CHERI 上虚拟机实现中的陷阱：移植 CRub 的要点 y

Hanhaotian Liu[a] 、Tetsuro Yamazaki[a] 和Tomoharu Ugawa[a]a 东京
大学，东京，日本

摘要 CHERI（能力硬件增强型 RISC 指令）是一种旨在解决内存安全问题的新型硬件。通过用硬件能力替代传统指针，它增强了现代软件系统的安全性。虚拟机（VM）就是能够受益于 CHERI 保护的一类系统，因为其中可能存在潜在的内存漏洞。

然而，开发适用于 CHERI 的虚拟机并进行移植并非易事。基于传统架构对 C 语言未定义行为的假设，会带来许多细微的陷阱。这些假设与 CHERI 更严格的内存安全模型相冲突，从而导致意外故障。

尽管此前已有一些工作讨论过移植虚拟机的过程，但它们关注的是整体移植过程，而不是 CHERI 上虚拟机实现中的陷阱。虽然已有 CHERI 编程指南，但它面向通用编程，并未涉及虚拟机特有的问题。

我们将 CRuby 移植到 CHERI，作为一项案例研究，并调查了此前将虚拟机移植到 CHERI 的相关工作。我们根据发现的问题的成因对其进行分类和讨论。

本文阐述了各类别中特定于虚拟机的陷阱。大多数陷阱源于 C 语言中的未定义行为；尤其是，虚拟机的实现技术和惯用法往往假设传统架构上的某些行为，而这些行为在 CHERI 上并不成立。我们还讨论了相应的变通方案及其影响。

我们通过将这些变通方案应用于 CRuby 移植版本，并调查此前案例研究中的代码库，验证了这些方案的有效性。

这项工作丰富了关于在 CHERI 上开发和移植虚拟机的知识，并将有助于指导构建更安全虚拟机的工作。

ACM CCS2012软件及其工程 → 虚拟机； 软件安全；**关键词** CHERI、移植、Ruby、虚拟机、未定义行为

《编程的艺术、科学与工程》

提交日期 2025年10月2日**发布日期** 2026年2月15日**DOI**
10.22152/programming-journal.org/2026/11/2

© Hanhaotian Liu、Tetsuro Yamazaki 和 Tomoharu Ugawa 本作品采用“知识共享署名 4.0 国际许可协议”授权。发表于《编程的艺术、科学与工程》，第 11 卷第 1 期，2026 年，第 2 篇文章；共 26 页。

1 引言

软件安全至关重要，长期以来一直是计算机科学研究的课题，尤其是在防止恶意方利用漏洞方面 [24, 26, 31]。内存漏洞仍是此类攻击的主要目标。确保安全性对所有软件都至关重要，对编程语言的虚拟机（VM）尤为如此。它们为其上运行的高级语言提供支持，而这些语言构成了当今许多面向消费者的服务和应用程序。虚拟机旨在为用户程序提供更安全的执行环境，但与其他复杂软件系统一样，它们本身也可能存在缺陷 [7, 23]。这些缺陷可能会损害其所支持的高级语言的安全保证，包括内存安全。

由于虚拟机的特性，维护虚拟机的安全性颇具挑战。虚拟机本身十分复杂，代码库规模庞大，因而带来了广泛的攻击面。因此，很难识别并消除所有潜在漏洞。另一方面，攻击手段也在不断演变，进一步增加了确保安全性的难度。

CHERI（能力硬件增强型 RISC 指令） [28, 29] 有望防止虚拟机实现中的内存安全问题。它是一种指令集架构（ISA）扩展，引入了基于能力的安全模型，并已在 RISC-V 和 ARM 的 Morello 中实现。CHERI 以不可伪造、由硬件强制实施的能力取代以地址表示的传统指针，从而实现细粒度内存保护，解决空间内存安全和时间内存安全问题。

然而，开发虚拟机并将其移植到 CHERI 并非易事，其中存在许多细微的陷阱。即使是移植轻量级解释器，也会面临诸多挑战，MicroPython 的移植工作 [23]便说明了这一点。主要困难源于 C 语言中的未定义行为，在 CHERI 环境下，其表现往往不同于虚拟机开发者的预期。例如，保守式垃圾回收（GC）依赖于一些在形式上属于未定义、但在传统系统上确实成立的行为。尽管 CHERI 编程指南 [30] 警告了相关陷阱并提供了变通方案，但其范围面向通用软件，并且也承认移植语言运行时会带来非平凡的挑战。此前已有多项工作移植了 MicroPython [23], 、JavaScriptCore [16],和 Rust [17],，但它们并未聚焦于这些陷阱。

为了探究虚拟机特有的陷阱，我们将 CRuby 移植到 CHERI 上作为案例研究，并调查了以往的工作。CRuby 是一个完全以 C语言 编写的复杂虚拟机。在移植过程中，我们专注于运行时系统和解释器，不包括 JIT编译器。我们也未纳入对外部函数接口（FFI）的支持。尽管外部函数接口被认为是安全问题的来源，但我们将其排除在外，因为可以利用 CHERI 的隔离分区功能隔离外部函数的执行，正如 Chisnall 等人为 Java 本地接口提出的方案 [11]。目前，我们移植的 CRuby 已通过 CRuby 仓库中共计 34,046 个测试用例中的 29,609 个。

本文对 CHERI 上虚拟机特有的问题进行分类。对于每个类别，我们描述具有代表性的问题，并探讨其成因以及可能的变通方案。尽管以往的案例研究报告了与编译器相关的问题和 Rust 特有问题，本文重点关注运行时系统中的问题。我们的研究问题如下：

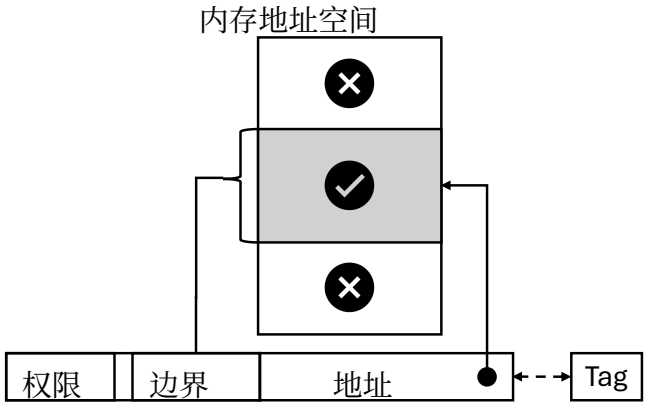


图1 CHERI 中的能力。

研究问题 1 在开发和移植到 CHERI 的过程中，编程语言虚拟机有哪些特有的陷阱?

