1 常见的内存安全防御类型

防御类型	传统软件方案	硬件辅助方案
空间安全性保护	SoftBound[4]	CHERI[21], SHORE[24]
时间安全性保护	CETS[15]	Shakti-MS[25], HWST128[26]
指针完整性保护	代码指针完整性[16]	STAR[27], ARM PA[22]
随机化保护	地址空间随机化[18]	Morpheus[28]
权限保护	数据执行保护[19]	SealPK[29], Donky[30]

4 RISC-V 架构内存安全防御设计介绍

RISC-V 指令集架构为设计者提供了同时协调软硬件设计的能力, 许多新颖的硬件辅助内存安全防御方案在 RISC-V 架构上实现. 这些方案主要可分为基于边界检查、对象身份的防御机制, 基于信息流的防御机制以及基于页面权限的防御机制. 下面依次对这些分类进行介绍.

4.1 基于边界检查与对象身份的防御机制

边界检查机制通过在对象创建时记录对象的基地址和大小信息, 在访问对象内容时检查指针访存地址在基地址和大小构成的地址区域内, 提供对内存对象空间安全性的保护. 对象的边界信息一般在对象创建时生成, 跟随对象指针一同传递, 并在对象指针访存时用于执行检查[21–25,31,32]. 基于对象身份的防御机制可以提供额外的对象身份信息, 在指针访存时验证指针与被访问对象身份, 实现内存时间安全性保护[26,33]. 近年来基于 RISC-V 架构设计的基于边界检查和对象身份识别的内存安全防御方案如表 2 所示.

表 2 RISC-V 架构基于边界检查和身份识别的内存安全防御方案

防御方案	安全性保护	边界存储模式	边界检查形式	边界传播支持	运行时开销百分比 (%)
CHERI[21,33]	时空安全性	胖指针	多种指令	流水线	6.8
SHORE[24]	空间安全性	分离	专用访存指令	流水线	37.69
HWST128[26]	时空安全性	分离	专用访存指令	流水线	94.89
Shakti-MS[25]	时空安全性	胖指针	专用检查指令	流水线	13
HeapSafe[31]	时空安全性	内嵌索引	专用检查指令	协处理器	22.4
In-Fat Pointer[32]	时空安全性	内嵌索引	多种形式	流水线	12

4.1.1 边界存储模式

胖指针防御方案[21,25] 通过扩展指针长度, 将边界信息保存至指针中. 扩充后的指针长度一般为 128 位, 对边界信息和指针地址部分进行了压缩. 硬件需要对胖指针在流水线和内存中的传播提供支持. CHERI[21] 在胖指针的基础上增加了对象类型信息, 并通过增加密封位和使用标签内存实现了不可修改指针, 提供了对“最小权限原则”的支持. Shakti-MS[25] 在栈、堆和只读数据段的低地址部分放置 64 比特随机值 Cookie, 用于标记受保护对象地址空间的下界. 胖指针方案一般用于提供内存空间安全性保护. 通过与软件结合[33] 或使用胖指针保存额外的元数据[25], 可以提供对内存时间安全性的保护. 实现内存时间安全性的方法主要包括延迟重用和锁-秘钥机制. Shakti-MS 通过 Cookie 计算密钥信息, 与边界信息一同保存至胖指针中.

分离存储方案[24,26] 将边界信息单独存储, 通过对象指针的地址部分或指针的地址寻找对应的边界信息. 分离存储方案需要使用额外的指令处理边界信息的创建、传播和销毁. 这种方案对边界信息的大小和数量没有限制, 但是在通过指针检索或传递边界信息时存在运行时开销. SHORE[24] 将指针地址部分和边界信息分开存储, 在流水线中将边界信息存储至影子寄存器中, 在内存中将边界信息存储至影子内存中, 指针地址部分仍然按照传统指针形式存储. HWST128[26] 在 SHORE 基础上, 实现了锁-秘钥机制, 向影子寄存器中加入了秘钥和指向对象锁的指针.

内嵌索引分离存储方案[31,32] 同样采用分离存储方式保存边界信息, 但是将边界信息的索引内嵌在指针中. 这种方案免去了边界信息随指针传播的运行时开销, 但是边界信息的数量有限. HeapSafe[31] 从程序的静态列表中获取边界信息的标签; In-Fat Pointer[32] 将对象分为 3 个类别, 为每个类别设计了单独的边界信息存储模式.

