UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Extending Keystone for Enclave Exclusive I/O Access

Erweiterung von Keystone zur Ermöglichung eines exklusiven I/O-Zugriffs für Enklaven

Masterarbeit

im Rahmen des Studiengangs
IT-Security
der Universität zu Lübeck

vorgelegt von
Valentin Maurice von Plata

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Till Eifert
Claudius Pott
Christopher Peredy

Lübeck, den December 13, 2025

Abstract

While many TEEs provide confidentiality and integrity, they typically neglect availability, particularly with respect to I/O. As a result, enclaves remain dependent on untrusted device drivers and MMIO registers controlled by the host OS, leaving I/O devices vulnerable to tampering and DoS attacks. Recent work on secure I/O for the RISC-V TEE Keystone demonstrates how can isolate devices from the host. However, the approach still relies heavily on the host for IEC and network access, limiting both security and performance. This thesis addresses these limitations through three main contributions. First, we introduce a host-independent IEC mechanism for Keystone. To achieve this, we port and extend the Elasticlave shared memory model to modern Keystone, enabling enclave to enclave communication without host OS involvement. Our evaluation on a Starfive VisionFive 2 board shows that this approach significantly reduces communication overhead compared to Keystone's native OCALL channels. Second, we explore the integration of the seL4 microkernel as a replacement runtime for Keystone enclaves, motivated by its strong formal verification guarantees and richer OS functionality. Although we achieved an initial successful boot of seL4 inside enclaves, architectural constraints, particularly seL4's reliance on static memory layouts, prevented full integration within the available time frame. Third, we propose and implement a PoC for NPS, which forces the untrusted host OS to route all network traffic through a network enclave. We demonstrate the practicality of our designs through two secure I/O applications: one for a simple serial device and one for a NIC, incorporating a network driver and stack inside the enclave. These prototypes illustrate that the combination of IEC, secure I/O, and NPS enables flexible and efficient protection of both simple and complex devices on RISC-V.

Zusammenfassung

Zwar bieten viele TEEs Vertraulichkeit und Integrität, jedoch wird in der Regel die Verfügbarkeit, insbesondere in Bezug auf I/O, vernachlässigt. Infolgedessen bleiben Enklaven von nicht vertrauenswürdigen Gerätetreibern und MMIO-Registern abhängig, die vom Host-Betriebssystem gesteuert werden. Dadurch bleiben I/O-Geräte anfällig für Manipulationen und DoS-Angriffe. Jüngste Arbeiten zu sicherer I/O für das RISC-V TEE Keystone zeigen, wie Geräte vom Host isoliert werden können. Der vorliegende Ansatz ist jedoch nach wie vor stark vom Host für die Inter-Enklaven Kommunikation (IEC) und dem Netzwerkzugriff abhängig. Diese Abhängigkeit hat negative Auswirkungen auf die Sicherheit und die Leistung. Unsere Arbeit befasst sich mit diesen Einschränkungen und leistet drei wesentliche Beiträge. Zuerst wird ein hostunabhängiger IEC-Mechanismus für Keystone eingeführt. Zu diesem Zweck wird das Elasticlave shared memory Modell auf das moderne Keystone portiert und erweitert. Dadurch wird die Kommunikation zwischen Enklaven ohne Beteiligung des Host-Betriebssystems ermöglicht. Wir zeigen außerdem, dass der Kommunikationsaufwand bei Verwendung unseres IEC-Mechanismus auf dem Starfive VisionFive 2-Board im Vergleich zu den nativen OCALL-Kanälen von Keystone erheblich reduziert wird. Ein weiterer Beitrag widmet sich zudem der Integration des seL4-Mikrokernels als Ersatz-Runtime für Keystone Enklaven. Diese Vorgehensweise wird durch die starken formalen Verifizierungsgarantien und die reichhaltigeren Betriebssystemfunktionen des seL4-Mikrokernels motiviert. Obwohl ein initialer, erfolgreicher Start von seL4 innerhalb von Enklaven erreicht werden konnte, verhinderten architektonische Restriktionen, insbesondere die Abhängigkeit von seL4 von statischen Speicherlayouts, eine vollständige Integration innerhalb des zur Verfügung stehenden Zeitrahmens. Ein weiterer Beitrag ist ein Design und PoC für NPS, welches wir implementiert haben. Das Design zwingt das nicht vertrauenswürdige Host-Betriebssystem dazu, den gesamten Netzwerkverkehr über eine Netzwerk-Enclave zu leiten. Die Praktikabilität der vorliegenden Entwürfe wird anhand von zwei sicheren I/O-Anwendungen demonstriert. Die erste Anwendung ist für ein einfaches serielles Gerät konzipiert, die zweite für eine Netzwerkkarte, die einen Netzwerktreiber und einen Stack innerhalb der Enklave enthält. Die Prototypen veranschaulichen, dass die Kombination aus IEC, sicherer I/O und NPS einen flexiblen und effizienten Schutz sowohl einfacher als auch komplexer Geräte auf RISC-V ermöglicht.

Usage of LLMs

Large Language Models (LLMs) were employed to a limited extent during this thesis. They were solely used for language refinement, such as enhancing readability, grammar and sentence structure. The actual research of this thesis was conducted independently, without reliance on LLMs.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

_____

Lübeck, den December 13, 2025

Acknowledgements

First, I would like to thank my supervisors Till, Claudius, and Chris for the weekly meetings, their valuable feedback, and the continuous support they provided throughout the entire thesis. Especially towards the end, they invested a lot of time and effort, for which I am very grateful. I would also like to thank Prof. Dr. Thomas Eisenbarth for giving me the opportunity to work on this topic.
Finally, I would like to thank my family, who were a great source of support, particularly during the final stage of this thesis.

Contents

1 Introduction

Embedded systems have become an integral part of modern life and are increasingly connected to the Internet. Their applications span a wide spectrum—from convenience-oriented scenarios such as smart homes to safety-critical domains including healthcare, industrial automation, and environmental monitoring. While these systems offer significant benefits, their growing connectivity and the sensitive nature of the data they process make them attractive targets for attackers.

A major subset of embedded systems are Internet of Things (IoT) devices. The demand for IoT technology continues to rise, driven by its ability to simplify everyday tasks—for example, by optimizing energy consumption through coordinated temperature sensors and heating controllers. However, past incidents have shown that IoT devices are frequently insufficiently protected. A prominent example is the Mirai botnet attack in 2016, which compromised hundreds of thousands of devices and leveraged them to launch large-scale DDoS attacks against services such as OVH and Dyn[1]. Many of these devices were only secured with factory-default passwords, allowing attackers to easily gain control. Once compromised, Mirai disabled remote administration interfaces, rendering devices unreachable. In such cases, availability was completely lost, a critical failure for systems that must operate continuously. These highlights that availability is a crucial security property for embedded systems.

To improve the security posture of embedded devices, many modern systems integrate a TEE. TEEs allow sensitive code to execute in isolated environments (enclaves), ensuring that even a fully compromised host OS cannot tamper with enclave applications. Despite extensive research on TEEs, most existing solutions focus primarily on confidentiality and integrity [Wan+22], often leaving availability outside their scope. This observation forms the central motivation for this thesis: we aim to strengthen availability in TEEs, even in scenarios where a remote attacker gains full control over the untrusted host OS.

In RT-TEE, Wang et al. [Wan+22] introduce two components for achieving availability: a secure scheduler and secure I/O. In this thesis we want to focus on the latter. To achieve secure I/O, we want to allow enclaves to exclusively claim I/O devices. This isolates such devices from the untrusted host OS.

Achieving secure I/O, however, is challenging in practice. Many existing TEEs either lack support for exclusive device access or require substantial hardware modifications. In con-

[1] `https://www.malwarebytes.com/de/what-was-the-mirai-botnet` Accessed 2025-12-01

trast, RISC-V offers powerful hardware isolation mechanisms such as PMP.
In this thesis, we build upon a recent case study on secure I/O for Keystone and identified three limitations which motivate the three goals of this work: designing a host-independent IEC, integrating a stronger runtime foundation, and ensuring that the host OS can only access the network through the secure I/O subsystem. Our goals are documented in detail in Section 4.2. With our designs for the goals we improve the isolation between enclaves and the host OS. Our performance evaluation further demonstrates massive performance improvements over Keystone's native OCALL communication.

1.1 Contributions

The contributions of this thesis are summarized below:

Inter-Enclave Communication (IEC) We extend Keystone with an IEC mechanism that allows for host-independent communication between enclaves. We achieved this by porting the Elasticlave memory model into Keystone and extended it with a communication mechanism over shared memory.

Performance Evaluation of IEC We evaluated the performance of the IEC mechanism on the hardware board Starfive VisionFive 2 which revealed massive improvements over communication with native Keystone communication channels.

Extension of Secure I/O Through IEC We extend a Keystone-based secure I/O approach with our IEC mechanism, improving performance and strengthening isolation between the enclaves and the untrusted host OS.

Proof of Concept Implementation for NPS We implemented a PoC for NPS which forces the host OS to schedule the network enclave in order to access network services.

Demo Applications We provide two demo applications protecting both simple and complex I/O devices from interference by the untrusted host OS. Thus, illustratating the feasibility of efficient secure I/O.

2 Preliminaries

This chapter provides the background necessary to understand the design and implementation of secure I/O on RISC-V. It is organized into four parts. We begin by introducing the fundamental concepts required for secure I/O, including a definition of secure I/O, an overview of TEEs, relevant RISC-V security features, and the Keystone TEE framework. The second part presents a case study on secure I/O for RISC-V, which forms the foundation of this thesis. Based on the findings of that study, we derived the three main goals of our work. Next, we discuss IEC and introduce the general concepts that motivate and inform our system design. Thereafter, we present the Elasticlave memory model [Yu+22], which we later incorporate into our final IEC design. The chapter concludes with an introduction to the seL4 microkernel, which we consider as a potential alternative to Keystone's Eyrie runtime in our extended design.

2.1 I/O and Secure I/O

According to Silberschatz et al. [SGG18], many devices expose registers that allow the processor to send commands or exchange data with them. One way to access these registers is through MMIO, where the device registers are mapped into the physical memory address space. This allows the processor to interact with the device using ordinary memory operations such as *read* and *write*.
Some devices, such as NICs, must transfer large amounts of data at once. In these cases, DMA is used to perform the transfer efficiently. The processor first writes the data to a memory buffer and then provides the address of this buffer to the DMA controller, which copies the data directly to the device without further processor involvement. MMIO and DMA are often used together. For example, a NIC is typically configured via MMIO, while packet data is transferred directly into or out of memory using DMA to achieve high performance.
To detect whether a device has data available, two approaches are commonly used. With polling, the processor repeatedly checks a device's status register until new data arrives. Alternatively, devices may rely on interrupts: when data becomes available, the device triggers an interrupt, causing the CPU to execute an interrupt handler before resuming its previous execution. Polling may be advantageous for devices that produce data at very high rates, but interrupts are generally more efficient for most devices as they avoid

unnecessary busy-waiting.

2.1.1 Secure I/O

In the context of TEEs, secure I/O refers to the ability of the TEE to communicate with an I/O device in a manner that cannot be tampered with by untrusted software [SAB15]. The primary goal is to grant enclaves exclusive and protected access to specific devices, thereby preventing the untrusted host operating system from manipulating them. Without such protection, the host OS could modify device registers to alter device behavior or even deny access entirely. In IoT scenarios, such manipulation could corrupt sensor data or, in the worst case, lead to costly outcomes, including hardware damage. Section 3.1 reviews prior approaches to secure I/O and Section 2.5 introduces a secure I/O approach of a case study in more detail.

2.2 Trusted Execution Environments

Sabt et al. [SAB15] provide a general definition of TEEs and describe their fundamental building blocks. They note that the increasing complexity and connectivity of modern systems have driven a shift toward integrating trusted computing capabilities directly into embedded devices. Early solutions relied on TPMs, which offer tamper-resistant hardware components capable of ensuring system integrity and safeguarding cryptographic keys. However, TPMs expose only a limited set of predefined APIs and do not support the isolated execution of third-party applications. As a result, TEEs have emerged as a more flexible alternative, enabling arbitrary applications to run within a protected execution environment. TEEs rely on a separation kernel that partitions the system into isolated components and enforces strict isolation guarantees between them.

2.2.1 Trust

In addition to providing a secure execution environment, a TEE also establishes a foundation of trust. Sabt et al. [SAB15] introduce several key concepts related to trust in such systems. Trust can be classified as either static or dynamic. Static trust evaluates the system's trustworthiness against predefined security requirements and is verified only once, prior to system operation. Dynamic trust, by contrast, relies on the current state of the running system. Because this state evolves over time, the system's trustworthiness must be assessed continuously.
Sabt et al. [SAB15] also define the RoT as a trusted entity that fulfills two primary roles. The first is trusted measurement, which evaluates the integrity of specific system components. The second is the determination of a trust score, a boolean value indicating whether

the system is deemed to be in an acceptable integrity state. To prevent manipulation of either the measurements or the trust score, the RoT is implemented as a well-protected separate hardware component.
Most TEEs employ a form of hybrid trust. Static trust is established before system operation begins, while the RoT performs integrity checks during the boot process to ensure the secure initialization of the TEE. Once bootstrapping is complete, the separation kernel enforces ongoing integrity and isolation during runtime.

2.2.2 Building Blocks

According to Sabt et al. [SAB15], TEEs are composed of five fundamental building blocks:

- *Secure Boot* ensures that only authorized code is executed during system startup. This is achieved through a chain of trust: the integrity of the initial boot code is verified by a protected hardware component, and each subsequent boot stage is validated against the reference measurement of the preceding one. If the integrity of any component is compromised, the boot process is aborted.
- *Secure Scheduling* provides fair and efficient coordination between the TEE and the untrusted system. Its purpose is to prevent either environment from launching a DoS attack.
- *Inter-Environment Communication* establishes the interface through which the TEE and the untrusted system can exchange data. Depending on the design of this interface, communication may be unidirectional or bidirectional.
- *Secure Storage* guarantees confidentiality, integrity and freshness for storage data which can only be accessed by authorized entities.
- *Trusted I/O Path (Secure I/O)* protects the communication between the TEE and I/O devices.

It is important to note that existing TEEs do not always implement all of these building blocks. In particular, *Secure Scheduling* and *Secure I/O* are often omitted, as these components primarily address availability concerns [Wan+22], and availability is frequently considered out of scope for typical TEE threat models.