研究问题 2 这些陷阱有哪些可能的变通方案？它们会产生什么影响？由于我们专注于移植虚拟机本身，因此不会进一步利用 CHERI 的特性来增强虚拟机安全性，例如为堆对象设置紧致边界。这留待未来工作完成。

2 CHERI

我们将 CRuby 移植到了 CHERI 架构。本节将介绍 CHERI 的特性。CHERI 提供两种模式：纯能力模式 和混合模式。由于我们的移植目标是纯能力模式，因此本文始终假设使用纯能力模式。

2.1 CHERI 架构

CHERI 是一种 ISA 扩展，通过为传统硬件增添架构能力，实现细粒度内存安全等安全特性。它用能力取代传统架构中的指针，并引入标记内存，以跟踪地址空间中能力的有效性。

能力是计算机科学中的经典概念，代表一种不可伪造的权威凭证 [14]。在 CHERI 中，能力被实现为一种双字胖指针，其中包含一个整数地址和元数据，包括边界和权限（如图1所示）。能力的地址部分类似于传统系统中的普通指针。压缩边界定义了该能力获准访问的范围，而权限则控制其在内存中授权区域内可以执行哪些操作。每个能力还关联一个1位有效性标签。有效性标签用于追踪该能力是否有效，能否执行前文所述的操作。

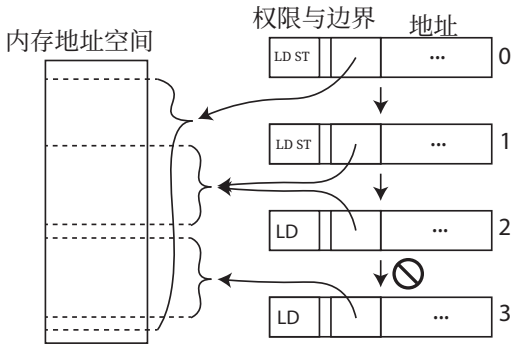


图 2 能力单调性。

单调性 单调性是能力的一项关键安全属性。它确保在派生过程中，能力的权限和边界不会增加。只有在能力被原样复制，或其权限和/或边界被修改得更加受限时，才会保留有效性标签。

图 2 展示了单调性属性的一个示例。在该示例中，箭头表示能力的派生。能力 0 对其指向的内存区域具有加载与存储权限（LD/ST）。能力 1 由能力 0派生而来。它与能力 0 具有相同的权限，但边界较小，因此有效且可以构造。能力 2 也有效，因为它由能力1 派生而来，具有相同的边界，但权限少于能力 1。然而，能力3 无法构造，因为它试图具有比其派生来源能力 2 更大的边界。

密封 密封是一种使能力不可变且不可解引用的属性。密封能力不应被修改。违反密封属性所导致的后果取决于 CHERI 架构的语义和实现。可能会导致标签清除或触发硬件异常。

能力密封的一种形式是密封入口能力。密封入口能力是指向包含执行权限的程序代码且已封存的能力。它们用于增强控制流鲁棒性 [1] ，以防止对这些能力进行不当操纵和使用。每当 CHERI-RISC-V 将程序计数器中的值存储到用于保存返回地址的链接寄存器时，就会创建密封入口能力。函数指针也是密封入口能力。

2.2 CHERI C/C++ 运行时环境

CHERI团队已在硬件实现之上开发了完整的软件栈，以支持在CHERI平台 [9]上使用C语言/C++语言进行软件开发。它包括基于FreeBSD的能力感知型CheriBSD [10] 操作系统，以及实现CHERI C/C++编程语言的Clang/LLVM工具链 [8] 。

清单 1 对(u)intptr_t变量执行二元运算。

```
1 uintptr_t binop(uintptr_t lhs,
ui
lh
```

CHERI C/C++ [30] 是广泛使用的 C/C++ 语言的方言。它们提供空间和时间内存安全，以及用于访问能力特性的扩展，包括用于操作能力地址的cheri_address_set 。此外，取地址运算符“&”会生成一个指向操作数的窄能力。

(u)intptr_tand ptraddr_t Types (u)intptr_t 整数类型用于在 C/C++中存储指针。它们也以能力的形式实现，因此大小是传统架构上的两倍（128 位）。为了只存储能力的地址部分，CHERI C/C++ 提供了新的ptraddr_t 类型。

二元运算中的元数据继承由于(u)intptr_t 类型是能力，因此包含元数据。对于二元运算，如果只有一个操作数是(u)intptr_t 变量，该(u)intptr_t 变量的元数据将由结果继承。然而，当两个操作数都是(u)intptr_t 变量时，结果的元数据便不明确。在这种情况下，设计规定由左操作数的元数据传递给结果。清单 1 展示了一个示例，其中两个 uintptr_t 变量相加。结果res 将继承左操作数lhs的元数据。

内存管理在 CHERI C/C++ 中，malloc 返回的能力，其边界仅覆盖已分配的内存区域。撤销能力是指从程序中移除该能力的所有副本及其所有派生项。CHERI C/C++ 中的 free函数经过改造，支持撤销。释放某项能力后，对应的内存区域会被标记为隔离区，在执行周期性撤销之前无法再次分配。撤销完成后，所有指向隔离区内存区域的能力都会失效。这提供了时间内存安全性，可防范释放后重新分配使用漏洞。

静态内存分配的边界也只会覆盖对象所需的内存区域。例如，如果静态分配了一个 64 位整数，那么表示其地址的能力，其边界将仅覆盖它占用的 8 字节。

3 移植 CRuby

在本研究中，我们将 CRuby [2] 移植到CHERI，以此作为案例研究，探究开发与移植虚拟机中的陷阱。CRuby 是 Ruby 的参考实现

解释器自1995年首次发布以来，一直被用于执行各类应用程序，例如 Web 应用程序。截至2025年，CRuby 的源代码已在 GitHub 上公开，[1] 粗略统计，由 47 万行 .c 文件和 13 万行 .h 文件组成。

我们的移植工作按以下步骤进行。首先，我们使用 CHERI C 编译器编译源代码，并在编译过程中报告错误时逐一修复。我们还运行了 CRuby 仓库提供的测试，并修复测试失败的问题。我们运行了basictest、bootstraptest和test目录中的所有测试。目前，我们的实现通过了basictest中的全部 894 个测试用例、bootstraptest中的 1,987 个测试用例中的 1,979 个，以及test中的 31,165 个测试用例中的 26,736 个。我们尚未移植 CRuby 的即时编译器和外部函数接口（FFI）。我们移植的 CRuby 版本是 3.4.1。对于 CHERI，我们使用了仿真 CHERI-RISC-V。我们在 CRuby 代码库中修改的代码总计 464 行，分布在 40 个文件中，约占整个代码库的 0.077%。

3.1 设计选择

在移植过程中，除了将在第 4 节讨论的挑战之外，还需要完成常规的移植任务。例如，我们对 CRuby 代码库中用于适配不同平台的宏进行了定义，使其适用于 CHERI-RISC-V。

移植过程中一个值得一提的设计选择是，我们将 VALUE 定义为uintptr_t。在 CRuby 中，VALUE 是一种表示 Ruby 语言中值的数据类型，尽管它在整个 CRuby 代码库中都有使用。它承载着不同类型的信息，包括指向对象的指针、整数和其他数据类型。它还用于各种算术和逻辑运算。在 CRuby 中，VALUE 根据平台定义为某种无符号整数；在虚拟机执行期间，它经常与指针相互转换。我们将 VALUE 定义为 uintptr_t ，因为在 CHERI 中，只有 uintptr_t 是能够携带指针信息的无符号整数类型。这一决定使 VALUE 的大小发生变化，从 64 位增至 128 位。与以往的案例研究相比，将 VALUE 定义为 uintptr_t并非独创。在 MicroPython 的移植过程中，表示 MicroPython 对象的mp_obj_t 类型由 uintptr_t [23]表示。在 JavaScriptCore 的移植过程中，表示 JavaScript 语言中值的 JSValue 类型也由能力表示 [16]。

在我们的实现中，我们没有将存储在VALUEs中的能力边界收紧到单个对象。相反，每个VALUE 的边界都适配于包含该能力所指向对象的堆页。在 CRuby 中，堆由一个或多个不连续的堆页组成，这些内存区域以页为单位进行malloc分配和free释放。标准 CHERI C 库提供的 malloc 函数会为能力设置与malloc分配区域相适配的边界。因此，在当前实现中，每个VALUE都有权限访问同一堆页中的任意位置。通过为每个对象设置紧致边界来增强虚拟机安全性，将留待未来工作完成。

[1] https://github.com/ruby/ruby （访问日期：2025-07-17）。

表 1 问题类别。

类别	CRuby	JavaScriptCore	MicroPython	Rust
无效派生指针	●	-	-	-
解引用歧义指针	●	●	-	-
原地重新分配	-	-	●	-
使用整数类型的填充位	●	●	-	-
修改临时能力引入的编译-由编译器生成	●	●	●	-
非能力类型上的指针算术	●	●	●	●

