arXiv:2603.08400v1 [cs.CR] 9 Mar 2026

勿信任何事：无需运行时软件可信计算基的实时操作系统安全性（扩展版）

Eric Ackermann、Sven Bugiel,
CISPA 亥姆霍兹信息安全中心，德国萨尔布吕
肯 {eric.ackermann,bugiel}@cispa.de

摘要

嵌入式设备面临着日益扩大的威胁面：应用软件、操作系统内核和外围设备中的漏洞都会危及嵌入式设备的完整性。现有的计算机体系结构防御机制在其安全模型中，最多只能全面考虑上述威胁向量中的两种。

本文旨在利用一种新颖的能力架构来弥补这一不足。为此，我们结合了一种令牌 能力方法，该方法适用于构建不可信操作系统，并能在无需改动外围设备硬件的情况下防御恶意设备。

首先，我们围绕旧式硬件组件开发并评估了能力架构的完整现场可编程门阵列实现。此外，我们还介绍了一种基于 Zephyr、没有运行时软件可信计算基的软实时操作系统。为此，我们将 Zephyr 的子系统解耦为小型、彼此隔离的组件。所有在运行时存在的子系统，包括调度器、分配器和直接内存访问驱动程序，以及所有外围设备，都完全不可信。我们认为，我们的工作为未来安全关键型嵌入式设备实现更严格的安全设计奠定了基础。

1 引言

得益于计算机体系结构的进步，嵌入式设备中僵化且昂贵的硬件电路正日益被通用、廉价片上系统（SoC）上运行的灵活软件所取代。尽管这种软件化降低了工程成本并缩短了产品上市时间，却增加了嵌入式设备中的安全漏洞，进而危及关键服务的完整性和可用性。漏洞通常存在于应用软件、操作系统内核，或具备直接内存访问（*DMA*）功能的外设中。

嵌入式设备中的应用软件漏洞十分普遍：一项针对打印机的代表性研究发现，在分析的固件映像中，80% 存在已知漏洞 [29]。从更大范围来看，现有的攻击分类体系列出了数千个针对嵌入式设备的已知通用漏洞披露 [57, 70]。利用应用软件中的漏洞可作为跳板，进而攻陷整个设备。为此，攻击者还可利用操作系统内核中的漏洞：在 Zephyr 等实时操作系统中，内核与用户空间的隔离是可选功能，且并非所有兼容的片上系统都支持该功能 [68]。因此，任何软件漏洞都会使攻击者获得对设备的完全控制权。即使在 Linux 中，内核与应用软件通过内存管理单元（MMU）隔离，仍不断发现可用于内核利用的漏洞 [22, 23, 48, 67, 82]。最后，片上系统内部的设备 也可能成为攻击途径：Beniamini [17] 展示了对 Broadcom 移动 Wi-Fi 芯片的无线攻击利用，将一个无害的直接内存访问设备变成硬件木马。Markettos 等人 [50] 展示了一种伪装成无害雷电接口外设的植入物，但它会通过 PCIe 封装发起直接内存访问攻击。

针对这三种威胁向量——应用软件、内核和设备——现有的所有防御措施都只能防范其中两种。通常，内核利用内存管理单元来防范已遭入侵的应用软件和设备。然而，内存管理单元的设计要求信任内核能够正确配置内存管理单元，因此无法实现无运行时软件可信计算基。设计不依赖可信内核的操作系统绝非新想法——事实上，Dennis 和 Van Horn [30] 早在1966年就通过能力架构的概念探讨了这一问题。若干现代能力架构沿用这一思路，以减少甚至消除操作系统中需要信任的部分 [35, 43, 80, 83]。然而，它们都有一个共同的局限：将设备排除在威胁模型之外。因此，它们要么以更大的硬件 *TCB*换取最小化的软件可信计算基，要么需要信任其驱动程序 [24]。因此，现有架构都无法在操作系统和设备同时不可信的情况下实现安全性。

最近，Ackermann 等人 [1] 提出了 Northcape 概念，旨在构建一种新颖的能力架构来弥补这一差距。Northcape 将所有应用软件、内核和设备都视为潜在威胁。其核心思想是在系统总线层级实现能力内存访问控制。然而，在我们的工作之前，Northcape 仅仅是一个理论概念，尚未实现，也未证明其可行性。

贡献 我们的工作旨在回答以下研究问题：嵌入式系统如何提供安全

性，以抵御恶意应用软件、操作系统内核和外设，同时不依赖运行时软件可信计算基并保持软实时保证？我们采用软硬件协同设计来解决这一问题，该设计同时包含 *Bredi SoC* 和 *Skadi RTOS*。

首先，我们将在第 4 节介绍 Bredi（Breði）。Bredi 是 Northcape 的首个硬件实现，基于 cva6 RISC-V 处理器 [85] 构建，配备未经修改的以太网子系统 [4]，并带有面向现场可编程门阵列的直接内存访问 [2],。Bredi 解决了 Northcape 的两项安全性缺陷：首先，与 CHERI-TrEE [72], 类似，Northcape 依赖可信软件进行中断处理。通过修改 cva6，我们消除了这部分可信软件，从而支持不可信中断服务例程。Northcape 也未涵盖实时可用性。借助不可屏蔽中断，我们确保即使存在攻击者，关键服务仍可用。Bredi 还提升了 Northcape 的独占访问安全性，并通过我们新颖的子系统限制解决了能力标签猜测攻击问题。

其次，我们将在第 5 节介绍 Skadi（Skaði）操作系统。Skadi 是经过修改的 Zephyr RTOS [86]，它利用 Northcape 提供的操作，在操作系统组件之间实施运行时隔离，将其划分为子系统，使这些子系统彼此互不信任。借助 Bredi 操作，Skadi 为子系统私有或由子系统共享的所有数据结构提供完整的空间和时间隔离，包括对任意数据结构实施由硬件强制执行的独占访问。子系统调用原语可通过原子地更改可访问能力集合和控制流，实现子系统间的同步函数调用。Skadi 将所有操作系统服务实现为隔离的子系统，因此没有内核，也没有运行时软件可信计算基。外设、驱动程序和中断服务例程也不再属于可信计算基。我们将在不久的将来以开源许可证发布 Skadi 和 Bredi。

最后，我们将在第 6 节和第 7 节讨论原型的安全性和性能。结果表明，在计算基准测试等理想场景中，Skadi 的速度与不安全的引用实现一样快。对于 I/O 密集型场景，尽管 Skadi 和 Bredi 目前会增加响应时间的开销，但系统的性能仍足以满足实际应用需求。由于安全性是本项工作的重点，Skadi 和 Bredi 仍为未来的性能优化留下了空间

在硬件和软件层面。总体而言，我们认为，Skadi 和 Bredi 能够通过为硬件和软件栈实现安全设计，持续改善未来嵌入式设备（例如网络基础设施设备）的安全态势。

2 背景：Northcape 架构

我们首先在第 2.1 节简要介绍对象能力系统。然后介绍 Northcape，涵盖

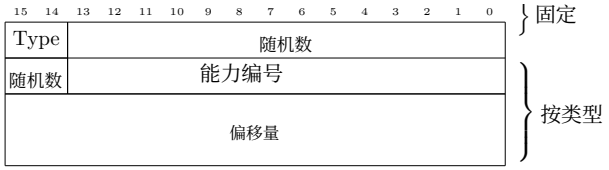

图1：Northcape 令牌格式（显示 32 位偏移量类型）。

第 2.2 节中的能力表示与转换，以及第 2.3 节中的操作。本节介绍在 Bredi 中实现的 Northcape。我们将在第 8.1 节讨论对 Northcape 所做的改进 [1]。最后，附录 A 对 Northcape 进行了深入讨论。

2.1 对象能力系统

对象能力架构最早由 Dennis 和 Van Horn [30] 于 1966 年提出，它将计算机的物理内存划分为段，每个段具有按字节粒度指定的起始地址和结束地址。主体（子系统）必须持有针对其访问的每个对象（段）的明确访问权限（能力）。能力的不可伪造性确保了安全性。

能力架构不需要特权操作模式。相反，显式的子系统调用指令会在彼此隔离的子系统之间传递控制权。

最后，一个子系统向另一个子系统委派能力，可以受控地共享特定数据结构，从而消除环境权限 [55]。因此，能力委派和子系统调用有助于彼此隔离的子系统协作，使不可信内核 [35]成为可能。

2.2 Northcape 能力架构

表示形式。文献中出现了两种不同的对象能力表示形式：1）像 CHERI [80] 这样的胖指针系统会区分能力和内存地址。能力必须加载到显式的能力寄存器中，并在 CPU 指令中作为源地址或目标地址引用。CHERI 能力本身包含段边界和访问权限。能力的不可伪造性依赖于对存放有效能力的内存位置进行标记，这需要单独的标签位图。2）像 RV-CURE [43] 这样的令牌能力系统会将能力元数据与表示形式分开。RV-CURE 将能力标识符嵌入指针的高位。该标识符通过中央元数据表（CMT）解析为段边界。能力的不可伪造性依赖于 *MAC* 标签。

与 RV-CURE 类似，Northcape 是一种基于令牌的能力系统。如图1所示，Northcape 将 64 位指针分为低位的偏移量 和高位的随机数 及能力编号 。该编号用于标识一个 CMT 条目，其中存储着元数据。

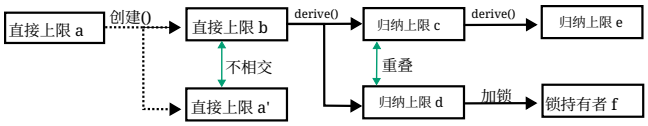

图 2：Northcape 系统中的能力类型。

Northcape 通过引入可变的偏移量类型，使偏移量和编号的大小可变，从而在支持的能力最大大小与数量之间实现权衡。我们支持 8 至 32 位的偏移量，按 8 位递增，因此同时可实现使用 32 位偏移量的 2^{14} 个能力，以及使用 8 位偏移量的 2^{38} 个能力。我们的原型已考虑以最小工作量调整这些值。

Northcape 能力令牌分为偏移量、数值和随机数，因此可以与 C 指针互换使用。关键在于，可以将能力令牌解释为内存中对象的起始位置。随后，通过递增偏移量即可访问对象的各个字段。偏移量的调整会自动进行，从而实现与旧式指针运算的向后兼容性。相比之下，RV-CURE 和 CHERI 要求使用显式指令来访问能力 [43, 80]。

能力类型。以下将以文本形式介绍 Northcape 的能力类型及其相互关系，如图 2 所示。

Northcape 能够确保子对象保护，这是现代对象能力系统的一项常见特性 [43, 80, 83]。因此，它区分直接能力和间接能力：直接能力拥有其解析到的段。Northcape 的操作可确保所有现存的直接能力彼此不相交。

间接能力授予对直接能力子段的访问权限。同一直接能力的间接能力可能相互重叠，并且可以递归创建：可以从间接能力创建间接能力，从而缩小段边界（单调性 [80]）。Northcape 会跟踪每个间接能力的父级，以保障时间安全。

通过锁定 操作，子系统可以获得对能力的独占访问 权限。为此，锁定 会创建一个锁持有者能力。与间接能力一样，锁持有者能力可以允许访问父级直接能力或间接能力的子集。不过，只要锁持有者存在，就无法访问所有重叠能力。

能力不可伪造性。成功伪造需要猜中 一个随机数：其概率为 $1/2_{16}$，因为 Northcape 使用 16 位随机数。RV-CURE 采用相同的设计
并依赖于成功猜中的可能性很低 [43]。

我们在此提供了一项关键改进：能力可以选择性地携带子系统 *ID* 绑定限制，仅允许某个特定子系统访问该能力。创建能力时必须指明这一点；对于现有的、未采用此保护的能力，也可以使用限制来添加该保护。

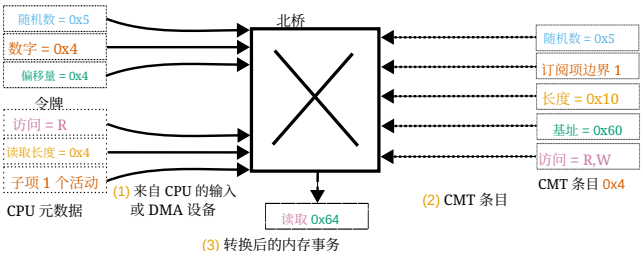

图 3：Northcape 能力令牌的转换。

访问带有子系统 ID 绑定限制的能力时，Northcape 会确保能力中的 32 位子系统 ID 与活动子系统相匹配。为此，Northcape 会在子系统调用期间跟踪活动子系统的 ID。因此，当其他子系统猜中随机数时，由于其子系统 ID 不匹配，仍无法访问该能力。重要的是，该 ID 不会构成环境权限，因为所有访问仍只能通过能力进行 [55]。

能力转换。 图 3 展示了如何将能力解析为物理地址：CPU 或直接内存访问控制器上的软件通过系统总线发起内存事务 (1)。启用了 Northcape 的北桥将把 提供的地址解释为能力令牌，并解析其中的编号、偏移量和随机数部分。编号字段唯一标识一个 CMT 条目。北桥将取回该条目 (2)。首先，它会通过验证随机数及任何子系统 ID 限制来确保 不可伪造性 。然后，它会确定请求的 偏移量 和长度 是否仍在能力的边界内，从而确保空间安全。如果解析出的能力是间接能力或锁持有者，北桥将递归 处理，直至到达直接能力，并确保所有父级均有效且未锁定。这对于时间安全至关重要，特别是对于 撤销 和 独占访问。最后，北桥将计算物理地址，即 CMT 条目中存储的基址加上令牌的偏移量 (3)。

2.3 Northcape 能力操作

我们将通过三个示例介绍 Northcape 的操作：内存共享、动态内存分配和控制转移。操作的完整列表及详细说明见表 1。

安全内存共享。 考虑直接内存访问传输：驱动程序向设备指明位置 地址 和长度 l。假设设备会在发出完成信号前，向 地址 传输或从中传输最多 l 字节。其他子系统可以调用 DMA 驱动程序中的子系统接口来启动事务。调用时可以将能力令牌作为参数传入。发起调用的子系统可以将其有权访问的任意能力的任意子集委托给 DMA 驱动程序。这可以通过派生 或 克隆 操作完成，这些操作可从某项能力创建间接能力。其中，派生 允许缩减能力

表 1：Northcape 操作详细信息。

语法	简短描述	完整描述
创建$(c_a,l,r,p) \longrightarrow c'_a,c_b$	拆分 a 直接能力分成两个互不重叠的直接能力。	将能力c_a 拆分为两个互不重叠的部分。创建一个新的能力c_b 并相应地修改c_a 的元数据。调用方还可以指定长度l 以及限制r 以及该段的权限p 。p 和r 必须不宽于该权限以及c_a的限制，且l 必须小于或等于c_a的长度。如果$l = l_a$，则使c_a 失效。仅可用于直接能力，要求c_a 未被锁定且引用计数为零。
合并$(c_a,c_b,r,p) \longrightarrow c_m$	合并两个相邻的直接能力到一个直接能力。	此操作与创建相反。它会从物理相邻的能力创建一个新能力 c_m c_a,c_b ，并赋予新的限制 r 和新的权限 p。权限 p 以及新的限制r 可以比输入能力的限制更加宽松。输入能力会被销毁。要求这些能力的引用计数为零且未被锁定。
派生$(c_a,l,o,r,p) \longrightarrow c_i$	创建一个间接能-力，来自直接或间接能力。	派生 操作用于从输入能力c_a创建间接能力，而这些能力仅由除内存分配器之外的所有子系统和设备使用。一个间接能力具有一个作为直接能力的（祖）父级，以及可能与之重叠的兄弟能力。为此，c_a 可以是直接能力，也可以是另一种间接能力。输出能力 c_i 是始终为间接能力，并且与其父级相比，可以具有额外的偏移量 o 和缩短的长度 l 。父级。
克隆$(c_a,r,p) \longrightarrow c_i$	derive 的变体*derive* ，其保留的 能力边界。	derive 的同义词*derive* ，其内部调用 *derive* ，偏移量 o 为 0，长度 l 等于段的长度。适用于从某项能力中移除权限或添加限制，或对一项能力添加限制子系统想要共享的能力，尤其是在该段长度克隆的调用方不知道该能力。
锁定$(c_a,r,p) \longrightarrow c_l$	获得独占访问权限能力并返还锁持有者能力。	尝试获取对某项能力的独占访问。输入能力c_a 可以是直接能力或间接能力。如果为c_a的祖父直接能力设置了可锁定权限，则会创建一个锁定持有者能力c_l 。可锁定权限可防止滥用锁定来发起拒绝服务攻击。只要它仍然有效，c_l 就可用于访问通过c_a可见的同一段（或其中相同的部分）。与此同时，与c_a 重叠的其他能力无法使用。
丢弃$(c_a) \longrightarrow b$	销毁 间接或锁持有者能力如果它没有 生存子项。	如果c_a 没有任何引用，则将其销毁，并返回操作是否成功。只能用于间接能力和锁持有者能力。如果c_a 是由*lock*返回的锁持有者能力，则父级直接能力也会作为副作用被解锁。因此，调用方*drop*无需知道令牌解析为何种类型的能力。
撤销$(c_a,r,p) \longrightarrow c'_a$	销毁 能力，创建新能力并覆盖段用零。	销毁与c_a 关联的 CMT 条目，并创建新的直接能力c'_a ，用于该能力所标识的段，赋予完整权限并移除限制。覆盖长度为 0 字节的段。返回新的能力。只能用于直接能力。权限 p 和新的限制r 可以比输入能力的那些更为宽松。
检查$(c_a) \longrightarrow b,l,r,p$	确定能力元数据	读取任意能力 c_a 的元数据。递归解析 c_a 以确定基址和长度如有必要（对于锁持有者能力而言，它们不直接编码这些信息），并且可锁定权限。该操作会根据是否存在子系统 ID 限制而略有不同，适用于c_a：如果没有此类限制，或者该限制与操作的调用方匹配，该操作将返回所有上述元数据。如果c_a 设有设置子系统 ID 限制，则该操作仅返回R，W，X，I 权限和限制。在其他情况下，c_a 将返回错误代码。
限制（c_a,r,p,o^+,l^-）$\longrightarrow b$	Make 能力边界和 权限更严格。	就地限制能力。此操作可应用于所有类型的能力语义略有不同：对于间接能力，可以指定额外的偏移量 o^+ 和长度减数 l^- 。间接能力的边界会相应缩小。对于其他能力，这些参数将被忽略。此外，所有未在*both* p 和 c_a 的元数据中指定的权限都会被丢弃。仅当c_a 是直接能力时，才会应用 r；否则，尝试设置或清除它都不会产生任何效果。r 仅在 c_a 当前没有限制时才会应用。
调用（c_a）	执行子系统调用。	此操作描述执行子系统调用的过程。输入一个能力c_a ，并设置子系统 ID 限制后，将非架构的 子系统标识符寄存器设置为所指示的身份，并将控制权以原子方式传输给已识别段中的代码。

而克隆 会保留边界。此外，还可以缩小边界，并使用限制移除访问权限。检查 操作允许 DMA 驱动程序确定能力的元数据，以进行合理性检查。

DMA 驱动程序可以将收到的能力（或其子集）委派给 DMA 设备。Northcape 将确保空间安全，即以字节粒度将 DMA 设备限制在该能力的边界内。传输完成后，驱动程序可以使用丢弃销毁已委派的能力。只有不存在存活子能力的能力才能成功执行此操作；该操作会使能力失效，从而确保时间安全。

在某些情况下，例如 Xilinx DMA [2], 使用的 DMA 描述符，DMA 设备及其驱动程序会轮流读取和写入同一对象。此时，DMA 驱动程序可以使用锁定来获得对描述符的独占访问，防止设备发起检查时刻到使用时刻攻击（TOCTOU 攻击）。

内存管理。如前所述，内存由直接能力独占拥有。这对于 slab 分配器很有用。可以向分配器提供一个大型直接能力作为初始堆。当子系统通过子系统调用请求内存时，分配器可以使用创建，从堆中切分出直接能力。这样，已分配的能力可以被锁定，同时分配器仍能访问堆。

释放的内存可以通过合并操作归还给堆，该操作将两个相邻的能力合并为一个，并恢复访问权限。结合使用创建和合并，足以实现 slab 分配器 [84]。

最后，与创建不同，撤销操作可用于已锁定或具有子项的直接能力。它会覆盖关联段中的内容并将其置零，确保不会泄露任何机密信息，同时创建一个不含子项的新输出能力。撤销操作很有用，例如，不可信分配器可以在子系统崩溃后回收其内存。

相互隔离的控制转移。最后，North- cape 提供子系统调用操作调用，它允许原子地同时更改 CPU 的程序计数器和活动子系统 ID，从而改变保护域。调用只能针对具有特殊设置子系统 *ID*限制的能力的第一个字节调用：因此，调用允许转移到显式入口点，同时确保子系统 ID 不会被其他子系统劫持。

设置子系统 ID 限制必须在创建能力时指定。至关重要的是，子系统只能为自己的子系统 ID 创建设置子系统 ID 限制。否则，劫持其他子系统 ID 将轻而易举。对此规则，我们有一个例外：特殊的子系统 ID 0 可以创建任意设置子系统 ID 限制。从概念上讲，子系统 ID 0 专供可信的 Skadi 加载器使用；该加载器需要此特权来引导其余子系统。加载器会在加载时结束时使子系统 ID 0 无法访问，从而防止滥用。

3 安全模型

Lefeuvre 等人 [45]为隔离机制提出了以下安全模型：**C**（机密性）：主体不能读取其保护域之外的内容，**I**（完整性）：主体不能写入其保护域之外的内容，**A**（可用性）：主体不能阻止其他保护域正常执行。我们将这一安全模型应用于由 Skadi 和 Bredi 组成的组合系统。在该系统中，主体指的是子系统。子系统的保护域（用能力术语来说：保护域 [30]）包含明确委派给它的所有能力。这包括每个子系统各自的代码段和数据段，以及作为子系统调用参数接收的任何能力。对于**C** 和**I**，我们保证空间和时间安全性，而无需显式执行内存扫描 [81]。最后，我们扩展**A**的定义，将软实时执行纳入其中。

我们进一步要求，对需要保护属性**C** 和**I** 的能力进行独占访问：默认情况下，子系统可以假定自己独占访问私有代码和数据段，但明确委托给其他子系统的子集除外。子系统可以使用锁定 操作独占访问共享能力，此后该机制必须保证**C** 和**I**。此外，子系统可以使用丢弃操作取消共享某项能力，从而重新获得该能力的 **C** 和 **I**。

将硬件可信计算基（可信计算基）定义为 CPU、能力强制北桥以及包括总线在内的系统内存。软件可信计算基仅由*Skadi* 加载器构成。启动序列结束时，加载器会被销毁，因此不存在运行时软件可信计算基。可信计算基之外的所有软件和硬件组件均假定为公开恶意。侧信道攻击和物理攻击属于独立问题，因此不在讨论范围内。

我们进一步将可用性可信计算基（*aTCB*）定义为可信计算基与所有参与处理中断的软件过程的组合。请注意，不属于可信计算基的 aTCB 组件仅在**A**方面可信——它们可以攻击安全目标**C** 和**I**，例如操纵寄存器。

该**A** 属性与我们的**I** 属性相冲突：如果某个软件子系统禁用中断并进入无限循环，就可能导致其他子系统再也无法执行。因此，我们需要能够抢占 子系统，让其他关键子系统运行。然而，要抢占 这样的对手，就必须能够阻止它禁用中断。为了在不退回到使用 CPU 强制实施的特权环的情况下解决这一冲突，我们放宽了**A** 目标：只有在对手不禁用中断，或者可用性关键子系统是一个中断处理程序时，才要求满足该属性。我们明确允许可用性关键子系统不是第一级中断处理程序，但依赖上文定义的 aTCB 来保证中断处理程序链上的可用性。

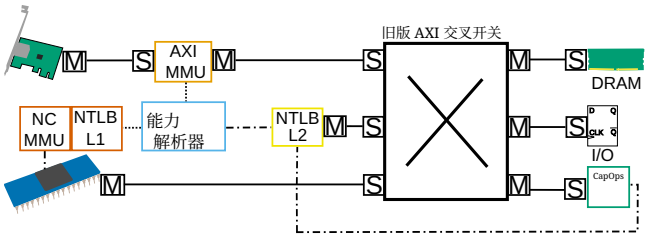

图 4：Bredi 架构。实线表示 AXI，点线表示 AXI-Stream，点划线表示定制总线。

4 BrediSoC 实现

本节介绍 Northcape 的硬件实现 Bredi。我们将简要介绍 Bredi 的核心组件，然后讨论 Bredi 的三个核心特性，这些特性对于实现我们的安全目标至关重要。首先，第 4.1 节讨论缓存层级如何隐藏递归能力解析的开销，这对我们的可用性目标（**A**）至关重要。其次，第 4.2 节讨论为将中断服务例程排除在可信计算基之外而对中断处理所做的必要改动，这对于实现机密性（**C**）、完整性（**I**）和可用性（**A**）是必不可少的。最后，第 4.3 节讨论实现**A**所需的定制 CPU 指令。所有组件的详细说明见附录 B。

Bredi 基于 cva6 [85]，这是一款支持 Linux 的顺序提交应用程序处理器。我们将 RV64G 指令集（64 位字长、完整寄存器组、完整浮点单元支持）配置为带有 PMP 和 sv39 MMU [62]。我们集成了基于专有直接内存访问的 Xilinx AXI 以太网子系统 [2, 4]，用于评估直接内存访问的兼容性与性能（**A**）。我们还使用了 Xilinx 专有的 AXI 交叉开关 [3]。此外，我们实现了用于 Northcape 支持的新型组件，如图 4 所示：面向 cva6 的带 TLB（NTLB L1）的 Northcape 内存管理单元（NC 内存管理单元）、AXI 内存管理单元、能力解析器、能力操作（CapOps）以及二级 Northcape TLB（NTLB L2）。

AXI 内存管理单元。 AXI 内存管理单元对 DMA 设备强制实施内存访问限制，类似于输入输出内存管理单元或 CHERI 能力检查器 [24]。它通过前端总线（AXI 4 [10]）与 DMA 设备对接，执行访问控制，并将能力令牌转换为物理地址（如图 3 所示），然后将总线事务转发至 AXI 互连。请记住，该转换对设备而言是不透明的。

AXI 内存管理单元没有缓存，而是依赖能力解析器和 NTLB L2。这会导致性能略有下降，但能显著减小芯片面积。

cva6 内存管理单元。 Bredi 的 cva6 包含一个指令 Northcape 内存管理单元和一个数据 Northcape 内存管理单元，它们在 cva6 流水线中执行与 AXI 内存管理单元相同的任务。cva6 MMU 包含一个小型全相联 TLB（NTLB L1），可将能力令牌映射到边界、权限和限制，并通过能力解析器处理未命中。

能力解析器。 所有 cva6 和 AXI 内存管理单元都通过轮询仲裁器与单个 Northcape 能力解析器接口。这样一来，无论系统中存在多少个内存管理单元，能力解析和解析逻辑都只需实现一次，从而减少芯片面积。该解析器采用流水线设计，每个周期可接收一个解析请求并发送一个解析响应。

能力操作模块。 Northcape 系统的倒数第二个组件是操作模块，它通过硬件有限状态机实现了我们在第 2.3 节介绍的 Northcape 操作。为此，它是系统中唯一能够通过 NTLB L2 写入 CMT 的设备。设备和软件可通过内存映射 I/O 访问该模块。当某个子系统开始设置操作时，内存映射 I/O 接口会锁定到该子系统 ID，从而确保并发设备无法干扰交互过程。

NTLB L2。 最后，操作模块和解析器共用组相联写直达双端口 NTLB L2。与 NTLB L1 不同，NTLB L2 将能力令牌映射到完整的 *CMT* 条目。因此，能力解析器和操作单元都可以操作 NTLB L2 条目，无需访问内存中的 CMT，从而显著减少内存访问。双端口访问免去了缓存仲裁的需要，有利于可用性（**A**）。

4.1 贡献 1：缓存层次结构（A）

如第 2.2 节所述，Northcape 能力可以是直接的，也可以是间接的。CMT 条目提供了转换能力令牌以及强制执行段边界、权限和限制所需的所有信息。实际上，CMT 条目可以像 TLB 中的页表条目一样被缓存。