2.3 RISC-V Security Features

RISC-V [Wat16] is an open instruction set architecture (ISA) introduced by the University of California, Berkeley in 2010. As stated by RISC-V International, “the RISC-V ISA and

extensions ratified by RISC-V International are royalty free and open base building blocks for anyone to build their own solutions and services on" [RIS]. This openness is one of the key motivations for Lee et al. [Lee+20] to adopt RISC-V in their Keystone PoC implementation. Because RISC-V imposes no licensing fees and encourages extensibility, hardware manufacturers can design their own SoCs and introduce custom extensions while remaining fully RISC-V compatible. This flexibility allows Keystone to support a wide range of hardware platforms, as all share the same standardized base ISA.

2.3.1 Privilege Modes

RISC-V provides several architectural features relevant to system security. By design [Wat16], the ISA defines three primary privilege levels: machine mode (M), supervisor mode (S), and user mode (U). M-mode is the highest privilege level and serves as an abstraction of the underlying hardware, offering direct access to physical memory [Wat16]. All interrupts and traps are initially handled in M-mode, which may subsequently forward them to S-mode. In contrast, S-mode operates with virtual memory and is therefore the typical execution environment for operating systems. An additional component that can be implemented in M-mode is the SBI [Gro], which exposes platform-specific functionality through a standardized interface. The SBI enables communication between S-mode and M-mode. Conceptually, SBI calls are similar to system calls, but occur between S-mode and M-mode rather than between U-mode and S-mode.
U-mode, the least privileged level, is intended for executing application code [Wat16]. Its purpose is to protect the operating system in S-mode from potentially malicious or faulty user programs. This design closely parallels the protection-ring model used in x86 architectures [Int24].

2.3.2 Physical Memory Protection (PMP)

Another security feature is PMP [RIS24]. PMP is fully managed by M-mode and enables the isolation of physical memory regions by enforcing fine-grained access permissions. For each protected region, access rights for read, write, and execute operations (r, w, x) can be specified, and these permissions are enforced by the hardware whenever lower-privileged modes attempt to access memory.
A PMP region is defined through a PMP entry, which consists of an address register and an 8-bit configuration register (together forming a PMP register). The address register holds the starting address of the protected region, while the configuration register defines both the size of the region and the corresponding access permissions. Each PMP entry also has an associated priority, where lower-numbered entries take precedence over

higher-numbered ones. RISC-V allows the protection of up to 64 memory regions, though the exact number depends on how many PMP registers are supported by the respective hardware.

2.4 Keystone TEE

Lee et al. [Lee+20] observe that although most major CPU manufacturers provide their own TEEs, these solutions are typically designed for specific trade-offs and offer only limited opportunities for customization. Such restrictions can constrain the design of applications running inside the TEE. Because only CPU vendors can modify the underlying TEE architecture, users must either wait for vendor updates or purchase new hardware to benefit from improved functionality. As an example, Lee et al. point to Intel SGXv1, where enclaves must have static sizes, system calls are not supported, and known side-channel vulnerabilities (such as the page-fault side channel [XCP15]) remain exploitable. Overcoming these limitations often required enlarging the TCB, and many of these constraints were only lifted with the introduction of Intel SGXv2 [Lee+20].
To address these limitations, Lee et al. developed Keystone, an open-source framework that can be customized for the requirements of the applications. Keystone allows the TEE architecture to be adapted to specific TCB requirements and threat models. This flexibility is made possible by leveraging RISC-V security features (see Section 2.3). Because RISC-V provides three privilege modes (see Section 2.3.1), Keystone adopts a modular design: security-critical functionality is handled in M-mode by a SM, resource management is implemented by RTs operating in S-mode, and EAPPs execute in user mode. Figure 2.1 presents an overview of the Keystone architecture. The following subsections describe each component in detail.

2.4.1 Security Monitor

The SM is the most privileged component of Keystone and executes in M-mode [Lee+20]. Its primary responsibility is to enforce the security guarantees of the TEE. Because the SM forms part of the trusted computing base, it is designed to be minimal to reduce its attack surface, while remaining sufficiently extensible. To enforce enclave isolation in hardware, the SM relies on RISC-V's PMP mechanism described in Section 2.3.2.
Figure 2.2 illustrates how Keystone uses PMP to implement secure isolation. During system boot, the first PMP entry (which has the highest priority) is configured to protect the SM's memory region by disabling all access permissions for other components [Lee+20]. The last PMP entry (lowest priority) is set to grant the untrusted operating system full access to the entire physical memory. Due to PMP's priority rules, the OS can access any

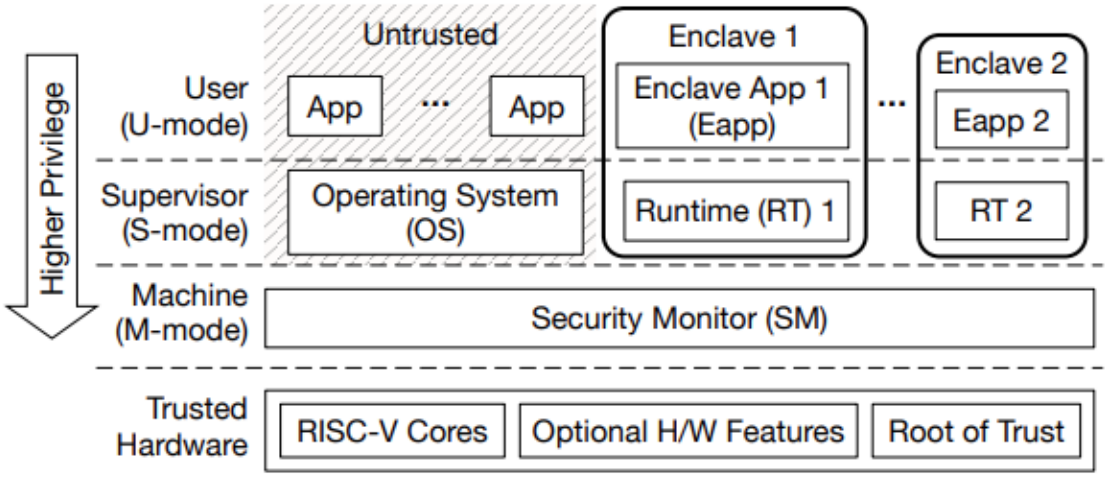


Figure 2.1: Taken from [Lee+20]. Overview of all Keystone components and their corresponding privilege modes.

memory region except those protected by higher-priority entries. To create an enclave, a host application interacts with the host OS, which issues an SBI call to the SM to request the creation of a new enclave. Since the SM does not perform resource management, the OS must first allocate a suitable memory region for the enclave. The SM validates the request and then configures a new PMP entry to protect the enclave memory. This is done by assigning the region to an unused PMP entry and denying all access rights to it. Because enclave PMP entries always have higher priority than the OS entry, the host OS cannot access enclave memory under any circumstances. Before transferring control to the newly created enclave, the SM enables access rights for the enclave's memory region and simultaneously revokes access to memory protected by the OS PMP entry. This ensures that the enclave can access only its own memory region. When the enclave exits, the SM removes its access permissions and re-enables the permissions associated with the OS entry. If the enclave is to be destroyed rather than paused, the SM releases the PMP entry, making the memory available to the OS again.

Keystone also reuses the OS PMP entry during enclave execution to support shared memory buffers between the enclave and the host OS, as described further in Section 2.4.3. It is important to note that each enclave requires a dedicated PMP entry, and therefore a dedicated PMP register to protect its memory region. As a result, a system with N PMP registers can support at most N - 2 simultaneous enclaves.

Keystone [Lee+20] builds on OpenSBI[2] as the foundation for its SM, enabling communication between the SM executing in M-mode and both the runtime and the host operating

[2] `https://github.com/riscv-software-src/opensbi` Accessed: 12-10-2025

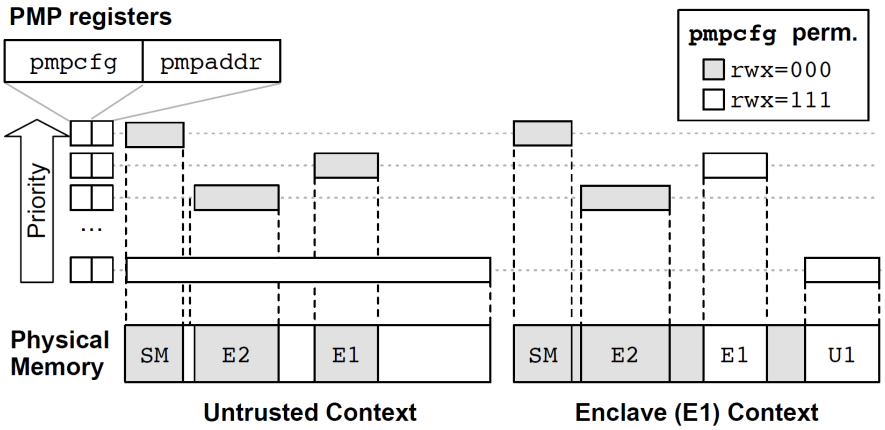


Figure 2.2: Taken from [Lee+20]. Overview of PMP usage in Keystone.

system running in S-mode. All interrupts and exceptions first arrive in M-mode, and therefore at the SM [Lee+20]. The SM may delegate specific interrupts and exceptions to S-mode using the corresponding delegation registers, `mideleg` and `medeleg` [RIS24]. To protect the host OS from potential DoS attacks by enclaves, the SM programs a machine timer before transferring control to an enclave. When the timer interrupt fires, it is handled first in M-mode, allowing the SM to periodically regain control to protect the host OS from DoS attacks.

Keystone provides several security primitives by default: Secure Boot, a Secure Source of Randomness, and Remote Attestation [Lee+20]. Secure Boot establishes a chain of trust (see Section 2.2.1 and Section 2.2.2), rooted in the RoT. However, Keystone currently simulates this process because the RoT is hardware-dependent. Supporting a new hardware platform would therefore require implementing the RoT in a zeroth-stage bootloader with access to device-specific secrets. Secure Randomness can be requested from the SM via an SBI call and is used, among other purposes, to derive attestation keys. Remote Attestation allows an external verifier to confirm that a genuine enclave is executing and that its initial state has not been tampered with. The SM performs both the measurement and the attestation process.

Another building block supported by Keystone is Inter-Environment Communication, described in more detail in Section 2.4.3. However, Keystone does not support availability primitives from the building blocks defined in Section 2.2.2, as it lacks a *Secure Scheduler* and *Secure I/O*.

2.4.2 Enclave Privilege Modes

Another key component of Keystone is the runtime [Lee+20]. Each enclave is restricted to its own isolated memory region and is internally divided into two parts: the EAPP (U-mode) and the RT (s-mode). The runtime is responsible for managing enclave resources, handling virtual memory, and providing enclave-specific system functionality. Conceptually, it resembles a lightweight operating-system kernel. Keystone's default runtime is Eyrie, a highly modular and minimal design that allows developers to include only the components needed for their specific enclave. This modularity enables tailoring the enclave's TCB to its requirements. For example, an EAPP may choose to rely on system calls delegated to the host OS or use standard libraries such as libc. In addition to communicating with the SM, the runtime mediates communication with the host OS through an untrusted shared memory buffer. In this design, the relationship between the EAPP and the runtime parallels the familiar division between user space and kernel space in traditional operating systems such as Linux or Windows.

2.4.3 Inter-Environment Communication (OCALLs)

Keystone implements Inter-Environment Communication (see Section 2.2.2) through an edge-call interface [Lee+20]. Each enclave is restricted to its own isolated memory region, and therefore communication with the host OS must occur via OCALLs, which are implemented through the edge-call interface using an untrusted shared memory buffer (see Figure 2.2). During enclave creation, the host OS allocates this shared memory region and passes its physical address to the SM. The SM forwards the address to the runtime, and when control is transferred to the enclave, the SM reuses the OS PMP entry to grant both the enclave and the host OS controlled access to only this shared buffer. This ensures that shared memory is accessible for communication, while all other enclave memory remains protected. OCALLs operate similarly to rpcs: each OCALL includes an index identifying the handler function in the host application, along with additional parameters as needed. When an EAPP issues an OCALL, the runtime intercepts the corresponding system call and, upon recognizing it as an OCALL, writes the OCALL identifier and its parameters into the shared memory buffer. The runtime then requests an enclave exit via an SBI call to the SM, specifying that the exit was triggered by an OCALL. After the enclave is stopped, execution resumes in the Keystone driver, which extracts the reason for the exit and returns it to the host application. Because the host application has the shared memory region mapped, it can immediately read the OCALL parameters. At enclave initialization, the host application registers multiple OCALL handlers, each identified by an ID shared between the enclave and host, which allows it to dispatch incoming OCALLs. Once the

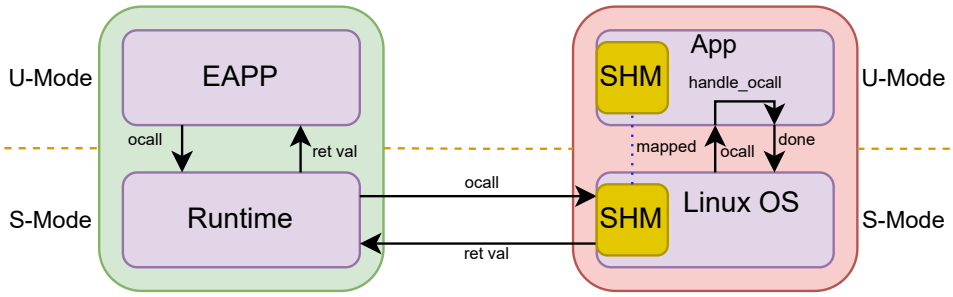


Figure 2.3: Taken from the report of the case study. Overview of OCALL routes that need to be taken to reach the host OS/host app and to get back to the issuing enclave.

host application processes the OCALL and writes the return values back into the shared buffer, it requests that the enclave be resumed, again via the Keystone driver and an SBI call. The enclave continues execution with access to the updated shared memory contents. An overview of an OCALL round trip between the EAPP, host OS, and host application is illustrated in Figure 2.3. More details about the general enclave lifecycle are provided in Section 2.4.4.

2.4.4 Enclave Lifecycle

Terminology for Call Types in Keystone

In this thesis, several types of calls occur across different privilege levels and software layers in Keystone. For clarity, we use the following terminology:

- **ocall**: Invocation from an EAPP (U-mode) to the untrusted host OS/host app (S-mode/U-mode).
- **icall**: Invocation from an EAPP (U-mode) to a other EAPP (U-mode).
- **ioctl call**: Invocation from the host app (U-mode) to the host OS Keystone driver (S-mode).
- **sbi call**: Invocation from S-mode to M-mode (via SBI).
- **ecall**: Invocation from a lower privilege level to a higher privilige level.
- **runtime syscall**: Invocation from an EAPP (U-mode) to the Keystone runtime (S-mode).

