USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

EKC: A Portable and Extensible Kernel Compartment for De-Privileging Commodity OS

Jiaqin Yan, *Shanghai Jiao Tong University, Southern University of Science and Technology;* Qiujiang Chen, Shuai Zhou, and Yuke Peng, *Southern University of Science and Technology;* Guoxing Chen, *Shanghai Jiao Tong University;* Yinqian Zhang, *Southern University of Science and Technology*

https://www.usenix.org/conference/usenixsecurity25/presentation/yan-jiaqin

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

EKC: A Portable and Extensible Kernel Compartment for De-Privileging Commodity OS

Jiaqin Yan[*†], Qiujiang Chen[†], Shuai Zhou[†], Yuke Peng[†], Guoxing Chen[*✉], Yinqian Zhang[†✉]
[*]Shanghai Jiao Tong University, [†]Southern University of Science and Technology
yan2364728692@gmail.com, {12012211, zhous2021, pengyk}@mail.sustech.edu.cn,
guoxingchen@sjtu.edu.cn, yinqianz@acm.org

Abstract

Kernel compartmentalization through privilege separation is an effective solution for reducing the trusted computing base of modern operating systems (OS) with monolithic kernels. However, existing approaches to kernel compartmentalization often depend on higher-privileged software or platform-specific hardware features, posing challenges to their portable deployment and practical application.

In this paper, we propose Embedded Kernel Compartment (EKC), a kernel compartment that embeds itself to a commodity OS as a privileged, isolated compartment. EKC is both portable across multiple ISAs without hardware modification and extensible to multiple OSes, even those developed in different languages. Moreover, EKC can serve both kernel components and user-space applications, enabling security critical tasks and providing sensitive data storage.

We implemented a prototype of EKC in Rust, which has been successfully ported to run on multiple ISAs (RISC-V and ARM) and extended to be compatible with various OS kernels (FreeRTOS, rCore, and TinyLinux) with additional security services. Through comprehensive analysis and evaluation, the results demonstrate that EKC is a practical and effective solution for kernel compartmentalization.

1 Introduction

The design of commodity operating systems (OS), such as Linux [3], FreeBSD [26], and FreeRTOS [17], often adopts a monolithic kernel architecture, where all kernel components reside in one single address space. However, as the monolithic kernel grows in size and complexity, the absence of robust isolation mechanisms significantly expands its attack surface. Numerous CVEs [24] associated with the commodity monolithic kernels allow one malicious kernel module to unlawfully acquire kernel privileges, endangering the entire kernel memory space and ultimately threatening the confidentiality of sensitive data within the kernel, as shown in Fig. 1.a. For example, DirtyPT [52] exploits a use-after-free vulnerability [23] in the page table management module of Linux that allows malicious code to arbitrarily modify the memory map to access the entire kernel memory.

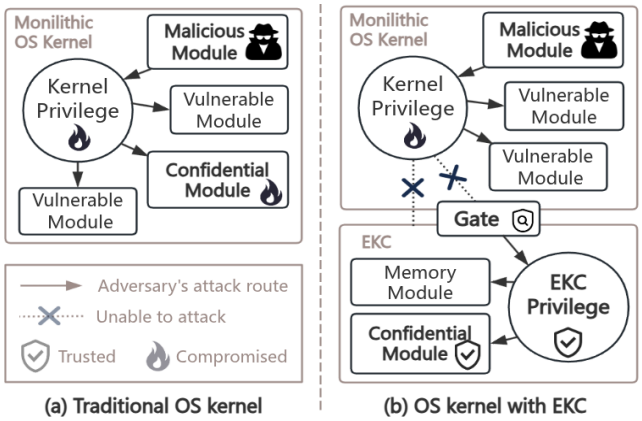


Figure 1: (a) Issues with traditional monolithic kernels and (b) Solution of EKC

The Principle of Least Privilege (PoLP) [40] serves as a critical defense mechanism. It mandates that each module operate with the minimal privileges necessary to perform its intended functions. Kernel compartmentalization is a key strategy for implementing PoLP [31]. This approach involves decomposing the kernel into multiple isolated compartments, each granted only the privileges required to fulfill its specific functionality. In such a design, confidential data can be stored within a dedicated, isolated compartment, thereby restricting the access privileges of potentially malicious kernel modules to sensitive information. This effectively delineates boundaries between high-risk code and confidential data within the OS kernel, thereby mitigating the potential impact of vulnerabilities in high-risk modules.

[1]The SUSTech authors are affiliated with the Research Institute of Trustworthy Autonomous Systems.

Kernel compartmentalization has been studied in various works. HAKC [34] isolates the kernel code and data, relying on hardware features (Arm MTE [4] and PAC [5]). Therefore, they can hardly be ported to run on other ISAs, such as RISC-V and x86. In contrast, software-based approaches, such as LVDs [36], KernelGuard [53], and SILVER [41], rely upon privileged system software (e.g. VMM [43]), not only permitting system software's complete access to sensitive data [54], but limiting their applications to only VM-based settings. Other software-based solutions, such as Nested Kernel [25] and SKEE [15], eliminate the dependency on underlying system software. However, they have largely overlooked the challenge of restricting access to control registers by de-privileged kernel compartments [13].

Due to the limitations of existing approaches, we propose **Embedded Kernel Compartment (EKC)**, a kernel compartment that embeds itself to a commodity OS as a privileged, isolated compartment with separate code and data regions. EKC is (1) secure against adversaries with OS kernel privileges; (2) portable across multiple ISAs without requiring any hardware modification; (3) extensible to multiple OSes, even those not developed in the same programming languages. EKC achieves these properties by successfully de-privileging the commodity OS kernels with minimal kernel modification to ensure a set of security invariants. EKC provides a set of well-defined interfaces, via a restricted gate, for both OS kernel components and user-space applications, which could interact with EKC to perform security critical tasks or store sensitive data. As such, EKC establishes a foundation for a variety of secure applications, including tamper-proof logs and cryptographic services, enhancing kernel security and privacy in diverse computing scenarios.

To validate the concept of EKC, we developed a prototype in the Rust programming language. Our prototype features a small trusted computing base (TCB) of less than 10,000 lines of Rust code, the majority of which is written in safe Rust and verified by the Rust compiler. EKC has been successfully ported to multiple RISC ISAs, including RISC-V64, Arm32, and Arm64. We have successfully deployed EKC on QEMU [6], Raspberry Pi 4b board [38], and Allwinner D1-H board [11]. The system has been extended to support multiple OS kernels, including Unix-like OSes (TinyLinux [47], rcc [8], rCore [7]) and real-time OSes (FreeRTOS [17]). These OSes are implemented in different programming languages, including C and Rust.

To showcase its practical applications, we have developed several security services on EKC, such as a tamper-proof log [12] for measured boot and a cryptographic service [20] for WolfSSH [51]. The performance of EKC was evaluated using UnixBench [10] and RTOSBench [9]. The results indicate that although EKC incurs some overhead in interrupt handling, the overall impact on performance in practical scenarios remains small.

In summary, this paper makes the following contributions:

- It proposes a novel kernel compartmentalization solution for securely embedding a portable and extensible kernel compartment into commodity OS kernels.
- It presents the design and implementation of EKC, a Rust-based kernel compartment that is compatible with multiple OS kernels and portable to multiple RISC ISAs.
- It provides extensive evaluations of the functionality, performance, and security of EKC to demonstrate the feasibility of the solution.

2 Background and Concepts

Kernel Compartmentalization. Kernel compartmentalization is a specific form of software compartmentalization that segments the kernel into distinct parts, each with assigned privileges. This approach enhances system security by isolating kernel components and strictly regulating their interactions.

As described in [31], kernel compartmentalization can be abstracted into five fundamental actions: *creating/destroying compartments, calling/returning from compartments*, and *assigning privileges to compartments*. For each compartment, careful consideration must be given to the privileges associated with its actions, as well as whether it can be accessed or interrupted by other compartments.

In kernel compartmentalization abstraction, the *security* of a compartment is defined by integrity, confidentiality, and availability. Specifically, a *secure* compartment must meet the following criteria:

- The kernel partitioning mechanism must be *secure*, as compartment security becomes irrelevant without it.
- It cannot be directly read or modified by untrusted compartments, ensuring confidentiality and integrity.
- Its control flow cannot be interrupted by untrusted compartments, guaranteeing availability.

Additionally, for the first criterion, the *security* of a kernel compartmentalization mechanism relies on a critical prerequisite: Only trusted programs, such as the hypervisor or VMM, are permitted to *create/destroy compartments and assign privileges*. This ensures that control over compartmentalization is maintained by highly secure entities.

RISC-V. RISC-V [50] is an open standard ISA based on RISC principles [46]. Distinguished from other ISAs, RISC-V operates with an open-source license, granting free access to anyone. Based on current trends, RISC-V will become the third mainstream architecture after x86 and Arm [29].

Hardware-Based Privilege Level. Currently, most mainstream hardware architectures have at least three hardware-based privilege levels for the supervisor mode, the OS, and user processes. The privilege levels are raised by traps and correspondingly lowered by trap returns. Their designations vary across different mainstream architectures [1, 45, 50]. In

general, they are:

- *User Level*: called Ring 3 in x86, EL0 in Arm/Arm64 and U-mode in RISC-V.
- *OS Level*: called Ring 0 in x86, EL1 in Arm/Arm64 and S-mode in RISC-V.
- *Supervisor Level*: does not exist (or called SMM [1] in Intel) in x86, called EL3 in Arm/Arm64 and M-mode in RISC-V.

The terms *User Level*, *OS Level*, *Supervisor Level* will be used to refer to these three types of hardware privilege levels in the following description, without loss of generality across different hardware architectures.

Regardless of architecture, commodity OS kernels typically operate at a uniform *OS Level*, which may contain potentially malicious modules. In addition, since the basic x86 architecture lacks a dedicated *Supervisor Level*, EKC is designed to operate without necessarily relying on it.

Memory and Interrupt Management. Most mainstream hardware architectures handle memory and interrupt using the same set of control and status registers as follows:

- Root Page Table Register (`RPTR`). This register serves the dual purposes of enabling address translation and storing the first-level page table base address.
- Interrupt Vector Table Register (`IVTR`). This register stores the address of the S-mode interrupt handler.
- Interrupt State Register (`ISR`). This register controls the enabling of different types of interrupts.

These registers have distinct names in different architectures [1, 45, 50], as summarized in Table 1.

Table 1: Register names related to memory and interrupts in different hardware architectures

Register	x86	Arm/Arm64	RISC-V
`RPTR`	CR3	TTBRx	satp
`IVTR`	IDTR	VBAR	stvec
`ISR`	IF	CPSR/DAIF	sie

Without loss of generality across hardware architectures, the acronyms `RPTR`, `IVTR`, `ISR` will be used in the following description.