因此，cva6 MMU 配备了一个小型的全相联一级 NTLB。NTLB L1 将能力 ID 映射到边界、限制和权限，从而有助于在一个周期内完成地址转换过程。因此，缓存命中不会阻塞 *cva6* 的流水线，这与 sv39 MMU 中使用的 TLB 类似 [62]。我们在第 7.4 节中的评估表明，NTLB L1 的高命中率所提供的性能与不启用内存保护的基准情形没有区别。

然而，在完成诸如锁定或丢弃之类的 Northcape 操作后，NTLB L1 条目可能会过期。过期条目可能被用于读取或写入子系统保护域之外的数据，从而危及**C** 和**I**。我们的实现会在每次操作结束时销毁过期的 NTLB L1 条目，以确保**C** 和**I**。为此，操作模块会向各个 L1 NTLB 广播所有已修改能力的 ID。如果此能力属于缓存内容，就将其移除，并通过解析器触发新的递归解析。对于撤销或锁定操作，会使整个 NTLB L1 失效，以确保其中不包含目标的任何重叠能力。

NTLB L2 也可能包含过期条目。不过，NTLB L2 缓存采用直写策略。所有操作

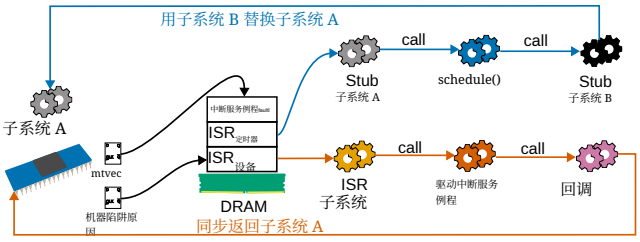

图 5：Bredi cva6 中的向量中断。

仅影响输入能力及其直接父级的操作（派生、丢弃、克隆、创建、合并、检查、限制）不会产生过期条目，因此这些操作后无需使缓存失效。另一方面，锁定和撤销可能影响输入能力的任何重叠能力。因此，这些操作可能留下过期条目，需要将其移除以确保**C**和**I**。

针对 NTLB L2 中过期条目问题，一种简单的解决方案是，在锁定和撤销时刷新缓存。我们提供了以下**A**方面受益的解决方案：在锁定或撤销后，将所有缓存条目标记为可能过期。每次 NTLB L2 缓存命中过期条目后，我们都会强制能力解析器对该能力执行完整的分层解析。如果该条目最终有效，我们就移除其过期标记。否则，我们会将该过期条目从缓存中移除。

4.2 贡献 2：不可信中断服务例程（C，I，A）

处理中断对 Bredi 来说是一项特殊挑战：一方面，中断服务例程位于可信计算基之外。因此，我们需要防止中断服务例程访问被中断子系统的寄存器内容，或破坏子系统 ID 保护（这对**C**、**I**至关重要）。然而，中断服务例程对**A**至关重要：子系统不应能够阻止中断服务例程执行。最后，为实现实时运行（**A**），抢占式调度是理想选择。我们对 cva6 实施以下扩展以满足这些要求：寄存器堆叠、向量中断、主动抢占、不可屏蔽中断。

寄存器堆叠（C、I）。我们向 cva6 添加第二份寄存器组副本。对于每条发出的指令，我们跟踪其是否属于中断状态：中断状态从 ISR 的第一条指令开始，到中断返回*mret* 指令结束。中断状态中的指令使用一份寄存器组，而非中断状态中的指令使用另一份。这样，寄存器就不会从被中断子系统泄漏到 ISR。

除了通用寄存器和浮点寄存器外，两种模式下的控制状态寄存器也各自独立。因此，中断服务例程无法修改运行中子系统的 CPU 状态。这也防止中断服务例程更新 *mepc*（中断地址），确保中断服务例程始终返回到被中断的指令。因此，中断服务例程无法以被中断子系统的标识符劫持 执行。同时，我们还确保在同一种模式下，控制状态寄存器的行为与 RISC-V 规范 [58] 完全一致。实际上，这项指令集架构变更只影响中断处理程序中的少数几行汇编代码。

操作模块中的内存映射 I/O 寄存器也针对两种状态分别复制。因此，ISR 可以像往常一样执行能力操作，而不会干扰被中断子系统与该模块之间的交互。

中断向量化（C、I）。我们实现了一种与所提出的 FastIRQ 扩展相匹配的中断向量化方案 [63],见图 5。CPU 使用向量表，根据中断原因将每个中断原因映射到一个 Northcape 能力令牌。Skadi 使用中断和异常子系统的 set-subsystem-id 令牌填充该表。因此，cva6 在处理中断时会对处理程序执行子系统调用，从而确保被中断子系统与 ISR 之间的子系统 ID 隔离。

至关重要的是，cva6 MMU 还会分别跟踪中断状态和非中断状态下的子系统 ID，从而确保中断期间非中断 ID 不可变。

主动抢占（A）。定时器中断无需进行寄存器堆叠：触发定时器中断不会使 cva6 切换到 IRQ 模式。因此，与设备中断服务例程不同，我们仍使用非中断服务例程寄存器组，从而能够重新调度。Skadi 使用这一机制进行抢占：每个子系统都会收到一个只写能力令牌，该令牌解析后指向定时器中断的向量表入口。每次子系统调用时，各子系统都会在其代码段中为定时器中断注册定时器存根 的地址。这样，触发定时器中断时，当前子系统的定时器存根就会执行。该定时器存根可以保存并清除寄存器，然后向调度器发起子系统调用；调度器可以据此重新调度，并返回到任意子系统。子系统可以禁用定时器中断，因此此机制需主动启用。

不可屏蔽中断（A）。最后，可以使用一次写入寄存器将中断原因设置为不可屏蔽。不可屏蔽中断（NMIs）总会被响应，除非 CPU 已在执行 NMI。因此，可以确保关键中断服务例程及时执行。

4.3 贡献3：自定义指令（A）

在 Northcape 中，能力访问和子系统调用是隐式操作。换言之，能力的访问方式与虚拟地址类似，使用普通的加载、存储和控制流指令。因此，与采用显式能力指令的能力系统（如 CHERI [80] 和 RV-CURE [43], ）不同，Northcape 无需为**C** 和**I**提供自定义指令。不过，我们提供了三种定制方案来提升系统的性能，这对**A**至关重要。

Bredi 提供了一条用于子系统调用的自定义指令调用，它是*jalr* 的一种变体，用于检查目标地址是否为有效的子系统调用目标，否则触发异常，从而省去一次检查 操作。

第二条自定义指令根据位掩码将寄存器清零，在一次双向子系统调用中可节省 18 个周期。

最后，我们提供自定义控制状态寄存器，用于访问操作模块，省去了内存映射 I/O 接口所需的内存屏障，每次操作可节省数十个周期。

5 Skadi RTOS 实现

我们的 Skadi 操作系统利用 Bredi，最大限度地实现操作系统的隔离。我们先概述设计，再详细介绍三项核心贡献：第 5.1 节讨论如何实现无运行时软件可信计算基（这对机密性**C**和完整性**I**至关重要）；第 5.2 节讨论如何实现无特权内核（**C**、**I**）；第 5.3 节讨论 Skadi 子系统调用的性能优化（这对可用性**A**至关重要）。附录 C 对 Skadi 的实现进行了更详细的讨论。

Skadi 基于 Zephyr [86] 实时操作系统。Zephyr 具有高度的模块化和可配置性：它提供一个最小化的操作系统内核，实现调度和初始化。此外，还可选择构建大量组件，包括网络协议栈和设备驱动程序。Zephyr 针对内存有限、处理能力较弱的嵌入式设备进行了优化。因此，Zephyr 遵循单内核范式，即内核、可选组件和应用程序链接为单个二进制文件，并在单一地址空间中执行。在具有内存隔离硬件支持的开发板上，例如配备内存保护单元（MPU）的开发板，Zephyr 可以将应用程序逻辑与内核及子系统隔离开来。然而，内核或子系统一旦遭到入侵，整个系统也必然会受到影响 [68]。

Skadi 利用 Zephyr 的高度模块化设计，通过使用子系统调用，在现有组件之间引入相互运行时隔离，形成子系统。我们进一步将内核拆分为初始化逻辑（大部分成为 Skadi 加载器的一部分）、调度器（成为非特权子系统）以及中断处理（也成为一个子系统）。因此，Skadi 没有特权内核。图 6 展示了 Skadi 子系统及其依赖关系的示例。

5.1 贡献 1：无运行时可信计算基（C，I）

Skadi 子系统分别编译为可重定位的 ELF 二进制文件。因此，在引导过程中，子系统需要重定位 到各自受保护的地址空间中。

因此，我们提供了一种最小化的特权固件：*Skadi* 加载器。加载器在概念上属于*BrediSoC*，并假定在不可变只读存储器中受到物理保护。

系统重置后，Skadi 加载器从子系统 ID 0 开始执行。此时，只有 Northcape 的根能力存在，它涵盖整个物理地址空间，且

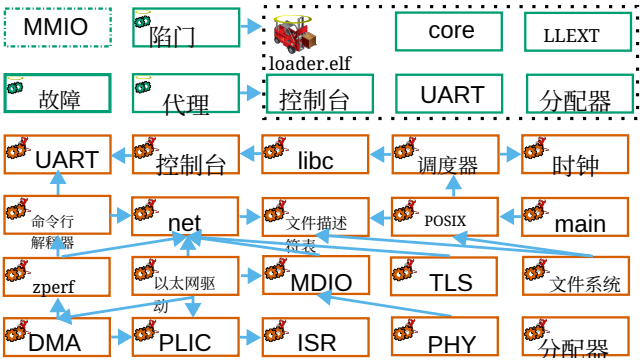

图 6：Skadi 中的分区子系统以及加载器架构（虚线框）。箭头表示子系统调用。前两行的子系统和加载器均受信任：故障子系统仅在发生异常后负责停止系统，而陷门和代理 在加载时结束后将无法访问。内存映射 I/O 能力由加载器创建，并委派给特定子系统。

其编码方式使类型、标识符和随机数均为零。实际上，加载器最初可以将其与普通的 32 位物理地址互换使用。加载器为根能力添加子系统 ID 限制 0，使其无法被子系统和 DMA 设备访问。

随后，加载器开始重定位 子系统：首先按照第 2.3 节所述，使用创建 操作为代码、数据和 BSS 段分配能力。接着，加载器将经 LZ4 [28] 压缩的 ELF 段解压到这些能力中。之后执行标准的 ELF 运行时链接和重定位流程：我们使用 RISC-V 的大型代码模型 [25], 编译子系统，其中代码和数据符号通过与各个函数同处一处的重定位表进行引用。重定位表条目宽 64 位，可直接容纳 Northcape 令牌。因此，子系统可以将数据和子系统调用作为简单的 ELF 符号导入和导出，加载器则可通过标准链接器符号表解析这些符号。这使我们能够使用标准编译器和链接器，无需修改工具链。相比之下，RV-CURE 和 CHERI 需要自定义编译器支持 [43, 80]。

子系统还可以通过导入 ELF 符号来导入*MMIO* 映射，该符号编码了内存映射 I/O 区域的物理地址和长度，加载器会据此创建相应的能力。为了与 Zephyr 的构建时设备树解析保持兼容，我们将导入的内存映射 I/O 区域声明为函数，并使用其地址。

子系统可以注册初始化函数以执行设置工作。为此，加载器会为每个子系统分配一个来自 Northcape 32 位子系统 ID 空间的单调子系统 ID。随后，它会为该子系统 ID 创建一个设置子系统 ID 能力，该能力指向一个简单的初始化跳板。它还会对持有该子系统代码和数据的能力施加子系统 *ID* 绑定限制，确保——

这些能力无法被其他子系统使用。加载器可以通过子系统调用 进入初始化跳板，从而确保与该子系统相互隔离。随后，跳板会跳转到该子系统的代码段，并从分配给它的子系统 ID 开始执行，但无法危害加载器或任何其他子系统。

所有子系统一旦加载并初始化完毕，加载器便会通过转换为陷门子系统来销毁自身。陷门 子系统会对根能力剩余部分执行撤销 操作。这会使所有子系统 ID 为 0 的设置子系统 ID 能力失效，从而无法重新在该子系统 ID 下获得执行权限。因此，加载器可访问的所有机密信息（例如指向子系统代码和数据的直接能力）都会变得不可访问。随后，陷门 会在没有有效返回地址的情况下调用调度器 子系统。从此以后，只剩下 ID 非零的非特权 子系统。因此，Skadi 成功实现了无运行时软件可信计算基。

5.2 贡献 2：NoOS 内核（C,I）

Skadi 没有内核。相反，所有内核功能都通过分区隔离的子系统实现，这些子系统通过子系统调用协同工作。关键在于，子系统调用无需像 CHERIoT RTOS [9] 中的*Switcher* 那样依赖可信中介。

一个调度器子系统维护线程控制块并实现调度策略。取消操作点（例如取消操作点，如*k_*先进先出队列_获取）也在此处实现。在取消操作点处等待是通过子系统调用完成的。

第 4.2 节中介绍的主动抢占机制使我们能够实现抢占式调度。请记住，每次子系统调用都会更新定时器中断的处理程序。因此，每个子系统都可以在自己的代码段中提供一个定时器存根，并确保在启用定时器中断时将其注册。随后，该定时器存根可以通过让出端点执行子系统调用，由让出端点决定系统是否应重新调度。如果活动线程发生变化，调度器可以在其自己的栈上执行一次常规的上下文切换，然后直接返回到另一个子系统。否则，它将返回到定时器存根。

设备中断和异常通过独立的子系统进行处理。加载器会为每个中断原因注册正确的子系统。故障子系统负责在遇到异常时安全地停止系统。它

没有其他函数，并且仅针对异常进行注册。中断服务例程 子系统包含特定于 CPU 的一级中断服务例程。在 Bredi 上，它将通过子系统调用把中断直接转交给二级中断控制器（PLIC），后者将调用中断设备驱动程序的中断服务例程。

典型的库功能既可以内联实现，也可以作为子系统提供：*libc* 子系统实现标准 C 库函数，例如*printf*。基本字符串函数（如*memset*）也可以选择内联到各个子系统中，以提升性能。我们还实现了一个不可信分配器

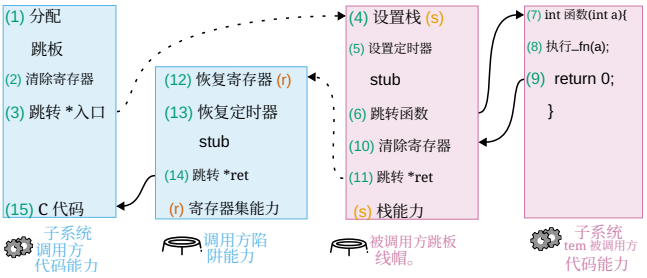

图 7：完整的子系统调用。实线表示控制流，而虚线表示子系统调用。

遵循第 2.3 节中所介绍策略的子系统。

最后，所有驱动程序都是子系统。驱动程序可以导出类似于 Zephyr 设备模型的设备结构体 [87]。加载器会派生相应的能力，并使其可供其他子系统使用；其他子系统可以通过子系统调用这些能力来调用设备 API。驱动程序可以向 PLIC 子系统注册，以处理中断请求。最后，DMA 驱动程序可以委派能力给设备，策略类似于第 2.3 节所述。关键在于，DMA 设备只能使用明确委派给它们的能力。因此，Skadi 为 DMA 设备提供了字节粒度的时空安全。

5.3 贡献 3：调用约定(A)

我们的子系统调用约定详见图 7：调用方分配一个返回跳板 (1)，将所有寄存器清零以防止信息泄露 (2) 并跳转到 被调用方跳板 (3)。被调用方跳板分配一个栈(4) 并设置定时器存根 (5)。此后，调用 C 函数(6-9)。最后，被调用方跳板将其寄存器清零 (10) 并返回 (11)。随后，调用方跳板恢复寄存器组和定时器存根(12-13) 然后返回到 C 代码 (14

15)。至关重要的是，跳板的代码和数据属于每个子系统的代码段和数据段，因此它们不需要可信中介。所使用的设置子系统 ID 的能力由 Skadi 加载器在重定位期间创建

我们的子系统调用约定的一个核心细节是分配每个子系统的寄存器组和栈。每个 Skadi 子系统会将构建时可配置数量的栈帧和寄存器组分配到一个池中。分配使用原子位图或单处理器逐中断模式位图，在确保对称多处理和中断安全性的同时提供大小与性能的权衡。分配使用 $\mathcal{O}(1)$ 尾随零计数 （计算尾随零）指令。如果有可用的栈帧，我们可以在单个周期内分配一个。第 7.3 节表明，这有助于实现快速的子系统调用。如果所有栈帧均被占用，我们可以自旋或触发异常。

先前提出的子系统调用约定则使用每线程栈，并由调用方与被调用方共享该栈。与 Skadi 相比，这虽然减少了内存占用，但必须在返回时清空栈 [9, 69] 或依赖

于 未初始化能力 [38, 83],从而产生运行时开销。

跳板通过类函数宏生成，这些宏接受调用名称和签名。因此，修改现有 Zephyr 组件只需为子系统调用添加包装宏。使用 _Generic，我们区分 void 跳板，并据此清除返回寄存器。

我们对调用方跳板进行去重并精简，从而减小代码大小：调用方跳板宏会生成一个外部函数调用，其名称编码了调用名称和签名。在构建时，我们为每个子系统生成一个源文件，根据符合该模式的未解析函数名称来实现调用方跳板。

6 安全讨论

Skadi 和 Bredi 能够应对我们在本文开篇所指出的攻击途径——应用软件、操作系统内核和外设。解决这一挑战的关键思路，是结合能力文献中的不可信操作系统方法 [8, 9, 35]，并在系统总线层面以向后兼容的方式实施能力机制。由此，我们能够防范不可信的直接内存访问设备 [1, 51]。表 2 列出了 Skadi 和 Bredi 提供的详细安全保证及其关键实现机制。简言之，Skadi 组件的强隔离为子系统的整个保护域提供机密性（**C**）和完整性（**I**）。作为唯一受信任的软件组件，加载器在加载时之后即被销毁，因此 Skadi 实现了无需运行时软件可信计算基。例如，CheriTrEE [72]也使用了类似的可信加载器来实现相同目的。由于 Skadi 的隔离对于在不损害可用性（**A**）的前提下实现**C** 和**I** 至关重要，因此在本节剩余部分，我们将论证隔离为何不会被破坏。

为了保护子系统私有代码和数据的**机密性**与**完整性**，我们将子系统的 ELF 段重定位到绑定子系统 ID 的能力中。通过子系统 *ID* 绑定限制，可以防止针对这些能力的猜测攻击：如果具有不同子系统 ID 的子系统猜中了随机数，只要它未能同时劫持子系统 ID，访问时仍会遇到异常。我们的子系统调用跳板经过设计，可防止在调用和返回时未经控制权传输就更改子系统 ID。不使用子系统调用且无法使用子系统 ID 0，就无法更改子系统 ID，从而防止劫持。由于寄存器堆叠和*IRQ* 模式，中断服务例程无法干扰被中断的子系统。

动态分配的内存受到能力令牌不可伪造性的保护，并可选择通过锁定 或绑定到子系统 ID 的限制来实现独占访问。默认情况下，不对分配的内存添加子系统限制，因此可以与其他子系统或设备共享该内存。分配器不能使用撤销 来破坏**C** 或**I**，因为撤销 会覆盖数据并使重叠的能力失效。

最后，**可用性**依赖于不可屏蔽中断。我们假设中断子系统与软件中断服务例程不会阻塞中断执行，因为它们属于 aTCB。中断服务例程子系统之间的子系统调用经过设计，能够满足实时要求。中断服务例程子系统之外的攻击者无法阻止它们执行，因为中断可以设置为不可屏蔽。为此，IRQ 屏蔽是一次性写入的。

7 性能评估

我们的评估衡量了 Bredi 和 Skadi 带来的性能开销。我们在 Digilent Genesys 2 现场可编程门阵列开发板上实现了原型。L1 和 L2 NTLB 的大小分别为 16（指令）/32（数据）和 512。L1 NTLB 是全相联的，而 L2 NTLB 的相联度为 8。我们的基准测试涵盖芯片面积、代码大小、计算和网络性能，以及实时适用性。我们在 Bredi 上对 Zephyr 和 Skadi 进行了比较。

此外，在可能的情况下，我们还尝试在 Linuxv5.10.7 上评估等效场景。这是一种苹果与橘子的比较，因为 *Zephyr* 和 *Linux* 的许多应用程序接口和系统服务存在显著差异。我们仍决定进行这项比较：与 Linux 的比较表明，Skadi 的性能与一种常用通用操作系统处于同一数量级，这证实了我们关于 Skadi在现实世界中的实用性的主张。

7.1 硬件：芯片面积和频率

我们使用 Xilinx Vivado 2024年.2 [5] ，以 50 MHz 合成 Bredi，与原始 cva6 设计类似 [85]。表 3 报告了 Bredi、Bredi-cva6 和原始 cva6 各组件所需的 FPGA 基元数量。我们的设计满足所有时序要求。

我们的芯片面积和频率结果表明，将 Northcape 集成到现场可编程门阵列或专用集成电路设计中是可行的。二级 NTLB 的芯片面积最大，这是由复杂的冒险规避逻辑和高相联度造成的。

7.2 软件：代码大小

我们在表 4 中详细列出了 Skadi 的三种配置的代码大小：一个简单的 hello-world 示例、一个基本的基于套接字的 HTTP 服务器，以及 zperf（Zephyr 对 iperf 的实现）。

我们报告二进制文件的总大小（磁盘大小），以及文本、数据、只读数据（rodata）和 BSS 段的大小。二进制文件中还有很大一部分是子系统的元数据——重定位段、符号表等，这些元数据会在加载阶段后回收。我们在 *ELF* 开销行中给出这些元数据在*LZ4* 压缩之前的大小总和。最后，对于 Skadi，我们报告初始化后的分配内存总量。Zephyr 仅使用 .bss 段。

表 2：总　　结：硬件和软件提供的具体安全保证及其关键机制　　.

Goal	Bredi	Skadi
空间内存隔离（C，I）	子系统 ID 保护和限制、随机数、单调性操作（第 2.2 节）	私有代码和数据、无运行时软件可信计算基、无内核（第 5.1 节、第 5.2 节）
时间　　内存隔离（机密性、完整性）	锁定 独占访问，跟踪父级与子级关系-及递归解析（第 2.2 节）	无运行时软件可信计算基、无内核、不受信任的调度器-和分配器（第 5.1 节、第 5.2 节）
DMA隔离（机密性、完整性）	AXI 内存管理单元（第 4 节）	不可信DMA驱动（第 2.3 节、第 5.2 节）
ISR隔离（C、I）	寄存器堆叠、中断向量化（第 4.2 节）	由加载器管理的ISR表（第 5.2 节）
抢占（A）	主动抢占硬件支持（第 4.2 节）	跳板中的定时器存根（第 5.3 节）
不可屏蔽中断（A）	不可屏蔽中断硬件支持（第 4.2 节）	中断服务例程子系统（第 5.2 节）
快速操作（A）	缓存层次结构（第 4.1 节）、指令（第 4.3 节）	子系统调用约定（第 5.3 节）

表 3：查找表中的面积　　s（查找表）、触发器（FFs）、多路复用器（MUXs）和块随机存取存储器（块 RAM）在 Bredi 和 cva 中的占用面积　　6.

组件	查找表 (%)	触发器 (%)	F7 多路复用器 (%)	F8 多路复用器 (%)	块 RAM (%)
AXI 内存管理单元	4 085 (2.00)	832 (0.20)	2 (0.01)	0 (0.00)	0 (0.00)
解析器	1 640 (0.80)	506 (0.12)	58 (0.06)	0 (0.00)	0 (0.00)
操作	9 280 (4.55)	4 675 (1.15)	251 (0.25)	0 (0.00)	2 (0.45)
二级 NTLB	20 830 (10.22)	2 230 (0.55)	357 (0.35)	57 (0.11)	70 (15.73)
Bredi cva6	63 245 (31.03)	39 971 (9.81)	4 367 (4.29)	1 580 (3.10)	28 (6.29)
原始 CVA6	46 764 (22.95)	25 817 (6.33)	2 060 (2.02)	426 (0.84)	28 (6.29)

表4：Skadi 和 Zephyr 的代码大小（　　）。

类别	Hello World	HTTP	zperf
磁盘大小	738 890	2 435 932	3 109 444
	49 036	556 952	842 216
Text	405 550	1 465 822	3 792 040
	26 016	174 838	389 116
Data	14 796	56 852	79 852
	9524	54 806	94 526
只读数据	71 376	149 124	237 008
	-	-	-
BSS	103 192	1 935 060	1 918 092
	13 493	327 266	358 519
子系统	12	21	23
被调用方　　跳转指令	219	450	490
调用方　　跳转指令	50	222	235
栈 / 寄存器 Sets	4/4	16/16	12/12
ELF 开销	1 872 687	4 608 420	6 137 930
内存总量	1 774 432	7 258 816	10 289 304

我们的代码大小评估表明，Skadi 的隔离开销相当显著。正如我们在第 5 节所述，Zephyr 是一种单内核操作系统：在默认配置下，所有操作系统组件都位于同一地址空间。另一方面，Skadi 的组件会在运行时进行重定位：因此，所有组件都需要携带额外的元数据，例如重定位项，这解释了磁盘大小的增加。运行时内存用量的增加则可归因于子系统调用跳板、每子系统栈，以及部分库函数的内联。

虽然 Skadi 的大小开销较高，但对于大多数现代应用场景而言，它仍然足够小。尤其是如今，嵌入式系统中的内存容量主要受功耗预算 而非成本限制，同时能效也在不断提升 [18]。因此，未来的嵌入式设备很可能会配备更大的内存，使这一缺点在实际应用中变得无关紧要。

7.3 系统：微基准测试

我们在表 5 中提供了 Skadi 操作的微基准测试。我们记录相应驱动程序函数的总持续时间，以及操作模块所使用的周期数。

对于表 6 中的子系统调用微基准测试，我们测量子系统调用的单向和双向持续时间，并与 Zephyr 系统调用进行比较。为此，我们在基线中启用了可选的 PMP 保护用户空间。

总而言之，除撤销 之外的所有操作均在不到 1,000 个周期内完成，其中包括驱动程序开销（主要是将参数装入寄存器）。对于撤销，瓶颈在于将动态随机存取存储器中的段清零，这是安全性所必需的。然而，撤销

表 5：Northcape 操作的微基准测试。

基准测试	CPU 周期	操作周期
创建	629.27 ± 4.39	11.00 ± 0.00
派生	621.97 ± 6.61	11.00 ± 0.00
drop	255.88 ± 4.50	12.00 ± 0.00
解锁	124.81 ± 1.29	17.00 ± 0.00
合并	407.13 ± 3.20	16.00 ± 0.00
克隆	322.35 ± 5.30	14.00 ± 0.00
撤销 (1 MiB)	39 894.21 ± 59.62	34 934.54 ± 5.90
lock	402.68 ± 5.76	16.00 ± 0.00
检查	284.28 ± 2.86	8.00 ± 0.00
限制	382.45 ± 3.96	10.00 ± 0.65

表 6：Northcape 子系统调用的微基准测试，与通过 PMP 实现的传统 Ze 比较。

基准测试	CPU 周期	指令
系统调用（单向）	364.16 ± 2.76	140 ± 0
系统调用（双向）	485.26 ± 2.75	196 ± 0
子项调用（单向）	250.77 ± 12.26	113 ± 0.00
子项调用（双向）	402.56 ± 13.43	180 ± 0.00

仅在极少数情况下使用，例如在若干启动步骤中，因此该开销在实践中无关紧要。

单向和双向子系统调用分别耗费约 251 和 403 个周期，优于传统系统调用的 364 和 485 个周期。这是因为 Skadi 退役的指令数更少，这得益于我们的自定义指令（第 4.3 节）。请注意，子系统调用对于相互隔离至关重要，而系统调用只能将特权内核与用户空间隔离，反之则不然。

