Tock：从研究到保护1000万台计算机

Leon Schuermann
lschuermann@princeton.edu 普林斯顿大学

Brad Campbell
bradjc@virginia.edu 弗吉尼亚大学

Branden Ghena
branden@northwestern.edu 西北大学

Philip Levis
pal@cs.stanford.edu 斯坦福大学

Amit Levy
aalevy@princeton.edu 普林斯顿大学

Pat Pannuto
ppannuto@ucsd.edu 加利福尼亚大学圣迭戈分校

摘要

Tock 于 10 年前起步，是由学者开发的一款研究型操作系统，旨在帮助其他学者构建城市传感应用。借助一种新语言（Rust）和新的硬件保护机制，Tock 实现了“在 64 kB 计算机上安全、高效地进行多道程序设计 ”。如今，Tock 已成为一个开源项目，拥有活跃的用户和贡献者社区。它部署在数据中心服务器的信任根硬件和数百万台笔记本电脑上；它被用于开发汽车产品和航天产品、可穿戴电子设备以及硬件安全令牌，同时仍是操作系统研究的平台。本文着重探讨 Tock 的技术设计对其普及的影响、使用类型安全语言（Rust）所面临的挑战和意想不到的益处（尤其是在安全敏感场景中），以及学术界支持一款用于生产环境的开源操作系统的经验。

CCS 概念：• **软件及其工程 → 操作系统**。

ACM 参考文献格式：
Leon Schuermann、Brad Campbell、Branden Ghena、Philip Levis、Amit Levy 和 Pat Pannuto。2025年。《Tock：从研究到保护1000万台计算机》。载于*ACM SIGOPS* 第 *31* 届操作系统原理研讨会（*SOSP '25*），*2025*年*10*月*13*日至*16*日，首尔，大韩民国。ACM，纽约，美国，14页。https://doi.org/10.1145/ 3731569.3764828

1 引言

本文介绍了 Tock 的设计、演进与部署。Tock 是一款以 Rust 编写的安全嵌入式操作系统（OS）。它最初是一个研究型操作系统，旨在探索 Rust 语言如何提升嵌入式系统安全性，如今，Tock 运行于数据中心服务器中的信任根硬件上，可安全启动笔记本电脑，并为汽车系统和航天系统提供支持。与此同时，Tock 仍是操作系统研究的平台。

本文介绍了我们推动 Tock 演进的经历：它最初是一个研究系统，后来发展成为一个开源项目，部署于数千万台计算机上（图 1）。我们希望这些经历能为系统研究提供有价值的见解。首先，Tock 的独特约束——从以安全性为重点的操作系统，到在资源高度受限的系统中实现多道程序设计——迫使它作出一系列有趣且非常规的设计决策与权衡。其次，Tock 是最早完全使用 Rust 编写的操作系统内核之一，早于 Rust 获得操作系统研究人员的认可 [27] 和实践者的采用 [11, 12, 33]。在使用 Rust 的过程中，我们遇到了意料之外的挑战，也发现了新的机遇；这些经验对更广泛的系统社区具有参考价值。第三，Tock 实现了实际采用并得到广泛部署，且经常用于安全关键型应用程序，其发展主要由学术界贡献者推动。我们将回顾它如何实现这一目标，以及学术研究与开源项目之间的关系。

我们在第 2 节重新审视 Tock 最初的设计。许多影响 Tock 设计的初始目标和约束至今仍然适用。例如，使用 Rust 实现隔离和最小特权，同时几乎不增加中央处理器或状态开销，至今仍是 Tock 的一项核心优势。同样，Tock 没有采用大多数嵌入式操作系统所用的单一保护域设计，而是选择通过硬件介导来隔离内核和应用程序。这不仅对安全性至关重要，也更广泛地支持了以 C 编写的遗留应用程序，以及跨多个团队的产品开发管理。操作系统早期的大量修订都在应对如何利用 Rust 实现这一设计所带来的挑战，而我们早期做出的许多决策至今仍然适用。

第 3 节探讨了自首次发布以来 Tock 如何不断演进。随着 Tock 的开发不断推进、Rust 日渐成熟，我们发现 Tock 内部的一些设计与 Rust 的健全性要求并不兼容；要解决这个问题，就必须大幅重新设计系统调用 ABI、内核主循环以及 Tock 内核扩展抽象（“胶囊”）。



本作品采用知识共享署名国际 4.0 许可协议授权。*SOSP '25*，*2025*年*10*月*13*日至*16*日，首尔，大韩民国。© 2025 版权归所有者/作者所有。ACM ISBN 979-8-4007-1870-0/2025/10。
https://doi.org/10.1145/3731569.3764828

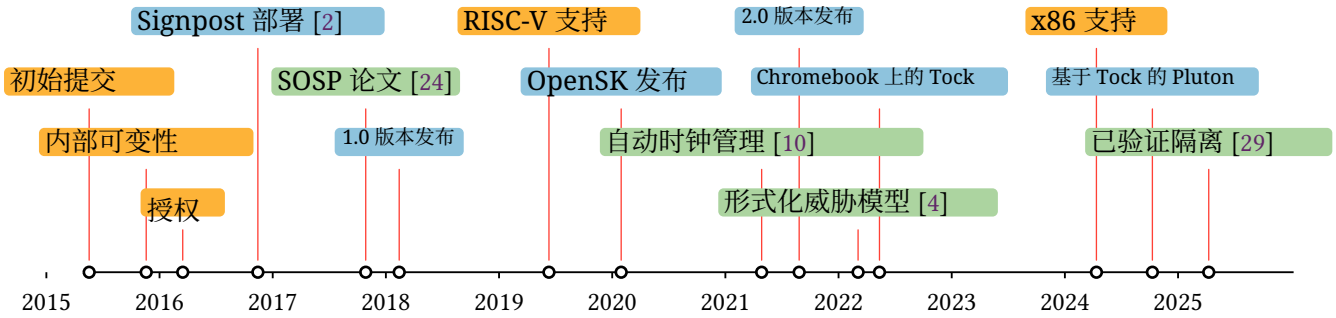

图 1：Tock 开发、部署和[illegible]初旨在[illegible]规模的传感器网络平台提供安全的多程序运行环境。此后，其他领域的用户也开始采用 Tock，主要用于硬件信任根设备，这又推动了更多开发；其中包括修订系统调用接口、制定形式化威胁模型、加强隔离保证，以及支持更多 CPU 架构和硬件保护机制。

Tock 应用领域重点的扩展推动了这些发现与改动。操作系统最初旨在支持低功耗城市传感系统和可穿戴电子设备。然而，首次发布后不久，Tock 便吸引了开发硬件信任根芯片的公司和非营利组织的关注。这些设备通常是资源受限的微控制器，是所有计算系统安全的基础。它们存储密钥、验证引导映像，并且是验证系统未遭入侵的第一步。对于这些用途，采用 Rust 而非 C 编写内核的优势显而易见。此外，新加入的开发者也大力推动改进对 Rust 用户空间应用程序的支持。

在第 4 节中，我们探讨了 Tock 如何利用 Rust 强大且富有表现力的类型，在内存隔离之外提供各种保证。在过去十年中，Tock 演化出了基于类型的构造，用于强制执行调用约定，从而完全避免常见的驱动程序错误；用于支持复杂软件多个层级之间的内存共享；以及用于跨平台泛化硬件虚拟化层——所有这些都无需动态内存分配。这些改进可以为未来考虑使用 Rust 的操作系统内核开发者提供指导。

随着时间的推移，Tock 内核核心代码的规模始终保持稳定，尽管新增了大量功能（见图 5）。虽然 Tock 一直禁止在驱动程序中使用 语言特性，第 4 节进一步讨论了 Tock 如何改进其接口和设计，以限制并减少 的使用。如今， 仅在少数有限的场合使用：必须与不受 Rust 类型约束的软件或硬件交互时（例如硬件内存映射 I/O 寄存器和进程边界），以及创建用于限制对敏感内核接口访问的能力时。

尽管 Rust 有诸多优势，我们的经验也凸显了这样一个挑战：在面对不受语言约束的硬件和应用程序进行编程时，如何保留语言安全性不变量（第 5 节）。与大多数嵌入式系统一样，Tock 内核完全由事件驱动，且只有一个栈。然而，Rust 的内存模型与多线程、同步执行密不可分：生命周期对应栈帧，而借用的持续时间与一次函数调用相同。在实现异步系统调用边界时，要维护 Rust 的别名和可变性不变量颇具挑战，最终迫使 Tock 进行了重大重新设计。正确对接 DMA 硬件同样困难。内部可变性等机制支持包括不同层之间循环依赖在内的常见内核架构，但也使 Tock 与其他使用 Rust 开发的嵌入式软件兼容性较差。我们分享这些见解，以阐明使用 Rust 构建操作系统时为何需要特别的设计考量，而使用其他语言时则无需如此。