4.1.2 保护目标

边界检查机制一般面向栈、堆以及全局数据等对象提供内存安全保护[24,25,32]. 由于针对堆对象的攻击较为常见, 部分机制[31,33] 面向堆对象提供保护. 由于硬件资源有限, 多数边界检查机制并不在子对象粒度提供内存安全保护. 一方面是由于实现子对象粒度保护的设计比较复杂, 另一方面是部分大型程序存在使用对象内部跨子对象指针的情况[21]. In-Fat Pointer 通过引入布局表描述复合类型对象的内存布局, 提供了对子对象粒度的内存安全保护. CHERI 使用特殊选项对源码重新编译后能够提供子对象粒度的内存安全保护, 但部分大型程序无法正常运行.

4.1.3 对指令集的修改

边界检查方案可以通过扩展访存指令的语义[32] 或增加专用的指令[21,24,26,31] 执行边界检查. 专用指令需要通过修改编译器替代普通访存指令或在访存指令之前插入. SHORE、HWST128、CHERI 提供了支持边界检查的访存指令; HeapSafe 增加了专门用于边界检查的指令; In-Fat Pointer 将边界检查操作加入普通访存指令的语义中. 为了支持边界信息在流水线中传递, 边界检查方案需要提供专用指令和专用寄存器实现边界信息的传递、加载和存储操作. CHERI 和 In-Fat Pointer 提供了大量指令对能力机制或子对象粒度内存安全性提供支持.

4.1.4 对微体系结构的修改

边界检查方案一般会通过硬件实现加速边界检查操作. 边界检查操作可以通过流水线内的边界检查单元实现同步检查[24–26,32], 也可以通过 RoCC 等协处理器实现异步检查[31]. 异步检查可以降低边界检查的运行时开销, 但是存在非法访问成功执行到异步检查报错的时间窗口. 对于边界信息与对象指针分离存储的方案, 需要对边界信息的检索过程进行加速. 加速过程可以通过地址缓冲[32] 或缓存实现. HeapSafe 通过内容关联存储 (CAM) 保存边界信息, 因此对于边界信息的存储数量存在限制.

4.1.5 运行时开销

胖指针防御方案和内嵌索引分离存储方案使用硬件实现边界信息传递[21,25,31,32], 运行时开销较低; 分离存储方案传递和检查边界信息需要访问影子内存[24], 运行时开销较高; 使用锁-密钥机制实现时间安全性保护需要额外维护时间安全性信息, 也会增加一部分运行时开销[25].

4.2 基于信息流的防御机制

基于信息流的防御机制主要通过对指针或数据提供完整性保护, 覆盖内存安全性保护的一个子集, 可以以较低的运行时开销防御特定的内存安全攻击. 目前在 RISC-V 上的实现大致可分为 3 个类型: 静态信息流保护, 动态信息流保护以及基于标签内存的保护. 近年来基于信息流在 RISC-V 架构上实现的防御机制如表 3 所示.

表 3 RISC-V 架构基于信息流的内存安全防御方案

防御方案	信息流类型	信息流内容	安全性保护	运行时开销百分比 (%)
RvDfi[34]	静态	数据流图	数据流完整性	17.8
FineDIFT[35]	动态	数据类别	指针完整性	5
HyperFlow[36]	动态	机密性, 完整性	系统调用完整性	12
Dover[37,38]	标签	规则定义	规则定义	最高415[38]
OptTAG[39]	标签	特殊标记	空间安全性	55
HDFI[40]	标签	敏感标记	代码指针完整性	2
STAR[27]	标签	指令和指针状态	指针完整性	2.74
Morpheus[28]	标签	数据类别	随机化保护	0.84

4.2.1 信息流防御分类

静态信息流保护方案[34,41] 在软件编译时生成静态的控制流图或数据流图, 通过硬件辅助在运行时保证代码指针或数据指针的使用限制在控制流图或数据流图内. 一般用于实现控制流完整性或数据流完整性保护. RvDfi[34] 作为典型的硬件辅助实现的静态数据流保护方案, 将数据流图保存于特殊的数据结构中, 在运行时检查每个内存地址的读写指令是否与数据流图匹配.

编译时静态生成的信息流由于缺乏运行时信息需要进行过度近似, 导致静态信息流的精度较低[35]. 动态信息流保护方案[35,36] 通过动态信息流跟踪技术提供高精度、运行时的信息流图. 基于纯软件方式实现的动态信息流保护方案需要通过编译器或二进制重写工具插入大量实现信息流传递的代码, 性能开销较高[42]. 硬件辅助的动态信息流保护方案通过硬件加速信息流传播和保存等方式降低性能开销. 根据信息流传播、检查策略的不同, 动态信息流保护方案可以保护敏感数据不受不可信数据的影响, 也可以用于信息流隔离. FineDIFT[35] 以内存区域为单位隔离代码指针、数据指针和普通数据. HyperFlow[36] 通过使用超立方体标签模型维护不同信息流的机密性和完整性关系.