尽管如此，子系统调用并非没有代价：调用方跳板需要保存并恢复全部被调用者保存寄存器集合，并执行寄存器组设置与栈分配。函数调用可以依赖现有的栈指针，只需保存和恢复其修改的被调用者保存寄存器，因此可以快得多。事实上，仅对寄存器进行操作的 RISC-V 函数，其开销可以低至两条指令（一次调用和一次返回）[58]。因此，子系统调用将不可避免地带来性能开销，从而在 Skadi 与 Zephyr 之间形成安全性与性能权衡。

表7：Skadi、Zephyr 和 Linux 的计 测试。

基准测试	Linux	Zephyr	Skadi	Δ
Coremark/秒	92.10	110.02	110.14	0.13
流复制（秒）	11.23	6.04	6.00	−0.04
流缩放（秒）	12.75	6.66	6.65	−0.01
流加法（秒）	17.53	9.03	9.02	−0.02
流三元组（秒）	18.62	9.50	9.50	0.00

表8：Skadi、Zephyr 和 Linux 的网络基准 MQT 略）。提供子系统调用次数（ping 和 iperf 为总计，其余为每次迭代）作为上下文。

基准测试	平均值±标准差		Δ
Ping 延迟（毫秒）	2.56 ±	0.08	
	（Zephyr）0.88 ±	0.02	× 2。92
	（Linux）1.92 ±	0.15	x 1.33
子系统调用	59788 ±	-	
iperf TCP（千比特/秒）	1499.33 ±	2.52	
	(Zephyr) 4396.67 ±	6.81	x 0。34
	(Linux) 9031.17 ±	1733.20	x 0。17
子系统调用	1051756 ±	-	
接收持续时间（纳秒）	2 138 493.63 ±	13 707.45	
	(Zephyr) 534 142.81 ±	8120.03	x 4。00
	(Linux) 1 346 524.21 ±		x 1。59
子系统调用	290.05 ±	11.43	
发送持续时间（纳秒）	1 516 145.02 ±	11 670.62	
	（Zephyr）530 697.96 ±	13 202.11	x 2.86
	（Linux）1 050 532.63 ±	203 162.69	x 1.44
子系统调用	235.08 ±	2.43	
MQTT-TLS（纳秒）	2 057 174 000.84 ±	5 772 516.87	
	1 993 484 845.47 ±	30 590.98	x 1.03
子系统调用	2058.80 ±	147.17	

7.4 系统：计算性能

我们使用 EEMBC Coremark [33] 和 Stream [52] 来衡量计算与内存性能。结果见表7。扩展结果见附录中的表10。总体而言，Skadi 与 Zephyr 在 Bredi 上的差异可以忽略不计。这可以解释为 L1 NTLB 的命中率达到 100%，不会给 cva6 的流水线带来任何延迟，并且在基准测试期间没有子系统调用。

Linux 的性能相对较低，这可以用 TLB 缺失来解释。我们采用 cva6 的默认设置，分别配置 16 个指令和数据 TLB，它们由内核和用户空间共享。我们并不声称这一结果具有普遍适用性。我们仅提供这一数值，以指出 Zephyr 和 Skadi 的性能与 Linux 处于同一数量级。

7.5 系统：网络协议栈性能

我们使用 *ping* 测量 ICMP回显往返时间，使用 *iperf* 和 *zperf* 测量 TCP吞吐量，使用硬件时间戳测量网络协议栈延迟，并采用与 CHERIoT [8] 类似的 MQTT 场景来评估网络性能。对于网络协议栈延迟，我们使用 ha1588 硬件时间戳单元（TSU）采集的 硬件时间戳，以确定网络数据包经由 MAC-PHY接口发送或接收的精确时间。然后，以 ha1588 的时间为基准，分别确定发送或接收时间与数据包从套接字返回或写入套接字的时间之间的延迟。在 Linux 上，接收和发送延迟通过标准软件时间戳测量，因此测量值会略微偏低。

表 9：Skadi 和 Zephyr 的 IRQ基准测试结果

基准测试	平均值±标准差（纳秒）	Δ
独立	68 423.16 ± 1529.02	
	24 127.16 ± 32 149.03	x 2.84
子系统调用	1144.68 ± 18.55	
单调周期	136 000	
	70 000	x 1。94
单调	58 826.32 ± 18 714.18	
	8392.42 ± 12 120.04	x 7.01
子系统调用	7.02 ± 0.14	

我们的 MQTT 场景基于 Zephyr 的 MQTT 发布者示例，并启用了 TLS；我们测量在 100 次迭代中发布三条消息所需的时间。Linux 上省略*MQTT*，因为 MQTT 实现特定于 Zephyr。

我们在表 8 中报告了最重要的结果。完整结果见附录 D 中的表11。Skadi 和 Northcape 在我们的互联网控制消息协议和 zperf 场景中带来的性能开销相当显著。具体而言，我们测得基准方案与 Skadi 场景之间的开销约为 3 至 4 倍。这是 Skadi 提供了强隔离的结果，子系统调用数量较多也印证了这一点。不过，我们认为，从绝对数值来看，Skadi 的性能仍足以满足大多数嵌入式应用的需求。具体而言，除 TCP吞吐量外，所有测量结果相较于 Linux 的开销均低于两倍。由于 Zephyr 的 TCP 吞吐量也低于 Linux，我们推测 Zephyr 的网络协议栈在我们的平台上的整体性能较低。

我们认为未来有两个优化方向。首先，我们的隔舱概念与 Zephyr 的子系统尽可能保持一致。这是系统最细粒度的隔舱划分；而在实际场景中，某些隔舱可以合并为一个，从而减少子系统调用。例如，Linux [49]中的以太网驱动程序和 DMA 驱动程序就是如此。其次，可以考虑消除以太网和 DMA 驱动程序、网络子系统与调度器之间的子系统调用，但代价是破坏现有的 API 和驱动程序模型。

7.6 系统：实时能力

为评估 Skadi 的实时能力，我们采用了文献中多种适宜评估方法的折中方案：我们评估 Zephyr 的调度器微基准测试 [37]，以及周期性中断响应时间和未发生中断未命中时的最大中断频率 [13]。

为检验 Skadi 上的可用性保证，我们将外部中断设为不可屏蔽，并在忙等待时禁用中断。Zephyr 场景则在启用中断的情况下运行，否则将无法完成。

我们的中断请求实验结果见见表 9。扩展结果以及我们的调度器微基准测试可见于附录中的表 12 和表 13。

总的来说，Skadi 在固定延迟、延迟范围和标准差方面都会产生开销。在中断请求场景中，Skadi 的开销为 3 至 7 倍。与第 7.5 节测得的网络开销类似，这是由隔离造成的（参见所需的子系统调用次数）。不过从绝对数值来看，Skadi 仍能在约 110 毫秒的千分之一内响应中断，且标准差相当。

调度器基准测试的结果表明，Skadi 的强隔离带来的开销大多在 3 至 10 倍之间。一个值得注意的异常值是线程取消——发生这种情况时，Skadi 需要在每个子系统中调用一次回调，以释放可能仍与该线程关联的栈。同样，尤其考虑到绝对数值，我们认为这些结果表明 Skadi 适合作为软实时系统。

8 相关工作

我们讨论三个相关工作领域：首先，介绍与先前发表的 Northcape 概念 [1] 的差异。然后，将 Bredi 和 Skadi 与相关能力架构进行比较，最后讨论非能力方法。

8.1 对 Northcape 的改进

Ackermann 等人 [1] 提出了 Northcape，将其设想为一种纯理论的能力系统，不包含硬件实现和操作系统。此外，Bredi 对 Northcape 的实现从三个关键方面改进了这一概念。

防止猜测能力令牌和子系统标识符。 Northcape 最初在一个架构寄存器 *tid*（任务 ID）中使用秘密的子系统标识符。每个已锁定的能力都只能由锁定它的子系统访问，并通过*tid* 来区分子系统。因此，独占访问始终是按子系统划分的。这种机制有一个关键缺陷：*tid* 可能泄露或被猜中，从而绕过独占访问。我们移除了*tid*寄存器，并重新设计了锁定机制和限制机制：二者现在彼此独立。因此，锁持有者令牌和子系统 *ID* 限制是我们的贡献。

扩展访问权限集合。 除了能力访问权限读取、写入、执行和可锁定外，Bredi 还增加了中断请求_可访问、可缓存_数据和可缓存_*TLB*。中断请求_可访问用于控制中断服务例程是否可以访问能力，例如，可用于防止中断服务例程覆盖定时器存根，否则会破坏 ISR 隔离。缓存权限可用于特定子系统中的硬实时处理。

隐式子系统调用。 最后，Bredi 对调用 操作进行了两项更改。首先，Bredi 中的子系统调用可以使用常规跳转指令完成。为此，

cva6 MMU 确保相互隔离，而无论 启动子系统调用的是哪条指令：任何试图通过跳过第一条指令来破坏控制流，或从设置子系统 ID 能力中读取机密数据的行为，都会触发异常。调用 的存在仅用于优化性能。其次，Bredi 的调用 不会返回上下文指针，而是将上下文保存在 Skadi 中的设置子系统 ID 能力内。这提高了软件的灵活性。

8.2 能力系统

大多数能力系统在其核心概念中并未考虑来自设备的攻击（*DMA* 攻击）（有些系统只是因为它们早于 DMA 的出现 [21, 47, 79]）。CAPSTONE [83] 和 CHERI [80] 等能力系统侧重于保护软件，使其免受系统级软件对手的攻击。若干相关能力系统也只关注进程内隔离，即在可信内核环境中，于进程地址空间内隔离彼此不信任的组件或库 [31, 43, 46, 76]。

近期，CHERI 扩展了对类似 IOMMU 的功能的支持 [24]。所提出的 CapChecker 使用一组 CHERI 能力进行初始化，DMA 设备通过编号来识别这些能力。随后，DMA 事务会相对于相应能力进行解释，从而强制实现空间内存安全。至关重要的是，这项工作依赖可信驱动，会严格限制设备可使用的能力数量，并且要实现全面安全，还要求改动 DMA 设备的硬件。Bredi 及其底层的 Northcape 没有这些限制，能够提供空间和时间内存安全，并允许 CPU 和设备创建能力。

此外，也有人提出在 CHERI 平台上采用用户空间驱动，通过能力访问设备 MMIO 的安全子集，而无需系统调用 [32, 54]。不过，这种设计需要一个可信内核：内核级驱动程序必须设置输入输出内存管理单元或图形转换表（即 GPU 的内存管理单元），以防驱动程序滥用设备，成为混淆代理。这些机制以页为粒度运行，因此原则上无法防止 Thunderclap [50] 等攻击。

此外，CHERIoT RTOS [9] 与 Skadi 类似，由一个完全特权的仅启动时加载器引导非特权隔舱。不过，在 CHERIoT RTOS 中，Switcher 和分配器仍属于运行时可信计算基 [9]。dIPC 系统 [74] 同样为 CODOMs架构 [73] 提出了通过跳板和可信中介实现直接进程间通信的方案，这与 CHERIoT RTOS 类似。此外，CHERI 领域的其他不可信操作系统要么依赖特权运行时纳内核 [35]，要么未将 DMA 设备纳入威胁模型 [7, 9]。

关于M^3 操作系统 [14, 39, 40, 59]的相关工作采取了不同的方法：计算机系统可以解聚为承载CPU或加速器的隔离瓦。M^3 内核随后可以借助分布式来促进各瓦之间或瓦与系统内存之间的交互能力系统。这由类似于Bredi AXI MMU的逐瓦访问控制组件强制执行。关键在于，与其他微内核操作系统一样，M^3 微内核构成了运行时软件可信计算基。此外，由于处理器核心使用内存管理单元或物理内存保护机制等传统内存保护，因此不支持具有相互隔离的本地子系统调用 。

Bastion [61] 实现了类似于M^3, 的架构，但依赖于在 OpenTitan 上运行的自定义固件，而非M^3 ，并使用访问控制列表（ACL）进行内存访问控制。作者还提出将基于令牌的能力方案作为未来工作。

8.3 非能力方法

与安全地隔离内核软件中的漏洞不同，也可以努力防止内核代码中出现可利用的漏洞。微内核操作系统正是这样做的：它们将内核中所有不严格要求以高硬件特权运行的部分移至用户空间，从而缩小整个系统的攻击面 [20, 34, 42]。随后使用高性能进程间通信，在各隔舱之间进行通信。SeL4 [44] 将这种方法与微内核的形式化验证相结合，形式化证明可信计算基中不存在漏洞。我们认为 SeL4 是一种与缩减运行时可信计算基相正交的方法：SeL4 致力于防止软件可信计算基遭到利用，但代价是每次更改可信计算基都需要进行成本高昂的重新验证 [34]。Skadi 则旨在消除可信计算基，但代价是性能、芯片面积和代码大小。未来工作中可以结合这两种方法，例如对 Skadi 加载器和 Bredi 硬件可信计算基进行形式化验证。

机密计算也致力于缩减软件可信计算基。这项技术的商用实现包括 ARM CCA [12],、Intel TDX [26] 和 AMD SEV [6]。此外，也有人提出了 CHERI-TrEE——一种面向 CHERI 架构的机密计算系统 [72]。针对 CCA，已有多种方案被提出，以便在设备、操作系统和应用软件遭到攻陷时仍能提供安全保障 [19, 41, 75, 88]。特别是，HiveTEE [41]承诺能够将任意数量的安全应用隔舱与不可信设备、操作系统和虚拟机监控器隔离开来，其安全模型与 Skadi 类似。然而，机密计算技术依赖运行时可信软件监控器来配置硬件安全原语并实现隔舱间切换，因此可信软件组件中的漏洞仍会危及整个系统的安全性 [66]。此外，底层硬件保护机制限制了可实现的粒度（例如，HiveTEE 的页粒度或 16 字节 MTE 内存区域）。CHERI-TrEE 借助 CHERI 提供字节粒度内存保护和子系统调用，但它依赖可信软件中断处理程序作为其运行时可信计算基，且未考虑不可信设备 [72]。

机密计算领域持续取得的进展表明，

即使在基于内存管理单元的系统中，也可以将必要的可信计算基降至最小。不过，采用基于能力的内存访问控制有两项优势：能力支持委派和通过操作进行非特权修改，使组件能够在无需可信中介的情况下，高效且安全地共享内存。反过来，这有助于实现子系统调用，而子系统调用是 Skadi 中相互隔离互不信任组件的关键构建模块。此外，能力可提供字节粒度的安全性，而内存管理单元处理的是更大的页面。总之，Skadi 和 Bredi 可以通过函数粒度、相互不信任和字节粒度，保护任何软件免受其他任何运行时软件或硬件的侵害。

然而，Bredi 不涉及总线窃听或冷启动攻击等物理攻击，而 SGX、CCA、安全加密虚拟化和 TDX 能够提供针对这类攻击的安全防护。未来可以扩展 Bredi，以弥补这一不足。

最后，Skadi 仅提供一种用于隔离的机制 。因此，对于 Lefeuvre 等人 [45],定义的属性，我们能够提供安全性和可靠性 ，并且与旧版 Zephyr 模块具有有限的源代码兼容性 。至关重要的是，我们的工作重点是实现无需运行时可信计算基的安全性，而 Lefeuvre 等人调查的所有工作都未能做到这一点。Skadi 可以利用自动隔离策略 来提升易用性 [45]。

9 结论与未来工作

我们介绍了 Skadi 操作系统和 Bredi SoC，后者是首个实现 Northcape 能力架构的方案，并在概念上进行了重大改进。Skadi 和 Bredi 相结合，使嵌入式设备即使面对恶意应用软件、操作系统内核和外设 也能实现安全，且无需运行时软件可信计算基 ，同时具备软实时适用性。

在本研究的范围内，我们主要关注原型的安全性 。通过添加自定义编译器优化，或采用可避免域切换的其他编程模型，还可以进一步提升性能 [65]。

Bredi 和 Skadi 为未来研究奠定了很有前景的基础。Skadi 和 Bredi 所提供的高度安全性，加上其实时运行特性，使它们适合作为赋能技术，例如用于 5G 基站等高安全性网络基础设施设备。硬件支持的子系统 *ID*与限制 相结合，也是一种非常强大的原语，可按每种能力为系统外设提供额外的安全保证。

参考文献

[1] Eric Ackermann、Noah Mauthe 和 Sven Bugiel。《进行中的工作：Northcape：嵌入式实时基于能力的寻址》。发表于操作系统与虚拟化安全研讨会，2024年。网址 https://ieeexplore.ieee.org/abstract/document/10628723。[2] 超威半导体公司。《AXI DMA LogiCORE IP 产品指南》。LogiCORE IP 产品指南 PG021，2022年。网址 https://docs.amd.com/r/en- US/pg021_axi_dma。[3] 超威半导体公司。《AXI 互连 v2.1》。LogiCORE IP 产品指南 PG059，2022年。网址 https://www.xilinx.com/support/documents/ ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf。[4] 超威半导体公司。《AXI 1G/2.5G 以太网子系统 v7.2》。LogiCORE IP 产品指南 PG138，2024年。网址 https://docs.amd.com/r/en-US/pg138- axi-ethernet。[5] 超威半导体公司。Vivado，2024年。网址 https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2024-2.html。[6] 超威半导体公司。AMD 安全加密虚拟化（SEV），2025年。网址 https://www.amd.com/en/developer/sev.html。[7] Hesham Almatary、Michael Dodson、Jessica Clarke、Peter Rugg、Ivan Gomes、Michal Podhradsky、Peter G. Neumann、Simon W. Moore 和 Robert N. M. Watson。《CompartOS：面向嵌入式系统的 CHERI 分区隔离》，2022年。网址 http://arxiv.org/ abs/2206.02852。[8] Saar Amar、David Chisnall、Tony Chen、Nathaniel Wesley Filardo、Ben Laurie、Kunyan Liu、Robert Norton、Simon W. Moore、Yucong Tao、Robert N. M. Watson 和 Hongyan Xia。《CHERIoT：为嵌入式设备实现完整内存安全》。发表于 MICRO'23，2023年。网址 https: //dl.acm.org/doi/10.1145/3613424.3614266。[9] Saar Amar、Tony Chen、David Chisnall、Nathaniel Wesley Filardo、Ben Laurie、Hugo Lefeuvre、Kunyan Liu、Simon W. Moore、Robert Norton-Wright、Margo Seltzer、Yucong Tao、Robert N. M. Watson 和 Hongyan Xia。《CHERIoT RTOS：面向低成本嵌入式设备上细粒度内存安全隔舱的操作系统》。发表于 PLOS'25，2025年。网址 https://dl.acm.org/doi/ 10.1145/3731569.3764844。

[10] Arm Ltd. AMBA AXI 和 ACE 协议规范，2011年。网址：https://developer.arm.com/documentation/ihi0022/d/。

[11]Arm Ltd. AMBA AXI-Stream 协议规范，2021年。网址：https: //documentation-service。arm。com/static/ 64819f1516f0f201aa6b963c。

[12] Arm Ltd. Arm CCA 安全模型，2021年。网址：https://documentation-service。arm。com/static/610aaec33d73a34b640e333b。

[13] Rafael V. Aroca 和 Glauco Caurin。实时操作系统（RTOS）比较。载于*WSO-* 操作系统研讨会，2009年。URL http://home.iitj.ac.in/~saurabh.heda/Papers/ Survey/RTOS% 20Performance%20Comparison%20- 2008.pdf。

[14]Nils Asmussen、Till Miemietz、Sebastian Haas 和 Michael Roitzsch。通过将计算与隔离分离来实现对核心的不信任。系统架构杂志，159，2025年。URL https://linkinghub.elsevier.com/retrieve/pii/S1383762124002650。

[15]Roberto Avanzi。QARMA 分组密码族。含零因子的环上的近似 MDS 矩阵、具有非对合中心轮的近似对称 Even-Mansour 构造，以及低延迟 S 盒的搜索启发式。《*IACR* 对称密码学汇刊》，2017年。URL https://tosc.iacr.org/index.php/ToSC/article/view/583。

[16] Raad Bahmani、Ferdinand Brasser、Ghada Dessouky、Patrick Jauernig、Matthias Klimmek、Ahmad-Reza Sadeghi 和 Emmanuel Stapf。{CURE}：一种具有{CUstomizable}和弹性安全飞地的安全架构。发表于*USENIX* 安全大会 *2021*，2021年。网址：https://www.usenix.org/system/files/sec21- bahmani.pdf。

[17] Gal Beniamini。Project Zero：无线攻击：利用 Broadcom 的 Wi-Fi 协议栈（第 1 部分），2017年。网址：https: //googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html。

[18] Luca Benini、Alberto Macii 和 Massimo Poncino。嵌入式存储器的能效感知设计：技术、架构与优化技术综述。《*ACM* 嵌入式计算系统汇刊》，2(1)，2003年。网址：https://dl.acm.org/doi/10.1145/605459.605461。

[19]Andrin Bertschi、Supraja Sridhara、Friederike Groschupp、Mark Kuhne、Benedict Schlüter、Clément Thorens、Nicolas Dutly、Srdjan Capkun 和 Shweta Shinde。Devlore：将 Arm CCA 扩展至集成设备——从内存迈向中断隔离之旅，2024年。URL：http://arxiv。org/abs/2408。05835。

[20] Carl van Schaik 和 Gernot Heiser。《ARM 和分段架构上的高性能微内核与虚拟化》载于 嵌入式系统微内核国际研讨会，2007年。网址：http: //hdl.handle.net/1959.4/39903。[21] Nicholas P. Carter、Stephen W. Keckler 和 William J. Dally。《面向快速基于能力的寻址的硬件支持》。*ACM SIGOPS*《操作系统评论》，28（5），1994年。网址：https://dl.acm.org/doi/10.1145/381792.195579。

[22] Haogang Chen、Yandong Mao、Xi Wang、Dong Zhou、Nickolai Zeldovich 和 M. Frans Kaashoek。Linux 内核漏洞：最先进的防御措施与未解决的问题。载于第二届亚太系统研讨会，2011年。URL：https://dl。acm。org/doi/10。1145/ 2103799。2103805。

[23] Yueqi Chen 和 Xinyu Xing。SLAKE：通过促进 Slab 操纵来利用 Linux 内核中的漏洞。载于*CCS'19*，2019年。URL：https://dl。acm。org/ doi/10。1145/3319535。3363212。

[24] Jianyi Cheng、A.Theodore Markettos、Alexandre Joan- nou、Paul Metzger、Matthew Naylor、Peter Rugg 和 Timothy M. Jones。《面向异构加速器的自适应 CHERI 分区》。载于*ISCA'25*，2025年。网址：https://dl.acm.org/doi/ 10.1145/3695053.3731062。

[25]Kito Cheng 和 Jessica Clarke。《RISC-V ABI 规范》，2024年。网址：https: //github.com/riscv-non-isa/riscv-elf-psabi-doc/releases/download/draft-20240829-13bfa9f54634cb60d86b9b333e109f077805b4b3/riscv-abi.pdf。

[26] Pau-Chen Cheng、Wojciech Ozga、Enriquillo Valdez、Salman Ahmed、Zhongshu Gu、Hani Jamjoom、Hubertus Franke 和 James Bottomley。《Intel TDX 揭秘：自上而下的方法》。*ACM Comput. Surv.*，56（9），2024年。网址：https://dl.acm.org/doi/10.1145/ 3652597。

[27] David Chisnall、Colin Rothwell、Robert N.M. Watson、Jonathan Woodruff、Munraj Vadera、Simon W. Moore、Michael Roe、Brooks Davis 和 Peter G. Neumann。《超越 PDP-11：支持内存安全 C 抽象机的架构》。*ACM SIGARCH* 计算机架构新闻，43(1)，2015年。网址：https: //dl.acm.org/doi/10.1145/2786763.2694367。

[28] Yann Collet。Lz4/lz4：极速压缩算法，2025年6月。网址：https://github.com/lz4/ lz4。

[29] Ang Cui、Michael Costello 和 Salvatore J. Stolfo。《当固件修改发起攻击：一个案例

研究：嵌入式系统漏洞利用》。发表于 NDSS'13，2013年。网址：http://ids.cs.columbia.edu/sites/default/files/ndss-2013.pdf。[30] Jack B. Dennis 和 Earl C. Van Horn。多道程序计算的编程语义。《ACM 通讯》，9(3)，1966年。网址：https: // dl.acm.org/doi/10.1145/365230.365252。[31] Kha Dinh Duy、Kyuwon Cho、Taehyun Noh 和 Hojoon Lee。Capacity：面向现代 ARM 架构、通过密码学强制实施的进程内能力。发表于 CCS'23，2023年。网址：https://dl.acm.org/doi/10.1145/3576915.3623079。[32] Friedrich Doku、Jonathan Laughton、Nick Wanninger 和 Peter Dinda。CAPIO：利用能力安全地对通用设备进行内核旁路，2025年。网址：http://arxiv.org/abs/2512.16957。[33] EEMBC。Coremark。嵌入式微处理器基准测试联盟，2025年。网址：https://github.com/ eembc/coremark。[34] Kevin Elphinstone 和 Gernot Heiser。从 L3 到 seL4：20年来的 L4 微内核研究让我们学到了什么？发表于 SOSP'13，2013年。网址：https://dl.acm.org/doi/ 10.1145/2517349.2522720。[35] Lawrence Esswood。CheriOS：利用能力硬件和最小化虚拟机监控器，设计不可信的单地址空间能力操作系统。博士论文，剑桥大学，2021年。网址：https://doi.org/10.17863/CAM.74163。[36] Ronald Fagin、Jurg Nievergelt、Nicholas Pippenger 和 H. Raymond Strong。可扩展哈希——一种用于动态文件的快速访问方法。《ACM 数据库系统汇刊》，4(3)，1979年。网址：https://dl.acm.org/doi/10.1145/320083.320092。[37] A. Garcia-Martinez、J.F. Conde 和 A. Vina。现代实时操作系统性能评估的综合方法。发表于 Euromicro，1996年。网址：https://ieeexplore.ieee.org/abstract/document/546366。[38] Merve Gülmez、Håkan Englund、Jan Tobias Mühlberg 和 Thomas Nyman。Mon CHÉRI：利用条件能力缓解未初始化内存访问。发表于 SP'25，2025年。网址：https://ieeexplore.ieee.org/abstract/document/11023423。[39] Sebastian Haas、Mattis Hasler、Friedrich Pauls、Christopher Dunkel、Yogesh Verma、Nilanjana Das 和 Michael Raitza。一种将硬件加速器安全集成到……中的架构方法

可信MPSoC平台。载于国际嵌入式多核/众核片上系统研讨会，2025年。URL https://ieeexplore.ieee.org/abstract/document/11311008/.[40] Matthias Hille、Nils Asmussen、Pramod Bhatotia和Hermann Härtig。SemperOS：分布式能力系统。载于USENIX ATC'19，2019年。URL https://www.usenix.org/system/files/atc19-hille.pdf.[41] Haoyang Huang和Fengwei Zhang。HiveTEE：由RME与MTE协同辅助的可扩展细粒度隔离域。即将发表于IEEE信息取证与安全汇刊，21，2026年。URL https://fengweiz.github.io/paper/ hivetee-tifs26.pdf.[42]Henrik Karlsson和Roberto Guanciale。基于能力控制的时间与空间分区的分区内核。载于RTSS'25，2025年。URL https://ieeexplore.ieee.org/abstract/document/11315078/.[43]Yonghae Kim、Anurag Kar、Jaewon Lee、Jaekyu Lee和Hyesoon Kim。RV-CURE：面向完全内存安全的RISC-V能力架构。IEEE计算机汇刊，74(10)，2025年。URL https://ieeexplore.ieee.org/abstract/document/11087533.[44] Gerwin Klein、Kevin Elphinstone、Gernot Heiser、June Andronick、David Cock、Philip Derrin、Dhammika Elkaduwe、Kai Engelhardt、Rafal Kolanski、Michael Norrish、Thomas Sewell、Harvey Tuch和Simon Winwood。seL4：操作系统内核的形式化验证。载于SOSP'09，2009年。URL https://dl.acm.org/doi/10.1145/1629575.1629596.[45]Hugo Lefeuvre、Nathan Dautenhahn、David Chisnall和Pierre Olivier。SoK：软件隔离。载于SP'25，2025年。URL https://ieeexplore.ieee.org/abstract/document/11023477/.[46] Michael LeMay、Joydeep Rakshit、Sergej Deutsch、David M. Durham、Santosh Ghosh、Anant Nori、Jayesh Gaur、Andrew Weiler、Salmin Sultana、Karanvir Grewal和Sreenivas Subramoney。密码学能力计算。载于MICRO'21，2021年。URL https: // dl.acm.org/doi/10.1145/3466752.3480076.[47] Henry M. Levy。基于能力的计算机系统。2014年。ISBN 978-1-4831-0106-4.[48] Zhenpeng Lin、Yuhang Wu和Xinyu Xing。DirtyCred：在Linux内核中提升特权。载于CCS'22，CCS '22，2022年。URL https://dl.acm.org/doi/10.1145/ 3548606.3560585.