最后，我们将在第6节中重新探讨 Tock 广泛采用背后的问题。大多数研究系统都未能得到采用。能够继续保持开源的更是少之又少，而许多此类系统最终主要是从学术界转交给专门的初创公司或单一的成熟组织。Tock 开源项目仍由学者主导维护，大部分（但并非全部）开发工作，尤其是核心功能和安全敏感功能，均由大学的教师和学生完成。这样做也带来了挑战：学术压力促使人们专注于创新研究，而非工程；行业压力则促使人们优先满足产品交付期限，而非将贡献提供给上游。紧迫的产品需求（例如将现有代码适配到有限资源中）可能与着眼长远、做出可持续且可维护的设计决策相冲突。

我们认为，尽管面临挑战，这种模式仍使该项目得以成为一个切实可行的研究平台

，用于操作系统、安全硬件、传感器网络和嵌入式验证领域，同时促进了原本不会发生的跨公司合作。随着其他研究平台逐渐获得关注，Tock 所采用的模式或可成为将学术研究转化为广泛应用的一种选择。

归根结底，尽管这项长期的协作努力影响并推动了 Tock 的演进，该项目仍得以专注于最初的研究问题和设计。

2 Tock 的原始设计

Tock 的原始设计面向多程序运行、由电池供电或通过能量采集供电的传感器网络平台。在这些场景中，主要约束是：面对有缺陷或恶意的内核驱动程序和应用程序时，系统必须具备鲁棒性和安全性；而至关重要的是，还要降低能量消耗。Tock 还需要能够在种类繁多的硬件平台上运行并具备可移植性，这些平台各自配备不同的加速器、传感器、无线协议和物理接口。

Tock 面向一类具有以下特征的计算机：1. 32 位 Cortex-M 架构，2. ~100kB RAM 和不到 1MB 的可执行非易失性存储，3. 没有虚拟内存，但具备原始的内存保护，4. 丰富的片上硬件外设和加速器，以及 5. 对无线电收发器的底层访问。

例如，Tock 的首次部署是 Signpost [2], ，这是一个由传感器网络平台组成的网络，分布在两所大学校园内。每个部署单元由若干相互独立但彼此互联的电路板组成，每块电路板都配有一个微控制器和多种不同的传感器（温度、压力、运动、声音、光、无线电频谱等），以及 WiFi、LoRa 和蓝牙收发器，用于彼此通信、与行人通信以及与数据收集服务器通信。尽管硬件配置差异很大，每个安装点都运行着一些常见的应用程序（例如服务器和点对点通信、电路板间协调等），以及一些特定于地点和硬件的应用程序（例如车辆检测、空气质量监测等）。最后，Signpost 安装点由太阳能供电，因此受到电力限制。

基于这些应用程序和硬件约束，我们推导出了 Tock 的许多初始核心设计决策（图 2）。例如，Signpost 中应用程序和传感器的多样性意味着 Tock 应能够运行多个彼此隔离（且可能存在错误或恶意）的并发应用程序，而其内核必须具有高度可扩展性，以支持新的微控制器和外围设备。反过来，Tock 目标平台的功耗限制要求采用一种设计，尽可能延长设备处于硬件休眠状态的时间。我们通过完全异步的内核设计实现了这一点，并通过系统调用接口将这种异步性延伸至用户空间应用程序。最后，这些部署的广泛分布意味着它们必须可靠，并避免应用程序与驱动程序之间以及应用程序彼此之间发生故障联动。为此，我们采用了无堆内核设计，并严格分离与应用程序相关的状态。

这些设计选择大多不同于低资源微控制器中常见的典型嵌入式框架 [3, 14, 36]。大多数投入生产的嵌入式系统都以可扩展性为设计目标，但每个组件（例如设备驱动程序、虚拟化层或应用程序）都可以不受限制地访问系统和内存——系统并不存在隔离机制。通常，所有应用程序功能都会与系统的其余部分一同编译和交付，且无法动态替换。由于大多数系统不会将应用程序与内核分离，因此没有系统调用接口，驱动程序也可以自由暴露任意接口，这些接口往往会混用异步 I/O 和阻塞 I/O。它们的一个共同点是避免堆分配；由于难以推断资源耗尽的情况，嵌入式系统通常不鼓励堆分配，但许多其他操作系统内核会使用它来满足应用程序动态变化的需求。

新近出现的类型安全语言，加上低资源微控制器对硬件隔离的支持日益普遍，这两者相结合，使 Tock 能够与相关系统截然不同，同时仍将资源开销降至最低，并支持高能效运行。例如，Tock 利用 Rust 强大的类型安全和内存安全特性，在不同的胶囊（Tock 的内核扩展）之间提供隔离，而无需动用昂贵且稀缺的硬件内存保护资源。这一设计受到先前研究系统的启发；这些系统使用类型安全语言实现内核扩展，其中包括 Spin 使用 Modula3 [7] 以及 Singularity 使用 Sing# [20]。Rust 的所有权类型让我们得以利用这类语言，同时避免动态内存分配和垃圾回收带来的资源或性能开销。

然而，我们发现 Rust 也对操作系统的设计和架构施加了一些意想不到的限制。具体来说，在将许多相互连接的子系统组合成一个操作系统时，所有权和借用的概念带来了挑战。这反过来又影响了 Tock 的编程模型。

本节接下来的内容将聚焦于 Tock 的核心设计决策。我们先从内部可变性带来的挑战及其对 Tock 内核架构的影响谈起，随后介绍它实现内核可扩展性的机制，以及它如何在无堆设计中维护特定于应用程序的状态。我们将说明 Tock 如何利用语言级隔离实现内核隔离，以及如何通过进程支持传统的多道程序设计，并借助

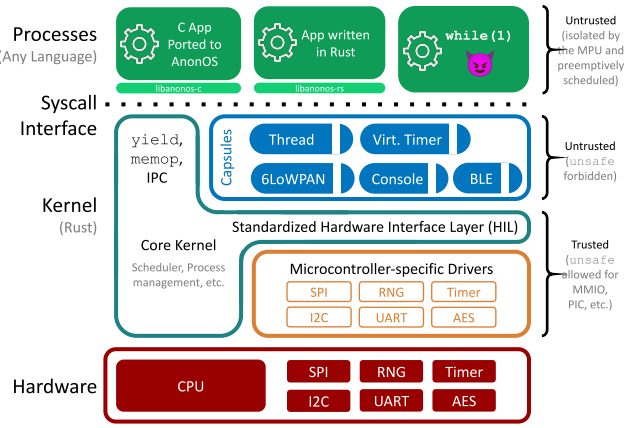

图 2：Tock 系统分为四类组件。进程通过硬件隔离，采用抢占式调度，可以用任何语言编写，且不受信任。它们通过定义明确且范围狭窄的系统调用接口与内核交互。内核由通用核心、受信任的硬件专用适配器，以及胶囊组成；胶囊是协作式调度的半可信扩展，使用 Rust 类型系统进行隔离，并通过狭窄且严格受限的接口访问进程内存和硬件。

硬件隔离。最后，我们将介绍 Tock 的异步内核和系统调用。

2.1 内部可变性

Rust 的一个标志性特征是其所有权模型。任何 Rust 对象都可以被移动，使原所有者失去所有权；也可以被借用一段时间（通过获取一个引用）。对于任何被借用的对象，最多只能有一个独占的可变引用，或者一个或多个共享的不可变引用，二者不能同时存在。这一限制对于保持类型安全至关重要 [17]。

Rust 的许多安全性改进都建立在这些基本概念之上。然而，这种模型增加了构建操作系统的难度。为说明这一点，请考虑操作系统中的两个组件：文件系统和底层存储驱动程序。文件系统执行写入时，会修改自己的内部状态，并通过引用调用存储驱动程序。随后，存储驱动程序会修改存储子系统的内部状态、执行操作，并在完成后再次通过引用向文件系统驱动程序发出异步回调。这种传统设计无法直接用 Rust 实现。这是因为这两个驱动程序都会修改各自的内部状态（因此必须通过唯一的可变引用调用对方），但它们之间形成了循环引用——在这种情况下，编译器无法验证可变引用的唯一性。

