The seL4® Foundation

https://sel4.systems/Foundation

# The seL4 Microkernel
# An Introduction

Gernot Heiser
gernot@sel4.systems
Revision 1.2 of 2020-06-10

# seL4 微内核简介

## seL4 白皮书翻译

作者：刘崇　译
组织：中国科学院计算技术研究所
时间：2022 年 7 月 19 日
版本：1.0

# Introduction

本白皮书介绍和概述了 seL4。我们解释了 seL4 是（和不是）什么，并探讨了它的定义特征。我们解释了是什么让 seL4 成为安全和安全关键系统以及通常嵌入式和网络物理系统的首选操作系统内核。特别是，我们解释了 seL4 的保证故事、其与安全和安全相关的功能，以及它的基准设置性能。我们还讨论了典型的使用场景，包括遗留系统的增量网络改造。

## 关键字

seL4, microkernel, performance seL4，微内核，性能

# 目录

# 第 1 章 seL4 是什么？

## seL4 是一个操作系统微内核

操作系统 (OS) 是控制计算机系统资源并加强安全性的低级系统软件。与应用软件不同，操作系统拥有对处理器的更高特权执行模式（内核模式）的独占访问权，从而可以直接访问硬件。应用程序只能在用户模式下执行，并且只能在操作系统允许的情况下访问硬件。

操作系统微内核是操作系统的最小核心，它将以更高权限执行的代码减少到最低限度。seL4 是可追溯到 1990 年代中期的 L4 微内核系列的成员。（不，seL4 与 seLinux 无关。）

## seL4 也是一个管理程序

seL4 支持可以运行完全成熟的客户操作系统（如 Linux）的虚拟机。根据 seL4 对通信通道的强制执行，来宾及其应用程序可以相互通信，也可以与本机应用程序通信。

在第 2 章中了解更多关于 seL4 是微内核的含义及其用作管理程序的信息。并在第 7 章中了解真实世界的部署场景，包括将安全性改进到遗留系统的方法。

## seL4 被证明是正确的

seL4 带有一个正式的、数学的、机器检查的实现正确性证明，这意味着内核就其规范而言在非常强烈的意义上是 “无错误” 的。事实上，seL4 是世界上第一个在代码级别具有这种证明的操作系统内核 [Klein et al., 2009]。

## seL4 可证明是安全的

除了实现的正确性之外，seL4 还提供了进一步的安全执行证明 [Klein et al., 2014]。他们说，在正确配置的基于 seL4 的系统中，内核保证了机密性、完整性和可用性等经典安全属性。有关这些证明的更多信息，请参见第 3 章。

## seL4 通过功能通过细粒度访问控制提高安全性

能力是访问令牌，它支持对哪个实体可以访问系统中的特定资源进行非常细粒度的控制。它们根据最小权限原则（也称为最小权限原则，POLA）支持强大的安全性。这是高度安全系统的核心设计原则，以 Linux 或 Windows 等主流系统的访问控制方式是不可能实现的。

seL4 仍然是世界上唯一一个既基于能力又经过正式验证的操作系统，因此可以说是世界上最安全的操作系统。更多关于第 4 章的能力。

## seL4 确保时间关键系统的安全性

seL4 是世界上唯一一个对其最坏情况执行时间 (WCET) 进行了完整而合理的分析的 OS 内核（至少在公开文献中）[Blackham et al., 2011, Sewell et al., 2017] . 这意味着，如果内核配置得当，所有内核操作在时间上都是有界的，并且这个界是已知的。这是构建硬实时系统的先决条件，在严格限制的时间段内无法对事件做出反应是灾难性的。

## seL4 是世界上最先进的混合关键操作系统

seL4 为混合关键性实时系统 (MCS) 提供强大的支持，其中必须确保关键活动的及时性，即使它们与在同一平台上执行的可信度较低的代码共存。seL4 通过一个灵活的模型实现了这一点，该模型保留了良好的资源利用率，这与使用严格（和不灵活）时间和空间分区的更成熟的 MCS 操作系统不同 [Lyons et al., 2018]。有关 seL4 的实时和 MCS 支持的更多信息，请参阅第 5 章。

## seL4 是世界上最快的微内核

传统上，系统要么是（某种）安全的，要么是快速的。seL4 是独一无二的，因为它两者兼而有之。seL4 旨在支持广泛的实际用例，无论它们是否对安全（或安全）至关重要，并且要求出色的性能。有关 seL4 性能的更多信息，请参见第 6 章。

## seL4 发音为 “ess-e-ell-four”

不推荐使用发音 “sell-four”。

## 如何阅读本文档

本文档旨在为广大读者所接受。但是，为了完整起见，我们将在某些地方添加一些更深入的技术细节。

这样的细节将用辣椒标记，就像左边的那个一样。如果你看到这个，那么你就知道如果你认为技术描述对你的口味来说太 “辣”，或者你只是对这个级别的细节不感兴趣，你可以安全地跳过标记的段落。只有其他冷淡的段落会假设你已经阅读了它。

技术部分

辣椒出现在章节标题中的地方，例如这里，这表明整个章节是相当技术性的，可以跳过。

# 第 2 章 seL4 是微内核和管理程序，它不是操作系统

## 2.1 宏内核与微内核

为了理解主流操作系统（如 Linux）和微内核（如 seL4）之间的区别，让我们看一下图2.1。

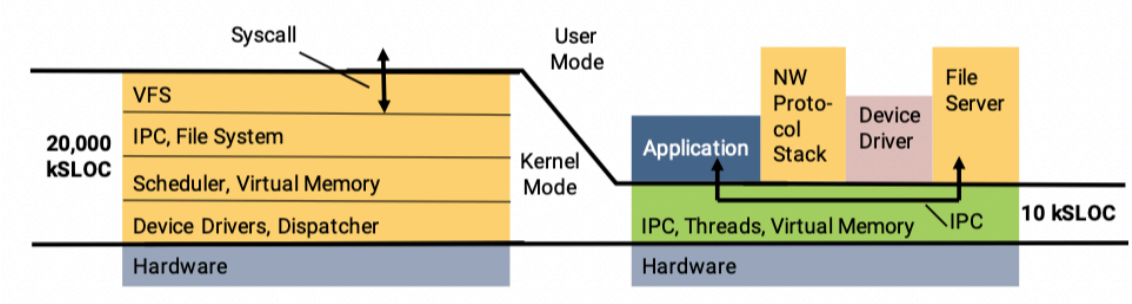


**图 2.1:** 操作系统结构：宏内核（左）与微内核（右）。

左侧展示了系统架构的（相当抽象的）视图，例如 Linux。黄色部分是操作系统内核，它为应用程序提供文件存储和网络等服务。实现这些服务的所有代码都在硬件的特权模式下执行，也称为内核态或管态——这种执行模式可以不受限制地访问和控制系统中的所有资源。相反，应用程序在非特权模式或用户态下运行，并且不能直接访问许多必须通过操作系统访问的硬件资源。操作系统在内部由多个层构成，其中每一层都提供由下面的层实现的抽象。

特权模式代码的问题在于它很危险：如果这里出现任何问题，没有什么可以阻止损害。特别是，如果此代码有一个漏洞，攻击者可以利用该漏洞以特权模式运行攻击者的代码（称为权限提升或任意代码执行攻击），那么攻击者可以对系统做他们想做的事。这些缺陷是我们在主流系统中经历的许多系统攻击的根本问题。

当然，软件错误大多类似生活中的事实，操作系统也不例外。例如，Linux 内核包含大约 2000 万行源代码（20 MSLOC）；我们可以估计它实际上包含数以万计的错误 [Biggs et al., 2018]。这显然是一个巨大的攻击面！这个想法是通过说 Linux 有一个大的可信计算基 (TCB) 来捕捉的，TCB 被定义为整个系统的子集，它必须被信任才能正确运行以确保系统安全。

微内核设计背后的想法是大幅减少 TCB，从而减少攻击面。如图2.1右侧的示意图所示，内核，即系统在特权模式下执行的部分，要小得多。在精心设计的微内核中，例如 seL4，它的源代码量级为一万行（10 kSLOC）。这实际上比 Linux 内核小了三个数量级，并且攻击面相应地缩小了（可能更多，因为错误的密度可能比代码大小的线性增长增加更多）。

显然，就操作系统服务而言，在如此小的代码库中不可能提供相同的功能。事实上，微内核几乎不提供任何服务：它只是硬件的一个薄包装，足以安全地复用硬件资源。微内核主要提供的是隔离，是程序可以在不受其他程序干扰的情况下执行的沙箱。而且，至关重要的是，它提供了一种受保护的过程调用机制，由于历史原因称为 IPC。这允许一个程序安全地调用另一个程序中的函数，其中微内核在程序之间传输函数输入和输出，重要的是，强制使用接口：“远程”（包含在不同的沙箱中）函数只能在一个导出点被调用，并且只能由明确授权的客户端（已被赋予适当的能力，请参阅第 4 章）。