3.2 失败测试用例

失败测试用例总计有 4,437 个。其中，2,931 个失败源自同一个测试程序，该程序用于测试抽象语法树。这个程序导致测试框架崩溃，因此其中包含的所有测试用例都无法完成。另一个主要的失败原因是缺少“libyaml”库。818 个测试用例依赖于 Psych 模块，而该模块需要此库。

我们尚未确定其余688个失败用例的原因。它们可能包含我们移植过程中尚未解决的陷阱。

4 向CHERI的各类移植中的陷阱

在移植CRuby的过程中，我们发现了若干与CHERI能力模型及其实现相关的问题。我们将这些问题与先前案例研究中报告的问题一并按成因归为若干类别，如表1所示。我们调研了三项案例研究：

- JavaScriptCore [16]：将带有JIT编译器的生产级JavaScript虚拟机移植到CHERI。MicroPython [23]：将面向嵌入式系统的小型Python解释器移植到
- CHERI。Rust [17]：将Rust编译器、运行时和库移植到CHERI。

尽管此前的案例研究报告了与编译器和 Rust 特有问题相关的问题，本文重点关注运行时系统中的问题。表格中的圆圈表示我们在移植过程中发现了该类别的问题，或这些问题已在此前的案例研究中得到报告。这种分类有助于我们了解这些问题的总体性质，并总结出构建和将虚拟机移植到 CHERI 的一般要点。尽管此前的案例研究报告了某些类别中的问题，我们还发现了这些类别中尚未被报告的问题。

大多数问题都源于 CRuby 对 C 语言未定义行为所作的假设。C 标准中存在许多未定义行为 [22]。然而，大多数 C程序并不严格遵循 C 标准 [25]。特别是，虚拟机通常会对未定义行为作出假设，以实现语言

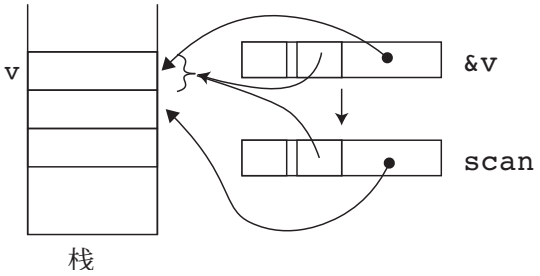


图 3 派生栈顶指针。

清单 2 CRuby 中栈扫描的伪代码。1 void set_stack_

```
enc
p =
NU
*sc
```

高效地实现这些特性。因此，软件失去了可移植性，也无法在 CHERI 上运行。

在本节的其余部分，我们将详细讨论这些类别，并展示其中一些问题的具体示例。我们还将讨论可能的变通方案及其对性能和安全性的影响。

4.1 无效派生指针

虚拟机通常会从为其他用途创建的指针派生出另一个指针。例如，CRuby 会获取栈上局部变量的地址，并派生出一个用于栈扫描的指针。在所有指针都是内存地址的传统架构上，这样做是可行的。然而，在 CHERI 中，这类做法并不合法，因为指针是带有边界的能力。

问题 CRuby 使用保守式垃圾回收。在根扫描期间，保守式垃圾回收需要确定栈的范围。如清单 2 所示，它从局部变量的地址获取栈顶地址。由于调用 set_栈_end 时其栈帧位于栈顶，局部变量 &top 的地址即为栈顶地址，如图 3 所示。栈扫描例程栈_scan 会迭代

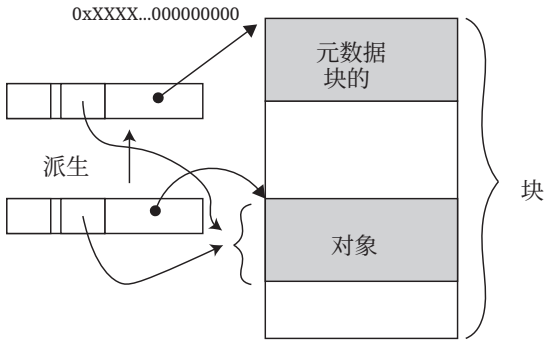


图 4 在块结构堆中派生指针。

从栈顶到栈底，使用以栈顶地址初始化的指针 scan。在此迭代过程中，它递增 scan，从而派生出一个指针。

这段代码在 CHERI 中无效，因为“&”运算符返回的能力，其边界仅覆盖局部变量 top。因此，派生指针 scan 也会继承仅限于 top 的边界。当栈扫描循环在第二次迭代中解引用scan 时，会导致能力边界错误，因为 scan 超出边界。

可能的问题 虚拟机的内存管理器中也可能出现这类问题。例如，在块结构堆 [15]，中，内存管理器有时会将元数据存储在内存块的起始位置。为了访问该块的元数据，内存管理器会如图 4 所示，从指向块中某个对象的指针派生出一个指向块起始位置的指针，而该起始位置按 2^n边界对齐。如果指向对象的指针的边界仅覆盖该对象，则派生出的指向元数据的指针无效。尽管 CRuby 也采用块结构堆，但我们在移植过程中没有将指针边界缩小到单个对象。相反，我们保留覆盖整个块的边界。因此，没有出现这个问题。

变通方法及其影响 我们在移植 CRuby 时采用的解决方案是保留一个超级能力，其边界足够宽，能够覆盖未来可能访问的整个内存区域，例如整个栈。当虚拟机需要创建一个在超级能力所覆盖区域内操作的指针时，可以安全地从该超级能力派生出新指针。在栈扫描的情况下，CHERI C 编译器已经维护了一个覆盖整个栈的栈指针。因此，我们在清单 2 中从栈指针创建 scan 。

总体而言，这种变通方法的一个潜在缺点，涉及在 CHERI 上运行的软件的安全性。传递覆盖大范围内存区域的能力可能会引入漏洞。为缓解这一问题，我们可以按照 Chisnall 等人为 Java 本地接口提出的做法，将需要超级能力的例程隔离到独立分区中 [11]，，从而限制其影响。不过，隔离分区本身会带来开销，因此需要在安全性与性能之间权衡。

要点 从一个指针派生出另一个指针时，我们必须确保原始指针具有足够的边界和权限。如果不够，就必须保留一个具有足够边界和权限的超级能力。

4.2 解引用歧义指针

在 C 标准中，将整数转换为指针类型时，结果由实现定义（C 标准第 6.3.2.3 节 [19]）。转换得到的指针可能无效，对其解引用会导致未定义行为。在 CHERI 中，将非指针值转换为指针不会产生有效的能力，因此对其解引用会导致能力标签故障。不过，保守式垃圾回收会假设栈上所有看起来像指针的值都可能是指针，并尝试对其解引用。此类类似指针的值称为歧义指针。

问题 保守式垃圾回收 [4] 是一种垃圾回收实现技术，无需准确了解指针所在位置。它会扫描栈中看起来像指针的值，并将其作为根来处理。

清单 2 展示了 CRuby 中栈扫描的伪代码。在垃圾回收_标记过程中，它会测试每个按指针大小对齐的值v是否看起来像指针，如果是，就对其解引用。该测试包括检查v 是否具有表示堆中某个对象起始地址的位模式。然而，如果某个整数恰好具有与对象地址相同的位模式，垃圾回收器就会将该整数转换为指针，并尝试对其解引用。在 CHERI 中，这种转换得到的指针没有设置有效性标签，因此解引用尝试会失败。

同样的问题也在移植 JavaScriptCore 的案例研究中有所报告（ [16] 中的第 4.1.2 节）。JavaScriptCore 也会保守地扫描栈；如果不修复此问题，就会在垃圾回收期间导致标签违规。