这体现了一种设计权衡。一种可能的做法是，在驱动程序之间通过消息传递进行交互。在这种模式下，所有操作系统组件都由一个中介程序负责。组件可以通过非引用 ID 向中介程序寻址，从而交换数据。这避免了循环引用。但这种做法用一个中央管理器取代了组件之间直接明了的函数调用，增加了复杂性和开销。

第二种可能的方法基于内部可变性。具有内部可变性的对象即使通过共享的不可变引用进行引用，也能改变其状态。这使两个组件可以使用共享的不可变引用相互引用，而共享的不可变引用不会对循环引用施加任何限制。因此，组件之间可以采用传统的引用方式。使用内部可变性需要付出两方面的代价：某些状态变更会增加额外的运行时检查；此外，开发者还必须谨慎确保对单个驱动程序的重入调用不会破坏一致性。

最终，Tock 选择利用内部可变性，在组件之间共享传统风格的（循环）引用，从而提供更明确的程序结构。

2.2 使用胶囊扩展内核

胶囊是 Tock 主要的内核扩展机制。我们使用胶囊来实现系统调用驱动程序、虚拟化器（在单个硬件外设上复用多个相互独立用户的组件），以及其他非特权内核基础设施。

胶囊的特别之处在于，我们利用 Rust 的内存安全和类型安全保证来实现隔离。我们将胶囊实现为 crate（Rust 的编译单元）中的 Rust 结构体，并禁止使用 `unsafe` Rust 代码。这样，每个胶囊只能访问自身状态，并通过实例化胶囊时提供的明确且安全的应用程序接口与系统其他部分交互。

这种方法能够在编译时实施细粒度隔离，且几乎没有运行时开销。我们只要求胶囊保证活性（例如，不要忙等或 `panic` 内核）。Tock 中的大多数非特权代码都通过胶囊实现；只有底层芯片支持代码、内存映射 I/O 外设驱动程序和核心内核基础设施需要以其他特权的非胶囊代码实现。

2.3 硬件隔离的用户空间进程

Tock 还可以运行抢占式调度且由硬件隔离的用户空间进程。最初，设计这些进程是为了将嵌入式系统中常见的复杂状态机拆分为独立的应用程序，而无需将所有代码都重写为 Rust。我们的愿景是通过支持多道程序设计，让多个以不同编程语言（例如 C 和 Lua）编写的独立应用程序在同一个利用率不足的嵌入式平台上运行，从而利用大多处于空闲状态的微控制器实现新功能。这最终促成了针对应用程序的形式化威胁模型，其中规定：

应用程序数据必须保密，即使是对内核也如此，除非应用程序明确将其公开。我们通过向胶囊提供有限且安全的应用程序接口来实现这一点。此外，应用程序在运行时不能相互拒绝服务——换句话说，它们是彼此不信任的。

2.4 使用无堆内核和授权实现可靠性

在内存受限的硬件平台上支持多个具备高可用性的应用程序，会带来一项可靠性方面的关键挑战：应用程序可能通过持续向内核请求资源而耗尽内存。具体来说，内核组件通常需要维护与……相关、但用户空间应用程序无法访问的状态。如果使用共享内核堆进行此类分配，一个应用程序就可能耗尽这些资源，从而损害其他应用程序的可用性。

Tock 通过一项关键设计特性授权来防止这种情况：内核没有全局堆，只依赖静态内存分配和自身的栈。所有动态内存分配都放在进程的内存区域中，并通过平台的内存保护机制确保用户空间无法访问。这确保进程只能耗尽自身内存，且与应用程序相关的状态不会超出进程生命周期。

2.5 全面异步

为满足多道程序设计和低功耗方面的约束，我们为 Tock 采用了完全异步的内核架构。这使其能够充分利用异步硬件（例如支持 DMA 的外围设备），在不同应用程序的任务之间进行多路复用，并在没有剩余任务时让硬件进入低功耗休眠状态。Tock 采用分阶段异步机制，如 TinyOS [23], ，而非微内核中更常见的消息传递；后者更容易依赖动态内存分配 [1, 18–21， 25, 32, 35]。

值得注意的是，Tock 的构建早于 Rust 原生支持异步编程模型，因此它采用单栈架构，并通过内核组件之间的循环引用和回调来通知操作完成。 第 5.3 节将探讨这一设计的影响。

然而，Tock 并未止步于此：除了实现完全异步的内核，我们还让系统调用接口也完全异步。要执行任何耗时较长的操作，例如通过通用异步收发传输器打印一个字节缓冲区，应用程序会向内核发出一系列调用：与内核共享数据缓冲区，注册一个回调（称为“上调”），供内核在操作完成时调用，然后请求开始执行该操作。从此以后，应用程序可以执行其他任务，甚至可以同时启动另一个异步操作。最后，应用程序可以 让出执行权给内核，以等待操作完成，从而让系统进入低功耗状态，并使内核最终调用已注册的回调。

3 演进

如今，Tock 的主要用途（学术界以外）是用于信任根硬件平台，包括双因素身份验证（2FA）设备，以及笔记本电脑和服务器中的安全协处理器。这些应用场景与最初的传感器网络场景具有相似的硬件约束和应用需求。不过，它们也存在显著差异。这些系统既出乎意料地适配了 Tock 原始设计的某些部分，也要求对其他部分作出修改。有一个案例中，Tock 的通用性过于繁重，难以满足产品需求，最终导致了硬分叉。另一个案例中，一家公司使用 Tock 几个月后，转而为其服务器的硬件信任根设计了另一款定制 Rust 操作系统。

3.1 硬件信任根

信任根硬件设备在许多方面与传感器网络硬件相似。它们通常只有很少的内存（几十或几百 KB），并使用单个 CPU 核心，通常为 32 位。它们没有虚拟内存，但支持单地址空间内存保护。它们包含许多专用的、有时是专有的加速器和外围设备，尤其用于密码学。

它们的软件需求也与 Tock 最初为之设计的城市感知网络相似。硬件信任根通常运行多个独立开发的应用程序，而这些应用程序需要能够跨硬件版本移植。它们通常需要支持多种应用程序语言：虽然 Rust 对新服务和应用程序很有吸引力，但用户也希望继续使用现有的、经过充分测试且可靠的 C 应用程序。应用程序和操作系统都能从具备抵御应用程序及驱动程序漏洞的能力中大为受益。

早期采用者包括谷歌的两款产品：符合 FIDO 标准的双因素身份验证设备 OpenSK[15], ，以及用于 Chromebook 笔记本电脑的硬件信任根 Ti50 [16]。随后，数据中心服务器硬件信任根供应商以及其他终端用户设备制造商也开始采用。

经过数年的合作与开发，我们逐渐适应了硬件信任根与城市感知网络之间的若干重要差异。其中一些差异与 Tock 的设计轻松兼容，另一些则引发了矛盾。本节将重点介绍其中最重要的差异，以及这些差异如何改变了 Tock 的设计。

3.2 异步代码的复杂性

Tock 最初旨在支持超低功耗运行，这是无线传感器网络和 Signpost 系统的常见要求。此前的操作系统和应用程序已经证明，异步 I/O

能够显著节省能量，因为单个线程可以并行执行许多操作，从而减少唤醒时间 [22]。因此，Tock 同时采用了事件驱动的单线程内核和异步系统调用接口。

然而，硬件信任根要么不受能量限制，要么即使受限，程度也远低于无线传感器网络。笔记本电脑或服务器上的硬件信任根，其能耗与整个系统相比微不足道。基于 USB 的双因素身份验证设备通过 USB 端口获得稳定电力；即使是无线蓝牙设备，也无需严格优化其占空比，因为占空比更多取决于人的反应时间（秒），而非 I/O 时序（微秒至毫秒）。

因此，尽管异步内核让 Tock 能够通过单个栈支持许多同时运行的应用程序和服务，但对许多用户而言，它在用户空间中带来的阻碍多于益处。信任根应用程序通常是顺序状态机。一个简单的同步操作，例如“等待响应并设置超时”，可能会变成六七次系统调用：分配缓冲区、注册两个回调、发出命令，然后等待回调。

这种不匹配促使 Tock 的早期采用者之一 Oxide 在使用几个月后放弃了它。他们原本想将 Tock 用于其基板管理控制器（BMC）的固件。不过，他们意识到自己的应用程序集合非常固定，而且全部依赖简单直接的顺序状态机。他们发现，内核需要处理异步操作交错所带来的复杂性，而用户空间又需要将其转换为同步 API，这同样很复杂，因此决定用 Rust 编写一个新的同步操作系统，名为 Hubris [9]。

