



Improving the Isolation in Unikernels by Leveraging Microkernel Design

D.A. Amarasinghe
Index number: 20000103

Supervisor: Dr. C.I. Keppitiyagama
Co-supervisor: Mr. Tharindu Wijethilake

June 30, 2025

Submitted in partial fulfillment of the requirements of the
B.Sc. (Honours) in Computer Science Final Year Project



Declaration

I certify that this dissertation does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Candidate Name : D.A. Amarasinghe



29-06-2025

Signature

Date

This is to certify that this dissertation is based on the work of Mr. D.A. Amarasinghe under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Supervisor Name : Dr. C.I. Keppitiyagama



30-06-25

Signature

Date

Co-supervisor Name : Mr. Tharindu Wijethilake

Signature

Date

Abstract

Modern systems demand strong isolation guarantees to protect against software vulnerabilities and maintain robust security boundaries. Traditional operating systems often fall short due to their large trusted computing bases and reliance on discretionary access control mechanisms. The seL4 microkernel, with its formally verified design and capability-based access control, presents a compelling foundation for building highly secure, isolated systems. This study explores the feasibility of running unikernel applications such as Rumprun, on top of seL4, aiming to combine the minimalism and efficiency of unikernels with the strong isolation guarantees of seL4. Two different integration approaches were attempted by directly running a Rumprun binary in a minimal seL4 environment and embedding a Rumprun application within a CAMkES component. Both efforts encountered technical obstacles. Despite these practical limitations, the theoretical underpinnings of seL4’s capability system suggest that high isolation levels are achievable if integration complexity can be addressed. Initial experiments highlight that a hybrid architecture, combining minimal seL4 or CAMkES components with standalone unikernels might strike a balance between performance and isolation. Future directions include debugging existing integration challenges, evaluating the performance implications of different architectural decisions, and exploring alternative unikernel frameworks for better alignment with seL4’s static and modular design principles. The findings support that seL4 is well-suited for hosting unikernels in security-critical environments, provided that toolchain and architectural hurdles are resolved.

Preface

In completing the requirements for my B.Sc in Computer Science Final Year Project (SCS4224), this research represents a significant milestone in my academic journey. The work began with a comprehensive review of existing literature, carefully analyzing relevant publications that shaped the direction of the study. Building upon this foundation, I progressed through the design and implementation phases, primarily guided by the official documentation and manual of seL4. This stage showcases my independent effort and technical competence in developing and implementing the proposed solution.

While the core development was undertaken independently, I also made use of partial prior works related to kernel integrations, which provided valuable insights and direction during the process. The seL4 manual played a crucial role in helping me understand and apply essential concepts, tools, and commands throughout the research. The original contributions of this project are presented in the results section, where extensive experimentation and analysis have led to new insights in the field. All relevant prior work and resources have been properly acknowledged.

Acknowledgement

I am deeply grateful to the University of Colombo School of Computing (UCSC) for providing me with the opportunity to undertake this research project. This experience has been profoundly impactful in shaping both my academic and personal growth. I extend my sincere appreciation to Dr. C. I. Keppitiyagama, my supervisor, and Mr. Tharindu Wijethilake, my co-supervisor at UCSC, for their expert guidance, constructive feedback, and steadfast support throughout this journey. Their mentorship played a vital role in the successful completion of this work. Special thanks are also due to contributors and members of the seL4 community, whose discussions, documentation, and prior works particularly those involving kernel integration were instrumental in overcoming technical challenges and deepening my understanding. Lastly, I am truly thankful to my friends and family for their continuous encouragement and support, which have been a constant source of motivation and strength during this endeavor.

List of Acronyms

AEP	Asynchronous Endpoint
API	Application Programming Interface
ARM	Advanced RISC Machines
BSD	Berkeley Standard Distribution
CMU	Carnegie Mellon University
CPU	Central Processing Unit
EPT	Extended Page Table
FIFO	First In First Out
IEEE	Institute of Electrical and Electronics Engineers
IPC	Inter Process Communication
KPTI	Kernel Page Table Isolation
QEMU	Quick Emulator
KVM	Kernel-based Virtual Machine
MINIX	Mini-UNIX
MIPS	Microprocessor without Interlocked Pipelined Stages
MPK	Memory Protection Key
MTE	Memory Tagging Extension
OS	Operating System
PKRU	Protection Key Rights Register
PKU	Protection Key for Userspace
POSIX	Portable Operating System Interface
RISC	Reduced Instruction Set Architecture
RPC	Remote Procedure Call
SPARC	Scalable Processor ARChitecture
TCB	Trusted Code Base
TLB	Translation Look-aside buffer
UNIX	UNiplexed Information Computing System
VM	Virtual Machine
DSRM	Design Science Research Methodology

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Background	2
1.2.1	Hypervisor	2
1.2.2	Microkernel	2
1.2.3	Unikernels	3
1.3	Gap and Research Questions	4
1.4	Aim and Objectives	5
1.5	Research Scope	7
1.6	Significance of the research	7
1.7	Methodology & Evaluation Criteria	8
1.7.1	Research Methodology	8
1.7.2	Evaluation Methods	10
2	Literature Review & Related Work	13
2.1	Evolution of Microkernels	13
2.1.1	MINIX: As an Educational Microkernel	13
2.1.2	Portability of Kernel Design	14
2.1.3	POSIX Standards	16
2.1.4	First Generation Microkernels	16
2.1.5	Second Generation Microkernels	16
2.1.6	Current Generation Microkernels	17
2.1.7	seL4 Microkernel	18
2.2	Evolution of Unikernels	18
2.2.1	Minimized Monolithic Kernels	19
2.2.2	Clean-Slate Unikernels	20
2.3	Inter Process Communication (IPC)	20
2.3.1	Inter Process Communication in Monolithic Kernels	21
2.3.2	Inter Process Communication in Microkernels	22
2.4	Process Isolation	24
2.4.1	Domain Isolation	25
2.4.2	Extended Page Table Switching	26
3	Design	28
3.1	Setting up the environment	28
3.2	Selecting a Compatible Unikernel	29
3.3	Design 1 : Intra-Unikernel Isolation	29
3.4	Design 2 : Inter-Unikernel Isolation	31
4	Implementation	33
4.1	Test Environment	33
4.2	Implementation of design 1	33
4.2.1	Obtaining the Source	33

4.2.2	Preparing Rumprun Dependencies	33
4.2.3	Building a Rumprun Application	34
4.2.4	Encountering Runtime Issues	34
4.2.5	Diagnosing the Issue	34
4.2.6	Identifying the Root Cause	35
4.3	Implementation of design 2	35
4.3.1	Root Task	36
4.3.2	User Apps	40
5	Results Evaluation	41
5.1	Qualitative Analysis of Isolation Aspects	41
5.2	Performance Benchmarking of seL4	42
5.2.1	IPC Performance	43
5.2.2	Fault overhead	43
5.2.3	Network throughput	44
6	Conclusion & Future Work	46
6.1	Limitations	46
6.2	Future Directions	47
	List of References	48

1 Introduction

In cloud computing and virtualization, the need for secure and efficient isolation has become important. Isolation refers to the ability to separate computing environments to ensure they do not interfere with each other (Khajehei 2014). This improves the security, stability, and resource utilization between isolated environments. Isolation can be done in various forms including process isolation, memory isolation, file system isolation, network isolation, etc (Khajehei 2014).

Virtualization technologies are crucial in effectively achieving such forms of isolation in cloud environments. Virtualization allows multiple isolated environments to coexist on the same physical hardware by abstracting computing resources. Two main technologies to achieve virtualization are Virtual Machines (VMs) and Containers. VMs provide strong isolation through a software layer known as a hypervisor. It introduces a dedicated Operating System (OS) and a kernel for the applications. Hence, VMs are heavier in terms of performance and resource usage. Contrary to VMs, containers are lightweight in terms of performance and resource usage but provide less isolation compared to VMs as the shared kernel can be a single point of failure (Barik et al. 2016).

As the limitations of both VMs and containers became more apparent, researchers and practitioners began exploring alternative solutions that could combine the performance benefits of containers with improved security guarantees. One such emerging technology is the unikernel. In the past few years, unikernels became a popular alternative to costly VMs because of their lightweight nature. Even though the performance drawbacks in VMs can easily be solved using unikernels the lack of isolation stays as it is (Sung et al. 2020). This is because unikernels only have a kernel space and the assumption is that all the components within a single unikernel must trust each other. However, there can be situations where an application has untrusted components as well. Therefore it is not safe to include them all together in the same kernel space (Sung et al. 2020).

1.1 Motivation

While unikernels have demonstrated significant performance advantages over VMs, they often lack in providing robust isolation necessary for multi-tenant environments. Containers, on the other hand, offer lightweight and fast deployments but often fall short of providing strong isolation, making them more vulnerable to security threats (Sultan et al. 2019). Most research has been conducted to improve the performance of unikernels by emphasizing their efficiency and lightweight nature. However, it is essential not to discard the importance of isolation when making improvements to the performance. Microkernels can be considered as a better way of providing isolation as it has a minimum Trusted Code Base (TCB) that runs in the privileged mode with lower-level services such as IPCs, memory management, thread management, etc. Improving the isolation of unikernels by

implementing them on top of the microkernel architecture could be a turning point toward balancing performance and isolation in a virtual environment.

1.2 Background

Over the years virtualization has transformed the face of modern computing, particularly with the rise of cloud computing. Initially, it was implemented for monolithic kernels such as Linux (Kolyshkin 2006). As Figure 1.1 shows, monolithic kernels are efficient in terms of performance because they allow direct access to hardware resources and system services by bringing user processes into kernel space. However, the integration of numerous services within the kernel space also indicates that any fault or vulnerability can potentially affect the entire system, posing significant risks to security and stability (Drebes & Nanya 2010).

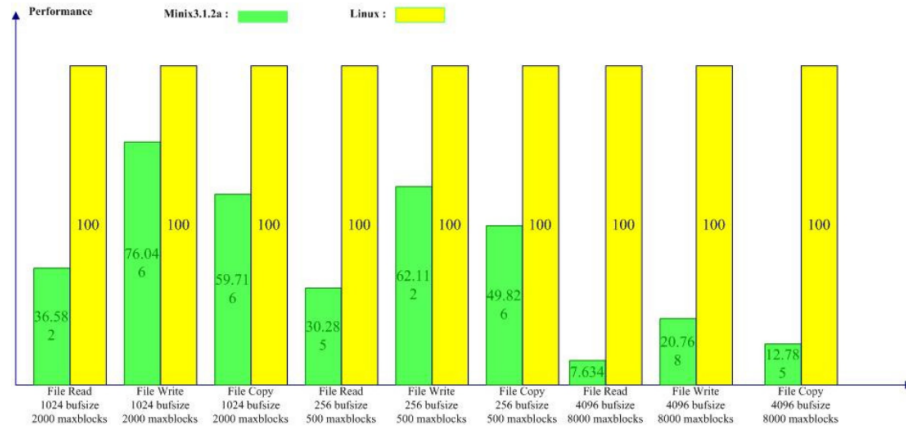


Figure 1.1: File system benchmarks (Miao 2011)

1.2.1 Hypervisor

Since virtualization was primarily based on monolithic architecture, to manage multiple VMs, a layer named hypervisor was introduced. There are two types of hypervisors, referred to as bare-metal (Type 1) and hosted (Type 2) (Rodríguez-Haro et al. 2012). The difference between type 1 and type 2 is that, the type 1 hypervisors run directly on the host machine's hardware as a lightweight OS, while type 2 hypervisors run as a software layer on top of the OS. See Figure 1.2. In the case of type 2 hypervisors, from the host machine point of view, each VM is another process (Rodríguez-Haro et al. 2012).

