# Framekernel: A Safe and Efficient Kernel Architecture via Rust-based Intra-kernel Privilege Separation

Yuke Peng*
Southern University of Science and Technology, China

Hongliang Tian*
Ant Group, China

Jinyi Xian
Southern University of Science and Technology, China

Shuai Zhou
Southern University of Science and Technology, China

Shoumeng Yan
Ant Group, China

Yinqian Zhang ✉†
Southern University of Science and Technology, China

## ABSTRACT

This paper introduces the framekernel architecture, a novel approach to operating system (OS) design that utilizes safe language-based, intra-kernel privilege separation. This architecture combines the security advantages of microkernels with the performance efficiencies of monolithic kernels. A framekernel is composed of a privileged OS Framework and de-privileged OS Services. The Framework encapsulates all low-level, unsafe operations into high-level, safe abstractions, thereby enabling the Services to implement a wide array of functionalities in a safe language. The primary challenge in designing framekernels lies in maintaining a minimal Trusted Computing Base (TCB)–the Framework– while supporting extensive functionalities. This paper outlines the design principle and rules that facilitate this balance. Our Rust-based framekernel prototype, ASTERINAS, validates the framekernel concept by supporting over 130 Linux system calls and a broad range of OS features and device drivers, based on a significantly reduced TCB.

## CCS CONCEPTS

• **Security and privacy** → **Operating systems security**; • **Software and its engineering** → **Operating systems**.

*Both authors contributed equally to this research.
†Corresponding author: yinqianz@acm.org

This work is licensed under a Creative Commons Attribution International 4.0 License.
*APSys '24, September 4–5, 2024, Kyoto, Japan*
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1105-3/24/09
https://doi.org/10.1145/3678015.3680492

## KEYWORDS

Operating Systems, Rust, Memory Safety, Framekernel

**ACM Reference Format:**
Yuke Peng, Hongliang Tian, Jinyi Xian, Shuai Zhou, Shoumeng Yan, and Yinqian Zhang ✉. 2024. Framekernel: A Safe and Efficient Kernel Architecture via Rust-based Intra-kernel Privilege Separation. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '24), September 4–5, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3678015.3680492

## 1 INTRODUCTION

Memory safety is crucial for operating systems because it helps prevent unauthorized or incorrect memory access, thereby reducing the risk of crashes, data corruption, and security breaches. Operating system kernels are typically written in languages like C and C++ that do not inherently provide memory safety, so achieving this often requires formal verification. One prominent example is seL4 [7], the first formally verified microkernel. In seL4-based operating systems, most services run in user space on top of the microkernel, isolating them from the trusted computing base (TCB). While this approach enhances security by reducing the TCB size, it introduces significant communication overhead between the microkernel and the user-space services. Alternatively, it's possible to formally verify the memory safety of a monolithic kernel, as shown by the CertiKOS project [5]. However, formal verification of a monolithic kernel demands extensive manual effort and becomes increasingly challenging as the codebase grows in size and complexity. Consequently, this approach may not be practical for large-scale kernels with substantial legacy code.

More recent work in operating system design focuses on using memory-safe programming languages to address memory safety at a more fundamental level. Examples of these are Verve [12], Biscuit [4], and RedLeaf [10], written in C#, Go, and Rust, respectively. However, even in these safer languages, certain low-level operations that are crucial for kernel-level tasks, such as direct pointer manipulation

| | Microkernels | | Monolithic kernels | | **Framekernel** |
|---|---|---|---|---|---|
| | seL4[7] | RedLeaf[10] | Verve[12] | Biscuit[4] | **Asterinas** |
| Kernel Languages | C | Rust | C# | Go | **Rust** |
| Comm. Overheads | Large | Small | Small | Small | **Small** |
| Drivers in TCB | No | Yes | Yes | Yes | **No** |
| User Languages | Any | Rust | C# | Any | **Any** |

**Table 1: A comparison between prior representative memory-safe OS kernels and Asterinas, our framekernel prototype. Among them, Asterinas is the first one that satisfies all criteria for a practical, general-purpose OS kernel.**

or hardware register interactions, require "escape hatches" where safety guarantees are bypassed. To mitigate the risks associated with these unsafe operations, the design of these kernels encapsulates them within higher-level APIs that are presumed to be safe. These APIs are supported by relatively small, carefully scrutinized cores. In Verve, this core is called the "Nucleus." Biscuit relies on the Go runtime to manage low-level operations, while RedLeaf employs a microkernel architecture. The strength and weakness of these work are summarized in Table 1.