Having introduced the Keystone components and their interactions (see Section 2.4), we

now turn to the processes involved in the Keystone enclave lifecycle. An additional component relevant here is the Keystone driver within the host operating system. This driver is responsible for issuing SBI calls from S-mode to the SM, as such calls cannot be invoked directly from user space. Moreover, because the SM does not perform memory management, the driver is also tasked with allocating the memory required for each enclave.

Note

The statement that SBI calls can be issued only from S-mode is not entirely accurate. SBI calls are implemented as simple wrappers around the RISC-V `ecall` instruction, and `ecall` is executable from U-mode as well. Consequently, it is technically possible to invoke an SBI call directly from U-mode, since all traps initially arrive in M-mode. However, in most RISC-V systems that support S-mode, M-mode is configured to delegate traps originating from U-mode to S-mode. This delegation is essential for maintaining proper privilege separation and ensuring that user applications cannot directly access machine-level functionality. An advantageous feature of the RISC-V architecture is that both U-mode and S-mode system calls use the same *ecall* instruction. This design reduces the number of required instructions and contributes to the overall simplicity and efficiency of the architecture.

To make use of SBI calls, we briefly introduce the SM interface that both the Keystone driver and enclaves rely on to access M-mode functionality. The Keystone driver can invoke the following SBI calls: *sbi_create_enclave*, *sbi_run_enclave*, *sbi_resume_enclave*, and *sbi_destroy_enclave*. Because SBI calls cannot be made directly from a host application running in U-mode, the driver exposes each SM operation through a dedicated `ioctl`. This enables the host application to create, run, resume, and destroy enclaves. When an enclave is interrupted (for example due to an OCALL) the host application handles the request and then resumes the enclave via the resume `ioctl`.
Enclaves themselves use *sbi_stop_enclave* and *sbi_exit_enclave* to interact with the SM. As with the driver and host application, the runtime exposes these SM functions to the EAPP through runtime syscalls. As with the driver and host application, the runtime exposes these SM functions to the EAPP through lightweight runtime syscalls. The stop syscall is used when the enclave must temporarily yield control (e.g., to perform an OCALL), whereas the exit syscall signals that the enclave has completed execution and control should return to the host.
With the SM interfaces established, we now provide an overview of the Keystone enclave lifecycle, illustrated in Figure 2.4. For clarity, the figure omits certain initialization steps, such as the details surrounding *sbi_create_enclave*. As shown, lifecycle execution begins when the host application prepares the enclave by configuring the size of the memory re-

gions to be used for the untrusted shared buffer and for the enclave memory. The driver allocates the necessary memory and passes the base address and size to the SM, which protects the memory region using PMP. Both the driver and the SM assign a unique enclave identifier. Once initialization is complete, the driver returns to the host application.
To start the enclave, the host application issues a run `ioctl`. The driver identifies the enclave and calls *sbi_run_enclave*. Inside the SM, the host context is saved and control is transferred to the enclave by performing a context switch. The runtime then boots and eventually transfers execution to the EAPP's entry point. When the EAPP finishes, it issues a runtime syscall to exit the enclave. The runtime forwards this request using *sbi_exit_enclave*, prompting the SM to restore the previously saved host context and return execution to the driver. Control continues in the run `ioctl`, which then returns to the host application, completing the lifecycle.
Two SBI calls, *sbi_stop_enclave* and *sbi_resume_enclave*, are not depicted in Figure 2.4, as the figure focuses on the basic lifecycle. These calls are essential when the enclave must interact with the host, such as during an OCALL (see Section 2.4.3). To perform an OCALL, the EAPP issues a runtime syscall, which the runtime handles by invoking *sbi_stop_enclave* and indicating that the stop was triggered by an OCALL. The SM switches context to the host OS, passing along the stop reason. The driver forwards this information to the host application, which then processes the OCALL by reading the request from shared memory. After completing the OCALL, the host application resumes enclave execution via a resume `ioctl`, which results in *sbi_resume_enclave*. Execution inside the enclave continues exactly where it was suspended earlier.

2.4.5 Enclave Execution

After developing the EAPP and constructing a corresponding host application, the enclave can be compiled. This compilation process produces an enclave package, which can be executed either in QEMU or on supported hardware platforms. The package consists of four components: the host application, the Eyrie runtime, the EAPP, and a trusted loader. Because Keystone enclaves support in-enclave ELF loading, the loader must be included within the enclave package and is treated as a trusted component. When the enclave package is executed, its contents are unpacked, and the host application is launched with the remaining components passed as arguments. The host application determines the amount of memory required to hold the provided ELF binaries and allocates the corresponding physical memory through the Keystone driver. The loader is placed at the beginning of this memory region so that it executes first when the enclave starts. Directly after the loader, the runtime ELF is loaded, followed by the EAPP ELF. Finally, the physical base address of the enclave memory region and its total size are communicated to the

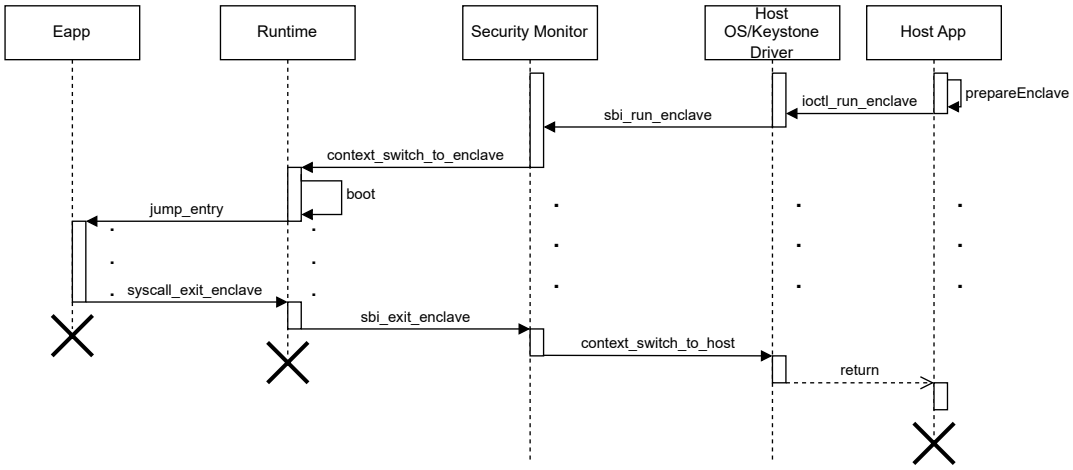


Figure 2.4: Derived from a source-code analysis of Keystone. Overview of the enclave lifecycle in Keystone. Both the Keystone Driver and the Security Monitor are never destroyed while the system operates.

SM, which then configures PMP to protect the enclave's memory region.

Note

In-enclave ELF loading allows an enclave to load both the EAPP and runtime ELF files autonomously. To support this functionality, a trusted loader must be embedded within the enclave. The loader is responsible for retrieving and loading both ELF files from the enclave's memory region.

Once the enclave has been prepared, the host application initiates its execution via the *ioctl_run_enclave* function. A detailed recap of the subsequent steps is provided in Section 2.4.4, which outlines the enclave lifecycle.
Because the loader is placed at the beginning of the enclave's physical memory region, it is the first component executed after the context switch into the enclave. During this initial switch, the SM writes the physical base addresses of the enclave regions into registers `a0-a7`, enabling the loader to retrieve these values during execution. Within Keystone, the runtime is responsible for managing the enclave's own root page table. To support this, the loader allocates memory for the root page table and maps the runtime ELF into it. This mapping step is necessary because the loader executes in physical addressing mode, whereas the runtime uses virtual addressing. The loader therefore prepares the root page table on behalf of the runtime by defining the virtual address ranges to map, including the virtual entry point of the runtime. After mapping the runtime, the loader maps the entire enclave physical memory region as well, ensuring that the runtime can subsequently ac-

cess and manage all enclave memory. At this point, the root page table is fully prepared. Next, the loader writes the predefined virtual entry point of the runtime into the `stvec` register. Although `stvec` typically holds the address of a trap handler (for example to handle page faults) Keystone repurposes this mechanism to trigger the first transfer of execution into the runtime. Before proceeding, the loader also restores the parameters originally passed to it by rewriting them from the stack into registers `a0-a7`, ensuring that the runtime receives these arguments during its booting. Finally, the loader switches the enclave into virtual addressing mode by writing the address of the prepared root page table into the `satp` register.

2.5 Case Study: Secure I/O on RISC-V

We now present a detailed summary of the case study "Secure I/O on RISC-V" conducted by Burmester and von Plata at the Institute of IT-Security, University of Lübeck, in 2024. This summary serves to introduce the case-study design, which we later use to derive our own thesis goals (see Section 4.2) and to motivate improvements over the existing approach.

The case study builds upon a Keystone pull request for secure I/O[3], which provides support for exclusively assigning a serial I/O device to an enclave. The case study extends this mechanism to handle more complex devices, including NICs. The core objective is to instruct the SM which I/O devices should belong exclusively to the secure world. These devices are referred to as Secure Devices. Device identification is performed via entries in the device tree. A device is marked as secure by adding a `secure-status` flag set to `okay`, while simultaneously setting its standard `status` flag to `disabled`. The latter prevents the untrusted host operating system from using the device. Using the device tree for this purpose is advantageous, as it can be protected from tampering through mechanisms such as secure boot.

During system boot, the SM parses the device tree to extract all Secure Devices. Keystone's design reserves one PMP region with the highest priority to protect the SM itself, and one PMP region with the lowest priority to protect the entire address space for the host OS. To protect Secure Devices, the SM configures an additional PMP entry for each secure I/O device, thereby securing the full MMIO region of that device. An enclave that claims a Secure Device can then map the device's MMIO region and access its registers. Conversely, the host OS loses the ability to access those MMIO registers, since they are now protected by PMP.

To maintain a clean separation of concerns, the case study introduces a dedicated driver

[3] `https://github.com/keystone-enclave/keystone/pull/418` Accessed: 2025-04-19

enclave. This enclave is responsible for claiming and managing the device, and for providing services to a client enclave. Client enclaves request I/O operations by communicating with the driver enclave. Using this mechanism, the case study demonstrates that simple devices (such as serial devices) can be securely claimed by a driver enclave, enabling the client enclave to interact with them indirectly.
The case study further extends this design to support significantly more complex devices. To demonstrate the general feasibility of secure I/O even for complex peripherals, the case study implements a prototype that exclusively assigns a NIC to a network driver enclave and uses it to establish a secure communication channel. A overview of this architecture is presented in Figure 2.5. In this design, a network driver enclave claims full control over the NIC and functions as a server for other client enclaves. To increase isolation from the host OS, both the NIC driver and the network stack are moved into the network enclave (into its runtime in S-mode). As a result, the enclave application can issue runtime syscalls to send and receive packets. When the system starts, the network enclave initializes its network stack and NIC driver. Client enclaves can then send packet payloads to the network enclave, which forwards them to the NIC driver. The network stack interfaces directly with the NIC driver to transmit the packet. Reception follows the same path in reverse.
The prototype was successfully implemented on a RISC-V hardware board, the Starfive VisionFive 2. To preserve network connectivity for the host operating system, one of the board's two NIC ports was assigned to the host OS, while the other was claimed exclusively by the network driver enclave.
To support enclave to enclave communication, the case study relies on Keystone's OCALL mechanism (see Section 2.4.3). As illustrated in Figure 2.5, both enclaves interact indirectly through the host application. This communication pattern is implemented using blocking OCALLs: the network enclave performs a blocking OCALL and waits until the client enclave notifies it that an enclave call should be processed. If the client enclave requires a return value, it similarly performs a blocking OCALL and waits for a notification from the network enclave indicating that the request has been handled. These notifications are likewise implemented using OCALLs. In this design, the completion of a blocking OCALL signals that the corresponding enclave call is processed. The return path is implemented by issuing a simple OCALL that informs the host application to resume the other enclave's previously blocked OCALL.
Because Keystone does not natively support direct ICALLs, the case study simulates ICALL like behavior using sequences of blocking OCALLs in exactly the manner described above.

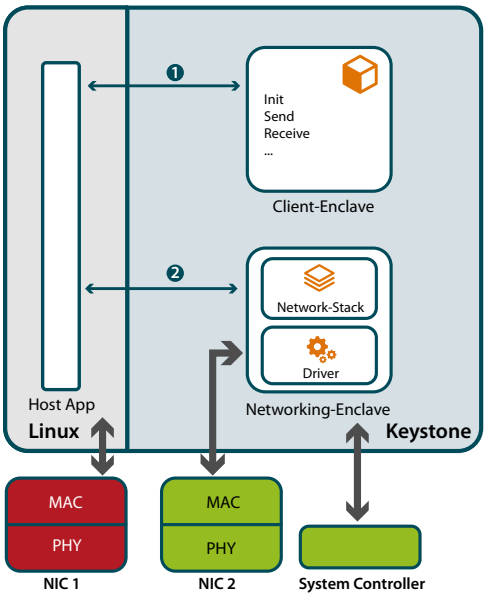


Figure 2.5: Taken from the report of the case study. Design Overview of secure I/O approach for a NIC. The client enclave and network enclave communicate over the host OS.

2.6 Inter-Enclave Communication

In addition to inter-environment communication (see Section 2.4.3) between the untrusted system and the TEE, enclaves may also communicate directly with one another through IEC. Most TEEs adopt the spatial isolation model [Yu+22], in which an enclave's virtual address space is divided into public and private memory regions. This model is effective for enforcing isolation, but it also restricts enclaves to accessing only their own private memory. Communication with other enclaves (or with the host OS) must occur through public memory regions. Cerberus [Lee+22] refers to this model as "No Sharing", highlighting the fact that memory is not directly shared between enclaves.
Yu et al. [Yu+22] point out that enabling memory sharing under spatial isolation requires both a trusted coordinator enclave and a secure channel. Without these, shared memory approaches would be vulnerable to attacks. Keystone [Lee+20] is an example of a TEE that follows the spatial isolation model. As described in Section 2.5, Keystone can simulate IEC using sequences of blocking OCALLs mediated through an untrusted shared buffer. However, Yu et al. observe that such mechanisms introduce substantial performance overhead due to expensive data copying, repeated context switching, and the encryption and decryption required to maintain channel security. Because of these limitations, a variety

of alternative approaches to IEC have been proposed. We review these designs in more detail in Section 3.2.