Rust. Rust [33] is a programming language with a rich type system and ownership model, which guarantees most of the safety features such as memory safety and thread safety. As a result, Rust avoids most potential program vulnerabilities and is used to build reliable and efficient software. Nevertheless, in specific cases of functional requirements, *unsafe Rust* is employed to circumvent the compiler's constraints. Consequently, additional technologies such as code analysis are indispensable for ensuring the security of the code in these cases. There has been a lot of work devoted to analyzing *unsafe Rust* [16, 32].

3 Overview

3.1 Motivation

In a commodity kernel, all modules share the same privilege level, allowing malware to access the entire kernel memory space if one component is compromised. Kernel compartmentalization is an effective solution to this problem [31]. It safeguards sensitive data from kernel vulnerabilities and malicious code by elevating a specific kernel compartment to a higher privilege level, granting it access to sensitive data. Meanwhile, other compartments, which contain potentially untrustworthy code, are assigned a lower privilege level and are inaccessible to sensitive data.

Existing work on kernel compartmentalization has many limitations. First, many existing solutions rely heavily on privileged software support [19, 36, 39, 41, 53] or hardware features [15, 30, 34], leading to deployment challenges. Cross-platform operating systems, including Unix-like systems [3, 8, 26] and real-time systems [17, 55], frequently encounter pervasive memory security issues. Implementing a distinct kernel compartmentalization scheme for each platform in such systems incurs significant costs while providing limited benefits. Furthermore, users encounter difficulties in selecting a compartmentalization scheme that aligns with the requirements of their specific platforms and production environments, rendering platform-specific solutions impractical for widespread adoption.

Second, many solutions only reduce the OS's attack surface without taking user processes into account [25, 28, 34, 36]. However, in many confidential computing scenarios, user processes have confidential computing requirements that must be enforced against an untrusted kernel. [18, 48]. The user usually needs a higher level of security module to enforce data privacy.

Finally, most existing kernel compartmentalization solutions are theoretical architectures centered around the idea of decoupling an OS kernel [14, 25, 35] or sandboxing kernel modules [28, 36]. However, based on our analysis of most commodity OS implementations, we propose that it is feasible to design an *embedded* compartment that can be reused across various OS kernels. This compartment would manage all other compartments, reducing the deployment costs of kernel-level compartmentalization mechanisms while delivering greater overall benefits.

Therefore, EKC is motivated by the limitation of existing kernel compartmentalization schemes and aims to achieve the following design goals:

- **Security**. EKC should be a secure compartment that runs alongside the OS kernel at the OS level. Only trusted parts could create compartments and assign privileges to compartments. EKC's confidentiality, integrity and availability should be enforced against untrusted parts [31].
- **Compatibility**. EKC is designed with minimal reliance

Table 2: Privilege model

	EKC	OS kernel	User Process
User space	R/W	R/W[2]	R/W/X
OS space	R/W	R/W/X	
Trampoline	R/W/X	X	X
EKC space	R/W/X		

[1] R=Readable, W=Writable, X=Executable
[2] EKC can assign OS-inaccessible pages to users.

on hardware features or specific software configurations, which allows for easy portability across different platforms and ISAs. EKC can be embedded to work with various OS kernels and integrate distinct security services as needed.

- **Efficiency.** EKC should deliver low overheads while maintaining strong security guarantees, ensuring that enhanced protection does not compromise performance.

3.2 Threat Model and Privilege Model

In our threat model, EKC is assumed to be part of the trusted computing base (TCB), which is trusted to be free of exploitable vulnerabilities. This relies both on the safety guarantees provided by the Rust programming language and any security audits performed on the assembly code and *unsafe* Rust within EKC. Both the OS kernel and the user process are untrusted. The OS kernel could be exploited by loading malicious kernel modules or hijacking its control flows.

The adversary's goal is to compromise the confidentiality and integrity of EKC by exploiting a compromised compartment (e.g., the OS kernel or the user process). The adversary may exploit any design flaws in EKC to breach the isolation boundary and access the memory space of EKC. However, physical attacks and side-channel attacks are out of scope.

Our privilege model enforces the principle of least privilege [44] by ensuring that each compartment operates with only the minimal privileges necessary for its functionality. As shown in Table 2, EKC has the full privilege to access and manage all compartments. The OS kernel and the user process are restricted to their own space. If desired, EKC can be configured to assign pages that are inaccessible to the OS kernel to the user process. Both EKC and the OS kernel can access the Trampoline; however, only the EKC is authorized to modify it. All memory accesses and instructions must adhere to the appropriate privilege level.

3.3 Challenges

The challenges and insights encountered during the development of EKC are summarized as follows.

C1: De-Privileging the OS kernel. EKC and the OS kernel both operate at the *OS Level*. To ensure security, EKC must effectively de-privilege the OS kernel, preventing it from breaching the boundary between itself and EKC.

Solutions: EKC de-privileges the OS kernel by restricting its direct access to page tables, trap handlers, and physical memory. (see Sec. 4)

C2: Serving both the OS kernel and the user process. Both of them need services from the secure compartments to enforce their data security, including user process.

Solutions: We define a set of standardized interfaces between EKC and the OS kernel, referred to as the EKC API, which requires only minimal modification to the OS kernel to ensure compatibility. To align with existing Application Binary Interfaces (ABIs), EKC provides services to user process via an interrupt-based Service Call (see Sec. 5.3).

C3: Securing EKC. Since EKC is part of the TCB, it is essential to ensure that its implementation is free of exploitable vulnerabilities.

Solutions: We implement EKC using the Rust programming language, leveraging its memory safety and thread safety guarantees to minimize attack surfaces. Additionally, EKC features a small TCB size, which facilitates easier security validation. We have conducted a security analysis of EKC's unsafe Rust and assembly code (see Sec. 7.1).

C4: Modularizing EKC for compatibility with various OSes and ISAs. EKC is designed as a cross-platform architecture capable of supporting a variety of OS kernels. Moreover, embedding EKC should minimize its impact on the underlying OS.

Solutions: EKC leverages only common hardware features and provides standardized interfaces to OS kernels. EKC and OS kernels are decoupled; they do not share a heap or stack and do not need to be compiled or linked together (see Sec. 5).

3.4 Applications

EKC is expected to provide secure applications for both the OS kernel and the user process. Some applications are listed below.

Tamper-proof logs. Recording detailed logs of security-related system events is crucial for system auditing [12]. In EKC, the Entry/Exit Gate plays an essential role in control flow transfer. When the OS kernel requests EKC to modify crucial parameters, the Entry/Exit Gate records the operation in the EKC space. This log can be queried by the user process to promptly detect malicious behavior in the system.

OS kernel measurement. EKC calculates the SHA-256 of the OS kernel at boot time and stores it in EKC. The OS kernel cannot modify this hash value while the user process can request it from EKC. This application ensures that the OS kernel image remains unaltered during system boot.

Crypto services. EKC serves as an ideal component that provides cryptographic operations for the user process, such

as encryption and hashing, in accordance with cryptographic service provider standards like PKCS#11 [20]. Cryptographic software can leverage this service to protect private keys in EKC and perform encryption and signing, as used in protocols such as SSL and TLS.

Kernel compartmentalization. The OS kernel is de-privileged by restricting its ability to create/destroy the compartments and assign privileges. Further separation with the least privilege can be achieved by managing compartment creation and destruction at runtime, as discussed in Sec. 5.1.

4 Security Invariants for EKC

To ensure the security of the compartmentalization mechanism and establish EKC as a secure compartment, three invariants must be enforced.

Invariant 1: EKC manages address translations.

This invariant de-privileges the OS kernel in **address space management**, ensuring that EKC is the sole compartment with permissions to create, destroy, and assign privileges to other compartments. Inv. 1-1 to Inv. 1-3 enforce Invariant 1 by leveraging address translation to confine a dedicated memory region for page table management. This ensures that no other compartments can access them.

Inv. 1-1: Only EKC can configure the Root Page Table Register (`RPTR`). Under the premise that only virtual addresses can be accessed, this sub-invariant restricts the OS kernel from using page tables not explicitly provided by EKC.

Inv. 1-2: Page tables managed by EKC are inaccessible to the OS kernel. EKC manages page tables in a dedicated memory region that is inaccessible to the OS kernel via virtual address or physical address. This ensures that the OS kernel cannot modify the address translations maintained in these page tables.

Inv. 1-3: The OS kernel can modify address translation through well-defined APIs provided by EKC. To effectively handle memory management requests from the OS kernel, EKC must provide well-defined APIs that permit legitimate operations, while preventing unauthorized actions, such as illegal access to page tables.

Invariant 2: EKC mediates all trap events.

This invariant de-privileges the OS kernel in **trap handling**, ensuring EKC's control flow integrity and providing secure applications for user process. Inv. 2-1 through Inv. 2-3 support Invariant 2 by guaranteeing that EKC always inspects traps first while leaving routine interrupt handling to the OS kernel unaffected.

Inv. 2-1: Only EKC can configure the trap handler in the Interrupt Vector Table Register (`IVTR`). EKC initially configured `IVTR` to guarantee that whenever an interrupt occurs, the control flow should be first transferred to EKC for security scrutiny. This sub-invariant ensures that the OS kernel cannot bypass EKC to redirect the trap into itself.

Inv. 2-2: Traps triggered by EKC are handled within EKC. Allowing the OS kernel to handle interrupts triggered by EKC directly could violate EKC's control flow integrity. This sub-invariant eliminates such risks by requiring that, when EKC is executing, it must either handle interrupts itself or disable them.

Inv. 2-3: Traps occurred outside EKC are delegated to the OS kernel. Although all traps transfer control flow to EKC first, most of them need to be handled by the OS kernel (e.g., timer interrupts). EKC delegates the handling of these traps to the OS kernel, reducing the attack surface of EKC.
Remarks. To protect secure applications from the OS kernel, as discussed in Sec. 3.4, EKC can incorporate additional trap handling routines to directly handle traps triggered by these applications.

Invariant 3: EKC's memory space is inaccessible to the OS kernel.

This invariant ensures that the OS kernel is de-privileged from modifying EKC's memory, via either virtual addresses mapped to EKC's physical addresses or accessing these physical addresses directly. Inv. 3-1 through Inv. 3-3 are used to support Invariant 3, by preventing the OS kernel from accessing EKC's memory.

Inv. 3-1: The virtual address of the EKC is inaccessible to the OS kernel. EKC's virtual address space is isolated from the OS kernel by ensuring that no virtual address in the OS kernel 's page table maps to EKC 's physical memory.

Inv. 3-2: The physical address of EKC is inaccessible to DMA (Direct Memory Access) devices. Since EKC only controls the page table to restrict virtual address access, additional mechanisms are required to prevent unauthorized DMA access to EKC's memory.