The feasibility of writing a feature-rich, general-purpose OS kernel in a safe language using a small, trusted core remains uncertain. One might expect that the small cores of these kernels would be feature-rich and serve as the sole TCB of kernel memory safety, but this is not always the case. For example, the core of Verve [12] only has built-in support for four I/O devices, meaning that adding more device drivers demands extending the TCB further. In RedLeaf [10], device drivers depend on "trusted libraries", which contain unsafe code outside the core's boundaries. Furthermore, RedLeaf does not support hardware-isolated user processes. Biscuit [4] is similarly limited. Though Biscuit only implements 58 system calls and two relatively complete drivers, it contains 90 uses of Go's unsafe routines inside the core for purposes like writing user memory and accessing device registers. For all three kernels, the TCBs of these three kernels go beyond their cores and tend to grow larger with the addition of OS features and drivers.

To support the development of general-purpose operating systems with a solid yet minimal foundation, we propose a novel OS architecture called framekernel. In this architecture, a kernel is divided into two distinct parts: a privileged part that contains low-level operations without safety guarantees and a de-privileged part that is strictly written in a memory-safe language. The privileged part, known as the OS Framework (or simply "the Framework"), offers secure abstractions to support the de-privileged part, which contains the OS Services (referred to as "the Services"). The Services are responsible for implementing various OS features and device drivers while relying on the Framework's secure underpinnings.

The framekernel architecture combines the security advantages of a microkernel with the performance benefits of a monolithic kernel. Similar to a monolithic kernel, framekernel modules coexist within the same address space, facilitating communication through shared memory and function calls. However, unlike traditional monolithic kernels, the memory safety of the entire kernel relies exclusively on the Framework. This approach yields a compact TCB, akin to that of a microkernel, enhancing overall system security.

The primary challenge in designing a framekernel mirrors that of a microkernel: maintaining a minimal TCB while supporting rich functionalities. However, a significant difference exists—a microkernel achieves isolation through hardware-based methods, whereas a framekernel utilizes a language-based approach for intra-kernel privilege separation. Therefore, we have adapted the minimality principle of microkernels [6] to suit framekernels:

**The minimality principle.** *A component is tolerated inside the privileged OS Framework only if moving it outside the Framework would prevent the de-privileged OS Services from implementing required functionalities safely.*

Following this principle, we have established a set of rules to guide the separation: for each class of OS resources or functions, what should remain within the Framework and what can be safely delegated to the Services:

- **Separate kernel- and user-space CPU states.** The Framework should provide abstractions allowing the Services to inspect and manipulate CPU registers of user-space processes. This ability is essential for the Services to handle system calls or CPU exceptions from the user space freely. In contrast, the Services should not be allowed to arbitrarily manipulate kernel-mode CPU registers, as these registers are crucial for kernel memory safety. Otherwise, the Services could compromise the control flow integrity of the kernel by modifying the program counter or stack pointer.
- **Separate typed and untyped memory.** Accessing main memory with raw pointers is unsafe by nature but is a common demand throughout a kernel. For example, to read inputs from or write outputs to the user space, the kernel has to access the memory specified by user pointers. As another example, device drivers often need to exchange data with their devices by accessing DMAable data structures or buffers directly. To accommodate such a common usage, we propose to classify memory into two classes: typed vs untyped. Typed memory is relevant to kernel memory safety. One obvious example is the code, stack, and heap of a safe language kernel as they store type-safe objects; Overriding such memory corrupts kernel memory safety. Untyped memory, on the other hand, is irrelevant to kernel memory safety. For example, the memory pages are mapped to a user space or those prepared for DMA to/from

devices can be treated as untyped as the kernel does not trust such memory nor should it store type-safe objects on them. Thus, the Framework should provide abstractions to allow management and manipulation of untyped memory.