2.7 Elasticlave Memory Model

Elasticlave [Yu+22] introduces a shared memory model that enables efficient and flexible communication between enclaves. It provides three key features. First, each enclave maintains a local view of its permissions for a shared region. This allows, for example, one enclave to have write access while another has read access to the same region. Second, enclaves may dynamically adjust their permissions for a shared region, provided that they do not exceed the maximum permissions assigned by the region's *Owner* at the time of sharing. Third, Elasticlave supports lock-based synchronization: enclaves can acquire exclusive locks on shared regions, and these locks can be transferred atomically to other enclaves. Elasticlave enforces these locks globally, ensuring that only the enclave holding the lock can access the region.
Elasticlave also defines two roles for shared regions: *Owner* and *Accessor*. The *Owner* is responsible for creating a shared memory region and for *sharing* it with other enclaves. *Owners* may *destroy* a region at any time. An *Accessor* is permitted to use a region only if explicitly authorized by the *Owner*. An enclave may simultaneously act as both *Owner* and *Accessor*, depending on the region and context.
Elasticlave exposes an interface for enclaves, summarized in Table 2.1. Each shared region is uniquely identified by a *uid*. An *Owner* creates a shared region by specifying its size and receives the corresponding *uid*. This is done with the *create* function. The *uid* can then be used to *map* the region into the enclave's virtual address space or to *unmap* it. Once a region is mapped, the *Owner* implicitly becomes an *Accessor* as well. To grant another enclave access to a region, the *Owner* issues a *share* operation, specifying the target enclave's *eid*, the *uid* of the region, and the maximum permissions that the *Accessor* may hold. *Accessors* may subsequently *change* their permissions, but only within the bounds established by the *Owner*. These permission changes apply locally within each enclave. The only exception is the lock permission: when set, it grants exclusive global access to the region. Locks can be released by removing the lock permission via another *change* operation, or they can be transferred directly to another enclave using the *transfer* operation, which takes the *uid* and the *eid* of the next lock holder. When a shared region is no longer needed, enclaves may *unmap* it, and the *Owner* may *destroy* the region by specifying its *uid*.

Instruction	Permitted Caller	Semantics
`uid = create(size)`	owner of `uid`	create a region
`err = map(vaddr, uid)`	accessor of `uid`	map VA range to a region
`err = unmap(vaddr, uid)`	accessor of `uid`	remove region mapping
`err = share(uid, eid, P)`	owner of `uid`	share region with an enclave
`err = change(uid, P)`	accessor of `uid`	adjust the actual access permission to a memory region
`err = destroy(uid)`	owner of `uid`	destroy a memory region
`err = transfer(uid, eid)`	current lock holder	transfer lock to another accessor

Table 2.1: Taken from [Yu+22]. Overview of the Elasticlave interface.

2.8 Microkernel seL4

The seL4 microkernel is part of the L4 family of microkernels, whose development dates back to 1996 [Hei25]. In contrast to monolithic operating systems such as Linux or Windows, seL4 follows a microkernel architecture that aims to keep the kernel extremely small, containing only the minimal mechanisms required for an operating system. Components such as drivers, network stacks, and file systems run as user-mode applications, fully isolated from one another yet able to communicate through secure ipc mechanisms. This design ensures that a vulnerability in one component , for example a device driver, does not compromise the entire system.
seL4 offers two distinguishing features. First, it is the world's first microkernel with a complete formal verification at the source-code level [Hei25], providing mathematical proof that the implementation conforms to its specification. Beyond the verified source code, the compiled binary has also been shown to be free of certain classes of bugs, extending the verification chain down to the executable level. Second, seL4 includes formal proofs that it enforces the classical security properties of confidentiality, integrity, and availability. This combination of functional correctness and verified security properties makes seL4 one of the most trustworthy operating-system kernels available.
In addition to its strong security guarantees, seL4 is also the fastest microkernel currently known [Hei25], making it suitable for RTOS scenarios. The microkernel further supports virtualization, enabling guest operating systems such as Linux to run in virtual machines. This capability allows existing Linux drivers to be reused within seL4-based systems with minimal integration effort.

3 Related Work

This chapter surveys related work in three areas relevant to the goals of this thesis. We begin in Section 3.1 with an overview of existing approaches to secure I/O, as this topic aligns most directly with our primary objective. The subsequent sections broaden the scope by discussing additional research that informs the remaining goals of our work, including IEC and NPS.

3.1 Secure I/O

Although Sabt et al. [SAB15] identify secure I/O as a fundamental building block of a TEE, many existing TEEs do not yet provide a comprehensive mechanism for implementing it. This includes well-established systems such as Intel SGX. To understand the current landscape and derive insights for our own design, we review notable approaches to secure I/O.

In ARM TrustZone, it is possible to isolate I/O devices from the untrusted world, but these devices cannot be assigned exclusively to individual enclaves [Bah+21]. Similarly, ARM-based TEEs such as OP-TEE[4], do not yet support secure access to many essential I/O devices. While protecting an I/O device's MMIO region can be achieved by mapping it into the secure world, doing so prevents the rich execution environment (REE) from using its own drivers. Consequently, the REE drivers must be ported into the TEE, a process that enlarges the TCB and is technically challenging. A naïve alternative is to reuse the original REE driver within the TEE, but these drivers are typically large, multi-purpose, and thus significantly increase the TCB. A possible compromise is to manually trim the driver so that it only retains the minimal functionality required to operate the device. However, as Guo et al. [GL22] demonstrate, this trimming process is labor-intensive, error-prone, and requires substantial trial and error. To address this, Guo et al. [GL22] propose an automated solution. Their system records the interactions of an existing driver with a device and generates an interaction template, stored in a driverlet. A trustlet within the TEE can then replay these interactions at runtime. While promising, this approach supports only simple I/O devices. Complex peripherals such as NICs are unsuitable because their behavior depends on the dynamic contents of network packets.

An alternative to record-and-replay is LDR [Yan+24], which pursues similar goals but

[4] `https://github.com/OP-TEE/optee_os` Accessed 2025-12-01

with improved performance and security. Yan et al. [Yan+24] observe that most Linux drivers depend on kernel subsystem functions and library routines that TEE operating systems typically do not provide. LDR therefore reuses existing TEE library functions when possible and forwards remaining calls to the untrusted world. Drivers in LDR execute inside a sandbox protected by ARM domain access control. Unlike driverlets, LDR runs real drivers in the secure world but splits them into two cooperating components: an SW driver inside the TEE and an NW driver outside it. The SW driver is derived by removing the initialization, configuration, and VFS layers from the original Linux driver. At runtime, the SW and NW drivers collaborate to emulate full driver functionality.

Wang et al. [Wan+22] similarly note that migrating full driver logic into the TEE runtime results in an excessively large TCB. Their system, RT-TEE, adopts concepts reminiscent of LDR. They introduce an I/O reference monitor that provides sandboxing and driver debloating. Drivers are split into two components: a feature-rich but untrusted instance that runs in a sandbox, and a minimal trusted instance that uses interaction templates, similar to [GL22]. The reference monitor verifies both spatial and temporal access control: it ensures that processes access only authorized devices and only for a bounded duration. For simple I/O devices, the minimal instance is sufficient. For more complex devices such as NICs, the feature-rich instance is used.

A fundamentally different approach is taken by Cure, proposed by Bahmani et al. [Bah+21], which introduces hardware extensions to provide enclaves with exclusive access to I/O devices. Each enclave is assigned a unique identifier, and CPU cores are extended with an additional register that stores the ID of the currently executing enclave. A new bus-protocol signal propagates this ID throughout the system. The arbiters for both main memory and the peripheral bus are modified to enforce access control based on enclave ID. The SM configures these arbiters to associate MMIO regions with specific enclaves. As a result, enclaves gain exclusive control over assigned devices through hardware-level enforcement.

Schneider et al. [Sch+21] pursue an approach similar to the case study discussed in Section 2.5. Their objective is to provide exclusive access to I/O devices by composing multiple enclaves. They differentiate between unit enclaves and composite enclaves which combine several unit enclaves. To protect an I/O device, a dedicated driver enclave is created, and the device's MMIO region is mapped exclusively into this enclave, enforced through PMP isolation. Communication between the driver enclave and the device is direct via the MMIO region, i.e. without overhead. Communication between the enclaves is implemented by shared memory. Approaches for such communication are presented in the following section.

3.2 Inter-Enclave Communication Approaches

Inter-Enclave Communication addresses the same fundamental challenge as inter-process communication, but in the context of communication between enclaves rather than processes. Prior work on IPC, such as [VJ15], shows that shared memory provides the lowest overhead and highest performance among IPC mechanisms. For this reason, we now examine existing approaches that use shared memory to enable communication between enclaves.

In many current TEEs, including Intel SGX and RISC-V Keystone [Lee+20], IEC can only be implemented by relying on the untrusted host operating system. Communication is performed through public memory and typically requires encryption, decryption, and repeated data copying [Yu+22]. As a result, not only must the untrusted OS be involved, but communication incurs substantially higher overhead than necessary.

Schneider et al. [Sch+21] address IEC by allocating, in addition to private memory, a shared memory region between pairs of unit enclaves. The SM manages all memory regions and uses enclave identifiers to determine which enclaves may access each shared region. Both communicating enclaves receive shared ownership of the region, enabling direct data exchange.

Yu et al. [Yu+22] extend this idea by introducing a new shared-memory abstraction, Elasticlave, which provides protection against malicious enclaves and eliminates unnecessary data copies. Elasticlave satisfies two key properties: First, the owner of a shared region must explicitly authorize each enclave that may access it. Second, for applications with predefined sequences of memory accesses, all non-malicious enclaves are guaranteed to execute these sequences correctly, even in the presence of malicious enclaves. These guarantees ensure consistency and security. Elasticlave implements them by granting enclaves exclusive access rights to a region, which can be transferred to other enclaves once an operation is complete. Because access rights must be explicitly granted, malicious enclaves cannot obtain them.

Cerberus [Lee+22] provides a formal treatment of enclave memory sharing. The authors define four sharing models. In No Sharing, enclaves remain fully isolated. In Arbitrary Sharing, enclaves may freely access each other's memory. In Capped Sharing, enclaves may access at most k shared regions and Single Sharing is a special case where k=1. Cerberus formally proves that the Single Sharing model preserves confidentiality and integrity, but only for immutable shared regions. Extending this proof to mutable regions is significantly more complex because access rights must change dynamically to prevent interference.

Dep-TEE [Pan+25] also relies on shared memory for IEC but decouples memory protec-

tion from address translation, thereby improving scalability and performance. Dep-TEE achieves this separation using RISC-V's Supervisor PMP (SPMP) mechanism. Memory isolation is enforced in a manner similar to Keystone's use of PMP but implemented via SPMP registers. For IEC, each enclave receives a `shm_key` that links the owner of a shared region to an enclave requiring access. A unique `enclave_key` is used to authenticate enclaves and to transfer ownership of shared regions. Ownership transfer is performed by the SM in M-mode through reconfiguration of SPMP registers.

3.3 Trusted Networking Approaches

Schwarz [Sch22] introduces TrustedGateway, a gateway architecture designed to isolate core networking functionality, referred to as NetTrug, from the untrusted host operating system. The system leverages ARM TrustZone to assign the NIC exclusively to the secure world, with isolation enforced entirely through hardware. Prior to booting the host OS, the NIC is fully removed from the OS's control, ensuring that only the secure world can interact with it. Within the secure world, a minimal I/O framework is used to operate the NIC, thereby keeping the TCB small. Similar to the twin-driver strategy employed in LDR [Yan+24], TrustedGateway splits the NIC driver into two components: a trusted driver running in the secure world and an untrusted driver running in the normal world. The trusted driver performs all security-critical I/O operations, while the untrusted driver handles only non-critical configuration tasks, such as power management. Because the host OS no longer has access to the NIC's MMIO registers, TrustedGateway provides the host with a virtio-based virtual NIC to maintain ordinary network functionality. All traffic from the virtual NIC is forwarded to the secure world, where NetTrug inspects and filters packets to prevent network attacks originating from the host OS, such as spoofing.

4 Problem Statement

This chapter begins by outlining the main motivation for our work. We then present the goals of the thesis and explain the rationale behind each of them. Finally, we define the threat model that underpins the assumptions and security considerations of our design.

4.1 Motivation

Most TEEs provide guarantees of confidentiality and integrity, while explicitly treating availability as out of scope. This is true even for widely deployed systems such as Intel SGX. In such TEEs, the untrusted operating system retains full control over device drivers and device MMIO registers. Consequently, the secure world must rely entirely on the untrusted world to interact with I/O devices. In scenarios where enclaves need to communicate with peripheral devices (for example when monitoring sensors) the untrusted OS can tamper with device registers, manipulate driver behavior, or deny access altogether. Such interference effectively undermines the purpose of using enclaves in the first place. Furthermore, communication between the secure and unsecure world typically depends on public, untrusted memory. This often necessitates encrypting and decrypting all exchanged data, introducing non-negligible performance overhead.
Wang et al. [Wan+22] identify two properties required to extend TEEs with availability guarantees: a secure scheduler and secure I/O. In this thesis, we focus on secure I/O, introduced earlier in Section 2.1.1 and recognized as one of the fundamental building blocks of TEEs (see Section 2.2.2). While several secure I/O approaches exist (see Section 3.1), most rely on ARM-based platforms or require hardware modifications.
Because RISC-V provides strong security primitives, particularly PMP, we build upon the results of the case study presented in Section 2.5, which already explores a PMP-based secure I/O mechanism. However, we identified three key limitations in the case study design. These limitations form the basis for the three goals of this thesis.

4.2 Goals

Our general objective is to provide secure I/O, enabling the secure world to operate with minimal dependence on the untrusted host operating system. The case study design discussed in Section 2.5 does not meet this requirement. We identified three key limitations,

which we address through the following goals:
As our general goal is secure I/O, we want to ensure that the secure world can operate mostly independent from the unsecure world. The case study design (see Section 2.5) doesn't achieve such host independence. There are three issues with the design, which we address with the following goals:

1. **(G1) Host-Independent Inter-Enclave Communication:** provide a host-independent way for enclaves to communicate with each other inside the secure world of Keystone.

2. **(G2) Using seL4 as a new Runtime:** replace the default Eyrie runtime in Keystone with the more capable, formally verified seL4 microkernel.

3. **(G3) Network Port Sharing:** force the host OS to access the network through the same NIC port that is controlled by the secure world.