1.2.2 Microkernel

Microkernel architectures such as seL4 recently became popular as they came up with security and isolation from the low-level design (Matos & Ahvenjärvi 2022). seL4 has two distinct use cases as it can operate as an OS as well as a type 1

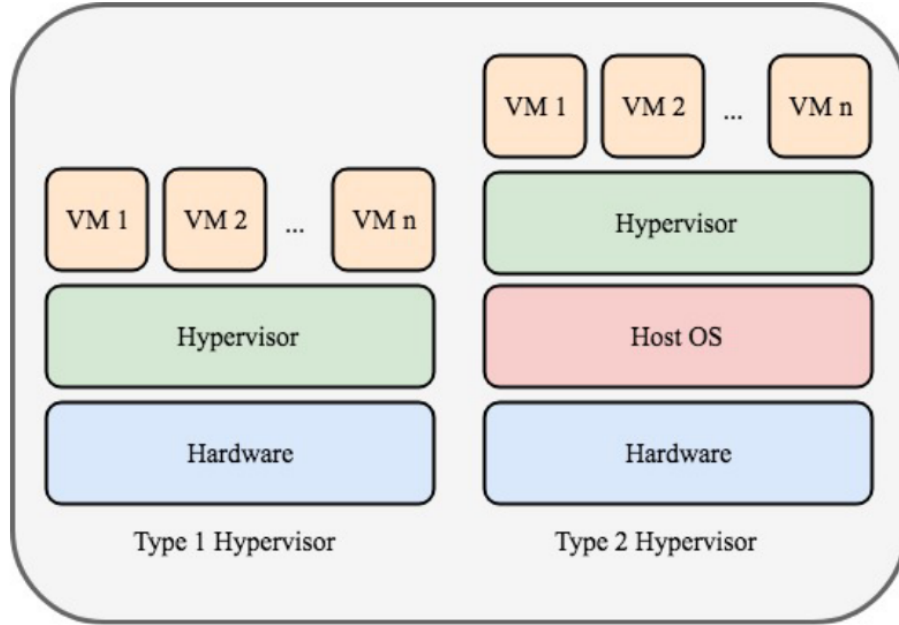


Figure 1.2: Two types of hypervisors (Alnaim et al. 2019)

hypervisor. Because of that, it can directly allocate hardware resources (such as Central Processing Unit (CPU), memory, and storage) to VMs or guest operating systems. This mitigates the risks associated with larger hypervisor attack surfaces, as the hypervisor functionality remains tightly integrated and isolated within seL4’s minimal trusted computing base.

Furthermore, seL4’s formal verification ensures that critical properties, such as code-level functional correctness and scheduling policies, are rigorously verified, offering a robust foundation for virtualization (Klein et al. 2009). This direction not only enhances the trustworthiness of virtualized environments but also underscores the growing interest in leveraging microkernel architectures for secure and efficient virtualization solutions.

1.2.3 Unikernels

Unikernels represent a specialized form of lightweight virtualization, where an application is compiled with only the necessary components of an operating system, tailored to run as efficiently as possible on a specific hardware platform. Unlike traditional VMs, unikernels lack the general-purpose nature of typical OSes, resulting in extremely small footprints and optimized performance (Madhavapeddy et al. 2013).

However, this minimalism comes with trade-offs, such as reduced isolation compared to full-fledged VMs, because of the single kernel space. To address this challenge, a promising direction is integrating unikernels with microkernel architectures such as seL4, which can work as a bare-metal hypervisor. Microkernels are designed to provide minimal core functionality, enhancing security by isolat-

ing components in separate address spaces. Implementing unikernels as minimal VMs atop a microkernel like seL4 will become a turning point towards a balanced environment of both performance and isolation.

1.3 Gap and Research Questions

While significant advancements have been made in optimizing the performance aspect of virtualized components, for example, following a single kernel space approach like unikernels has demonstrated significant performance improvements compared to traditional VMs, there remains a notable gap in research addressing the isolation aspects of these lightweight virtualized components. Enhancing the isolation capabilities of unikernels when deployed on microkernel architectures like seL4 could potentially bridge this gap by offering improved security guarantees without compromising on the performance benefits that unikernels provide.

Moreover, as seL4 gains traction for its ability to act as a type 1 hypervisor without requiring a full-fledged operating system, there emerges a compelling case for exploring its role in supporting multiple isolated applications running as seL4-based unikernels on a shared server. This aligns with trends in cloud computing where efficiency and security are important, suggesting a future direction where seL4-based unikernels could serve as lightweight, highly isolated deployment units. However, the specific mechanisms and optimizations required to achieve this dual goal of performance and isolation within seL4-based unikernels remain underexplored, constituting another significant avenue for future research. While current research has demonstrated the potential of unikernels and microkernels like seL4 individually, there exists a clear research gap in integrating these technologies to enhance isolation capabilities while preserving their performance advantages.

A key feature of seL4 that distinguishes it from other microkernel architectures is its rigorous capability-based access control model. This model enforces strict access permissions through capabilities, which are unforgeable tokens representing access rights to system resources. Unlike traditional discretionary access control systems, seL4's capability model ensures that components cannot access or interfere with resources unless explicitly granted permissions via capabilities. This mechanism is critical for building highly secure and isolated execution environments, which is particularly beneficial when running multiple unikernels that should remain isolated from each other, even in the presence of potentially compromised components.

In addition, seL4 promotes a component-based approach to application development, where software systems are decomposed into small, isolated components that communicate only through explicitly defined interfaces. This architectural principle naturally aligns with the minimalist and specialized nature of unikernels, enabling the development of systems with a strong separation of concerns and a minimal trusted computing base. However, the integration of this design philosophy with unikernel execution models remains largely theoretical, with little

empirical research investigating its effectiveness in real-world scenarios involving performance-sensitive and security-critical workloads. Therefore, despite the recognized strengths of seL4 in providing formal verification, strong isolation guarantees, and a secure execution environment, there is a lack of comprehensive studies exploring how these features can be leveraged in the context of unikernel-based deployments (Klein et al. 2009). Bridging this gap requires a detailed examination of the interaction between seL4’s security mechanisms and the operational behavior of unikernels, particularly in managing inter-process communication, capability distribution, and resource allocation. Addressing these questions can significantly contribute to the body of knowledge in secure systems design, especially in environments that demand both high assurance and high performance.

The research primarily revolves around exploring the following research questions.

1. How can integrating the seL4 microkernel with unikernels improve the isolation of virtual components by leveraging the inherent isolation from the microkernel design and the higher level of abstraction, such as file system isolation, provided by unikernels?

This research explores integrating the seL4 microkernel with unikernels to enhance isolation capabilities without compromising performance. The approach leverages seL4’s inherent strengths in enforcing strict isolation through formal verification and capability-based access control, combined with the lightweight, single-purpose design of unikernels. The focus is on identifying architectural modifications and design strategies that enable improved separation between components and controlled resource access within the unikernel environment. The overarching objective is to achieve robust isolation at both the microkernel and application levels while preserving the efficiency and minimalism that make unikernels attractive for modern cloud and edge deployments.

2. What are the performance trade-offs when using microkernel-based isolation in unikernel environments compared to traditional approaches?

This focuses on evaluating the outcomes of integrating microkernels with unikernels in terms of both performance and security. It aims to quantify the improvements in isolation and understand any potential performance impacts, comparing these results to traditional operating systems. This will help in assessing the practicality and benefits of this approach in real-world applications.

1.4 Aim and Objectives

This research aims to investigate and enhance the isolation capabilities of unikernels through the integration of microkernel architecture, specifically using seL4 as the microkernel and Rumprun as the unikernel framework. This integration seeks to leverage seL4’s formally verified isolation mechanisms and capability-based access control alongside Rumprun’s lightweight and efficient execution model. The

research will evaluate the security and performance characteristics of such a system, with a particular focus on identifying and minimizing performance trade-offs compared to traditional virtualization approaches such as virtual machines and containers.

The research process intended to achieve the following objectives.

Research Question	Objectives
How can integrating the seL4 microkernel with unikernels improve the isolation of virtual components by leveraging the inherent isolation from the microkernel design and the higher level of abstraction, such as file system isolation, provided by unikernels?	<ul style="list-style-type: none"> • To design and implement a prototype unikernel system that leverages the seL4 microkernel for enhanced isolation. • To conduct a comprehensive evaluation of the isolation capabilities of the seL4-based unikernel system through security testing and analysis. • To optimize the unikernel-microkernel integration for maintaining or improving performance metrics such as CPU utilization.
What are the performance trade-offs when using microkernel-based isolation in unikernel environments compared to traditional approaches?	<ul style="list-style-type: none"> • To establish a controlled experimental environment to benchmark the performance of the seL4-based unikernel system. • To measure and compare key performance metrics between the seL4-based unikernel system, traditional operating systems. • To analyze the performance trade-offs associated with using microkernel-based isolation in unikernel environments.

Figure 1.3: Research Objectives

1.5 Research Scope

The following areas will be addressed under the scope of this research.

- Integration of Unikernels with seL4 Microkernel
 - Exploring how microkernel architecture can enhance isolation in unikernels while maintaining performance advantages.
 - Focus on designing and implementing a prototype system where unikernels leverage the isolation capabilities of the seL4 microkernel.
- Performance Evaluation and Comparison
 - Conduct thorough performance benchmarks to measure metrics.
 - Evaluate performance trade-offs between seL4-based unikernels and traditional operating systems as it is crucial to understand the practical implications of using microkernel-based isolation.
- Security and Isolation Analysis
 - Identify potential vulnerabilities and evaluate how well the isolation mechanisms prevent unauthorized access or interference.
 - Verify and validate the boundaries between the unikernel instances and the underlying seL4 microkernel.
 - Compare the isolation effectiveness of the seL4-based unikernel system with traditional monolithic kernel-based solutions.
- Documentation and Knowledge Contribution
 - Document the research methodology, findings, and conclusions comprehensively.

1.6 Significance of the research

Advancing research in integrating seL4-based unikernels for improved isolation and performance optimization extends to several key beneficiaries and application domains. First, cloud service providers stand to benefit significantly from enhanced security and efficiency. By deploying seL4-based unikernels, providers can offer stronger isolation guarantees between multi-tenant applications on shared infrastructure, thereby improving trust and compliance with data protection regulations. This capability is particularly relevant in multi-tenant environments where isolation breaches arise significant risks.

Moreover, enterprises adopting cloud-native architectures and edge computing scenarios would gain from the improved resource efficiency and scalability offered by seL4-based unikernels. These lightweight units of deployment could streamline

the deployment and management of microservices, IoT applications, and real-time data processing tasks. By minimizing overhead and maximizing performance through seL4’s microkernel design, organizations can achieve better utilization of hardware resources while maintaining robust isolation. In addition to commercial applications, the researchers studying the area of operating system design and virtualization techniques would have a niche to seek for new methodologies and optimizations. This research will contribute new insights into how microkernels such as seL4, can be leveraged to address the inherent security limitations of unikernels. Therefore, By documenting the design, implementation, and evaluation of a seL4-based unikernel system, this research will provide a detailed methodology that can be replicated and extended by other researchers. Additionally, the comparative analysis of isolation and performance trade-offs with traditional VMs and containers will offer a valuable reference point for future studies, enabling a deeper understanding of the strengths and weaknesses of various virtualization technologies.

Edge computing deployments often operate in less secure environments compared to centralized cloud data centers. Enhancing isolation in edge devices using seL4-based unikernels can protect against local attacks and unauthorized access, making edge computing more reliable for critical applications. For example, smart city infrastructure, which relies on edge devices to process data from sensors and cameras, can use this approach to safeguard against attacks that could disrupt urban services or compromise privacy.