- **Separate task switching and scheduling.** Multitasking is a crucial feature of most modern kernels, which must be supported by the Framework. Therefore, the process of context switching, which includes the error-prone and unsafe logic of saving and restoring CPU states, must be performed within the Framework. However, scheduling strategies (e.g., Linux's CFS) are complex to implement and can be decoupled from context switching. Thus, scheduling strategies should be moved to the Service and pluggable into the Framework.
- **Separate event dispatching and handling.** A kernel is responsible for handling system events such as CPU exceptions and external interrupts. The registration of handler routines for these events involves low-level, unsafe operations and hence must be managed by the Framework. However, handling CPU exceptions originating from user space or external interrupts from devices does not necessitate unsafe operations. As such, this responsibility can be appropriately delegated to the Services.
- **Separate system and peripheral devices.** We classify devices into two categories: system devices and peripheral devices. System devices (or controllers) manage fundamental aspects of the system. A logic bug, rather than a safety one, in a system device driver can breach fundamental assumptions of the kernel, potentially compromising kernel memory safety. Examples of system devices on the x86 architecture include APIC and IOMMU. In contrast, peripheral devices like storage, network, and graphic devices do not impact the system as extensively. Given these distinctions, the Framework should limit its abstractions to only expose the I/O ports or memory spaces of peripheral devices. Moreover, to mitigate potential bugs in the safe drivers of peripheral devices that might corrupt typed memory via DMA, the Framework must enable IOMMU. These strategies collectively foster safe driver development while ensuring kernel memory safety.

Guided by the minimality principle and the five rules, we develop a framekernel prototype named Asterinas for the x86-64 architecture, using Rust for its implementation. This prototype has been released as open-source [2]. As shown in Figure 1, Asterinas comprises two components: the privileged Asterinas Framework and the de-privileged Asterinas Services. The Asterinas Framework offers a safe, expressive, and minimal foundation for crafting secure and efficient kernel code. Asterinas Framework encapsulates all low-level, unsafe operations that manage and interact with the hardware and user space into high-level, safe abstractions, providing the ability to manage user-space CPU states and untyped memory, customize task scheduling policies and event handling logic, and access the I/O ports and memory of peripheral devices. With these abstractions, Asterinas Services can implement a rich set of OS features in safe Rust. The design and security invariants of the Asterinas Framework are further discussed in Section 4.

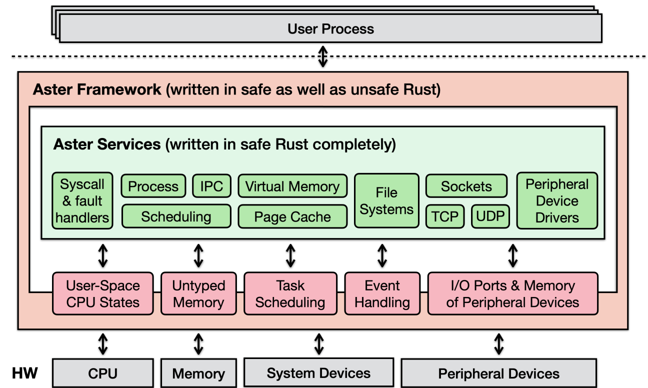


**Figure 1: Architecture of Asterinas, a prototype of framekernel.**

Building upon this foundation, Asterinas Services implement a substantial subset of Linux features, including preemptive scheduling, file systems (e.g., Ext2), sockets (e.g., TCP/UDP over IPv4), and device drivers (e.g., VirtIO-blk and Virtio-net). These features collectively support over 130 Linux system calls. The entire Asterinas codebase comprises over 50,000 lines of Rust code, with the Framework accounting for only about 20%—a significant reduction of TCB compared to other Rust-based kernels like RedLeaf, which has a TCB of 60%. See Section 5 for more detail.