对于 *seL4 IPC* 是什么和不是什么的更深入的解释，我建议阅读我的博客*How to (and how not to) use seL4 IPC*。

微内核系统使用这种方法来提供单片操作系统在内核中实现的服务。在微内核世界中，这些服务只是程序，与应用程序没有什么不同，它们运行在自己的沙箱中，并提供 IPC 接口供应用程序调用。如果服务器受到攻击，那么这种攻击仅限于服务器，它的沙箱会保护系统的其余部分。这与单体案例形成鲜明对比，在宏内核中，操作系统服务的受损会危及整个系统。

这种影响可以量化：我们最近的研究表明，在已知的被归类为关键（即最严重）的 Linux 攻击中，29% 将通过微内核设计完全消除，另外 55% 将得到充分缓解，不再符合关键标准 [Biggs et al., 2018]。

## 2.2 seL4 是微内核，而不是操作系统

seL4 是一个微内核，专为通用性而设计，同时最小化 TCB。它是可以追溯到 90 年代中期的 L4 微内核家族的成员；图2.2显示了 seL4 的出处。它是由我们在 UNSW/NICTA 的团队开发的，现在被称为可信赖系统 (TS)。当时，我们在开发高性能微内核方面拥有 15 年的经验，并且在实际部署方面拥有良好的记录：我们的 OKL4 微内核在数十亿高通蜂窝调制解调器芯片上发货，而我们的 L4 嵌入式内核从零五年左右开始在所有最新 iOS 设备（iPhone 等）的安全飞地上运行。

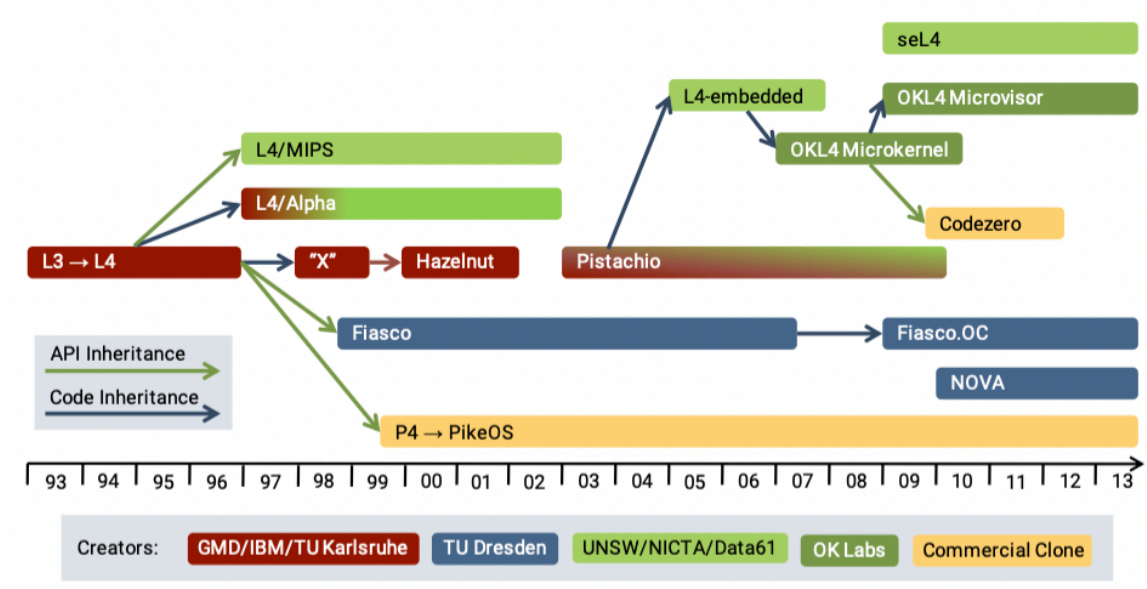


图 2.2: L4 微内核家族树。

作为一个微内核，seL4 不包含任何常见的操作系统服务。此类服务由在用户模式下运行的程序提供。除了上面阐述的巨大优势之外，微内核设计也有缺点：这些组件必须来自某个地方。有些可以从开源操作系统（例如 FreeBSD 或 Linux）移植过来，也可以从头开始编写。但无论如何，这是一项重要的工作。

为了扩大规模，我们需要社区的帮助，而 seL4 基金会是使社区能够为基于 seL4 的系统合作开发或移植此类服务的关键机制。最重要的是设备驱动程序、网络协议栈和文件系统。我们有相当数量的这些，但还需要更多。

一个重要的促成因素是组件框架；它允许开发人员专注于实现服务的代码，并使大部分系统集成自动化。目前有两个主要的 seL4 组件框架，都是开源的：CAmkES 和 Genode。

CAmkES 是一个针对嵌入式和网络物理系统的框架，这些系统通常具有静态架构，这意味着它们由一组定义的组件组成，一旦系统完全启动，这些组件就不会改变。

Genode 在许多方面是一个更强大和更通用的框架，它支持多个微内核，并且已经附带了丰富的服务和设备驱动程序，尤其是对于 x86 平台。可以说它比 CAmkES 使用起来更方便，并且肯定是快速启动复杂系统的方法。但是，Genode 也有缺点：1. 由于它支持多个微内核，并不像 seL4 那样强大，因此 Genode 是基于最小公分母的。特别是，它不能使用 seL4 的所有安全功能。2. 它没有保证的故事。第 3.2 节对此进行了详细介绍。

## 2.3 seL4 也是一个管理程序

seL4 是一个微内核，但它也是一个管理程序：可以在 seL4 上运行虚拟机，并在虚拟机 (VM) 内部运行主流操作系统，例如 Linux。

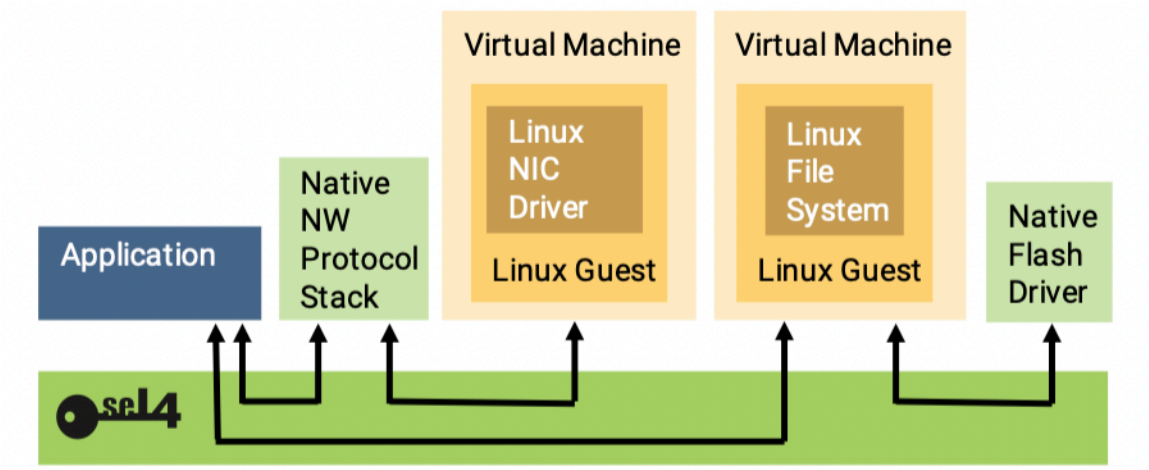


图 2.3: 使用虚拟化将本机操作系统服务与 Linux 提供的服务集成。

这启用了另一种供应系统服务的方法，即让 Linux VM 提供它们。这种设置如图2.3 所示，它显示了如何从多个 Linux 实例中借用一些服务，这些 Linux 实例在不同的 VM 中作为来宾操作系统运行。

在这个例子中，我们提供了两个系统服务：网络和存储。网络由直接在 seL4 上运行的本机协议栈提供，lwIP 或 PicoTCP 是常用的协议栈。我们没有移植网络驱动程序，而是从 Linux 中借用一个，通过运行一个带有精简 Linux 客户机的 VM，该客户机仅具有 NIC 驱动程序。协议栈通过 seL4 提供的通道与 Linux 通信，应用程序同样通过与协议栈通信获得网络服务。请注意，在图中所示的设置中，应用程序没有到 NIC-driver VM 的通道，因此无法直接与其通信，只能通过 NW 堆栈；这是通过 seL4 的基于能力的保护实现的（参见第 4 章）。

为存储服务显示了类似的设置；这次文件系统是在虚拟机中运行的 Linux 系统，而存储驱动程序是本机的。同样，组件之间的通信仅限于所需的最小通道。特别是，应用程序无法与存储驱动程序通信（通过文件系统除外），并且两个 Linux 系统无法相互通信。