By addressing the isolation challenges of unikernels through the integration with seL4, this research not only advances theoretical knowledge but also provides practical solutions that can be directly applied to enhance the security and efficiency of cloud and edge computing environments.

1.7 Methodology & Evaluation Criteria

This research will follow the Design Science Research Methodology (DSRM), which is well-suited for addressing the design and knowledge challenges associated with enhancing isolation in unikernels through microkernel architecture, specifically seL4. This methodology comprises three interrelated cycles: the relevance cycle; which identifies the problems, requirements, and evaluation criteria from the environment and feeds them into the design science research process, the design cycle; which generates and refines artifacts that meet the identified requirements and evaluation criteria, and finally the rigor cycle; which ensure the research process is rigorous and contributes new knowledge to the knowledge base (Hevner 2007). The key components of each of the cycles are listed below.

1.7.1 Research Methodology

This research adopts a structured methodology to explore the integration of the seL4 microkernel with the Rumprun unikernel framework, to enhance isolation

while maintaining performance. The methodology involves identifying relevant stakeholders, defining technical and research requirements, and producing artifacts that demonstrate and evaluate the integration.

Key stages include the design and development of a prototype unikernel system utilizing seL4's isolation features, followed by iterative refinement to address implementation challenges and optimize performance. A controlled experimental environment will be used for rigorous performance benchmarking and security analysis, comparing the seL4-based system against traditional unikernels.

The research also includes a thorough analysis of seL4's capability-based isolation in the context of unikernel deployment. Findings will be documented and shared through research publications, contributing to ongoing work in operating systems, virtualization, and secure systems design.

- Stakeholders
 - Users of virtualization technologies seeking enhanced isolation and performance
 - Researchers in the fields of operating systems, virtualization, and security.
 - Developers and engineers working on unikernels, microkernels, and related technologies.
- Requirements
 - Enhancements in low-level isolation mechanisms for unikernels closer to the operating system kernel.
 - Understanding the relationship between the inherent isolation provided by microkernel design (seL4) and the isolation requirements of unikernels.
 - Discussion and validation of using microkernels to improve isolation in unikernel-based virtualized environments.
 - Performance and isolation comparison between traditional monolithic kernels and microkernel-based implementations of unikernels.
- Artifacts
 - A prototype unikernel system utilizing the isolation features of the seL4 microkernel.
 - Comprehensive documentation and analysis of the relationship between seL4 microkernels and unikernel isolation.
- Design and Development
 - Select a suitable unikernel framework and integrate it with the seL4 microkernel.
 - Design a unikernel-based system architecture leveraging the isolation capabilities of seL4.

- Develop and implement the prototype, ensuring effective interaction between the unikernel and the seL4 microkernel.
- Iterative Refinement
 - Conduct iterative development cycles to refine the prototype based on feedback and initial testing results.
 - Address technical challenges and optimize the system for enhanced performance and isolation.
- Performance Evaluation
 - Set up a controlled experimental environment to test the seL4-based unikernel prototype and traditional unikernel.
 - Conduct performance benchmarks to measure metrics such as CPU utilization, and network throughput.
- Security and Isolation Analysis
 - Perform security testing, including penetration tests and vulnerability assessments, to evaluate isolation strength.
 - Compare the isolation effectiveness of the seL4-based system with traditional operating systems.
- Knowledge Contribution
 - Document the methods, findings, and conclusions in detail.
 - Prepare research papers and presentations to disseminate the results to the academic and professional community.
 - Update the knowledge base with newly discovered insights and artifacts.

1.7.2 Evaluation Methods

Previous studies have proposed various methods for evaluating isolation in virtualization environments, especially under conditions involving potential malicious attacks. A notable work in this area addresses virtualization security and isolation from the perspective of resilience to attacks (Hakamian & Rahmani 2015). The study focuses on identifying where security policies should be applied within the system to maximize isolation and reduce the risk of isolation failure.

The authors utilized a semi-Markov model to evaluate security measures in different layers of the virtual machine environment, such as the application layer and guest operating system memory. Their findings indicate that, on average, an attacker could cause an isolation failure within 190 hours with an 89% probability if left unchecked. The semi-Markov model enabled a sensitivity analysis that highlighted critical areas for improving isolation, particularly the application layer and memory-related defenses, which were shown to have a significant impact on increasing the system’s mean time to isolation failure.

While this probabilistic approach effectively identifies areas for implementing security measures, it does not fully align with the goals of this research. Rather than determining where security policies should be applied, the aim of this research is to evaluate whether running unikernels on seL4 microkernel improves isolation itself. Focus is on measuring the direct effectiveness of seL4 in enforcing memory, fault, and file systems across and within unikernel instances rather than identifying specific layers for implementing defensive measures.

1. **Memory Isolation** - To assess memory isolation, experiments will be conducted where each unikernel attempts to access memory beyond its allocated boundaries. These tests involve read and write operations targeting memory regions assigned to other unikernels. Observing seL4's response to such access attempts will provide insights into its effectiveness in enforcing memory boundaries and preventing unauthorized access.
2. **Fault Isolation** - This will be evaluated by intentionally triggering faults in a controlled unikernel environment. This includes introducing crashes or exhausting specific resources within one unikernel, followed by monitoring the stability and responsiveness of other unikernels. By isolating these faults within a single unikernel, seL4's ability to contain errors and prevent their propagation across unikernels can be verified, highlighting its role in fault resilience.
3. **File System Isolation** - this can be evaluated by attempting to access or modify files and directories assigned to other unikernels. Each unikernel will perform file operations such as writing and reading files within its designated storage area while testing for unauthorized access attempts by other unikernels. This includes verifying that each unikernel's file descriptors remain isolated, without leakage or cross-access vulnerabilities. Successful containment of each unikernel's file operations will demonstrate seL4's ability to restrict access, ensuring that each instance operates within its defined storage boundaries.

To assess the effectiveness of isolation when running unikernels on seL4, it is crucial to define specific evaluation criteria. Each criterion addresses a particular aspect of isolation necessary for secure, reliable operation in multi-component systems. The primary evaluation criteria include Memory Isolation, Fault Isolation, and File System Isolation.

1. Memory Isolation

Memory isolation ensures that each unikernel instance operates within a separate memory space, preventing unauthorized access to the memory of other instances. We need to consider whether one unikernel can access or modify the memory allocated to another unikernel and the effectiveness of seL4's memory management in maintaining strict boundaries between unikernel memory spaces.

2. Fault Isolation

The primary focus of this is to contain errors within the unikernel in which they occur, preventing faults from propagating and affecting other unikernels or the broader system.

3. File System Isolation

It is essential to prevent unikernels from accessing or modifying each other's files and directories. Evaluation points for file system isolation include, Whether unikernels are restricted to specific sections in the file system, preventing cross-access to the files of other unikernels.

2 Literature Review & Related Work

A preliminary literature review was conducted to explore the broader context of isolation in virtualized environments, with a particular focus on comparing key aspects such as inter-process communication and isolation mechanisms between monolithic and microkernel-based architectures. The review also examined hardware-assisted isolation techniques and component-based system designs within single-address spaces. Although the survey emphasized microkernel-based systems, the insights gained, particularly about low-level isolation and system modularity, are highly relevant and form a solid foundation for understanding the isolation capabilities of unikernel-based architectures in the later stages of this research.

2.1 Evolution of Microkernels

The debate between monolithic and microkernels has been a major topic in operating system design, each with its advantages and disadvantages. Monolithic kernels prioritize performance whereas microkernels prioritize modularity and isolation (Tanenbaum & Woodhull 2005, Torvalds 1997). Most of the exploration of different operating systems started with well-known UNiplexed Information Computing System (UNIX) system.

UNIX is one of the earliest OSs and has influenced the development of most modern operating systems. Development of UNIX started in the early 1960s at AT&T's Bell Telephone Laboratories (Tanenbaum & Woodhull 2005). The source code of UNIX was open to study until the UNIX version 7 came. After UNIX version 6 was released AT&T realized its commercial value and licensed version 6 for both education and commercial use. According to Tanenbaum & Woodhull (2005), the previous versions were only licenses for educational institutions. However, license of the version 7 specifically excluded the usage of source code for educational purposes. The source code was being studied by many universities. Most developments after Version 7 focused primarily on conceptual advancements, with minimal emphasis on practical implementation. In response to this, Professor Andrew S. Tanenbaum decided to develop his operating system from scratch called Mini-UNIX (MINIX) which is compatible with UNIX v7, and let the universities study the source code (Tanenbaum & Woodhull 2005).

2.1.1 MINIX: As an Educational Microkernel

MINIX is considered to be the root of many operating systems. According to Tanenbaum if MINIX was not there then there was no Linux, no Linux means no Android, because Android runs on Linux. MINIX have two advantages over UNIX. The source code of MINIX was available to study and MINIX was developed in a more modular manner than UNIX (Tanenbaum & Woodhull 2005). For

example, the memory manager, file system, and device drivers are independent of the kernel and can operate independently. Hence, the MINIX kernel is much smaller than UNIX which is around 4000 lines of code (Tanenbaum & Woodhull 2005). Smaller kernel code implies a smaller number of faults in the kernel. Ball et al. (2006) and Albinet et al. (2004) found that most of the kernel error occurs due to device drivers that are embedded into the kernel. It also implies the fact that microkernels have a much smaller number of errors.

Because of MINIX kernel being much smaller it did not require much of a memory space. According to Tanenbaum & Woodhull (2005), in the beginning, it did not even require a hard disk. At the time the cost of hardware was much higher, and being able to run without a hard disk was a huge advantage. One of the main differences between UNIX and MINIX is that they are designed for different purposes. UNIX was focused on efficiency while MINIX was focused on readability (Tanenbaum & Woodhull 2005).

2.1.2 Portability of Kernel Design

Tanenbaum believed that the emergence of Reduced Instruction Set Architecture (RISC) architectures will gradually take over the Intel line of 80x86. Thus developing an operating system that is more coupled with the architecture is not aligned with the timeline (DiBona & Ockman 1999). At the same time, Linus Torvalds began the development of a new operating system, Linux; which was tightly coupled with Intel's 80x86 architecture (Torvalds 1997). This set the stage for the well-known debate between Linus and Tanenbaum about the portability of each of their operating system kernels.

Torvalds (1997) states that even though 'simplicity' is said to be one of the key things in microkernels, it's doubtful because of the complexity of its implementation. Linux kernel is organized around its main services which are process handling, memory management, file system, network management, and drivers for the hardware. When designing the kernel there are architecture-specific modules (headers) and a common module that can be used through Application Programming Interfaces (APIs) (Torvalds 1997). The abstraction layer between the kernel and user applications is also important because there are several standards that the designer should adhere to. The standard should also be binary compatible with any legacy application to be able to run on the new operating system (Torvalds 1997).

There can be several problems associated with these standards as well. For example, for some architectures, there might be no standards defined, in such cases the architecture is free to define a standard but it's also limited to the capabilities of the architect and there can be several standards defined for the same architecture. According to (Torvalds 1997), the issues of multiple interfaces to the same OS are handled by a concept called 'personality'. Micro kernel-based operating systems have a personality server for each personality so that the operating system is independent of the personality of the interface (Torvalds 1997). Even though

this is the case, most operating systems have a single primary personality, and other personalities are used for things such as backward compatibility. Because of multiple personalities, Linux is said to be reasonably portable.