As the development of additional OS subsystems and device drivers progresses, we anticipate that the size ratio of the Framework to the entire codebase will decrease even further, This will allow us to add more functionalities while keeping a small TCB for memory safety. As of now, Asterinas does not perform as well as Linux due to its immaturity. We are working hard on incorporating performance optimizations. Our experience so far has found no inherent limitations in the framekernel architecture that prevents Asterinas from achieving the same level of performance as Linux.

## 2 RUST FOR KERNEL DEVELOPMENT

Rust is an efficient system programming language featuring strong memory safety and thread safety guarantees, making it a preferable option for operating system development [1, 3, 8, 10, 11]. Rust has also made its way into Linux [9].

Rust establishes its safety guarantees through its ownership model, which requires the compiler to manage all

references to objects to prevent concurrent mutations and memory-related errors. However, this ownership model inherently restricts the programming language's expressive power, posing challenges for operating system development that often involves low-level and concurrent system accesses. Therefore, Rust offers the "unsafe" keyword as a way to bypass the language's strict type system and its ownership model when needed.

However, bypassing the type system means that OS developers must take responsibility for ensuring type safety, thereby maintaining the assumptions on which the Rust compiler relies. Failure to do so would invalidate the safety guarantees provided by the ownership model. In these cases, the compiler often cannot verify whether the unsafe code adheres to its expected constraints.

The Rust documentation offers a non-exhaustive list of practices to avoid undefined behaviors in typical user-space applications. However, when working in the kernel space, using the unsafe mechanism requires additional care and consideration due to the increased complexity and potential risks. For example, the Control flow integrity requires the Services logic, trap handlers, user programs, and so on should run in their program orders.

# 3 THE FRAMEKERNEL ARCHITECTURE

In this paper, we answer the question of whether it is feasible to develop a feature-rich, general-purpose OS kernel using a safe programming language, supported by a minimal trusted "core". Our solution is the framekernel architecture, which divides the kernel into a privileged OS Framework and de-privileged OS Services. The Framework consolidates all low-level, unsafe operations into high-level, safe abstractions, enabling the Services to safely implement a broad array of OS features and device drivers.

This section outlines the trust model of a framekernel, details its design goals, and discuss the concepts of typed and untyped memory, which have been mentioned in Section 1.

## 3.1 Trust Model

We describe which hardware and software components a framekernel trusts and which it does not. The memory safety of a framkernel depends solely on these trusted components.

The hardware TCB includes the CPU and system devices (or controllers). On an x86 machine, these system devices include Memory Management Unit (MMU), Input-Output Memory Management Unit (IOMMU) and Advanced Programmable Interrupt Controller (APIC). A framekernel does not trust peripheral devices, such as block devices or network interface cards connected through the PCI/PCI-E bus.

The software TCB consists of the toolchain for the safe language, the privileged OS Framework of the framekernel, and the bootloader and other firmware that runs before the framekernel. The de-privileged OS Services or user-space processes running atop the framekernel are not trusted.

## 3.2 Design Goals

The framekernel architecture demands the OS Framework to meet the following four design goals simultaneously.

- **Soundness.** The Framework is considered sound if no undefined behaviors may be triggered via its API by any safe code, provided that the safe code is verified by the toolchain.
- **Expressiveness.** The Framework should enable developers to implement a wide range of OS functionalities in the safe language using its APIs. It is especially important for the Framework to enable safe driver development, considering that device drivers often constitute a significant portion of the codebase in a fully-fledged OS kernel, such as Linux.
- **Minimalism.** As a principal component of the TCB, the Framework should be as small as possible. It should not incorporate any functionality that can be safely and efficiently implemented outside of the Framework.
- **Efficiency.** The APIs provided by the Framework should incur minimal overheads. Ideally, these APIs should be realized as zero-cost abstractions.