To motivate these goals, we now summarize the issues identified in the case study design described in Section 2.5.
The primary limitation of the case study architecture lies in its implementation of IEC through multiple OCALLs. As described in Section 2.4.3, OCALLs fully rely on untrusted public memory shared with the host OS. This not only violates our requirement for host independence but also introduces significant performance overhead. Each OCALL incurs several context switches and chaining multiple OCALLs, as required for simulated IEC, results in substantial cumulative latency. Goal **(G1)** directly addresses both challenges by eliminating host dependence and drastically reducing the number of context switches required for enclave to enclave communication.
A second limitation arises from the restricted capabilities of the Eyrie runtime running in S-mode. To maintain a small TCB, Eyrie implements only minimal functionality. In the case study, this leads to dependencies on the host OS for configuring certain NIC registers, preventing the enclave from fully controlling the device. To overcome these restrictions, Goal **(G2)** proposes replacing Eyrie with the seL4 microkernel. seL4 provides a richer set of OS features while remaining minimal and formally verified, making it a suitable alternative. For more details on seL4 and its benefits refer to Section 2.8.
The final limitation concerns the handling of NIC ports. In the case study, two NIC ports were used. One assigned exclusively to the network enclave and one to the host OS. While this appears to offer separation, it does not provide full isolation. An attacker still has incentives and opportunities to disrupt secure-world networking services. On the one hand, the host OS (or an attacker with OS-level privileges) can tamper with NIC configuration parameters that affect the secure world. On the other hand, a DoS attack on the "secure"

NIC port remains feasible, allowing the attacker to disable secure network services. In addition, many hardware boards provide only a single NIC port, making the two-port design impractical for general use. Goal **(G3)** addresses these issues by enforcing NPS, requiring the host OS to use the exact same NIC port as the secure world. Under this design, any attempt by the host OS or a remote attacker to disrupt the secure port would simultaneously remove its own network access, thereby eliminating the incentive to attack.
In addition to these three primary goals, we implement two PoC applications, described in more detail in Section 6.4. The first demonstrates secure I/O combined with our IEC mechanism: a server enclave exclusively claims a serial device in the secure world, while a client enclave accesses this device by communicating with the server enclave through IEC. The second showcases secure I/O in combination with NPS and IEC. Here, a NIC is exclusively assigned to the network enclave in the secure world. A client enclave can transmit packets to the Internet via this enclave. Due to NPS the host OS is also required to route its network traffic through the same secure-world NIC.

4.3 Threat Model

We assume a strong software-only attacker who can't tamper with the physical hardware. In the enclave context, such an attacker can have full control of the untrusted host OS and launch malicious enclaves. We assume the attackers goal is to bring the device under their control and persist their control, this might include shutting down remote management and tampering with I/O devices. In addition, we further assume the system runs on an embedded device with remote management, where the attacker wants to gain exclusive device access by restricting access for management channels, i.e., peripheral devices. In such a case, the device operator could no longer receive monitoring information or cannot send updates to the device for administration purposes.
We further assume the implementation of the TEE and the underlying hardware features to be bug-free.

5 Design

This chapter presents the design proposals developed to address each of the thesis goals outlined in Section 4.2. For each goal, we first introduce the selected design and then discuss alternative approaches that were considered during the design process. The chapter is structured into three sections, each corresponding to one of the goals and detailing the rationale behind our choices.

5.1 Host-Independent Inter-Enclave Communication

Our first goal, **(G1)** (see Section 4.2), is to design and implement a host-independent IEC mechanism. Instead of developing such a security-sensitive component from scratch, and given that this thesis pursues three distinct goals, we chose to investigate existing enclave memory models that have already undergone security evaluation and could be integrated into Keystone.
As discussed in Section 3.2, shared memory approaches consistently provide the best performance for IPC and therefore also for IEC. During our survey of related work, we identified several promising shared-memory models for enclaves, which we compared in detail in Section 3.2. Our secure I/O architecture assumes one driver enclave serving multiple client enclaves, and in this context we found Elasticlave [Yu+22] to offer strong flexibility together with robust security guarantees. For this reason, Elasticlave aligns best with the requirements of our target design.

> **Note**
>
> Throughout the remainder of this thesis, the term shared memory region refers to a region established between two enclaves using the Elasticlave interface.

We now describe the final IEC design pursued in this thesis, illustrated in Figure 5.1. The goal is to integrate the Elasticlave memory model into Keystone, enabling the creation of shared memory regions between enclaves. These regions can be directly mapped into each enclave's EAPP in U-mode, allowing enclaves to exchange data without requiring syscalls to the runtime or OCALLs to the host OS. This achieves both host independence and minimal communication overhead.
Having presented our final IEC design, we now discuss an alternative memory model

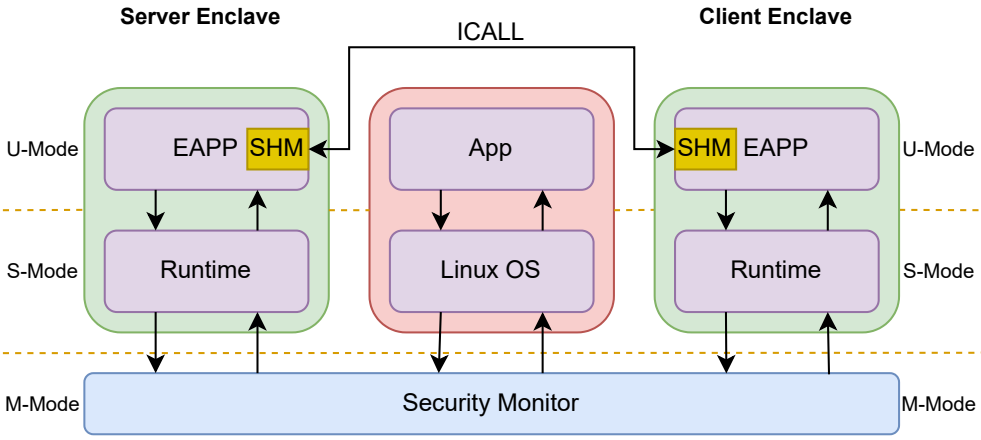


Figure 5.1: Design Overview for IEC. SHM stands for shared memory and is created by the server enclave over the SM. The same region is mapped by the client enclave, so both enclaves can communicate over the shared memory by performing ICALLs. There is no need for interaction with the host OS while performing ICALLs.

that we evaluated and explain why we ultimately decided against adopting it. As noted earlier, Elasticlave [Yu+22] introduces dynamic state transitions, which complicate future formal verification. Motivated by this concern, we also examined the Cerberus memory model [Lee+22].

Cerberus employs a single-sharing abstraction in which one enclave hosts a shared region that other enclaves may access. At first glance, this model aligns well with our architecture of a single driver enclave serving multiple client enclaves. However, Cerberus restricts the shared region to read-only access for the other enclaves. This restriction was introduced to simplify formal verification, particularly with respect to confidentiality and integrity guarantees. While suitable for certain narrowly scoped scenarios, such a read-only sharing model is insufficient for general-purpose IEC. Many use cases require enclaves to exchange data bidirectionally, and secure I/O scenarios frequently depend on write-capable communication channels. The limitation is especially problematic in a network scenario, where NIC operations commonly require enclaves to send and receive data through the same shared region.

Because Cerberus's restrictions prevent its use in a flexible, write-capable IEC mechanism, we chose instead to implement the Elasticlave memory model, whose expressiveness and capability for bidirectional sharing better support our secure I/O design goals.

5.2 Using seL4 as Alternative Runtime

Our second goal, **(G2)** (see Section 4.2), is to overcome the limitations of the Eyrie runtime by replacing it with the seL4 microkernel. Lee et al. [Lee+20] have previously demonstrated that seL4 can be used as a Keystone runtime. However, their prototype is outdated and supports only earlier versions of both seL4 and Keystone. Because our secure I/O design requires the current Keystone codebase, achieving this goal necessitates a new port of seL4 to Keystone.

Figure 5.2 illustrates the resulting system architecture when seL4 is used as the enclave runtime. As a microkernel, seL4 provides only fundamental OS services. Higher-level components—such as complex device drivers—must therefore execute as user-level applications in U-mode. These applications rely on seL4 for essential primitives, such as access to timers, and communicate with one another via seL4's built-in IPC mechanisms. The figure focuses on the driver enclave, since a client enclave does not require an extended runtime and can continue to operate using the minimal Eyrie runtime. Importantly, the IEC mechanism designed in Section 5.1 is a prerequisite for integrating seL4: even with seL4 as the runtime, enclaves must still communicate securely and efficiently, particularly between the driver enclave and its client enclaves.

Although the design in Figure 5.2 is presented in a general form, one of our demonstration applications requires exclusive control of a NIC. Consequently, the runtime must also incorporate a NIC driver together with a suitable network stack.

There are two promising approaches for supporting a NIC driver within the seL4-based runtime. First, we could reuse the existing Linux NIC driver by executing it inside a virtual machine running Linux as a guest OS on top of seL4 (VM approach). This method is relatively straightforward and highly general, but the additional virtualization layers inevitably introduce performance overhead. Second, we could develop a native seL4 NIC driver using the well-documented seL4 driver frameworks (native approach). This approach is more technically demanding but offers significantly better performance. It is therefore the preferred option.

Unlike Eyrie, seL4 already provides access to established network stacks such as lwIP and picoTCP. During our investigation, we identified an example from the SDDF driver framework[5] that demonstrates the combined use of a network driver, the lwIP stack, and an application. The structure closely resembles the architecture shown in Figure 5.2, with the only addition being a network stack positioned between the driver and the application. This example confirms the feasibility of implementing a network enclave on seL4 and shows that, aside from the seL4 port and the NIC driver, the essential components for

[5] `https://github.com/au-ts/sddf/tree/main/examples/echo_server` Accessed: 12-10-2025

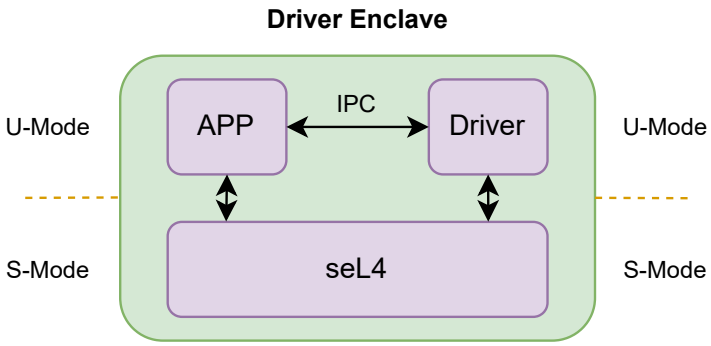


Figure 5.2: Design idea for a seL4 based runtime for the driver enclave. The apps in U-mode that run on top of seL4 can communicate through the seL4 IPC mechanism.

evaluation on real hardware are already available.

5.3 Network Port Sharing

Our third and final goal, **(G3)** (see Section 4.2), seeks to remove the dependency on hardware platforms equipped with two network ports and to ensure that the host OS must schedule the network enclave in order to obtain network connectivity. This idea is conceptually related to an aspect of TrustedGateway [Sch22], where network services are partially moved into the secure world and exposed to the host OS through a virtual network interface controlled by the trusted domain. In contrast, our approach places the entire NIC driver and network stack within the secure world (i.e., inside the network enclave), enabling complete isolation from the host OS. To reintroduce network connectivity for the host OS, we create a virtual network interface that is fully controlled by the network enclave.

Linux provides tun/tap virtual network devices[6], which are well suited for this purpose. Since Linux serves as the host OS in Keystone, we can leverage these devices to provide a virtual interface to the host. A tun/tap device behaves similar to a NIC: packets written to the device are delivered to a userspace application rather than sent over the physical wire, and incoming packets are injected into the Linux network stack as if they originated from a physical NIC. Tun devices operate on IP packets, while tap devices operate on raw Ethernet frames.

Figure 5.3 illustrates our final design for NPS. As a host app needs to schedule the net-

[6] `https://www.kernel.org/doc/Documentation/networking/tuntap.txt` Accessed: 2025-12-01

work enclave to get network access, we decided to let the same app create the tun/tap device. Any host-side process, including the Linux kernel itself, may then send or receive packets via this virtual interface. Once a packet is processed by the Linux network stack, it is written to the tun/tap device, which forwards it to the host application instead of transmitting it over the wire. The host application buffers these packets, and the EAPP in the network enclave periodically retrieves them. The enclave can then transmit the packets using the NIC driver. This design does not require the enclave to process packets through its internal network stack when using a tap device, since Ethernet frames can be forwarded directly. Likewise, for incoming packets, the EAPP receives them from the NIC driver and forwards them to the host application, which injects them into Linux via the tun/tap device. From Linux's perspective, these packets appear indistinguishable from packets received on a physical NIC.

By delegating the creation and management of the tun/tap device to the host OS, we avoid increasing the TCB of the network enclave. At the same time, the host OS is now forced to schedule the network enclave whenever it requires network access, because the tun/tap device is the only network interface available to it. This fully eliminates the dependency on a second NIC port and significantly improves the isolation between the secure and unsecure world.

We also evaluated an alternative design in which the network enclave itself exposes a virtual network interface to the host OS. This would require integrating a virtio backend inside the enclave, similar to QEMU's use of tun/tap devices for virtual networking. However, implementing such functionality directly within the enclave would significantly increase its TCB. For this reason, we instead opted to let the host OS manage the tun/tap device while the enclave retains exclusive control over the physical NIC.

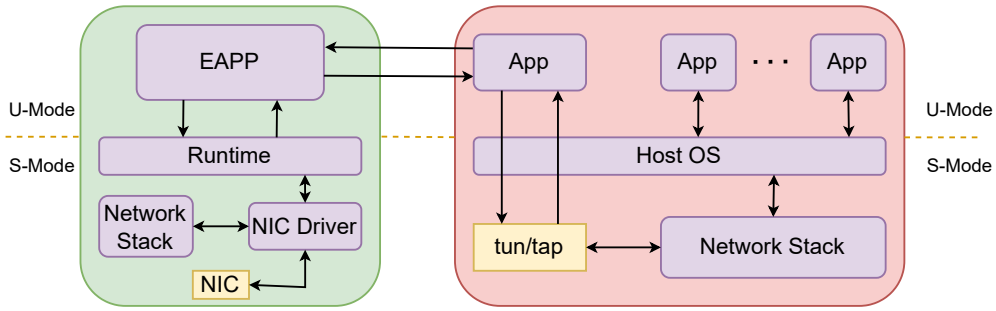


Figure 5.3: Usage of a tun/tap device in Linux to achieve NPS. One host app creates the device. When the Linux OS or other apps send packets, the tun/tap device receives the packets and forwards them to the host app that created it. The EAPP can then collect new packets from that host app to send them out. Receiving works the other way around.