*当用作管理程序时，seL4 在适当的管理程序模式下运行（Arm 上的 EL2，x86 上的 Root Ring-0，RISC-V 上的 HS），这是比客户操作系统更高的特权级别。就像作为操作系统内核运行时一样，它只完成必须在特权（管理程序）模式下执行的最少工作，并将其他所有工作留给用户模式。*

*具体来说，这意味着 seL4 执行世界切换，这意味着它会在 VM 的执行时间到时切换虚拟机状态，或者由于某些其他原因必须切换 VM。它还捕获虚拟化异常（英特尔术语中的“VM 退出”）并将它们转发到用户级处理程序，称为虚拟机监视器 (VMM)。然后，VMM 负责执行所需的任何仿真操作。*

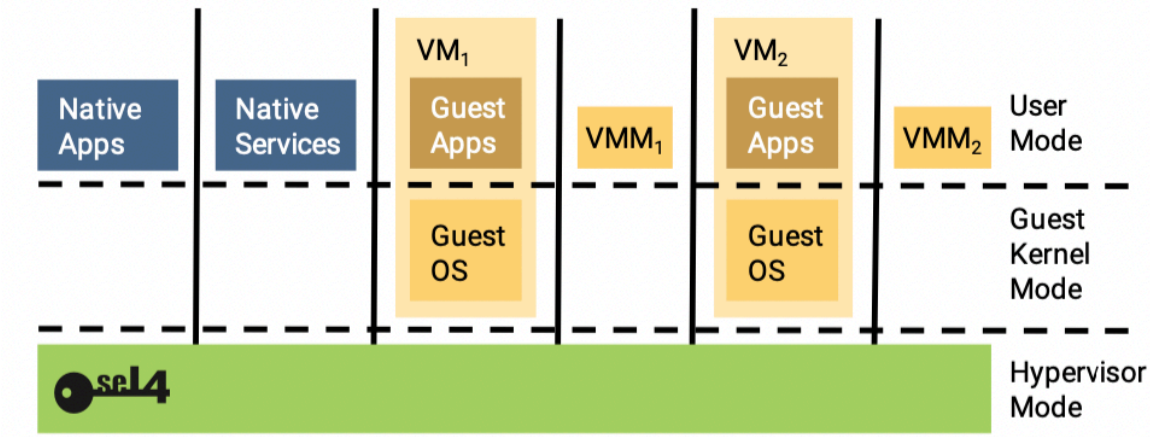


图 2.4: seL4 虚拟化支持与用户模式 VMM。

*每个 VM 都有其 VMM 的私有副本，与来宾操作系统以及其他 VM 隔离，如图2.4所示。这意味着 VMM 无法打破隔离，因此不会比来宾操作系统本身更受信任。特别是，这意味着无需验证 VMM，因为只要客户操作系统*

（通常是 *Linux*）未经过验证，就不会增加真正的保证。

## 2.4 seL4 不是 seLinux

许多人将 seL4 与 seLinux 混淆（可能是因为 seL4 可能被误认为是 seLinux 第 4 版的简写）。事实是 seL4 什么都没有与 seLinux 有关，除了两者都是开源的。它们不共享代码或抽象。seLinux 不是微内核，它是 Linux 内置的安全策略框架。虽然在某些方面比标准 Linux 更安全，但 seLinux 也存在与标准 Linux 相同的问题：巨大的 TCB 和相应的巨大攻击面。换句话说，seLinux 是一个根本不安全的操作系统的附加组件，因此从根本上来说仍然是不安全的。相比之下，seL4 从头开始提供防弹隔离。

简而言之，seLinux 不适合真正的安全关键用途，而 seL4 是为它们设计的。

# 第 3 章 seL4 的验证故事

2009 年，seL4 成为世界上第一个在源代码级别具有机器检查的功能正确性证明的操作系统内核。这个证明当时有 200,000 行证明脚本，是有史以来最大的证明之一（我们后来认为它是第二大的证明）。它表明一个功能正确的操作系统内核是可能的，在此之前一直被认为是不可行的。

从那时起，我们将验证的范围扩展到更高级别的属性，图3.1 显示了证明链，如下所述。重要的是，我们随着内核的不断发展保持了证明：只有在不破坏证明的情况下才允许提交到主线内核源代码，否则也会更新 poofs。这种证明工程也是一个新事物。我们的 seL4 证明构成了迄今为止积极维护的最大证明库。这套证明现在已经增长到超过一百万行，其中大部分是手动编写的，然后是机器检查的。

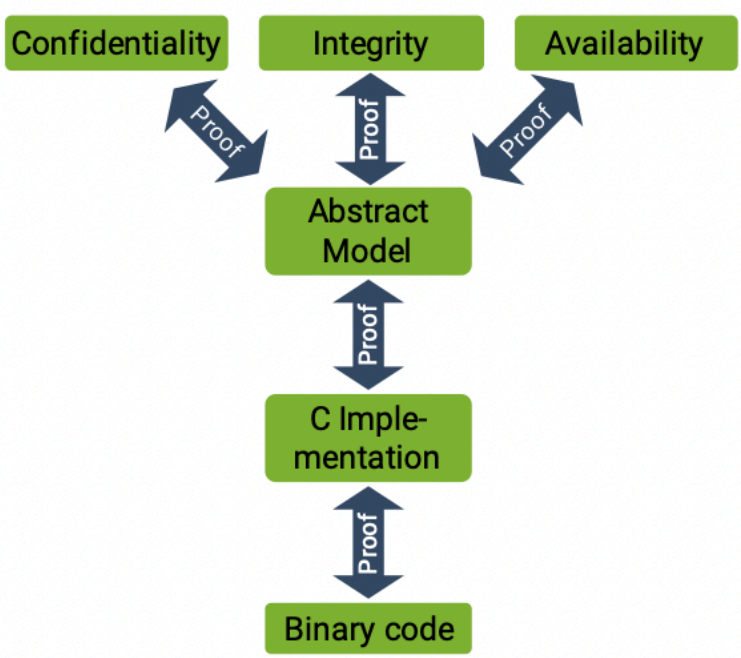


图 3.1: seL4 的证明链。

## 3.1 正确性和安全执行

### 功能正确性

seL4 验证的核心是功能正确性证明，即 C 实现没有实现缺陷。更准确地说，有一个内核功能的正式规范，用称为高阶逻辑 (HOL) 的数学语言表示。这由图中标有抽象模型的框表示。然后，功能正确性证明表明 C 实现是对抽象模型的改进，这意味着 C 代码的可能行为是抽象模型允许的行为的子集。

这种非正式的描述掩盖了很多细节。如果您想知道，这里有一些。

*C 不是形式语言；为了允许在定理证明器（我们使用 Isabelle/HOL）中对 C 程序进行推理，必须将其转换为数学逻辑 (HOL)。这是由用 Isabelle 编写的 C 解析器完成的。解析器定义了 C 程序的语义，并根据这个语义赋予它在 HOL 中的意义。正是这种形式化被我们证明是对数学（抽象）模型的改进。*

*请注意，C 没有官方的数学语义，并且 C 语言的某些部分是出了名的微妙，并且不一定定义得那么好。我们通过将 C 语言的使用限制在定义明确的语言子集来解决这个问题，为此我们有明确的语义。但是，这并不能保证我们对该子集的假设语义与编译器的相同。更多关于下面的内容。*

证明意味着我们想知道的关于内核行为的一切（除了时序）都由抽象规范表达，内核不能以规范不允许的

方式运行。除其他外，这排除了针对操作系统的常见攻击，例如堆栈粉碎、空指针解引用、任何代码注入或控制流劫持等。

## 翻译验证

内核的无错误 C 实现很棒，但仍然让我们受制于 C 编译器。这些编译器（我们使用 GCC）本身就是有缺陷的大型复杂程序。所以我们可以有一个没有错误的内核，它被编译成一个有错误的二进制文件。

在安全关键领域，编译器错误并不是唯一的问题。编译器可能是完全恶意的，其中包含在编译操作系统时自动构建在后门中的特洛伊木马。木马可以扩展为在编译编译器时自动添加，即使编译器是开源的，也几乎无法检测到！Ken Thompson 在他的图灵奖演讲 [Thompson, 1984] 中解释了这种攻击。

为了防止有缺陷或恶意编译器，我们还验证了编译器和链接器生成的可执行二进制文件。具体来说，我们证明二进制是（被证明是正确的）C 代码的正确翻译，因此二进制改进了抽象规范。