Indeed, Linux was tightly coupled with x86 architecture, especially with 80386 which Linux was originally written for (Torvalds 1997). Because of that reason, it was not considered as a portable operating system. Even though that was identified as a weakness in the operating system implementation Linus didn't try to avoid using x86-specific features. Linux was started as a project to identify what exactly can be done with the CPU, aiming to get the maximum usage of the CPU. According to Torvalds (1997) the portability issue was related to the implementation not to the design.

'Portability' and 'Ability to run on different architectures' are two different concepts. 'Portability' refers to running the same program in different systems or architectures while 'Ability to run on different architectures' refers to developing some program in a way that runs on a different system but the implementation can be different from system to system. According to Torvalds (1997), the first project in which Linux was involved in portability was Alpha RISC instruction set architecture by Digital. After that several ports were developed for other architectures such as Scalable Processor ARChitecture (SPARC), MIPS, PowerPC, and Advanced RISC Machines (ARM) (Torvalds 1997). Apart from these architectures, the Linux kernel had been ported to virtual environments such as Mach and L4 microkernels (Torvalds 1997).

For better portability, the architecture-specific code should be simple and minimal. The architecture independence code should be shared across different platforms. That concept is called a 'Virtual machine' (Torvalds 1997). If the virtual machine specifies too much it may not efficiently map onto some hardware. According to Torvalds (1997), a virtual machine has two layers of abstraction.

1. The layer between kernel to user application
2. The layer between the hardware to the kernel

Many strategies have been adopted to allow portability in Operating Systems. One is defining system call interface standards such as Portable Operating System Interface (POSIX). Adhering to POSIX is important in designing and implementing operating systems in which the underlying hardware infrastructure may change over the operating system's life cycle. For example POSIX 1003.1b provides the standards for fixed-priority preemptive scheduling (Obenland 2000). The operating system has to have 32 priority levels to be compatible with POSIX. It defines three types of scheduling policies for processes that are on the same priority level; `SCHED_FIFO`, `SCHED_RR`, and `SCHED_OTHER`. `SCHED_FIFO` implements first come first serve base scheduling while `SCHED_RR` implements the round robin and `SCHED_OTHER` for architecture dependent scheduling policies. To be able to be portable across different platforms, usage of `SCHED_OTHER` should be limited (Obenland 2000).

2.1.3 POSIX Standards

As different CPU architectures emerged, a proper standard between OS and CPU as well as OS and user applications was needed. Thus, POSIX was introduced by IEEE as a series of standards in the early 1980s (Tanenbaum & Woodhull 2005). Even though the applications and platforms have evolved dramatically over time, the standards have not changed much (Atlidakis et al. 2016). Therefore it is required to analyze whether adhering to POSIX is efficient in modern computing systems. A study by Atlidakis et al. (2016) has given insights on which abstractions provided by POSIX worked well with current computing systems and whether it was missing any abstractions needed by modern systems by doing a static and dynamic analysis on usage of POSIX in Ubuntu, Android, and OS X.

However modern user applications tend to rely on high-level frameworks and libraries for IPC and thread management (Atlidakis et al. 2016). Those high-level frameworks may have been implemented using POSIX standard APIs but it is abstracted from the user applications by providing a more layered interface. Later several extensions APIs were written to address missing interfaces of POSIX for several functionalities such as IPC, graphics, and sound like 'ioctl' for device-specific input/output operations (Tanenbaum & Woodhull 2005). The study showed that Android and Ubuntu were not implementing all the POSIX interfaces but OS X has implemented all according to the POSIX standards 2013.

2.1.4 First Generation Microkernels

The idea of the earliest microkernels was born at Carnegie Mellon University (CMU) by Richard Rashid and Avie Tevanian; The Mach (Rashid et al. 1989). However, not all the versions of Mach are microkernels. An early version of Mach was not a success because it was originally developed as a replacement for the monolithic UNIX, which inherited some UNIX like ideas. However, according to Miao (2011) Mach's external pager is considered a significant breakthrough in the conceptual development of microkernel architecture. According to Miao (2011), due to the high overhead in IPC, the first generation of microkernels was not very successful; however, it concluded with promising ideas for true microkernels.

2.1.5 Second Generation Microkernels

In 1995 Liedtke (1995) showed that the inefficiency and inflexibility of microkernels are not because of the design but because of its inefficient implementation. To enable security measures, it is crucial to provide an abstraction to physical address space, otherwise, one user process could result in using memory addresses that belong to another process. Liedtke (1995) suggested leaving the memory management and paging outside the kernel by recursively constructing address spaces outside the kernel. The microkernel could provide three operations **MAP**, **GRANT**, and **FLUSH**. Pages that are accessible within an address space can be granted by the owner of that particular address space to another address space (Liedtke 1995).

After the grant, the granter can no longer access that page, only the grantee can. The **MAP** is also similar to **GRANT** however, after mapping, both the mapper and recipient can access the page. **FLUSH** refers to removing access to mapped pages of address space. After flushing, the flusher and its successors can access that page. According to Liedtke (1995) this was completely safe because when the mapping happens the recipient agrees on such access removal. This further extended to remove only a subset of rights to the page when flushing.

However, according to the theory out of three operations, **GRANT** and **MAP** should need the support of IPC to get the agreement between the granter/mapper and the recipient. In that case, the memory management and paging also should contribute to the performance bottleneck of microkernel which is IPC. However Liedtke (1995)'s L3 kernel allowed passing messages directly between processes without considering the authentication and security. Thus, it has to be implemented in the servers on user space. It showed a significant improvement compared to first-generation Mach with 10 times performance improvement in system calls (Miao 2011).

2.1.6 Current Generation Microkernels

Lessons learned from the implementation of L3 microkernels contributed valuable insights towards the development of L4 microkernels. According to Elphinstone & Heiser (2013), Liedtke has demonstrated that L4's IPC could be implemented in a way that it performs 10-20 times faster than other existing microkernels. Traditionally, when a message or data needs to be exchanged between two processes it copies the data from the address space of the sender into a temporary buffer in the kernel and then copies it again from the buffer in the kernel to the address space of the receiver (Miao 2011). Later, this was further improved by simplifying message structures in IPC and focusing on synchronous IPC which reduces the buffering and copying cost in the kernel (Mutia 2010).

However, in a multitasking environment, this can lead to significant inefficiency as other processes are idle while one process is waiting for a response. According to Heiser & Elphinstone (2016), even though to overcome this, multiple threads can be employed, it still adds unnecessary complexity in terms of resource sharing and race conditions for systems that run single-threaded tasks. This issue was addressed by adding asynchronous support in the seL4 microkernel. seL4 introduced basic form of asynchronous IPC known as asynchronous notification (Heiser & Elphinstone 2016). This was further refined into Asynchronous Endpoints (AEPs). AEP allowed non-blocking messages to the receiver and the receiver has the flexibility to wait or poll (Mutia 2010). This involved having a virtual register mechanism and mapping those virtual registers into physical registers. Over time, the benefit of in-register message passing faded away and legacy POSIX read write were replaced by pass-by-reference using shared memory (Elphinstone & Heiser 2013). This is also called 'zero-copy' message transferring because messages can be passed from one address space to another without physically copying them (Heiser & Elphinstone 2016).

2.1.7 seL4 Microkernel

The seL4 microkernel represents a significant advancement in operating system design, prioritizing security, reliability, and formal correctness. As a high-assurance microkernel, seL4 provides a minimal set of abstractions such as threads, virtual address spaces, and inter-process communication (IPC) upon which higher-level system components can be built. Its minimalistic design, comprising around 10,000 lines of C code depending on the architecture and configuration, contributes directly to its suitability and verifiability. This makes seL4 a strong candidate for systems where correctness and security are paramount.

A defining characteristic of seL4 is its formal verification, which ensures that the implementation conforms to its high-level specification with mathematical certainty. As stated by Klein et al. (2009), seL4 is the first general-purpose operating system kernel to undergo complete formal verification, meaning that it is provably free of implementation bugs within the verified scope. This level of assurance is particularly important in safety-critical domains such as aerospace, automotive, and medical systems, where unintended behaviors can have severe consequences. Formal verification also guarantees critical security properties such as integrity and confidentiality, making seL4 well-suited for use in trusted computing bases and isolation-enforced system architectures.

Another key feature of seL4 is its capability-based access control model. In contrast to traditional access control mechanisms that are often discretionary or role-based, seL4 employs a strict, fine-grained approach using unforgeable tokens known as capabilities (The seL4 Authors and Contributors 2024). These capabilities are the sole means by which threads can access kernel-managed resources, ensuring that access rights are explicitly granted and strictly enforced. Capabilities are stored in a hierarchical structure known as a capability space (CSpace), allowing for efficient delegation, revocation, and control of authority. This model provides strong guarantees of resource encapsulation and isolation, which is essential for building secure systems composed of mutually distrusting components.

In addition to these architectural and security strengths, seL4 is capable of functioning as a type-1 hypervisor, running directly on hardware without the need for a host operating system. This bare-metal operation mode allows seL4 to manage hardware resources directly, enabling the deployment of isolated user-level systems or virtualized environments with minimal overhead. Its support for virtualization, particularly on ARM and x86 platforms, makes it suitable for scenarios where lightweight yet secure partitioning of execution environments is needed such as in embedded systems or edge computing infrastructures.

2.2 Evolution of Unikernels

Unikernels represent a paradigm shift in operating system architecture, emphasizing minimalism, security, and performance. They are specialized, single-address-

space machine images constructed by compiling application code alongside only the necessary operating system components. This approach results in lightweight, efficient, and secure execution environments tailored for specific applications.

The concept of unikernels emerged as a response to the increasing complexity and overhead associated with traditional operating systems and virtualization technologies. Early efforts aimed to streamline application deployment by eliminating unnecessary OS components, leading to faster boot times, reduced attack surfaces, and improved resource utilization. Over time, unikernels have evolved to support a variety of use cases, from cloud computing to edge and IoT deployments.

Unikernels can be broadly categorized into two types based on their development approach.

2.2.1 Minimized Monolithic Kernels

One category of unikernels is derived by minimizing existing monolithic kernels, reducing them to include only the components necessary for running a specific application. This approach allows developers to reuse mature, well-supported kernel code while eliminating the complexity and overhead typically associated with general-purpose operating systems. Two prominent examples of this category are OSv and Unikernel Linux (UKL) (Raza et al. 2023). OSv is designed to support single-application workloads in cloud environments and is based on the Linux kernel. It aims to provide near-native performance while retaining compatibility with standard POSIX interfaces and existing Linux-based applications. Similarly, Unikernel Linux (UKL) transforms the traditional Linux kernel into a unikernel environment without forking the kernel itself. This ensures that UKL maintains compatibility with the broader Linux ecosystem, making it possible to leverage existing tools, drivers, and development workflows in a unikernel context.

A particularly significant unikernel framework within this category is Rumprun, which plays a central role in this research. Rumprun is based on the rump kernel components of NetBSD and enables the construction of POSIX-compliant unikernels by isolating and reusing specific kernel subsystems as modular, library-like components. One of the core strengths of Rumprun lies in its ability to run unmodified or minimally modified existing applications as unikernels. This is achieved by linking the application with the necessary rump kernel libraries at compile time, resulting in a self-contained and single-purpose unikernel binary. The modular design of rump kernels allows developers to include only what is required by the application, reducing the attack surface and resource usage. Rumprun supports a wide range of features including networking, file systems, and threading, which further facilitates the development of practical unikernel systems. Its emphasis on portability and standards compliance makes it an ideal choice for research into combining lightweight virtualization with secure isolation.

By building upon an existing and reliable operating system like NetBSD, Rumprun strikes a balance between legacy compatibility and the unikernel phi-

losophy of minimalism and performance. It abstracts the complexities of kernel development away from the application developer, enabling them to focus on the application logic while still benefiting from the performance and isolation advantages of unikernels. These features make Rumprun particularly attractive in research scenarios that explore hybrid designs such as the integration of unikernels with microkernels like seL4 to enhance system isolation without incurring traditional virtualization overheads.