Tock 的另一位早期采用者 Ti50 很早就创建了分支，并持续参与了多年；但随着其分支与 Tock 主线逐渐分化，他们的参与也日渐减少。他们同样对异步系统调用有所顾虑，但原因不同：代码空间。他们的 MCU 基于 RISC-V，而 RISC-V 的 LLVM 代码生成相对不成熟，可能导致系统调用开销很高。与此同时，他们交付固件的时间极其紧迫。于是他们分叉了 Tock，引入了一个阻塞式 系统调用。这样一来，他们就能将由 4 个系统调用（subscribe、command、yield、unsubscribe）组成的序列合并为一次调用。虽然其他开发者也有兴趣在主线内核中引入这种新系统调用，但 Ti50 创建分支后，并没有积极推动此事，最终这项提议也就不了了之。此后，我们引入了一种新的 系统调用变体，将在下文介绍；它简化了使用异步系统调用接口创建同步库的过程。

如今，尽管一些 Tock 用户对采用同步系统调用还是异步系统调用持观望态度，但另一些用户认为它们对于其信任根和汽车应用至关重要。然而，异步机制还带来了额外的复杂性，因此必须彻底重新设计系统调用 API。我们发现，原系统调用的异步语义与健全的 Rust 用户空间不兼容，除非大量使用不安全代码。

3.3 用户空间中的 Rust 与 Tock v2.0

Tock 最初的设计旨在允许用户空间进程使用任何语言（包括汇编语言）编写，但几乎所有应用程序都是用 C 编写的。早期非学术界的采用者表示，他们也希望能在用户空间中使用 Rust。虽然 Tock 有一个 Rust 用户空间库，但我们很快发现它并不健全。要以尽可能少的不安全代码妥善支持 Rust 用户空间，就必须重新设计系统调用 ABI、内核核心循环和胶囊接口。

3.3.1 遵循所有权

由此产生的挑战是，必须在系统调用 ABI 中体现 Rust 的内存所有权语义，并由内核本身加以强制执行，而不是将其托付给胶囊。 （共享缓冲区）和 （注册回调）这两种系统调用都允许用户空间共享其内存中资源的引用：对于 ，共享的是数据；对于 ，共享的是代码。最初的胶囊 API 只是验证提供的值，然后将封装这些引用的类型的所有权移交给胶囊代码，即 Tock 半可信的内核扩展。

问题出现在用户空间尝试通过再次调用 或 来撤销其中一个缓冲区或函数指针时。虽然系统调用语义规定，必须使用胶囊来将旧值替换为新值，但编译器无法强制执行这一点：胶囊拥有这些包装器，并且总能把旧的包装器存到某个变量中。Rust 用户空间必须能够仔细跟踪哪些对象持有对哪些资源的引用，以及这些值的生命周期；但胶囊提供的是一种不透明的接口，可以任意延长持有引用的时间。半可信的内核扩展若任意持有并使用用户空间引用，就会破坏 Rust 的所有安全性保证，从而大大降低 Rust 用户空间对安全关键型应用程序的价值。

3.3.2 交换语义

如今， 和 系统调用 API 采用“交换”语义，类似于 TinyOS 管理缓冲区的方式 [23]。当用户空间调用其中一个接口时，该调用会返回先前共享的缓冲区或函数指针。首次调用时，内核会返回一个表示空值的占位值：对于 ，这是一个长度为零的缓冲区；对于 ，则是一个特殊的“null”回调。用户空间负责确保

当内核持有引用时，这些引用始终有效。在内核中，胶囊不再负责管理这些缓冲区或回调；它们由内核持有和管理。胶囊只能通过闭包中的临时 Rust 引用访问它们，这确保胶囊无法取得这些类型的所有权。最后这项变更要求彻底重写内核核心循环。

3.3.3 Tock v2.0

这次对系统调用 API（及 ABI）的重新设计意味着，现有应用程序将无法继续运行。我们曾讨论过让两个版本并行存在，但代码大小的代价过高。因此，Tock v2.0 做出了彻底的重大变更。除了 和 的交换语义外，Tock 还新增了 的一个变体，使内核能够以只读方式访问用户空间缓冲区。事实证明，这对于信任根应用程序不可或缺：它们通常会将公钥存储在只读闪存中，并希望将这些值传入内核以执行加密操作。如果没有 ，用户空间就必须将这些值复制到 RAM 中；否则，内核可能会无意中尝试写入只读内存，从而触发不可恢复的故障。

3.4 进程加载

Tock 的进程架构支持彼此独立的应用程序，内核可以分别加载、启动和停止它们。此架构旨在支持现场部署的灵活性（例如 Signpost）。在早期的一种信任根应用中，Tock 并非最底层的可信软件。相反，可信引导加载程序会验证一个包含 Tock 和一组应用程序的签名映像。虽然这要求将它们重新打包在一起并添加签名，但这些改动都在 Tock 之外，因此无需修改内核本身。

其中一种信任根用例引入了新的安全模型，允许对应用程序分别签名和替换。这意味着 Tock 内核必须能够在执行进程二进制映像之前，对其进行一系列密码学检查。由硬件外设实现的密码学操作是异步的，因此进程加载序列不得不从简单地同步检查标头和完整性，改为一个多步骤状态机。首先检查进程的结构和标头完整性，然后检查密码学完整性和真实性，最后检查其是否可运行。为此，还必须将 Tock 的引导序列从一段顺序执行的代码改写为异步状态机。

转而支持签名应用程序，带来了一个主要好处和一个重大缺点。好处是，这为动态加载并运行新应用程序铺平了道路，无需重新启动。此前，一些用户一直认为这项功能值得拥有，但并非必不可少。由于进程加载如今采用异步状态机，这件事变得容易了许多：系统只需触发内核检查新进程即可。缺点则是代码复杂度增加。对于代码空间有限且不需要动态进程加载的系统（例如使用单个签名映像的系统），新方法增加了大量不必要的代码。因此，内核支持这两种方式，并可在编译时进行配置：构建版本的引导序列可以调用同步加载器，也可以调用异步加载器。

3.5 外部依赖

Tock 发展过程中的最后一项变化，涉及许多安全的 Rust 系统都曾面临的一个问题：外部依赖。由于 Rust 代码可以执行任意操作，因此引入外部依赖就意味着必须信任它不会破坏系统所依赖的安全性保证。例如，Rudra 展示了许多外部依赖可能被用来破坏内存安全性的情形 [6]。最初，Tock 对此采取了非常严格的立场：唯一允许的外部依赖是 Rust crate，它属于 Rust 标准库的一部分。

然而在实践中，第三方库提供了大量有用的功能，其中许多功能难以或不适合重新实现或纳入项目。例如，信任根应用场景严重依赖密码学，而且往往需要在特权内核驱动程序中使用。在某些情况下，硬件会提供密码学原语，但更高层的协议（如流密码、签名验证或证书验证）通常必须由软件实现。密码学的正确实现不仅关乎内存安全，还对安全性有着更高要求；因此，重新实现密码或协议，或自行维护实现并不可取，尤其是在已有维护良好的第三方库，甚至是经过形式化验证的第三方库时 [13]。对于某些依赖树有限且维护可靠的密码学库，Tock 对外部依赖规则作出了明确例外。

不过，在更多情况下使用第三方库会很有用，例如网络和无线协议。信任第三方库仍然是一个尚待解决的研究问题 [26,34]。

4 发现 Rust 的机遇

Rust 富有表现力的类型系统为 Tock 的开发带来的益处不止于内存安全和类型安全保障。我们能够利用类型系统对操作系统和嵌入式系统设计中的更高层次属性进行编码，从而在编译时检查无效和不可接受的系统组合，改进缓冲区管理，并提供富有表现力的硬件接口。

4.1 强制正确组合

Tock 支持多种硬件平台，并且设计上可配置，以适配种类丰富的设备。我们

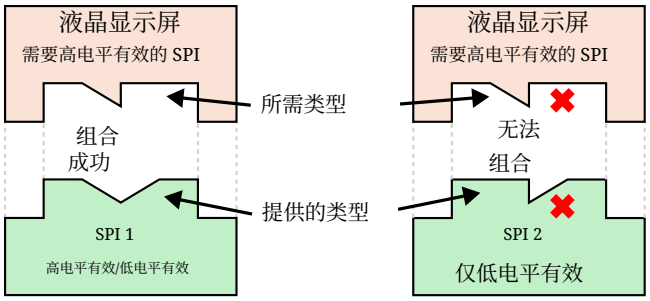

图3：在配置时强制执行分层要求。更高层、与硬件无关的驱动程序会指定自身需求，并基于较低层、表达其能力的外设驱动构建。将这些要求纳入 Rust 类型系统后，Tock 便能在编译时确保配置有效。