另外一些保护方案[27,28,39,40] 依赖标签内存技术实现. 标签内存对内存中的机器字进行扩展, 用于存储标签. 流水线需要扩展寄存器宽度, 并增加对标签规则的支持, 实现标签在流水线中的传递. 标签内存支持指令标签和数据标签, 结合标签规则可以实现对数据流和控制流的完整性保护[27]. 此外标签内存也可以用于标注程序中的特殊数据, 实现特殊的安全特性[21] 或加速软件安全方案的实现[39]. Dover 处理器[37] 使用软件定义标签规则硬件结合执行的方案, 能够通过标签规则[38] 实现较完整的内存时空安全性保护; OptTAG[39] 通过使用标签标记需要边界检查的数据指针对软件边界检查方案 SoftBound[4] 进行加速; HDFI[40] 使用标签进行敏感数据隔离; STAR[27] 通过指令标签和数据标签实现对指针完整性的保护; Morpheus[28] 为不同类别的数据分别进行地址随机化, 并定期进行随机化流动. Morpheus 通过标签实现了代码、指针与其他数据的分类, 并用于加速随机化流动过程.

4.2.2 信息流的表示

静态信息流保护方案通过编译器将信息流图编码为指令或数据结构. 方案为信息流图的每个顶点分配唯一的身份标识. 身份标识保存在特殊的数据结构中, 或是直接由指令携带. 信息流图的边通过身份标识的使用顺序表示. RvDfi 通过程序中每条读指令对应的有效写指令集合表示数据流, 插入检测代码验证读写指令是否匹配. 动态信息流保护方案对于信息流有多种表示形式[34]. FineDIFT 以内存区域为单位对信息流进行跟踪, 使用包含边界信息的元数据表表示信息流[35]; HyperFlow 使用一对比特向量表示特定机密性和完整性等级的信息流[36]. 基于标签内存的保护方案一般通过 1–6 比特宽的标签表示信息流. HDFI、OptTAG 等方案使用 1 比特标签进行数据隔离, 或标注特殊数据; Morpheus 使用 2 比特标签对数据进行分类; STAR 使用 4–6 比特标签表示指针类别和关键指令的状态; Dover 处理器标签位宽依赖于软件定义的标签规则. Morpheus、STAR 等方案通过修改缓存结构支持标签在缓存中的存储[27,28].

4.2.3 信息流传递和检查规则

静态信息流保护方案通过验证身份标识的使用顺序是否对应信息流图中的边维护信息流的完整性. 动态信息流保护方案和基于标签内存的方案需要定义信息流从指令的源寄存器到目的寄存器、从寄存器到内存的传播规则, 实现信息流传播对指针算术运算、访存等程序语义的支持. 信息流传递和检查规则一般由方案直接定义, 并在硬件上实现; Dover 处理器支持软件定义标签规则并由硬件执行, 但是运行时开销较高.

4.2.4 运行时开销

静态信息流保护方案通过硬件实现加速了信息流图检查操作. 硬件辅助方案 RvDfi 的性能开销仅为纯软件方案的 1/9[34]; 动态信息流保护方案的开销主要来自对信息流的跟踪和维护. 对于依赖标签内存技术的方案, 其运行时开销取决于具体的标签策略; 维护固定标签策略的方案[27,28,39,40] 会对特定策略进行优化, 运行

时开销一般低于相同功能的基于 Dover 处理器等可编程标签策略方案的实现[38].

4.3 基于页面权限的防御机制

基于页面权限的防御机制[29,30,43,44] 主要通过扩展页面内存权限的管理实现内存安全性保护. RISC-V 架构下实现的基于页面权限的防御方案如表 4 所示. 防御方案通过为特定内存范围的权限进行控制, 保护范围内数据不被泄露、篡改或恶意执行. 防御方案一般通过扩展或优化指令集架构提供的页面权限管理机制实现. 传统的页面内存权限由操作系统或底层硬件机制管理, 用户程序要修改权限需要通过特殊的系统调用实现, 调用开销较高; 部分防御方案实现了用户态页面权限控制, 和传统的页面权限管理机制共存, 以较低的开销提供精细控制. 不过用户态页面权限控制的粒度较粗, 用户态权限控制指令容易受到控制流劫持和注入攻击 [29], 需要特殊设计[30] 或结合控制流保护方案进行部署.