[49] Linux 贡献者。Xilinx_axienet_main.c，2025年。URL https://github.com/torvalds/linux/blob/master/drivers/net/ethernet/xilinx/ xilinx axienet main.c。

[50] A. Theodore Markettos、Colin Rothwell、Brett F. Gutstein、Allison Pearce、Peter G. Neumann、Simon W. Moore 和 Robert N. M. Watson。《Thunderclap：通过不可信外设的直接内存访问探索操作系统输入输出内存管理单元保护中的漏洞》。载于 *NDSS'19*，1995年。URL：https: // www.ndss-symposium.org/wp-content/uploads/ 1995年/02/ndss2019_05A-1_Markettos_paper.pdf。

[51] A. Theodore Markettos、John Baldwin、Ruslan Bukin、Peter G. Neumann、Simon W. Moore 和 Robert N. M. Watson。《立场论文：使用 CHERI 能力防御直接内存访问》。载于*HASP'21*，2021年。URL：https://dl.acm.org/doi/ 10.1145/3458903.3458910。

[52] John D. McCalpin。《当前高性能计算机中的内存带宽与机器平衡》。*IEEE*计算机学会计算机体系结构技术委员会（*TCCA*）通讯，2（19-25），1995年。

[53] 梅法强、Lei Zhang、Chongyan Gu、Yuan Cao、Chenghua Wang 和 Weiqiang Liu。《基于现场可编程门阵列的高灵活性、轻量级高速真随机数生成器》。发表于*IEEE*计算机学会年度超大规模集成电路研讨会，2018年。网址：https: //ieeexplore.ieee.org/document/8429400/。

[54] Paul Metzger、A. Theodore Markettos、Edward Tomasz Napierała、Matthew Naylor、Robert N. M. Watson 和 Timothy M. Jones。《利用用户空间进程和 CHERI 分区高效降低低级 GPU 驱动程序的权限》。发表于 *CCS'25*，2025年。网址：https://www.cl.cam.ac.uk/~tmj32/ papers/docs/metzger25-ccs.pdf。

[55]Mark S. Miller、Ka-Ping Yee 和 Jonathan Shapiro。《能力迷思的破除》。技术报告 SRL2003-02，约翰斯·霍普金斯大学系统研究实验室，2003年。网址：https://www-users.cse.umn.edu/classes/Fall-2019/csci5271/papers/SRL2003-02.pdf。

[56] Tim Newsome 和 Megan Wachs。《RISC-V 外部调试支持》，2024年。网址：https://github.com/riscv/riscv-debug-spec/releases/download/ task_group_vote/riscv-debug-draft.pdf。

[57] Dorottya Papp、Zhendong Ma 与 Levente Buttyan。嵌入式系统安全：威胁、漏洞与攻击分类法。载于隐私、安全与信任年度会议，2015年。URL http:// ieeexplore.ieee.org/document/7232966/。

[58] David A. Patterson 与 Andrew Waterman。*RISC-V* 读本：开放架构图集。2017年。ISBN 978-0-9992491-1-6。[59] Friedrich Pauls、Sebastian Haas 与 Yogesh Verma。面向安全 AI 加速器集成的最小化信任平台。载于国际片上系统设计会议，2025年。URL https://ieeexplore.ieee.org/abstract/document/11329663/。

[60] PULP项目。Pulp-platform/axi。pulp-platform，2025年。网址：https://github.com/pulp-platform/axi。

[61] Francesco Restuccia、Zhenghua Ma、Joseph Zucker- man、Andres Meza、Biruk Seyoum、Luca Carloni 和 Ryan Kastner。《BASTION：一种用于基于NoC的片上系统平台中安全集成第三方IP的框架》。《密码硬件与嵌入式系统汇刊》，2025年第4期，2025年。网址：https:// escholarship.org/uc/item/8dz9c84n。

[62]RISC-V国际组织。《RISC-V指令集手册：第二卷》，2025年。网址：https://docs.riscv.org/reference/isa/_attachments/riscv-privileged.pdf。

[63]RISC-V国际组织。核心本地中断控制器（CLIC）RISC-V特权架构扩展，2025年3月17日：。网址 https://github.com/riscv/riscv-fast-interrupt/releases/download/v0.10/clic.pdf。

[64] John K. Salmon、Mark A. Moraes、Ron O. Dror 和 David E. Shaw。并行随机数：简单得就像 1、2、3。发表于高性能计算、网络、存储与分析国际会议，2011年。ISBN 978-1-4503-0771-0。网址 https://doi.org/ 10.1145/2063384.2063405。

[65] Vasily A. Sartakov、Stefan Brenner、Sonia Ben Mokhtar、Sara Bouchenak、Gaël Thomas 和 Rüdiger Kapitza。EActors：使用 SGX 实现快速灵活的可信计算。发表于*Middleware'18*，2018年。网址 https://doi.org/ 10.1145/3274808.3274823。

[66] Benedict Schlüter、Christoph Wech 和 Shweta Shinde。Heracles：针对 AMD SEV-SNP 的选择明文攻击。发表于 *CCS' 25*，2025年。网址：https://heracles-attack.github.io/Heracles-CCS2025.pdf。

[67] Alireza Shameli-Sendi。了解 Linux 内核漏洞。计算机病毒学与黑客技术期刊，17(4)，2021年12月。网址：https://doi.org/10.1007/s11416-021-00379-x。

[68]Miguel Silva、David Cerdeira、Sandro Pinto和Tiago Gomes。面向互联网的操作系统

物联网低端设备：分析与基准测试。*IEEE*物联网期刊，6（6），2019年。URL https://ieeexplore.ieee.org/abstract/document/8824106/.[69] Lau Skorstengaard、Dominique Devriese和Lars Birkedal。关于具有局部能力的机器的推理。载于编程语言与系统，2018年。[70] MITRE公司。MITRE EMB3D，2025年。URL https://emb3d.mitre.org/.[71] TIS委员会。工具接口标准（TIS）可执行与链接格式（ELF）规范，1995年。URL https://refspecs.linuxbase.org/elf/ elf.pdf.[72]Thomas Van Strydonck、Job Noorman、Jennifer Jackson、Leonardo Alves Dias、Robin Vanderstraeten、David Oswald、Frank Piessens和Dominique Devriese。CHERI-TrEE：能力机上的灵活安全飞地。载于*EuroS&P'23*，2023年。URL https://ieeexplore.ieee.org/abstract/document/10190507.[73] Lluïs Vilanova、Muli Ben-Yehuda、Nacho Navarro、Yoav Etsion和Mateo Valero。CODOMs：用以代码为中心的内存域保护软件。*ACM SIGARCH*《计算机体系结构新闻》，42（3），2014年。URL https://dl.acm.org/doi/10.1145/2678373.2665741.[74] Lluís Vilanova、Marc Jordà、Nacho Navarro、Yoav Etsion和Mateo Valero。直接进程间通信（dIPC）：重新利用CODOMs架构以加速进程间通信。载于*EuroSys'17*，2017年。URL https: //dl.acm.org/doi/10.1145/3064176.3064197.[75] Chenxu Wang、Fengwei Zhang、Yunjie Deng、Kevin Leach、Jiannong Cao、Zhenyu Ning、Shoumeng Yan和Zhengyu He。CAGE：用GPU扩展补充Arm CCA。载于*NDSS'24*，2024年。URL https://www.ndss-symposium.org/wp-content/uploads/2024-763-paper.pdf.[76]Robert N. M. Watson、Jonathan Anderson、Ben Laurie和Kris Kennaway。Capsicum：面向{UNIX}的实用能力。载于*USENIX Sec'10*，2010年。URL https://www.usenix.org/events/ sec10/tech/full_papers/Watson.pdf.[77] Robert N. M. Watson、Jonathan Woodruff、Peter G. Neumann、Simon W. Moore、Jonathan Anderson、David Chisnall、Nirav Dave、Brooks Davis、Khilan Gudka、Ben Laurie、Steven J. Murdoch、Robert Norton、Michael Roe、Stacey Son和Munraj Vadera。CHERI：面向可扩展软件隔离的混合能力系统架构。发表于*SP'14*，2015年。网址：https://ieeexplore.ieee.org/abstract/document/7163016.[78] Paul R. Wilson、Mark S. Johnstone、Michael Neely 和 David Boles。动态存储分配：一项调查与批判性评述。发表于 内存管理，1995年。[79] Stanley Winkler。计算机通信——影响与意义。华盛顿特区，1972年。网址：https://archive.computerhistory.org/resources/access/text/2022/12/102714088-05-01-acc.pdf.[80] Jonathan Woodruff、Robert N. M. Watson、David Chisnall、Simon W. Moore、Jonathan Anderson、Brooks Davis、Ben Laurie、Peter G. Neumann、Robert Norton 和 Michael Roe。CHERI能力模型：在风险时代重新审视RISC。发表于*ISCA'14*，2014年。网址：https://ieeexplore.ieee.org/document/6853201.[81] Hongyan Xia、Jonathan Woodruff、Sam Ainsworth、Nathaniel W. Filardo、Michael Roe、Alexander Richardson、Peter Rugg、Peter G. Neumann、Simon W. Moore、Robert N. M. Watson 和 Timothy M. Jones。CHERIvoke：使用CHERI能力表征指针撤销，以实现时间内存安全。发表于*MICRO'19*，2019年。网址：https://dl.acm.org/doi/10.1145/3352460.3358288.[82] Wen Xu、Juanru Li、Junliang Shu、Wenbo Yang、Tianyi Xie、Yuanyuan Zhang 和 Dawu Gu。从碰撞到利用：揭示Linux内核中的释放后使用漏洞。发表于*CCS'15*，2015年。网址：https://dl.acm.org/doi/10.1145/2810103.2813637.[83] Jason Zhijingcheng Yu、Conrad Watt、Aditya Badole、Trevor E. Carlson 和 Prateek Saxena。Capstone：用于可信、无信任内存安全访问的基于能力的基础架构。发表于*USENIX Sec'23*，2023年。网址：https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jason.[84] A. S. Yurchenko。动态分段内存分配算法。控制论，17(1)，1981年。网址：https://doi.org/10.1007/BF01307036.[85]Florian Zaruba 和 Luca Benini。应用级处理的成本：22纳米FDSOI技术下、Linux就绪的1.7 GHz 64位RISC-V核心的能耗与性能分析。*IEEE*超大规模集成（*VLSI*）系统汇刊，27(11)，2019年。网址：https://ieeexplore.ieee.org/abstract/document/8777130。

[86] Zephyr 项目成员和个人贡献者。Zephyr。Zephyr 项目，2025年。网址：https://github.com/zephyrproject-rtos/zephyr。

[87]Zephyr 项目成员及个人贡献者。《设备驱动模型》，2025年。网址：https://docs。zephyrproject。org/latest/kernel/drivers/index。html。

[88] Yiming Zhang、Yuxin Hu、Zhenyu Ning、Fengwei Zhang、Xiapu Luo、Haoyang Huang、Shoumeng Yan 和 Zhengyu He。《{SHELTER}：将用户空间隔离扩展至 Arm {CCA}》。发表于*USENIX Sec'23*，2023年。网址：https://www。usenix。org/conference/usenixsecurity23/presentation/zhang-yiming。

Northcape：概述

Northcape 是一种硬件能力架构，最初由 Ackermann 等人提出 [1]。Northcape 的核心思想是（从概念上讲）将能力强制执行从 CPU 转移到系统总线上，使其适用于软件和硬件设备。我们提出的 Skadi 操作系统依赖于我们对 Northcape（Bredi）的实现，以实现其安全保证。

在最初的发表版本中，Northcape 作为一个纯概念提出，并未实现。本工作的贡献之一是以硬件实现 Northcape。为此，必须对原始 Northcape 系统做少量修改。

本节介绍 Northcape 架构的关键概念。第 4 节将讨论在 Bredi 中实现 Northcape 的方式。我们先介绍术语，再介绍能力的实现和相关操作，最后讨论子系统调用。为便于阅读，我们将按照实际实现的形式介绍 Northcape。原始 Northcape 架构与此实现之间的差异将在第 8.1 节讨论。

A.1 术语与设计目标

Northcape 将 CPU、网络接口控制器、加速器等总线主设备统称为数据使用者，将内存控制器和内存映射 I/O 外设等总线从设备统称为数据存储区。数据使用者运行一个或多个子系统，这些子系统可以是隐式的（例如通过网络接口控制器发送和接收数据包），也可以是显式的（概念上彼此独立的 CPU 软件部分）。有一种特殊的软件子系统，即加载器子系统，在概念上属于 Northcape 硬件，因此是可信的。其他所有子系统均不可信。数据使用者也可以运行子系统，为同一设备上的其他子系统提供子系统调用。子系统调用是系统调用的泛化形式；它们支持非特权子系统之间的调用，具有相互隔离。数据存储区包含段，这些段由北桥和从设备解读的物理地址 以及以字节为粒度的长度标识。为此，北桥在动态随机存取存储器中维护一张能力元数据表 (CMT)，其条目包含这些边界及其他元数据。最后，子系统持有能力令牌，这些令牌授权它们在权限 规定的范围内访问各段。为此，能力系统维护能力令牌与 CMT 条目之间的双射 。

A.2 能力的实现

Northcape 能力有两种不同形式：*CMT* 条目和 能力令牌。从概念上讲，能力令牌是能力系统的前端：软件和设备直接使用它。为此，令牌采用与 64 位总线地址向后兼容的编码方式。与 CHERI [80], 等胖指针架构不同，令牌不包含任何元数据，例如段边界信息或访问权限。相反，能力硬件需要在后端将令牌解析为 CMT 条目。CMT 条目编码了边界信息和其他元数据。凭借所提供的能力令牌和 CMT 条目，能力硬件可以判断某次内存访问是否被允许。

请注意，据我们所知，将能力分为能力令牌和 CMT 条目的想法由 RV-CURE [43] 首创。虽然 Northcape 使用了相同的基本思路，但主要区别在于，Northcape 使用能力令牌而非 总线地址，并在 北桥 中实现令牌到 CMT 条目的解析。相比之下，RV-CURE 使用能力令牌代替 虚拟地址，并在 *CPU* 中 实现解析。本节余下部分将讨论能力令牌和 CMT 条目的表示方式，以及它们如何支持第 A.3 节所述的能力操作。

在 Northcape 中，能力令牌 c 在概念上由标识符 n、不可预测的随机数 σ 和偏移量 o组成，如图1所示。n 用于标识 CMT 中的一个入口，该入口指向数据存储中的一个段。o 用于标识每次访问时该段中的相对起始位置。 σ 用于确保令牌的不可伪造性，防止重分配后使用和重放攻击。为此，即使攻击者知道n 以及 CMT 条目的所有用户可控元数据，也必须无法猜出σ。在我们的实现中， σ从一列随机数中选取。通过使用分组密码和不可预测的密钥对单调随机数进行加密，可以高效地生成这类数列，这与计数器模式加密 [64]类似。我们采用 Qarma-64 分组密码，该密码专为将随机数截短至 16 位或更少而设计 [15]。Qarma 所用的密钥以及初始随机数均由轻量级的基于硬件的随机数生成器的输出派生 [53]。由于随机数

不会泄露给不可信设备和软件，因此在初始化后，随机数会按顺序递增。请注意，我们选择不直接依赖随机数生成器来计算 σ：σ 非常短，只有 16 位。因此，随机数生成器中的任何偏差都可能导致 σ 变得可预测。通过分别使用随机数生成器为 Qarma 密钥和随机数提供种子，我们能够实现显著更高的比特安全性。

与 RV-CURE 的另一处相似之处在于我们的编码方式：n 和 σ 存储在 64 位指针的高位中，低位则留给 o。这样既能兼容旧式 64 位指针，也支持同一段内的 [27, 43]指针运算。与 RV-CURE 的一个关键区别是，我们没有提供将指针转换为能力令牌的专用指令。系统中的所有指针都隐式地作为能力处理，所有访问均受到保护。

设计令牌表示时面临的一项挑战，是平衡任意时刻可处于活动状态的能力数量与最大能力大小。我们采用了以下折中方案：在令牌中增加两个头部位，用于标识能力令牌类型，不同类型对能力标识符和偏移量的位数分配也不同。我们提供 8、16、24 和 32 位偏移量。扣除令牌大小和消息认证码所占的位数后，标识符字段分别剩余 38、30、22 和 14 位。对于偏移量为 32 位的令牌，我们分配能力标识符 $[0, 2^{14} - 1]$；对于偏移量为 24 位的令牌，则分配标识符 $[2^{14}, 2^{22}]$，依此类推。因此，我们最多可支持 2^{38} 个能力，段大小最大可达 2^{32} 字节。

CMT 条目解析能力令牌后，能力硬件需要解析关联的 CMT 条目，以判定访问是否允许。能力令牌关联着不同的元数据属性，如图 3 所示。首先，CMT 条目中编码的能力类型对于确定允许对某个段执行哪些操作至关重要。

我们区分两种基本的能力类型：直接能力和间接能力，如图 2 所示。

直接能力直接能力拥有其指向的段，也就是说，直接能力可用于撤销引用同一物理内存段的所有间接能力。直接能力也可以进行切分，以创建新的、互不重叠的直接能力。

为在电源循环后引导系统，北桥会在复位时创建一个众所周知的根能力。这是一个直接能力，拥有一个包含整个物理地址空间 的段，并采用一种编码方式，使得对 32 位物理地址的朴素使用会被解释 为以相应偏移量访问根能力。直接能力具有以下访问权限位：读取、写入、执行、可锁定和中断可访问。读取、写入和执行权限与大多数基于分页的内存保护机制中的权限相同。可锁定权限表示能否通过相应的能力操作获得对该能力的独占访问；可以禁用此权限，以防止通过锁定造成拒绝服务。中断可访问权限位决定了能否在中断服务例程中访问某项能力。此位的主要用途是保护系统免受某些身份冒充攻击，详见 B.2 节。除此之外，它还可用于在硬件中强制将函数和数据结构标记为*IRQ* 安全。中断可访问权限仅在 CPU 设备上进行解释，并取决于读取、写入和执行权限——如果这些权限均未设置，则完全无法访问该能力。

间接能力间接能力派生自直接能力或其他间接能力，在这两种情况下都会限制访问。这种方法用于提供受限的子对象能力，类似于 CHERI 和 RV-CURE [43, 80]。

Northcape 内存管理至关重要的是，在 Northcape 系统中，直接能力仅供分配器使用。因此，在内存压力或系统崩溃时，分配器可以撤销能力，从而防止子系统窃取内存。这会使 CMT 中的入口失效并创建一个新入口，同时将该段初始化为零。CMT 的设计与查找实现相结合，确保此后任何使用引用已撤销直接能力的间接能力的操作都会导致总线错误。撤销操作还会用 0 字节覆写物理内存段，以确保机密信息不会泄露给分配器。撤销时覆写段的做法，加上我们即将介绍的锁定机制，使我们能够定义一个不包含分配器的可信计算基。这一策略的第二个好处是，分配器可以为大于请求大小的段分配直接能力。针对分段内存的高效分配算法通常会定义一个最小值段大小，以限制外部碎片，因此分配的段可能大于请求的大小 [84]。在 Northcape 中，分配器随后可以派生一个大小恰好符合请求的间接能力，确保应用程序无法对该段进行越界读取或写入。分配器还可以安全地将元数据存储在应用程序无法访问的段中。

间接能力便于进行子系统调用 间接能力有一种称为仅执行 能-力的特殊形式[1]。仅执行能力是 Northcape 的子系统调用 操作的核心，它支持在相同特权级下不同子系统之间进行切换，并实现

[1]仅执行能力并非通过不同类型来标识；相反，它们是通过带有额外限制的间接能力实现的，具体如 A.5 节所述

相互隔离。这与 Plessey 250 中的受保护过程调用，以及 Dennis 和 Van Horn 原始能力系统中的 enter 指令 [30, 79]是同一概念：子系统（例如网络协议栈）可以为其公共API创建仅执行能力。随后，这些仅执行能力可以提供给其他子系统（如应用软件），使其能够跳转到网络协议栈。跳转会同时改变控制流和保护域，即可寻址段的集合。因此，默认情况下，子系统及其调用方（用户子系统，或可能是另一个子系统）彼此隔离 。用户子系统和子系统之间可以使用调用约定，通过寄存器传递能力令牌，例如供网络协议栈发送或接收数据的缓冲区。这一原语结合独占内存访问，有助于设计无需运行时可信计算基的 *Skadi* 操作系统。

间接能力促进独占内存访问

间接能力的第二种特化形式是锁持有者能力，它允许独占访问创建该能力时所依据的能力。它通过第 A.3 节所述的锁定操作，从直接能力或间接能力创建而来。根据我们操作的设计，每个锁持有者能力都可以递归解析为一个直接能力，我们称之为该锁持有者能力的基础能力。在锁持有者能力的生命周期内，尝试解析基础能力的任何其他（孙）级子能力或基础能力本身都会导致错误。因此，知晓该锁持有者能力的子系统能够独占访问基础能力，同时无法读取或写入原先锁定的能力边界之外的内容。锁持有者能力也可以通过下文概述的限制机制绑定到特定子系统；该机制与独占访问相互独立。

关键在于，Northcape 使用锁定来解决其他能力架构（如 CHERI 和 CAPSTONE）通过撤销来解决的问题：恢复对先前共享的段的独占访问 [81, 83]。通过锁定解决这一问题的优势在于，获得独占访问是可逆的。此外，锁定和撤销的实现都很简单，且无需在内存中使用 CMT 以外的数据结构。间接能力可以通过丢弃操作销毁；该操作还会减少父级的引用计数。同一操作也用于放弃独占访问。

限制机制防止猜测攻击 仅依赖随机数 σ 来实现能力不可伪造性的一个缺点是，攻击者可以猜测 某项能力的随机数，其概率为 $1/2_{16}$。请注意，该能力的标识符 n 可能是可预测的，因为它选自一个单调递增的序列。因此，Northcape 支持对能力的使用施加限制： 能力既可以绑定到某个设备 （例如 DMA 控制器），也可以绑定到在设备上运行的某个子系统 。设备由其各自的端口加以标识，这意味着无需对设备做任何改动。然而，为了将能力限制到特定的子系统，能力系统需要一种方法来确定当前正在运行哪个子系统。Bahmani 等人[16]已经探讨过这一问题：作者建议向 CPU 增加一个额外的特权架构 寄存器，该寄存器在上下文切换时由可信特权软件写入。该寄存器保存当前正在运行的子系统的子系统标识符。随后，寄存器值被复制到 AXI 系统总线的用户 位中，使相连设备能够识别当前正在请求访问该设备的子系统。我们按以下方式对这一概念进行迭代。Northcape 试图尽可能减少可信软件的数量。因此，我们不希望依靠 CPU 特权级别来保护子系统 ID 寄存器。相反，我们将子系统 ID 寄存器设为非架构的，即软件无法访问，并将其移至北桥。如第 A.3 节所述，该寄存器在子系统调用期间被隐式 更新。为此，我们为能力引入了额外限制 的概念，它由一个类型以及一个取决于该类型的 64 位载荷组成：

对于类型子系统_标识符_绑定，限制负载编码了能力专属绑定到的设备和子系统标识符。如果设置了此限制，北桥会将请求的设备（由总线端口标识）和上一次子系统调用中设置的子系统 ID，与能力中编码的限制进行比较；如果请求来自不同的子系统，则拒绝该请求。这样，即使不同的子系统猜出了能力编号和随机数，也无法访问相应的段。

对于仅执行能力，可以指示类型子系统_标识符_设置，这会指示 CPU 在子系统调用期间相应地设置子系统 ID 寄存器。CPU 上的子系统切换正是通过这种方式反映在能力系统中的。图 8 和图 9 展示了一个示例场景中的子系统 ID 切换。具有子系统_标识符_设置的能力也可以同时具有读取和执行权限。这用于有状态子系统；该能力可以嵌入代码和数据，例如全局变量或栈。在这种情况下，子系统_标识符_设置限制意味着，对于读写访问，它是子系统_标识符_绑定的。因此，只有在子系统调用操作期间切换子系统 ID 后，即在被调用的子系统内部，才能读取该段中包含的数据。由此，子系统的调用方可以执行子系统调用，但无法读取该子系统的指令和数据。

最后，还有一种设备_解释型，它能将 64 位限制主体透明地传递到 AXI 用户位，再传至目标设备，这与 CURE [16] 中的原始概念类似。它可用于实现设备特定的逐子系统限制，例如在网络接口控制器中强制执行逐子系统网络策略。

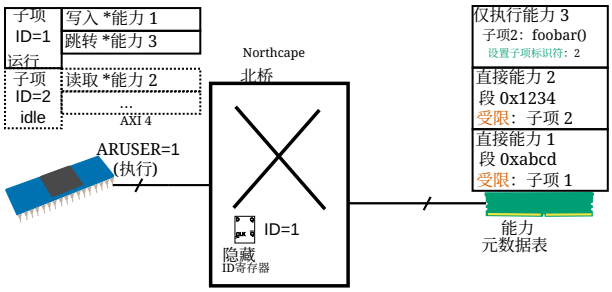

图 8：两个持有子系统受限能力的子系统示例场景：子系统 1 和子系统 2 各自持有一个私有数据能力，且子系统 2 持有一个仅执行能力，用于跳转到子系统 2。假设所有能力也都具有读取权限。子系统 1 可以读取能力 1，并通过能力 3 执行子系统调用，但不能读取能力 3 或能力 2。

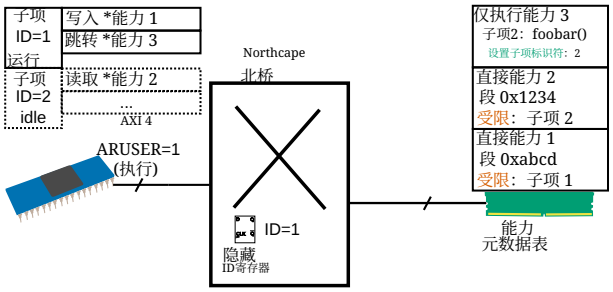

图 9：子系统 1 对子系统 2 的能力执行子系统调用后，图 8 所示的场景。现在由子系统 2 控制，它可以读取并执行能力 3，也可以读取能力 2，但不能读取能力 1。

请注意，能力限制 这一概念与锁定 操作正交：通过使用间接 能力和派生 操作，同一段可以同时供不同子系统访问。锁定 操作使其中一个子系统能够获得独占访问权。

Northcape 限制可用于与 Capstone 和 CHERI [77, 83] 中的密封能力 相同的目的：相互保护子系统调用方和被调用方的私有数据。现有的这两个系统都提供密封 操作，允许与另一个不受信任的保护域共享能力，同时阻止该不受信任的保护域使用此能力。只有预期的保护域才能使用解封 操作，从而访问关联的内存。例如，这可用于应传回给回调的 cookie 参数。通过使用 Northcape 中的子系统 ID 限制，可以将能力令牌传递给不受信任的子系统，同时阻止该子系统使用此令牌。

Northcape 中的实现具有一个优点：无需显式执行密封和解封操作，从而降低了开销。不过，Northcape 的方案也有一个缺点：不可信子系统可能会试图操纵令牌。对此可以采取措施，例如，在使用令牌之前，使用我们的 检查能力操作验证其完整性，并使用固定的能力偏移量。

A.3 操作

为保证安全性，子系统不得直接操作能力。事实上，在 Northcape 中，CMT 条目是非体系结构性的 。相反，子系统可以使用以下操作与能力系统交互：创建$(c_a,\ l,\ r,\ p) \longrightarrow c'_a,\ c_b$将一个能力$c_a$拆分为两个不重叠的部分。创建一个新的能力$c_b$，并相应地修改$c_a$的元数据。调用方可以为该段指定长度$l$，以及限制$r$和权限$p$。

p和r必须与c_a的权限和限制相同或更严格，并且l必须小于或等于c_a的长度。如果$l = l_a$，则使c_a失效。此操作只能用于直接能力，且要求c_a未锁定并且引用计数为零。

合并$(c_a,\ c_b,\ r,\ p) \longrightarrow c_m$此操作与创建相反。它从物理相邻的能力$c_a$、$c_b$创建一个新的能力$c_m$，并采用新的限制$r$和新的权限$p$。权限$p$和新的限制$r$可以比输入能力所具有的权限和限制更宽松。输入能力会被销毁。要求这些能力的引用计数为零且未被锁定。

Wilson 等人 [78] 的一项全面研究指出，在实践中，通用 slab 内存分配算法执行的操作只有拆分（在我们的模型中为创建 ）和合并段。当应用程序一次分配并归还许多小段时，合并尤其有助于减少碎片。

派生$(c_a, l, o, r, p) \longrightarrow c_i$ 派生 操作用于从输入能力 c_a创建间接能力，除内存分配器外，所有子系统和设备都专门使用这些能力。间接能力有一个作为直接能力的（祖）父级，并且可能存在重叠的兄弟能力。为此，c_a 可以是直接能力，也可以是另一种间接能力。输出能力 c_i 始终是间接能力，与其父级相比，还可以具有额外的偏移量 o 和缩短后的长度 l 。因此，派生 适用于类似 CHERI 和 RV-CURE[43, 77] 的子对象保护。

克隆$(c_a, r, p) \longrightarrow b$这是派生 的同义操作，其内部调用派生 ，偏移量 o 为 0，长度 l 等于段的长度。此操作适用于移除权限或为子系统希望共享的能力添加限制，尤其是在调用方不知道该能力段长度的情况下克隆。

锁定$(c_a,\ r,\ p) \longrightarrow c_l$尝试获得对某项能力的独占访问。输入能力$c_a$可以是直接能力，也可以是.....