6 Implementation

Building on the design objectives and conceptual foundations outlined in the previous chapter, this chapter details the implementation of those designs and the challenges encountered along the way. We begin with the implementation of the IEC mechanism, which represents the core contribution toward improving isolation between the secure and unsecure world. We then describe the issues faced during the integration of seL4 as an alternative runtime. This is followed by the implementation of the NPS mechanism. Finally, to demonstrate how all components interact in practice, the chapter concludes with two demonstration applications.

6.1 Inter Enclave Communication

The implementation of our IEC design (see Section 5.1) began with an in-depth analysis of both the Keystone enclave lifecycle and the Elasticlave memory model [Yu+22]. A detailed examination of the enclave lifecycle is provided in Section 2.4.4, while Section 2.7 summarizes the key mechanisms of the Elasticlave model.
As described in Section 2.4.4, Keystone assumes a static memory model: all enclave memory regions must be allocated and finalized prior to enclave execution. This design conflicts with our IEC goals, which require the ability to create and manage shared memory regions dynamically at runtime. In the following subsections, we describe how we integrated Elasticlave into Keystone and outline the required modifications to Keystone's architecture in order to support Elasticlave's dynamic shared memory capabilities.

6.1.1 Integrate Elasticlave into Keystone

The Elasticlave memory model (introduced in Section 2.7) was originally implemented for Keystone[7] by Yu et al. [Yu+22]. However, this implementation targets an older Keystone version and is no longer compatible with the current codebase. Since the secure I/O approach described in Section 2.5 is only supported by the most recent Keystone version, we adopted the original Elasticlave implementation as a reference and ported the complete model to the latest Keystone release. In the following, we summarize the key modifications required to support the Elasticlave memory model within Keystone.

[7] `https://github.com/jasonyu1996/elasticlave` Accessed: 12-10-2025

To expose the Elasticlave interface (see Section 2.7) to enclaves, Keystone must support dynamic creation and protection of shared memory regions. Because these regions require isolation properties equivalent to enclave memory, they must be secured using PMP registers. Since PMP configuration is restricted to M-mode, all Elasticlave operations that affect PMP must communicate with the SM. This communication is implemented using SBI calls. As discussed in Section 5.1, our goal is to allow eapps to directly access shared regions in U-mode, without relying on the runtime or host OS. However, because PMP protections can only be managed by the SM, the EAPP must invoke Elasticlave operations through runtime system calls, which then delegate to the SM.

Keystone employs a three-level privilege architecture. Consequently, all EAPP system calls first enter the runtime in S-mode. The first step in our integration was therefore to extend the runtime's system-call interface with wrappers for all Elasticlave functions. Each wrapper corresponds to a runtime syscall and internally performs an *ecall* instruction [RIS24], using registers `a0-a7` to transfer arguments and return values. Two exceptions arise: the *map* and *unmap* function. The Elasticlave interface expects the caller to provide the virtual address at which the shared region should be mapped. However, Keystone assigns all enclave memory management responsibilities to the runtime, not the EAPP. Therefore, we adapted these functions to take only the region's *uid* as argument and to return the virtual address of the region instead of an error code.

For the Elasticlave operations *create*, *change*, *share*, *transfer*, and *destroy*, the runtime acts purely as a proxy: it forwards the arguments it receives through a corresponding SBI call to the SM, where the actual logic is executed. In contrast, the *map* and *unmap* operations require new functionality within the runtime itself. Keystone's original runtime does not support dynamic mapping of physical memory into an enclave's virtual address space. Because the runtime is responsible for managing the enclave's root page table, we extended its memory management subsystem to support on-demand mapping and unmapping of shared regions. This enables enclaves to incorporate Elasticlave regions dynamically into their virtual address spaces, as required by our IEC design.

Similar to runtime system calls, SBI calls are also implemented using the *ecall* instruction. However, when the *ecall* is issued from S-mode, execution traps into M-mode, transferring control to the SM. The SM includes an *ecall* handler responsible for processing Elasticlave SBI calls, among others.

When a *sbi_elasticlave_create* call arrives from the runtime, the SM must initiate the creation of a new shared region. At this point, however, a fundamental limitation of Keystone becomes apparent. As outlined in Section 2.4.4, Keystone follows a static memory model, meaning that enclave memory regions are fixed at creation time. The SM itself does not perform memory management, as its TCB must remain minimal. Consequently, enclaves

cannot be assigned new memory regions at runtime using the standard Keystone mechanisms. To address this limitation, we follow the strategy used in the original Elasticlave implementation. Although the SM cannot allocate memory directly, it can communicate indirectly with the Keystone driver by leveraging the enclave interruption mechanism. Specifically, the SM can use the *sbi_stop_enclave* path, normally triggered when an enclave performs an OCALL, to signal a request to the driver. Figure 6.1 illustrates this communication flow. When the SM receives a *sbi_elasticlave_create* call, it invokes *stop_enclave* internally and sets a elasticlave interrupt reason. The requested region size is stored in register `a1`, while the interrupt reason is stored in register `a0`. After the enclave's execution has been suspended, control returns to the driver at the point immediately following its previous *sbi_resume_enclave* call. Upon re-entering the driver, the request is processed within the *process_sm_request* function. The driver reads the requested size from register `a1`, allocates a suitable physical memory region, and communicates the base address of this newly allocated region back to the SM through a subsequent *sbi_resume_enclave* call.
Before control returns to the enclave, the SM finalizes the creation of the shared region by configuring a new PMP entry to protect the allocated physical memory region and assigning a *uid* to the new region. Finally, the SM responds to the original *sbi_elasticlave_create* call by returning the uid to the runtime, which in turn forwards it to the EAPP that initiated the creation request.
A further challenge arises from Keystone's static memory model in relation to IPI handling. During system boot, the SM is initialized on every hart, and each hart maintains its own independent set of PMP registers. To ensure consistent protection guarantees, all PMP updates must be synchronized across all harts. Keystone accomplishes this synchronization using IPIs, but by default, IPIs are only issued during enclave creation and destruction. These IPIs propagate PMP changes system-wide so that no hart can inadvertently bypass memory-protection settings. Because Elasticlave introduces additional PMP regions for shared memory, we extended Keystone to ensure that an IPI is dispatched to all harts whenever a new shared region PMP entry is created. Without this modification, shared-region protections would not be consistently enforced across cores.
When the SM receives an *sbi_elasticlave_map* or *sbi_elasticlave_unmap* call, it resolves the physical address of the shared region using the provided *uid*. The physical address is then returned to the runtime, which is responsible for performing the actual mapping or unmapping in the enclave's virtual address space.
Upon receiving a *sbi_elasticlave_share* call, the SM first verifies that the calling enclave is indeed the *Owner* of the shared region. If the check succeeds, the SM associates the region's *uid* with the specified enclave *eid*, thereby granting that enclave the role of *Accessor* for the region.

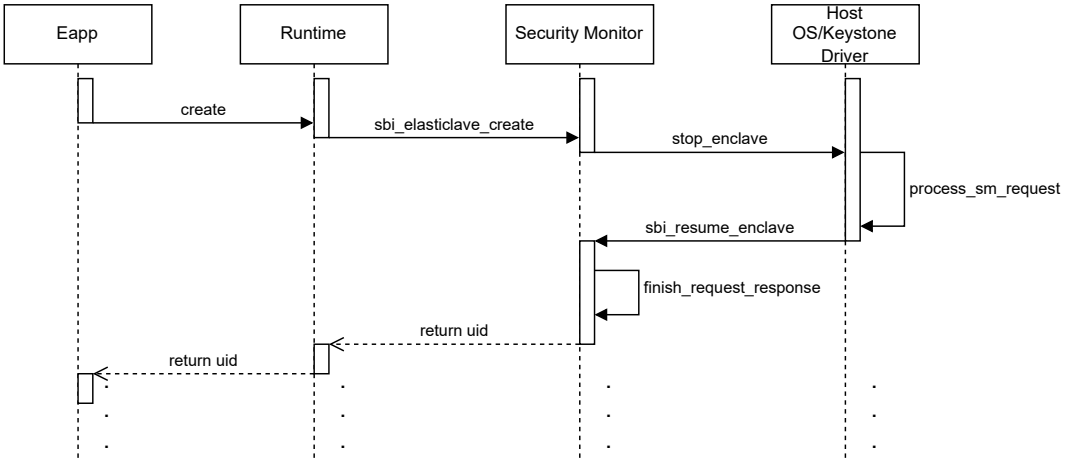


Figure 6.1: Overview of the Elasticlave region creation process, including the communication flow between the Security Monitor and the Keystone driver for allocating a new shared memory region. For clarity, the figure assumes that the enclave is already running.

The *sbi_elasticlave_transfer* call transfers a lock from the current lock holder to another enclave. The SM verifies that the calling enclave currently holds the lock and that the target enclave is permitted to receive a lock. If both checks succeed, the lock holder is updated to the target enclave's *eid*.

The *sbi_elasticlave_change* call is used to adjust a shared region's permissions dynamically. Permissions consist of combinations of (R)ead, (W)rite, (X)execute, and (L)ock bits. For RWX changes, no global update is required because PMP permission updates affect only the enclave's own hart. Locking, however, is different: a lock must be globally enforced so that no other enclave running on another hart can access the region concurrently. When an enclave requests a lock by setting the L bit, the SM checks whether the region is already locked by another enclave and whether the caller is permitted to obtain a lock. If the request is valid, the SM designates the caller as the lock holder and propagates the necessary PMP updates via IPIs to all harts currently running enclaves that are Accessors of the region. Releasing a lock similarly requires a global update so that other enclaves can subsequently acquire the lock.

For *sbi_elasticlave_destroy*, the SM first checks whether the calling enclave is the region's *Owner*. Only Owners may delete shared regions. If the check succeeds, the SM removes the PMP entry corresponding to that shared region.

6.1.2 Final Inter-Enclave Communication

The implementation of the Elasticlave model [Yu+22] in Keystone provides the foundation for our IEC mechanism. Various communication models could be layered on top of the shared region, but given that our secure I/O design follows a client–server paradigm, where a server enclave exclusively controls an I/O device and client enclaves access it through the server, we adopt the same paradigm for the IEC mechanism.
A core challenge arises from the fact that Elasticlave specifies the semantics of shared regions but does not define how two enclaves initially coordinate the creation and exchange of such a region. As described in Section 2.7, a shared region is created by an *Owner* using the *create* call, which returns a *uid*. The *Owner* can then map the region into its virtual address space using *map*. However, an *Accessor* must know the relevant *uid* in order to map the same region, and it must be granted permission through a preceding *share* call. Because shared regions are created dynamically at runtime, no static setup (where regions and their *uid*s are pre-established) is feasible. Thus, explicit synchronization between the *Owner* and all *Accessors* is unavoidable.
To resolve this, we use a small number of blocking ocalls during the initial setup phase to coordinate the first-time configuration of a shared region between the server and the client. This host involvement is temporary: ocalls are required only until both enclaves have mapped the region. After this point, all further communication takes place exclusively through the shared region, independent of the host.
A summary of the final IEC design is shown in Figure 6.2. The server enclave registers the ICALL handlers that clients may invoke. These ICALLs are implemented as RPC operations over the shared region, reusing Keystone's edge-call interface inside the eapps. The server signals that it is ready to accept client connections and exposes its *eid*. The client determines its own *eid* and requests the server's *eid*. Once obtained, the client contacts the server and communicates its own *eid*. The server creates the shared region using *create*, then maps it using *map*. Because an Elasticlave region initially has no permissions, the server enables its own read/write access via *change*. The server then shares the region with the client using *share*, granting maximum read/write permissions. The server sends the region's *uid* to the client. The client maps the region using *map* and updates its local rights to read and write using *change*.
Once both enclaves have mapped the shared region and configured their permissions, the client can invoke ICALLs directly over the shared buffer. From this point onward, communication is fully host-independent.

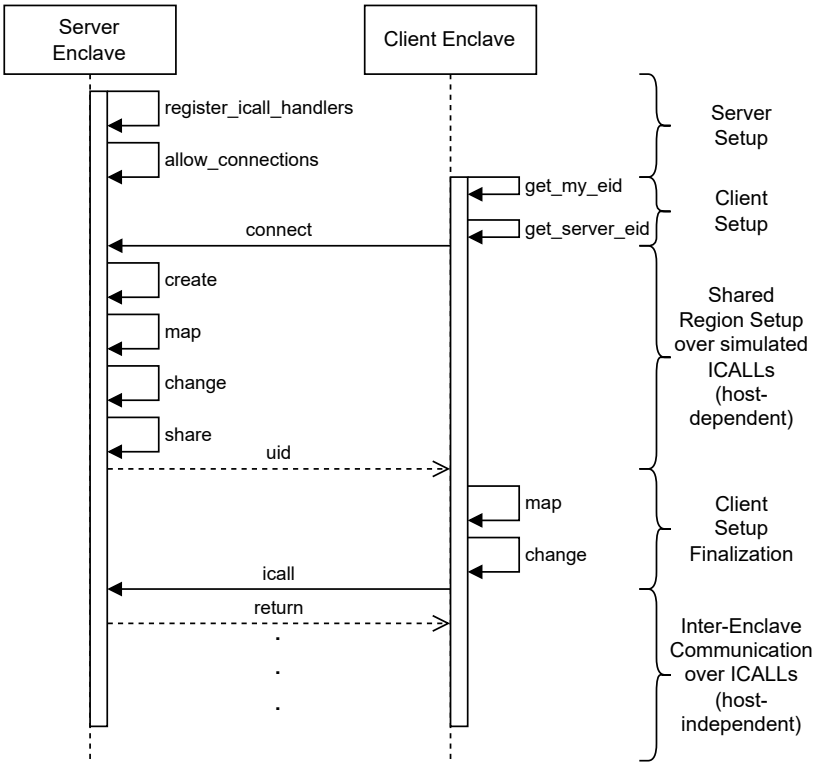


Figure 6.2: Overview of the IEC mechanism. The ICALLs for the initial setup of the shared regions are implemented via blocking OCALLs. After the initial setup, the ICALLs are then used completely host-independently by using an RPC mechanism (from the client server model) via the shared region. In order to achieve a more simplified representation, the following components were excluded from the figure: SM, runtime, host app and host OS.