表 4 RISC-V 架构基于页面权限的内存安全防御方案

防御方案	硬件权限管理机制	改进和优化	运行时开销百分比 (%)
RIMI[43]	RISC-V PMP	DMP	0.88
SealPK[29]	类似Intel MPK	增加权限域数量	14.81
Donky[30]	类似Intel MPK	实现进程内隔离	4
ROLoad[44]	任意页面权限控制机制	保护指针完整性	约为0

RISC-V 指令集架构提供了物理内存保护 (physical memory protection, PMP) 机制, 为特定的内存区域设置读写和执行权限; RIMI[43] 基于 PMP 机制提出了域内存保护 (domain memory protection, DMP) 机制, 通过增加指令实现内存隔离, 阻止代码和数据的跨域访问. Intel 内存保护秘钥技术 (memory protection key, MPK) 实现了用户态可配置的读写权限域; SealPK[29] 对其进行了改进, 增加了权限域的数量并在 RISC-V 上实现. ROLoad[44] 利用只读页不可修改的特性保存敏感数据, 实现指针完整性. Donky[30] 通过用户态监视器实现了内存隔离机制, 利用隔离机制保护对 MPK 的修改, 解决了以往内存保护秘钥方法需要额外的控制流完整性以实现内存隔离的问题. 然而, 上述方案仍然无法对页面内违反时间空间安全性的行为提供有效保护.

5 其他架构的内存安全防御方案

表 5 列举了 x86-64、RISC-V64、ARM AArch64 等常见的 64 位指令集架构的特点.

表 5 常见 64 位指令集架构对比

指令集架构	复杂性	指令长度	硬件实现	内存安全扩展/特性
x86-64	复杂指令集	变长	软件模拟	MPX、CET、SGX和MPK
RISC-V64	精简指令集	定长32位	开源项目	依赖社区
ARM AArch64	精简指令集	定长32位	软件模拟	PA、MTE和BTI

x86-64 采用复杂指令集, 不同指令间耦合较紧密, 增加了指令流水化实现的难度. 变长指令集的使用降低了代码大小的同时, 也为防御方案的专用指令保留了足够的编码空间.

x86-64 指令集对于旧代码有良好的兼容性, 提供硬件内存安全支持的方式主要是通过增加硬件安全扩展或硬件安全特性. 目前基于 x86-64 架构的内存安全防御机制设计多采用软件模拟处理器架构的方式设计, 需要使用等效指令替代等方式进行性能测试.

ARM AArch64 架构采用精简指令集, 不同指令为松耦合关系, 利于指令流水化实现. 定长指令集的使用降低了指令预取和解码的复杂度, 但是用来扩展指令集的编码空间有限. 目前基于 ARM AArch64 架构的内存安全防御机制的研究主要依赖 ARM AArch64 提供的商业即可得 (commercial-off-the-shelf, CoTS) 内存安全特性. ARM 官方提供了芯片仿真模型用于对这些内存安全特性进行实验.

64 位 RISC-V 架构 (RISC-V64) 同样采用定长 32 位精简指令集. RISC-V 架构指令集作为开放指令集, 存在大量的开源片上系统实现. 内存安全防御方案的设计者可以在设计软件的同时对 ISA 和硬件进行修改. 开源实现使得设计者可以使用自定义的硬件实现在 FPGA 上进行仿真, 从而降低了安全方案测试和验证工作的难度. 目前基于 RISC-V64 架构的内存安全防御机制研究主要依赖设计者在开源片上系统的修改. 不过 RISC-V 指令集架构标准中仍然缺少专用于保护内存安全的指令集扩展.

开放生态为 RISC-V 架构带来的另一个好处是 RISC-V 架构可以有选择地吸收其他架构上的安全防御方案设计. 如表 6 所示, 下面对 ARM AArch64 和 x86-64 等架构上的部分安全防御方案进行讨论.