通过将系统组件划分为可组合的层来实现这种灵活性。例如，硬件定时器外设实现了一种通用接口，允许客户端以与硬件无关的方式与该外设交互。随后，另一层可以利用此接口，为多个客户端对单个定时器资源进行虚拟化。

然而，这种灵活性也带来了挑战：为某个硬件平台配置 Tock 内核实例时，必须正确匹配与微控制器无关的驱动程序（例如通过 SPI 连接的传感器）和相应的特定硬件外设实现（例如 SPI 控制器）。在这种组合中，各层假设和限制即便只有细微差异，也可能导致（不易察觉的）错误行为。驱动程序通常位于不同文件中，由不同开发者在不同时间编写。错误地认定某种驱动程序堆叠有效，是一种很容易犯、却很难调试的错误。

相反，我们发现可以利用 Rust 的类型系统在编译时验证组合是否正确。具体而言，我们允许驱动程序通过类型注解表达其所提供接口的高层特性。这些接口的客户端则对类型约束进行注解，从而只接受提供兼容语义的驱动程序。

例如，SPI 通信总线 allow 多个外部设备通过共享总线与微控制器通信。每个设备都有一个独立的“片选”（CS）引脚，用于指示软件已选择该设备使用总线。不同设备要求 CS 引脚为高电平有效、低电平有效，或可配置。这些信息与微控制器无关，可由更高层的驱动程序编码。类似地，有些 SPI 硬件仅支持高电平有效引脚、仅支持低电平有效引脚，或支持配置有效片选电平。即使在同一个微控制器中，多个 SPI 外设也可能有不同的此类限制。这是微控制器专用硬件驱动程序应了解的信息。要正确组合，SPI 驱动程序必须

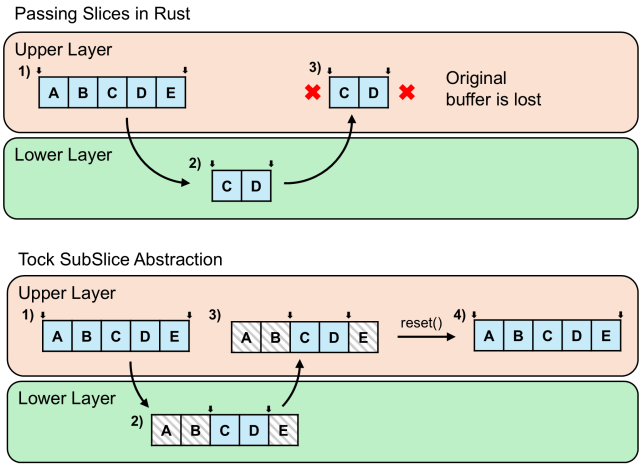

图4：Tock 的 抽象允许内核组件来回传递缓冲区、随意调整其大小，同时保留恢复对完整底层缓冲区访问的能力。

既支持与其连接的外围设备相同的有效电平极性，也配置为该极性（图3）。

通过在 Rust 类型中使用模板常量，我们可以表达硬件驱动程序的能力以及特定于芯片的驱动程序的要求，并确保它们得到正确配置。不匹配会在编译时通过类型错误被捕获。

4.2 缓冲区管理

为了支持内核中的并发，同时仅使用一个栈，Tock 内核 API 大量依赖分阶段异步机制。分阶段异步操作包含两个部分：用于启动操作的方法调用，以及用于表示操作完成的回调。

Rust 的缓冲区管理使分阶段异步 API 变得复杂。在 Rust 中表示缓冲区的规范方式是使用切片，它由一个指针和长度组成。Rust API 通常将切片作为临时引用传递，其生命周期会在函数返回时结束。然而，异步操作必须能够一直持有该引用，直到操作完成。例如，支持 DMA 的无线电驱动程序需要在 函数调用结束后继续对缓冲区保持唯一的“所有权”，直到异步中断表示已接收到完整的数据包。

Tock API 通过传递具有 生命周期的切片引用，使子系统能够持有缓冲区，超出函数调用的持续时间。 生命周期是 Rust 中一种特殊的生命周期，表示该引用在整个程序运行期间都有效（Tock 会在启动时静态分配其缓冲区）。由于 Rust 所有权规则禁止存在多个唯一的可变引用，将一个 引用传递给驱动程序后，调用者将无法再访问它；

从概念上说，调用该方法会将缓冲区的所有权“移动”到驱动程序中。驱动程序会在完成回调中将这个永久引用传回客户端。

然而，当尝试传递动态数据子集时，这种跨软件层传递缓冲区所有权的方法就会失效。调用方不能简单地传递缓冲区的一个子集（即递增起始指针），因为这样就意味着永远放弃该缓冲区这部分内容的所有权，而且源自同一缓冲区的两个子切片之后也无法重新合并成原来的较大缓冲区。每一层都必须处理完整的缓冲区，以便将其整体归还给最初的调用方。Tock 内核接口的早期版本尝试解决这一问题：除了接受切片外，还接受单独的偏移量和长度，用来描述要操作的缓冲区子集（例如，要使用密钥加密的缓冲区部分）。

遗憾的是，要求接口同时传递切片、子集偏移量和子集长度，既繁琐又容易出错。Tock 引入了两种机制来缓解这一问题。第一种是类型（图4），使各层能够像访问长度恰好等于有效数据长度的缓冲区那样访问缓冲区，同时保留整个缓冲区的所有权。第二种机制使用参数，使软件层能够表达缓冲区大小要求。这些参数在编译时即可获取，因此高层可以构造静态缓冲区，确保其大小足以满足各项依赖的要求。

4.3 内存映射 I/O 抽象

微控制器通常使用内存映射 I/O (MMIO) 与硬件外设进行交互。外围设备会占用一些内存地址，并在这些地址上响应内存读取或写入事件，通常会触发底层硬件操作。将数据手册中的硬件规格转换为原始内存地址和紧凑结构体，以正确映射硬件的内存布局，并为每个内存映射寄存器建模访问权限，既繁琐又容易出错。

Tock 利用 Rust 宏系统提供了一种用于描述内存映射 I/O（MMIO）的领域专用语言。这种高级规范与典型的数据手册十分接近，因此很容易将其转换为代码并避免出错。生成的数据结构会将每个 MMIO 地址封装为一种唯一类型，该类型仅提供该地址所支持的读取或写入操作。这些特定于地址的类型还可以定义子类型，s，用于编码有效枚举值，以及内存字中的偏移量和长度。现在，位移代码会自动生成，无需再手动编写容易出错的代码。

```
// To call this function, the caller must
// have a ProcessMgmtCap instance to pass in,
// which is an empty marker trait that is
// otherwise unused and elided at runtime.
fn destroy_process(&self, _cap: &ProcessMgmtCap) {
    }// Capabilities can only be created during
// platform initialization. They can then be
// passed into privileged drivers.struct MyCap;
unsafe impl ProcessMgmtCap for MyCap;
let privileged_driver = ProcessManagementDriver {
  cap: MyCap,


}
```

图 1：Tock 使用标记特征来限制对内核中敏感 API 的访问，使不受信任的胶囊能够获得特殊权限，但前提是受信任的平台初始化创建并传递一个仅在编译时有效的能力。若尝试在没有该能力的情况下调用 API 函数，编译器将在编译时报告错误。

4.4 使用能力的特权 API

Rust 区分安全代码和 代码，编译器也可以按 crate（*Rust* 编译单元）分别拒绝代码。这使 Rust 代码能够拥有受限 API——标记为 ——只能在特定上下文中调用。对于操作系统而言，这尤其有用，因为某些特权功能必须对外提供（例如，停止进程），但许多与进程交互的内核模块（例如，实现用户空间计时器的模块）绝不能调用这些功能。不过，虽然将函数标记为 可提供强制执行机制，但它只提供非此即彼的访问控制，无法根据上下文决定何时授予访问权限。Tock 通过“能力”机制（图 1）解决了这一问题。该机制基于编译器强制执行的 ，创建零大小类型（因此在运行时没有额外开销），以表达受限 API，并利用类型系统来管理使用这些 API 所需的权限。