变通方法及其影响 在 CHERI 中，通过查看关联的有效性标签，我们可以准确判断内存位置中的值是否为指针。针对这一问题的变通方法是，检查要解引用的歧义指针的有效性标签是否已设置，而不是采用基于位模式的测试。这种变通方法反而可能提升性能，因为它让我们能够避免标记那些只有在解引用类似指针的整数时才可达的死对象。此类类似指针的整数确实存在；在运行 basictest 测试套件期间，我们观察到有 350 个类似指针的整数通过了 CRuby 垃圾回收在标记前用于筛除非指针的检查。Wang 等人 [27] 还报告称，在他们为 Ruby 实现的复制式垃圾回收中，需要固定栈指针所指向的对象，以避免错误地更新类似指针的整数。利用有效性标签进行精确扫描，也为实现无需固定对象的复制式垃圾回收创造了机会。

要点 CHERI 中的保守式垃圾回收在解引用歧义指针之前，必须先检查关联的有效性标签。

列表3 MicroPython 中解析器部分代码的伪代码。1 void* parser_

```
allo
cur
new
new
NU
chu
12
chu
size
ret
```

4.3 原地重新分配

在 C 标准中，realloc 函数会释放作为参数传入的指针所指向的旧对象（C 标准第 7.20.3.4 节 [19]）。因此，从 realloc 返回后，对该指针进行操作的行为未定义。即使 realloc 返回的指针碰巧指向相同地址，情况仍然如此。事实上，在 CHERI 中，realloc 返回的能力的边界可能与传递给 realloc 的能力不同。相关指令见《CHERI C/C++ 编程指南》（第 4.3.2 节 [30]）。

自定义分配器也存在这种情况。虚拟机通常会使用自定义分配器，以判断原地重新分配是否成功，甚至在特定用例中保证原地重新分配成功。使用此类自定义分配器的客户端代码可能会在原地重新分配成功时继续使用原始指针。如果自定义分配器为其返回的指针指定了紧致边界，这种做法在 CHERI 中就行不通，因为在扩展重新分配的情况下，旧指针的边界无法覆盖扩展后的区域。

问题 这一情况见于 MicroPython 的移植（ [23] 中的第 4 节）。在解析和字符串驻留过程中，它会从垃圾回收器中分配一个内存块，并将其划分给实际对象。代码清单 3 展示了解析器的伪代码，其中CHERI 纯能力 是此问题的修复方案。当当前 __ _ _ __块用尽时，它会尝试使用m_renew_maybe 对一个块 进行原地重新分配，如图 5 所示。如果原地重新分配成功，m_renew_maybe 会返回

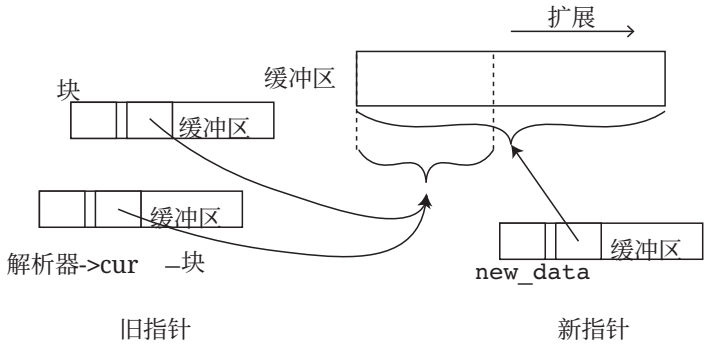


图 5 MicroPython 中的原地重新分配。

新指针。在原版 MicroPython 中，该指针与原始指针相同。因此，当返回的新_数据不为 NULL 时，原版 MicroPython 会将其丢弃，并继续使用原始指针块。然而，在 CHERI 移植版中，返回的指针不再与原始指针相同，而是具有更宽的边界。因此，使用解析器_分配 返回的指针写入扩展区域，会导致能力边界故障。

变通方法及其影响 如代码清单 3 所示，MicroPython 的 CHERI 移植会将返回的指针重新赋值给所有保存旧指针的位置。在本例中，这些位置是 解析器->当前_块 和局部变量 块。请注意，之前从 解析器_分配 返回的所有指针都无需更新，即使它们指向当前块的中间位置，因为扩展区域不会通过这些指针访问。

一般情况下，旧指针可能存储在许多位置，有时很难静态枚举这些位置。在这种情况下，我们可能需要一种特殊机制来跟踪这些位置。另一种可能的变通方法是通过间接寻址访问之后可能被重新分配的内存区域。这两种方法都会带来显著的性能开销。

最好的变通方法是不依赖原地重新分配，即使使用自定义分配器也是如此。理论上，原地重新分配只是一种可由分配器自行决定是否采用的优化。客户端代码不应依赖它，即使从模块化的角度来看也是如此。虽然可以设计一种 API，在传统架构上保证原地重新分配，同时允许客户端代码继续使用原始指针，但这在 CHERI 上是不可能的。

要点 为 CHERI 开发虚拟机时，我们应避免依赖原地重新分配。

4.4 使用整数类型的填充位

虚拟机通常会将较小的值打包到整数中，例如位图，或通过显式位操作计算并手动打包的位域。虚拟机还会将较小整数的数组视为较大整数的数组，以利用数据并行性。在这种情况下，程序员

有时会假设整数值的所有位都可用。然而，有些位可能是填充位 （C 标准第 6.2.6.2 节 [19]）。填充位的值未指定，也可能受到特定于机器的限制。在 CHERI C 中，整数类型之一 (u)intptr_t 是以能力实现的，其高 64 位存放元数据，例如边界。(u)intptr_t 中对应的位是填充位，只有 (u)intptr_t的低 64 位可用于表示整数值。

此外，在这些情况下经常使用的移位操作，如果移位量大于或等于类型的宽度（即不包括填充位在内的位数），其结果是未定义的（C 标准第 6.5.7 节 [19]）。在 CHERI C 中，我们观察到，如果移位量为 64 或更大，对(u)intptr_t值执行左移操作会产生随机值。[2]

问题 1 CRuby 使用位图来标记垃圾回收中的存活对象。位图声明为uintptr_t数组。要定位位图中的第i位，CRuby 会计算索引 $=\lfloor i/S \rfloor$和位偏移量 $=i$ 模S，其中S 是uintptr_t中的位数。通过使用索引 和位偏移量，它按以下方式设置第i位：

```
位图[索引] |= 1 << 位偏移量;
```

这里，S 由sizeof(uintptr_t)计算得出，其中包含填充位。因此，位偏移 可能大于或等于 uintptr_t的宽度（64），在这种情况下，对 1 左移位偏移 位会失败。即使左移成功并得到了正确的值（该值大于或等于 2^{64}, ），按位或操作的结果仍未定义。在 CHERI 中，高半部分的位不会改变。

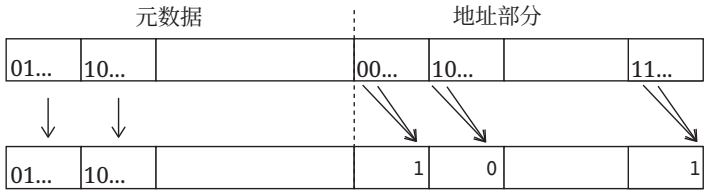
问题2 CRuby 将隐藏类 [6], 的标识符shape_id嵌入到对象头的高位中，而对象头的类型是能力类型 VALUE。因此，以下用于设置shape_id 的代码无法正常工作：

```
#define SHAPE_FLAG_SHIFT ((SIZEOF_VALUE * 8) - SHAPE_ID_NUM_BITS)
RBASIC(obj)->flags |= ((VALUE)(shape_id)  <<SHAPE_FLAG_SHIFT);
```