Inv. 3-3: No specialized instructions exist to perform DMA. In some hardware architectures, there exist specialized instructions that can directly access physical address space. EKC must incorporate targeted solutions specific to such architectures.

All the sub-invariants collectively serve as sufficient conditions to guarantee the security of EKC. Therefore, the following chapters will focus on satisfying all sub-invariants to address **C1**.

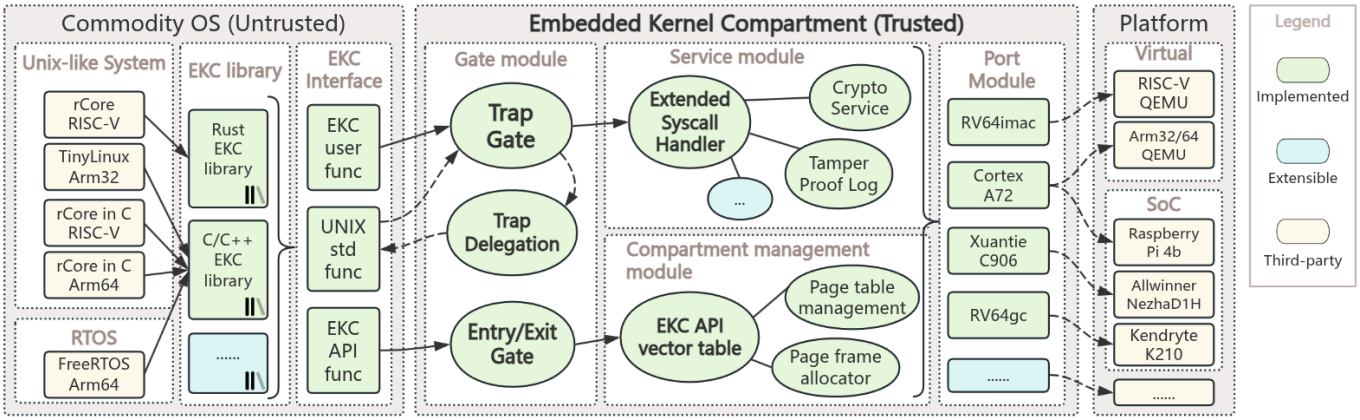


Figure 2: Overall design of EKC

5 Design

To ensure compartmentalization security, EKC must comply with all invariants previously discussed in Sec. 4. As illustrated in Fig. 2, EKC consists of two main components: the compartment management module and the gate module. The compartment management module efficiently manages all page tables and page frames, enforcing Inv. 1-1, Inv. 1-2, Inv. 2-1, Inv. 3-1, Inv. 3-2 and Inv. 3-3. The gate module, comprising the Trap Gate and the Entry/Exit Gate, serves as the primary communication channel with external systems, enforcing Inv. 1-3, Inv. 2-2, Inv. 2-3. The service module, although it does not enforce any invariants, can provide comprehensive security services for user processes. Developers can implement custom built-in service modules to support specific kernel-level functionality.

To address **C4**, the port module is designed to support a wide range of hardware platforms and instruction sets. To enable secure service integration and the functionality of EKC API, supporting libraries are implemented in both Rust and C/C++. These libraries facilitate EKC's adaptation across multiple operating systems, including various Unix-like and real-time systems.

As outlined in section 2, a kernel compartmentalization mechanism involves five key actions: *creating/destroying compartment, assigning privilege*, and *calling/returning from compartments*. Therefore, the EKC workflow, structured around these actions, will be detailed in the subsequent parts of this section. The sections in which these invariants are satisfied are briefly summarized in Table 3 and further elaborated upon in the following subsections.

Table 3: Summary of security invariants

Invariant	Solution	Section
Inv. 1-1	External support available	5.2
Inv. 1-2	Restrict page table mapping	5.1
Inv. 1-3	Well-defined interfaces via the Gate	5.3
Inv. 2-1	External support available	5.2
Inv. 2-2	Trap handler design	5.3
Inv. 2-3	Trap delegation design	5.3
Inv. 3-1	Restrict page table mapping	5.1
Inv. 3-2	MMIO address translation enabled	5.2
Inv. 3-3	Instruction Scanning	5.2

5.1 Creating EKC Compartment

In the original OS kernel startup process (depicted in Fig. 3.a), four kernel modules are initialized sequentially before the control returns to the initial user process. In contrast, during the OS kernel startup with EKC (shown in Fig. 3.b), the OS kernel initializes its page table via EKC API instead of directly configuring the MMU and sets up a trap delegation handler in place of a traditional trap handler. After initialization, any privileged operations required by the OS kernel must invoke EKC API to communicate with and return control from EKC.

In EKC's initialization, various compartments are created. To comply with **C4**, this startup process is designed to minimize its impact on the OS kernel's initialization. EKC ensures that the creation of compartments interferes minimally with the standard initialization routines. As illustrated in Fig. 3.b, the startup process comprises three stages:

- Page table initialization: EKC initializes two independent page tables. The first is dedicated to EKC, granting access solely to its own space. The second is allocated to the OS kernel, which lacks access rights to the EKC space but retains full access to its own OS kernel space.
- Trap initialization: EKC registers a global trap handler and configures trap delegation handler from the OS kernel.
- Gate initialization: EKC initializes all implemented Appli-

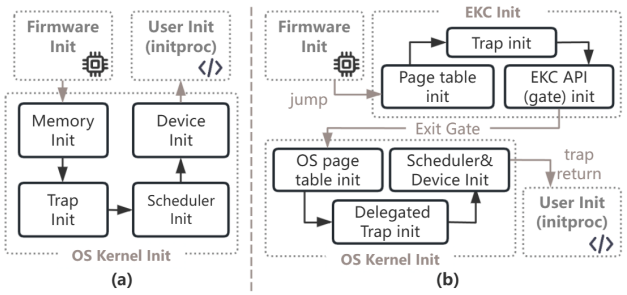


Figure 3: (a) Commodity OS startup; (b) Commodity OS startup with EKC

cations components, such as tamper-proof logs and crypto services. Syscalls with IDs greater than 0x400 are reserved as Service Call, enabling access to these services.

After the startup process, EKC uses the Exit Gate to transfer control to a predefined address, corresponding to the OS kernel's entry point. This transition involves constructing a proxy context with the same parameters as those provided by the firmware, ensuring minimal modifications to the OS kernel.

In summary, three key tasks must be completed during EKC's startup to create a compartment:

- **Create page table for EKC and the OS kernel.** As shown in Fig. 4, the physical address space is divided into three distinct regions: EKC space, OS kernel space, and Trampoline. EKC initializes two separate page tables, each representing the memory space of a distinct compartment. These page tables assign different access rights to each memory region.
- **Configure the compartment domain ranges.** To support Inv. 3-1, EKC ensures that OS kernel's virtual address is never translated to a physical address within EKC space. This prevents OS kernel from accessing EKC's compartment. Additionally, to support Inv. 1-2, the physical address of any page table must reside within EKC space, which deprivileges OS kernel and prevents it from accessing the page tables.
- **Establish Entry/Exit Gate and Trap Gate, then return to the OS kernel.** EKC initializes the trap handler using trap delegation in Trampoline, ensuring that the handler cannot be replaced. This design prevents the OS kernel from interrupting EKC while allowing user process to invoke and return from EKC. Next, EKC initializes the Entry/Exit Gate in Trampoline, enabling OS kernel to call and return from EKC. The Entry/Exit Gate manages the page tables, control flow, and context switch under kernel privilege. Consequently, a proxy context is established for both EKC and the OS kernel, with the Entry/Exit Gate responsible for securely switching between them.

5.2 Assigning/Restricting Privilege

As outlined in the invariant properties (Sec. 4), access to `RPTR` and `IVTR` must be restricted to satisfy Inv. 1-1 and Inv. 2-1. Additionally, restricting direct access to physical addresses is necessary to support Inv. 3-2 and Inv. 3-3. Achieving these requirements necessitates external support. We propose two solutions: instruction scanning or the use of hardware-assisted mechanisms.

Solution 1: instruction scanning. Since EKC has full control over the page tables, it can identify pages with executable permissions and inspect the instructions they contain. Any instructions that attempt to access `RPTR`, `IVTR`, or perform DMA operations can be removed. Instruction scanning is a widely adopted approach on mainstream platforms, though it may be less efficient compared to hardware-assisted methods.

Solution 2: hardware-assisted mechanisms. Specific hardware features can be leveraged to address the requirements of Inv. 1-2 and Inv. 2-2. For instance, Arm64 provides two *TTBR* registers, allowing EKC to exclusively use one for secure address translation. Similarly, RISC-V offers the TVM feature that restricts OS kernel from accessing the `RPTR` register.

For Inv. 3-2, the Input-Output Memory Management Unit (IOMMU) plays a critical role in isolating the address of DMA-capable I/O devices from unauthorized memory regions. When a device supports DMA with IOMMU, the MMIO address space of the DMA remapping unit is never mapped in the OS kernel's page table. Consequently, the IOMMU, under EKC's control, ensures that EKC space remains inaccessible.

On platforms that support extended ISAs with DMA capabilities, the only viable approach may involve inspecting the corresponding hardware extension features provided by the ISA to identify suitable hardware-assisted mechanisms.

5.3 Calling/Returning from EKC

In EKC architecture, there are three kinds of control flow transfer among the three roles:

- user process traps into OS kernel (e.g., syscall).
- user process traps into EKC (e.g., using Applications).
- OS kernel accesses EKC (e.g., allocating pages).

The Entry/Exit Gate facilitates call and return between any two roles. It ensures that privilege switching and control flow transfer are performed atomically, maintaining consistency and security during the process.

Privilege Switching between EKC and the OS kernel via Entry/Exit Gate. In EKC architecture, the Entry/Exit Gate is responsible for modifying `RPTR`, enabling privilege level transitions through context switching and updating the root page table address, as illustrated in Fig. 4. The page table used during EKC execution grants read and write access to

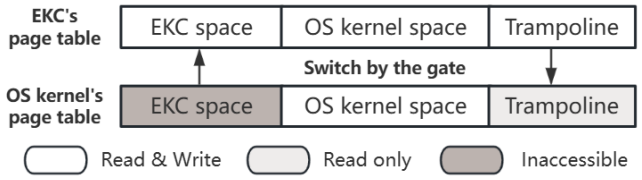


Figure 4: Privilege separation and switching via Entry/Exit Gate between EKC and the OS kernel

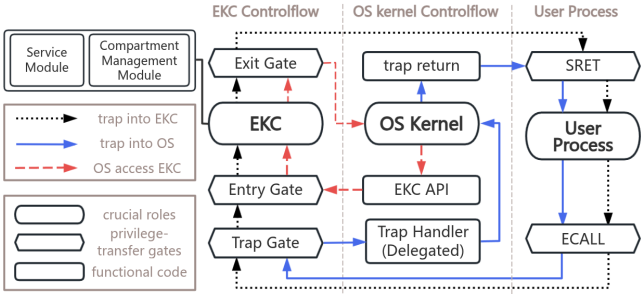


Figure 5: Control flow management in EKC

the entire address space. However, the OS kernel is prohibited from accessing EKC or modifying the Trampoline.