ARM v8.3-A 指令集引入了指针认证 (pointer authentication) 特性, 该特性利用了 64 位地址空间中指针高位可以不参与地址转换过程的特点, 通过对指针加密,

生成特殊的指针认证码 (pointer authentication code, PAC), 放置于指针的高比特位中. 加密后的指针需要经过指针验证去掉 PAC 才能进行内存访问. 指针的加密和验证操作都通过专用指令完成. 加密过程通过硬件加密模块实现, 需要指针值、秘钥和修改器 3 个操作数. 指针认证通过加密方式对指针提供保护. 最初只使用栈帧寄存器 SP 作为修改器, 保护函数调用过程中保存在栈中的返回地址. 但是使用 SP 作为修改器容易受到攻击者控制 PAC 生成和复用加密指针的攻击[22]. Liljestrand 等提出了 PARTS[22], 对修改器的使用进行了加固, 将指针的类型信息加入修改器中, 并扩展了指针认证的使用范围到所有数据指针和代码指针. PACMem[45]、AOS[46] 等安全方案通过指针认证保护对象指针, 并将对象的基地址、大小以及其他元数据保存在表中, 将认证生成的 PAC 作为元数据表项的索引. 这种方法可以以较小的元数据表实现对象的内存时间空间安全性保护, 但是由于指针验证过程比较复杂, 需要硬件进行加速. 此外使用 PAC 作为索引还存在碰撞冲突问题[12,46]. ARM 指针认证移植到 RISC-V 架构需要在 RISC-V 架构上复现地址转换逻辑、硬件加密模块, 不仅工程量大, 复现的性能也可能不如原有架构, 移植的难度较高.

表 6 ARM AArch64 和 x86-64 指令集架构内存安全防御方案

防御方案名	安全性保证	指令集架构	硬件特性	性能开销百分比 (%)
PACMem[45]	时空安全性	ARM AArch64	ARM PA	68.73
PARTS[22]	指针完整性	ARM AArch64	ARM PA	19.5
AOS[46]	时空安全性	ARM AArch64	ARM PA	8.4
染色方法	时空安全性	ARM AArch64	ARM MTE	无记录
Intel MPX[23]	时空安全性	x86-64	Intel MPX	50
CHEx86[47]	时空安全性	x86-64	无	14
No-FAT[13]	时空安全性	x86-64	无	8
HeapCheck[12]	时空安全性	x86-64	无	1.5
ZeRØ[48]	指针完整性	x86-64	无	约为0

ARM v8.5-A 指令集引入了内存标签扩展 (memory tagging extension, MTE), 同样利用了指针高位不参与地址转换的特性, 使用指针高位的 4 比特保存标签, 内存需要对标签内存提供支持. 目前对于 MTE 的应用主要以染色方法为主. 染色方法为对象及其指针分配相同的标签, 作为颜色. 指针携带颜色在程序中传递和计算, 在解引用时验证指针和对象的颜色是否匹配. 染色方法只能提供概率性的内存安全保护, 检测内存安全错误的概率受到标签位数和染色策略的影响[12]. 在 RISC-V 架构上, lowRISC[49] 等已经实现了标签内存支持, 部分安全方案应用更复杂的标签策略提供保护[27].

Intel 在 2013 年提出了 Intel MPX (Intel memory protection extensions)[50], 通过指针内容检索边界信息, 进行边界检查, 实现内存安全性保护. 但是由于早期设计缺陷, 以及较高的性能开销, 最终被废弃[23]. Intel 在 2015 年提出了 Intel SGX (Intel software guard extensions)[51], 提供可信计算环境的硬件支持. Intel 在 2017 年提出了 Intel CET (Intel control flow enforcement technology)[52], 引入了影子栈以及间接分支跟踪技术, 针对 ROP 等高级控制流劫持攻击提供防御. Intel 内存保护密钥技术提供了一种在用户空间控制页面读写权限的方法, 支持用户在不通过 mprotect 系统调用的前提下, 对页面访问权限使用最多 16 个域进行管理[53]. 目前, 在 RISC-V架构上已经出现了对可信计算环境[54]、内存保护密钥技术[29,30] 的替代方案.

近年来出现了一批基于 gem5 处理器架构模拟器[55] 实现的 x86-64 架构内存安全防御方案[12,13,48]. 这些方案的实现不依赖于商业处理器提供的安全扩展, 而是自行设计硬件提供安全支持. No-FAT[13]、HeapCheck[12] 通过硬件辅助边界检查面向堆内存对象提供内存时间空间安全性保护, 运行时开销为 1.5%–8%. ZeRØ[48] 通过使用标签区分代码、数据和指针, 修改缓存结构存储指针类型信息, 以约为 0 的开销实现指针完整性保护. 对于这些不依赖内存安全扩展或特性的防御方案, 移植到 RISC-V 架构所需的修改较小, 设计者也能够控制硬件实现的细节, 方便在原有方案基础上进行进一步改动, 移植的难度较低.