在移植 JavaScriptCore 时也报告了相关问题 [16]。

问题 3 CRuby 统计字符串中 UTF-8 前导字节的数量时，会利用数据并行。清单 4 展示了用于统计数量的函数伪代码。该函数统计位置 s 中最高有效位为 0 或次高有效位为 1的字节数；s 的类型为 uintptr_t。该函数的调用方使用 uintptr_t 指针遍历字符串，每次处理 sizeof(uintptr_t) 个字节。然而，在 CHERI 中，uintptr_t 的位有一半是填充位。因此，如图 6 所示，该函数只会处理一半的位，并返回错误的计数结果。

[2]如果编译器发现移位量是一个较大的常量，就会消除移位指令，同时继续使用被消除指令的输出寄存器作为移位结果。



■ **图 6** UTF-8 前导字节并行计数中的失败。

■ **清单 4** 统计 UTF-8 前导字节的伪代码。

```
计数_utf8_前导_字节_使用_字(uintptr_t* s) { uintptr_t d  = *s;
```

变通方法及其影响 解决方案是对这些用途使用精确宽度整数类型，例如uint64_t。这类类型规定了确定的宽度，且没有填充位（C 标准 [19] 第 7.18.1.1 节）。这种变通方法不会带来性能开销。

要点 对于位图以及通过显式位操作计算出的手动打包位域，我们应使用精确宽度整数类型。

4.5 修改编译器引入的临时能力

如第 2.1 段所述，CHERI 中的能力可以封存，以防止其被修改。例如，栈上的返回地址已封存。任何试图修改密封能力的行为都会导致能力已封存 故障。

即使程序员没有显式修改能力，CHERI C 编译器也可能引入能力修改。在 CHERI C 中，如果二元运算的一个操作数类型为(u)intptr_t，则运算结果的值类型为(u)intptr_t。由于 CHERI C 中的(u)intptr_t 是通过能力实现的，结果会继承类型为(u)intptr_t的操作数的元数据。为此，CHERI C 编译器会生成代码，复制(u)intptr_t 操作数，并将其地址部分替换为二元运算的结果。如果(u)intptr_t 操作数由密封能力转换而来，则此二元运算会导致能力已封存 故障。

问题1 当 CRuby 在发生错误时生成回溯，会使用 libunwind 收集返回地址。收集到的返回地址是已封存的入口能力。CRuby 会对它们进行操作，以查找包含这些返回地址的函数符号。列表 5 展示了用于查找包含地址 trace_地址 的函数符号的伪代码，图 7 则对此进行了说明。对于每个（函数）符号，它都会计算该符号的地址 源地址。然后，它会计算 d，即距离

列表 5 查找包含 trace_地址的函数符号。1 for (j = 0; j < 符号表_

```
cou
(ui
地址
地址
10
```

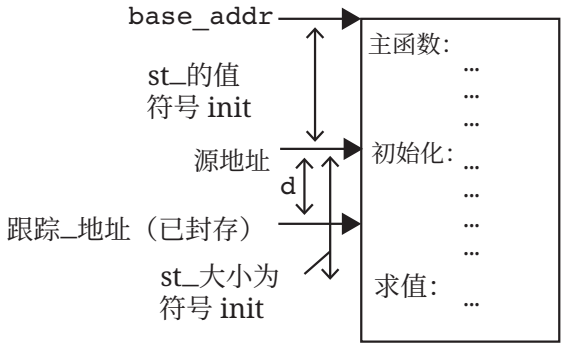


图 7 查找跟踪_地址 （列表 5 的示意图）。此示例中的函数为init 。

列表 6 哈希函数。

```
1 st_numhash(st_data_tn)2 {3 枚举{s1 = 11, s2 = 3};
4
```

位于 跟踪_地址 和 源地址之间。后者的计算试图修改已封存的入口能力跟踪_地址，并导致 能力已封存故障。

问题 2 CRuby 支持直接线程代码解释器 [3]。线程代码解释器的指令序列并非由指令 ID（例如字节码编号）组成的数组，而是由运行时例程地址组成的数组，每个例程负责解释一条字节码指令，后面紧跟其操作数。不过，虚拟机有时出于调试等原因，需要获取当前指令的 ID。因此，CRuby 会在初始化期间创建一个哈希表，用于查找字节码信息。哈希表的键是运行时例程的地址，也就是密封入口能力。因此，列表 6 中的哈希函数会尝试修改已封存的入口能力，从而导致 能力已封存故障。

■列表 7 测试 *VALUE* 中的位。 *1 int RB_IMMEDIATE_P(VALUE v) { 2*

re
0x
IM
对

问题3 在 CRuby 中，VALUE 类型的低三位是标记位；如果 VALUE 是指针，这些位均为零。清单 7 中的 RB_IMMEDIATE_P 函数会检查这些位，以判断给定值 v是否为立即数（而非指针）。由于我们将 VALUE 定义为 uintptr_t，v 是一个能力。因此，测试中的按位与操作会更新 v 的地址部分，从而创建一个继承 v元数据的临时能力。随后会检查这个临时能力的地址部分是否为零。请注意，使用 -O1 或更高级别的优化进行编译时，不会创建这个临时能力。

保守式垃圾回收会调用 RB IMMEDIATE P 函数，检查栈上的每个值_ _ 看起来是否像一个指针。因此，栈上的返回地址也会传递给 RB_IMMEDIATE_P。然而，返回地址是已封存的。因此，按位与操作无法更新 v的地址部分。请注意，第4.2节中用于解引用歧义指针的变通方法无法解决这个问题，因为返回地址是有效的能力。

变通方法及其影响 我们在执行操作之前，将 (u)intptr_t 操作数转换为非能力类型，例如uint64_t ，如下所示：

```
return (((uint64_t) v) & RUBY_IMMEDIATE_MASK) != 0;
```

这种变通方法可以防止编译器引入临时能力。虽然操作结果无法转换回指针，但这种变通方法是有效的，因为操作结果只用作整数。

这种变通方法不会带来任何性能开销。相反，它消除了创建临时能力所需的不必要指令。这是因为编译后的代码会先提取操作数的地址部分，并将其视为整数类型来执行操作。随后，它会更新操作数的地址部分以创建临时能力。然而，在这些问题中，这个临时能力并未被用作能力，只有其地址部分被用作整数。

对于需要将结果转换回指针的情况，此变通方法不起作用。不过，在这种情况下，如果操作数是密封能力，则不应允许执行这些操作，因为这些操作试图从密封能力创建派生指针。

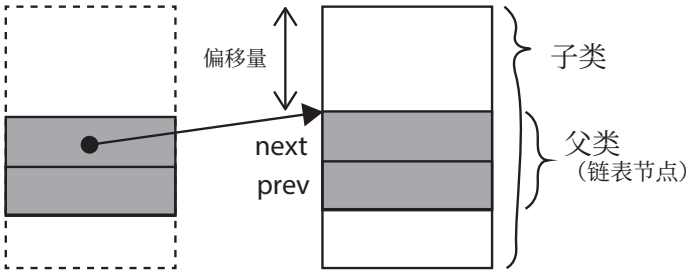


图 8 C语言中的类继承和向下转型。

请注意，在较新版本的 CHERI 中，[3] 本节中的问题不会导致运行时异常，因为修改密封能力地址的语义已改为使该能力失效。不过，为避免产生不必要的临时能力，此变通方法仍然有效。

相关问题 如果二元运算的两个操作数都是能力，例如 (u)intptr_t，CHERI C 编译器会发出警告。这是因为无法明确结果应从哪个操作数继承元数据。MicroPython 将 (u)intptr_t转换为整数类型，以抑制这些编译器警告 [23]。