5 Rust 中发现的陷阱

Rust 在实现操作系统方面具有诸多优势，但也给编写底层代码带来了独特挑战。在 Tock 的开发过程中，我们发现，构建操作系统时沿用的传统经验和做法，有时已不再适用，甚至可能导致违反 Rust 为保障安全性和健全性所要求的一些不变量。

5.1 面向操作系统的 Rust 内存模型

Rust 在安全性方面的许多创新，都着眼于对内存使用方式施加严格限制。例如，回想一下，Rust 值可以拥有多个不可变共享引用，也可以拥有一个可变引用，但绝不能

两者兼有。编译器通过生命周期和借用机制检查这些不变量。

然而，在操作系统环境中，这些规则很难强制执行。运行进程这一概念，加上应用程序代码并非用 Rust 编写，导致编译器无法掌握足够信息来强制维护内存不变量。Theseus [8] 等替代方案通过要求所有操作系统组件和应用程序都使用 Rust 编写，并利用编译器检查来避免这些问题。然而，我们的经验表明，这并不实用：现实世界中的设备必须能够运行用 C 编写的旧版软件，而进程隔离也能带来至关重要的优势。

相反，Tock 采用硬件内存保护机制和多种特权模式，来隔离不可信的用户空间应用程序。因此，Tock 必须在 Rust 的静态编译时安全不变量与用户空间—内核边界之间建立衔接：要么削弱 Rust 预期得到维护的不变量集合，要么在内核与用户空间组件的交互过程中执行动态检查。实际上，Tock 两种方法兼而用之，以便在与不可信代码交互时高效地维持 Rust 的健全性。我们将通过重新审视 Tock 的 机制来说明这一点，该机制用于在内核与用户空间应用程序之间交换数据 [24]。

5.1.1 可变别名应用程序内存

当 Tock 应用程序发起一次 系统调用时，它会传递一个指针和长度，以便与内核胶囊共享缓冲区。虽然胶囊在内核模式下运行，能够访问所有用户空间内存，但由于只能使用安全 Rust，它们无法访问任意内存。应用程序通过显式 *allow* 缓冲区，授权这些内核组件读写其内存中的特定区域。在内核中，我们通过一个 句柄来表示此类缓冲区；在确认发起调用的进程仍然存活后，便可使用该句柄安全地获取指向其底层内存的 Rust切片引用。

这种机制与其他操作系统中内核和用户空间应用程序共享内存的方式类似。然而，Tock 使用 Rust 带来了一些额外挑战，导致我们最初的实现并不健全。首先，应用程序可以通过两次*allow*系统调用共享同一个缓冲区，使内核胶囊获得指向同一内存的两个可变引用。这破坏了 Rust 的别名 *XOR* 可变性不变式。写入第一个缓冲区可能会改变第二个缓冲区的内容，而编译器却假定第二个缓冲区的内容保持不变。这种做法并不健全；实际上，违反 Rust 的别名假设曾导致可被利用的安全漏洞 [28]。

为了解决这个问题，理想情况下，Rust 编译器可以静态地确定两个共享缓冲区绝不会重叠，但由于应用程序是分别编译的，这一点无法实现。另一种做法是让 Tock 通过运行时检查来拒绝重叠的共享缓冲区。然而，这会带来对于 Tock 面向的系统而言，难以接受的运行时开销。因此，我们选择减少 Rust 要求我们为共享缓冲区维护的不变量。通过将其类型签名从可变字节切片（ ）改为对内部可变的单元格（ ）的共享切片，Rust 不再假定缓冲区的内容在其引用存在期间保持稳定。

5.1.2 违反 Rust 的类型不变量

与内核共享进程内存，引发了另一个问题：如何满足 Rust 对健全性的严格要求。应用程序通过再次发出缓冲区长度为零的 allow 系统调用，从内核收回共享缓冲区。内核持有对此零长度缓冲区的引用，即使其地址为 null 也不例外。然而，这并不健全：Rust 要求引用的地址绝不能为 null。基于这一要求，Rust 会执行一种称为*利基填充*的优化：它可以利用某类型的不可居值来编码其他信息。例如，既然引用绝不能为 null，那么类型为 的值与类型为 的值占用相同的空间，因为 的 变体可以直接编码为 null 值。然而，这意味着 Tock 将任意的用户提供指针用作零长度切片引用地址的做法，违反了 Rust 的类型安全性。相反，我们必须在此操作中进行动态运行时检查，并使用一个人为构造的非 null 指针来创建与内核共享的零长度切片引用。

5.1.3 强调 边界

这类错误表明，虽然 Rust 能让开发者对程序中的安全部分充满信心，但要确保安全代码的边界维护 Rust 所要求的大量安全性不变量，也需要格外谨慎。在许多情况下，C 等较早出现的语言所要求的不变量较少（例如，关于别名的不变量）。而且，尽管违反这些不变量的后果往往很隐蔽，且取决于编译器优化，但仍可能在代码库中任意的安全和不安全部分引入严重的安全性漏洞。

5.2 缺乏 指导

如上一节所述，明确区分安全代码和不安全代码，既是正确使用 Rust 的一项重要特性，也是必要条件。然而，在 Rust 开发的早期阶段，几乎没有关于如何以及何时在大型软件项目中使用 ，以确保其长期可维护性和正确性的指导。我们的经验表明，对操作系统进行模块化，并禁止在某些 crate 中使用 ，可以为开发者提供明确的指导，并鼓励与 Tock 设计保持一致的外部贡献。然而，在确实允许使用 的 crate 中，我们发现开发者经常以新的方式使用 ，

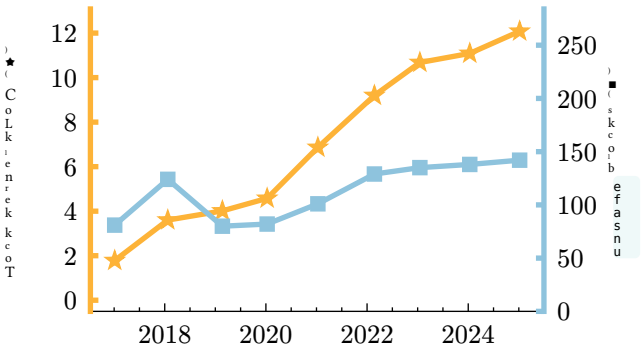

图 5：随着时间推移，由于增加了功能和抽象，Tock 内核的规模不断扩大；与之相比，使用 Rust 编写的内核代码量如何变化。尽管内核在十年间显著增长，代码块的数量始终保持稳定。

尤其是为了支持定制的硬件特性，然后在这种用法之上分层构建实现。随着时间推移，这导致出现了大量既难以分析、又难以移除的用法。

与此同时，Rust 采用了更狭义的安全性定义，具体聚焦于这样一种理念：安全代码不会出现未定义行为[30]。这与功能正确性的其他形式不同，Tock 有时会用来实现这些形式。

事后看来，我们本应围绕构建更多抽象，既为贡献者提供指引和支持（例如 Tock 的能力），也限制内核中对的多样化使用。尽管该项目成功地让在内核中的使用保持稳定且处于较低水平，但要限制其普及仍需持续进行维护工作。

5.3 使用外部软件库

尽管 Rust 拥有健全的包管理和依赖生态系统，但就复用现有驱动程序和软件库而言，Rust 对 Tock 的帮助仍然非常有限。

一个主要挑战在于，Tock 内核的异步、单栈设计与现有库不兼容：大多数现有代码要么采用阻塞架构，要么与 Tock 的多编程模型不兼容，要么使用 Rust 的来进行异步计算。然而，我们发现，与基于回调的 Tock 设计相比，存在显著的额外开销。

此外，正如第 3 节所述，我们谨慎地将外部代码集成到 Tock 中，因为代码和依赖树都难以审查。是否存在将外部代码安全地集成到操作系统内核中的方法，仍是一个开放问题。我们预计，需要借助更多工具，分析并持续监控外部软件包在长期软件包更新过程中的情况。

5.4 验证仍然有用

Rust 的类型系统为编写健壮的代码提供了强大基础，并能激励开发者这样做。然而，逻辑正确性完全取决于程序员的责任。对于正确性要求微妙复杂的子系统，我们发现 Rust 并不能显著帮助避免错误。尤其是定时器虚拟化和内存保护这两个子系统，已经出现了许多细微的逻辑错误。我们正在探索将验证技术集成到 Rust 中，以提高这些具有挑战性的子系统的正确性。