2.2.2 Clean-Slate Unikernels

A distinct category of unikernels comprises those designed and implemented from the ground up, specifically tailored for certain programming languages or application domains. Unlike approaches that minimize existing kernels, these systems are built with unikernel principles at their core, allowing developers to tightly couple application logic with only the essential operating system functionality. This clean-slate design enables precise control over performance, resource usage, and security, and often leads to highly optimized systems with minimal attack surfaces. A notable example is MirageOS, a library operating system written entirely in OCaml. It compiles applications into unikernel images, offering strong static type safety and eliminating many common classes of runtime errors. By leveraging OCaml’s advanced type system and modularity, MirageOS enables the development of secure, high-assurance applications suited for deployment in both cloud and embedded environments.

Other examples include HaLVM and IncludeOS, which each target specific language ecosystems and deployment goals. HaLVM (Haskell Lightweight Virtual Machine) facilitates the creation of unikernels written in Haskell, a functional programming language known for its emphasis on purity and formal reasoning. HaLVM builds upon the Xen hypervisor to execute Haskell-based applications as isolated unikernel instances, making it a valuable platform for research in verified systems and high-assurance computing. In contrast, IncludeOS is a minimal unikernel built in C++ with a focus on ease of use and cloud-native workloads. It allows developers to compile services directly into self-contained binaries that can boot on virtual hardware without a traditional operating system. IncludeOS offers support for common runtime features such as networking and system calls, while maintaining a compact and efficient footprint. Together, these clean-slate unikernels highlight the flexibility of the unikernel model and its applicability across diverse programming paradigms and use cases.

2.3 IPC

Since in microkernels, most of the services moved to the user space from kernel space, the context has to be switched from user mode to kernel mode to perform privileged tasks on behalf of the service (Tanenbaum & Woodhull 2005). Therefore the communication between processes cannot be done directly unlike in monolithic

kernels. Thus, message-passing mechanisms were introduced (Mutia 2010). IPC was implemented in both monolithic and microkernels in many different ways.

2.3.1 Inter Process Communication in Monolithic Kernels

Monolithic kernels mainly use signals, named pipes, sockets, and UNIX V IPC to communicate between processes.

i signals

According to Mutia (2010), this is considered as an earlier mechanism used in IPC. The number of signals allowed was limited and was around 30 (Mutia 2010). Even though the execution time is minimal introducing new signals is not straightforward. The logic to handle the signal can be defined in each process accordingly except for **SIGSTOP** and **SIGKILL**. When handling the signals it follows a self-first approach. First, it will look for any logic within the process if no logic is defined within the process the kernel takes a default action for the signal (Mutia 2010).

ii pipes

Pipes are implemented using two file structures. Each structure contains two vectors. One for reading and one for writing (Mutia 2010). If a process wants to write some data it first checks whether the space is enough and it was not locked by another write or read process. Same thing applies when reading from a pipe as well. Reading additionally allows non-blocking reads as part of asynchronous IPC (Mutia 2010). If the pipe is locked or does not contain any data, an error will be returned immediately to the caller process. There is a special version of pipe called named-pipes which follows First In First Out (FIFO) principle. Unlike in traditional pipes 'named-pipes' uses entities defined in the file system.

iii sockets

While FIFO pipes can only be created as byte streams, sockets allow a sequence of data-grams to be created (Mutia 2010). The advantage of this is it enhances independence and error detection at the packet level. A corruption in one data-gram does not affect other data-grams.

iv UNIX V IPC

UNIX uses three mechanisms in IPC which are shared memory, semaphores and message queues (Mutia 2010). Common authentication mechanisms have been implemented in all three of them in which a process can use system calls to pass a reference to the kernel to access the resources. Linux as a monolithic kernel based OS uses a vector of **message queues** (Mutia 2010). Each entry points to a data structure that describes the relevant message queue. See Figure 2.1. This message queue data structure consists of a permission structure that contains the metadata relevant to permission, a pointer that points to the actual message, and two queues; one as the writer's queue and another one as the reader's queue.

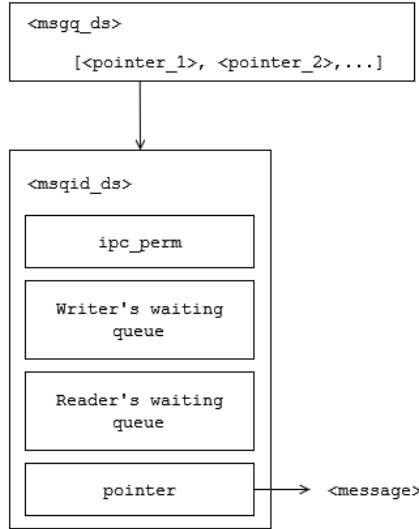


Figure 2.1: Message queue basic structure

The length of the message can be restricted to a certain limit, and by that, the processes that perform write operations have to wait in the writer's queue until a read process reads at least one message from the queue. As mentioned by Mutia (2010) Readers can read messages in FIFO manner or apply filters in which the reader is interested in a particular topic.

Semaphores, on the other hand, works similarly but uses two atomic instructions called 'test' and 'set' (Mutia 2010). This is mainly used in accessing resources. Once a process finishes using a resource it releases it by incrementing the semaphore so that another waiting process gets the signal that the resource is available.

Shared memory can be used when processes need to quickly share a large number of data. The same physical address gets mapped into different or the same virtual addresses in each process's address spaces.

2.3.2 Inter Process Communication in Microkernels

Because of having many unprivileged services in user space, microkernels have to use IPC each time two processes need to communicate with each other (Tanenbaum & Woodhull 2005). The kernel handles the bare minimum possible in IPC such as message passing. These messages are exchanged through message registers and each such message consists of a message tag (Mutia 2010). First-generation microkernels used buffered message passing while later microkernels such as L3 and L4 allowed unbuffered message passing through pass-by-reference mechanism (Elphinstone & Heiser 2013). OKL4 which is a variant of the L4 microkernel family uses two operations 'send' and 'receive' both of them are synchronous (Mutia 2010). Later asynchronous notification was allowed in the 'send' operation but its usage was limited.

Upon an IPC request, the kernel first checks whether the target process or

thread is capable of getting the message. If the target is waiting for a message then the kernel directly sends the message (Mutia 2010). If the target is not capable of having the message then it checks whether the IPC request is blocking or non-blocking. If the request is blocking it pushes the caller process into the IPC queue of the receiver process. If the request is non-blocking it aborts immediately. In asynchronous message passing, the target process does not need to invoke an IPC receive operation (Mutia 2010). Each thread or process has its own **Acceptor** register which contains the notify flags it is interested in receiving notifications for, and a **NotifyMask** register which specifies what notify flags are allowed to send notification upon receiving a message (Mutia 2010). If the notify flag of the message is presented in the target process' **Acceptor** register and **NotifyMask** register, then the message is sent immediately.

As long as the IPC is involved in the switch between user mode and kernel mode it inherits the main performance bottleneck which is context switching. Many research studies have been done to mitigate the context-switching involvement during an IPC call. However according to Mi et al. (2019), most of them are using some specific architectures such as intel which has special instructions like **VMFUNC** and Extended Page Table (EPT) features. Recent research by Mi et al. (2019) utilized intel's **VMFUNC** to directly switch to the callee's address space in an IPC invocation without context switching (Mi et al. 2019). This allows. However, the use of **VMFUNC** raises another problem of high VM exits even during the usual execution of the operating system. This was solved by implementing another layer beneath the microkernel not to switch between user mode and kernel mode during an IPC invocation (Mi et al. 2019). Skybridge is a novel inter-process communication (IPC) mechanism designed for efficient cross-core data sharing which demonstrates significant performance improvements over traditional IPC methods. See Figure 2.2. The performance evaluation was done using a key-value store with SQLite3.

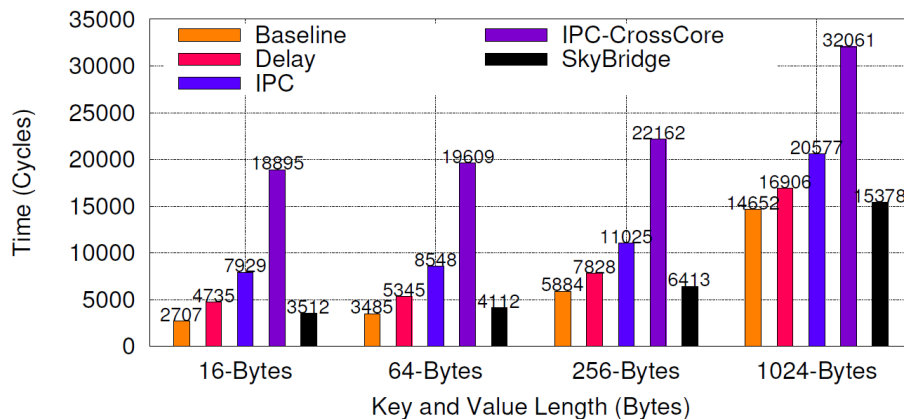


Figure 2.2: Performance comparison of Skybridge (Mi et al. 2019)

Usage of EPT allowed user processes to switch the virtual address space without trapping into the kernel (Mi et al. 2019). Supporting IPC at the user level indeed has issues with security aspects. If some malicious process writes its imple-

mentation of the **VMFUNC** instruction, it might give access to the receiver’s address space. As Mi et al. (2019) mentioned, this was eliminated by scanning each process’s binary to replace any illegal construction of **VMFUNC** instruction.

The main performance bottleneck of microkernels is indeed due to the huge IPC overhead. However, it is not only because it requires context switches each time, partially because of the implementation issues related to many modern CPUs (Mi et al. 2019). Many processors follow multiple levels of caching as L1, L2, L3, and so on. Not only these but also the Translation Look-aside buffer (TLB) get poisoned by IPC related information which also invokes cache and TLB misses each time when memory accessing. This behavior was tested by Mi et al. (2019) by calculating the average IPC time of a null system call and interpreting it in a function call using delay. See Figure 2.2. Each process provides an interface and who can call through that interface using IPC during the registration (Mi et al. 2019). This helps to map the address space of the receiver into the sender’s address space when they want to communicate. when a sender sends an IPC it first saves its state into the stack then invokes **VMFUNC** instruction to use the receiver’s page table in EPT (Mi et al. 2019). Then all the addresses that need to be translated will be translated through the receiver’s page table instead of the sender’s page table (Mi et al. 2019). Since this avoids using VM exits, it reduces the cost of the guest’s physical addresses into the host’s physical addresses.

2.4 Process Isolation

Microkernels need to harmonize isolation and performance to provide efficient functionality. Achieving only the isolation leads microkernels to inefficiencies in performance. According to Mi et al. (2019), a single round trip of an IPC costs around 1500 cycles in seL4. As in Figure 2.3, Gu et al. (2020) demonstrated that seL4 consumes 44% of its time on IPC for an average task. The SkyBridge IPC design by Mi et al. (2019) has optimized it to 400 cycles for a single round trip. However, microkernels have significant progress yet to be achieved compared to traditional monolithic kernels, which cost much fewer cycles for calling kernel components.