要点 对 (u)intptr_t 执行运算以获得不一定是 (u)intptr_t 的整数结果时，应在运算前将 (u)intptr_t 操作数强制转换为结果类型。

4.6 非能力类型上的指针算术

指针算术在虚拟机实现中很常见。通常，指针会被强制转换为整数，对该整数进行计算后，再将其转换回指针。根据 C 标准，此操作的结果由实现定义（C 标准第 6.3.2.3 节 [19]）。在 CHERI 中，只有当整数类型为 (u)intptr_t，且原始指针的边界涵盖新地址时，此操作才有效。这是合理的，因为 (u)intptr_t 是唯一一种能保证在不做修改的情况下强制转换回来时恢复原始指针的整数类型。不过，程序员有时会使用为其他用途定义的整数类型，例如 size_t。

问题 在 C语言 中，通过将超类实例嵌入子类实例来模拟类继承，是一种常见的编程惯用法。一个典型的用例是将包含前驱 和后继 成员的链表节点结构嵌入子类实例中，如图 8 所示。子类实例作为一系列此类链表节点进行维护，并使用通用的链表操作例程进行操作。从链表中取出节点后，会将其向下转型 为原始子类。操作系统中经常可以看到这种编程惯用法 [21]。

[3]其语义在版本9中发生了变化。

■代码清单 8 向下转型。1 static rb thread t *_ _2 thread_sched_waiting_
th
*)

CRuby 的线程管理器中也使用了相同的惯用写法。CRuby 的线程结构rb_thread_t 包含一个成员sched.waiting_reason。该成员的行为类似于rb_thread_t的父类。清单 8 展示了向下转型的伪代码，用于根据给定的w（即成员sched.waiting_reason 的地址）恢复指向rb_thread_t 的指针。宏offsetof 会得到该成员相对于rb_thread_t 起始位置的偏移量。问题在于，这段代码将w 转换为size_t 后进行指针算术。因此，转换回rb_thread_t时会产生无效能力。

MicroPython 和 Rust 中的问题 在 MicroPython [23] 和 Rust编译器 [17]的移植过程中也观察到了类似问题。MicroPython 不会将uintptr_t 转换为通用类型mp_obj_t，而是先将uintptr_t转换为mp_uint_t ，再转换为mp_obj_t。由于mp_uint_t 在 MicroPython 移植中不是能力类型，因此，即使原始的uintptr_t 是从指针转换而来的，所得的mp_obj_t 值也无法用作指针。

在 Rust 中，有一个库提供线程间通信通道，可用于发送和接收整数。若要通过该通道发送指针，该库会先将指针转换为整数再传递，之后再将其转换回指针。在 CHERI 中，接收到的指针无法解引用。

变通方法及其影响 针对这个问题，我们的变通方法是将 size_t 改为uintptr_t。此变通方法会引入额外指令来操作能力，这是生成有效能力所必需的。

要点 在 CHERI 中，(u)intptr_t 应用于指针算术。更一般地说，对整数类型进行指针算术并不可移植。

4.7 CHERI 的问题

最后一类问题与 CHERI 硬件及其软件栈有关。CHERI 相对较新，目前仍在积极研究和开发中。与任何复杂的

表 2 运行基准测试所用的环境。

属性	VALUE
CPU 型号	第 12 代 Intel(R) Core(TM) i9-12900
CPU 频率	固定为 2.4 GHz，关闭 Turbo Boost 和动态频率调节
CPU 设置	隔离一个逻辑 CPU 并关闭超线程
操作系统	Ubuntu 22.04
clang 版本	15.0.0

开发中系统一样，它可能存在问题。在本小节中，我们报告移植 CRuby 期间观察到的 CHERI 问题。[4]

mprotect CRuby 在垃圾回收期间使用 mprotect 暂时禁止对页面进行任何访问。不过，我们观察到，一旦移除页面的权限，恢复权限后该页面上的所有能力都会失效。在较新的版本中，[5] 如果在恢复权限时向 mprotect 传递 PROT_CAP 标志，能力就会保持有效。

makecontext CRuby 使用 makecontext 实现协程。下面声明的 makecontext 函数接收一个函数 func 以及任意数量的附加参数，并创建一个包含 func 和这些附加参数的 *context* 。

void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...)

之后，可以使用创建的上下文调用函数 func 并传入这些附加参数。但是，makecontext 会将附加参数以 uint64_t 的形式复制到上下文中。因此，如果某些附加参数是能力，那么它们会失效，并在func 尝试对其进行解引用时导致能力标签故障。

5 性能评估

为了说明我们移植的 Ruby 中变通方案所引入的性能开销，我们比较了原始 Ruby 虚拟机与我们移植的 Ruby 虚拟机的性能。两者均为 3.4.1 版本。

5.1 基准测试

我们使用了官方 Ruby GitHub 仓库中包含的 894 个微基准测试程序。[6] 这些基准测试衡量 Ruby 虚拟机各个方面的性能，包括方法调用、算术运算、类对象操作以及常见算法的应用。我们运行了所有基准测试，但不包括 垃圾回收

[4]我们通过 CHERI-CPU Slack 上的个人通信确认，这些问题属于 CHERI。[5] 25.03 版本。[6] https://github.com/ruby/ruby/tree/master/benchmark （访问日期：2025-11-07）。

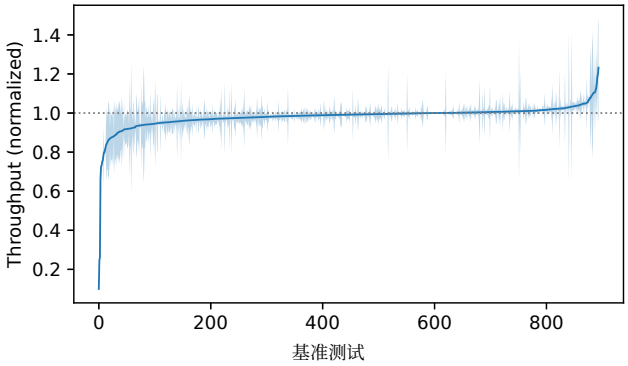


图 9 使用 $\pm 1\sigma$ 时，移植后的 Ruby 相对于原始 Ruby 的吞吐率比。越高 er 越好。

子目录（默认设置中已将其排除）。我们还排除了 3 个基准测试：转储_全部_文件、虚拟机_纤程_分配和虚拟机_纤程_计数，因为原版 Ruby 虚拟机会报错。

5.2 实验设置

我们用于运行基准测试的环境设置如表 2 所示。我们在 x86 架构 CPU 上运行这两个虚拟机。

为了使移植后的 Ruby 能在 x86 架构上编译并运行，我们仅做了最少的修改，并保留了此前为移植所做的修改。由于无法检查指针有效性，因此无法展示垃圾回收的影响，所以两个版本都以保守方式运行垃圾回收。

5.3 评估

图 9 绘出了移植后的 Ruby 虚拟机相对于原始 Ruby 虚拟机的吞吐率，并按各项基准测试进行了归一化。数值越高越好。横轴按每项基准测试 20 次运行所得的平均吞吐率比排序。阴影区域表示 ± 1 标准差。

图中显示，大多数基准测试的吞吐率比接近 1。少数离群值的吞吐率明显较低，几乎比原始 Ruby 低 $10\times$ 。这些离群值对应的基准测试与纤程操作有关。Ruby 中的纤程是轻量级并发原语，可实现协作式多任务处理。Ruby 代码库包含多种纤程实现，可根据底层架构进行优化。在我们移植的 Ruby 中，我们采用了依赖标准库函数的通用实现；而原始 Ruby 使用的是针对 x86 优化、以汇编编写的实现。