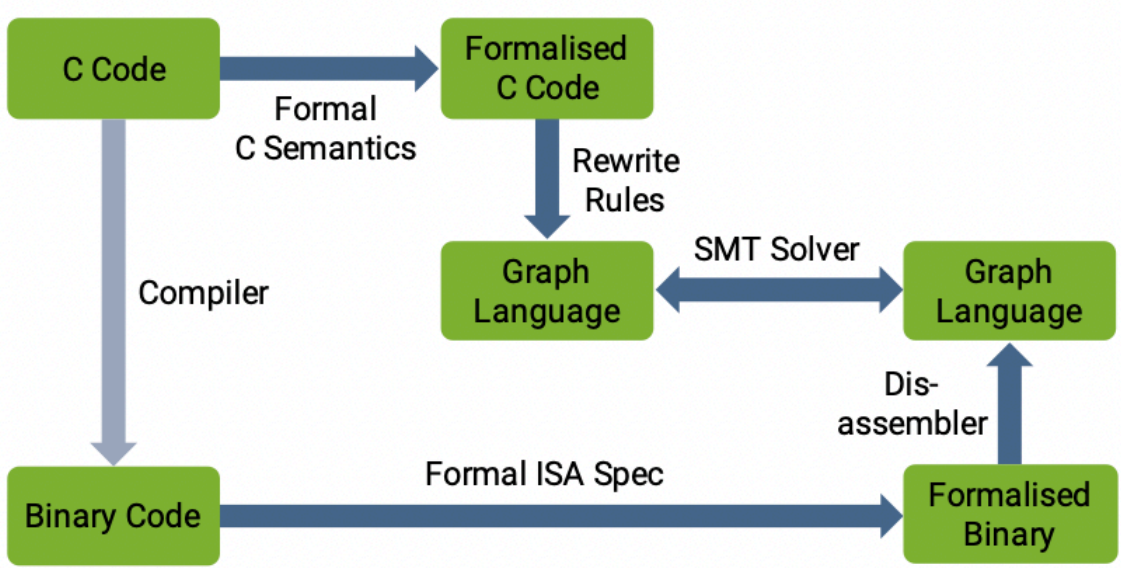


**图 3.2:** 翻译验证证明链。

*与 C 代码的验证不同，此证明不是手动完成的，而是通过自动工具链完成的。它由几个阶段组成，如图 3.2所示。处理器指令集架构 (ISA) 的形式化模型将定理证明器中的二进制形式化；我们使用 RISC-V ISA 的 L3 形式化，以及 Fox 和 Myreen [2010] 的经过广泛测试的 L3 Arm ISA 形式化。*

*然后，用 HOL4 定理证明器编写的反汇编程序将这种低级表示转换为图形语言中的高级表示，该表示基本上表示控制流。这种转换被证明是正确的。*

*通过 Isabelle/HOL 定理证明器中可证明正确的转换，将形式化的 C 程序翻译成相同的图形语言。然后我们有两个程序，在相同的表示中，我们需要证明它们是等价的。这有点棘手，因为编译器应用了许多启发式驱动的转换来优化代码。我们通过对 C 程序的图形语言表示的重写规则应用了许多这样的转换（仍然在定理证明器中，因此可以证明是正确的）。*

*最后我们得到了两个非常相似但不相同的程序，我们需要证明它们具有相同的语义。从理论上讲，这相当于停机问题，因此无法解决。在实践中，编译器所做的决定性足以使问题易于处理。我们通过在多个 SMT 求解器上以小块的形式将程序抛出来做到这一点。如果其中之一可以证明所有对应的部分具有相同的语义，那么我们就知道这两个程序是等价的。*

*还要注意，被证明可以改进抽象规范的 C 程序，以及我们证明与二进制等价的 C 程序是相同的 Isabelle/HOL 形式化。这意味着我们对 C 语义的假设脱离了证明所做的假设。总之，证明不仅表明编译器没有引入错误，而且我们使用的 C 子集的语义与我们的相同。*

### 安全属性

图 3.1 还显示了抽象规范与高级安全属性机密性、完整性和可用性（这些通常称为 CIA 属性）之间的证明。这些声明抽象规范实际上对安全性有用：它们证明在正确配置的系统中，内核将强制执行这些属性。

具体来说，seL4 强制要求：

- **机密性**：seL4 不允许实体在没有明确授予数据读取权限的情况下读取（或以其他方式推断）数据；
- **完整性**：seL4 不允许实体在没有明确授予数据写入权限的情况下修改数据；
- **可用性**：seL4 将不允许一个实体阻止另一个实体对资源的授权使用。

这些证明目前没有捕捉到与时间相关的属性。我们的机密性证明排除了隐蔽存储通道，但目前不排除隐蔽时间通道，这些通道被 *Spectre* 等攻击使用。防止定时通道是我们正在努力的事情 *[Heiser et al., 2019]*。同样，完整性和可用性证明目前不涵盖及时性，但我们的新 *MCS* 模型 *[Lyons et al., 2018]* 旨在涵盖这些方面（参见第 *5.2* 节）。

### 证明假设

所有关于正确性的推理都基于假设，无论推理是正式的，如 seL4，还是非正式的，当有人考虑为什么他们的程序可能是“正确的”时。每个程序都在某些上下文中执行，其正确行为不可避免地取决于对该上下文的一些假设。

机器检查的形式推理的优点之一是它迫使人们明确这些假设。不可能做出未陈述的假设，如果它们依赖于未明确陈述的假设，证明将不会成功。从这个意义上说，形式推理可以防止忘记假设或不清楚假设。这本身就是验证的一大好处。

seL4 的验证做了三个假设：

- **硬件按预期运行**。这应该是显而易见的。内核受底层硬件的支配，如果硬件有问题（或者更糟糕的是，有木马程序），那么无论您运行的是经过验证的 seL4 还是任何未经验证的操作系统，所有的赌注都将落空。验证硬件超出了 seL4 的范围（和 TS 的能力）；其他人正在为此努力。
- **规格符合预期**。这很难，因为人们永远无法确定一个正式的规范意味着我们认为它应该意味着什么。当然，如果没有正式的规范，同样的问题也会存在：如果规范是非正式的或不存在的，那么显然不可能精确地推断出正确的行为。

  可以通过证明规范的属性来降低这种风险，就像我们对安全证明所做的那样，这表明 seL4 能够强制执行某些安全属性。然后将问题转移到这些属性的规范上。它们比内核规范简单得多，减少了误解的风险。

  但归根结底，数学世界和物理世界之间总是存在鸿沟，没有任何终结的推理（正式的或非正式的）可以完全消除这一点。形式推理的优点是你确切地知道这个差距是什么。
- **该理论是正确的**。这听起来像是一个严重的问题，因为定理证明器本身就是大型而复杂的程序。然而，实际上这是三个假设中最不值得关注的一个。原因是 Isabelle/HOL 定理证明器有一个小核心（大约 10 kSLOC），可以根据逻辑公理检查所有证明。而且这个核心已经从广泛的形式推理领域检查了许多大大小小的证明，因此它包含正确性关键错误的机会非常小。

### 证明状态和覆盖范围

seL4 已经或正在针对多种架构进行验证：Arm、x86 和 RISC-V。其中一些比其他更完整，但缺少的部分通常正在处理或等待资金。详情请参阅 seL4 项目状态页面。

## 3.2 CAmkES 组件框架

CAmkES 是一个组件框架，允许您从架构上推理系统，即作为具有定义通信通道的沙盒组件的集合。图3.3 显示了主要的抽象。

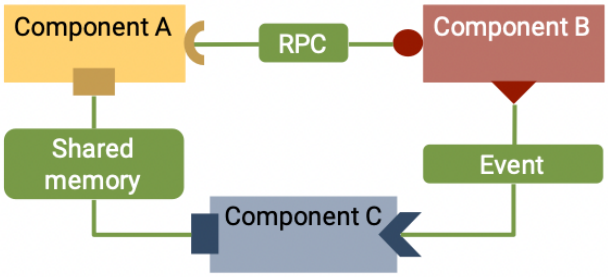


图 3.3: CAmkES 组件和连接器。

- **组件**表示为方框。它们代表程序、代码和数据，由 seL4 封装。
- **接口**显示为组件上的装饰。它们定义了一个组件如何被调用，或者如何调用其他组件。一个接口要么被导入（调用另一个组件的接口），要么被导出（能够被另一个组件的导入接口调用），除了共享内存接口，它是对称的。
- **连接器**通过将导入与导出接口链接起来，像接口一样连接。CAmkES 中的连接器始终是一对一的，但广播或多播功能可以在此模型之上通过构建将输入复制到多个输出的组件来实现。

CAmkES 系统以正式的架构描述语言（CAmkES ADL）指定，其中包含对组件、它们的接口和链接它们的连接器的精确描述。CAmkES 对系统设计者的承诺是，ADL 中指定的内容（如图3.3 所示）是可能交互的忠实表示。特别是，它承诺除了图中所示的之外，没有任何交互是可能的。