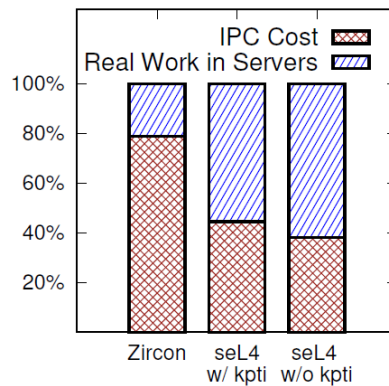


Figure 2.3: IPC overhead in seL4 (Gu et al. 2020)

2.4.1 Domain Isolation

UnderBridge; recent research by Gu et al. (2020) proposed a new mechanism that reduces microkernels’ high IPC overhead. The key idea was to supply an isolated domain environment for user space processes within the kernel itself without moving them outside it and provide efficient cross-domain interactions (Gu et al. 2020). It used Intel’s Protection Key for Userspace (PKU) for the isolation to be efficient even with cross-domain interactions. Even though it is architecture-specific, the idea of isolating user processes in kernel space is brilliant for a microkernel to have a performance boost.

Those processes that run in isolated domains can communicate with other isolated processes and the kernel efficiently without expensive IPC. The number of such isolated domains being limited is also considered and UnderBridge provides server migration (Gu et al. 2020). Each process can decide whether to run in an isolated kernel domain or user space as in traditional microkernels based on how performance-critical it is. Use of PKU enforces having a separate page table for the kernel otherwise unprivileged applications might directly access the memory related to the kernel (Gu et al. 2020). PKU enabled CPUs enforces permission checks when the U/K bit of a page entry is set to ‘U’. With the help of Kernel Page Table Isolation (KPTI), UnderBridge was able to separate pure kernel pages from user-accessible pages of the kernel (Gu et al. 2020). However, the problem was that Memory Protection Key (MPK) only checked permissions for read/write operations leaving the execution (instruction fetching) permission available to any process in the kernel address space. According to Gu et al. (2020), this was somehow solved using a new register Protection Key Rights Register (PKRU) and a new instruction *wrpkr*. Figure 2.4 shows the cost of the new instruction compared to existing instructions.

Instruction	Cost (in cycles)
Indirect Call and Return	24
syscall and sysret	150
Write CR3 (without TLB FLUSH)	226
VMFUNC (with EPT Switching)	146
wrpkr	28

Figure 2.4: Cost comparison of instructions (Gu et al. 2020)

Even though UnderBridge managed to achieve isolation through MPK, a few security issues were still there regarding the execution instruction not being checked for permission. Some compromised user processes could install their page table and get full access to the kernel address space. Even though the pure kernel address space can be separated using KPTI, it could access memory addresses belonging to other processes.

2.4.2 Extended Page Table Switching

Process isolation is not a characteristic only for microkernels. Monolithic kernels also need to isolate user processes even though those reside in the kernel space mainly due to security reasons. Since hardware did not support much for isolation in earlier CPUs, most of the kernels implemented software solutions by minimizing the TCB (Narayanan & Burtsev 2023). Because security attacks are growing nowadays, hardware support for handling isolation increased, resulting in special instructions like `VMFUNC` which was introduced by intel on their Skylake processors (Mi et al. 2019). IPC is possible with `VMFUNC` without the need of the kernel (Narayanan & Burtsev 2023). The execution flow continues in the same thread but uses different page tables after the execution of `VMFUNC` instruction. Since it is using the same thread the next immediate instruction after `VMFUNC` should be valid in the address space of the target process. To ensure that it is valid a trampoline page is mapped to both source and target processes' address spaces (Narayanan & Burtsev 2023).

This page table transition involved saving and restoring the state of the two processes before and after the method invocation. An attacker could bypass this state persistence by scanning the byte sequence to identify functionally similar instructions for `VMFUNC` (Gu et al. 2020). By doing that it is possible to avoid the `VMFUNC` and move to the next instruction which should be resolved using the page table of the target process (Narayanan & Burtsev 2023). This results in overlapping between isolated domains. Thus a proper isolation mechanism should not allow overlapping as well as to construct functionally similar byte sequences to `VMFUNC`. SkyBridge by Mi et al. (2019) has overcome the issue related to functionally similar byte sequence for `VMFUNC` by performing a binary scanning mechanism. However, this might not work where the execution is done using just-in-time compilation.

ARM also introduced memory tagging extensions. MPK by intel also follows tagged memory mechanisms to isolation processes (Narayanan & Burtsev 2023, Blair et al. 2023). Usage of the tagged memory concept initially had its limitations due to the number of possible isolated domains being limited to 16 in most architectures (Mi et al. 2019). Processes can be isolated in the same address space by tagging the pages of the address space with a 4-digit code that specifies the domain that the page belongs to (Narayanan & Burtsev 2023). See Figure 2.5 With 4 bits it had 16 possible isolated domains within a single address space. A workaround has been done to extend the number of isolated domains by virtualization (Gu et al. 2020). However, it was complex as it involved many system calls. A process can access a page only if the protection key rights for user space (`pkru` register) match the page's tag.

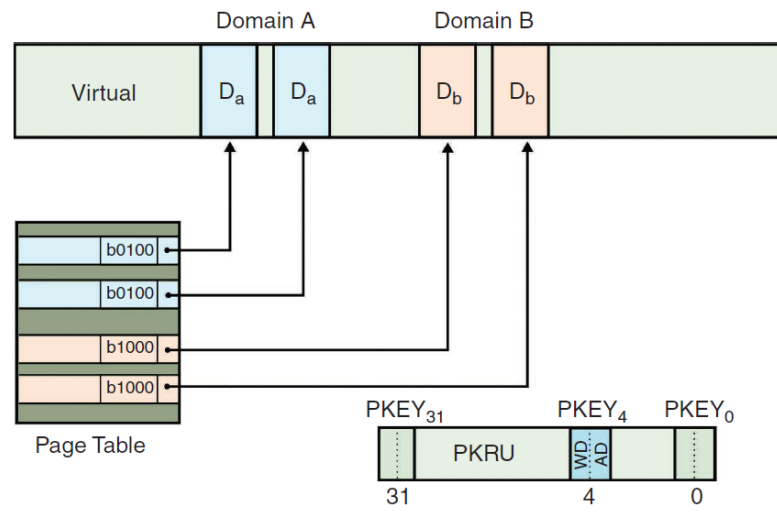


Figure 2.5: Domain isolation with tagged memory (Narayanan & Burtsev 2023)

3 Design

This chapter presents the design considerations and methodologies adopted to investigate isolation properties, utilizing the seL4 microkernel. The focus is placed on the setup of the test environment, the selection and configuration of supporting tools, and the two distinct experimental design approaches undertaken. These design choices serve as the foundation for evaluating both intra-unikernel and inter-unikernel isolation mechanisms, leveraging seL4’s formal guarantees and component architecture.

The test environment was configured on a machine running a 64-bit Ubuntu operating system, equipped with an Intel® Core™ i5-10300H CPU @ 2.50GHz (8 core) and 16GB of RAM. To emulate the deployment and execution of the designed system, QEMU : quick emulator was used. Each emulated instance was allocated specific amounts of memory tailored to the needs of individual experiments, which will be detailed in the corresponding sections. The seL4 microkernel’s toolchain was used extensively, taking advantage of its CMake-based build system with Ninja as the generator to manage and compile build files efficiently. Additionally, benchmarking tools, provided as a separate project by the seL4 ecosystem, were integrated to support performance evaluations during experimentation.

Two core design strategies were explored in this work. The first focused on examining isolation within a single unikernel instance by utilizing the CAMkES (Component Architecture for microkernel-based Embedded Systems) framework. This approach allowed the investigation of fine-grained, internal component separation and communication. The second strategy addressed the broader scenario of isolation across multiple unikernel instances, each independently hosted, with attention to the seL4 mechanisms that enable strict separation between co-existing systems. Together, these approaches provide a comprehensive examination of isolation at both intra- and inter-system levels, forming the basis for the experimental analysis presented in subsequent chapters.

3.1 Setting up the environment

The first step in the process was to set up a simple application on seL4 to confirm that the operating environment was working correctly and that all configurations were properly in place. With the help of the clear and detailed documentation provided by seL4, this initial setup was relatively straightforward. This early experiment acted as a baseline to ensure that the microkernel was running as expected and laid the groundwork for integrating unikernel-based components in later stages.

To maintain a consistent and hardware-independent testing environment, the experiment was carried out using QEMU. This approach allowed for reliable exe-

cution without being affected by hardware-specific variations. By running a basic program on seL4, it was possible to confirm that key functionalities such as process creation, basic input/output operations, and output handling were all working as intended.

3.2 Selecting a Compatible Unikernel

After verifying the functionality, the next step was selecting a suitable unikernel. This unikernel had to meet several criteria to be integrated with seL4.

- It needed to be adaptable to the seL4 microkernel environment
- Since the idea is to use seL4 as a bare metal hypervisor, the selected unikernel should be able to run on bare metal
- Since unikernels rely on minimized OS abstractions, the unikernel should provide sufficient library support and system call compatibility to interface with seL4.

In selecting a suitable unikernel for integration with the seL4 microkernel, several options were initially considered, including Unikraft, MirageOS, IncludeOS, and Rumprun. Rumprun emerged as a strong candidate due to a few key advantages. Firstly, it is developed in C, which aligns well with seL4's C-based implementation, making integration more straightforward at a technical level. Additionally, Rumprun is based on a minimized version of NetBSD, offering a broad range of driver support, a crucial factor given that seL4 itself provides limited native driver functionality. This support enables POSIX-based applications to seamlessly leverage NetBSD libraries without requiring deep modifications to interact with the underlying microkernel.

Another practical advantage was that preliminary work on integrating Rumprun with seL4 had already been initiated by the community. This partial integration provided a valuable starting point, reducing the groundwork needed for adapting Rumprun to the seL4 environment. With Rumprun selected, the next phase involved tailoring it to operate within seL4's unique architecture. This adaptation focused on aligning system call handling and library usage to function within a single address space, as is typical with unikernel designs. These efforts lay the foundation for enabling efficient and secure isolation, which is central to the experiments and evaluations conducted in the later stages of this research.

3.3 Design 1 : Intra-Unikernel Isolation

The first design approach investigates the feasibility and effectiveness of introducing intra-unikernel isolation within a single Rumprun-based application by leveraging the CAMkES component framework on top of the seL4 microkernel.

Traditionally, Rumprun unikernels operate within a single address space, compiling application code and necessary OS components into a single binary. While this model offers performance benefits, it provides limited internal isolation between different parts of the application, which may become a concern in systems requiring strict separation of concerns or modular trust boundaries. This approach seeks to address that limitation by decomposing a Rumprun application into multiple isolated components, each deployed as a separate CAMkES component communicating through formally defined interfaces.

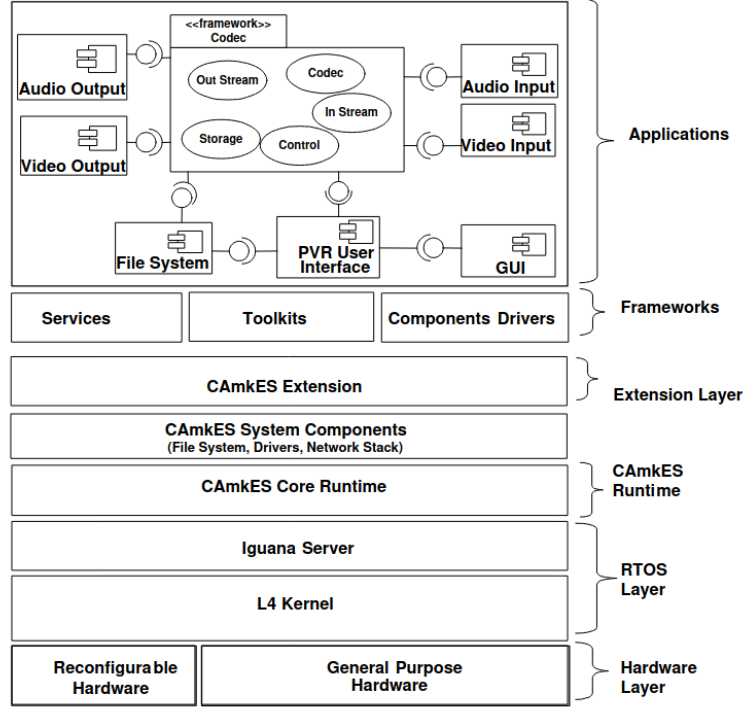


Figure 3.1: The CAMkES layered architecture (Kuz et al. 2007)

Figure 3.1 illustrates the layered architecture of CAMkES. At the foundation is the hardware layer, consisting of the CPU, memory, buses, and peripheral devices. Directly above this is the RTOS layer, which includes the microkernel (seL4 in this case) and an optional supervisory operating system. While components such as device drivers or network stacks can be embedded within this layer, they can also be implemented as isolated CAMkES components residing in higher architectural layers.