It is particularly challenging to achieve both soundness and expressiveness at the same time. Expressiveness requires delegating more OS resources or controls to the Services, but this delegation increases the risk of memory safety issues, hindering the soundness guarantees. To strike the right balance, we have articulated five rules in Section 1.

## 3.3 Typed vs Untyped Memory

Typed memory refers to the physical memory regions that can potentially compromise the system's memory safety. These regions include the system code or data segments, the Rust heap, and the regions used by system devices. To ensure system soundness, the Framework implements security measures to protect these regions including preventing untrusted entities from direct access to typed memory and enforcing the restrictions imposed by the Framework's APIs.

Typed memory can be converted to untyped memory and vice versa. During system runtime, the size of typed memory may dynamically expand or shrink. For example, when the MMU establishes a new mapping, it allocates a physical memory region from the pool of untyped memory and reclaims the memory when it removes a mapping. To ensure memory protection during these conversions, the Framework incorporates appropriate compile-time and runtime measures to safeguard the integrity and security of the system.

# 4 ASTERINAS FRAMEWORK

Figure 1 presents the architecture of framekernel, which includes the Framework and the Services. The Framework provides the Services with five functionalities 1) Untyped memory access, 2) User-space CPU states modification, 3) Task Scheduling, 4) Event handling, and 5) Peripheral devices I/O access. In addition to these components, the Framework provides utilities that require `unsafe` code (Section 4.6).

## 4.1 Untyped Memory Access

The Framework employs `Frame` to help the Services access untyped memory. `Frame` is a structure defined in the Framework and each instance of `Frame` is assigned an index corresponding to a physical page.

A `Frame` allocator manages the untyped memory regions. It establishes all untyped memory regions during system initialization. The Framework may request memory page allocation or deallocation at runtime, leading to conversions between typed and untyped memory. The conversion is achieved using Rust's drop mechanism, which allows the reclamation of the typed memory and its return to untyped memory. The Framework maintains **Inv1** and **Inv2** for memory management to ensure memory safety.

**Inv1:** IOMMU maps to untyped memory pages.

**Inv2:** For the user space, MMU creates mappings only to untyped memory.

The Framework lets the Services pass in a `Frame` instance as a parameter when establishing the MMU or IOMMU mapping. A `Frame` object owned by the Services always points to a valid untyped memory page. Thus, the Framework allows the Services to map virtual addresses to a physical memory page that the `Frame` points to. The Framework takes ownership of the `Frame` to keep it from reclamation.

## 4.2 User-Space CPU States Modifications

The Framework uses `UserMode` structure to protect different CPU states. `UserMode` allows the Services to enter the user mode with the specified CPU states. However, the Framework uses typed memory to keep the Services from accessing the kernel CPU states, which ensures the control flow integrity of user programs and exception handlers while keeping the efficient function calls and system calls interfaces. **Inv3** summarizes the measures taken by the Framework.

**Inv3:** Kernel CPU states must reside in the typed memory.

## 4.3 Task Scheduling

Tasks abstract CPUs' execution flows, supporting context switches and scheduling. Tasks follow **Inv3** to ensure context switches will not affect control flow integrity. Besides, the Framework does not implement a scheduling algorithm. Instead, it provides a Rust trait for the Services so that the Services can register a scheduler of their choice. Moreover, the Framework sets up a guard page to defend against stack overflows which brings to the corruption of typed memory and unintentional changes to irrelevant execution contexts. **Inv4** gives the property for the integrity of stacks.

**Inv4:** Kernel's execution contexts must be immune to corruptions, adding runtime checks if necessary.

## 4.4 Event Handling

System events can be divided into two categories: external interrupts and CPU exceptions. Design considerations for addressing these events differ.

External interrupts are used to address asynchronous events that interact with external devices or systems. The Framework provides `IrqLine` structure to do events callback. Each `IrqLine` instance is associated with a specific interrupt number. When an external interrupt occurs, the Framework identifies the corresponding `IrqLine` instance based on the interrupt number and invokes the registered callback function.