是直接能力。如果可锁定权限已设置在c_a的祖父直接能力上，则会创建一个锁持有者能力c_l。可锁定权限可防止滥用锁定来实施拒绝服务。只要c_l仍然有效，就可以通过它访问c_a可见的同一段（或部分）区域。与此同时，其他与c_a重叠的能力都不能使用。

丢弃(c_a) $\longrightarrow b$ 如果能力 c_a 没有任何引用，则将其销毁，并返回操作是否成功。只能用于间接能力和锁持有者能力。如果 c_a 是由 锁定返回的锁持有者能力，则作为副作用，父级直接能力会被解锁。因此，丢弃 的调用方无需知道令牌解析为何种类型的能力。克隆和丢弃 用于对间接能力实施协作式引用计数机制，这对于共享内存段很有用。在使用撤销 销毁父级后，丢弃 还可以释放能力，以防止 CMT 条目泄漏。

撤销(c_a,r, p) $\longrightarrow c'_a$ 销毁与 c_a 关联的 CMT 条目，并为由该能力标识的段创建一个具有完整权限且移除限制的新直接能力 c'_a。将该段覆盖为 0 字节。返回新能力。只能用于直接能力。权限p 和新限制r 可以比输入能力的权限和限制更宽松。撤销 旨在让分配器在进程崩溃或内存压力下回收内存；当子系统继续使用该段的间接能力时，使用它可能导致总线错误。它只接受直接 能力，以防止子系统窃取内存。这是 Northcape 与 Capstone 的线性能力之间的主要区别：通过在撤销时覆盖该段，我们可以确保不会泄漏机密信息，而无需支持未初始化能力，从而简化实现。不幸的是，Capstone 没有讨论这些权衡，也没有提出未初始化能力的硬件实现 [83]。

Northcape 中的撤销也与针对 CHERI [81] 提出的撤销扩展有显著差异。最重要的是，Northcape 的撤销仅适用于特殊情况，而 CHERIvoke 使用撤销来保证时间内存安全，即取消共享段。在 Northcape 和 CHERIvoke 中，撤销都会带来一个问题：最初会留下孤立能力，导致 CMT（Northcape）或物理地址空间（CHERIvoke）杂乱无序。在 CHERIvoke 中，这个问题通过定期扫描整个物理地址空间以查找孤立能力来解决，但会造成显著开销。Northcape 则通过*drop*() 和 *lock*() 操作提供了更轻量的内存安全原语。*drop*() 操作会销毁用户应用程序使用的间接能力，从而防止重分配后使用和释放后使用漏洞。由于它只需修改被释放的能力及其父级（以调整其引用计数），因此开销很低。同时，引用计数机制确保不会产生孤立的间接能力（除非拥有其指向内存的直接能力被撤销）。其指向的内存被撤销）。*lock*() 操作会授予对能力的独占访问，类似于 Capstone [83] 中的线性能力。它可用于许多需要 CHERI 使用撤销的场景，例如，在某个段与其他子系统或设备共享后，重新获得对其中数据的独占访问。*lock*() 的开销略高于*derive*()。在 Northcape 中，*lock*() 总会以透明方式修改拥有该存储的直接能力——否则，任何持有与最初被锁定能力不同的间接能力的人都能绕过锁。因此，要锁定或解锁能力，Northcape 每次都需要沿着父能力树遍历到直接能力。不过，由于引用计数机制，只要不同时使用*revoke*()，*lock*() 就不会产生孤立能力。

另一方面，Northcape 中的 撤销() 会立即销毁直接能力，从而使由其派生的所有间接能力和锁持有者能力成为孤立能力。撤销() 还会覆盖数据段，因此会产生很大的开销，具体取决于段的大小。它仅用于在能力操作被错误使用的情况下重新获得内存，例如子系统崩溃之后。由于所有能力都会被递归解析，直至到达直接能力，撤销之后，孤立的间接能力无法访问相关联的段，因此它们不构成安全性风险。尽管如此，孤立能力仍会使 CMT 变得杂乱，并可能导致无法创建新的能力。针对这一问题有两种可能的解决方案。首先，可以使用丢弃() 销毁运行中的子系统仍然知晓的各类间接能力——当子系统在尝试使用这些能力时收到总线错误，便会得知这些能力已被孤立。如果没有任何运行中的子系统仍能访问可解析为孤立能力的能力令牌，则必须对 CMT 进行穷尽扫描。请注意，其开销仍低于 CHERIvoke，因为预计 CMT 远小于整个物理内存。此类穷尽扫描可由软件在 CMT 已满或接近已满时触发，而无需在每次执行之后都执行。由于预计 撤销() 的执行非常不频繁，在实际应用程序中几乎可能永远不需要执行这种扫描。

检查(c_a) $\longrightarrow b$，l，r，p 读取任意能力 c_a 的元数据。必要时递归解析 c_a，以确定基址和长度（对于无法直接编码这些信息的锁持有者能力）以及可锁定权限。此操作是从子系统调用返回或检查函数指针可接纳性所必需的，详情请参见 A.5 节。此操作的行为会根据 c_a 上是否存在子系统 ID 绑定限制而略有不同：如果没有此类限制，或者该限制与此操作的调用方匹配，则操作会返回上述所有元数据。如果 c_a 有设置子系统 ID 限制，则操作只返回R，W、X、I 权限 和限制。这足以确定通过子系统调用调用 c_a 是否安全，或者

将其用作返回地址，但不会泄露任何不必要的信息——请参见 A.5 节。在其他情况下，c_a 会返回错误代码。

限制$(c_a, r, p, o^+, l^-) \longrightarrow b$就地使能力更具限制性。这可应用于所有类型的能力，但语义略有不同：对于间接能力，可以额外指定偏移量 o^+ 和长度被减数 l^-。间接能力的边界会相应缩小。对于其他能力，这些参数将被忽略。此外，所有未同时在两者 p 和c_a的元数据中指定的权限都会被丢弃。L 权限仅在c_a 为直接能力时才会被丢弃；否则，尝试设置或清除它不会产生任何效果。r 仅在c_a 当前没有限制时才会被应用。

缩小可访问子集对网络缓冲区尤其有用：网络协议栈通常会针对每一网络层逐一移除网络头部和尾部，直到仅剩应用数据。每一层的处理完成后，无需再保留对上一层元数据的访问，因此我们可以安全地使其不可访问。这在解析网络数据包时也有助于内存安全。移除权限并添加限制对动态重定位很有用；我们将在第 5 节介绍 Skadi 操作系统时讨论这一点。

调用(c_a) 此隐式操作用于描述执行子系统调用。输入具有设置子系统 ID 限制的能力c_a 后，将非架构子系统标识符寄存器设置为指定的身份，并原子地将控制权转移到已识别段中的代码。调用实现了类似于 Plessey System 250 或 Dennis 和 Van Horn 提出的原始能力架构 [30, 79] 的子系统调用。Northcape 的一个关键区别在于，它没有用于实现调用的显式指令；相反，当设备（目前仅支持 CPU）执行某个段中的第一条指令时，只要该段带有设置子系统 ID 限制，且该限制指向的子系统 ID不同于当前正在执行的子系统 ID，就会隐式地执行子系统调用。为此，可以在软件中结合使用常规跳转指令和修改后的调用约定来触发子系统调用；我们的 Skadi 操作系统使用跳转寄存器（jr）RISC-V 指令。我们将在 A.5 节中更详细地讨论 Northcape 子系统调用。

限制 和检查 操作保证在常数上界时间限制内完成。调用 操作保证具有取决于输入能力层次结构高度的常数上界时间限制。其余操作不具有时序保证。

在 Northcape 中，北桥实现了上述定义的操作（包括调用。我们成功地实现了这些操作，无需为 CPU 添加新指令，因为调用是一种隐式操作）。它提供了一个内存映射 I/O 接口，数据使用者可以利用该接口调用上述操作，并传递参数和接收响应。因此，Northcape 提供的灵活性高于将操作编码在指令中的能力系统。

。能够识别 Northcape 的操作系统可以利用设备树或 ACPI 表等基础设施，指明当前内存映射 I/O 接口的版本，以及（可能还包括）其他限制，例如支持哪些操作。因此，必要时修改接口很容易，无需提供向后兼容性。内存映射 I/O 方案还使非 CPU 设备（上运行的固件）也能访问 Northcape 的操作。

A.4 数据结构

Northcape 完全依赖 CMT 进行簿记，无需额外的数据结构。这是相对于例如 RV-CURE [43],的一项优势，后者需要额外的数据结构来维护其等效的间接能力。CMT 是主存中由北桥维护的哈希表。它为系统中的每个活动能力包含一个入口，其中包括边界信息和其他元数据。在为 CMT 选择数据结构时采用了两项设计目标：第一，CMT 应支持常数时间查找。这对于兑现 Northcape 系统所承诺的实时保证至关重要。然而，只有调用、限制 和检查 操作保证实时完成。其余操作可能花费任意长的时间，因此不需要对*CMT* 条目插入提供时序保证。第二，CMT 在内存中应物理上连续 。由此，该表可以与 Northcape 内存管理共存：它只是占用一个在硬件 TCB（具体而言是北桥）之外不可访问的段。

CMT 实现为直接访问表，并使用线性探测来解决冲突。换句话说，CMT 是一种桶大小固定为 1 的哈希表。因此，每次令牌查找只需对一个能力进行一次内存访问。每个能力令牌中包含的标识符 n 用作哈希表的密钥。新分配的能力 ID 发生冲突时，会递增该能力 ID 并再次尝试插入（线性探测）。这会导致最坏情况下的插入时间与能力 ID 的大小呈线性关系。不过，对于我们希望能够在有界时间内执行的操作，Northcape 无需保证插入的时间约束。

这一决策有三个原因：首先，所有条目都填充到相同大小，因此我们可以使每个哈希表目录的大小与条目相匹配。其次，我们可以将整个 CMT 保存在单个段中，这使其与操作系统中的内存分配器共存变得容易得多。最后，这使我们能够通过一次内存访问完成查找。为了支持大量活跃能力并提高内存效率，可以使用可调整大小的可扩展哈希 [36]来调整 CMT 的大小。

每个入口的随机数 σ都存储在表中，以便快速比较实际随机数与所提供的随机数。

因此，北桥可以确保指向该入口的令牌在访问时有效。结合丢弃和撤销能力，这是实现时间安全的机制。

除上述属性外，间接能力的条目还包含其父级的能力令牌，也就是派生出它们的能力。在验证对间接能力的访问时，北桥递归查找其父级，直到到达基础直接能力。因此，一旦基础直接能力被撤销，对派生能力的所有使用都会导致验证错误。锁定后，也会使用相同的递归查找来强制执行独占访问。因此，无需显式扫描指向已撤销段的能力，也无需生命周期令牌；与现有的撤销方案（如 CHERIvoke [81]）相比，这是一个改进。

A.5 子系统调用

子系统调用扩展了现代 CPU 和操作系统中常见的系统调用概念。非特权软件可以使用系统调用，将控制权转交给运行在更高特权级的软件，例如操作系统内核。随后，特权软件应代表非特权软件执行某项操作，例如将数据写入硬盘上的文件。为此，非特权软件通常会遵循调用约定，使用操作所需的参数配置 CPU寄存器，并触发同步中断。中断会提升 CPU 当前运行的特权级，并将控制权转交给操作系统注册的中断处理程序。随后，操作系统可以执行该操作，并将控制权交还给非特权软件。从安全性角度来看，硬件和操作系统可确保非特权软件无法访问属于特权软件的代码和数据。然而，非特权软件的代码和数据无法免受操作系统的保护，而且非特权软件通常无法访问设备。为此，需要采用机密计算等独立解决方案[75]。

子系统调用与系统调用从根本上说具有相同的用途。不过，Northcape 等无环能力系统（例如 Plessey System 250 [79]）没有硬件特权级别。相反，子系统调用会将控制权转交给在相同特权级运行的另一个子系统，从而确保子系统之间相互隔离 。具体而言，调用方无法访问被调用方的数据结构，反之亦然，但作为参数显式传递的能力除外。这使非特权子系统能够直接调用网络设备的驱动程序等，而无需与网络协议栈交互。这对于不太符合 POSIX套接字网络语义的操作尤其有用，例如，通过精确时间协议（PTP）进行时间同步。除了确保子系统相互隔离之外，在子系统调用中传递能力还能为调用处理程序提供时间与空间安全 。在 Linux 等传统操作系统中，无法信任非特权软件传递的参数，因此必须对其进行清理，这通常需要耗费大量开销来复制缓冲区（复制_从_-*user()*，复制_到_*user()*）。

能力架构中的子系统调用并非新概念。事实上，Dennis 和 Van Horn 在关于能力系统的开创性著作中就已提出这一概念 [30]。不同的能力架构对这一概念有不同的称呼，例如，在 Plessey System 250 中，子系统调用被称为受保护过程调用。在 Northcape 的语境下，我们认为“子系统调用”一词更准确地体现了该操作的目的：向应用软件中的某个子系统（例如文件系统、网络协议栈）请求执行某项操作。

为了实现子系统调用，Northcape 北桥会跟踪每个已连接设备当前正在运行的子系统。初始时，假定运行的是可信加载器子系统 (0)。此后，每当 CPU 执行跳转指令，以能力令牌作为地址触发取指时，如果取指所用的能力具有指令取指、设置_子系统_*ID* 限制，且其子系统 ID 与当前运行的子系统不同，就表示发生了子系统调用。因此，北桥可以更新当前子系统。CPU寄存器可用于在调用方和被调用方之间传递标量参数或能力令牌；为防止信息泄露，调用方和被调用方应在调用和返回前清空寄存器。

尽管 Northcape 是无环系统，可信加载器子系统所拥有的特权却非常有限：加载器是系统中唯一被允许为自身以外的子系统创建设置_子系统_ID 能力的子系统。任何其他拥有该权力的子系统都可以通过从其自身代码创建一个设置_子系统_ID 令牌，轻易绕过子系统 ID 保护机制。然而，为外部子系统 ID 创建设置_子系统_ID 能力的权力是引导所必需的——我们要求加载器至少创建一项此类能力，以便能够开始在该子系统 ID 上执行。因此，我们将此权力限制于加载器子系统。第 5 节讨论了我们如何在操作系统设计中防止滥用此特权。为便于本次讨论，假定存在一种机制，可为子系统显式公开为子系统调用的入口点创建设置_子系统_ID 令牌。进一步假定存在一种机制，向子系统提供其获准调用的子系统调用所对应的设置_子系统_ID 令牌。最后，假定子系统可以为所有调用的返回地址生成设置_子系统_ID 令牌。我们将在第 5 节讨论这些机制如何实现。给定一个用于子系统调用的设置_子系统_ID 以及一个设置_子系统_ID 返回地址令牌，子系统现在即可调用

子系统，就像调用普通函数一样。为此，它们可以在 RISC-V 上使用诸如 *jr* 和 *ret* 之类的伪指令。然而，传递参数和返回值以及能够使用栈，需要一种特殊的调用约定，如图 7 所示。

在子系统调用期间，调用方（从调用返回后）和被调用方除了程序计数器外，都不能依赖任何寄存器。因此，Northcape操作系统（Skadi）会将子系统调用函数封装在蹦床代码中，以便将栈指针等寄存器保存到上下文能力中，并在之后恢复。子系统被调用方一获得 CPU 的控制权，就会运行被调用方版本的蹦床代码；被调用方返回后，则立即运行调用方版本的蹦床代码。上下文能力嵌入在蹦床代码本身中，因此可以通过程序计数器相对寻址加载，无需信任其他寄存器来访问它。蹦床代码还会在调用子系统以及从子系统调用返回之前清除所有寄存器，以防止信息泄漏。借助蹦床代码，子系统可以用 C 等高级语言编写，甚至可以在多次调用之间保留自定义状态。Skadi 还支持从子系统调用中访问静态变量和全局变量。

我们希望以子系统调用实现的三个细节来结束对子系统的讨论，这些细节对于为子系统调用提供安全性至关重要。

首先，在将子系统 ID 从调用方更改为被调用方之前，北桥拒绝除从段起始处以外的任何取指。由此，调用方不能简单地跳过子系统的某些部分以强制产生非预期行为，例如省略安全检查。更改子系统 ID 之后，段中的所有数据和指令都可以按照与具有子系统_标识符_绑定限制的段相同的方式读取。

其次，被调用方跳板使用检查操作来确定所提供的返回地址是否带有设置_- 子系统_标识符限制。如果未进行此检查，调用方将能够在被调用方的子系统上下文中执行任意代码。在解引用函数指针时也会进行相同的检查。

最后，每个子系统只能使用自己的子系统标识符创建设置_子系统_标识符限制。与上一段中的情况类似，如果不这样做，子系统就可能冒充任何其他子系统的身份，并绕过子系统_标识符_绑定 限制。此规则对加载器子系统有一项例外，其标识符为子系统 ID 0。加载器必须能够创建任意的设置_子系统_标识符 限制，才能启动其他子系统。因此，从理论上讲，加载器能够绕过子系统_标识符_绑定 限制并窃取子系统的上下文指针。因此，我们将加载器纳入可信计算基。加载器可以很小，完全驻留在硬件中，并且在启动子系统后即可终止。如果在启动时已知所有子系统，加载器就不必在运行时存在。如果加载器未向创建的子系统提供一个设置_子系统_标识符 为子系统 ID 0 的标识符，那么除非重置系统，否则就无法再次启动它。

B Northcape 硬件实现

本节概述 Bredi，即我们对 Northcape 的实现，这是截至 [1] 的一个理论概念。我们将首先概述 Bredi 组件以及它们如何与片上系统的其余部分进行接口。然后，我们将详细介绍我们所引入的各个 Bredi 硬件组件。最后，我们讨论 Bredi 对中断的支持，这既依赖于新的 Bredi 组件，也依赖于我们对所使用 CPU 的小幅修改。

B.1 概述

图 4 包含构成 Bredi SoC 的硬件组件：基于 cva6 [85], 的支持 Northcape 的 CPU、AXI 内存管理单元（MMU）、能力解析单元和能力操作单元。Bredi 还使用两级 Northcape 转译后备缓冲器（NTLB），以加速将能力令牌转换为总线地址：CPU 包含一个小型一级 NTLB，并针对指令和数据各复制一份。AXI 内存管理单元省略了缓存，从而减少芯片面积。能力解析器与操作模块共享一个更大的二级 NTLB。

B.1.1 AXI 内存管理单元

AXI 内存管理单元负责对 DMA 设备强制执行内存访问限制，类似于输入输出内存管理单元（英特尔）或系统内存管理单元（ARM）。它通过前端总线与数据用户（DMA 设备）对接，执行访问控制，并将能力令牌转换为物理地址，再将总线事务转发至数据存储（动态随机存取存储器或内存映射 I/O 外设）。我们目前支持标准 AXI 4 总线，包括来自 AXI 5 [10] 的原子事务。我们已使用 Xilinx [3] 的专有 AXI 交叉开关、PULP 平台 [60]的交叉开关以及赛灵思 AXI DMA [2]验证了我们的实现。

AXI 内存管理单元由两个独立的状态机实现：一个用于 AXI 接口的读通道（AR 和 R），另一个用于写通道（AW、W、B）。图10对此进行了说明。对于所有非原子请求，这些状态机可以完全独立地处理事务。AXI5 引入了原子事务，使支持原子事务的外围设备能够对数据进行原子修改，例如原子加法或原子交换操作 [10]。cva6 CPU 使用 AXI5 原子操作来实现 RISC-V 原子指令 [85]。AXI5 原子操作始终在写地址（AW）通道上触发，但根据其类型，它们也可能通过读取通道返回数据。因此，读侧和写侧状态机之间采用简单的握手协议，为读取状态机提供转发原子事务所需的信息，具体取决于

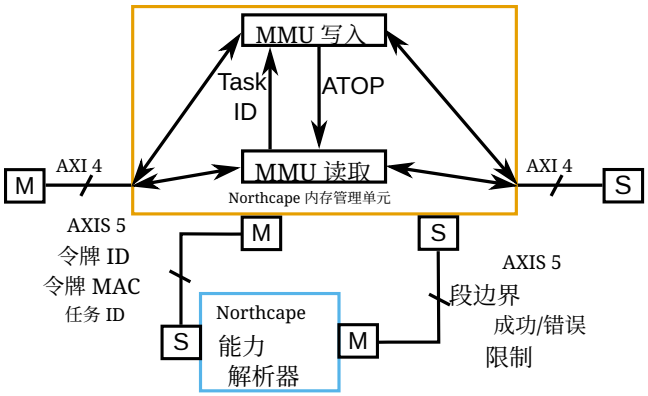

图10：AXI 内存管理单元的架构：两个独立状态机分别用于读写通道，并通过相互通信支持子系统 ID 和原子事务。

其类型，它们也可能通过读取通道返回数据。因此，读侧和写侧状态机之间采用简单的握手协议，为读取状态机提供转发原子事务所需的信息。

对于所有类型的事务，内存管理单元最初会解析通过通道地址通道传递的能力令牌。它将能力 ID 和标签，以及与之相连的子系统和设备转发给能力解析器。能力解析器会在 CMT 中执行（可能递归的）查找，并返回段的起始地址和长度，以及与该段相关联的设置子系统 *ID* 或由设备解释限制。如果标签或 ID 无效，或者设备和子系统无权访问该能力，则段的长度为零。否则，它会将输入令牌中提供的偏移量加到该段的基地址上，并判定请求的起始地址和结束地址是否位于该段的边界内。请求的起始位置和结束位置均取决于 AXI 请求的参数。例如，固定突发可能会多次读取同一个地址，而递增突发会读取单调递增的地址。回绕类型的突发也可能回绕并读取低于能力偏移量与段基地址所指定地址的地址 。发生越界请求时，内存管理单元会生成符合协议的错误响应。否则，它会通过主接口将请求转发到数据存储器。如果段的基地址和长度不是 AXI 总线宽度的整数倍，内存管理单元就需要确保相连的设备不能越界读写。为此，内存管理单元会实现完整的 AXI 地址译码逻辑，以确定传输的每个传输拍访问的物理地址，并计算掩码以防止越界访问。读取状态机将段外的数据覆盖为零，以防止信息泄露；写入状态机则对 AXI 写选通信号进行掩码处理，以防止修改段边界之外的数据。

最后，读取状态机还为 Bredi cva6 以外的设备实现子系统调用功能。为此，它会维护一个用于标识当前子系统的寄存器。该寄存器也可由写入状态机读取。设备可以通过*ARUSER* 信号指示事务是指令取指还是数据取指。因此，每当对具有设置子系统 *ID* 限制的能力执行指令取指时，当前子系统都会更新。通过比较当前子系统的新旧值，内存管理单元可以识别出子系统调用。在这种情况下，它会拒绝任何未从该段起始位置开始的请求。如 A.5 节所述，这对保障子系统调用操作的安全性至关重要。

对于每个请求，两个状态机都会将当前子系统标识符、设备标识符以及最近一次设备特定限制（如有）转发至数据存储。能力操作模块利用这些信息，对能力操作强制执行受子系统 *ID* 约束的限制。

B.1.2 Bredi CPU

我们已将简化版 AXI 内存管理单元集成到 cva6 中。Bredi 的一次性全局使能位用于在我们的 Bredi MMU 和旧版 sv39 内存管理单元之间进行切换。这使我们能够保留与 Linux 等传统操作系统的向后兼容性。与此同时，更改内存管理单元控制与状态寄存器不会对 Bredi 产生影响。Northcape 令牌的设计免去了对 cva6 进行其他修改的需要：由于令牌的行为类似于 64 位地址，cva6 在转换前对地址所作的所有假设仍然成立，无需进行大规模修改。

此外，无需为子系统调用设置专用指令。相反，当对具有设置子系统 ID 限制的能力进行取指时，系统会识别为子系统调用。因此，Skadi 可以使用标准 RISC-V 寄存器跳转指令跳转至子系统并从中返回。如果发生此类取指，Bredi 指令内存管理单元会更新其内部维护的子系统 ID，并使用更新后的子系统 ID 进行能力验证。由此，控制流和 保护域以原子方式发生改变。在修改后的 cva6 中，所有指令都会标注其取指时对应的子系统 ID。这样可以确保数据内存管理单元始终针对指令验证正确的子系统 ID；例如，即使在子系统调用取指之后才执行调用前发出的存储指令，也能保证这一点。同时，如果发生推测预测错误，cva6 可以将子系统 ID 回滚到与正确的下一条指令相关联的 ID。

当前子系统 ID 和设备特定限制也会传递给系统中的任何内存映射 I/O 设备。例如，能力操作模块依赖设备和子系统标识符来确定对所提供能力令牌执行的操作是否被允许。为此，与 Bahmani 等人 [16],类似，我们利用 AXI 协议中包含的用户自定义信号*ARUSER* 和*AWUSER*

列 [10]。

B.1.3 NTLB L1

为加快转换速度，数据和指令 cva6 内存管理单元都包含一个小型全相联一级地址转换后备缓冲器（NTLB L1）。NTLB L1 将能力令牌映射到能力解析器给出的响应，其中包括该能力的段边界、权限和限制。这些信息足以判断能力访问是否被允许。每当某项能力更新时，NTLB L1 都会使该能力对应的条目失效，因此只有已更新 能力会导致一级缓存未命中。根据我们的操作设计，除撤销、锁定 和对锁持有者令牌执行的丢弃 操作外，其他操作都只能修改层次结构中级别最高的能力。因此，执行此类操作后，只需使 NTLB L1 中的特定条目失效；随后 cva6 内存管理单元会在能力解析器中重新查找并更新其缓存。如果执行撤销、锁定 或丢弃操作，则会使整个 NTLB L1 失效以确保安全性：如果不这样做，攻击者就可能利用过期缓存条目绕过独占访问和撤销机制。清空缓存后，所有能力令牌在 NTLB L1 中都会未命中，并由能力解析器处理。由于在这种情况下出于相同原因也会使整个 NTLB L2 失效，因此它会执行全新的 CMT 查找，从而导致访问故障。

B.1.4 能力解析单元

AXI 和 cva6 内存管理单元通过 AXI-Stream 总线 [11] 与 Bredi 能力解析器单元对接。内存管理单元向解析器提供所给令牌中的能力 ID 和标签，以及其设备和子系统。解析器随后对标识符执行查找，判断该能力是否有效，并将段的边界以及设置子系统标识符或设备解释的限制返回给内存管理单元。内存管理单元最终决定特定访问是否被允许——它需要考虑能力令牌中指定的偏移量，并且对于 AXI 内存管理单元，还需要考虑突发类型（固定、递增或回绕）、大小和长度 [10]。内存管理单元还会识别访问是否与 CMT 重叠并拒绝该访问，从而防止通过修改能力元数据实现权限提升。