The Entry/Exit Gate is accessible to the OS kernel as it is mapped to the virtual address of the Trampoline, with read and execute permissions explicitly granted by EKC. To satisfy the requirements of Inv. 2-3, the Entry/Exit Gate disables all interrupts during EKC execution.

Privilege Switching between EKC and user process via Trap Gate. To satisfy **C2**, EKC provides a Trap Gate for user process. The Trap Gate is an assembly code segment within the Trampoline responsible for managing interrupts. When an interrupt occurs, the Trap Gate examines the trap context to determine whether the interrupt should be handled by EKC. If so, it invokes the Entry/Exit Gate and subsequently returns control to user process.

Privilege Switching between OS kernel and user process via Trap Delegation. When the trap context indicates that the OS kernel should handle an interrupt, the Trap Gate executes a trap delegation procedure. This dynamic mechanism supports Inv. 2-2. If EKC generates an interrupt, the Trap Gate must disable trap delegation during EKC execution to satisfy the requirements of Inv. 2-3.

Control Flow Management. Consider all the three privilege switching schemes above together, the complete control flow management process involving these gates is illustrated in Fig. 5.

For traps from user process to the OS kernel, a trap delegation mechanism is implemented via the Trap Gate. The trap first enters EKC, which then delegates it to the OS kernel. The OS kernel processes the trap and returns control directly to user space.

For traps from user process to EKC, Service Call is a set of syscalls managed by EKC rather than OS kernel. These syscalls allow the user process to request specific security operations from EKC. Developers can implement syscall handlers in EKC, accompanied by the corresponding drivers in the user library.

To support Inv. 1-3, EKC provides a set of interfaces through the Entry/Exit Gate, collectively referred to as **EKC API**, which is invoked when control flow transfers from the OS kernel to EKC,. To satisfy the requirements of **C4**, the EKC API is standardized in a syscall-like format and provides a range of functionalities to support the basic memory management needs of the OS kernel, such as page table activation and page table frame allocation. Additionally, the EKC API can be leveraged to provide security-related applications for the OS kernel.

5.4 Runtime Compartments Management

Creating/Destroying Compartment at Runtime. The OS kernel can request the creation or destruction of compartments through the EKC API. EKC rigorously verifies the validity of all requests to ensure that they do not compromise the integrity of the compartmentalization mechanism.

When the OS kernel attempts to create more compartments using the EKC APIs, it must provide the following information:

- The owner process of the new compartment.
- The address space range allocated to the compartment.
- The initial context (or binary file) for the compartment.

EKC verifies that the address area is not occupied by other existing compartments and updates the page table mappings accordingly. Once validated, EKC prepares a dedicated context and page table for the new compartment. When an interrupt occurs, the Trap Gate always switches the context to the OS kernel to handle the interrupt.

Updating Compartment Privilege at Runtime. Depending on the security level of each compartment, EKC supports configurable privilege rules as follows: (1) The private memory region of a compartment must not be mapped by other page tables. (2) The page table of a compartment must not map memory outside its assigned private region.

In the basic EKC architecture, rule (1) is enforced by EKC, while the OS kernel imposes no such restrictions. As the highest-privileged compartment, EKC can access any other compartment but cannot be accessed by other compartment. EKC is responsible for assigning individual privileges to each compartment and ensuring that the mapping rules are updated accordingly to enforce isolation.

Calling/Returning from different OS Compartments at

Runtime. In EKC, page tables with different mapping rules are treated as distinct compartments. Therefore, switching between page tables with different mappings is equivalent to transferring control flow to a different compartment.

To implement call/return semantics, the OS kernel invokes the EKC API to switch to the target page table and register context, then jumps to a fixed address within the destination compartment. Several arguments are passed into the compartment based on the caller's register context. The target compartment must register a handler at the designated entry point to process the incoming call.

6 Implementation

In this section, more details of EKC implementation are presented in order to explain how EKC is implemented and deployed in various ISAs and OSes.

6.1 Compartment Management Module

Compartment Memory Management. As shown in Fig. 6.b, assuming that the firmware base address is `BASE`, the EKC image is loaded at `BASE+OFFSET1`, and the OS kernel at `BASE+OFFSET2`, where `OFFSET2>OFFSET1`. The bootloader jumps to `BASE+OFFSET1`, which is the entry point of EKC.

In the EKC mapping (Fig. 6.a), the EKC space's virtual address is identically mapped to physical memory, ranging from `BASE+OFFSET1` to `BASE+OFFSET2`. This region contains the EKC binary and critical runtime components such as page tables, the EKC stack, and the trap handler. Similarly, the OS kernel space is also identically mapped, starting from `BASE+OFFSET2`. The Trampoline has three pages with the largest virtual page numbers, containing the Entry/Exit Gate, the Trap Gate, and the proxy context, respectively. These pages are mapped to the code regions of EKC.

In the OS kernel mapping (Fig. 6.a), the entire OS kernel space is also identically mapped, allowing the operating system to boot normally. The Trampoline is also mapped but with execute-only permission. During execution, the OS kernel may configure virtual memory by calling EKC via Trampoline. It cannot map any physical memory that falls within the reserved EKC memory region.

Compartment Privilege Control. As mentioned in Sec. 5.2, modifications to `RPTR` and `IVTR` by the OS kernel must be prohibited to ensure that EKC is the only compartment permitted to assign privilege to other compartments.

EKC addresses this challenge via instruction scanning [27]. Instruction scanning is a portable and efficient way to identify undocumented or faulty CPU instructions. Before EKC assigns executable permissions to a page table of the OS kernel, the compartment management module performs instruction scanning to verify all instructions in the page. Upon successful verification, the page is marked as read-only and executable.

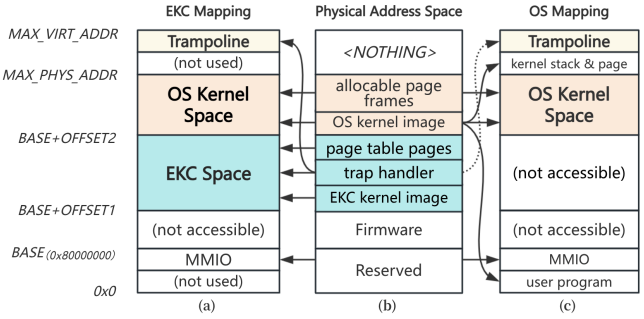


Figure 6: An address space management (a) Physical address space (b) EKC mapping (c) OS kernel mapping

To enforce this protection mechanism, EKC enforces two additional restrictions on page table management:

- Instruction scanning is conducted on every executable page. If the compartment management module identifies instructions that attempt to update `RPTR`, `ISR`, `IVTR`, the page would be considered invalid and rejected by EKC.
- The $W \oplus X$ is strictly enforced. A physical page frame cannot be writable and executable at the same time. Once a physical page is marked as executable, the writable permission is removed from all corresponding virtual pages.

While the instruction-scanning-based solution is effective, a hardware-assisted alternative is available on RISC-V. The RISC-V ISA provides more efficient and direct control over `RPTR` and `IVTR` via the Trap Virtual Memory (TVM) bit and trap delegation logic. When the TVM bit is set, modifications to *satp* in S-mode will raise an interrupt handled by OpenSBI. The RISC-V trap delegation mechanism ensures that OpenSBI has priority in handling such traps.

6.2 Gate Module

Entry/Exit Gate. As introduced in Sec. 5.3, the Entry Gate must complete the following tasks sequentially and **atomically** to protect the control flow integrity:

- Disable timer and software interrupts.
- Switch the context from OS kernel to EKC.
- Switch to the page table of EKC.

After these, EKC handles control flow in the correct privilege level. Similarly, the Exit Gate reverses all the actions above atomically to return OS kernel.

We can utilize `RPTR` to implement a hardware agnostic approach to ensure the atomicity of the execution of the gates. As illustrated in Alg. 1, `RPTR` will be set to an invalid memory address first. Since the translation look-aside buffer (TLB) is not cleaned immediately after modifying `RPTR`, TLB ensures that the code executes normally. Then the Entry Gate loads EKC context and `RPTR` of EKC. After swapping to new `RPTR`,

EKC handles requests of OS kernel. If OS kernel directly jumps to `<label 1>`, targeting to switch page table without switching context, `RPTR` conflicts with the predefined value and the system panics. With `RPTR` and TLB, the atomicity of the Entry Gate is ensured.

Algorithm 1 Entry Gate implementation

Input: *params* for *call_ekc* from OS
Output: *NO RETURN*

```
RPTR ← 0
ctx = load_EKC_context()
<label 1>
a0 ← get_pagetable_address_from(ctx)
swap(RPTR, a0)
if a0 != 0 then
    panic()
end if
clean_TLB()
call_ekc(params)
```

EKC API. EKC builds an EKC API vector table, which contains the address of EKC API's handlers in Trampoline. EKC API uses a syscall-like design: The value in a7 register is EKC API id, which is used to query the handler's address in the EKC API vector table. The OS kernel is responsible for preparing parameters in a0-a6 register for a specific EKC API and jumping to the Entry Gate. The Entry Gate handles the privilege switching and transfers the control flow to the handler program defined in the EKC space.

Trap Gate. In the trap initialization, OS kernel sets the trap delegation handler via EKC API. As shown in Alg. 2, the Trap Gate verifies interrupt type first. For Service Call, Entry/Exit Gate will be triggered immediately and EKC handles the interrupts. Other traps are delegated to the handlers in OS kernel. The Trap Gate in the Trampoline is marked as execute-only for OS kernel.

Algorithm 2 Trap Gate implementation

Input: *ctx* from caller
Output: handler return value

```
t = interrupt_type()
if t is Service Call then
    a_0 = EKCAPI_TYPE_INTR
    ret = EKC_apicall(a_0, ctx)
    return ret
end if
OS_handler = fetch_config(OS_HANDLER)
ret = OS_handler(t, ctx)
return ret
```

Service Call. In this system, Service Call is defined as a system call whose syscall ID exceeds 0x400, distinguishing it from standard system calls. Service Call is utilized to implement security applications, such as cryptographic service and tamper-proof logs.