6 可行的研究方向

表 2 列举了 RISC-V 架构基于边界检查的内存安全防御方案的平均性能开销. 表 6 列举了其他指令集架构内存安全防御方案的平均性能开销. 其他指令集架构实现的基于边界检查的内存安全防御方案, 如 HeapCheck, 已经能够以 5% 以下的运行时开销实现对内存空间完整性的保护, 但是在 RISC-V 架构下实现的方案性能开销尚未达到这一水平[21,32,33], 存在优化的空间. 目前的大部分边界检查方案主要针对堆对象提供内存安全性保护; 栈对象和全局对象的内存安全保护仍然

存在讨论空间. 另外, 对于子对象粒度的内存安全性保护, 现有的设计仍然比较复杂. 对于时间安全性保护, 大多通过软件方式或锁-密钥机制提供保护, 性能开销仍然较高[25,26,32].

基于静态信息流的防御方案需要处理静态分析导致的信息流图过度估计问题. 动态信息流保护方案虽然提供了高精度的信息流图, 但是合并不同输入产生的信息流图仍然是一个挑战. 基于标签内存和动态信息流跟踪的保护方案依赖预定义的规则控制信息流的传播和错误检查. 基于软件规则的保护方案能够定义更加灵活的规则, 但是性能开销仍然较高[38]. 标签内存保护方案的内存开销与标签宽度成正比, 如何利用尽可能少的标签位宽实现高安全性的保护方案也是一个挑战.

7 结论

综上所述, 硬件辅助内存安全防御方案的设计在性能开销较低的情况下实现了较完整的内存安全性防护. 但同时也存在与传统代码兼容性较差、提供的内存安全性防护不完整、依赖于特定的分配器实现等问题. RISC-V 架构由于指令集架构开源, 为内存安全防御方案的软硬件协同设计提供了更多的灵活性. 相较于依赖现有商业处理器硬件扩展或纯软件方式模拟 CPU 设计的 x86-64 架构和 ARM 架构的处理器, 大量的开源 RISC-V 架构片上系统项目降低了基于 RISC-V 架构的内存安全防御方案的测试和验证工作的难度. 相较于 x86-64 和 ARM 架构, RISC-V 架构拥有更加开放的生态. RISC-V 架构下的内存安全防御方案设计没有历史包袱, 可以进行更加激进的设计; 同时设计者可以控制硬件实现的细节, 更加深入地将硬件修改整合至体系架构中; 对于其他架构的安全特性, 设计者可以进行进一步优化, 实现在 RISC-V 架构上, 提供性能开销更低、安全性更高的改良版本. 此外, 许多 RISC-V 架构下的硬件辅助内存安全方案应用了标签内存、动态信息流跟踪、能力模型指针等技术, 一些开源方案对这些技术提供了硬件支持. 一些难以在 x86-64/ARM 架构上实现的低开销、高安全性保证的方案有望在 RISC-V 架构上得以实现.

参考文献

1 Stone M. The state of 0-day in-the-wild exploitation. USENIX Association, 2021.

2 Miller M. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. Microsoft, 2019.

3 Cowan C, Pu C, Maier D, *et al*. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. Proceedings of the 7th Conference on USENIX Security Symposium. San Antonio: USENIX, 1998. 5.

4 Nagarakatte S, Zhao JZ, Martin MMK, *et al*. SoftBound: Highly compatible and complete spatial memory safety for C. Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. Dublin: ACM, 2009. 245–258.

5 Song D, Lettner J, Rajasekaran P, *et al*. SoK: Sanitizing for security. Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP). San Francisco: IEEE, 2019. 1275–1295.

6 Lee B, Song CY, Jang Y, *et al*. Preventing use-after-free with dangling pointers nullification. Proceedings of the 22nd Annual Network and Distributed System Security Symposium. San Diego: NDSS, 2015.

7 Strackx R, Younan Y, Philippaerts P, *et al*. Breaking the memory secrecy assumption. Proceedings of the 2nd European Workshop on System Security. Nuremburg: ACM, 2009. 1–8.

8 Chen S, Xu J, Sezer EC. Non-control-data attacks are realistic threats. Proceedings of the 14th USENIX Security Symposium. Baltimore: USENIX, 2005. 177–191.

9 One A. Smashing the stack for fun and profit. Phrack Magazine, 1996, 7(49): 14–16.

10 Solar designer. Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63. (1997-08-10).

11 Wojtczuk RN. The advanced return-into-lib (c) exploits: Pax case study. Phrack Magazine, 2001, 11(58): 4–14.

12 Saileshwar G, Boivie R, Chen T, *et al*. HeapCheck: Low-cost hardware support for memory safety. ACM Transactions on Architecture and Code Optimization, 2022, 19(1): 10.