能力解析器由所有内存管理单元共享。这便于实现面积与性能权衡：由于我们的 L2 TLB 目前基于寄存器构建，所有内存管理单元理论上可以同时访问它，仅在缺失单元处进行仲裁。在缓存命中率较高的情况下，这将缩短访问时间，但代价是把能力解析逻辑大量复制。为降低这一面积开销，我们对来自各内存管理单元的请求进行仲裁，并使用单一的能力解析器。在假定 L1 转译后备缓冲器命中率较高、且 AXI 的活动程度适中内存管理单元与 CPU 相比，这在我们设想的全部使用场景中均是如此，我们认为这一权衡是最优的。

如图11所示，解析器采用三级流水线实现。第一级流水线将输入捕获到寄存器中。这是打断关键时序路径所必需的，因为我们的轻量级仲裁器只会将授权信号捕获到寄存器中，而不会捕获输入。第二级流水线负责执行实际的内存查找。为此，它使用与二级 TLB 的自定义请求/响应接口，提供能力 ID 和标签，并接收检索到的 CMT 条目和元数据，如第 4 节所述。最后一级流水线接收上一阶段传来的 MMU 提供的元数据和 CMT 条目，并验证是否允许此次访问。还可以选择启用一个输出流水线寄存器阶段（图中未示出）。

为了能够递归地解析能力，解析器阶段会循环返回查找阶段，并发出递归请求。递归请求会填充与 MMU 请求相同的数据结构。因此，该数据结构会存储原始请求中的元数据，以便构造最终响应，例如层次结构中最高层能力的边界。MMU 会将这些字段留空，综合编译器会相应地将不需要的字段优化掉。回环请求会缓冲在寄存器中，以防止出现组合环路，并且始终优先于 MMU 请求。因此，我们每个周期都可以发出一个递归请求，从而在无需复杂状态机或流水线阻塞器的情况下，以 L2 TLB 所允许的最快速度完成递归解析。

正如我们在 B.1.6 节中详细说明的那样，发生 L2 TLB 命中时可以跳过递归能力解析。此规则的一个例外是锁持有者能力，由于大小限制，这类能力不包含段边界。如果递归中的第一个 CMT 条目是锁持有者，我们就必须始终开始递归，并在第一次递归调用中从 CMT 条目中获取边界。

经过可选的输出寄存器级后，解复用器会将解析器响应转发给相应的内存管理单元。为了尽可能减小该组件的芯片面积，我们使用*TDEST* AXI-Stream 控制信号来指示响应所属的端口。

B.1.5 能力操作模块

能力操作模块在硬件中实现了 A.3 节介绍的操作。为此，它通过 AXI4-Lite总线为软件和硬件使用者提供有文档说明的内存映射 I/O 接口，并通过有限状态机实现这些操作。图12概述了操作模块的架构。使用者在 MMIO输入寄存器中指定输入令牌，将限制和段长度等其余操作参数写入其余输入寄存器，并通过将操作码写入相应的寄存器来启动操作，例如

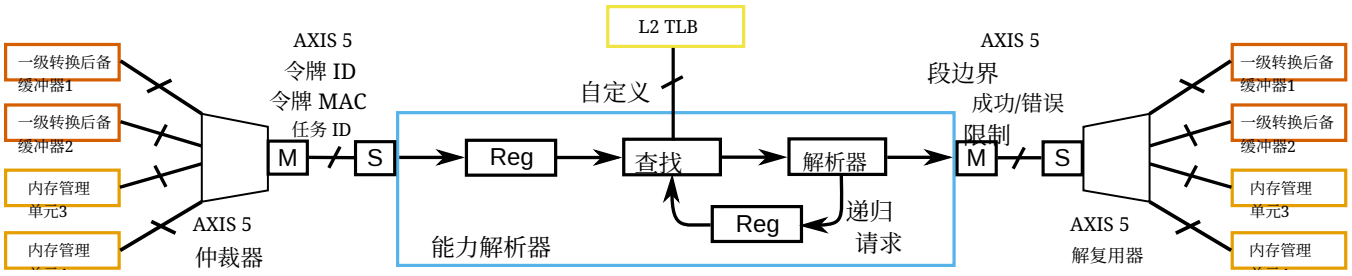

图11：Bredi 能力解析器的架构：包含 ID 哈希、CMT 查找和 CMT 条目解析阶段的三级流水线。

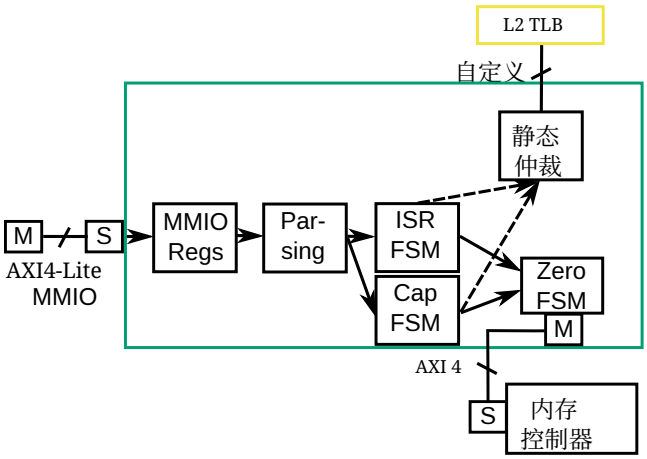

图12：Bredi 能力操作模块的架构：有限状态机执行各项操作，而内存映射 I/O 寄存器负责为其提供输入，并将输出返回给调用设备。

此类参数。所有输入寄存器均为只写，以防止输入令牌泄露给其他子系统和设备。设备可以轮询操作寄存器，检查操作是否完成，也可以等待中断。随后，可以从相应寄存器读取操作输出令牌（如果指定的操作会产生令牌）。操作发起方的设备和子系统（由内存管理单元提供）保存在操作模块的寄存器中；只有当读取请求来自同一子系统时，操作模块才会返回输出令牌。否则，将返回零响应。事务创建者读取输出寄存器后，中断便会清除，此时可以发起下一项操作。

操作模块采用分层状态机实现，包含三个子状态机：一个状态机在撤销时将 CMT 和主存中的段清零；一个状态机用于为中断服务例程实现检查；最后一个状态机实现其余操作——操作。操作状态机还封装了 Qarma-64 分组密码，用于计算随机数序列，并从中选取能力标签 σ。Qarma 密钥和初始随机数取自真随机数发生器的输出，该发生器也包含在操作模块中。操作状态机还复制了能力解析器中已有的能力查找逻辑，因此不会干扰解析器中对时间要求严格的查找。这也使操作模块能够以较低的时钟频率进行综合，并针对芯片面积进行优化。

操作状态机部分采用了重复设计：一个副本实现所有 Bredi 操作，供中断之外使用，并由一份内存映射 I/O 寄存器组副本控制。第二个副本仅实现检查操作，供中断期间的子系统调用使用。这样，当非中断子系统正在与操作模块交互时，我们仍能接收中断；这对于系统的实时保证至关重要：除了检查之外，Bredi 操作无需实时完成。如果使用操作模块执行操作时必须禁用中断，那么在下一次中断能够被接收之前，就会出现不确定的延迟。如果未启用中断，中断服务例程将无法执行子系统调用；若执行此类调用，就会改变操作模块的内部状态（例如输入能力令牌），却无法将其恢复到接收中断之前的状态。操作模块还拥有与 L2 TLB 的独立接口（见第 4 节），并使用静态仲裁器，确保在访问 L2 TLB 时，中断服务程序有限状态机始终具有优先权。

我们还支持不可屏蔽中断执行 Bredi 操作，此时被中断子系统可能正在与操作模块交互。为此，中断服务例程可以使用中断服务程序内存映射I/O寄存器来启动操作，就像使用非中断服务例程寄存器组一样。此类操作在 非中断服务程序有限状态机处于空闲状态时启动，并由非中断服务程序有限状态机执行。额外的输出值寄存器存储由被中断子系统指定、但尚未被其读取的输入和输出，从而防止完整性或机密性遭到破坏

由中断服务例程读取。

能力操作模块通过其寄存器接口提供对真随机数生成器的访问。为此，在播种 Qarma 密钥后，操作模块会持续请求真随机数生成器生成由 64 位随机比特组成的序列。我们使用一个状态机封装真随机数生成器，并由它将输出收集到一个 64 位移位寄存器中。操作模块通过一个内存映射 I/O 寄存器提供每个 64 位序列。寄存器一经读取，该序列便会立即 丢弃。因此，后续由软件或硬件触发的读取将返回一个新的随机序列，从而确保序列的隐私性。对于依赖秘密随机数的硬件设备和软件而言，这项功能都很有用：它们可以从能力操作模块获取随机数，而无需信任单独的设备或驱动程序。

假设两个设备（例如，两个 CPU 内核，或一个 CPU 和一个加速器）想要同时通过操作模块执行能力操作。由于操作可能包含多次读取和写入，因此引出了如何确保操作原子性 的问题：两个设备可能会同时尝试发起不同的操作，从而可能导致某个操作开始执行，而这并非任一设备所期望的。

我们对内存映射 I/O 接口进行锁定 ，以确保设备之间互斥：由于访问总线上的仲裁，在任意给定周期内，同一时间只有一个设备能够写入寄存器接口。第一个写入寄存器文件的子系统会获得操作模块的锁定 。只要锁定仍然有效，其他设备甚至同一设备上的其他子系统尝试读取寄存器文件时都会收到错误，所有写入也都会被忽略。只有持有锁定的设备和子系统才能访问寄存器接口。读取事务结果后，锁定即会释放，同时输出也会在同一周期内清除。

目前，我们接受利用此锁定接口对操作模块发起拒绝服务攻击所带来的风险。如果对此有所顾虑，可以在寄存器接口中添加超时计数器 ：如果事务未能在超时时间内完成，操作模块就会清除所有输入和输出，并解除锁定。

在 Northcape 系统中，根能力并未受到特殊处理——每次系统重置后，都会隐式创建相应条目；该条目由解析器在 CMT 中解析，并可用作操作的输入。唯一的区别是，标识符和标签被硬编码为 0，而非通过计算得出。

操作模块通过位图 跟踪已占用和空闲的 CMT 槽位，并在操作成功提交时更新该位图。位图存储在大小可配置的 SRAM 存储器原语中，可在可配置的行大小下实现与技术相关的性能和面积权衡。为此，我们利用了 Northcape 相较于 CHERI [80] 等方案的一项设计优势：能力仅限于 CMT，而 CMT 只占系统动态随机存取存储器的一小部分（在我们的实现中为 2^{13} 个条目）。这大幅限制了位图的大小，因此可以将整个位图 存储在 SRAM 中，而无需将其缓存在缓存中。

能力 ID 的分配使用位图：能力操作有限状态机逐行从静态随机存取存储器中请求，直到找到一行包含未占用条目。我们使用并行前导零计数实现来选择每行中的第一个空闲槽位。因此，假设静态随机存取存储器有 r 行，我们可以在 $\leq r$ 个周期内找到一个空闲槽位，或在 r 个周期内可靠地检测出 CMT 已完全占满。因此，Northcape 在我们的平台上的操作可在可预测的时间内完成，其最坏情况执行时间仅取决于加载输入 CMT 条目和写入输出 CMT 条目的存储器访问延迟；对于同一种操作，这些条目的数量始终相同！

请注意，由于 Northcape 能力令牌中的偏移量大小 不同，像 32 位和 24 位偏移量能力这样的大型能力只能占用最低的 2^6 和 2^14 能力 ID。偏移量较小的能力则不受此限制。因此，我们尽量避免为小型能力分配标识符较低的能力槽位。为此，对于每种能力大小，我们不会从零开始迭代，而是从下一个更大能力类型的最低能力 ID 加一处开始。我们还记录每种偏移量类型中包含未分配 *CMT* 槽位的最后一行，从而避免在下一次操作中再次遍历已占满的行。

B.1.6 NTLB L2

最后，操作模块与解析器共享一个全相联直写 NTLB L2。与 NTLB L1 不同，NTLB L2 将能力令牌映射到完整 *CMT* 条目。NTLB L2 还实现了完整的共享能力缺失单元和回写单元。具有静态优先级、始终优先处理解析器的仲裁器在并行读取缓存的同时对缺失单元进行多路复用。

L2 TLB 的架构如图13所示。L2 TLB 向能力解析器和操作模块提供自定义请求-响应接口，并通过 AXI 4 接口访问主存中的 CMT。其核心采用可参数化缓存：利用综合参数可设置算法和大小。我们实现了使用能力标签标识缓存行的直接映射缓存，以及 n 路相联缓存和全相联缓存，从而提供用户可配置的芯片面积-性能权衡。为此，内存实现也有所不同：全相联缓存需要使用独立的寄存器，而 n 路相联缓存和直接映射缓存则围绕更节省空间的双端口 *SRAM* 原语构建。

L2 TLB 有两个并发访问端口，一个供能力解析器使用，另一个供能力操作模块使用。因此，采用真正的双端口静态随机存取存储器或基于寄存器的实现。发生缓存未命中时，可通过缺失单元检索能力。为此，我们使用带有静态仲裁的仲裁器，确保能力解

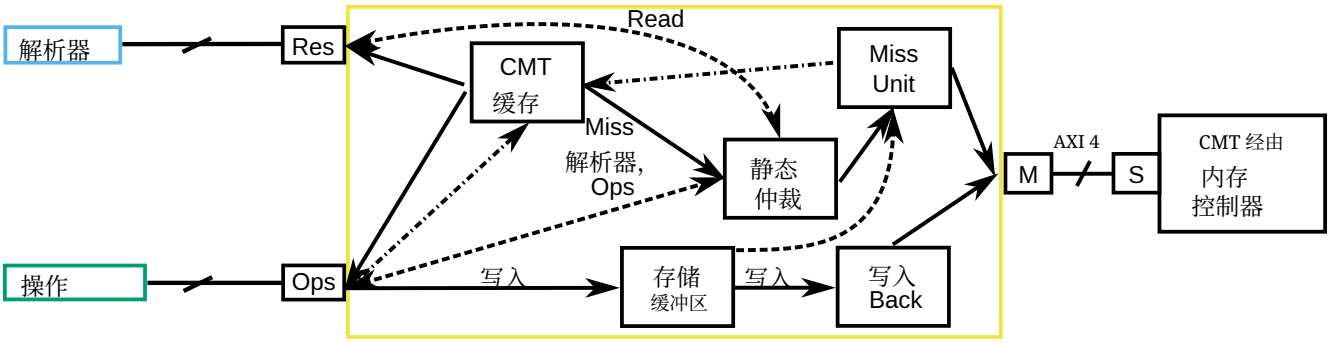

图13：Bredi 二级 TLB 的架构：由缺失单元和回写单元封装的参数化缓存。

析器始终拥有优先权。这一点十分必要，因为让解析器停顿可能导致多个内存管理单元同时停顿。如果检索到的 CMT 条目具有相应的缓存能力权限（可缓存_*TLB*），且请求来自解析器，则会将已解析的未命中写入缓存。这样做是为了使能力解析器能够跳过分层解析，具体说明如下。

虽然能力解析器端口不会发出写入操作，但能力操作端口会。为此，内存写回单元连接到操作模块端口。如果操作模块希望将能力写入内存，写回单元会立即启动（直写）。在内存控制器确认写入的周期中，写入的 CMT 条目也会提供给缓存。如果该条目可缓存，就将其写入缓存；否则，为确保缓存中没有过期数据，如果相应缓存行中包含此能力的旧 CMT 条目，就使该缓存行失效。

缺失单元与回写单元之间可能发生读写冲突：二者可能同时尝试访问同一个能力。在这种情况下，如果仍有可能，我们会中止缺失单元请求。如果无法中止，我们会缓冲待写入的 CMT 条目，并在更新缓存和生成响应时，使用该条目代替能力缺失单元的响应。

此外，缓存采用直写方式，因此回写单元仅供操作模块使用。事实上，操作模块执行的所有写入都会在当前操作开始之前，同时提交到缓存和内存。与回写缓存相比，这种设计主要是为了减少芯片面积。由于在我们的系统中，能力通常会长期存在，我们认为回写架构带来的性能提升很有限。我们还优化了 Xilinx 交叉开关和内存控制器的配置选项——每个 CMT 条目占 256 位，这也是内存控制器的最大总线宽度。因此，每个 CMT 条目都能在单个周期内完成传输和寄存，从而降低缓存的延迟和复杂度。

我们的 NTLB L2 设计允许能力解析器在最先请求的能力遇到缓存命中时，跳过对整个能力层级的解析。该方案借助以下不变式保证安全：*若某能力在 NTLB L2 中命中，则其父级直至最终祖父级均已由解析器解析，且此后未被更新。*该不变式由 NTLB L2、解析器与操作模块协同维持：首先，NTLB L2 向解析器指示某次访问是命中还是未命中。若命中，解析器可跳过对（层次结构剩余部分的）解析，并立即解析该能力。否则，它需要执行完整的递归能力验证。在为解析器进行解析的过程中，每遇到一次未命中，缺失单元都会将检索到的 CMT 条目写入缓存。这会使新缓存条目的元数据位推测性 被设置。

解析器完成递归解析后，若解析成功，便可关闭推测窗口，使缓存中的推测位被丢弃。解析器流水线确保在处理来自另一个内存管理单元的下一个解析请求之前，始终完成递归解析。因此，所有标记为推测的 CMT 条目都必须属于当前有效的分层解析，并且可以移除推测位而不违反不变式。若递归解析出错，解析器会向 NTLB L2 发出刷新信号，使所有推测条目失效。

操作模块使用 NTLB L2 进行能力查找。不过，为了保持不变式，操作模块发生缓存未命中时，不会将数据写入缓存。由于这是直写缓存，所有更新后的能力都会同时写入缓存和内存。除撤销、锁定 和 丢弃 之外的操作只能修改层次结构中的顶层能力，而解析器总会检查该能力。因此，对于大多数操作，无需额外处理即可保持不变式。最后，与 NTLB L1 一样，对于撤销、锁定 和丢弃，整个 NTLB L2 都会失效。

将能力写入提交到内存，在我们的平台上可能需要几十到几百个周期。因此，等待能力提交到内存会拖慢操作模块。为此，我们采用存储缓冲区：一个同步的先进先出（FIFO）内存，用于缓冲操作模块即将写入的能力。

只要缓冲区有可用空间，操作模块就可以继续运行，无需等待写入完全提交。回写单元 会通过实际将写入提交到内存来清空存储缓冲区。

存储缓冲区会与读取后写入 缺失单元产生冲突：如果某个能力先被写入缓存和存储缓冲区，随后从缓存中逐出，再通过缺失单元重新加载，最后通过回写单元提交到内存，那么缓存中将保留该能力的过时值。我们采用以下方式解决这一冲突：使用由寄存器实现的环形缓冲区 跟踪存储缓冲区中各能力的能力标识符。通过将缺失单元要加载的能力标识符与存储缓冲区中所有有效的能力标识符进行比较，检测读取后写入冲突。如果匹配，我们会暂停缺失单元 ，直到写入提交到内存。经验分析表明，这种冲突极少发生，不会对系统造成可测量的减速。在不存在冲突的情况下，缺失单元和回写单元都可以利用并发写端口在同一个时钟周期 内写入一个能力。因此，无需在这些单元之间进行仲裁，并且在不存在冲突的情况下，缺失单元的写入无需等待操作模块。

B.1.7 性能计数器

我们为 Northcape NTLB 一级和二级未命中添加了性能计数器。它们集成于 RISC-V 的运行时可配置性能事件框架中，所有软件组件均可读取这些计数器。性能计数器旨在优化对性能要求较高的软件，并且可以通过 *mcountinhibit* 控制寄存器禁用。

B.2 Bredi 中断架构

迄今为止，我们所述的子系统调用支持能够将操作系统划分为任意多个彼此隔离的小型子系统。然而，该机制假设子系统之间借助自愿 上下文切换和子系统调用进行切换。由于中断是异步发生的，因此这种方法无法支持中断。本节将讨论我们如何解决支持不可信中断服务例程所面临的以下挑战：

- 隔离 中断服务例程，使其与子系统及彼此相互隔离
- 让中断服务例程安全地返回 到被中断的子系统
- 防止在子系统和中断服务例程之间共享寄存器
- 采用不可信调度器的抢占式调度
- 中断处理程序的无条件可用性

B.2.1 中断服务例程的隔离

我们为 cva6 添加了一种非标准的中断向量化方案，如图 5 所示。该向量化方法与 RISC-V 快速中断（FastIRQ）的相关工作类似 [63]。然而，由于快速中断规范尚未获得批准，也未在 cva6 中实现，我们采用了一种更简单的自定义方案。

与 FastIRQ 方案完全相同，我们的向量表存储每个中断对应的中断服务例程地址。发生中断时，CPU 从地址 8·机器陷阱原因 + *mtvec* 中读取一个 64 位的处理程序令牌，然后跳转到处理程序。

我们还会在首次启用新的向量中断模式后，将向量表寄存器*mtvec* 设置为只读。这样，可信加载器子系统可以在加载时配置该寄存器，而其余不可信软件子系统无法更改其值。

向量表条目包含具有设置子系统标识符限制的能力令牌。因此，接受中断会导致对中断服务例程发起子系统调用，从而确保中断服务例程与子系统之间以及不同中断服务例程之间的隔离。Skadi 提供了一个中断服务例程子系统，该子系统使用我们的中断向量化方案永久注册以处理设备中断。请注意，在 cva6 中，所有设备共享一条 CPU 中断线——平台级中断控制器（PLIC）跟踪实际触发中断的设备，执行 IRQ 优先级管理，并将设备中断转发至 CPU。因此，在 Skadi 中，中断服务例程子系统始终通过子系统调用调用 PLIC 驱动程序。PLIC 驱动程序确定触发中断的设备，并对已注册的驱动程序子系统发起子系统调用。

B.2.2 从中断服务例程安全返回

在我们的中断请求模式下，中断服务例程必须使用中断返回指令 *mret* 终止执行，该指令会使 CPU 从发生中断的那条指令处继续执行。在完成这一步之前，中断处理程序无法返回被中断子系统，只能调用已明确开放的子系统调用 。除了重置程序计数器之外，从中断服务例程子系统返回时，我们还需要重置 Bredi MMU 中的子系统 ID。由于被中断指令地址 *mepc* 通常并非位于被中断子系统代码段的起始位置，因此我们无法依靠标准子系统调用语义返回被中断子系统。为解决这一问题，cva6 MMU 会区分中断状态下和正常（即非中断）状态下的当前子系统 ID。为此，它包含两个不同的子系统 ID 寄存器。CPU 在发生中断时进入中断状态，并在执行异常返回指令 *mret* 时退出该状态。因此，*mret* 会隐式地将活动子系统 ID 切换回被中断子系统的 ID。

B.2.3 防止共享寄存器

在标准 RISC-V 中断向量化方案中，中断服务例程能够访问 CPU 寄存器，从而便于在中断服务例程中进行重新调度，但代价是寄存器内容可能泄露。因此，我们在 cva6 中实现了第二组寄存器（寄存器堆叠）：每当处理设备中断时，系统就会切换到这第二组寄存器。为防止中断服务例程修改 CPU 状态，我们还对配置与状态寄存器进行了修改：*mstatus*、*mie* 和 *mepc* 均拆分为中断服务例程副本和非中断服务例程副本，使用哪一个副本由当前执行状态决定。在执行状态发生变化之前，软件无法访问相应的第二个副本。这样，被中断子系统就始终能够以相同的状态（例如浮点单元已启用）并从正确的指令处继续执行。

B.2.4 抢占式调度

重复的寄存器文件引出了一个问题：我们如何支持当前正在执行的子系统被抢占：由于 ISR 无法访问子系统的寄存器，因此无法重新调度。我们利用中断向量化方案来解决这个问题：为每个子系统提供一个可写、非 IRQ 可访问的能力令牌，该令牌指向向量表中与定时器中断关联的能力令牌。这样，每个子系统都可以在其代码段内 注册一个函数；当定时器中断被接受时，该函数会被调用，并执行一次子系统调用以进入调度器。随后，调度器可以确定是否需要抢占该子系统，并选择另一个可运行子系统。调度器可以对作为返回地址提供的能力令牌执行子系统调用，从而切换上下文至任一子系统。使用我们的非标准向量化模式接收定时器中断是寄存器压栈的一个例外，并且不会导致 *CPU* 转入 *IRQ* 模式。因此，调度器照常可以访问非 IRQ 寄存器组。显然，这种抢占方案是自愿的，因为它依赖当前子系统配合调度器。

B.2.5 中断服务的可用性

第 3 节中的安全模型旨在确保即使存在能够禁用 CPU 中断的攻击者，中断处理程序仍能执行。为了实现这一安全目标，同时不损害机密性和完整性，我们依赖不可屏蔽中断（*NMIs*）

，即软件无法禁用的中断。

由于 cva6 不支持不可屏蔽中断（NMI），我们实现了一种自定义方案：软件可以使用自定义控制状态寄存器将普通中断标记为不可屏蔽。与我们自定义的中断向量化模式类似，此位一旦设置便无法取消，从而确保不可屏蔽属性无法移除。启用此位后，我们修改过的 cva6 会将不可屏蔽

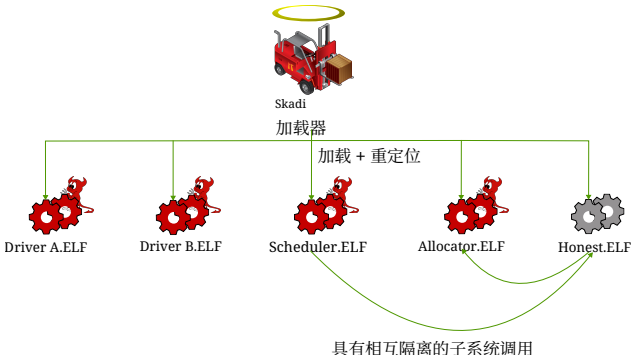

图14：Skadi 中操作系统组件的分解。

中断无视中断使能标志而触发，除非当前正在执行不可屏蔽中断。

C Skadi 实时操作系统实现

本节讨论 Skadi 实时操作系统的架构。Skadi 利用 Bredi 架构，尽可能加强操作系统的隔离，并消除运行时可信计算基。为此，它不依赖任何可信软件来保证子系统数据的机密性和完整性。只要参与中断处理的子系统未遭到破坏，它也能够满足实时期限。

本节首先讨论 Skadi 的整体架构。接着，我们将具体讨论 Skadi 加载器（Skadi 中唯一的可信组件），以及子系统的运行时重定位。之后，我们将重点介绍如何实现机密性、完整性和实时兼容性这些安全保证。最后，我们将简要讨论如何将现有实时操作系统子系统移植到 Skadi。

C.1 概述

Skadi 基于 Zephyr。Zephyr 是 Linux 基金会支持的一款实时操作系统。与 RIOT 和 Contiki-NG 等其他实时操作系统相比，Silva 等人将 Zephyr 描述为一种纳内核 [68]。Zephyr 提供一个最小化的操作系统内核，实现了调度、内存管理和特定于架构的配置等关键系统功能。此外，构建时还可选择启用多种子系统和设备驱动程序，包括完整的网络协议栈、文件系统支持，以及适用于多种片上系统系列和设备的设备驱动程序。Zephyr 还提供 llext，这是一种用于在运行时重定位 ELF 二进制文件的机制，旨在加载扩展。在最基本的配置中，所有操作系统组件以及用户子系统都能完全访问所有内存

并可使用特权 CPU 指令。如果所选片上系统支持，用户空间子系统还可选择通过内存保护单元（MPU）与操作系统隔离。为此，Zephyr 为用户空间子系统提供了系统调用。