To prevent EKC API and Service Call from entering undefined states, the following basic principles, as suggested by the Principle of Least Privilege [44], should be applied when checking parameters from OS kernel and user process:

- Null and invalid pointer validation.
- Pointers must not point to address in EKC memory space.
- When a pointer is encountered, the data it references should be copied into EKC memory space before usage.
- EKC cannot be accessed concurrently by multiple cores.

6.3 Port Module

The Port Module is a component of EKC responsible for adapting EKC to different ISAs. Therefore, the Port Module is highly ISA specific. Majority of the unsafe Rust code and assembly code of EKC resides in this module.

Elements in the Port Module. The Port Module specifies the following parameters of EKC:

- Memory Layout: The physical address range of EKC and OS kernel, and the entry address of OS kernel.
- Platform-specified configuration: The format of PTEs and instructions for reading/writing `RPTR`, `IVTR`, `ISR`, as well as the implementation of the Entry/Exit Gate and Trap Gate.
- Device drivers: The device drivers for specific hardware (e.g. character devices, TPM).

Instruction Scanning Rules. Although instruction scanning is a general solution, different instruction sets and different ISA extensions require distinct scanning methods. Therefore, providing appropriate rules for the corresponding platform is a prerequisite for the removal of illegal instructions. We implemented two scanning rules for Arm and RISC-V. When the following kinds of instructions are found in any executable pages of OS kernels, it would be removed:

	Arm	RISC-V
write RPTR	MCR p15, 0, *, c2, c0, *	csrrw stvec, *
write IVTR	MCR p15, 0, *, c12, c0, 0	csrrw satp, *

'*' means any register or any number.

6.4 Guidelines for Embedding EKC

This section outlines the key steps for developers to integrate EKC into a general-purpose OS kernel.

Boot Process. First, modify the kernel entry point defined in the `linker` script. Instead of using the default address provided by the bootloader, it is explicitly set to the address defined by EKC.

Next, `EKC-ALLOC` should be invoked in the `entry` assembly file, which is initially marked as R/W. This allows for rapid initialization of the early page table and mapping of critical memory regions such as MMIO. During subsequent kernel initialization, any direct modification to `satp` should be replaced with a call to `EKC-CONFIG`, and the trap handler should be modified to comply with the trap delegation format.

Memory Management. For OSes with page table support, the code segments responsible for modifying page table entries (PTEs) should be replaced with corresponding EKC calls. Typical cases include: (1) Page Fault Handling. In most OSes, when a page fault occurs, a physical memory page is allocated and mapped into the user address space. During this process, `EKC-ALLOC` should be invoked to ensure that the PTE is updated by EKC. (2) Device Mapping. For PTE modifications involving MMIO or pages that are mapped to fixed addresses (e.g., trap handlers), `EKC-ALLOC` should be invoked.

In OS kernels without a memory management module, such as certain real-time OSes (e.g., FreeRTOS [17]) or OSes designed for low-end CPUs (e.g., Linux-noMMU [3]), this step is generally unnecessary, as memory layout is static.

User Process Management. When a user process is created or terminated, a unique process tag (e.g., pid or tgid) is used to manage its corresponding page tables. `EKC-PTINIT` should be invoked to initialize new page tables and `EKC-FORK` to duplicate an existing address space during process creation. To support the execution of new binaries (e.g., via the `exec` syscall), `EKC-WRITE` can be used to load binary contents into the process's user space memory. Finally, during context switches, the OS kernel should propagate the process tag to the context switch routine. `EKC-ACTIVATE` must then be invoked to switch between per-process page tables.

7 Analysis and Evaluation

Deployed platforms and operating systems. As summarized in Table 4, EKC has been deployed across a diverse set of operating systems and hardware platforms. Among them, rCore [7] and rcc [8] are open-source educational operating systems, implemented in Rust and C respectively. FreeRTOS is a widely used open-source real-time operating system (RTOS) kernel. TinyLinux is a minimalist variant of the Linux kernel, derived from version 2.6.35. All experiments were conducted on the following platforms:

- QEMU [6], a cross-platform system emulator.
- Kendryte 210 [2], equipped with a RV64GC 400MHz processor and 64MB SRAM.
- Allwinner D1-H board [11], equipped with a C906 RISC-V 1GHz processor and DDR3 1GB DRAM.
- Raspberry Pi 4b board [38], equipped with a Cortex-a72 AArch64 1.8GHz processor, and DDR4 1GB SDRAM.

Table 4: Deployed platforms and OSes

ISAs	Platform	OS kernel	OS type
Arm32	Qemu 6.3.0	TinyLinux	Linux-based
Arm64	Qemu 6.3.0	rCore in C	Unix-like
	Raspberry Pi 4b	FreeRTOS	Real-time
RISC-V	Qemu 6.3.0	rCore in C	Unix-like
	Allwinner D1H	rCore in C	Unix-like
	Kendryte 210	rCore	Unix-like

Code base. The current implementation of EKC comprises approximately **28K lines of safe Rust, 450 lines of unsafe Rust, and 2.2K lines of assembly code** (see Table 1). The unsafe portions, including both unsafe Rust and assembly codes, are primarily concentrated in the port module, which facilitates hardware and ISA support.

Integrating EKC into an operating system typically requires minor code modifications. For instance, embedding EKC into TinyLinux involved **16 files changed, 282 insertions and 77 deletions**. A detailed explanation of how EKC is ported to TinyLinux and how it works within the TinyLinux can be found in Appendix C.

7.1 Security Analysis

Under our threat model (see Sec. 3.2), two key aspects are considered in the security analysis of EKC: (1) the security of compartmentalization (**C1**); (2) memory safety of EKC's implementation, particularly the interfaces it exposes to OS kernel (**C3**). The first aspect is validated by an in-depth analysis of how invariants (see Sec. 4) address all potential vectors that an adversary maybe exploit to bypass compartment boundaries. For the second aspect, we examine relevant CWEs and conduct code analysis to identify and mitigate security vulnerabilities.

Enforcing Kernel Compartmentalization. We consider five attack vectors through which the OS kernel can breach the security of EKC:

- Access the virtual address of EKC space.
- Directly access the physical address of EKC space.
- Directly modify `RPTR` to change the page tables.
- Trigger interrupts to hijack the control flow of EKC.
- Exploit the vulnerabilities in the Entry Gate.

The security of EKC can be guaranteed as follows: (1) Inv. 1-1 and Inv. 3-1 ensure that the physical address of EKC space and its page tables are not mapped, resulting in a page fault if OS kernel attempts to access them. (2) According to Inv. 3-2 and Inv. 3-3, if DMA is permitted, IOMMU must be properly configured, and any instructions that could enable DMA access must be removed via instruction scanning. (3) Both the instruction scanning solution and hardware-based alternative (referred in Sec. 5.2) support Inv. 1-2 by preventing

such attacks. (4) Inv. 2-1 ensures that the trap handler resides in EKC thereby preventing control flow from escaping EKC during trap handling. Inv. 2-2 guarantees that the trap handler cannot be changed. (5) `RPTR` serves as an atomic lock to ensure that the code within the gate executes as an indivisible unit. Adversaries cannot execute a fraction of the code or alter the control flow after privileges have been elevated.

Beside these five attack vectors, the adversary may attempt to indirectly breach the security of EKC via improper calls to the interfaces. The main risks of the interface can be summarized as control-flow attack and data-oriented attack. EKC ensures the atomicity and security of Entry/Exit Gate execution via code analysis, thereby preventing control-flow hijacking. To mitigate data-oriented attacks, EKC rigorously validates all parameters provided by the OS kernel and user process, rejecting any illegal or malformed inputs, as further detailed in the following sections.

Analysis of Top CWEs. This section analyzes EKC's security by examining some representative CWEs from the *Weaknesses in the 2023 CWE Top 25* [22]. Additionally, as outlined in *CWE Research Concepts* [24], all CWEs are grouped into ten categories, summarized in Table 5.

Rust's secure compiler and static analysis tools address common vulnerabilities like *NULL Pointer Dereference*, *Insufficiently Random*, and *Integer Overflow*. These are detailed further in the analysis of code implementation. To secure interactions between EKC and the OS kernel, only basic types are used as parameters, and rigorous validation is performed to mitigate risks such as *Unrestricted File Upload*, *Out-of-Bounds Access*, and *Improper Neutralization*.

The risk of *Use After Free* vulnerabilities in Rust-based EKC is minimal. Even if a UAF issue were to occur, the adversary would still be unable to alter PTEs since page tables are confined to the EKC space. In contrast, other free page frames are allocated in the OS kernel space using separate allocators, thereby preserving strict spatial separation.

In multithreaded environments, EKC avoids race conditions by processing only one request at a time. In multicore configurations, each core operates independently without shared memory, significantly reducing the risk of concurrency-related issues.

EKC currently does not address the problem of *Missing Authentication for Critical Functions* yet. Although the interfaces exposed to OS kernel are limited, there is no authentication mechanism to verify OS kernel modules. This presents a potential risk: malicious modules, although unable to directly compromise EKC, could exploit EKC to target other kernel modules. Future improvements will focus on implementing authentication mechanisms for critical interfaces to mitigate this vulnerability.

Analysis of the Rust-Based Implementation. Assuming the Rust core library, architecture-support libraries (e.g. *cortex-a* [42]), and the Rust compiler tools are trused, the safe Rust code can be considered secure, which is guaranteed by the strong type safety and thread safety of Rust. The unsafe Rust code is checked by MirChecker [32], which identifies five types of undefined behaviors, i.e., memory safety, inline assembly, divided by zero, overflow, and panic. The result is shown in Table 6. Regarding the unresolved warnings given by MirChecker, we have manually checked the specific code corresponding to each warning:

- The majority of **Memory Safety (M)** warnings comes from dereferencing fixed address pointers, such as dereferencing PTEs, Trampoline and Trap Context.
- The majority of **Inline Assembly (I)** warnings are inevitable and involve simple access to CSR registers. Others are the call to Entry/Exit gate.
- The unresolved **Overflow (O)** warnings primarily pertain to bitwise overflow issues, particularly in the context of operations involving flag bits of page table entries.
- The unresolved **Panic (P)** situation are checked not lead to the exposure of sensitive data.

Regarding to assembly code, there are mainly two parts of assembly codes: Entry/Exit Gate and Trap Gate. As a result of the limited code size, we have meticulously reviewed every line of assembly code, validating that their execution is both atomic and secure. More details are described in Appendix A.

Furthermore, to analyze memory safety on the whole project, Rudra [16] is a static analyzer to detect common undefined behaviors. The report in Table 7 showed that EKC had no potential risks in the three aspects it checked.

7.2 Performance Evaluation

This section analyzed the evaluation results from the unit tests, bench tests, and system tests. The specific values for all evaluation results are listed in the Appendix B.