6.2 Challenges for Porting seL4 to Keystone Runtime

A potential design for integrating seL4 as a runtime was presented in Section 5.2. As noted there, this design presupposes a functional port of seL4 to the Keystone runtime environment. The following sections describe, in detail, the challenges we encountered for this port.
Keystone employs three privilege modes (see Section 2.3.1) relevant for enclave execution. Moreover, enclave execution depends on a host application that prepares, configures, and launches the enclave. Replacing the Eyrie runtime with seL4 therefore requires a precise understanding of these execution processes and of the interactions between the host application, the runtime, and the SM. A detailed description of these processes is provided in Section 2.4.5.

6.2.1 Booting seL4

To enable the runtime to boot seL4 instead of Eyrie, adjustments were required in both the build system and the boot sequence. As a first validation step, we relied on the publicly available seL4 test application[8], which provides a comprehensive suite of kernel-level tests and executes in U-mode. Our objective was to use this application to determine whether the ported seL4 kernel could be successfully launched as a Keystone runtime. For this purpose, the enclave package was adapted such that the EAPP ELF shipped with Keystone was replaced by the seL4 test application ELF, and the default Eyrie runtime ELF was replaced by the seL4 kernel ELF.
To support this change, the boot process required further modification. In Keystone, the loader writes a specific virtual entry point for the runtime into the `stvec` register. However, this entry point differs from the one expected by seL4. We therefore adjusted it to redirect execution to the correct seL4 entry point.
Additionally, seL4 expects a different set of boot parameters than those provided by the Keystone loader in registers `a0-a7`. Because Keystone's host OS allocates enclave memory at runtime, the physical base addresses of both the runtime and the EAPP vary between launches and must be communicated to the kernel at boot time. To accommodate this, we modified the seL4 kernel so that its *init_kernel* function uses the boot parameters provided by the loader.
These modifications collectively ensured that seL4 successfully boots inside a Keystone enclave and that control reaches the *init_kernel* function as expected.

[8] `https://github.com/seL4/sel4test/tree/master/apps/sel4test-driver` Accessed: 12-10-2025

6.2.2 Booting Issues

Despite the successful initial boot of seL4, more complex issues emerged during subsequent stages of kernel initialization. Extensive debugging efforts did not yield a workable solution, which ultimately led us to seek advice from seL4 developers. Their feedback revealed two fundamental limitations of seL4 that conflict with Keystone's execution model: (1) seL4 does not support dynamic memory layouts on non-x86 architectures, and (2) the kernel requires a statically known physical ELF load address, fixed at build time. Due to these architectural constraints, and given time limitations, we concluded that completing the full seL4 integration was infeasible. Instead, we implemented an alternative plan (see Section 6.2.3) to ensure that the remaining components of our work (IEC and NPS) could still be demonstrated. In the following, we describe the encountered issues in detail.
While kernel initialization, seL4 constructs and activates its own root page table and assumes that the entire physical memory of the platform is available. Although the loader-generated root page table is sufficient to reach the start of seL4 kernel initialization, seL4 needs a new page table for internal setup. Since seL4 assumes it has access to the entire physical memory, PMP violations may occur during the execution of enclaves. The reason is, that the host application allocates only a small contiguous memory region for the enclave, and the SM protects exactly this region using PMP. Consequently, the kernel must be restricted to the enclave memory defined by *dram_base* and *dram_size*. To prevent PMP faults, we modified seL4's kernel window mapping accordingly. Figure 6.3 illustrates the relevant memory layout. On RISC-V (64-bit), seL4 uses a 39-bit virtual address space, yielding 512 GiB of addressable space. This space is evenly divided into 256 GiB for user space (high addresses) and 256 GiB for kernel space (low addresses). The kernel space is directly mapped into the virtual address space using *map_kernel_window*. The layout reserves 1 GiB for kernel devices, 1 GiB for the kernel ELF, and maps the remaining 254 GiB as the kernel window. To restrict this mapping to the enclave, we adjusted the physical mapping range from 0 to *dram_base* and from *PADDR_TOP* to *dram_base+dram_size*. The virtual boundary was modified from *PPTR_TOP* to *PPTR_BASE+dram_size*.
The blocking issue concerned the physical placement of the seL4 kernel ELF. In Keystone, the physical ELF address is determined dynamically at enclave setup time. In contrast, seL4 requires the kernel ELF's physical address to be fixed at build time and hard-coded into the kernel. This mismatch means that, once seL4 installs its newly generated root page table during *map_kernel_window*, the kernel ELF is no longer reachable at the expected physical address, causing an immediate crash. We attempted to work around this limitation by modifying Keystone so that the kernel ELF is always placed at a constant physical address within the enclave, and by embedding this address into seL4. However, the kernel still crashed when switching to the new page table. The crash behavior

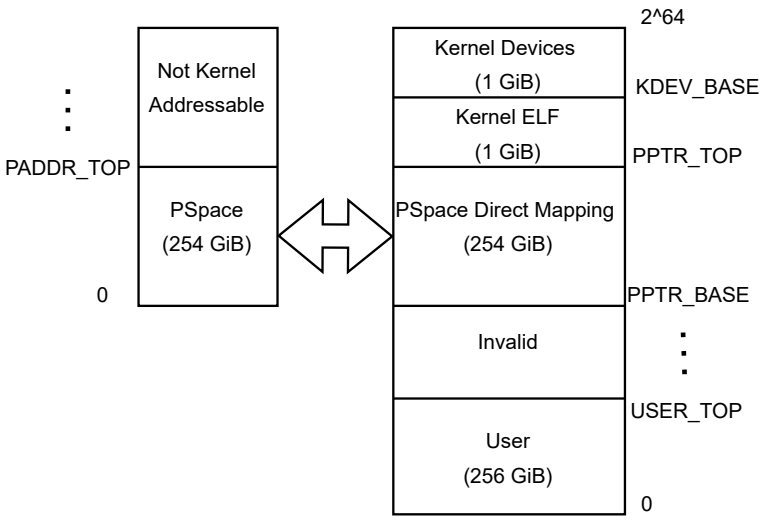


Figure 6.3: Memory layout of seL4 when using 64bit RISC-V and SV39 address translation for the page tables.

indicated that hard-coding the kernel ELF address is fundamentally incompatible with dynamically allocated enclave memory, confirming the limitation identified by the seL4 developers.

6.2.3 Alternative Approach

Given the challenges encountered during the seL4 integration, we adopted an alternative strategy based on Keystone's existing Eyrie runtime. The IEC mechanism described in Section 6.1 had already been implemented and tested for Eyrie, making it a suitable foundation for continued development. Moreover, the secure I/O implementation from the case study (see Section 2.5) could be reused with only small amount of modification. Integrating both components, our IEC mechanism and the secure I/O approach, proved straightforward and resulted in the expected functionality without significant technical obstacles. However, in Section 6.4, we describe some challenges encountered when attempting to utilize IEC and secure I/O with two enclaves.

6.3 Network Port Sharing

As outlined in Section 5.3, our NPS design relies on the use of a Linux tun/tap device to route all host-side network traffic through the network enclave. In the following, we describe the implementation of our PoC. We first address the host-side setup and subsequently the communication between the host app and the EAPP.
To create a tun/tap device, the host application first opens the device */dev/net/tun*, obtaining a file descriptor for configuring the virtual interface. An `ioctl` call on this descriptor is then used to assign a name to the new tun/tap interface, after which Linux creates the device accordingly. The host application proceeds by configuring the device with an IP address and activating it. Normally host apps could now specify the ip address of the device for their sockets, to send/receive packets to/from the device. However, our goal is to ensure that all network traffic from the host OS is routed through the tun/tap device. Linux routes packets based on its routing table, which contains a default route used for all traffic not matched by more specific routing entries. Each route is associated with a network interface responsible for handling traffic along that route. To enforce that the host OS exclusively uses the tun/tap interface, we delete the existing default route and create a new one that references our tun/tap device. From this point onward, all outbound and inbound traffic on the host OS passes through the host application, which can now receive or inject packets on behalf of the OS.
The next step is to enable communication between the network enclave and the host application. While the IEC mechanism from Section 6.1 was initially considered, it is designed for communication exclusively within the secure world. Extending Elasticlave [Yu+22] to support the untrusted host OS would require substantial modifications. The SM would need to maintain PMP regions for the host OS even during enclave execution, which would increase the TCB and involves SM design changes. In addition, the Keystone driver would require extensions to expose an Elasticlave interface to the host app in U-mode. Therefore, for our PoC, we decided to reuse ocalls (see Section 2.4.3). Ocalls naturally support communication from an enclave to its host application, and we only require non-blocking ocalls, which reduces the OCALL overhead.
In the final PoC, the host application creates and configures the tun/tap device as described above. The EAPP in the network enclave issues ocalls to request packets received by the tun/tap device. The host application acts as a proxy: it reads packets from the tun/tap interface and forwards them to the EAPP. After receiving a packet, the EAPP can pass it to the NIC driver inside the enclave, which ultimately transmits the packet over the physical network interface.
We successfully implemented this PoC on the RISC-V Starfive VisionFive 2 board. To val-

idate the design, we executed the modified host application and generated traffic (e.g., ICMP echo requests) in a separate process. The host application correctly intercepted these packets via the tun/tap device, and the EAPP was able to process and forward them to the NIC driver for transmission.

6.4 Putting it All Together: Demo Applications

This section demonstrates the applicability of our secure I/O approach across a spectrum of device types, from simple peripherals to complex hardware,demonstrating that the combined design supports a broad range of use cases. A further objective is to show that the secure I/O design from Section 2.5 can be seamlessly integrated with both the new IEC mechanism (Section 6.1) and the NPS mechanism (Section 6.3). To this end, we present two demonstration applications, detailed in Section 6.4.1 and Section 6.4.3.
As described in Section 6.1.2, the initial establishment of a shared memory region between enclaves requires several OCALLs. A practical challenge arises because OCALLs rely on each enclave's own untrusted shared buffer with the host OS. If two enclaves run in separate host processes, an ipc mechanism would be required to exchange the OCALL related data stored in these distinct buffers. To avoid this additional complexity, our implementation uses a single host application that launches both enclaves, assigning each enclave to a separate thread.
At the end of Section 2.5, we described how ICALLs can be emulated using sequences of blocking OCALLs. To forward the OCALL parameters from one enclave's thread to the other, the contents of the untrusted buffer of the first enclave must be copied into the untrusted buffer of the second enclave. Since the Keystone SDK does not natively support this functionality, we extended the SDK accordingly. Synchronization between blocking OCALLs is handled using mutexes.

6.4.1 First Demo: Secure I/O for Serial Device

The first demonstration combines the secure I/O mechanism from Section 2.5 with the IEC design introduced in Section 6.1.2. Its primary objective is to show that even a simple peripheral (here a serial device) can be protected and accessed securely through our approach. In this demo, a server enclave exclusively claims a serial device provided by QEMU. After claiming the device, the server enclave creates a shared memory region using our IEC mechanism. Once the shared region is established, both the server and client enclaves can communicate directly through ICALLs, without involving the host OS. When the client enclave wishes to write data to the serial device, it sends the message to the server enclave via an ICALL. The server enclave then performs the actual device write

operation on behalf of the client.

6.4.2 PMP Limitations

Contrary to initial expectations, the first demonstration revealed an unexpected shortage of available PMP registers. To address this issue, we conducted a detailed analysis of PMP register usage and identified inefficiencies in Keystone's PMP management.
On QEMU, Keystone supports a total of 8 PMP registers, meaning that at most 8 PMP regions can be protected simultaneously. Our inspection showed that, for both the server and client enclaves, Keystone consumed three PMP registers instead of the expected two. The reason is that Keystone establishes a TOR PMP region for regions with an arbitrary range, which applies to the enclave memory region. This results in the usage of two PMP registers for that region instead of only 1 PMP register. As described in Section 2.4.5, the Keystone driver allocates enclave memory without enforcing size constraints. Because arbitrary region sizes are permitted, these regions rarely satisfy the conditions required for more compact PMP encoding schemes. However, RISC-V provides an alternative PMP address-matching mode NAPOT (Naturally Aligned Power of Two), which allows a region to be protected using a single PMP register. NAPOT regions must satisfy two conditions: The memory region size must be a power of two and the base address must be aligned to that size. To force NAPOT for enclave memory regions, we modified Keystone's memory allocation process. Specifically, we adjusted the host application's calculation of enclave memory sizes to always round up to a power-of-two value. Additionally, we modified the Keystone driver to ensure that the resulting memory base address is aligned accordingly. These changes allowed the SM to configure enclave memory as NAPOT regions and thereby reduce each enclave's PMP consumption from 3 registers to 2 registers again.
With these adjustments, the first demonstration could be executed successfully. However, further analysis showed that the system still used all eight PMP registers. This aligns with the SM's behavior described in Section 2.4.1. Specifically, one PMP region is reserved exclusively for the SM itself, and another protects the memory of the host OS. Each enclave requires two PMP regions—one for its untrusted buffer and another for its enclave memory—resulting in a total of four PMP regions for the two enclaves used in the demonstration. In addition, secure I/O requires the SM to protect the MMIO range of the designated device, consuming another PMP region. Finally, the shared memory region created as part of the IEC mechanism must also be assigned a dedicated PMP region. In combination, these protections account for all eight PMP registers, leaving no capacity for additional PMP regions at runtime.

6.4.3 Second Demo: Secure I/O for NIC Device

For the second demonstration, we combine the secure I/O mechanism, the network stack, and the NIC driver from Section 2.5 with the IEC mechanism described in Section 6.1.2. The purpose of this demo is twofold: to demonstrate that even complex I/O devices—specifically a NIC—can be operated securely within our architecture, and to show that the approach functions correctly on real hardware (Starfive VisionFive 2). The network enclave (i.e., the server enclave) exclusively claims the NIC during initialization and subsequently sets up both the network stack and the NIC driver. Afterward, a shared region is established using the IEC mechanism, enabling direct communication between the client enclave and the network enclave.

To transmit a packet, the client issues an ICALL over the shared region. The network enclave receives the packet, forwards it to the NIC driver, and—after processing by the network stack—transmits it through the NIC hardware. The corresponding response packet is passed back to the client enclave, completing the ICALL sequence.

As the hardware board provides only 8 PMP registers, the PMP limitations discussed in Section 6.4.2 apply equally to this demonstration. Despite these constraints, the demo executes successfully on the board.