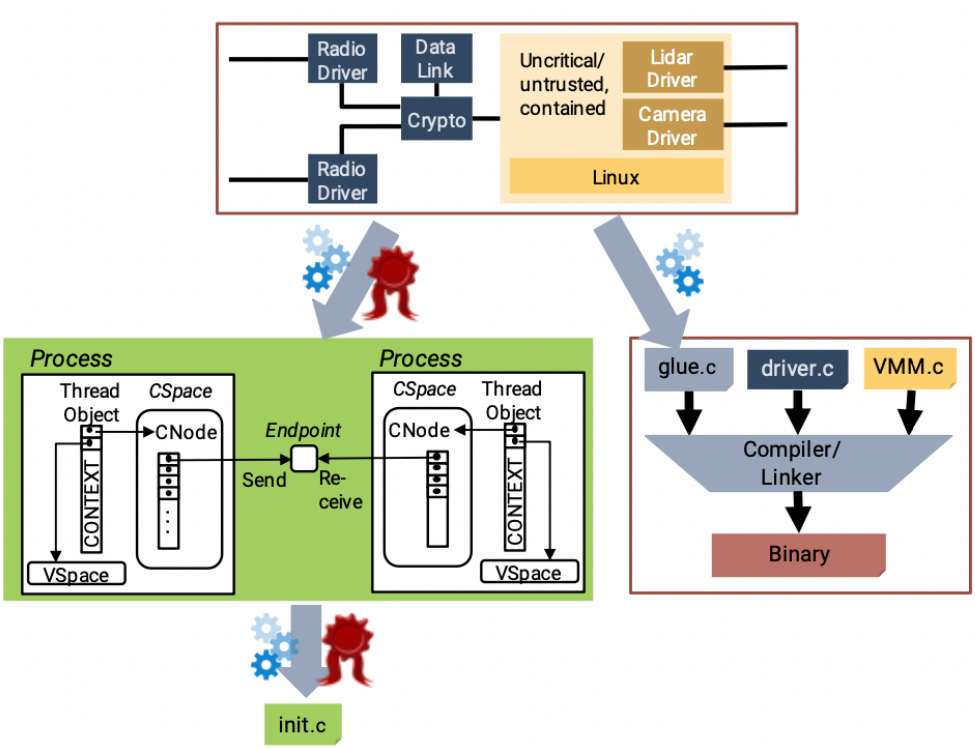


图 3.4: 已验证的架构映射和系统生成（请注意，并非所有验证步骤都具有全部强度）。生成的绿色框可证明是正确的。

当然，这个承诺依赖于 seL4 的执行，并且 ADL 表示必须映射到低级 seL4 对象和对它们的访问权限。这就是 CAmkES 机器实现的目标，如图 3.4所示。

在图中，架构（即 ADL 中描述的内容）显示在顶部。这是一个相当简单的系统，由四个本机组件和一个组件组成，其中一个组件包含一个虚拟机，该虚拟机托管一个带有几个网络驱动程序的 Linux 客户机。Linux VM 只通过 crypto 组件连接到其他组件，保证了它只能访问加密的链接，不能泄露数据。

即使这个简单的系统映射到数百个（如果不是数千个）seL4 对象，这表明 CAmkES 组件抽象提供的复杂性降低。

对于 seL4 级别的描述，我们有另一种形式语言，称为 CapDL（能力分布语言）。系统设计者从不需要处理 CapDL，它是纯粹的内部表示。CAmkES 框架包含一个编译器，它自动将 CAmkES ADL 转换为 CapDL，由指向左下的方框箭头指示。图左侧的框给出了 CapDL 中描述的 seL4 对象的（简化）表示。（它实际上是一个简单得多的系统的简化表示，基本上只是图 3.3顶部的两个组件以及它们之间的连接器。）

CapDL 规范是系统中访问权限的精确表示，它是 seL4 强制执行的。这意味着一旦系统进入 CapDL 规范所描述的状态，它就可以保证按照 CAmkES ADL 规范所描述的那样运行，因此架构级别的描述足以进一步推理安全属性。

因此，我们需要确保系统启动到 CapDL 规范所描述的状态。我们通过第二个自动化步骤来实现这一点：我们从 CapDL 生成启动代码，一旦 seL4 本身启动，控制并生成规范引用的所有 seL4 对象，包括描述活动组件的对象，并分发根据规范授予对这些对象的访问权限的功能（参见第 4 章）。在此初始化代码执行结束时，系统可证明处于 CapDL 规范所描述的状态，因此处于 ADL 规范所表示的状态。

从 ADL 规范生成的第三件事是组件之间的“粘合”代码。通过连接器发送数据需要调用 seL4 系统调用，其确切细节被 CAmkES 抽象隐藏。胶水代码正在设置这些系统调用。例如，“RPC”连接器将另一个组件提供的函数调用抽象为客户端组件执行的常规函数调用。

**笔记** 在撰写本文时，关于 CAmkES 和 CapDL 的证明尚未完成，但完成应该不远了。

另请注意，所提到的验证工作都没有处理通过时间通道的信息泄漏（目前）。这是一个尚未解决的主要研究问题，但我们处于解决它的最前沿。

# 第 4 章 关于能力

我们在第 1 章中遇到过能力，注意到它们是访问令牌。我们现在将更详细地研究这个概念。

## 4.1 什么是能力？


图 4.1: 能力是向特定对象传达特定权限的密钥。

如图4.1 所示，能力是一个对象引用；从这个意义上说，它类似于指针（功能的实现通常被称为“胖指针”）。它们是不可变的指针，从某种意义上说，能力将始终引用同一个对象，因此每个能力唯一地指定一个特定的对象。

除了指针之外，能力还对访问权限进行编码，实际上，能力是对对象引用及其传递给该对象的权限的封装。在基于能力的系统中，例如 seL4，调用能力是对系统对象执行操作的唯一方式。

例如，操作可能是调用组件中的函数。嵌入在能力中的对象引用指向该对象的接口，并传达调用该函数的权利（即组件对象上的特定方法）。能力可能会或可能不会同时传达将另一个能力作为函数参数传递的权利（将使用能力参数引用的对象的权利委托给组件）。

这是对 *CAmkES* 抽象级别发生的事情的高级描述。事实上，在 *CAmkES* 级别，功能本身被抽象掉了。 在下面，连接器由端点对象表示，客户端组件需要具有调用权的能力。

正是这种细粒度、面向对象的特性使功能成为面向安全系统的访问控制机制的选择。根据最小特权原则的要求，可以将赋予组件的权利限制在其完成其工作所需的绝对最低限度。

请注意，对象功能的概念与 *Linux* 所谓的“功能”完全不同（并且比它强大得多），后者实际上是具有系统调用粒度的访问控制列表 *(ACL)*。*Linux* 功能与所有 *ACL* 方案一样，都存在混淆代理问题，这是许多安全漏洞的根源，将在下一节中解释。*seL4* 能力没有这个问题。 *seL4* 能力也不易受到 *Boebert [1984]* 的攻击；这种攻击适用于直接在硬件中实现的功能，而 *seL4* 的功能由内核实现和保护。 *seL4* 对象有 *10* 种类型，均由能力引用：

- 端点用于执行受保护的函数调用；
- 回复对象表示来自受保护过程调用的返回路径；
- 地址空间提供围绕组件的沙箱（瘦包装器抽象硬件页表）；
- *Cnodes* 存储代表组件访问权限的能力；
- 线程控制块代表执行线程；
- *SchedulingContexts* 表示访问内核执行时间的特定分数的权利；
- 通知是同步对象（类似于信号量）；
- 帧代表可以映射到地址空间的物理内存；
- 中断对象提供对中断处理的访问；和
- 未类型化未使用（空闲）的物理内存，可以转换（“重新类型化”）为任何其他类型。

## 4.2 为什么引入能力

## 细粒度的访问控制

如上所述，能力提供了细粒度的访问控制，符合最小权限的安全原则（也称为最小权限原则，简称 POLA）。这与更传统的访问控制列表 (ACL) 的访问控制模型形成对比，后者用于 Linux 或 Windows 等主流系统，但也用于商业、据称安全的系统，例如 INTEGRITY 或 PikeOS。

要了解差异，请考虑访问控制在 Linux 中的工作方式：一个文件（以及适用于大多数其他 Linux 对象的文件模型）具有一组关联的访问模式位。其中一些位决定了它的所有者可以对文件执行哪些操作，其他位代表文件“组”的每个成员允许的操作，最后一组为其他所有人提供默认权限。这是一个面向主题的方案：它是主题（尝试访问的进程）的一个属性，它决定了访问的有效性，并且所有具有相同属性值（用户 ID 或组 ID）的主题都有相同的权利。此外，这些主体对具有相同访问属性设置的所有文件具有相同的权限。

这是一种非常粗粒度的访问控制形式，并且是对可以强制执行哪些安全策略的基本限制。一个典型的场景是，用户想要运行不受信任的程序（从 Internet 下载）来处理特定文件，但想要阻止程序访问用户有权访问的任何其他文件。这被称为限制场景，在 Linux 中没有干净的方法可以做到这一点，这就是人们想出重量级解决方法（我喜欢称它们为 hack）的原因，例如“chroot jails”、容器等。