Unit Test. In EKC, there are mainly 3 modules: compartment management module, gate module, and service module. The basic tasks of these are measured individually with the results shown in Table 8. We tried call/return from EKC and the OS kernel to user process with or without gates to calculate the time usage of the raw gate. For comparison, timer interrupt handling in FreeRTOS usually costs 50us. The latency of the gate is within acceptable limits. We also measured the time usage of instruction scanning for one page frame and the time usage of hashing the memory to get the measurement of the kernel image each executable page frame in OS kernel would bring a 2.8ms latency, and the latency of 554ms per page would affect kernel measurement during startup. Other service examples are also measured in time, but their performance is largely determined by the code implementation of the service, making them not as valuable a reference. More rigorous measurement methods for this can be found in system tests for services.

Table 5: Representative CWEs in the 2023 CWE top 25

Categories	Representative CWEs in 2023 Top 25 [22]
Improper Access Control	Improper Authorization (285), Missing Authentication for Critical Function (502)
Improper Interaction Between Multi-Correctly-Behav Entities	Unrestricted Upload of File with Dangerous Type (434)
Improper Control of a Resource Through its Lifetime	Use After Free (416), Out-of-bounds Read/Write (125/787)
Incorrect Calculation	Integer Overflow or Wraparound (190)
Insufficient Control Flow Management	Race Condition (362)
Protection Mechanism Failure	Use of Insufficiently Random Values (330)
Incorrect Comparison	Incorrect Regular Expression (185)
Improper Check or Handling of Exceptional Conditions	NULL Pointer Dereference (476)
Improper Neutralization	Improper Input Validation (20), Code Injection (94)
Improper Adherence to Coding Standards	NULL Pointer Dereference (476)

Table 6: Number of LoCs and warning from MirChecker

-	Number of LoCs			MirChecker Warning				
Module Name	S[1]	U[1]	A[1]	M[2]	O[2]	D[2]	P[2]	I[2]
CM Module[4]	1616	26	0	1	2	0	0	0
Service Module	1208	162	2	0	4	0	0	1
Gate Module	295	31	305	2	0	0	2	2
Port Module[3]	1321	34	227	5	1	0	2	9

[1] S=Lines of Safe Rust, O=Lines of Unsafe Rust, A=Lines of Assembly
[2] M=Memory-Safety, O=Numeric-Overflow, D=Divided-by-zero, P=Panic, I=Inline-Assembly
[3] The result is the average of all the current port modules.
[4] Compartment Management Module.

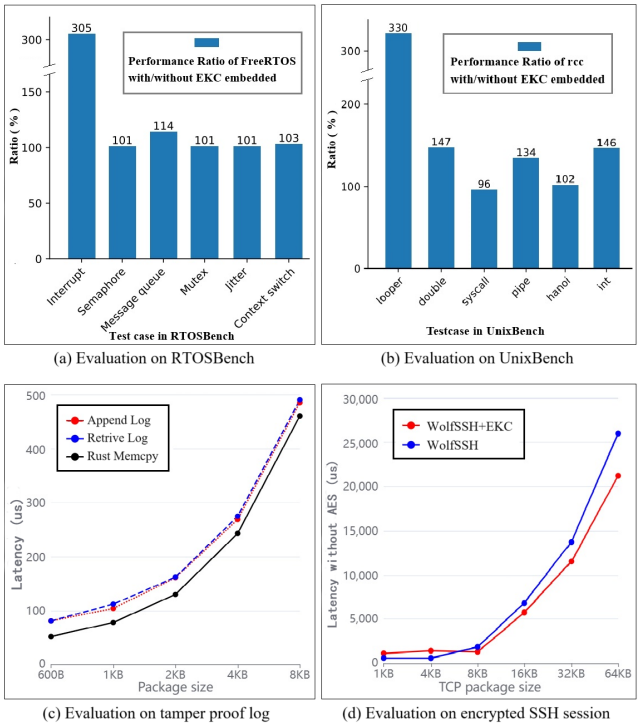
Table 7: Rudra's warning on the entire project

Warning type	Amount
Panic Safety	0
Higher-order Safety Invariant	0
Propagating Send/Sync in Generic Types	0

Table 8: Evaluation on basic modules in EKC

Module	Basic task of module	Latency
CMM[1]	memory hashing (4KB)	554 ms
CMM[1]	instruction scanning (4KB)	2.8 ms
Gate	Entry/Exit Gate call	7.8 us
Gate	Trap Gate call	0.14 us
Service	appending log (4KB)	269 us
Service	AES encrypt (4KB)	1783 us

[1] CMM=Compartment Management Module



(a) Evaluation on RTOSBench
(b) Evaluation on UnixBench
(c) Evaluation on tamper proof log
(d) Evaluation on encrypted SSH session

Figure 7: Evaluations on test bench and case studies

Bench Test. The RTOSBench [9] in FreeRTOS is executed both with and without EKC on Allwinner D1-H board (as shown in Fig. 7.a). In RTOSBench, six indicators are used to assess RTOS performance, with each result representing the average of multiple measurements to ensure accuracy and reliability. Since RTOS typically operates without memory management, most of the cases has only less than 5% overhead, while the primary overhead arises from interrupt delegation, about 205%. These factors introduce minimal performance degradation in most scenarios. However, interrupt processing stands out as an exception, with a performance overhead approximately three times the baseline. This delay is likely due to the additional processing required by EKC or similar mechanisms. Despite this, the overall impact on performance remains manageable, suggesting that EKC can be integrated into RTOS environments without significantly affecting efficiency in most use cases.

The UnixBench [10] in *rcc* is executed similarly on Raspberry Pi 4b board (as shown in Fig. 7.b). In UnixBench, various user processes with tasks are assigned to the operating system to run repeatedly, and performance is measured by the number of tasks executed within 10 seconds. Each result is the average of multiple measurements for consistency and accuracy. According to the results:

- EKC introduces almost no overhead (less than 5%) for general syscalls, but the *pipe* syscall experiences about 30% overhead due to the allocation of buffers.
- EKC has no overhead about mathematical operations like *hanoi* theoretically, but about 50% lags on *int* and *double*.
- The *fork* and *exec* in the *looper* utilize the EKC API frequently, resulting in 230% overhead compared to others.

These two bench tests show that EKC has a limited performance impact on commodity OS, with only some overhead in page table management and interrupt handling.

System Test. To assess the practicality of EKC, we evaluate two case studies as follows:

- We ported the WolfSSH [51] software to a modified FreeRTOS with EKC, storing the WolfSSH AES session keys within EKC. The keys cannot be directly accessed by other compartments, and encryption or decryption can only occur through EKC. In our experiment, a SSH client would establish session to the host with EKC, and EKC would help the host encrypt/decrypt all TCP packets.
- We record a tamper-proof log of kernel execution within EKC and allow user processes to access the log to verify its integrity. In out experiment, users will request EKC to read and write log information of different lengths.

The results of these evaluations are shown in Fig. 7.c and Fig. 7.d. In the tamper-proof log case, the overhead introduced by EKC remains constantly 30us, regardless of the log information size. In the SSH connection case, EKC functions as a cryptographic service provider, serving as the AES backend for WolfSSH to encrypt and decrypt TCP packets from a client. Despite the time required for AES encryption and decryption, the average additional cost introduced by EKC is around 17%, which is in acceptable limit.

These two cases demonstrate EKC 's practicality and efficiency in real-world applications, providing reliable performance with tolerable overhead.

8 Related Works

Previous works on kernel compartmentalization can be divided into three categories as follows. Some representative works are summarized in Table 9.

Hardware-Assisted Solutions. Some existing solutions depend on specific hardware features, for example, HAKCs [34] using Arm's PAC and MTE, SKEE [15] using Arm's TTBCR, KDPM [30] using Intel's MPK. However, they only support limited compartmentalization due to the inherent constraints of their respective hardware platforms. In contrast, EKC is a native cross-platform compartmentalization mechanism as it does not depend on platform-specific hardware.

Table 9: Existing kernel compartmentalization mechanisms

Works	Type[1]	Target[2]	Bounded Mechanism
Hilps [19]	P	U+K	Arm's TxSZ
HAKC [34]	P	K	Arm's PAC+MTE
KDPM [30]	P	K	Intel's MPK
SKEE [15]	P	K	Arm's TTBCR
xMP [39]	H	U+K	VMM
LXDs [35]	H	K	Sandboxing
NK [25]	S	K	Page table
XFI [28]	S	K	Intel's SFI

[1] P=Platform-assisted, H=Hypervisor-assisted, S=Software-based
[2] K=Only targeting OS, U+K=targeting both OS and user process

Hypervisor-Assisted Solutions. xMP [39], VMM-based [41, 49, 53], and sandboxing-based approaches [35, 37] depend on hypervisors to enforce least-privilege separation and enable fine-grained domain isolation. Although these methods are effective within virtual machines, these methods are less suitable for bare-metal systems or real-time operating systems. As a complementary solution, EKC fills this gap by providing effective kernel compartmentalization for bare-metal platforms.

Software-Based Solutions. Nested Kernel [25] is a kernel compartmentalization method that requires almost no hardware features on x86, instead relying on a virtualization-like technology called Secure Virtual Architecture (SVA) [21]. However, it fails to address the trap handling or restrict access to the control registers [13]. SFI-based solutions [28] establish protection through static analysis and runtime code instrumentation. However, their primary goal is to ensure the integrity of the control flow rather than enforcing kernel compartment privilege separation. Inspired by the state-of-the-art works, EKC successfully embedded into the kernel as a privileged compartment, providing secure and practical kernel compartmentalization.

9 Conclusion

We introduced EKC, a kernel compartment designed to integrate seamlessly into commodity OSes as a privileged, isolated module. EKC demonstrates strong portability across multiple ISAs and is extensible to various OS kernels. Our Rust-based implementation of EKC successfully on both RISC-V and Arm ISAs, and is compatible with FreeRTOS, rCore, and TinyLinux. Comprehensive evaluations confirm that EKC is a practical and effective solution for secure kernel compartmentalization.

Ethics considerations

We have carefully reviewed the ethical guidelines in the call for papers. We confirm that our research considers the potential negative impacts, obeys with laws, and contains no disclosures of vulnerabilities.

Open science

We confirm that this submission adheres to the Open Science guidelines of USENIX Security '25. All relevant artifacts associated with this work have been publicly released on `Zenodo`:

https://doi.org/10.5281/zenodo.15534623

The artifacts are organized into three separate archives, as detailed below:

- `RustEKC_src.zip`: Contains the full source code of EKC, including core modules, documentation, libraries, and various testbenches and examples. This archive supports Sections 5, 6, and 7.
- `payloads_src.zip`: Includes the source code of all integrated payloads (rCore, FreeRTOS, TinyLinux), illustrating how each OS kernel was modified to integrate with EKC. This archive supports Sections 6 and 7.
- `RustEKC_artifacts.zip`: Provides precompiled binaries executable in QEMU, enabling readers to directly evaluate EKC. This archive supports Section 7.