The CAMkES core runtime forms the core of the component architecture. It provides essential services and a static execution environment for CAMkES components. In this static model, all component instances are created at system boot time, and connections between components are established at compile time. This approach eliminates runtime overheads such as dynamic binding or inter-process communication (IPC) costs, often replacing them with efficient, direct procedure calls between components. Above the core runtime lies the extension layer, which includes advanced features such as support for dynamic component creation, bind-

ing, and configuration. These features are themselves implemented as CAMkES components and are only included when needed, allowing systems that do not require such flexibility to remain minimal and efficient. At the top are frameworks and user-defined components, which leverage the lower layers to form complete applications tailored to specific domains. This layered approach ensures a clear separation of concerns, promoting modularity and minimizing resource usage for systems with strict performance and memory constraints.

A key objective of this design is to assess the extent to which seL4’s capability-based access control can enforce meaningful isolation between components of a single application. By confining capabilities and explicitly controlling communication, it becomes possible to limit the privileges of each component, thus minimizing the potential impact of a compromised module. This approach allows for the evaluation of fine-grained isolation policies within the boundaries of a single application context and provides insights into how seL4’s security model applies even when components share a logical application space.

3.4 Design 2 : Inter-Unikernel Isolation

The second design approach focuses on enabling true multi-unikernel isolation by modifying the behavior of the root task within the existing `rumprun-seL4-demoapps` framework. In the current implementation of this repository, the seL4 kernel is used to boot into a single user-level Rumprun application. Upon initialization, the kernel starts the root task, which is responsible for configuring the environment, loading capabilities, and launching the unikernel. This process is currently designed to support only one application at a time, effectively making the system single-tenant despite the underlying microkernel’s strong isolation capabilities.

To overcome this limitation, this approach proposes extending and restructuring the root task into a lightweight cooperative scheduler capable of managing multiple Rumprun applications concurrently. Rather than launching a single predefined unikernel application immediately after system boot, the root task would instead instantiate a rumprun application as user space processes in the same environment. Each application would be initialized with its own capability space (CSpace), virtual address space (VSpace), and thread control block (TCB), allowing them to run in isolated user-level processes, fully supported by seL4’s formally verified isolation mechanisms.

A central element of this design is the integration of a cooperative scheduling mechanism within the root task. Once seL4 completes its boot sequence and control is transferred to the root task, the system will no longer immediately transfer control to a single application. Instead, the root task would first initialize a runtime environment containing metadata about multiple runnable applications. It would then select one of these applications to run based on a basic scheduling policy and activate it by mapping its address space and capabilities, effectively

switching the current execution context. When the selected application completes or voluntarily gives control to the root task, it then selects the next application to run. To support preemption and avoid CPU monopolization by a single application, timer interrupts can be configured via seL4’s notification objects. These interrupts can trigger the root task to regain control and perform a context switch, thereby simulating time-sliced multitasking behavior.

This design leverages seL4’s robust resource isolation features to ensure that all Rumprun processes run within the same environment, potentially sharing resources like timers and serial devices, but in a controlled and restricted manner using seL4 capabilities. The ability to isolate multiple unikernels as separate processes not only enhances fault isolation and security but also enables more efficient use of system resources in multi-tenant or edge computing scenarios. Unlike monolithic virtualization solutions, this model avoids the overhead of full virtual machines while still offering separation at the process level.

However, realizing this design entails several non-trivial challenges. The foremost among them is the need to significantly modify the default behavior of the root task. This includes implementing a custom runtime capable of handling process metadata, maintaining control flow, managing IPC between the scheduler and applications, and ensuring proper cleanup of resources upon application termination. Moreover, care must be taken when handling timers and serial devices, preventing resource contention between processes. Another complexity lies in handling blocking system calls such as `sleep()`, within the applications. Since Rumprun applications may rely on POSIX-like abstractions, the root task must be able to distinguish between voluntary blocking behavior and preemptive scheduling opportunities, all while remaining responsive to external interrupts and seL4 notifications.

Despite these challenges, this approach offers a powerful path forward for investigating secure multi-tenant execution of unikernels atop seL4. It brings together the performance benefits of unikernels with the formal isolation guarantees of seL4 while exploring the design space between static, single-purpose systems and dynamic, multi-application platforms. Ultimately, this design serves as a foundation for future systems where lightweight applications must co-exist securely within a shared yet tightly controlled operating environment.

4 Implementation

4.1 Test Environment

The evaluation and experimentation were carried out on a host machine running Ubuntu 22.04, with 512MB of memory allocated to the QEMU virtual machine. The target architecture for the experiments was x86_64, and both gcc and g++ compilers used were version 10. This work focused on the seL4 microkernel and its supporting infrastructure, using two main repositories.

- Design 1: seL4/camkes-manifest
- Design 2: seL4/rumprun-sel4-demoapps

Implementations of both designs utilized the cmake build system and ninja generator to generate build files and the scripts to perform the simulation on QEMU.

4.2 Implementation of design 1

The first implementation involved integrating a Rumprun application within the CAmkES (Component Architecture for Microkernel-based Embedded Systems) framework. This approach explores how lightweight unikernel applications can be run atop the seL4 microkernel, utilizing the CAmkES environment for modular, component-based systems.

4.2.1 Obtaining the Source

The process began by obtaining the necessary CAmkES source code using Google's **repo** tool, which simplifies managing multiple Git repositories. By initializing the manifest and syncing with the remote repositories, the complete source tree was downloaded, including the essential tools and modules.

4.2.2 Preparing Rumprun Dependencies

To enable Rumprun application support, the submodules under `/tools/rumprun` needed to be correctly initialized. These submodules, `buildrump.sh` and `src-netbsd` are crucial for building and running Rumprun-based applications. The `buildrump.sh` script automates the build process for the NetBSD components required by Rumprun, and `src-netbsd` contains the source code for the NetBSD userspace.

4.2.3 Building a Rumprun Application

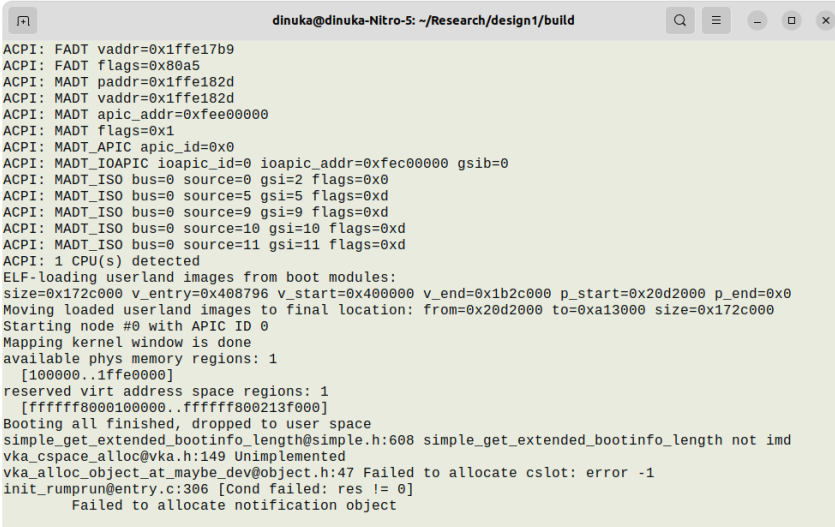
To verify that Rumprun applications function correctly within CAMkES, an existing application `rumprun_hello` was selected as a test case. A new build directory was created to isolate the build artifacts, and the build system was initialized using the `init-build.sh` script. Here the platform used was `x86_64`.

```
../init-build.sh -DPLATFORM=x86_64 -DCAMKES_APP=rumprun_hello -  
DSIMULATION=1  
  
ninja
```

The build process generated the necessary configuration files and a script to simulate the application using QEMU. This step successfully produced the required binaries and execution environment.

4.2.4 Encountering Runtime Issues

However, when executing the generated simulation script, a runtime error was encountered as shown in Figure 4.1.



```
dinuka@dinuka-Nitro-5: ~/Research/design1/build  
ACPI: FADT vaddr=0x1ffe17b9  
ACPI: FADT flags=0x80a5  
ACPI: MADT paddr=0x1ffe182d  
ACPI: MADT vaddr=0x1ffe182d  
ACPI: MADT apic_addr=0xfec00000  
ACPI: MADT flags=0x1  
ACPI: MADT_APIC apic_id=0x0  
ACPI: MADT_IOAPIC ioapic_id=0 ioapic_addr=0xfec00000 gsib=0  
ACPI: MADT_ISO bus=0 source=0 gsi=2 flags=0x0  
ACPI: MADT_ISO bus=0 source=5 gsi=5 flags=0xd  
ACPI: MADT_ISO bus=0 source=9 gsi=9 flags=0xd  
ACPI: MADT_ISO bus=0 source=10 gsi=10 flags=0xd  
ACPI: MADT_ISO bus=0 source=11 gsi=11 flags=0xd  
ACPI: 1 CPU(s) detected  
ELF-loading userland images from boot modules:  
size=0x172c000 v_entry=0x408796 v_start=0x400000 v_end=0x1b2c000 p_start=0x20d2000 p_end=0x0  
Moving loaded userland images to final location: from=0x20d2000 to=0xa13000 size=0x172c000  
Starting node #0 with APIC ID 0  
Mapping kernel window is done  
available phys memory regions: 1  
[100000..1ffe0000]  
reserved virt address space regions: 1  
[ffffff8000100000..ffffff800213f000]  
Booting all finished, dropped to user space  
simple_get_extended_bootinfo_length@simple.h:608 simple_get_extended_bootinfo_length not imd  
vka_cspace_alloc@vka.h:149 Unimplemented  
vka_alloc_object_at_maybe_dev@object.h:47 Failed to allocate cslot: error -1  
init_rumprun@entry.c:306 [Cond failed: res != 0]  
Failed to allocate notification object
```

Figure 4.1: Issue with TLS bound

The error pointed to a failure in allocating required kernel objects during the startup phase of the Rumprun application. Specifically, the `simple_get_extended_bootinfo_length` function, critical for obtaining extended boot information in seL4, was reported as not implemented.

4.2.5 Diagnosing the Issue

To determine whether the problem was isolated to Rumprun applications or indicative of a broader issue within the CAMkES framework, a different non-Rumprun-

based application was compiled and executed. This application ran successfully, ruling out a general CAmkES or seL4 malfunction and confirming that the issue was specific to Rumprun integration.

Upon deeper inspection, it was discovered that although `simple_get_extended_bootinfo_length` does have a valid implementation in the source code, the runtime could not reach this function. This led to investigating how the Rumprun application image was being packaged.