13 Ibn Ziad MT, Arroyo MA, Manzhosov E, *et al*. No-FAT: Architectural support for low overhead memory safety checks. Proceedings of the 48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA). Valencia: IEEE, 2021. 916–929.

14 Dhurjati D, Adve V. Efficiently detecting all dangling pointer uses in production servers. Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN 2006). Philadelphia: IEEE, 2006. 269–280.

15 Nagarakatte S, Zhao JZ, Martin MMK, *et al*. CETS:

Compiler enforced temporal safety for C. Proceedings of the 2010 International Symposium on Memory Management. Toronto: ACM, 2010. 31–40.

16 Kuznetzov V, Szekeres L, Payer M, *et al*. Code-pointer integrity. Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014). Broomfield: USENIX Association, 2014. 147–163.

17 Castro M, Costa M, Harris T. Securing software by enforcing data-flow integrity. Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation. Seattle: USENIX, 2006. 11.

18 PaX team. Address space layout randomization. http://pax.grsecurity.net/docs/aslr.Txt. (2003-03-15)[2023-05-25].

19 Solar Designer. Linux kernel patch from the Openwall Project (non-executable user stack). https://lkml.iu.edu/hypermail/linux/kernel/9706.0/0341.html. (1997-06-06)[2023-05-25].

20 Wahbe R, Lucco S, Anderson TE, *et al*. Efficient software-based fault isolation. Proceedings of the 14th ACM Symposium on Operating Systems Principles. Asheville: ACM, 1994. 203–216.

21 Watson RNM, Woodruff J, Neumann PG, *et al*. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. Proceedings of the 2015 IEEE Symposium on Security and Privacy. San Jose: IEEE, 2015. 20–37.

22 Liljestrand H, Nyman T, Wang K, *et al*. PAC it up: Towards pointer integrity using ARM pointer authentication. Proceedings of the 28th USENIX Conference on Security Symposium. Santa Clara: USENIX, 2019. 177–194.

23 Oleksenko O, Kuvaiskii D, Bhatotia P, *et al*. Intel MPX explained: A cross-layer analysis of the Intel MPX system stack. Proceedings of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems. Irvine: ACM, 2018. 111–112.

24 Dow HK, Li T, Miles W, *et al*. SHORE: Hardware/software method for memory safety acceleration on RISC-V. Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC 2021). San Francisco: IEEE, 2021. 289–294.

25 Das S, Unnithan RH, Menon A, *et al*. Shakti-MS: A RISC-V processor for memory safety in C. Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems. Phoenix: ACM, 2019. 19–32.

26 Dow HK, Li T, Parameswaran S. HWST128: Complete memory safety accelerator on RISC-V with metadata compression. Proceedings of the 59th ACM/IEEE Design Automation Conference. San Francisco: IEEE, 2022. 709–714.

27 Gollapudi RT, Yuksek G, Demicco D, *et al*. Control flow and pointer integrity enforcement in a secure tagged architecture. Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP). San Francisco: IEEE, 2023. 2974–2989.

28 Gallagher M, Biernacki L, Chen SB, *et al*. Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn. Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems. Providence: ACM, 2019. 469–484.

29 Delshadtehrani L, Canakci S, Egele M, *et al*. SealPK: Sealable protection keys for RISC-V. Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE). Grenoble: IEEE, 2021. 1278–1281.

30 Schrammel D, Weiser S, Steinegger S, *et al*. Donky: Domain keys-efficient in-process isolation for RISC-V and x86. Proceedings of the 29th USENIX Conference on Security Symposium. USENIX, 2020. 95.

31 De A, Ghosh S. HeapSafe: Securing unprotected heaps in RISC-V. Proceedings of the 35th International Conference on VLSI Design and the 21st International Conference on Embedded Systems. Bangalore: IEEE, 2022. 120–125.

32 Xu SJ, Huang W, Lie D. In-Fat Pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 2021. 224–240.

33 Filardo NW, Gutstein BF, Woodruff J, *et al*. Cornucopia: Temporal safety for CHERI heaps. Proceedings of the 2020 IEEE symposium on security and privacy (SP). San Francisco: IEEE, 2020. 608–625.