有了能力，这个问题就很容易解决，因为能力提供了一种面向对象的访问控制形式。具体来说，当且仅当请求操作的主体提供授权它执行操作的能力时，内核才会允许操作继续进行。在受限场景中，不受信任的应用程序只能访问已被赋予权限的文件。因此，Alice 调用该程序，将权限交给程序允许读取的一个文件，以及程序可以写入其输出的文件的权限，并且程序无法访问其他任何内容——适当的最低权限。

## 干预和授权

能力有更多好的特性。一个是插入访问的能力，这是由于它们是不透明的对象引用这一事实的结果。如果 Alice 被赋予了一个对象的能力，她无法知道该对象到底是什么，她所能做的就是调用对象上的方法。

例如，系统设计者可能假装赋予 Alice 的能力指的是文件，而实际上它指的是安全监视器的通信通道，而后者又持有实际的文件能力。监视器可以检查 Alice 请求的操作，如果有效，则代表她在文件上执行这些操作，同时忽略无效的操作。监视器有效地虚拟化文件。

干预具有执行安全策略之外的应用；该方法可用于数据包过滤、信息流跟踪等。调试器可以透明地插入和虚拟化对象调用。它甚至可以用来懒惰地创建对象：代替对象引用，*Alice* 被赋予了构造函数的能力，然后在创建对象后替换该能力。

功能的另一个优点是它们支持安全有效的权限委派。如果 Alice 想要让 Bob 访问她的某个对象，她可以为该对象创建（在 seL4 中为“mint”）一个新的能力并将其交给 Bob。然后 Bob 可以使用该功能对对象进行操作，而无需返回 Alice。（相反，如果 Alice 确实想留在循环中，它可以使用上面解释的虚拟化。）

新能力可能会减少权利；Alice 可以使用它来授予 Bob 对该文件的只读访问权限。而 Alice 可以随时通过破坏她交给 Bob 的派生能力来撤销 Bob 的访问权限。

委派功能强大，无法在 ACL 系统中轻松安全地完成。其使用的一个典型案例是设置自主管理资源的子系统。当系统启动时，初始进程拥有系统中所有资源的权限（除了内核自己使用的少量固定资源）。然后，这个初始资源管理器可以通过创建新进程（辅助资源管理器）并将特权授予系统资源的不相交子集来对系统进行分区。

然后，子系统可以自主地（无需回溯到原始管理器）控制它们的资源子集，而不能相互干扰。只有当他们想改变原来的资源分配时，他们才需要让原来的经理参与进来。

环境权威和困惑的副手

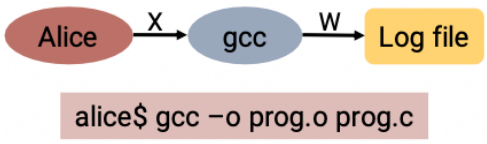


**图 4.2:** 编译器作为混乱的代表。

ACL 有一个无法解决的问题，通常称为混淆代理。让我们看一个 C 编译器。它接受一个 C 源文件并生成一个目标代码输出文件，文件名作为参数传递。要运行编译器，用户 Alice 必须对编译器具有执行权限，如图4.2所示。

假设编译器还在系统范围的日志文件中创建了一个条目用于审计目的。普通用户无法访问日志文件，因此编译器必须以提升的权限执行才能写入日志文件（传统上通过使其成为 setuid 程序来完成）。

如果 Alice 是恶意的，她可以欺骗编译器做它不应该做的事情。例如，Alice 可以在调用编译器时指定密码文件作为输出文件。除非编译器非常仔细地编写以避免任何潜在的滥用，否则编译器只会打开输出文件（密码文件）并用编译的目标代码覆盖它。Alice 不需要太多技巧就可以编写一个程序来编译，这样新生成的密码文件将赋予她不应该拥有的权限。

这里的基本问题是基于 ACL 的系统使用环境权限来确定访问权限。当编译器打开其输出文件进行写入时，操作系统通过查看编译器的主题 ID 来确定访问的有效性，以确定它是否有权访问该对象。由编译器决定操作是否有效，使编译器成为系统 TCB 的一部分，这意味着它必须被完全信任才能在任何情况下做正确的事情。

基于 ACL 的系统可以采用多种变通方法来缓解此处的特定问题，例如，确保密码文件和日志文件位于不同的安全域中（这不会阻止 Alice 破坏日志文件，这本身就是对于掩盖她的踪迹的攻击者来说这是一件有用的事情）。然后，这建立了通常的攻击和变通方法的军备竞赛，这对于好人来说总是一个失败的提议。

混乱是由于环境权威引起的：操作的有效性由代理（编译器）的安全状态决定，在这种情况下，代理是代表原始代理（Alice）进行操作的代理。为了获得适当的安全性，访问必须由 Alice 的安全状态决定。这意味着面额（对文件的引用）和权限（对文件执行操作的权利）必须耦合，这一原则称为无权不指定。如果是这种情况，那么编译器会调用指定对象（输出文件），并具有指定（来自 Alice）所附带的权限，并且 Alice 不能再混淆代理了。

这正是能力系统所强制执行的。在这样的系统中，Alice 需要具备三种能力：编译器的执行能力、输入文件的读取能力和输出文件的写入能力。她调用具有执行能力的编译器并将另外两个作为参数传递。当编译器随后打开输出文件时，它会使用 Alice 提供的功能，并且不会再出现混淆。编译器使用它自己拥有的单独功能来打开日志文件，从而将两个文件很好地分开。特别是，Alice 不可能欺骗编译器写入她自己无法访问的文件。

令人困惑的代理问题是功能的“杀手级应用”，因为 ACL 无法解决该问题。因此，下次有人试图向您推销“安全”操作系统时，不仅要询问他们是否有操作系统的正确性证明，还要询问它是否使用基于能力的访问控制。如果任何一个问题的答案都是“否”，那么你就会得到蛇油。

# 第 5 章 支持硬实时系统

seL4 被设计为保护模式的实时操作系统。这意味着，与传统的 RTOS 不同，seL4 将实时功能与内存保护结合起来，以确保安全性以及它对混合关键性系统的部分支持。

## 5.1 一般实时支持

seL4 有一个简单的、基于优先级的调度策略，易于理解和分析，这是硬实时系统的核心要求。内核本身永远不会调整优先级，因此用户可以控制。

另一个要求是有限的中断延迟。seL4 与 L4 微内核系列的大多数成员一样，在内核模式下执行时会禁用中断。这一设计决策极大地简化了内核设计和实现，因为内核（在单核处理器上）不需要并发控制。否则 seL4 的形式验证将是不可行的，但该设计也是实现出色的平均情况性能的推动力。

人们普遍认为，实时操作系统必须是可抢占的，除了较短的关键部分，以保持较低的中断延迟。虽然对于在简单微控制器上运行的传统无保护 *RTOS* 来说确实如此，但这种信念被误认为是保护模式系统，例如 *seL4*。原因在于，当在启用了内存保护的强大微处理器上运行时，进入内核、切换上下文和退出内核的时间非常重要，而且不比 *seL4* 系统调用少多少。在中断延迟方面，抢占式设计几乎没有什么好处，但复杂性方面的成本会非常高，使得抢占式设计不合理。

只要所有系统调用都很短，这就会起作用。在 seL4 中它们通常是，但也有例外。尤其是撤销一项能力可能是一项长期运行的操作。seL4 通过将此类操作分解为较短的子操作来处理这种情况，并且如果存在未决中断，则可以在每个子操作之后中止和重新启动整个操作。

这种方法称为增量一致性。每个子操作将内核从一致状态转换为另一个一致状态。该操作的结构使得在中止后，可以重新启动操作，而无需重复中止之前已成功的子操作。内核在每个子操作之后检查挂起的中断。如果有，它会中止当前操作，此时中断会强制重新进入内核，内核会处理中断。完成后，原始系统调用将重新启动，然后从中止点继续，保证进度。

我们对 seL4 进行了完整且可靠的最坏情况执行时间 (WCET) 分析，这是保护模式操作系统中唯一记录的分析 [Blackham et al., 2011, Sewell et al., 2016]。这意味着我们已经获得了所有系统调用延迟以及最坏情况中断延迟的可证明的硬上限。

这种 WCET 分析是支持硬实时系统的先决条件，也是使 seL4 在竞争中脱颖而出的功能。虽然已经对未受保护的 RTOS 进行了完整而可靠的 WCET 分析，但保护模式系统的行业标准方法是使内核承受高负载，测量延迟，取观察到的最差的一个并添加一个安全系数。不能保证通过这种方法获得的界限是安全的，并且不适合安全关键系统。

我们对 Arm v6 处理器的 seL4 进行了 WCET 分析。此后，由于 Arm 已停止提供有关最坏情况下指令延迟的所需信息，而英特尔从未为其架构提供这些信息，因此它已被搁置。然而，随着开源 RISC-V 处理器的出现，我们将能够重做这一分析。