图14展示了 Skadi 的结构设计。Skadi 利用了 Zephyr 的高度模块化设计。该实时操作系统沿用了 Zephyr 对内核、驱动程序和子系统的划分，并为这些组件引入了运行时隔离。我们想要强调的是，操作系统在运行时存在的所有组件都受到这种隔离机制的保护，无一例外。例如，中断服务例程（ISR）被封装到三个不同的子系统中：一个用于处理来自设备的中断，一个用于处理异常，另一个用于处理定时器中断。我们已在 B.2 节说明了这种方法的用途。与 Skadi 中的所有子系统一样，中断服务例程子系统无需可信，即可确保系统子系统的机密性和完整性；而要实现实时执行，只需信任两个中断服务例程子系统中的一个（设备中断子系统）。动态内存分配、设备管理和网络等其他所有操作系统功能，以及所有应用软件，也同样包含在彼此隔离的子系统中。

启动 Skadi 时，可信加载器组件使用 Northcape 创建 和派生 操作为操作系统组件分配缓冲区。随后，它重定位 每个操作系统组件和应用程序到各自的缓冲区中，并使用 Northcape 限制系统，将这些缓冲区的使用限制 在相关子系统内。从概念上讲，加载器最初拥有根能力；为此，它使用子系统 ID 保护，防止子系统读取或写入根能力。对于每个已分配的缓冲区，加载器都会放弃 对一部分内存的所有权，直到所有子系统都加载完毕。加载最后一个子系统后，加载器使用最后一次创建操作销毁根能力，销毁内存内容，并将释放的内存交给分配器，使其成为通用堆的一部分。

运行时，Skadi 使用 Northcape 子系统调用，在相互隔离的组件之间传递控制权。这模拟了传统微内核设计中的消息传递 [44]，但安全控制权传输依赖的是 Northcape 硬件支持，而非特权软件。为此，可以使用标准 ELF 符号标示导入和导出的子系统调用。Skadi 加载器负责解析来自其他组件的导入子系统调用并对其进行重定位。此外，加载器还会根据导出的符号，从代码段中派生出导出子系统调用的能力。

C.2 Skadi 子系统结构

Skadi 组件是 ELF 二进制文件 Skadi 按照可执行与可链接格式（ELF）标准 [71]对代码和数据进行重定位。为此，每个操作系统组件都被编译为单独的 ELF 二进制文件。子系统二进制文件作为加载器二进制文件的一部分提供，并使用 *LZ4* 压缩 [28]以减少磁盘空间占用。随后，加载器会在运行时对这些二进制文件执行标准 ELF 重定位。为此，加载器主要可以利用 Zephyr 现有 llext 扩展机制中的 ELF 加载和重定位实现。加载器的主要创新在于，它包含一个简单的可信分配器，使用 Northcape 的 创建 操作为 ELF 段分配独立的缓冲区，并将其作为重定位目标提供给 llext。加载器还使用 Northcape 限制机制，确保代码段和数据段只能由其所属的子系统访问。与现有的 llext 基础设施类似，加载器还会为子系统分配一个栈，并通过子系统调用进入该子系统专用的初始化函数；这些函数由一个特殊的 ELF 区域标示。

这种方法的主要优势在于，它能够以安全可靠的方式集成不可信的操作系统组件。例如，设备供应商可能只提供闭源二进制文件形式的设备驱动程序。同样，传统的用户空间组件也可以从供应商处以二进制形式购得。系统集成商可以利用 Skadi 的子系统分解方式，将此类不可信的闭源二进制文件与自己的专有驱动程序和设备组合起来，并依靠子系统隔离来确保其代码的完整性和机密性。结合 Bredi，还能确保不可信驱动程序无法将设备用作混淆代理来规避这些保护——这是相较于 CHERI [35] 等现有能力架构的一项关键优势。图 6 展示了 Skadi 所包含的子系统以及加载器架构。

导出和导入子系统调用同样利用现有的 ELF 符号语义来提高向后兼容性：子系统通过定义一个名称与子系统调用相同、类型为未定义的符号来导入 子系统调用。子系统通过定义一个名称与子系统调用相同、且位于一个特殊的已知区域中的符号来导出 子系统调用。这可以由任何符合 ELF 的链接器自动处理，无需工具链修改。此外，导入和导出的符号可用于在构建时确定子系统的加载顺序。为此，一个简单的脚本会建立子系统之间的“从何处导入符号”依赖关系，并计算子系统的拓扑排序，随后将其用作初始化序列。

为了解决子系统之间的依赖循环，Skadi 加载器提供了子系统调用 API，以动态解析符号。此 API 通过

加载器_代理 子系统进行代理——这缩小了加载器的攻击面，并可在加载器销毁后仍调用该 API 时停止执行。

旧版编译器可以生成与 Northcape 能力令牌兼容的代码 为了在 ELF 符号重定位期间支持任意能力令牌，Skadi 利用了 RISC-V ELF规范 [25] 中规定的大型代码模型。大型代码模型的目的是使 64 位 RISC-V 处理器上的应用程序能够将整个 64 位（虚拟）地址空间用于代码。中等任意（medany）和中等低位（medlow）代码模型依赖于分别带有 20 位和 12 位立即数的*U* 型和*I* 型RISC-V 指令序列[2]，使它们能够指示任意 32 位地址（medlow），或相对于程序计数器的任意 32 位偏移量（medany）。标准 ELF 重定位用于向 ELF 加载器指示在何处找到这些指令对，以及如何计算立即数。虽然 medany 和 medlow 非常适合 32 位程序，但对于 64 位程序，它们会对代码的加载位置施加限制。因此，增加了大型代码模型。在大型代码模型中，*C* 编译器会紧接在函数序言之前分配一张表，其中为该函数使用的每个外部符号各设一个 64 位入口。然后，它使用 U 型/I 型指令对来访问每个函数的符号表，类似于 medany 代码模型，并从该表中检索每个外部符号的实际值。虽然该机制旨在用于 64 位虚拟地址，但代码对外部符号在表中的位置不作任何假设。由于在 Northcape 系统中，能力令牌的宽度为 64 位，Skadi 加载器可以直接在符号表中指定 Northcape 令牌。这些令牌对应于派生能力，允许导入子系统对导出的被调用方跳板执行子系统调用，或访问大小受限的全局数据。因此，子系统可以通过导入相应的 ELF 符号，从其他子系统导入并使用代码和数据的符号。由此，Northcape 无需修改编译器和链接器，这相对于依赖编译器支持 [43, 80] 的 RV-CURE 和 CHERI 是一个显著优势。Skadi 利用这一点，既为导入的子系统调用提供令牌，也为同一子系统中的函数和数据提供令牌。至关重要的是，这使我们能够将代码、数据、BSS 和只读数据重定位到仅具有所需访问权限的不同段，而不必将它们包含在一个同时具有读取、写入和执行权限的缓冲区中。

通过 ELF 未定义符号请求访问内存映射 I/OELF 符号导入机制也用于向子系统提供对内存映射 I/O 寄存器的访问。我们将此功能与 Zephyr 现有的构建时设备树解析（通过宏实现）集成在一起——宏不再将设备寄存器解析为标量物理地址，而是像*DT_INST_REG_ADDR*这样的宏会解析为某个外部函数的地址。该函数的名称编码了所请求内存映射 I/O 区域的边界。例如，在我们测试平台的操作模块设备树条目中，*DT_INST_REG_ADDR* 会解析为函数名*skadi mmio 0 1107296256 4096*。__ _ _ _ _这表示一个起始地址为 42000000_{16} 、长度为 4096_{10} 字节的内存映射 I/O 区域。使用该函数的地址会使编译器声明一个名称如上所述的未定义（导入）符号。随后，编译器会嵌入一个 64 位重定位，在*DT INST REG ADDR* 宏用于解析 _ _ _设备寄存器的所有位置，将解析出的符号地址写入其中——按照惯例，这通常包含在一个 struct *device_config* 中，并在驱动程序的*structdevice*中引用。最后，所引用的函数实际上并未在任何地方定义。Skadi 加载器会检查未解析的导入符号是否符合内存映射 I/O 条目的编码方案；如果符合，就会为请求的内存映射 I/O 空间创建一个能力，并将能力令牌用作符号地址。通过这种方式，我们能够与使用设备树宏的现有驱动程序保持源代码级兼容，包括将能力令牌嵌入声明为*const*的结构体中。

生成式宏可自动创建子系统调用跳板 子系统调用约定通过用于调用方和被调用方的生成式宏实现，其中包含预处理器指令、编译器指令和内联汇编的组合。因此，对现有 Zephyr 组件的修改仅限于添加宏，以便使用 Skadi 子系统调用约定调用和导出子系统调用。

Skadi 对子系统调用采用以下约定。调用方和被调用方都会为子系统调用提供跳板。这些跳板有三个用途。首先，如 A.5 节所述，Northcape 只接受指向目标段第一个字节 的子系统调用。函数调用通常并非如此——相关代码可以从代码段中的任意位置开始。其次，在调用方发起子系统调用之前，以及被调用方返回之前，跳板会在需要时保存所有寄存器，并将其覆写为零，以防止信息在子系统边界之间泄露。第三，被调用方（子系统调用后立即）和调用方（从子系统调用返回 后立即）都不能信任程序计数器以外的任何寄存器。特别是，栈、线程和全局指针寄存器可能已被另一个子系统操纵，以窃取或伪造数据。因此，跳板会恢复先前复制到跳板代码段中的 上下文令牌，并获取栈、线程和全局指针寄存器对应的令牌。调用方跳板还会从上下文中恢复调用前的寄存器值

[2]U 型指令（如 *auipc* 和 *lui*）将寄存器的高 20 位设为绝对值或 PC 相对值，而 I 型指令（如 *add* 或 *ld*）则向低位加上 12 位偏移量。

子系统。调用方和被调用方跳板的宏会自动注册用于初始化子系统的函数，这些函数会为调用方和被调用方跳板准备上下文指针，因此无需为每次子系统调用手动执行此操作。调用方和被调用方跳板还会根据子系统的需求配置*mstatus* 中的 CPU 状态；例如，对于需要使用浮点单元的子系统，被调用方和调用方跳板会确保在跳转到 C 代码之前启用浮点单元。

我们在被调用方跳板中必须解决的一个关键问题是，防止栈在使用期间被覆盖。我们将此问题称为栈使用中问题（stack-in-use problem）。我们希望通过一个示例场景来解释该问题：考虑子系统 A、B 和 C。A 调用在 B 中实现的子系统调用*foo* ，导致 B 使用其子系统调用栈来保存例如返回能力令牌。为了计算 *foo*，B 需要调用 C 中的子系统调用*bar* 。当 C 使用自己的栈来计算 *bar* 时，B 需要保留自己的栈，以便返回到 A。现在假设在 *bar* 的执行期间，C 调用了一个函数指针。同时假设该函数指针解析为 B 中的子系统调用 *baz* 。在每个子系统只能访问恰好一个栈帧的朴素实现中，在 *baz* 的执行期间，B 会覆盖其调用 *bar* 时的状态。因此，在 B 从 *baz* 返回且 C 从 *bar*返回之后，B 试图继续执行 *foo* ，但使用的是 *baz* 返回时的栈帧，从而导致错误的结果。

我们采用以下方法解决栈占用问题。每个 Skadi 子系统都会分配若干栈帧，数量可在构建时配置。撰写本文时，默认数量为四个。子系统会将分配的栈存储在子系统上下文中，并附带一个位图，用于标示当前正在使用的栈。每个子系统被调用方跳板都可以利用该位图查找子系统的第一个可用栈。在从子系统调用返回之前，我们会重新加载当前位图以避免使用过时数据，并将该栈帧标记为可用。我们使用原子指令实现栈帧的获取和释放（获取时使用 amoor.d，释放栈帧时使用 amoxor.d）。因此，无需禁用中断请求，我们的实现也能做到 SMP 安全和 IRQ 安全。这种方法有一个缺点：我们无法确定栈帧是否会变为可用——目前的实现会无限期地遍历位图，等待栈帧变为可用。如果担心栈耗尽，可以额外统计尝试次数，并在超过阈值后触发受控恐慌。

初始化使用略微简化的跳板版本；虽然加载器需要将自身与子系统隔离，但子系统可以向加载器泄露其寄存器内容。此外，按照约定，加载器调用初始化函数时会传入它为该子系统分配的栈指针以及该子系统的标识符。

在 Skadi 中，调用方跳板主要声明在系统头文件中。这使使用方子系统能够透明地调用子系统调用。这种做法有一个缺点：如果天真地在头文件中实例化整个跳板，就会产生重复的调用方跳板以及未使用的调用方跳板。因此，我们提供了一种优化方案：在头文件中声明的调用方跳板会定义一个外部函数，其名称编码了子系统调用的名称和签名。该外部函数会产生一个未定义的 ELF 符号。在链接最终子系统之前，我们使用 Python 脚本扫描目标文件，并收集所有与调用方跳板对应的未定义 ELF 符号。随后，我们生成一个实现相应调用方跳板的源文件，编译并将其链接到子系统中，从而解析这些未定义的 ELF 符号。由此，我们对调用方跳板进行了去重。此外，由于编译器只会为实际使用的调用方跳板生成未定义的 ELF 符号，因此不会生成多余的调用方跳板。

调用方跳板在开始调用之前，按照 RISC-V ABI 将调用者保存寄存器 保存到寄存器集结构体 中，并在之后从同一结构体中恢复它们。在调用之前，整个寄存器组也会被清零。由此，Skadi 确保不会有信息从调用方泄露到被调用方。对寄存器组的访问使用与被调用方跳板分配栈相同的分配机制。

最后，我们的跳板实现还需要解决一个额外问题：如果线程被同步或异步取消，该线程可能会同时泄露调用方跳板的寄存器组以及被调用方跳板的栈——线程函数所使用的栈，以及该线程最后一次调用所涉及的所有寄存器组和栈都不会被清理，因为取消之后没有返回 。请注意，泄露的跳板不一定限于与该线程的处理函数相同的子系统——例如，如果线程在被取消时正在等待调度器对象（如信号量），这会永久阻塞调度器中的一个栈。为了解决此问题，调度器通过全局数据导出当前被调度线程身份的只读能力。调用方跳板和被调用方跳板使用此信息，将栈和寄存器组与当前线程关联起来。当线程被取消时，调度器会在每个子系统中调用一个回调，从而使所有子系统能够释放当前分配给该线程的寄存器组和栈。

跳板支持可变参数函数，例如*printf*、*open* 和*ioctl* 这类 C 函数是可变参数函数：它们接受任意数量的函数参数。按照惯例，这些函数至少有一个必需参数，用于告知函数传入了多少个参数：对于 printf，格式字符串包含此信息；而对于 open，则通过标志参数中某一位是否存在来表示。可变参数函数的声明中会包含省略号。C 代码可以借助不透明类型*va_list* 及其

相关访问器（如*va_start* 或*va_copy*）来处理。*va_list*也可以作为参数传递给其他函数——按照惯例，printf 这类函数通常会将参数转发给名称带有*v* 前缀的函数，例如*vprintf*。*vprintf* 接受一个*va_list* 参数，而不是省略号。无论如何，处理*va_list*的函数都应确定要从列表中提取多少个、何种类型的参数，并使用有状态的内置*va_arg* 遍历参数列表。可变参数的传递方式取决于架构：在 RISC-V 上，可变参数函数最多预期在整数参数寄存器 $[a0,a7]$ 和浮点参数寄存器 $[fa0,fa7]$，中分别接收 8 个参数。超出的参数会传递到栈上。同样，*va_list*在栈上实现为逻辑上连续的存储区域。每个存储单元的大小与机器寄存器相同 [25]。

可变参数函数对基于能力的架构 [27]构成挑战：编译器或运行时需要确保被调用方能够访问所有预期参数。与此同时，还必须注意防止将私有栈泄露给被调用方。我们提供一个特殊宏，用于为可变参数函数生成调用方跳板。该宏为函数调用生成一个包装，它将 gcc 的 __gnu_inline__ attribute 与内建函数 __builtin_va_arg_pack_len 结合使用，以确定该函数每次调用的可变参数数量。包装函数随后使用内建函数 __builtin_va_arg_pack，将可变参数函数的参数转换为 va_list。然后，va_list 被转发给第二个内部包装函数。该内部包装函数进而利用 RISC-V ABI 规范的优势：它既知道参数数量，也知道 va_list 的基址。因此，它可以使用 derive() 能力操作，从栈中派生出一个大小恰好等于 va_list 的能力。为此，请记住 va_list 中的每个参数都具有与机器寄存器相同的位数，从而确保所有预期参数都被转发，而栈的剩余部分对被调用方[3]不可访问。随后，va list 便可被转发_，如同任何其他能力令牌一样，并使用 va_arg 及其他典型宏进行访问。

特别是对于 *printf* 系列函数，还存在最后一个挑战：格式字符串中的格式说明符 *%s* 和 *%n* 表明存在指针参数。使用*%s* 时，相应的参数会被解释为指向零终止字符串的只读指针，并将该字符串传递给输出。使用 *%n* 时，该参数会被解释为指向整数的只写指针，库会在其中存储迄今为止输出的字节数。然而，将指向子系统只读数据段中字符串的能力令牌，或将栈上的 int 作为 printf 参数传递，会导致 libc 中发生访问故障，因为这些段受子系统标识符保护。因此，我们的 printf 包装函数会扫描格式字符串和可变参数列表，查找此类参数，并透明地为这些参数派生合适的能力。

跳板支持浮点单元。Zephyr 可选择提供对浮点单元（FPU）运算的支持。例如，我们会在基准测试中用它计算标准差。因此，我们在跳板中实现了 FPU 支持。为此，每个子系统都可以使用构建时标志指明是否要使用 FPU：如果不使用，则在所有调用方和被调用方跳板中忽略 FPU，以减少子系统调用的开销。否则，调用方跳板会以类似 Zephyr 在上下文切换期间处理 FPU 的方式来处理它：在调用方一侧，跳板使用机器状态寄存器中的 RISC-V *FS.dirty* 状态来判断自上次保存以来是否使用过 FPU。如果使用过，跳板就会将 FPU 寄存器保存到调用方寄存器集中；否则就跳过这一步。随后，调用方跳板会将 FPU 寄存器清零，并照常继续调用。返回后，它会将 FPU 状态设置为干净（表示已启用），并无条件地从调用方寄存器集中恢复 FPU 寄存器。

被调用方跳板只需将浮点单元设置为干净 状态，并在返回时将寄存器清零，但浮点返回寄存器除外。这样，被调用方跳板既能防止泄露私有寄存器信息，又能支持浮点参数和返回值的浮点调用约定。

动态分配器无需可信 Skadi 加载器包含一个基础分配器，它使用创建 和派生 操作。其主要用途是为重定位后的 ELF 二进制文件分配代码段和数据段。更复杂的分配器则实现为子系统，无需可信。这两种分配器实现都使用创建 操作，从未使用的能力中切分出长度按字对齐的直接能力。分配器子系统还会为直接能力强制设置最小长度，以限制碎片 [84]。两种分配器都使用派生 来限制 段长度，使其不超过实际请求的字节数，从而保障内存安全。如有需要，子系统可以使用派生 或限制 来缩短已分配能力的长度和/或减少其权限，还可以使用锁定 来获得独占访问。最后，Skadi 加载器会为两个分配器分别创建不重叠的能力，作为它们的初始堆。

实现能够感知 Northcape 能力的分配器还面临着一些额外挑战：当通过*free()* 方法将能力令牌返回给分配器时，分配器首先需要根据它发放的间接能力识别出直接能力。其次，它需要判断物理相邻的内存块是否也已释放，以便将它们合并，避免内存碎片不断增加。

[3]当然，通过可变参数函数按值传递结构体时，这种逻辑就会失效：在这种情况下，每个结构体成员都会填充至机器寄存器的大小，并通过*va_*列表传递。不过，虽然技术上允许这样做，[25], 我们在 Zephyr 中从未遇到过这种情况。

我们使用以下算法来解决这些额外挑战。分配器维护两个列表：一个空闲列表，其中存放可供分配的内存块；另一个是已分配列表，其中存放客户端当前正在使用的内存块。空闲列表中的内存块是以空闲列表头部开头的直接能力。另一方面，已分配列表由直接能力构成，其中仅包含已分配列表头部；该头部维护一个能力令牌，指向分配给调用方的直接能力。因此，分配器可以遍历和操作已分配列表，而调用方则能够通过锁定操作独占访问已分配能力。已分配列表头部还包含分配器发放的间接能力令牌。空闲列表头部还包含能力内存块的边界和大小，因此可以省去一次*inspect()* 操作。已分配列表实现为红黑树，并以分配器发放的间接能力作为键。空闲列表由两棵红黑树构成：一棵以内存块大小[4] 为键，另一棵以内存块基地址为键。我们采用 Zephyr 的红黑树侵入式实现，该实现本身不需要动态内存，并能保证键搜索的复杂度为 $\mathcal{O}(log(n))$，同时具备实时能力。分配操作可以通过红黑树上的搜索操作，以 $\mathcal{O}(log(n))$ 的复杂度实现最佳适配或首次适配分配策略。作为副作用，该操作会从空闲列表中移除已分配的内存块，将头部转换为已分配内存块头部，并将其插入已分配列表。释放操作首先在已分配列表中执行搜索操作，将间接能力映射到直接能力并获取边界信息。随后，它会在按内存块基地址排序的空闲列表中查找，以寻找已释放能力的物理相邻块。之后，分配器通过将已分配列表头部块与能力的剩余部分合并，使已释放能力线性化。如果在空闲列表中找到相邻块，就将其移除。然后，分配器执行 *merge* 操作，并将一个由已释放能力及其相邻块构成的新条目插入空闲列表。否则，将线性化后的已释放能力插入空闲列表。

内存分配并不总是需要子系统调用 动态分配器子系统有一个关键缺点：分配和释放内存都需要进行子系统调用，其开销高于简单的函数调用。为降低这一开销，我们为每个子系统提供了一个简单的本地分配器。本地分配器由各子系统 *.bss* 段中的连续内存区域和一个原子位图组成。该内存区域被划分为大小相等的内存块，其分配状态由位图跟踪。

对于每个分配请求，每子系统分配器都会选择第一个空闲块，并为其派生一个能力。

[4]我们使用内存块基地址作为次级键，以确保强顺序 g.

派生出的能力长度恰好是调用方请求的长度，以确保内存安全。

返回时，每子系统分配器可以使用检查操作来获取相应内存块的物理基址。利用内存块大小和内存区域的物理基址，可以高效地确定相应的内存块。

满足以下三个条件时，将使用本地分配器：

- 内存请求的大小小于内存块大小。
- 未请求可锁定权限。
- 至少有一个空闲块。

请注意，由于内存区域属于 .bss 段，试图锁定已分配的块会锁定整个.bss 段，可能导致子系统发生访问故障。所有不符合每子系统分配器要求的内存请求都会转发给动态分配器子系统。

本地内存分配器还有一项额外优势：已释放的数据无需清零。按照惯例，*malloc*系列函数会返回未初始化的数据。软件需要使用*calloc*风格的函数来请求用零覆盖的内存。因此，分配时可以省去开销高昂的*memset*操作。不过，如果使用远程分配器，这种情况会带来泄露信息的风险。因此，为了提高安全性，在执行释放子系统调用之前，总会覆盖已释放的内存。另一方面，对于本地分配器，则无需这样做。

libc 无需被信任 Skadi 子系统可以使用无状态的 C 库函数，包括字符串函数和 *printf* 系列函数，这些函数由 *libc*子系统提供。为此，*libc* 子系统包含一个重定位后的 picolibc 实现，也就是 Zephyr 默认使用的同一个 libc。必要时，*libc*子系统会通过子系统调用与其他组件交互。例如，它会使用控制台子系统提供的子系统调用，通过 *printf*进行字符输入/输出。*libc* 拥有自己的数据段，但我们对 picolibc 进行了配置，使所有处理都在栈上进行，从而避免子系统之间的数据泄漏。因此，数据段仅用于存储常量。

对于我们的 shell 子系统等非安全关键子系统，可以选择直接调用 libc 函数。为此，它们会相应地声明未定义外部符号，并依靠 Skadi 加载器直接重定位 libc 函数。因此，调用 libc 函数时，这些子系统会泄露 其子系统标识符给 libc，但能获得代码大小和延迟方面的优势。

安全关键子系统可以通过子系统调用来调用 C 库函数。为此，我们将一个名为 libSkadi 的 libc 存根静态链接到每个子系统中。该存根为标准 C 库函数声明调用方跳板。该存根随后

为标准 C 库函数提供实现，这些实现会调用调用方跳板。因此，在链接子系统时，链接器会使用 libSkadi 中定义的库函数，而不会生成未定义符号。

最后，如果需要，子系统可以内联选定的 C 库函数，例如字符串函数。这样可以提高性能，但代价是增大代码大小。此选项可在构建时选择，且默认启用。

Skadi调度器在 Zephyr 中，纳内核提供阻塞数据结构，例如信号量、队列和管道。这些数据结构的 API 包含取消点，例如*k sem take*：当线程调用此类函数且没有可用资源时（例如，信号量计数为 0），线程会阻塞，直到有资源可用。_ _

在底层，调用取消点会使调用线程在调度器中失去可运行状态。调度器会选取一个新的可运行线程，并执行上下文切换以继续执行。

调用*k_sem_give* 之类的函数，会使等待中的线程重新获得可运行状态。因此，在后续的上下文切换中，它们可以继续执行。

关键的是，所有 Zephyr 子系统在内部都依赖调度器提供的数据结构来实现取消。因此，我们选择将阻塞数据结构纳入调度器子系统，并通过子系统调用使其可访问。另一种设计方案还可以将核心调度算法移至另一个子系统。这样一来，恶意的调度策略就无法破坏这些数据结构的完整性不变式。

阻塞数据结构的内部实现包含链表等数据结构。因此，假设调度器与恶意组件共享了一个*structk sem* ，恶意组件就可以操纵该链表，从而实质上在调度器的保护域中获得写原语，破坏隔离。因此，调度器 API 将提供的能力令牌映射到所提供对象的动态分配影子副本。这种设计类似于 Linux 中的文件描述符或 Windows 中的句柄，它们在不直接暴露内核对象的情况下授予访问权限。与此同时，我们保持与 Zephyr 子系统完全的 API兼容性。

Skadi 子系统初始化我们提供两种调用初始化函数的方式：首先，有一个新的宏*SKADI SUBSYSTEM INITIALIZATION FUNC-_ _ _TIONS*。它指定一个函数，在子系统加载后立即运行，并在设置权限和子系统 ID 保护机制之前运行。例如，Skadi 蹦床机制会使用它来初始化私有数据，这些数据在子系统的剩余生命周期内保持只读。

我们还将 Zephyr 的 *SYS_INIT* 宏移植到了 Skadi。此机制允许用户指定要在某个运行级（例如 *EARLY* 或*POST KERNEL* ）运行的函数，并为其设定相对于该运行级中其他初始化函数的优先级 。为此，对于每个运行级，Skadi 加载器都会通过一个众所周知的被调用方跳板，向每个子系统查询下一个优先级。随后，它确定哪个子系统的优先级最低，并通过第二个被调用方跳板指示应运行下一个初始化函数。子系统使用与 Zephyr 相同的可迭代段机制来查找各自的初始化函数。负责返回下一个优先级并调用初始化函数的代码，包含在静态链接的*libSkadi*中。

Skadi 操作系统本身的初始化分为两部分：一部分由加载器执行，另一部分由调度器执行。加载器的初始化例程既负责平台初始化（包括将 .bss 段清零、设置初始栈等子系统），也负责 Zephyr 的*EARLY*,*PRE_KERNEL_1* 和*PRE_KERNEL_2*初始化运行级别。换句话说，加载器负责在调度器启动前调用初始化函数。早期初始化函数涉及的子系统包括设置初始分配器和早期控制台，以及重定位各子系统。完成*PRE_KERNEL_2*后，加载器通过子系统调用启动调度器。调度器接着创建主线程并执行自身的初始化。随后，调度器可以调用更高运行级别中的初始化函数，例如*POST_KERNEL* 和 *APPLICATION*。为此，它通过*loader_proxy*子系统调用加载器提供的子系统调用。完成最后一个运行级别后，调度器会对子*main()* 函数执行一次子系统调用。

Skadi 加载器在初始化后不再存在 正如我们在第 3 节中所述，Skadi 加载器是唯一的可信软件组件。它也是唯一以子系统 ID 0 运行的组件，因此只有加载器能够创建带有外部子系统标识符的 set-subsystem-id 令牌。这对于重定位和引导系统至关重要，但初始化完成后便不再需要此功能[5]。为防止加载器成为安全风险，它会在初始化后被销毁。