34 Feng L, Huang JY, Li LY, *et al*. RvDfi: A RISC-V architecture with security enforcement by high performance complete data-flow integrity. IEEE Transactions on Computers, 2022, 71(10): 2499–2512. [doi: 10.1109/TC.2021.3133701]

35 Chen KJ, Arias O, Deng QX, *et al*. FineDIFT: Fine-grained dynamic information flow tracking for data-flow integrity using coprocessor. IEEE Transactions on Information Forensics and Security, 2022, 17: 559–573. [doi: 10.1109/

TIFS.2022.3144868]

36 Ferraiuolo A, Zhao M, Myers AC, *et al*. HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto: ACM, 2018. 1583–1600.

37 Sullivan GT, DeHon A, Milburn S, *et al*. The Dover inherently secure processor. Proceedings of the 2017 IEEE International Symposium on Technologies for Homeland Security (HST). Waltham: IEEE, 2017. 1–5.

38 Dhawan U, Vasilakis N, Rubin R, *et al*. PUMP: A programmable unit for metadata processing. Proceedings of the 3rd Workshop on Hardware and Architectural Support for Security and Privacy. Minneapolis: ACM, 2014. 8.

39 Seo J, Bang I, Cho Y, *et al*. Exploring effective uses of the tagged memory for reducing bounds checking overheads. The Journal of Supercomputing, 2023, 79(1): 1032–1064. [doi: 10.1007/s11227-022-04694-y]

40 Song CY, Moon H, Alam M, *et al*. HDFI: Hardware-assisted data-flow isolation. Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP). San Jose: IEEE, 2016. 1–17.

41 Bellec N, Hiet G, Rokicki S, *et al*. RT-DFI: Optimizing data-flow integrity for real-time systems. Proceedings of the 34th Euromicro Conference on Real-time Systems. Dagstuhl: Leibniz-Zentrum für Informatik, 2022. 18:1–18:24.

42 Kemerlis V P, Portokalidis G, Jee K, *et al*. libdft: Practical dynamic data flow tracking for commodity systems. ACM SIGPLAN Notices, 2012, 47(7): 121–132. [doi: 10.1145/2365864.2151042]

43 Kim H, Lee J, Pratama D, *et al*. RIMI: Instruction-level memory isolation for embedded systems on RISC-V. Proceedings of the 39th IEEE/ACM International Conference on Computer Aided Design. San Diego: ACM, 2020. 1–9.

44 Tan WD, Li Y, Zhang C, *et al*. ROLoad: Securing sensitive operations with Pointee integrity. Proceedings of the 58th ACM/IEEE Design Automation Conference. San Francisco: IEEE, 2021. 307–312.

45 Li Y, Tan WD, Lv ZZ, *et al*. PACMem: Enforcing spatial and temporal memory safety via ARM pointer authentication. Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. Los Angeles: ACM, 2022. 1901–1915.

46 Kim Y, Lee J, Kim H. Hardware-based always-on heap memory safety. Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture. Athens: IEEE, 2020. 1153–1166.

47 Sharifi R, Venkat A. CHEx86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities. Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture. Valencia: IEEE, 2020. 762–775.

48 Ibn Ziad MT, Arroyo MA, Manzhosov E, *et al*. ZeRØ: Zero-overhead resilient operation under pointer integrity attacks. Proceedings of the 48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA). Valencia: IEEE, 2021. 999–1012.

49 Bradbury A, Ferris G, Mullins R. Tagged memory and minion cores in the lowRISC SoC. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.686.2584&rep=rep1&type=pdf. [2023-05-25].

50 Intel. Intel memory protection extensions enabling guide. https://www.intel.com/content/www/us/en/developer/articles/guide/intel-memory-protection-extensions-enabling-guide.html. (2016-01-15)[2023-05-25].

51 McKeen F, Alexandrovich I, Berenzon A, *et al*. Innovative instructions and software model for isolated execution. Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP). 2013, 1–8.

52 Intel. A technical look at Intel's control-flow enforcement technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html. (2020-06-13)[2023-05-25].

53 Park S, Lee S, Xu W, *et al*. libmpk: Software abstraction for Intel memory protection keys (Intel MPK). Proceedings of the 2019 USENIX Conference on USENIX Annual Technical Conference. Renton: USENIX Association, 2019. 241–254.

54 Weiser S, Werner M, Brasser F, *et al*. TIMBER-V: Tag-isolated memory bringing fine-grained enclaves to RISC-V. Proceedings of the 26th Annual Network and Distributed System Security Symposium. San Diego: NDSS, 2019.

55 Binkert N, Beckmann B, Black G, *et al*. The gem5 simulator. ACM SIGARCH Computer Architecture News, 2011, 39(2): 1–7. [doi: 10.1145/2024716.2024718]

(校对责编: 孙君艳)