## 5.2 混合临界系统

### 什么是混合临界系统？

临界性是安全领域的一个术语，与组件故障的严重性有关。例如，航空电子标准将故障从“无影响”（对车辆安全）分类到“灾难性”（生命损失）。组件越关键，所需的保证就越广泛（且成本更高），因此有强烈的动机将关键性保持在较低水平。

混合临界系统 (MCS) 由不同临界的（相互作用的）组件组成。其核心安全要求是组件的故障不能影响任何更关键的组件，因此可以确保关键组件独立于不太关键的组件。

MCS 的趋势源于对整合的渴望：传统上，关键系统会为每个功能使用专用的微控制器，即通过气隙进行隔离。随着功能的增加，这种方法会导致处理器（及其封装和布线）的激增，这会导致 MCS 旨在克服的空间、重量和功率 (SWaP) 问题。

这类似于在同一系统中具有受信任和不受信任的组件的安全概念，并且对操作系统的核心要求在这两种情况下都是强隔离。安全领域的挑战在于，安全性不仅取决于功能的正确性，还取决于及时性：关键组件通常具有实时要求，这意味着它们必须在截止日期前响应事件。

## 传统的 MCS 方法

传统的 MCS 操作系统在时间和空间上完全隔离组件，这种方法称为严格的时间和空间分区 (TSP)，以 ARINC 653 航空电子标准 [ARINC] 为例。这意味着每个组件都被静态分配了一个固定的内存区域，并且分区根据预先确定的时间表执行，具有固定的时间片。

TSP 方法保证了隔离，但有严重的缺点。最明显的一个是资源利用率低。每个实时组件都必须能够在其时间片内完成其工作，因此时间片必须至少是组件的最坏情况执行时间。组件的 WCET 可能比典型的执行时间大几个数量级，因为它必须考虑到特殊情况。

此外，确定 WCET 的安全界限通常很棘手。对于关键组件，必须非常保守地说服持怀疑态度的认证机构，这通常会导致高估。这意味着通常处理器的利用率大大不足。但是，由于严格的划分，松弛时间不能被其他组件使用，因此利用率低是一个固有的问题。基本上，通过保留气隙的强隔离，TSP 也保留了其较差的资源使用率。

TSP 的另一个大缺点是中断延迟本来就很高。以图 5.1 为例，它可能代表一种（高度简化的）自动驾驶汽车。关键组件是一个控制回路，它每 5 毫秒执行一次，以处理传感器数据并向执行器发送命令。它的 WCET 和时间片是 3 毫秒。车辆还与可以更新航路点的地面站通信。由于系统以 5 毫秒为周期运行，因此这是可以处理网络中断的延迟，极大地限制了网络吞吐量和对外部事件的响应能力。

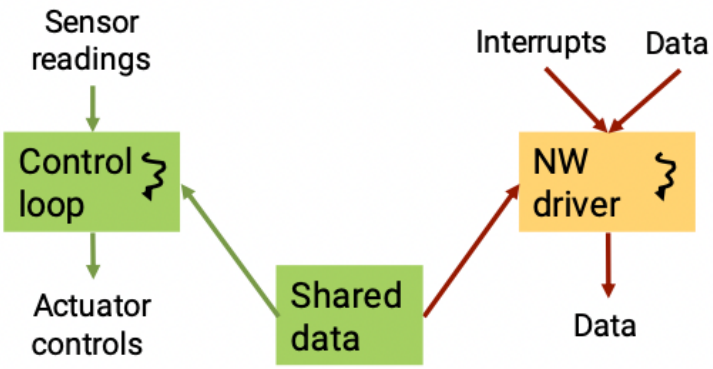


**图 5.1:** 混合临界系统的简化示例。

## seL4 中的 MCS 支持

MCS 的核心挑战是操作系统必须提供强大的资源隔离，但 TSP 过于简单（因此不灵活）。在空间资源方面，seL4 已经有了一个灵活、强大且可证明安全的模型：对象能力（见第 4 章）。MCS 支持将此扩展到时间：对处理器的访问现在也由功能控制。

seL4 的处理器时间能力称为调度上下文能力。一个组件只有拥有这样的能力才能获得处理器时间，并且它可以使用的处理器时间量被编码在能力中。这类似于空间对象的访问权限的工作方式。

在传统的 *seL4* 中（就像在它之前的大多数 *L4* 内核中一样），线程有两个主要的调度参数：优先级和时间片，它们决定了对处理器的访问。优先级决定了线程何时可以执行：如果没有更高优先级的线程可运行，它就可以运行。时间片决定了内核在抢占线程之前让线程运行多长时间（除非它之前被更高优先级的线程抢占为可运行）。

当时间片用完时，调度器将再次选择最高优先级的可运行线程（可能是刚刚被抢占的线程），并在优先级内使用循环策略。

*seL4* 的 *MCS* 版本将时间片替换为调度上下文对象的能力，该对象执行类似的功能，但以更精确的方式，这是隔离的关键：调度上下文包含两个主要属性。*(1)* 时间预算，类似于旧的时间片，限制线程在被抢占之前可以执行的时间。*(2)* 一个时间段，它决定了可以使用预算的频率：线程在每个时间段获得的时间不会超过一个预算，从而防止它独占 *CPU*，而不管其优先级如何。

调度上下文支持推理线程可以消耗的时间量，因此，还剩下多少时间。具体来说，它们可用于防止高优先级线程独占处理器。

应用于上面的例子，这意味着我们可以给（不太关键的）设备驱动程序比（关键的）控制组件更高的优先级。这允许驾驶员抢占控制权，从而实现高响应性。但预算限制将阻止驱动程序独占 *CPU*。例如，我们给控制器一个 *3* 毫秒的预算（它的 *WCET*）和一个 *5* 毫秒的周期（对应于它运行的频率）。我们给高优先级驱动程序一个 *3* 微秒的小预算，周期为 *10* 微秒，这意味着它在任何情况下都不会消耗超过 *30%* 的总处理器时间，但可以足够频繁地执行以确保良好的响应能力。重要的是，我们可以保证需要不超过 *60%* 的可用处理器时间的控件有足够的时间来满足其最后期限。

根据 MCS 的核心要求，通过保证关键期限而不考虑驾驶员的行为，我们将控制与不受信任的驾驶员隔离开来。特别是，驾驶员不需要被证明是安全关键的。

seL4 的时间能力模型解决了 MCS 的许多其他挑战，这些挑战超出了本白皮书的范围，我们将感兴趣的读者推荐给同行评审的出版物 [Lyons et al., 2018]。可以说，seL4 为任何适用于关键系统的操作系统提供了最先进、最灵活的 MCS 支持。

# 第 6 章 安全性不是性能不佳的借口

性能一直是 L4 微内核的标志，seL4 也不例外。我们为实际使用构建了 seL4，我们的目标是相对于我们之前拥有的最快内核，IPC 性能的损失不超过 10%。事实证明，seL4 最终击败了这些内核的性能。

它的性能优于任何其他微内核。这是一个很难证明的说法，因为比赛通常会将他们的表现数据放在胸前（有充分的理由！）

但是，我们会一有机会就公开提出这一绩效声明。如果有人不同意，他们需要出示证据。我们还通过许多非正式渠道了解到，其他系统的 IPC 性能往往比 seL4 慢 2 倍到慢得多，通常约为 10 倍。

少数独立的性能比较肯定支持我们的主张。

米等人。*[2019]* 比较了三个开源系统 *seL4*、*Fiasco.OC* 和 *Zircon* 的性能。它发现 *seL4 IPC* 成本大约比内核入口、地址空间切换和内核出口的硬件限制高出 *10-20%*。*Fiasco.OC* 比 *seL4* 慢两倍多（接近硬件限制的三倍），*Zircon* 比 *seL4* 慢几乎九倍。

顾等人。*[2016]* 比较了 *CertiKOS* 与 *seL4* 的性能，在 *CertiKOS* 中测量了 *3,820* 个往返 *IPC* 操作周期，而在 *seL4* 中测量了 *1,830* 个周期，这是两倍。然而，事实证明，*seL4* 基准测试套件 *sel4bench* 当时在处理 *x86* 上的计时器时存在错误，导致延迟过大。正确的 *seL4* 性能数据约为 *720* 个周期，或比 *CertiKOS* 快五倍以上。这是在 *CertiKOS* 提供非常有限的功能且没有基于能力的安全性的背景下。

# 第 7 章 实际部署和增量网络改造

## 7.1 一般注意事项

在计划使用 seL4 保护系统的安全性或安全性时，第一步应该是确定您需要保护的关键资产。目标应该是最小化这部分可信计算库，并使其尽可能模块化，每个模块都成为一个受自我保护的 CAmkES 组件。