6 学术界对开源生态系统的支持

如今，Tock 是一个由学术界主导的开源社区。它从未因技术转移而被一家工业用户独占，也没有成为一家专注于为操作系统提供支持或开发服务的初创公司。相反，它一直在学术管理和工业部署之间寻求平衡。目前领导团队中超过四分之三的成员是学者（其余四分之一来自工业界）。我们在学术界维护一个开源、面向工业部署的操作系统的经验，带来了若干挑战和关键益处，本文将在此进行探讨。

6.1 学术开源的挑战

在某种程度上，Tock 面临的一些最大挑战，刻板地说，是那些已有已知实现方案的“已解决问题”，但要应用这些方案仍需投入大量工程工作。我们曾多次看到，当问题不再处于艰难而有趣的阶段后，学术界的关注便逐渐消退。让负责特定任务的工程师来处理，就能轻松克服这一问题。但对于主要受研究动机驱动的独立贡献者来说，这就困难得多。最终，这可能导致整个系统的支持出现缺口。

举例来说，项目长期以来的一项需求是建立硬件在环测试系统。要支持多种架构、芯片和特定的电路板实现，就需要频繁地在真实硬件平台上进行测试。实际上，我们发现，一些较少使用的平台可能数周甚至数月无法正常运行，却一直无人察觉，直到有人偶然发现这个问题。自动化硬件测试框架显然是解决方案，但创建它需要投入巨大的工程工作，却没有明确的研究问题。与此同时，产业采用者有兴趣使用这一框架，但没有动力创建一个通用系统，来测试他们并不使用的硬件。经过多年的需求积累，最终通过委托一名全职工程师承担这项工作，才开发出了该框架。

学术界与工业界的合作也可能面临典型的时间安排不匹配。面对产品截止期限，产业采用者会有一些功能需求，无法等到

学术界考虑和讨论之后再解决。对于开源项目而言，这往往会导致出现分支，或在操作系统中添加下游定制内容。在一定程度上，这些情况是意料之中且可以接受的。Tock 的设计有意将各个组件分离，以便在必要时替换。但产业采用者实施的下游改动并不一定会回馈给更广泛的社区。有好几次，我们发现自己不得不重新实现一些功能，而这些功能此前已由工业界用户实现，却从未经过整理并分享给社区。

6.2 学术界开源的优势

扎根于学术界也为 Tock 带来了重要益处。学术界的庇护让 Tock 得以度过开发的最初几年。对于开源项目而言，在外部用户最初采用之前是一段充满挑战的时期。它们缺少采用者带来的能量和方向。对于操作系统而言，尤其棘手的是，早期操作系统只能以基本功能支持少数几个平台。通过增加驱动程序和平台支持，逐步发展到临界规模需要时间。在这一初期阶段，Tock 若必须证明自身价值以获得初创企业资金，将会非常困难。相反，Tock 多年来几乎完全没有资金。这让我们有空间进行试验和开发，而资金来源则是我们在学术界的日常工作。

Tock 发展到后期，在我们建立起用户群并获得支持后，Tock 的学术核心为我们提供了长远视角，使我们不受生产期限的制约。对某个用户来说，内部有用且便于实现的做法，往往并不适合其他用户或未来可能采用它的用户。在使用 Rust 开发系统时，时间紧迫可能会让人更想通过 逃生舱绕过编译器。这样做是用眼下看似运行良好的方案，换取未来可能出现的健全性问题。相反，Tock 的核心开发团队有底气花时间把“事情做好”。

第3.3节中提到的系统调用 API 更新，就是一个很好的例子。早在 2019年初，人们就意识到，为了支持 Rust 用户空间，需要修订系统调用接口。其他有关系统调用接口的问题也被收集起来，并于一年后的 2020年初正式启动了内核 2.0版本的工作。修订后的系统调用接口文档初稿于 2020年年中完成。Tock 2.0 最终于 2021年年中发布。修订后的设计经过了充分的讨论和审议，才最终部署；我们并不急于完成它，因为操作系统内核中的重大破坏性变更并不常见。不受生产期限和初创企业资金周期的影响，让我们有时间和耐心进行深入的技术讨论与考量。

在学术界托管的另一个好处是，可以让本科生参与项目。这些学生能够接受真实系统工程方面的培训，而 Tock 则获得了工程工作；此外，还有近乎无穷无尽的驱动程序可添加到操作系统中，但对任何特定用户而言，这些驱动程序都不是当务之急。许多功能都是由希望积累经验的本科生开发的。

对于研究生而言，Tock 已被证明是一个极佳的资源，能让他们探索难度较高、但范围仍相当受限的系统设计考量。多年来，研究生们使用 Tock 进行实验，例如通过自动时钟配置来节省能量 [10], 、在 Rust 系统中隔离现有的 C 语言库 [31], 、利用 Rust 类型系统在编译时自动强制执行缓冲区大小要求，以及缩减 Rust 系统的代码大小 [5]。最近的一项合作中，研究生们致力于代码验证系统，并对 Tock 所提供的受限接口和安全保证颇感兴趣 [29]。

7 结论

经过十年的开发、多个重要版本发布、数千次提交和拉取请求、数次开发者会议、数位研究生毕业并取得博士学位以及新教师加入，如今又有数千万台活跃设备在使用 Tock，本文分享了 Tock 设计的演进历程、我们使用 Rust（无论正确还是错误）的经验，以及该项目如何从学术界和工业界受益，又如何服务于二者。

我们希望这些技术细节和经验能为考虑使用 Rust 的操作系统开发者提供指引，并为实践者部署研究型操作系统的持续开发提供参考路径。

8 致谢

我们感谢指导编辑和匿名评审，他们就如何传达本文中的经验与教训提供了宝贵的反馈。

至少有 400 人为 Tock 开源项目作出了贡献。特别是 Johnathan Van Why、Hudson Ayers 和 Alexandru Radovici，他们是其他核心维护者，参与了本文所述的项目发展历程。Tock 最早的设计迭代得到了 Michael Andersen 的鼓励和影响，他还设计并构建了 Tock 最初针对的硬件平台。Niko Matsakis 最初建议在 Tock 的并发模型中采用内部可变性。Dominic Rizzo、Alyssa Haroldson、Bryan Cantrill 和 Bobby Reynolds 鼓励 Tock 的商业应用，并提供了大量反馈，这些反馈推动了项目的发展。

这项工作部分由美国国家科学基金会的资助项目 2303639、2443589、2144940、1505728 和 1505684，以及 Google Research Award 资助。

引用

[1] Mike Accetta、Robert Baron、William Bolosky、David Golub、Richard Rashid、Avadis Tevanian 和 Michael Young。1986年。Mach：面向 UNIX 开发的新型内核基础。载于《*USENIX 1986*年夏季技术会议暨展览会论文集》，1986年。USENIX协会，美国加利福尼亚州埃尔塞里托、美国佐治亚州亚特兰大，93–112。[2] Joshua Adkins、Branden Ghena、Neal Jackson、Pat Pannuto、Samuel Rohrer、Bradford Campbell 和 Prabal Dutta。2018年。面向城市规模传感的 Signpost平台。载于《*2018*年第*17*届*ACM/IEEE*传感器网络中的信息处理国际会议（*IPSN*）》，2018年。188–199。10.1109/IPSN.2018.00047[3] Apache软件基金会。Apache NuttX。检索于2025年8月31日，网址：https://nuttx.apache.org/ [4] Hudson Ayers、Prabal Dutta、Philip Levis、Amit Levy、Pat Pannuto、Johnathan Van Why 和 Jean-Luc Watson。2022年。面向实用嵌入式系统安全的分层信任。载于《第*15*届欧洲系统安全研讨会论文集》（*EuroSec '22*），2022年。计算机协会，法国雷恩，15–21。10.1145/3517208.3523752[5] Hudson Ayers、Evan Laufer、Paul Mure、Jaehyeon Park、Eduardo Rodelo、Thea Rossman、Andrey Pronin、Philip Levis 和 Johnathan Van Why。2022年。收紧 Rust 的腰带：缩减嵌入式 Rust 二进制文件。载于《第*23*届*ACM SIGPLAN/SIGBED*嵌入式系统语言、编译器与工具国际会议论文集》（*LCTES 2022*），2022年。计算机协会，美国加利福尼亚州圣迭戈，