移植后的 Ruby 平均运行速度为原版 Ruby 吞吐量的0.982× ，标准差为0.038×。

6 相关工作

先前移植至 CHERI 的虚拟机 MicroPython 是一种用于在微控制器和受限环境中运行的 Python 实现。将 MicroPython [23]移植至 CHERI 描述了整个综合过程，包括从所面临挑战中总结的要点以及对性能的评估。

Harris 等人 [17] 讨论了将 Rust 编程语言移植到 CHERI 所涉及的挑战及解决方案。这项工作涵盖了移植过程的各个方面，包括对编译器和标准库的修改。该案例研究还指出了 Rust 特有问题，例如 Rust 中缺少与 (u)intptr_t 相对应的类型。

将 JavaScriptCore 移植到 CHERI 的工作 [16] 也受到了研究。这项工作评估了将现代语言运行时移植到 CHERI 的可行性，以及源代码在 CHERI 上的兼容性。

未定义行为 在为 CHERI 实现内存安全的 C 时，Chisnall 等人 [12]总结了常见的 C 惯用法，这些惯用法最初会在 CHERI 的内存模型中引发错误。其中一部分依赖于实现定义行为或未定义行为。通过支持这些惯用法中的所有惯用法（除一种外），作者改进了 CHERI 指令集架构，以增强代码兼容性。被排除的惯用法指的是将指针存储在较小尺寸的整数中，这与我们在第 4.6 节提到的问题有关。其余最初存在问题的惯用法与本研究不同。这项工作研究了通用开源应用程序，而我们关注的是虚拟机实现。

关于未定义行为语义的研究包括 LenientC [25], ，这是一种 C 方言，为未定义、未指定和实现定义行为赋予精确语义，从而提升可移植性。它由 Safe Sulong 实现；Safe Sulong 是一个面向 Java 虚拟机、带有动态编译器的解释器。Hathhorn 等人 [18] 提出了 C 的负语义，拒绝包含未定义行为的程序，并提供了一种检测这些行为的工具。

CHERI 上的其他系统软件 除了虚拟机之外，也有将其他类型的系统软件移植到 CHERI 的研究。关于 CheriABI 的论文 [13] 提出了抽象能力 这一概念，用于描述内存访问。为研究这一概念，作者将 FreeBSD 和 PostgreSQL 移植到了 CHERI。适配过程中所做的修改被归纳并划分为 11 个类别。这些类别涵盖了我们遇到的大多数问题，但我们阐述了虚拟机环境中会出现哪些问题。

Jacob 和 Singer [20] 描述了将 BDW 保守式垃圾回收库 [5] 移植到 CHERI 的经验，其动机是确保垃圾回收元数据的安全。这是对将垃圾回收移植到 CHERI 所涉及复杂性的首次分析。他们展示了移植、性能评估以及运行时优化机会方面的挑战。与我们的移植工作一样，他们使用有效性标签，在标记阶段排除非能力值。

7 结论

在这项工作中，我们将 CRuby 移植到 CHERI，并调研了此前的三项案例研究，以识别开发和移植 CHERI 虚拟机时存在的陷阱。我们将这些陷阱分为六类，并考察了相应的变通方案及其影响。大多数陷阱源于 C 语言中的未定义行为；尤其是，虚拟机常用的实现技术和惯用法往往假定传统架构中的行为，而这些行为在 CHERI 上是无效的。例如，保守式垃圾回收依赖于指针和整数可以相互转换；位图和位域将小字段嵌入整数中，假定整数类型的所有位都可用，而在 CHERI 上，(u)intptr_t 是一个能力；此外，在虚拟机中，从为不同目的创建的另一个指针派生指针也很常见。

我们展示了以下应对这些陷阱的方法：维护具有宽边界的能力，在垃圾回收中检查有效性标签，避免依赖原地重新分配，以及使用适当的整数类型。尽管某些变通方案会引入性能开销或削弱CHERI的安全保证，CHERI也能够实现更准确的垃圾回收。

致谢 本工作的部分内容得到了日本学术振兴会通过日本学术振兴会科研费项目（编号JP23K24822）的资助。我们还要感谢格拉斯哥大学的Jeremy Singer和Kai Feng、CHERI团队以及Ruby社区在本项目期间给予的支持。

参考文献

[1] Martín Abadi、Mihai Budiu、Úlfar Erlingsson 和 Jay Ligatti。“控制流完整性：原理、实现与应用”。载于：第*12*届*ACM*计算机与通信安全会议论文集。CCS '05。美国弗吉尼亚州亚历山德里亚：美国计算机协会，2005年，第340–353页。ISBN：1595932267。DOI：10.1145/1102120.1102165。[2] 关于 *Ruby*。网址：https://www.ruby-lang.org/en/about/。（访问日期：2025-07-17）。[3] James R. Bell。“线程代码”。载于：《*ACM*通讯》16.6（1973年6月），第370–372页。ISSN：0001-0782。DOI：10.1145/362248.362270。[4] Hans-Juergen Boehm。“空间高效的保守式垃圾回收”。载于：《*SIGPLAN*通知》28.6（1993年6月），第197–206页。ISSN：0362-1340。DOI：10.1145/173262.155109。[5] Hans-Juergen Boehm和Mark Weiser。“非协作环境中的垃圾回收”。载于：《软件：实践与经验》18.9（1988年9月），第807–820页。ISSN：0038-0644。DOI：10.1002/spe.4380180902。[6] Craig Chambers 和 David Ungar。“定制化：面向 SELF（一种动态类型的面向对象编程语言）的优化编译器技术”。载于：《*SIGPLAN*通知》24.7（1989年6月），第146–160页。ISSN：0362-1340。DOI：10.1145/74818.74831。

[7] Yuting Chen、Ting Su 和 Zhendong Su。“Java 虚拟机实现的深度差分测试”。载于：*2019 年 IEEE/ACM 第 41 届国际软件工程大会（ICSE）*。2019年，第1257–1268页。DOI：10.1109/ICSE.2019.00127。[8] *CHERI Clang/LLVM 和 LLD*。网址：https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-llvm.html。（访问日期：2025-07-17）。[9] *CHERI 软件栈*。网址：https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-software.html。（访问日期：2025-07-17）。[10] *CheriBSD*。网址：https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheribsd.html。（访问日期：2025-07-17）。[11] David Chisnall、Brooks Davis、Khilan Gudka、David Brazdil、Alexandre Joannou、Jonathan Woodruff、A. Theodore Markettos、J. Edward Maste、Robert Norton、Stacey Son、Michael Roe、Simon W. Moore、Peter G. Neumann、Ben Laurie 和 Robert N. M. Watson。“CHERI JNI：将 Java 安全模型下沉至 C 语言”。载于：第二十二届编程语言与操作系统架构支持国际会议论文集。ASPLOS ’17。中国西安：美国计算机协会，2017年，第569–583页。ISBN：9781450344654。DOI：10.1145/3037697.3037725。[12] 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：为内存安全 C 抽象机提供架构支持”。载于：*SIGARCH*计算机体系结构新闻43.1（2015年3月），第117–130页。ISSN：0163-5964。DOI：10.1145/2786763.2694367。[13] Brooks Davis、Robert N. M. Watson、Alexander Richardson、Peter G. Neumann、Simon W. Moore、John Baldwin、David Chisnall、Jessica Clarke、Nathaniel Wesley Filardo、Khilan Gudka、Alexandre Joannou、Ben Laurie、A. Theodore Markettos、J. Edward Maste、Alfredo Mazzinghi、Edward Tomasz Napierala、Robert M. Norton、Michael Roe、Peter Sewell、Stacey Son 和 Jonathan Woodruff。“CheriABI：在 POSIX C 运行时环境中强制执行有效的指针来源并最大限度降低指针权限”。载于：第二十四届编程语言与操作系统架构支持国际会议论文集。ASPLOS ’19。美国罗得岛州普罗维登斯：美国计算机协会，2019年，第379–393页。ISBN：9781450362405。DOI：10.1145/3297858.3304042。[14] Jack B. Dennis 和 Earl C. Van Horn。“多道程序计算的编程语义”。载于：《*ACM*通讯》9.3（1966年3月），第143–155页。ISSN：0001-0782。DOI：10.1145/365230.365252。[15] David Detlefs、Christine Flood、Steve Heller 和 Tony Printezis。“Garbage-First 垃圾回收”。载于：第四届内存管理国际研讨会论文集。ISMM ’04。加拿大不列颠哥伦比亚省温哥华：美国计算机协会，2004年，第37–48页。ISBN：1581139454。DOI：10.1145/1029873.1029879。