调度器通过子系统调用触发 *POST_*内核 及更高级别的初始化运行级别，并在最后一个初始化运行级别完成后调用 *main()* 方法。因此，当加载器完成最后一个初始化运行级别后，便可以销毁自身并返回调度器。这得益于 陷门子系统。

trapdoor 子系统以子系统 ID 0 运行，因此能够对根能力执行操作。加载器完成最后一个初始化运行级别后，会立即通过子系统调用调用 Trapdoor，并将原始

[5]未来版本的 Skadi 可能会保留部分加载器，使其能够在运行时动态加载子系统。

返回地址作为参数传递给它，该地址由调度器子系统提供。Trapdoor 使用根能力上的 创建Northcape 操作，切分出一个新能力，其范围涵盖加载器先前占用的内存。随后，Trapdoor 对新能力执行 撤销操作，确保内存被零覆盖，不会泄露任何机密信息。然后将新能力交给分配器子系统，使其可用作通用堆。出于同样的原因，在调用 Trapdoor 之前，初始分配器的剩余堆也会转交给分配器子系统。作为倒数第二步，Trapdoor 使用 限制操作，移除根能力剩余部分的所有权限。因此，根能力不再能被直接使用，也无法从中派生任何新能力。不过，先前派生的内存映射 I/O 能力仍可提供设备访问。最后，Trapdoor 清除所有寄存器，并通过所提供的子系统调用能力返回调度器子系统，同时更改子系统 ID。

由于加载器确保没有任何子系统可以访问陷门的子系统调用，并且确保没有其他子系统在子系统 ID 0 下执行，因此之后无法再执行子系统 ID 0 下的代码。因此，除了销毁加载器之外，特权子系统 ID 0 也处于非活动状态。陷门返回后，Skadi 只执行已加载的子系统。

C.3 Skadi 的自定义指令集架构扩展

仅使用标准 RISC-V API 和 Bredi 的内存映射 I/O 寄存器接口，Skadi 即可实现功能与安全性。不过，我们在以下三个方面引入了显著的性能损失:

1. 通过 检查验证子系统调用中返回地址的有效性
2. 子系统调用返回时将寄存器清零
3. 使用加载和存储指令访问内存映射 I/O 寄存器接口

我们引入了三项指令集架构扩展以提升性能：用于访问操作模块的自定义子系统调用指令、自定义寄存器未初始化指令以及自定义控制状态寄存器（控制与状态寄存器）。

C.3.1 子系统调用指令

如 A.5 节所述，子系统调用的调用方和被调用方都需要验证，即将跳转到的能力是否带有设置_子系统_标识符限制。如果不进行此验证，跳转可能导致失去对自身子系统 ID 的控制权，从而破坏隔离。可以使用检查 操作执行此检查。虽然这能提供足够的安全性，但会消耗大量 CPU 周期，尤其是在子系统调用耗时较短时。

因此，我们专用的子系统调用指令会在跳转提交时执行此检查。为此，如果跳转后的第一条待执行指令并非从适合作为子系统调用跳转目标的能力中加载，子系统调用指令就会触发指令访问异常 。实现此功能只需对我们的硬件架构做少量修改：首先，我们的 cva6 MMU 会为每条取出的指令添加一个标志，用以表明根据 A.5 节详述的安全要求，该指令是否适合作为子系统调用目标。其次，我们使用跳转并链接寄存器（*jalr*）指令编码空间中一个当前保留的功能代码，来指示该跳转属于子系统调用。跳转指令的行为与普通的*jalr*相同，但在提交时，它会设置 CPU 提交阶段中的一个寄存器，指示下一条指令必须是有效的子系统调用目标。最后，如果子系统调用所跳转到的指令不是 有效的子系统目标，提交阶段就会触发异常。

这种方法能够提供足够的安全性：如果子系统调用无效，提交阶段就会触发异常，因此被调用的指令不会提交，攻击者也无法得逞。同时，安全检查本身不会增加任何周期开销。此外，这种方法也省去了对带有检查的函数点进行安全检查的需要。

C.3.2 快速寄存器清零指令

为了防止调用方与被调用方之间的信息泄露，双方都会清除除函数地址／返回地址寄存器和参数／返回值寄存器以外的所有寄存器。鉴于 RISC-V 的寄存器组规模较大（32 个整数寄存器和 32 个浮点寄存器），这需要耗费相当多的周期，对于短子系统调用尤其如此。

为了降低这一开销，我们在 cva6 中引入了未初始化寄存器：通过自定义指令，可以加载初始化掩码，有选择地将寄存器标记为未初始化。未初始化寄存器的读取结果始终为 0，与其被设为未初始化时所存储的值无关。向未初始化寄存器写入数据后，该寄存器便会变为已初始化，读取时会返回原始值。

我们的实现将初始化掩码以位图形式与寄存器文件一同存储。当相应寄存器尚未初始化时，每次寄存器读取都会返回零。每次寄存器写入都会清除位掩码中对应的位，使该寄存器再次可读。最后，该掩码同时包含整数寄存器和浮点寄存器，并一次性从源寄存器写入。自然地，中断服务例程和非中断服务例程寄存器组各自都有一份掩码。

总体而言，这项优化将清除寄存器组的最坏情况时间从大约 120 条指令缩短至 4 条指令。

C.3.3 Bredi 能力操作模块通过 CSR 访问

通过加载和存储指令读取和写入 Bredi 能力操作模块需要使用栅栏指令来确保访问顺序，从而导致性能下降。此外，读取和写入还需要经过缓存子系统和互连网络，耗费多个周期。

为了加快对操作模块的访问，我们实现了自定义控制与状态寄存器（控制状态寄存器）。每个控制与状态寄存器都映射到内存映射 I/O 接口中的一个寄存器。然而，与内存映射 I/O 寄存器组不同，控制状态寄存器不会重复。相反，指令是否在中断请求上下文中执行，决定了要使用寄存器接口的哪一个副本。

我们使用静态仲裁器将操作模块连接在 AXI 和 CSR 接口之间，并优先处理 CSR 接口，因为 CPU 是该接口的主要用户。

C.4 Skadi 零拷贝网络

借助 Bredi 能力系统，Skadi 实现了零拷贝网络，从而提升性能和安全性。为此，它从 Zephyr 网络协议栈中消除了两次数据包复制操作：Zephyr 将每个传入数据包从 DMA 缓冲区复制到网络结构体_数据包 中（发送时则反向复制）。对于使用套接字 API 的应用程序，每个数据包还会被复制一次，移入或移出用户提供的缓冲区。关键在于，网络结构体_数据包 结构由一个或多个网络结构体_缓冲区组成，其中保存了数据包实际数据（的指针）以及大小信息。因此，网络数据包不必在物理上连续。例如，如果现有缓冲区没有预留空间，就可以使用单独的网络结构体_缓冲区来添加或移除网络头部。

通过向 DMA 设备提供从*struct net_pkt*中的缓冲区派生出的间接能力，消除了第一次复制。为此，我们使用 zehpyr 的非连续 DMA API，确保网络数据包无需线性化即可进行零拷贝 DMA。具有散布-聚集能力的 DMA 设备原生支持此功能，例如我们的 Xilinx AXI DMA 设备 [2]。DMA 设备指示传输完成后，丢弃间接能力，确保设备失去对缓冲区的访问权限。

对于数据报套接字和流式套接字，我们通过修改套接字 API 消除了第二次复制。在传输时，我们提供一种自定义的 setsockopt 操作，客户端可用它来表明希望使用零拷贝网络。在这种情况下，客户端应为要传输的数据提供间接能力。随后，间接能力会被封装到 *structnet_buf* 中，并通过网络协议栈传递。为此，我们将预留空间设为 0，以便在新的 *struct net_buf*中添加网络头部。这样，调用方就能向网络协议栈提供一个只读能力。调用方还可以注册一个释放回调，使其在数据包传输完成后调用，从而允许调用方按需丢弃该能力。这也有助于等待数据包传输完成，例如，借此确定何时可以查询发送时间戳。

在接收端，我们提供一个基于*ioctl*的自定义 API，以访问网络数据。子系统可以调用*ioctl*，为当前位于套接字接收队列队首的网络数据包中的应用数据获取一个间接能力。随后，数据包会被移至一个专用等待队列。调用方随后可以处理应用数据，并通过另一个*ioctl*释放该数据包，从而允许网络协议栈释放该数据包，并释放授予它的间接能力。

C.5 Skadi 开发

我们已为 Zephyr 工具链添加了对 Skadi 子系统开发和测试的支持。任何源文件集合都可以通过 CMake 函数 创建_-*skadi*_子系统声明为 Skadi 子系统。该函数负责使用正确的代码模型、架构和包含路径标志编译源文件，并将它们链接为可重定位 ELF 二进制文件。该函数还会将新的扩展添加到已加载的 Skadi 扩展集合中。如前所述，在构建过程中，我们会计算扩展的初始化顺序，并将它们纳入二进制文件中。

新的 Skadi 子系统可以包含声明调用方跳板的头文件，以便将子系统调用导入其他子系统。它们还可以通过提供作为导出 ELF 符号的子系统被调用方跳板，导出子系统调用。

最后，可以使用 Zephyr 通用的封装命令（例如，*west debug*）调试 Skadi 子系统。为此，我们使用了与 cva6 一同提供的调试模块，该模块实现了 RISC-V 调试规范 [56]。在解释如何将其集成到 Skadi 和 Bredi 之前，我们将简要概述调试模块的工作原理。

调试模块是一个与 CPU 分离的硬件组件。它作为 AXI 从设备连接到北桥。它还有自己的总线主设备接口，可用于直接访问内存。它还可以向 CPU 触发调试中断。最后，调试模块提供了一个可连接到主机系统的 JTAG 接口。

除调试中断外，CPU 与调试模块之间的所有通信都以内存访问的形式进行。因此，一旦触发调试中断或命中断点，CPU 就会将程序计数器设为调试接口中的程序缓冲区 MMIO 寄存器。程序计数器中的默认指令会使 CPU 忙等。外部调试器可以通过 JTAG 向调试接口发送指令，例如检查寄存器内容或读取内存 [56]。

我们在 CPU 中添加了一个自定义控制状态寄存器，使得

调试模块的 MMIO基地址可配置。因此，我们可以在加载子模块之前为调试模块派生一个能力令牌。通过不按子系统限制该能力，调试模块便可不受当前子系统影响地使用。这样，我们就可以通过子系统标识符限制来保护根能力以及所有子系统的代码和数据，同时不影响调试模块的功能。如果不使用调试模块，只需在设备树中将其停用即可——这会使我们的加载器不为其创建能力，从而在加载器未激活时使其无法访问。

Zephyr 集成了 RISC-V 调试模块与 *west* 构建封装，并以 OpenOCD 作为主要调试器。OpenOCD 可用于加载并运行 Zephyr 二进制文件。它还可充当 GDB服务器，从而支持使用 gdb 进行源代码级调试。

Bredi 无需进行重大修改即可使用调试模块。我们将 OpenOCD 配置为通过 CPU 指令访问内存，并让调试模块的 AXI 主接口保持未连接状态。因此，gdb 可以像访问物理地址一样读取和写入能力令牌。这便于在 Skadi 子系统上下文中设置断点、进行单步执行以及打印值。由于调试器通过 CPU 访问寄存器和内存，它触发的所有内存请求都会自动受到当前子系统上下文中的 Bredi令牌转换约束。同时，由于调试模块的主接口未连接，它无法绕过 Bredi 访问内存。

我们甚至为 Skadi 子系统提供符号支持。为此，Skadi 加载器会输出一条 GDB命令，用于在重定位完成后的最终位置，从每个子系统的 ELF二进制文件中加载调试符号。因此，程序的调试方式与在不支持 Bredi 的传统操作系统上一样，无需为调试 Skadi 子系统提供特殊的硬件或软件支持。

D 详细评估结果

为便于提高透明度，我们提供了评估实验的更多测量值。表10和表 13 列出了计算与调度器基准测试结果的详细数据。表 12 包含 IRQ基准测试的详细结果。除了在 Bredi 上运行的 Zephyr、Linux 和 Skadi，我们还将上游 cva6 SoC 纳入其中作为引用。表 11 提供了网络基准测试的详细结果。由于上游 cva6 SoC 使用了不同的网络子系统，因此此处不将其作为引用。

表10：Skadi、Zephyr 和 Linux 的计算基准测试（Coremark）和内存基准测试（Stream）结果。在上游 cva6 Coremark，我们提供五次运行的迭代次数、平均时长和标准差，以及标准指标，即每次运行每秒平均迭代次数的中位数。对于 Stream，我们提供报告中的结果。

基准测试	Zephyr cva6 SoC	Linux Bredi	Zephyr Bredi	Skadi Bredi
Coremark 迭代-次	2000	1100	2000	2000
中位数 Core-mark 迭代-次/秒	110.00	92.10	110.02	110.14
Coremark 持续时间-tion 平均值 / 最小值 / 最大值 / 标准差/秒	18。18 / 18。18 / 18。18 / 0	11。95 / 11。93 / 11。96 / 0。01	18.18 / 18.18 / 18.18 / 0	18.16 / 18.16 / 18.16 / 0.00
Stream Copy 最佳速率/ MB/s	29.6	14.3	26.7	26.7
Stream Copy 平均值 / 最小值 / 最大值 / 标准差/秒	5.41 / 5.40 / 5.50 / 0.03	11。23 / 11。22 / 11。23/-	6.04 / 6.00 / 6.10 / 0.05	6.00 / 6.00 / 6.00 / 0.00
Stream 规模 最佳速率/ MB/s	25.4	12.6	24.2	24.1
Stream 规模 平均值 / 最小值 / 最大值 / 标准差/秒	6.34 / 6.30 / 6.40 / 0.05	12。75 / 12。75 / 12。76/-	6.66 / 6.60 / 6.70 / 0.05	6.65 / 6.65 / 6.65 / 0.00
Stream Add 最佳速率/ MB/s	27.6	13.7	26.7	26.6
Stream 加法 平均值 / 最小值 / 最大值 / 标准差-次/秒	8.73 / 8.70 / 8.80 / 0.05	17.53 / 17.52 / 17.60/-	9.03 / 9.00 / 9.10 / 0.05	9。02 / 9。02 / 9。02 / 0。00
Stream 三元组 最佳速率/ MB/s	27.6	12.9	25.3	25.3
Stream 三元组 平均值 / 最小值 / 最大值 / 标准差/秒	8.77 / 8.70 / 8.80 / 0.05	18.62 / 19.00 / 18.63/-	9.50 / 9.50 / 9.50 / 0	9.50 / 9.50 / 9.50 / 0.00

表11：Skadi、Zephyr 和 Linux 的网络基准测试完整结

基准测试	Linux	Zephyr	Skadi
Ping 延迟 最小值 / 平均值 / 最大值 / 标准差 ms	1.77 / 1.92 / 2.62 / 0.15	0.83 / 0.88 / 1.08 / 0.02	2.46 / 2.56 / 3.96 / 0.08
iperf 最小值 / 最大值 / 平均值 / 标准差 TCP 吞吐量（Kbits/s）	7580 / 14300 / 9031。17 / 1733。20	4380 / 4410 / 4396。67 / 6。81	1490 / 1500 / 1499。33 / 2。52
接收网络协议栈处理持续时-间 纳秒 最小值 / 最大值 / 平均值 / 标准差	1212720 / 1721960 / 1346524.21 / 136558.72	516193 / 564497 / 534142。81 / 8120.03	2111305 / 2180633 / 2138493.63 / 13707.45
发送网络协议栈处理持续时间-（纳秒）最小值 / 最大值 / 平均值 / 标准差	765200 / 1827240 / 1050532.63 / 203 162.69	514855 / 644431 / 530697.96 / 13 202.11	1491231 / 1559519 / 1516145.02 / 11670。62
MQTT-TLS 迭代时长 纳秒 最小值 / 最大值 / 平均值 / 标准差	-	1993429040 / 1993582320 / 1993484845.47 / 30590.98	2049993880 / 2065165720 / 2057174000。84 / 5772516。87

表 12：Skadi、Zephyr 和 Linux 的实时性能基准测试结果。还在上游 FPGA SoC 上的

基准测试	Zephyr cva6 SoC	Linux Bredi	Zephyr Bredi	Skadi Bredi
IRQ 延迟 独立 最小值 / 最大值 / 平均值 / 标准差 纳秒:	4820 / 88140 / 12050。74 / 21 226.22	1324020 / 9236700 / 3808793。68 / 3194695。59	4940 / 89960 / 24127.16 / 32 149.03	65720 / 72840 / 68423。16 / 1529。02
IRQ 延迟 单调 最小周期 / 最小值 / 最大值 / 平均值 / 标准差 纳秒:	68000 / 4740 / 62820 / 6061.68 / 7212.84	1312640 / 16432080 / 6866233。68 / 3989003。35 / 5691090。28	70000 / 4860 / 69740 / 8392.42 / 12120.04	136000 / 41300 / 114860 / 58826。32 / 18714。18

表 13：Skadi 和 Zephyr 的调度器宏基准测试详细结果。还包括上游 cv GA SoC 上的 Ze
准测试中的名称。

宏基准测试	Zephyr cva6 最小值 / 最大值 / 平均值 / 标准-差 纳秒	Zephyr Bredi 最小值 / 最大值 / 平均值 / 标准差 纳秒	Skadi Bredi 最小值 / 最大值 / 平均值 / 标准差-偏差 纳秒
线程让出（抢占式）上下文切换	4000 / 4960 / 4003.70 / 47.57	4040 / 4880 / 4067.06 / 64.08	13600 / 15840 / 13719。04 / 275。05
线程让出（协作式）上下文切换	4000 / 4960 / 4005.51 / 56.91	4040 / 4600 / 4061。95 / 36。63	13600 / 16560 / 13715.94 / 295.71
中断服务程序恢复被中断线程内核 - 从中断服务例程返回到被中断的线程线程	5080 / 7080 / 5934.75 / 124.00	5520 / 5840 / 5625。17 / 122。47	176800 / 198160 / 187828.02 / 3154.82
中断服务程序恢复不同线程内核 - 从中断服务例程返回到另一个线程	4800 / 5080 / 4839.92 / 60.18	5080 / 5400 / 5102。75 / 51。76	168520 / 188720 / 179111.04 / 3097.12
内核创建内核线程 - 创建线程	1727320 / 1805440 / 1765850.25 / 11137。18	2173200 / 2266240 / 2220408.44 / 13927.67	2594040 / 2692720 / 2642536.44 / 15995。25
内核启动内核线程 - 启动线程	8800 / 15640 / 12421.15 / 954.30	9240 / 15280 / 12168.32 / 900.14	87040 / 103040 / 95272。40 / 2539.70
内核发起的线程挂起 - 挂起线程	6520 / 9520 / 8426。09 / 488。19	6000 / 9120 / 8254.19 / 420.37	55480 / 68040 / 61603。74 / 1968.75
内核发起的线程恢复 - 恢复线程	4760 / 6720 / 5746.77 / 301.32	4840 / 6440 / 5699.66 / 259.41	52120 / 64800 / 57205.87 / 1813.72
线程中止内核调用 - 中止线程	6480 / 10480 / 9197.83 / 585.67	6520 / 10440 / 8873.57 / 576.07	1745440 / 1821880 / 1784033.93 / 13360。66
FIFO 立即写入内核调用 - 加法 数据到先进先出队列（无上下文切换）	3680 / 4280 / 3683.50 / 44.93	4160 / 4680 / 4189.51 / 44.12	32520 / 35320 / 32751。60 / 379。34
FIFO 立即读取内核调用 - 获取数据来自先进先出队列（无上下文切换）	2200 / 2240 / 2220。18 / 20。00	2280 / 2800 / 2280.68 / 17.23	26400 / 28000 / 26517.03 / 222.98
FIFO 放入分配即时内核 - 分配以便向先进先出队列添加数据（无上下文切换）	11800 / 12880 / 11858.65 / 79.53	11760 / 12800 / 11814.75 / 84.59	32000 / 34440 / 32446。15 / 307。58
FIFO 获取释放即时内核 - 释放从先进先出队列获取数据时（无上下文切换）	8480 / 9080 / 8511。24 / 60。18	8360 / 8880 / 8387.90 / 55.10	26280 / 29560 / 26362.45 / 217.27
先进先出队列.get.阻塞.k_to_k - 获取数据来自先进先出队列（带上下文切换）	5600 / 6440 / 5655。96 / 80。33	5560 / 6200 / 5601.61 / 78.28	18400 / 21880 / 19562.41 / 476.69
先进先出队列.put.wake+ctx.k_to_k - 加法数据到先进先出队列（含上下文切换）	8720 / 9320 / 8751。72 / 58。60	8720 / 9760 / 8747.90 / 62.04	23600 / 25640 / 23807。12 / 266。45
fifo.get.free.blocking.k_到_k - 空闲从先进先出队列获取数据时（含上下文 siwtch)	5720 / 6680 / 5831.20 / 129.82	5360 / 6000 / 5368。36 / 62。29	18800 / 21640 / 19073.69 / 457.81
fifo.put.alloc.wake+ctx.k_to_k - Al-定位以向先进先出队列添加数据（带 ctx 上下文切换)	8880 / 9480 / 8928.88 / 53.02	8840 / 9360 / 8858。37 / 59。92	23400 / 25720 / 23683.90 / 280.99
lifo.put.immediate.kernel - 加法数据到后进先出（无上下文切换）	3800 / 4400 / 3832.00 / 57.04	3800 / 4360 / 3827.90 / 48.25	31720 / 33880 / 31930.09 / 337.86
lifo.get.immediate.kernel - 获取数据来自后进先出（无上下文切换）	2040 / 2160 / 2059。94 / 20。24	2120 / 2640 / 2121.05 / 23.29	26520 / 28240 / 26596.54 / 189.74
lifo.put.alloc.immediate.kernel - 分配-位于 LIFO 中添加数据（无上下文切换）	11880 / 12760 / 11935.52 / 90.33	11760 / 12520 / 11800.40 / 70.76	31760 / 34000 / 31972.82 / 330.87
lifo.get.free.immediate.kernel - 释放从后进先出（无上下文切换）	8440 / 9080 / 8497.65 / 52.86	7640 / 8240 / 7646.59 / 40.08	26240 / 28560 / 26335。72 / 211。59
lifo.get.blocking.k_to_k - 获取数据从后进先出（含上下文切换）	5400 / 6400 / 5440.36 / 109.40	5800 / 6720 / 5845.51 / 99.03	18680 / 21800 / 19296.08 / 495.55
lifo.put.wake+ctx.k_to_k - 加法 数据到后进先出（带上下文切换）	8240 / 8840 / 8260.26 / 46.74	8240 / 8760 / 8267.70 / 48.32	23360 / 25120 / 23523.62 / 262.69
lifo.get.free.blocking.k_to_k - 释放从后进先出队列获取数据时（含上下文切换）	5080 / 5720 / 5084.98 / 48.46	5120 / 6280 / 5371。02 / 250。42	18280 / 20920 / 18530.25 / 421.76
后进先出.put.alloc.wake+ctx.k_到_k - Al-定位以向后进先出添加数据（含 ctx 切换）	8040 / 8640 / 8043.62 / 44.07	8480 / 9040 / 8485.47 / 36.26	22960 / 25360 / 23072.88 / 247.26

事件发布即时内核 - 发布 事件（没有唤醒任何内容）	4320 / 5280 / 4373。23 / 62。51	4280 / 4600 / 4317.59 / 38.72	17720 / 18880 / 17798.15 / 181.44
事件设置即时内核 - 设置 事件（没有唤醒）	3280 / 3880 / 3300.70 / 44.70	3360 / 3920 / 3363.98 / 44.35	17720 / 19440 / 17806.31 / 196.35
事件等待即时内核 - 等待 对于任意事件（无上下文切换）	2480 / 2680 / 2497。45 / 20。63	2560 / 3040 / 2560。48 / 15。21	16560 / 17960 / 16655。12 / 188。94
events.wait _all.immediate.kernel - 等待所有事件（无上下文切换）	2080 / 2680 / 2081.25 / 26.90	2120 / 2680 / 2140.54 / 26.31	16960 / 18240 / 17039.48 / 168.05
events.wait.阻塞.k_to_k - 等待 适用于任何事件（含上下文切换）	5400 / 6480 / 5682.37 / 187.66	5760 / 6520 / 5812。70 / 80。52	19960 / 23760 / 20954.17 / 664.17
事件.设置.唤醒+上下文.k _to_k - 设置 事件（含上下文切换）	11480 / 12120 / 11548.78 / 77.56	11000 / 12120 / 11302.39 / 123.53	24120 / 27400 / 24982.95 / 500.72
events.wait_所有阻塞.k_到_k - 等待所有事件（含上下文切换）	5960 / 6800 / 5988.22 / 100.26	5960 / 7120 / 6057.61 / 185.03	19840 / 24120 / 20589。43 / 526。27
事件.发布.唤醒+上下文.k_至_k - 发布 事件（含上下文切换）	11120 / 12160 / 11462。75 / 165。09	11080 / 11800 / 11110.87 / 87.48	25360 / 27480 / 25949。43 / 407。87
semaphore.give.immediate.kernel - 释放一个信号量（没有等待者）	2520 / 3120 / 2537.33 / 27.03	2600 / 2600 / 2600。00 / 0。00	17240 / 19400 / 17323.18 / 195.38
semaphore.take.immediate.kernel - 获取信号量（不阻塞）	1600 / 1640 / 1615。72 / 19。54	1680 / 1680 / 1680.00 / 0.00	16440 / 18400 / 16545.37 / 202.81
信号量.获取.阻塞.k_to_k - 获取信号量（上下文切换）	4560 / 5480 / 4594.25 / 98.46	4640 / 5280 / 4650.61 / 74.18	19480 / 23600 / 20730.89 / 655.34
semaphore.give.wake+ctx.k_to_k - 释放信号量（上下文切换）	8200 / 8840 / 8253.87 / 59.42	7880 / 8400 / 7883.42 / 39.12	23400 / 27560 / 24482.81 / 479.94
条件变量.等待.阻塞.k_to_k - 等待 用于条件变量（上下文切换）	7960 / 9800 / 8300.42 / 248.81	7920 / 9720 / 7997.11 / 139.09	27800 / 30960 / 28254.87 / 557.20
条件变量.signal.wake+ctx.k_to_k - 发送条件变量信号（上下文切换）	10280 / 11280 / 10334。79 / 83。94	10120 / 10880 / 10163.70 / 77.92	23680 / 26080 / 23806。55 / 260。22
立即内核栈压入 - 加法 数据到 k_栈（无上下文切换）	2680 / 3240 / 2701.75 / 32.25	2760 / 3600 / 2761.61 / 31.18	17960 / 20960 / 18153。81 / 334。97
栈弹出立即内核 - 获取 来自 k_栈的数据（无上下文切换）	2800 / 3160 / 2852.58 / 48.89	2840 / 3120 / 2861。03 / 54。69	20880 / 22640 / 21023。96 / 200。79
栈.弹出.阻塞.k_到_k - Get 来自 k_栈的数据（含上下文切换）	5120 / 6200 / 5178.09 / 173.92	5240 / 5760 / 5242.89 / 35.52	22440 / 26760 / 23344。40 / 822。05
栈.push.wake+ctx.k_to_k - 加法 数据 to k_栈（带上下文切换）	8440 / 9040 / 8451。10 / 54。21	8200 / 8960 / 8452。30 / 53。83	23280 / 25480 / 23410.61 / 273.15
mutex.lock.immediate.recursive. 内核 - 锁定互斥锁	1960 / 2640 / 1961.25 / 27.90	2000 / 2680 / 2021年.67 / 36.56	16560 / 18360 / 16670.11 / 217.02
mutex.unlock.immediate.recursive. 内核 - 解锁互斥锁	1440 / 6680 / 1462.51 / 168.14	1520 / 6320 / 1525.87 / 153.83	15640 / 24840 / 15763.30 / 369.33
堆即时分配 - 平均值 堆分配耗时	7120 / 7120 / 7120.00 / 0.00	6800 / 7360 / 6805.89 / 57.15	5360 / 6120 / 5420.21 / 127.93
堆即时释放 - 平均耗时 用于堆释放	7120 / 7120 / 7120.00 / 0.00	6800 / 7360 / 6805.89 / 57.15	5360 / 6120 / 5420.21 / 127.93