However, some CPU architectures do not distinguish between CPU exceptions and external interrupts. For instance, in x86-64, PCIe devices can signal some of the CPU exceptions through Message Signaled Interrupt by modifying the IRQ number. To prevent the CPU exception handler from performing illegal operations in this situation, the Framework introduces **Inv5** to fortify the CPU exception handler.

**Inv5:** CPU exception handler must check whether the exception happened before performing operations.

## 4.5 Peripheral Devices I/O Access

The I/O of devices is categorized into two forms: Port I/O (PIO) and memory-mapped I/O (MMIO). In the Framework, these are represented as `IoPort` and `IoMem`, respectively. Both the `IoPort` and `IoMem` instances are associated with a specific range of PIO or MMIO.

To manage PIO access, the Framework employs the `IoPort` distributor, which ensures that peripheral drivers cannot access the I/O ports used by system device drivers. The distributor accomplishes this through a two-step process. Firstly, during initialization, it creates a CPU architecture-specific port range. Secondly, the system device drivers within the Framework can use an internal API provided by the `IoPort` distributor to exclude the I/O ports used by system devices.

Similarly, the Framework uses the `IoMem` distributor to manage MMIO access. However, the `IoMem` distributor excludes the physical memory regions based on the memory distribution. This ensures that the allocated IoMem does not cross boundaries and access physical memory regions

| | rCore | RedLeaf | Tock | **Asterinas** |
|---|---|---|---|---|
| Peripheral drivers | ● | ◐ | ◐ | ○ |
| File systems | ● | ○ | ○ | ○ |
| Network stacks | ● | ○ | ○ | ○ |
| Schedulers | ● | ● | ● | ○ |
| IPC and Signals | ● | ● | ● | ○ |
| TCB | 100% | 60.34% | 72.68% | 20.89% |

**Table 2: A comparison between the TCB size of different Rust kernels. ●: all crates within the subsystem are included in the TCB; ◐ only some crates are included in the TCB; ○ no crate is included.**

beyond its intended scope, thereby safeguarding against out-of-bounds memory accesses. Summarizing the above steps, the Framework introduces **Inv6** and **Inv7**.

**Inv6:** System device driver must remove its I/O access in the distributor when system is initializing.

**Inv7:** MMIO distributor must exclude physical memory.

### 4.6 Concurrent Utilities

The Framework provides concurrent utility types whose implementation must rely on unsafe Rust, e.g., concurrent locks and concurrent data collections. There are two types of `unsafe` used by utilities. The first type of `unsafe` is used to implement `Send` and `Sync` functionalities such as concurrent locks. These locks are a crucial tool for Rust operating systems. The second type of `unsafe` is `UnsafeCell`, which is used to break Rust's reference restrictions and implement high-performance data structures or concurrent locks. Due to Rust's reference restrictions and performance limitations, some data structures, such as linked lists, can be challenging to implement using only safe Rust. Therefore, the Framework allows such data structures to exist, but they require extensive testing to ensure accurate implementation.

## 5 ASTERINAS SERVICES

On top of Asterinas Framework, we develop Asterinas Services, which implement a substantial subset of Linux features, such as virtual memory, processes, IPC, VFS, and sockets, providing over 130 system calls. It supports multiple file systems (e.g., Ext2, exFAT32, procfs, devfs, and ramfs), socket types (e.g., TCP, UDP, and Unix domain), and device drivers (e.g., VirtIO-console, VirtIO-blk, and VirtIO-net). All these functionalities of Asterinas Services are written in safe Rust. Asterinas Services demonstrates that Asterinas Framework is expressive enough to implement general-purpose, feature-rich OS kernels.