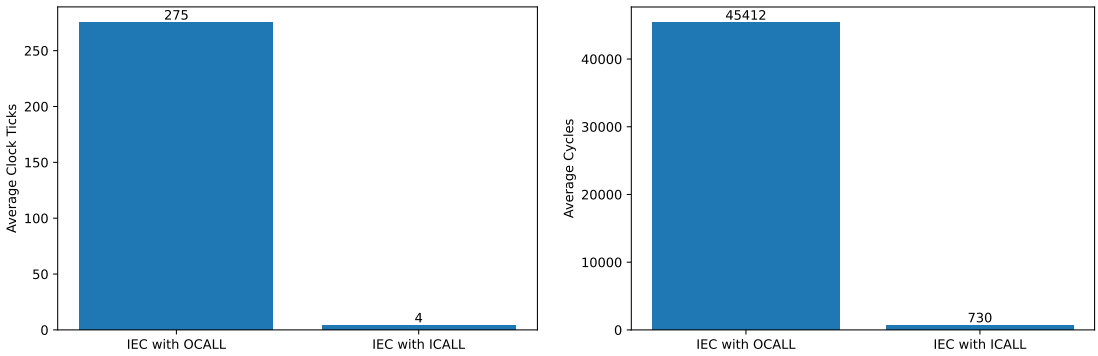
The NPS mechanism introduced in Section 6.3 is not directly integrated into this demonstration. The reason is a current limitation: the network enclave supports only a single client at a time, while NPS requires that both the host OS and at least one client enclave interact with the network enclave concurrently. Nevertheless, Section 6.3 shows that the standalone NPS PoC functions correctly. Achieving full integration would require introducing a scheduling mechanism within the network enclave to fairly handle multiple clients.

7 Performance Evaluation

This chapter evaluates the performance of our IEC mechanism, which implements enclave to enclave communication using ICALLs, by comparing it against the OCALL based approach. We begin by describing the experimental setup used to obtain the measurements. We then present the benchmark results for both communication mechanisms. Finally, we analyze the number of context switches required by each approach to explain the substantial performance differences observed.

7.1 Experimental Setup

All experiments were conducted on a StarFive VisionFive 2 single-board computer equipped with a StarFive JH7110 SoC. The board features the SiFive U74 core complex, consisting of four SiFive U7 cores and one SiFive S7 core. Each core provides a single hardware thread (hart) and operates at a frequency of 1.5 GHz. The platform supports 8 PMP regions with a minimum region granularity of 4096 bytes.
For the performance comparison, we evaluated two experimental setups. The first experiment measures the round-trip performance between a client enclave and a server enclave when enclave to enclave communication is performed via OCALLs (see the end of Section 2.5). The second experiment measures the same round-trip performance, but using ICALL based enclave to enclave communication (see Section 6.1.2).
Each experiment executes 5000 round-trip measurements. To capture timing information, we used two RISC-V instructions commonly employed in performance benchmarking: *rdtime* and *rdcycle* [RIS22]. The *rdtime* instruction reads a counter that increments with each tick of the real-time clock, effectively providing wall-clock timing. On the board, the real-time clock frequency is 4 MHz, and all harts share a synchronized timer. In contrast, the *rdcycle* instruction reads the cycle counter of the core on which the hart is running. To ensure fair and deterministic cycle measurements, all but one core were disabled when benchmarking with *rdcycle*.
For both experiments, we assume that the server enclave is already blocked and waiting for incoming requests. To measure the round-trip time (RTT), we start the timer immediately before the client enclave issues a request and stop it as soon as the client receives the corresponding response from the server enclave. The difference between these two timestamps yields the RTT for a single request.



(a) Clock Real Time for RTT

(b) CPU Cycles for RTT

Figure 7.1: Average values for both experiments using two different measurement instructions. For (a) we used the *rdtime* instruction. For (b) we used the *rdcycle* instruction. Each diagram compares the performance of both experiments.

7.2 Measurements

Figure 7.1 presents the measurement results for both experiments. Using the *rdtime* instruction, we observe an average of 275 clock ticks for IEC communication via OCALLs, compared to only 4 clock ticks for IEC communication via ICALLs. Based on these averages, this corresponds to a performance improvement of approximately a factor of 63. When using the *rdcycle* instruction, the results exhibit a similar trend. IEC communication via OCALLs requires an average of 45412 cycles, whereas ICALL-based communication requires only 730 cycles on average, yielding an improvement by a factor of roughly 62. Overall, the results demonstrate that our IEC mechanism provides substantial performance benefits for enclave-to-enclave communication, consistently improving performance by a factor of 62–63.

7.3 Context Switches (CS)

Note

We consider a context switch to occur for all types of calls (see the beginning of Section 2.4.4 for definitions) that transition between different privilege modes, as these calls internally rely on the *ecall* instruction. Additionally, any transition from the runtime to the host operating system is also counted as a context switch, because the security monitor saves and restores the execution context when switching between the secure and non-secure world.

The previous section demonstrated substantial performance improvements achieved

through our IEC mechanism. In this section, we examine the underlying causes of this difference. Specifically, we analyze the number of context switches required to complete a single round-trip request from a client enclave to a server enclave. As in our measurements, we assume that the server enclave is already blocked and waiting for incoming requests. Our analysis shows that the performance overhead of the OCALL based approach stems primarily from the large number of required context switches (16 per round trip), whereas IEC communication via ICALLs does not involve any context switches at all. A detailed breakdown is provided below.

We begin with the context switches involved in IEC communication over OCALLs. To send a request to the server, the client enclave first issues a runtime syscall (1 CS), allowing the runtime to notify the SM via an SBI call (1 CS) that an OCALL has occurred. The SM then switches execution to the host operating system (1 CS), which forwards the OCALL to the host application (1 CS). The host application subsequently signals the server enclave that a new client request is available. Since the server enclave is initially blocked in an OCALL waiting for incoming requests, this notification must traverse several layers as well.

To wake the server enclave, the host application prepares the client request as the return value of the server's blocking OCALL and then issues an ioctl call to request the enclave's resumption (1 CS). The host operating system forwards this request to the SM via another SBI call (1 CS). The SM then restores the server enclave's context and transfers control back to its runtime (1 CS). Finally, the runtime forwards the request to the server EAPP (1 CS). In total, the delivery of a client request to the server enclave generates 8 context switches. Since the response from the server to the client follows the same path, the overall number of context switches for a complete round trip sums to 16.

In contrast, when using our IEC mechanism with ICALLs, no context switches are required for enclave to enclave communication. After establishing the shared memory region during initialization, both the client and server enclaves map this region directly into their address spaces. As a result, the only runtime overhead is the lightweight RPC mechanism implemented on top of the shared memory region (icalls), which merely involves copying data to and from the shared region.

8 Conclusion

This chapter summarizes the key contributions of this thesis and discusses the insights gained throughout the work. In addition, it highlights opportunities for future improvements and extensions that can further advance the presented concepts.

8.1 Summary

This thesis set out to improve the availability and robustness of TEEs on RISC-V, with a particular focus on secure I/O in Keystone. While TEEs typically guarantee confidentiality and integrity, ensuring availability in the presence of a powerful OS-level attacker remains an open challenge. Our work addresses this gap by enhancing Keystone with mechanisms that strengthen enclave isolation, reduce host dependencies, and enable exclusive access to I/O devices. Our primary contribution is the integration of the Elasticlave memory model into Keystone, enabling a host-independent IEC mechanism. Building on this foundation, we implemented a shared memory RPC mechanism that allows enclaves to communicate directly without relying on the untrusted host OS. Our evaluation on real hardware shows that our IEC mechanism significantly outperforms Keystone's native communication paths, while simultaneously improving isolation.
Although the integration of the seL4 microkernel ultimately proved infeasible, we adopted an alternative approach that retained Eyrie while integrating our IEC and the secure I/O mechanisms from the case study.
Finally, we introduced NPS, a mechanism that forces the untrusted host OS to use the secure world's network enclave for all network access. Our PoC implementation on a RISC-V hardware board demonstrated that NPS is feasible using Linux's tun/tap interface and modest changes to the Keystone host application.
Two demonstration applications validated the practicality of our contributions.

8.2 Future Work

The proposed IEC mechanism (see Section 6.1.2) enables fully host-independent communication after the shared memory regions have been established. However, the initial setup phase still requires interaction with the untrusted host OS. This dependency results from two core limitations. First, the SM cannot allocate enclave memory dynamically, as

this would increase its trusted computing base. As a consequence, memory for shared regions must be allocated by the host OS, making initial setup inherently dependent on it. Second, the *eid* and *uid* values required for Elasticlave's *share* and *map* operations must be exchanged before enclaves can finalize a shared region. This bootstrapping phase currently relies on ocalls. Future work could explore mechanisms to eliminate or minimize this dependence. One approach is the use of static, preassigned *eid*s and *uid*s, allowing enclaves to establish shared regions deterministically without host involvement. However, this only addresses identifier exchange, so the challenge of memory allocation remains. Investigating safe and minimal SM extensions or lightweight memory allocation primitives that do not significantly expand the TCB could therefore be a valuable direction.
Another fundamental limitation observed during the implementation is the scarcity of PMP registers. Many RISC-V boards (including the Starfive VisionFive 2) offer only 8 or 16 PMP entries. Our design, especially when combining secure I/O with IEC and shared regions, can exhaust all available entries easily. This restricts scalability and limits the number of enclaves and shared regions that can coexist. A promising area of future research is the development of PMP virtualization. Such mechanisms could allow the SM to dynamically remap or reuse PMP entries while preserving strong isolation guarantees. There is a promising pull request for this already.[9]

[9] `https://github.com/keystone-enclave/keystone/pull/445` Accessed: 2025-12-01

References

[Bah+21] Raad Bahmani et al. “CURE: A Security Architecture with CUstomizable and Resilient Enclaves”. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1073–1090. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/bahmani.

[GL22] Liwei Guo and Felix Xiaozhu Lin. “Minimum viable device drivers for ARM trustzone”. In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys ’22. ACM, Mar. 2022, pp. 300–316. DOI: 10.1145/3492321.3519565. URL: http://dx.doi.org/10.1145/3492321.3519565.

[Gro] RISC-V Platform Runtime Services Task Group. *RISC-V Supervisor Binary Interface Specification*. https://github.com/riscv-non-isa/riscv-sbi-doc. Accessed: 2025-04-01.

[Hei25] Gernot Heiser. “The seL4 Microkernel - An Introduction”. In: *The seL4 Foundation* (2025). Revision 1.4 of 2025-01-08.

[Int24] Intel. *Intel® 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. Intel Corporation. 2024.

[Lee+20] Dayeol Lee et al. “Keystone: an open framework for architecting trusted execution environments”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys ’20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387532. URL: https://doi.org/10.1145/3342195.3387532.

[Lee+22] Dayeol Lee et al. “Cerberus: A Formal Approach to Secure and Efficient Enclave Memory Sharing”. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’22. ACM, Nov. 2022, pp. 1871–1885. DOI: 10.1145/3548606.3560595. URL: http://dx.doi.org/10.1145/3548606.3560595.

[Pan+25] Shangjie Pan et al. “Dep-TEE: Decoupled Memory Protection for Secure and Scalable Inter-enclave Communication on RISC-V”. In: *Proceedings of the 30th Asia and South Pacific Design Automation Conference*. ASPDAC ’25.

Tokyo, Japan: Association for Computing Machinery, 2025, pp. 454–460. ISBN: 9798400706356. DOI: 10.1145/3658617.3697763. URL: https://doi.org/10.1145/3658617.3697763.

[RIS] RISC-V International. *RISC-V Official Website*. https://riscv.org/about/. Accessed: 2025-04-01.

[RIS22] RISC-V. *The RISC-V Instruction Set Manual: Volume I Unprivileged Architecture*. RISC-V International. 2022.

[RIS24] RISC-V. *The RISC-V Instruction Set Manual: Volume II Privileged Architecture*. RISC-V International. 2024.

[SAB15] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. "TEE: What It is, and What It is Not". In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 1. 2015, pp. 57–64. DOI: 10.1109/Trustcom.2015.357.

[Sch+21] Moritz Schneider et al. "Composite Enclaves: Towards Disaggregated Trusted Execution". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (Nov. 2021), pp. 630–656. ISSN: 2569-2925. DOI: 10.46586/tches.v2022.i1.630-656. URL: http://dx.doi.org/10.46586/tches.v2022.i1.630-656.

[Sch22] Fabian Schwarz. "TrustedGateway: TEE-Assisted Routing and Firewall Enforcement Using ARM TrustZone". In: *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID '22. Limassol, Cyprus: Association for Computing Machinery, 2022, pp. 56–71. ISBN: 9781450397049. DOI: 10.1145/3545948.3545961. URL: https://doi.org/10.1145/3545948.3545961.

[SGG18] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Operating system concepts (10. ed.)*. 2018. ISBN: 978-1-119-32091-3.

[VJ15] Aditya Venkataraman and Kishore Kumar Jagadeesha. "Evaluation of Inter-Process Communication Mechanisms". In: 2015. URL: https://api.semanticscholar.org/CorpusID:6899525.

[Wan+22] Jinwen Wang et al. "RT-TEE: Real-time System Availability for Cyber-physical Systems using ARM TrustZone". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, pp. 352–369. DOI: 10.1109/SP46214.2022.9833604.

[Wat16] Andrew Waterman. "Design of the RISC-V Instruction Set Architecture". PhD thesis. EECS Department, University of California, Berkeley, Jan. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html.

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 640–656. DOI: 10.1109/SP.2015.45.

[Yan+24] Huaiyu Yan et al. "LDR: Secure and Efficient Linux Driver Runtime for Embedded TEE Systems". In: *Proceedings of the 2024 Annual Network and Distributed System Security Symposium, NDSS'24*. 2024.

[Yu+22] Jason Zhijingcheng Yu et al. "Elasticlave: An Efficient Memory Model for Enclaves". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 4111–4128. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/yu-jason.

Acronyms

DMA Direct Memory Access.

DoS Denial of Service.

EAPP Enclave Application.

ICALL Inter-Enclave Calls.

IEC Inter-Enclave Communication.

IoT Internet of Things.

MMIO Memory Mapped I/O.

NIC Network Interface Card.

NPS Network Port Sharing.

OCALL Outgoing Calls.

PMP Physical Memory Protection.

PoC Proof of Concept.

RoT Root of Trust.

RPC Remote Procedure Calls.

RT Runtime.

RTOS Real-Time Operating System.

SBI Supervisor Binary Interface.

SM Security Monitor.

SoC System on a chip.

TCB Trusted Computation Base.

TEE Trusted Exectution Environments.