121 132 10 1145 3519941 3535075 – 。 . / . Yechan Bae、[6] Youngsuk Kim、Ammar Askar、Jungwon Lim 和 Taesoo Kim。2021年。Rudra：在生态系统规模上查找 Rust 内存安全漏洞。载于《*ACM SIGOPS*第*28*届操作系统原理研讨会论文集》（*SOSP '21*），2021年。计算机协会，线上活动，德国，

84 99 10 1145 3477132 3483570 – 。 . / . [7] Brian N. Bershad、Stefan Savage、Przemysław Pardyak、Emin Gün Sirer、Marc E. Fiuczynski、David Becker、Craig Chambers 和 Susan Eggers。1995年。SPIN操作系统的可扩展性、安全性与性能。载于《第*15*届*ACM*操作系统原理研讨会论文集》（*SOSP '95*），1995年。计算机协会，美国科罗拉多州铜山，267–283。10.1145/224056.224077[8] Kevin Boos、Namitha Liyanage、Ramla Ijaz 和 Lin Zhong。2020年。Theseus：一次关于操作系统结构与状态管理的实验。载于《第*14*届*USENIX*操作系统设计与实现研讨会（*OSDI 20*）》，2020年。USENIX协会，1–19。[9] Bryan Cantrill。2021年。傲慢与谦逊。检索于2025年4月17日，网址：https://oxide.computer/blog/hubris-and-humility [10] Holly Chiang、Hudson Ayers、Daniel Giffin、Amit Levy 和 Philip Levis。2021年。电源时钟：面向嵌入式系统的动态多时钟管理。载于《*2021*年嵌入式无线系统与网络国际会议论文集》（*EWSN '21*），2021年。Junction Publishing，荷兰代尔夫特，139–150。[11] Thomas Claburn。2023年。微软正忙于用内存安全的 Rust 重写 Windows 核心代码。检索于 2023 年 10 月 7 日，网址：https://www.theregister.com/2023/04/27/microsoft_windows_rust/ [12] Jonathan Corbet。初探 Rust 在 6.1 内核中的应用。检索于2024年4月19日，网址：https://lwn.net/Articles/910762/ [13] Andres Erbsen、Jade Philipoom、Jason Gross、Robert Sloan 和 Adam Chlipala。2019年。简洁的高级密码学算术代码——有证明，无妥协。载于《2019年IEEE安全与隐私研讨会（SP）》style>,

2019 1202 1219 10 1109 2019 00005 。 – 。 . / [14] SP。 . FreeRTOS。FreeRTOS：面向微控制器和小型微处理器的实时操作系统。检索于2025年8月31日，网址：https://www.freertos.org/ [15] 谷歌。OpenSK。检索于2025年8月31日，网址：https://github.com/google/opensk [16] 谷歌。Ti50 版本发布。检索于2025年8月31日，网址：https:// chromium.googlesource.com/chromiumos/platform/ec/+/gsc_utils/ docs/ti50 firmware releases.md_ _

[17] Dan Grossman。2002年。命令式语言中的存在类型。载于第*11*届欧洲编程语言与系统研讨会论文集（*ESOP '02*），2002年。Springer-Verlag，柏林、海德堡，

21 35 10 1007 3 540 45927 8_3 – 。 . / - - - Per Brinch [18] Hansen。多道程序系统的核心。《*ACM*通讯》 13，4（1970年4月），238–241。10.1145/362258.362278[19] Michael Hohmuth、Michael Peter、Hermann Härtig和Jonathan S. Shapiro。2004年。通过使用不可信组件缩减可信计算基规模：小型内核与虚拟机监控器之比较。载于第*11*届*ACM SIGOPS*欧洲研讨会论文集（*EW 11*），2004年。计算机协会，比利时鲁汶，22– es。10.1145/1133572.1133615[20] Galen C. Hunt和James R. Larus。2007年。Singularity：重新思考软件栈。*SIGOPS*《操作系统评论》 41，2（2007年4月），37–49。10.1145/1243418.1243424[21] 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。2009年。seL4：操作系统内核的形式化验证。载于*ACM SIGOPS*第*22*届操作系统原理研讨会论文集（*SOSP '09*），2009年。计算机协会，大天空，蒙大拿州，美国，

207 220 10 1145 1629575 1629596 – 。 . / [22] . Kevin Klues 、 Vlado Handziski 、 Chenyang Lu 、 Adam Wolisz 、 David Culler、David Gay和Philip Levis。2007年。在设备驱动程序中集成并发控制与能量管理。*SIGOPS*《操作系统评论》 41，6（2007年10月），251–264。10.1145/1323293.1294286[23] Philip Levis。2012年。TinyOS开发十年的经验。载于第*10*届*USENIX*操作系统设计与实现研讨会（*OSDI 12*），2012年10月。USENIX协会，加利福尼亚州好莱坞，207–220。[24] Amit Levy、Bradford Campbell、Branden Ghena、Daniel B. Giffin、Pat Pannuto、Prabal Dutta和Philip Levis。2017年。安全且高效地对一台64kB计算机进行多道程序设计。载于第*26*届操作系统原理研讨会论文集（*SOSP '17*），2017年。计算机协会，中国上海，234–251。10.1145/3132747.3132786[25] Jochen Liedtke。1993年。通过内核设计改进进程间通信。载于第十四届*ACM*操作系统原理研讨会论文集（*SOSP '93*），1993年。计算机协会，北卡罗来纳州阿什维尔，美国，175–188。10.1145/168619.168633[26] Shravan Narayan、Craig Disselkoen、Tal Garfinkel、Nathan Froyd、Eric Rahm、Sorin Lerner、Hovav Shacham和Deian Stefan。2020年。在Firefox渲染器中改造细粒度隔离。载于第*29*届*USENIX*安全研讨会（*USENIX Security 20*），2020年8月。USENIX协会，699–716。[27] Vikram Narayanan、Tianjiao Huang、David Detweiler、Dan Appel、Zhaofeng Li、Gerd Zellweger和Anton Burtsev。2020年。RedLeaf：安全操作系统中的隔离与通信。载于第*14*届*USENIX*操作系统设计与实现研讨会（*OSDI 20*），2020年11月。USENIX协会，21–39。[28] NIST国家漏洞数据库。2023年。Rust Wasmtime crate中的CVE-2023-30624：Wasmtime在Rust运行时函数中存在未定义行为。2024年11月1日检索自https://nvd.nist. gov/vuln/detail/CVE-

2023 30624 - Vivien Rindisbacher、Evan [29] Johnson、Nico Lehmann、Tyler Potyondy、Pat Pannuto、Stefan Savage、Deian Stefan和Ranjit Jhala。2025年。TickTock：生产级嵌入式操作系统中的已验证隔离。载于第*31*届*ACM*操作系统原理研讨会论文集（*SOSP '25*），2025年。计算机协会。10.1145/3731569.3764856[30] Rust贡献者。2024年。不安全代码指南参考——术语表。2024年12月2日检索自https://rust-lang.github.io/unsafe-code-guidelines/glossary.html [31] Leon Schuermann、Jack Toubes、Tyler Potyondy、Pat Pannuto、Mae Milano和Amit Levy。2025年。搭建桥梁：通过Omniglot与外部语言进行安全交互。载于第19届

*USENIX*操作系统设计与实现研讨会（*OSDI 25*），2025年。USENIX协会，595–613。[32] Jonathan S. Shapiro、Jonathan M. Smith和David J. Farber。1999年。EROS：一种快速的能力系统。载于第十七届*ACM*操作系统原理研讨会论文集 （*SOSP '99*），1999年。计算机协会，南卡罗来纳州查尔斯顿，美国，170–185。10.1145/319151.319163[33] Jeffrey Vander Stoep。2022年。Android 13中的内存安全语言。2023年10月7日检索自https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html [34] David Terei、Simon Marlow、Simon Peyton Jones和David Maz- ières。2012年。安全Haskell。载于*2012*年*Haskell*研讨会论文集 （*Haskell '12*），2012年。计算机协会，哥本哈根，丹麦， 137 148 10 1145 2364506 2364524 – 。 . / . William Wulf 、 Ellis Cohen 、 William Corwin 、 Anita Jones 、 Roy Levin、Charles Pierson和Fred Pollack。1974年。HYDRA：多处理器操作系统的内核。《*ACM*通讯》 17，6（6月 1974 337 345 10 1145 355616 364017 ），– 。 . / . Zephyr项目。[36] Zephyr操作系统。2025年8月31日检索自https:// www.zephyrproject.org/