[16] Brett Gutstein。《使用 *CHERI* 能力实现内存安全：安全性分析、语言解释器与堆时间安全》。技术报告 UCAM-CL-TR-975。剑桥大学计算机实验室，2022年11月。doi：10.48456/tr-975。[17] Sarah Harris、Simon Cooksey、Michael Vollmer 和 Mark Batty。“《Rust for Morello：始终开启的内存安全，即使在不安全代码中》”。载于：第*37*届欧洲面向对象编程会议（*ECOOP 2023*）。Karim Ali 和 Guido Salvaneschi 编。第263卷。莱布尼茨国际信息学论文集（LIPIcs）。德国达格施图尔：Schloss Dagstuhl – Leibniz-Zentrum für Informatik，2023年，39:1–39:27。isbn：978-3-95977-281-5。doi：10.4230/LIPIcs.ECOOP.2023.39。[18] Chris Hathhorn、Chucky Ellison 和 Grigore Roşu。“《定义 C 语言的未定义性》”。载于：《*SIGPLAN*通知》50.6（2015年6月），第336–345页。issn：0362-1340。doi：10.1145/2813885.2737979。[19] WG14 ISO/IEC JTC1 SC22。《国际标准 *ISO / IEC 9899 C*编程语言 - *ISO/IEC 9899:1999(TC3)*，委员会草案 *2007-09-07*》。url：https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf。（访问日期：2025-07-17）。[20] Dejice Jacob 和 Jeremy Singer。“《Capability Boehm：使用能力硬件进行垃圾回收的挑战与机遇》”。载于：第*18*届 *ACM SIGPLAN/SIGOPS* 虚拟执行环境国际会议论文集。VEE 2022。瑞士：美国计算机协会，2022年，第81–87页。isbn：9781450392518。doi：10.1145/3516807.3516823。[21] Jakob Koschel、Pietro Borrello、Daniele Cono D’Elia、Herbert Bos 和 Cristiano Giuffrida。“《Uncontained：揭示 Linux内核中的容器混淆》”。载于：第*32*届 *USENIX*安全研讨会（*USENIX Security 23*）。加利福尼亚州阿纳海姆：USENIX协会，2023年8月，第5055–5072页。isbn：978-1-939133-37-3。url：https://www.usenix.org/conference/usenixsecurity23/presentation/koschel。[22] Juneyoung Lee、Yoonseung Kim、Youngju Song、Chung-Kil Hur、Sanjoy Das、David Majnemer、John Regehr 和 Nuno P. Lopes。“《驯服 LLVM 中的未定义行为》”。载于：《*SIGPLAN*通知》52.6（2017年6月），第633–647页。issn：0362-1340。doi：10.1145/3140587.3062343。[23] Duncan Lowther、Dejice Jacob 和 Jeremy Singer。“《Morello MicroPython：适用于 CHERI 的 Python解释器》”。载于：第*20*届 *ACM SIGPLAN*托管编程语言与运行时国际会议论文集。MPLR 2023。葡萄牙卡斯凯什：美国计算机协会，2023年，第62–69页。isbn：9798400703805。doi：10.1145/3617651.3622991。[24] Santosh Nagarakatte、Jianzhou Zhao、Milo M. K. Martin 和 Steve Zdancewic。“《SoftBound：适用于 C语言的高度兼容且完备的空间内存安全》”。载于：第*30*届 *ACM SIGPLAN*编程语言设计与实现会议论文集。PLDI ’09。爱尔兰都柏林：美国计算机协会，2009年，第245–258页。isbn：9781605583921。doi：10.1145/1542476.1542504。

[25] Manuel Rigger、Roland Schatz、Matthias Grimmer 和 Hanspeter Mössenböck。“《Java虚拟机上的宽松C执行：或《我如何学会不再担忧并运行代码》》”。载于：第*14*届托管语言与运行时国际会议论文集。ManLang 2017。捷克共和国布拉格：美国计算机协会，2017年，第35–47页。isbn：9781450353403。doi：10.1145/3132190.3132204。[26] László Szekeres、Mathias Payer、Tao Wei 和 Dawn Song。“《SoK：内存中的永恒战争》”。载于：*2013 IEEE*安全与隐私研讨会。2013年，第48–62页。doi：10.1109/SP.2013.13。[27] Kunshan Wang、Stephen M. Blackburn、Peter Zhu 和 Matthew Valentine-House。“《重新设计 CRuby 的内存管理：实践者报告》”。载于：*2025*年 *ACM SIGPLAN*内存管理国际研讨会论文集。ISMM '25。大韩民国首尔：美国计算机协会，2025年，第109–121页。isbn：9798400716102。doi：10.1145/3735950.3735960。[28] Robert N. M. Watson、Simon W. Moore、Peter Sewell 和 Peter G. Neumann。《*CHERI*简介》。技术报告 UCAM-CL-TR-941。剑桥大学计算机实验室，2019年9月。doi：10.48456/tr-941。[29] Robert N. M. Watson、Peter G. Neumann、Jonathan Woodruff、Michael Roe、Hesham Almatary、Jonathan Anderson、John Baldwin、Graeme Barnes、David Chisnall、Jessica Clarke、Brooks Davis、Lee Eisen、Nathaniel Wesley Filardo、Franz A. Fuchs、Richard Griswethwainte、Alexandre Joannou、Ben Laurie、A. Theodore Markettos、Simon W. Moore、Steven J. Murdoch、Kyndylan Nienhuis、Robert Norton、Alexander Richardson、Peter Rugg、Peter Sewell、Stacey Son 和 Hongyan Xia。《能力硬件增强 *RISC*指令：*CHERI*指令集架构（版本*9*）》。技术报告 UCAM-CL-TR-987。剑桥大学计算机实验室，2023年9月。doi：10.48456/tr-987。[30] Robert N. M. Watson、Alexander Richardson、Brooks Davis、John Baldwin、David Chisnall、Jessica Clarke、Nathaniel Filardo、Simon W. Moore、Edward Napierala、Peter Sewell 和 Peter G. Neumann。《*CHERI C/C++* 编程指南》。技术报告 UCAM-CL-TR-947。剑桥大学计算机实验室，2020年6月。doi：10.48456/tr-947。[31] Bennet Yee、David Sehr、Gregory Dardyk、J. Bradley Chen、Robert Muth、Tavis Ormandy、Shiki Okasaka、Neha Narula 和 Nicholas Fullagar。“《Native Client：用于可移植不可信 x86 本机代码的沙箱》”。载于：*2009*年第*30*届 *IEEE*安全与隐私研讨会。2009年，第79–93页。doi：10.1109/SP.2009.25。

作者简介

Hanhaotian Liuhht@g.ecc.u-tokyo.ac.jp。
https://orcid.org/0009-0006-1223-6987

Tetsuro Yamazakiyamazaki@csg.ci.i.u-tokyo.ac.jp.https://orcid.org/0000-0002-2065-5608

Tomoharu Ugawatugawa@acm.org。
https://orcid.org/0000-0002-3849-8639