另一个重要的准备工作是在您的平台上检查 seL4 的可用性和验证状态。显然你会想要一个经过验证的内核，这就是 seL4 的全部意义所在。然而，即使在内核未经过验证的平台上，它与经过验证的平台共享其大部分代码这一事实也将为您提供比几乎任何其他操作系统更高的保证。但请记住，如果没有验证，保证不会是它可以做到的。此外，如果您使用的内核未针对您的平台进行验证，或者以任何方式进行了修改，则不得提出任何验证声明。

此外，您还需要评估可用的用户级基础架构是否足以满足您的目的。如果没有，那么这就是社区可以帮助您的地方。有些公司专门为 seL4 的采用提供支持。此外，如果您自己开发任何普遍有用的组件，您应该认真考虑在适当的开源许可下与社区共享它们。回馈的人会发现更容易获得他人的帮助。

## 7.2 改造现有系统

seL4 的大多数实际部署不会运行所有本机程序。通常，有一些重要的遗留组件移植起来会很昂贵，因为它们太大或依赖于目前 seL4 不支持的太多系统服务。此外，本机运行此类遗留堆栈通常几乎没有安全性或安全性收益。

使用 seL4 的虚拟化功能通常是实用的方法，第 2.3 节显示了示例。

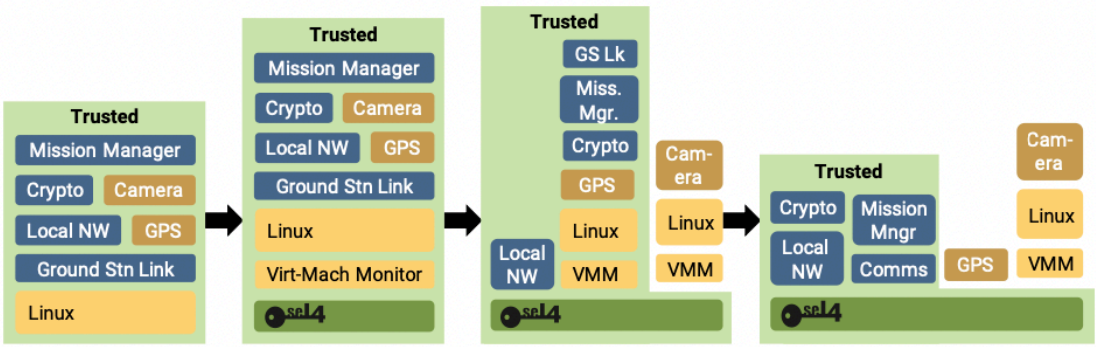


图 7.1: DARPA HACMS 计划期间波音 ULB 任务计算机的增量网络改造。

典型的方法是我们所说的增量网络改造，这是当时 DARPA 项目主管 John Launchbury 创造的一个术语。如图7.1所示，这通常从简单地将整个现有软件堆栈放入运行在 seL4 上的虚拟机中开始。显然，这一步在安全性和安全性方面没有任何好处，它只会增加（非常小的）开销。它的意义在于它提供了从哪里开始模块化的基线。

一个很好的例子是我们的 HACMS 项目合作伙伴在网络改造波音 ULB 自动直升机方面所做的工作。最初的系统在 Linux 上运行，第一步，团队将 seL4 放在下面。

下一步涉及两个组件：特别不受信任的摄像头软件被移至第二个虚拟机，同样运行 Linux，两个 Linux 虚拟机通过 CAmkES 通道进行通信。同时，网络堆栈被从虚拟机中拉出并转换为本地 CAmkES 组件，也与主虚拟机通信。

最后一步将所有其他关键模块以及（不受信任的）GPS 软件拉到单独的 CAmkES 组件中，从而移除原始的主 VM。最终系统由许多运行 seL4-native 代码的 CAmkES 组件和仅运行 Linux 和相机软件的单个 VM 组成。

结果是，虽然最初的系统很容易被 DARPA 聘请的专业渗透测试人员入侵，但最终状态却具有很强的弹性。攻击者可以破坏 Linux 系统并为所欲为，但无法突破并破坏系统的任何其余部分。该团队有足够的信心展示飞行中的攻击。

# 第 8 章 结论

seL4 是世界上第一个具有实现正确性（功能正确性）证明的操作系统内核。然后我们将验证扩展到二进制和安全执行属性，如第 3 章所述。

虽然到目前为止还有其他经过验证的操作系统内核，但 seL4 仍然定义了最先进的技术 [Heiser, 2019]：它拥有最全面的验证故事，它仍然是唯一经过验证的基于能力的操作系统，并且它拥有最多先进的实时支持。我们正在进行的研究旨在确保 seL4 将保持其在面向安全和安全的操作系统中的明确领导者地位，例如通过开创系统性和原则性的预防通过时间通道泄露信息 [Ge et al., 2019] .

除了这一技术领先地位之外，seL4 实际上仍遥遥领先于其继任者：虽然我们从一开始就为实际使用设计了 seL4，但几乎所有其他经过验证的操作系统内核都是学术玩具，远不具备实际应用能力。事实上，我们只知道另一个（最近）经过验证的系统实际上是可部署的（尽管在更有限的场景中）。

seL4 的实际准备情况是驱动设计的两个方面的结果：不妥协的性能焦点，如第 6 章所强调的，以及旨在支持最广泛的应用场景和安全策略的机制，后者由能力支持基于访问控制（第 4 章）。

十年来将 seL4 带入现实世界，包括对遗留系统进行网络改造（第 7 章），显然帮助我们改进和改进了系统，但我很自豪地说，轻微的增量更改就足够了。一个例外是 MCS 支持（第 5.2 节），它需要对模型及其实现进行相当重大的更改，但时间的特权管理是我们在原始设计时有意留在待办事项篮子中的一件事 [海瑟和埃尔芬斯通，2016]。

希望本白皮书能够让您合理地了解 seL4 是什么、您可以使用它做什么，以及重要的是，您为什么要使用它。我希望这将帮助您成为 seL4 社区的积极成员，包括加入和参与 seL4 基金会。

我希望这份文件会不断发展，我热衷于反馈。但最重要的是，我很想听听您部署 seL4 的经验。

# 参考文献


[1] Simon Biggs, Damon Lee, and Gernot Heiser. “The jury is in: Monolithic OS design is flawed”. In: *Asia-Pacific Workshop on Systems (APSys). Korea: ACM SIGOPS*. 2018.

[2] Bernard Blackham et al. “Timing analysis of a protected operating system kernel”. In: *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE. 2011, pp. 339–348.

[3] William Earl Boebert. “On the inability of an unmodified capability machine to enforce the*-property”. In: *Proc. 7th DoD/NBS Computer Security Conference*. 1984, pp. 291–293.

[4] Anthony Fox and Magnus O Myreen. “A trustworthy monadic formalization of the ARMv7 instruction set architecture”. In: *International Conference on Interactive Theorem Proving*. Springer. 2010, pp. 243–258.

[5] Qian Ge et al. “Time protection: the missing OS abstraction”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–17.

[6] Ronghui Gu et al. “{CertiKOS}: An Extensible Architecture for Building Certified Concurrent {OS} Kernels”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 653–669.

[7] Gernot Heiser and Kevin Elphinstone. “L4 microkernels: The lessons from 20 years of research and deployment”. In: *ACM Transactions on Computer Systems (TOCS)* 34.1 (2016), pp. 1–29.

[8] Gernot Heiser, Gerwin Klein, and Toby Murray. “Can we prove time protection?” In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2019, pp. 23–29.

[9] Gernot Heiser. *10 years seL4: Still the best, still getting better,* 2019. URL: https://microkerneldude.wordpress.com/2019/08/06/10-years-sel4-still-the-best-still-getting-better/ (visited on 08/06/2019).

[10] Gerwin Klein et al. “Comprehensive formal verification of an OS microkernel”. In: *ACM Transactions on Computer Systems (TOCS)* 32.1 (2014), pp. 1–70.

[11] Gerwin Klein et al. “seL4: Formal verification of an OS kernel”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 207–220.

[12] Anna Lyons et al. “Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time”. In: *Proceedings of the Thirteenth EuroSys Conference*. 2018, pp. 1–16.

[13] Zeyu Mi et al. “Skybridge: Fast and secure inter-process communication for microkernels”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–15.

[14] Thomas Sewell, Felix Kam, and Gernot Heiser. “Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis”. In: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2016, pp. 1–11.

[15] Thomas Sewell, Felix Kam, and Gernot Heiser. “High-assurance timing analysis for a high-assurance real-time operating system”. In: *Real-Time Systems* 53.5 (2017), pp. 812–853.

[16] ARINC Specification. *653, “Avionics Application Software Standard Interface.”*. 2015.

[17] Ken Thompson. “Reflections on trusting trust”. In: *Communications of the ACM* 27.8 (1984), pp. 761–763.