Overall, Asterinas consists of over 50,000 lines of Rust code, with Asterinas Framework comprising approximately 20% of all code, representing the size of the TCB. The remainder is attributed to Asterinas Services. We compare the TCB size ratio of Asterinas with other existing Rust-based kernels in terms of the entire codebase size. The comparative results are detailed in Table 2. Notably, For our TCB analysis, we evaluate each Rust crate in its entirety, classifying it as TCB or non-TCB based on whether unsafe Rust is allowed or not. Thus, the TCB of a kernel encompasses all crates identified as TCB within that kernel.

## 6 RELATED WORK

RedLeaf [10] is a microkernel OS that relies on Rust's type safety and memory safety for isolation, rather than hardware isolation mechanisms. It does not allow the use of `unsafe` keyword in user domains and only permits the kernel and trusted libraries to use `unsafe`. Asterinas is similar to RedLeaf but with two significant differences. Firstly, Asterinas still uses hardware isolation to run programs developed in multiple languages, as it aims to be general-purpose and provides Linux-compatible ABI. Secondly, Asterinas provides a device driver framework, which eliminates the need for writing trusted libraries for peripheral drivers.

Tock [8] is an embedded operating system that employs Rust safety features to support SoCs with limited hardware protection measures. Tock divides the kernel into trusted core libraries that can use `unsafe` code and untrusted capsules that are not allowed to use `unsafe` code. The design of Asterinas and Tock is different due to the inherent differences between embedded and general-purpose systems. Compared to embedded systems, general-purpose systems must support complex applications and drivers, which makes our task more challenging. Moreover, Tock's design causes the TCB to expand as the number of device drivers increases, leading to a significant increase in the difficulty of TCB audit.

## 7 CONCLUSION

This paper introduces the framekernel architecture which merges the security of microkernels with the performance of monolithic kernels by employing safe language-based intra-kernel privilege separation. The prototype of the framekernel architecture, Asterinas, effectively demonstrates the effectiveness of this approach by implementing over 130 Linux system calls and supporting various OS features with a TCB ratio of about 20%.

## ACKNOWLEDGMENTS

Yinqian Zhang is in part supported by National Key R&D Program of China (No. 2023YFB4503902), CCF-ANT Funds (CCF-AFSG RF20220012 and CCF-AFSG RF20230211), as well as several research grants from Ant Group. The authors from Ant Group are supported by the Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (Grant No. TD2019001).

## REFERENCES


[1] Andy-Python-Programmer. 2024. Aero OS. https://github.com/Andy-Python-Programmer/aero.

[2] Asterinas. 2024. The code repository of Asterinas. https://github.com/asterinas/asterinas.

[3] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: an Experiment in Operating System Structure and State Management. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, 1–19. https://www.usenix.org/conference/osdi20/presentation/boos

[4] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. 2018. The benefits and costs of writing a POSIX kernel in a high-level language. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). USENIX Association, Carlsbad, CA, 89–105. https://www.usenix.org/conference/osdi18/presentation/cutler

[5] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). USENIX Association, Savannah, GA, 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[6] LIEDTKE Jochen. 1994. On $\mu$-Kernel Construction. Proc 15th SOSP, 1994 (1994).

[7] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. [n. d.]. seL4: Formal Verification of an OS Kernel. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (New York, NY, USA, 2009-10-11) (SOSP '09). Association for Computing Machinery, 207–220. https://doi.org/10.1145/1629575.1629596

[8] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In Proceedings of the 26th Symposium on Operating Systems Principles (Shanghai, China) (SOSP '17). Association for Computing Machinery, New York, NY, USA, 234–251. https://doi.org/10.1145/3132747.3132786

[9] Linux. 2024. Rust — The Linux Kernel Documentation. https://docs.kernel.org/rust/index.html.

[10] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, 21–39. https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram

[11] rcore os. 2023. rCore. https://github.com/rcore-os/rCore.

[12] Jean Yang and Chris Hawblitzel. 2010. Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System. In PLDI. Association for Computing Machinery, Inc. https://www.microsoft.com/en-us/research/publication/safe-to-the-last-instruction-automated-verification-of-a-type-safe-operating-system/