Additionally, the most recent source code for EKC is maintained in the open-source repository:

https://anonymous.4open.science/r/RustEKC-34E4

All subsequent updates and bug fixes will be released through this repository.

Acknowledgments

We appreciate the anonymous reviewers for their constructive suggestions. The work was in part supported by National Key R&D Program of China under grant No. 2023YFB4503902 and National Natural Science Foundation of China under grant No. 62361166633 and No. 62472281.

References

[1] Intel® 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[2] Kendryte K210. https://www.canaan-creative.com/product/kendryteai.

[3] Linux. https://github.com/torvalds/linux.

[4] Memory Tagging Extension. https://developer.arm.com/documentation/108035/latest/Introduction-to-the-Memory-Tagging-Extension.

[5] Pointer Authentication Code. https://developer.arm.com/documentation/109576/latest/Pointer-Authentication-Code/.

[6] QEMU. https://www.qemu.org/.

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

[8] rCore in C. https://github.com/shili2017/rcc.

[9] RTOSBench. https://github.com/gchamp20/RTOSBench.

[10] UnixBench. https://github.com/kdlucas/byte-unixbench.

[11] Allwinner DE2.0. Referenced Sep 2024. https://linux-sunxi.org/images/7/7b/Allwinner_DE2.0_Spec_V1.0.pdf.

[12] Adil Ahmad, Sangho Lee, and Marcus Peinado. Hardlog: Practical tamper-proof system auditing using a novel audit device. In *IEEE Symposium on Security and Privacy*, pages 1791–1807, 2022.

[13] Adil Ahmad, Botong Ou, Congyu Liu, Xiaokuan Zhang, and Pedro Fonseca. Veil: A Protected Services Framework for Confidential Virtual Machines. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 378–393, 2024.

[14] Hesham Almatary, Michael Dodson, Jessica Clarke, Peter Rugg, Ivan Gomes, Michal Podhradsky, Peter G Neumann, Simon W Moore, and Robert NM Watson. CompartOS: CHERI compartmentalization for embedded systems. *arXiv preprint arXiv:2206.02852*, 2022.

[15] Ahmed M Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. SKEE: A lightweight Secure Kernel-level Execution Environment for ARM. In *23rd Annual Network and Distributed System Security Symposium*, pages 21–24, 2016.

[16] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, page 84–99, 2021.

[17] Richard Barry et al. FreeRTOS. *Internet*, page 18, 2008.

[18] Alessandro Bertani, Danilo Caraccio, Stefano Zanero, Mario Polino, et al. Confidential Computing: A Security Overview and Future Research Directions. In *Proceedings of the 8th Italian Conference on Cyber Security*, pages N–A, 2024.

[19] Yeongpil Cho, Donghyun Kwon, Hayoon Yi, and Yunheung Paek. Dynamic Virtual Address Range Adjustment for Intra-Level Privilege Separation on ARM. In *24th Annual Network and Distributed System Security Symposium*, 2017.

[20] Jolyon Clulow. On the security of PKCS# 11. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 411–425, 2003.

[21] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 351–366, 2007.

[22] CWE. Cwe view: Weaknesses in the 2023 cwe top 25 most dangerous software weaknesses. 2023. https://cwe.mitre.org/data/definitions/1425.html.

[23] CWE. Cwe-416: User after free. Referenced Mar 2024. https://cwe.mitre.org/data/definitions/416.html.

[24] CWE. Cwe view: Research concepts. Referenced Mar 2024. https://cwe.mitre.org/data/definitions/1000.html.

[25] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 191–206, 2015.

[26] Trung T Dinh-Trong and James M Bieman. The FreeBSD project: A replication case study of open source development. *IEEE Transactions on Software Engineering*, pages 481–494, 2005.

[27] Rens Dofferhoff, Michael Göebel, Kristian Rietveld, and Erik van der Kouwe. iScanU: A portable scanner for undocumented instructions on risc processors. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 306–317, 2020.

[28] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88, 2006.

[29] Samuel Greengard. Will RISC-V Revolutionize Computing? *Commun. ACM*, page 30–32, 2020.

[30] Hiroki Kuzuno and Toshihiro Yamauchi. KDPM: Kernel Data Protection Mechanism Using a Memory Protection Key. In *International Workshop on Security*, pages 66–84, 2022.

[31] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. SoK: Software Compartmentalization. In *IEEE Symposium on Security and Privacy*, pages 75–75, 2024.

[32] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Mirchecker: detecting bugs in rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2183–2196, 2021.

[33] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, pages 103–104, 2014.

[34] Derrick Paul McKee, Yianni Giannaris, Carolina Ortega, Howard E Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing Kernel Hacks with HAKCs. In *29th Annual Network and Distributed System Security Symposium*, pages 1–17, 2022.

[35] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, et al. LXDs: Towards isolation of kernel subsystems. In *2019 USENIX Annual Technical Conference*, pages 269–284, 2019.

[36] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and VM functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 157–171, 2020.

[37] Bryan D Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy*, pages 233–247, 2008.

[38] Raspberry pi Foundation. Raspberry pi 4 model b specifications, Referenced Dec 2024. https://raspberrypi.com/products/raspberry-pi-4-model-b/specifications/.

[39] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis.

xmp: Selective memory protection for kernel and user space. In *IEEE Symposium on Security and Privacy*, pages 563–577, 2020.

[40] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, 2003.

[41] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. In *2009 International Conference on Availability, Reliability and Security*, pages 74–81, 2009.

[42] Andre Richter. cortex-a crates.io. 2018. https://crates.io/crates/cortex-a/7.3.0.

[43] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, pages 39–47, 2005.

[44] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, pages 1278–1308, 1975.

[45] David Seal. *ARM architecture reference manual*. 2001.

[46] Daniel Tabak. *RISC systems*. 1990.

[47] tinyclub. tinylinux based on linux 2.6.35. 2011. https://github.com/tinyclub/tinylinux/.

[48] Dalton Cézane Gomes Valadares, Newton Carlos Will, Marco Aurélio Spohn, Danilo Freire de Souza Santos, Angelo Perkusich, and Kyller Costa Gorgônio. Confidential computing in cloud/fog-based Internet of Things scenarios. *Internet of Things*, page 100543, 2022.

[49] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX Symposium on Operating System Design and Implementation*, pages 181–194, 2002.

[50] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A Patterson, and Krste Asanovic. The RISC-V instruction set manual volume II: Privileged architecture version 1.7. 2015.

[51] wolfSSL Inc. Wolfssh, Referenced Aug 2024. https://www.wolfssl.com/products/wolfssh.

[52] Nicolas Wu. Dirty pagetable: A novel exploitation technique to rule linux kernel. 2023. https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html.

[53] Xi Xiong and Peng Liu. SILVER: Fine-grained and transparent protection domain primitives in commodity OS kernel. In *16th Research in Attacks, Intrusions, and Defenses*, pages 103–122, 2013.

[54] Min Xu, Xuxian Jiang, Ravi Sandhu, and Xinwen Zhang. Towards a VMM-based usage control framework for OS kernel integrity protection. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 71–80, 2007.

[55] Qiu Yi, Xiong Puxiang, and Tianlong Zhu. *The Design and Implementation of the RT-thread Operating System*. 2020.

Appendix A Unresolved MirChecker warning

Regarding the unresolved warnings given by MirChecker, we have confirmed the specific code implementation corresponding to each warning is secure.

Inline-Assembly Warning.

```
asm!("sfence.vma")
...
asm!("jr␣%0", in(OS_ENTRY_ADDRESS));
```

Some simple assembly code as above. They typically perform a single operation at a specific location, such as flushing the TLB, jumping to a constant address, etc. We can simply view these operations as risk-free. Because we know exactly what it is doing and what the expected outcome is.

Overflow Warning.

```
pub fn token(&self) -> usize {
    // calculate the value of satp register.
    return 8usize << 60 | self.root_ppn.0;
}
```

These warnings come from bit operations and are mainly about flag-bit operations as above. Here *8usize « 60* is only a flag-bit of *satp* register.

Panic Warning.

```
    ...
    // some assemblyinstruction that would jump
        to another place
    asm!(....);
    // cause panic, but not reachable
    panic!("not␣reachable.")
```

In fact, most of them are impossible to happen due to inline assembly code before. If, in very exceptional circumstances, there really is an illegal state execution, currently we choose to simply panic the kernel rather than try to deal with it.

Appendix B The detailed result of evaluation

In the benchmark test, the results are recorded directly. In our own test case, the CPU clock is used to evaluate the time usage. Each case is tested multiple times (>3), and the average value is recorded. The data presented in Sec. 7 may have been processed, but all are referenced from here.

The Raw Gate. We invoke EKC API and trap with and without the gate for 100K times to compute the cost, the average result are shown in 10.

UnixBench on *rcc*. The UnixBench would keep executing a program in 10 seconds, and record how many times it executed. The average value were recorded in 13.

RTOSBench on FreeRTOS. The RTOSBench already measured each cases multiple times and provide an average time usage as a report. The report were recorded in Table 12.

WolfSSH. The average time usage of sending packet is recorded in Table 13. The value in bracket is the time usage of AES encryption. Since the AES library in Rust has different performance with the one in C, the difference between the total packet sending time and the AES encryption time is used in the paper.

Tamper Proof Log. In log recording and reading, the main time is spent reading and writing memory, so rust memcpy is used as a reference to observe the performance of EKC as shown in Table 14.

Kernel Measurement and Instruction Scanning. This two basic functions are specifically evaluated, and the average value were recorded in Table 15.

Table 10: Performance of the raw gate

Time Cost	EKC API (x100K)	Trap(x100K)
with Gate	1390 ms	37 ms
without Gate	610 ms	23 ms

Table 11: UnixBench result on rcc

Testcase	rcc with EKC (times/10s)	rcc (counts/10s)
syscall	28231	27157
pipe	10249	13812
int	52016	76145
long	51723	76809
double	51632	76356
float	51735	75993
hanoi	4295	4409
looper	36	155

Table 12: RTOSBench result on FreeRTOS

Testcase	FreeRTOS+EKC (μs)	FreeRTOS (μs)
Context switch	53	51
INTR processing	1987352	649536
MQ[1]send	126	143
MQ[1]receive	37	36
MQ[1]signal	93	97
MQ[1]workload	337	295
Mutex release	134	137
Mutex aquisition	105	101
Mutex workload	167	165
Jitter	221378	218520
Semaphore wait	49	48
Semaphore signal	129	121
Sema. workload	327	322

[1] MQ=Message Queue

Table 13: WolfSSH evaluation

Testcase	wolfSSH+EKC (μs)	wolfSSH (μs)
establish session	15056	13732
send packet 1KB	1733(602)[1]	616(1.8)
send packet 4KB	3209(1783)	609(6.8)
send packet 8KB	4931(3629)	1862(13.2)
send packet 16KB	13003(7235)	6870(26.9)
send packet 32KB	25942(14397)	13797(52.2)
send packet 64KB	49914(28711)	26117(103.6)

[1] All the value in bracket is the time usage of AES library in rust/ C.

Table 14: Tamper proof log evaluation

packet size	append log (μs)	retrieve log (μs)	memcpy (μs)
20B	52	57	22
200B	58	60	29
600B	81	81	52
1KB	104	112	78
2KB	161	162	130
4KB	269	275	243
6KB	378	383	348
8KB	486	491	461

Table 15: Evaluation on hashing and instruction scanning

test case	time usage (ms)
OS kernel image hashing (256KB)	35472
OS kernel image hashing (1MB)	133050
OS kernel image hashing (4MB)	584768
instruction scanning (4KB)	2.8

Appendix C How EKC works in Arm32 TinyLinux

TinyLinux [47] is a light-weighted Linux kernel forked from Linux-2.6.35. Currently, 16 files changed, 282 insertions and 77 deletions have been made to embed EKC into TinyLinux.

Boot Process Modification

In the original Linux boot process (shown in Fig. 8.1a-1d), the boot loader load kernel `zImage` into `LOAD ADDRESS` and jump to it. Then, the decompressor would be decompress the linux kernel to `ENTRY ADDRESS` and execute the linux kernel. The linux kernel would build an early page table and enable MMU to execute itself in virtual address, and then setup the linux kernel. After embedding EKC, the bootloader will load EKC into the original `LOAD ADDRESS` to initialize EKC first (Fig. 8.2a). The `LOAD ADDRESS` and `ENTRY ADDRESS` are move to an higher address, to reserve a section of memory for EKC. After EKC initialization, the MMU is enabled, and EKC would jump to OS's new `LOAD ADDRESS`(Fig. 8.2b).

EKC does not know the structure of `zImage`, so it only the first page in kernel `ENTRY ADDRESS`, where locates the Entry Code (added in `head.S` of `Decompressor`) in zImage. The Entry Code calls the EKC-ALLOC to initialize more accessible memory. Then, the Entry Code make the area where the Decompressor is located executable (Fig. 8.2c), and calls the decompressor. After the Decompressor decompresses the kernel data to the entry address (Fig. 8.2d), the decompressor uses the EKC-ALLOC to make the kernel executable (simultaneously make it not writable) and then jumps to the kernel entry address (Fig. 8.2e).

Early Init Modification

When the Linux kernel is initialized, its executable memory are already been scanned, therefore no instructions that can modify `RPTR` and `IVTR` are available in Linux. Although not shown in detail in the figure, after the Linux kernel is executed, it only needs to keep the `.text` segment as executable, and set other segment to the appropriate read and write permissions using the EKC API.

In the early boot, Linux initialized several memory-related structures and used `bootmem` to manage memory. In order to prevent Linux from trying to access the EKC space (which would otherwise cause panic), it is necessary to reduce the available physical memory area read from the device tree in the early boot and remove the EKC space from it.

Linux mainly used the *create_mapping* function to create large-scale mappings. While retaining all the logic for updating the memory description structure, it replaced the statements that wrote to the page table with calls to the EKC API. Linux cannot write to the stvec register, so it uses the EKC-CONFIG to set the trap delegation address to its own trap handler. Since EKC saves the context before delegating the interrupt, the Linux interrupt handler needs to be modified slightly to adapt to the interrupt processing from EKC.

Kernel Shared Memory

In Linux and most Unix-like systems, user-level page tables usually share mappings for kernel space and device MMIO addresses. EKC continues this design principle. In the early system boot phase, the OS kernel can specify the memory area that needs to be shared through the `EKCAPI-CONFIG` interface. When EKC creates page tables for user processes, this shared area will be automatically included in the page table mapping without explicit specification by user space.

Memory Management Modification

In Linux, the memory management module of the Linux kernel (specifically, the buddy allocator) is tightly coupled with other modules, especially the file system. However, the actual part that allocate pages is the page fault handler. Therefore, in order to minimize unknown impacts, EKC only need to affect the behavior of page fault handler, as shown in Fig. 9. Of course, in order to reduce the number of page fault interrupts and recycle useless memory in a timely manner, we still need to update the page table in some specific places:

- When calling any function relevant to directly mapping, like `remap_vmalloc_range` and `remap_pfn_range`, call the EKC-ALLOC to directly map a range of MMIO address to specific virtual address.
- When calling any function relevant to deallocation, like `do_munmap`, call EKC-DEALLOC.

Process Management Modification

Only when the user process switches/creates/destroys, Linux needs to call the EKC API to synchronously switch/create/destroy the page table.

In `copy_process()` in `do_fork()`, the `pid` and `tgid` of the new process would be initialized. If new `tgid` are assigned, a new page table should also be initialized by `EKCAPI-INIT` with page table number `tgid`. In Linux management, there is a complete memory description structure (`mm_struct`) for this new process, so the page fault interrupt can manage all page allocation of the new process, although this will be a bit slow. If you need to improve performance, use `EKCAPI-FORK` to reduce the triggering of page faults and the number of page frame copies.

In `__switch_to()` in `context_switch()`, the user context are switched by updating *RPTR*. After embedding EKC, the current `tgid` should be given to this function, and `__switch_to()` would call `EKCAPI-ACTIVATE` with the provided `tgid`.

In `release_task()` in `do_exit()`, `EKCAPI-DESTROY` would be called to release all the allocated page tables.

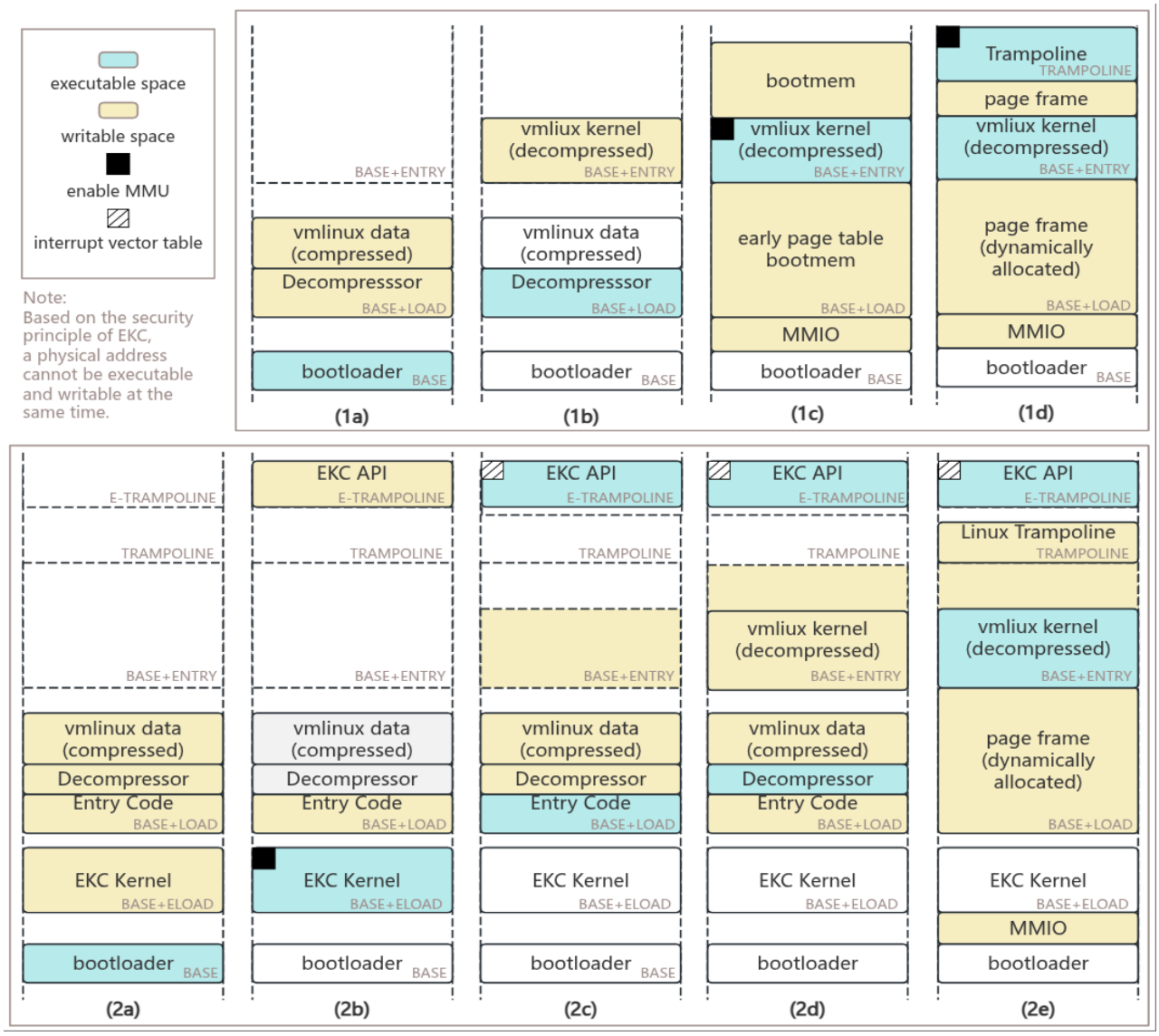


Figure 8: (1a-1d) The original Linux Kernel (2a-2e) The Linux Kernel embedded with EKC

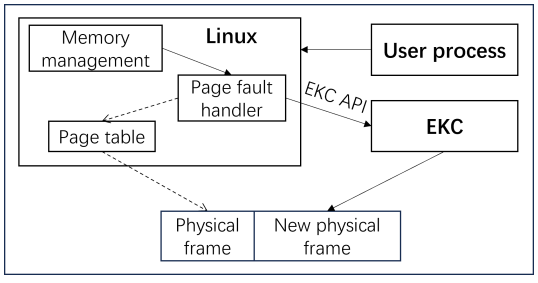


Figure 9: How EKC affect Linux's memory management

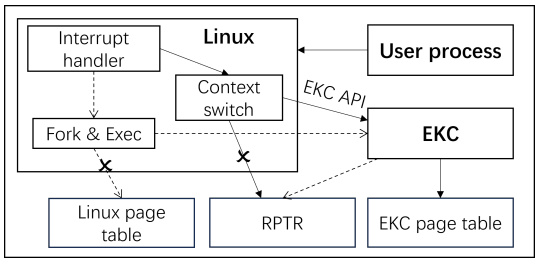


Figure 10: How EKC affect Linux's process management