4.2.6 Identifying the Root Cause

The root cause appeared to lie in how the seL4 runtime calculated the TLS (Thread Local Storage) region size. During execution, seL4 calculates the TLS bounds using information from the ELF file headers. However, in this case, the calculated size was smaller than expected. When Rumprun attempted to set the `sel4_ipc_buffer`, the calculated offset exceeded the allocated TLS region. As seL4 enforces strict memory bounds, this write operation was silently ignored, leading to a failure in setting up critical runtime components.

As a result, the runtime failed to resolve `simple_get_extended_bootinfo_length`, even though it was present in the compiled binary. This situation ultimately caused the Rumprun application to fail during initialization, rendering this implementation unusable.

Due to the above limitations and misalignment in runtime memory management, implementation 1 was unsuccessful. The issue highlights the complexity of integrating unikernel-based applications with microkernel architectures, especially when low-level runtime assumptions such as TLS bounds are violated. However, this failure also demonstrates a positive aspect of the seL4 microkernel:- its strict enforcement of memory boundaries. When the Rumprun application attempted to access a memory region outside the calculated TLS bounds, seL4 prevented the operation entirely and effectively and the process was unaware the region even existed. This behavior underscores the robustness of seL4's memory protection mechanisms, ensuring safety and isolation by design. A more tailored packaging strategy or runtime patching may be necessary to resolve this incompatibility in future iterations.

4.3 Implementation of design 2

Design 2 approached the problem differently by leveraging the `rumprun-sel4-demoapps` repository. This setup eliminates the use of CAmkES and instead treats Rumprun applications as standalone user-level processes. Isolation here is achieved at the process level, rather than via component boundaries.

In this design, the build system generates two images: one for the seL4 kernel and another for the user-level processes. The user image contains the root task (analogous to the `init` process in UNIX), the Rumprun application binaries, and the necessary NetBSD libraries.

Figure 4.2 shows the structure of `rumprun-sel4-demoapps`,

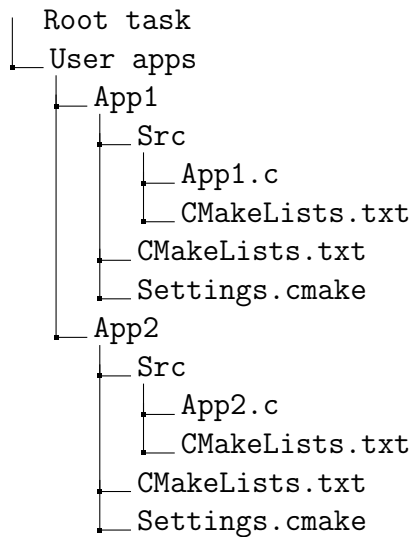


Figure 4.2: Project directory structure

4.3.1 Root Task

The `init` function is the first step executed during the `roottask`'s setup. It begins by retrieving the boot information provided by the seL4 microkernel using the `platsupport_get_bootinfo()` call. This information contains crucial details about the system's initial state, such as capability slots and memory regions. The thread running the root task is then labeled as "roottask" for easier debugging and visualization through the `NAME_THREAD` macro. Finally, the system environment (`env`) is initialized through `init_env`, setting up critical subsystems like memory allocation, capability management, virtual memory space (`vspace`) management, and timers.

The `init_env` function is responsible for setting up the execution environment, a critical environment objects that consolidates allocators, vspace managers, timers, and I/O subsystems. First, it statically reserves memory to manage the `musllibc` heap area without risking overwrites. The allocator is initialized, with a static pool for memory management, ensuring immediate dynamic resource allocation. The VKA (Virtual Kernel Allocator) interface is constructed next, providing a convenient abstraction over allocation manager for kernel resource management tasks like allocating capabilities.

The virtual memory setup begins by reserving key memory regions to prevent accidental reuse, including space for dynamic memory allocation and a larger pool for future system allocations. This ensures that applications like Rumprun have a stable environment for managing memory. Following this, a notification object is allocated to handle asynchronous events like timer and serial interrupts. Essential I/O subsystems are then initialized to support memory allocation, device mapping, and interrupt handling, providing a modular and portable interface for interacting with hardware. These steps collectively prepare the system for running complex unikernel-based applications within the seL4 environment.

Next, it creates a communication endpoint for inter-process communication between the main thread and other system components. If the system is configured in mixed-criticality mode, it also prepares a reply object for managing reply capabilities. A serial server thread is then launched to handle asynchronous serial I/O and an IRQ capability is allocated to handle interrupts from the serial interface.

Following this, a local timer is initialized and linked to the current thread to generate periodic interrupts. A time manager is set up to handle time allocation for Rump processes, ensuring proper integration with the timer infrastructure. The system then boosts the main thread's priority to the highest level to give it scheduling precedence. To distinguish incoming serial interrupts, a badge is applied to the notification object, which is then bound to the serial IRQ handler.

With these core components ready, the system may create an idle thread and optionally a CPU-intensive thread to simulate system load and assess scheduler behavior. Once all this groundwork is done, the Rump kernel user process is loaded. The system inspects the CPIO archive that holds binary payloads, extracts metadata such as file names and sizes, and prints this information for debugging. Finally, it logs the start of the application and launches the specified Rump process, transitioning from setup to execution.

The process begins by preparing to launch a new user-level application, typically a Rumprun binary, within the seL4 microkernel environment. It first allocates memory for initialization data and sets up shared memory for standard input, output, and error streams to enable terminal communication. The new process is assigned a high priority, just below the maximum to balance responsiveness with fairness to other critical threads like serial I/O handlers. A notification object is created to allow the process to receive timer events. The kernel provides the process with a uniquely badged endpoint for handling communication, faults, and RPCs. Capabilities granting access to key system services such as I/O, IRQ, timers, and scheduler control are copied into the new process. In seL4, capabilities are used to control access to kernel objects. Each capability defines a combination of permissions, where up to four types of access rights can be assigned. These rights are represented as a 4-bit tuple in the form (a, b, c, d). The bits correspond to specific permissions as below.

- The first bit (a) set to 1 indicates that the capability allows grant reply, which permits the holder to send a reply to a previously received message.
- The second bit (b) set to 1 indicates grant permission, allowing the holder to forward capabilities to another component.
- The third bit (c) set to 1 indicates read access to the resource.
- The fourth bit (d) set to 1 indicates write access to the resource.

There are nine commonly used capability configurations available in seL4.

<code>seL4_ReadWrite</code>	<code>(0, 0, 1, 1)</code>
<code>seL4_AllRights</code>	<code>(1, 1, 1, 1)</code>
<code>seL4_CanRead</code>	<code>(0, 0, 1, 0)</code>
<code>seL4_CanWrite</code>	<code>(0, 0, 0, 1)</code>
<code>seL4_CanGrant</code>	<code>(0, 1, 0, 0)</code>
<code>seL4_CanGrantReply</code>	<code>(1, 0, 0, 0)</code>
<code>seL4_NoWrite</code>	<code>(1, 1, 1, 0)</code>
<code>seL4_NoRead</code>	<code>(1, 1, 0, 1)</code>
<code>seL4_NoRights</code>	<code>(0, 0, 0, 0)</code>

It's also granted access to untyped memory and device memory, which are vital in seL4's capability-based resource model. The standard I/O memory is mapped into the child process, and an exclusive endpoint is created and passed to it for initialization. Once the setup is complete, the child process is spawned, and the initialization data is sent over.

After launching, the parent enters a loop that waits for the process to finish, crash, or make requests via RPC. Inside this loop, it waits for messages, distinguishing between replies to previous calls, new requests, or interrupts. If the message is an RPC reply, it processes the response. If it's a new message, it identifies the source using the message's badge. Timer messages are handled with a dedicated timer function, faults trigger logging and debugging routines, and unknown messages result in a failure flag. Interrupts are also handled: serial input is captured and written to the process's buffer, standard output is flushed to display what the app printed, and timer interrupts update the time manager. This loop continues until the process exits, either normally or due to an error. Finally, the result is returned to indicate the outcome of the application's execution.

By default, this setup supports running only one Rumprun application. To enable support for multiple applications, a cooperative scheduler was introduced. Each process is tracked in a `rump_env` structure, which includes fields for the process's binary name, its execution state, and the current active process index.

```
env.processes[0].bin_name = "app1.bin";
env.processes[0].state = PROCESS_RUNNABLE;
env.processes[1].bin_name = "app2.bin";
env.processes[1].state = PROCESS_RUNNABLE;
env.current_process_index = 0;
```

The scheduler attempts to switch between applications when a timer interrupt occurs. When a timer interrupt occurs, the current process is marked as `BLOCKED`, and the scheduler selects the next runnable process.

```

if (label == TIMER_LABEL) {
    rump_process_t *current_process
    = &env.processes[env.current_process_index];
    current_process->state = PROCESS_BLOCKED;
    info = handle_timer_rpc(rump_process, badge, info);
    schedule();
}

```

The timer RPC handler assigns a callback function to be triggered when the timer times out.

```

tm_register_cb(&env.time_manager, type, time, 0, id,
timer_callback, id);

```

When the callback get triggered, the waiting process become again runnable. The scheduler uses a simple loop to find the next process marked as RUNNABLE and invokes it.

```

void schedule() {
    int found = 0;
    while (!found) {
        for (int i = 0; i < N_RUMP_PROCESSES; i++) {
            rump_process_t *p = &env.processes[i];
            if (p->state == PROCESS_RUNNABLE) {
                env.current_process_index = i;
                p->state = PROCESS_RUNNING;
                found = 1;
                run_app();
            }
        }
    }
}

```

When a Rumprun application calls `sleep()`, it doesn't just sit idle and waste CPU cycles. Instead, the call initiates a more efficient and structured process involving inter-process communication and kernel-managed timers within the seL4 microkernel environment. The sleep request triggers a system call that travels down to the kernel-aware runtime layer, where it is handled as a Remote Procedure Call (RPC) directed to the timer server. The system recognizes this as a

timeout request, specifically identified by a designated operation code, and proceeds to extract the desired sleep duration and timeout type from the message registers.

This information is passed to the time manager using a function that registers a timer callback. Along with the sleep duration, the system also passes a reference to the sleeping process and a callback function to be triggered once the timeout expires. The time manager then schedules a timer event, linking it with the process and callback so that when the timer goes off, the callback can resume the process.

When the sleep duration has passed, the timer infrastructure triggers the callback function. This function identifies the process that initiated the sleep call using its ID and sends a signal via a specific kernel-level endpoint. That signal is what unblocks the process, effectively waking it up from its suspended state. At this point, the Rumprun process resumes exactly where it left off after calling `sleep()`, having efficiently paused without consuming unnecessary CPU resources.

4.3.2 User Apps

Although the current configuration supports running only a single application at a time, it is possible to include the binary of a second application within the CPIO archive. The root task can then read this archive and determine which application to execute next based on predefined logic or configuration. To enable this, the application's CMake settings should be configured to always include its binary in the CPIO archive, regardless of whether it is the currently selected app for execution. This approach facilitates greater flexibility and simplifies switching between different user applications without modifying the archive generation process.

```
<<seL4(CPU 0) [decodeCNodeInvocation/182 T0xffffffff801fe1c400  
"roottask" @401bcb]: CNode Copy/Min>
```

Despite these modifications, this design also faced limitations. The root cause was the inability to transfer certain capabilities such as timers across process boundaries. These capabilities are shared resources and must be carefully managed; launching a second application with access to these capabilities was not possible under the current framework as seL4 does not allow the capabilities to be copied or mint as shown in the above message. Furthermore, registering callbacks and handling timeout IDs for multiple processes proved problematic, as the underlying timeout registration mechanism does not support concurrent process callbacks.

5 Results Evaluation

5.1 Qualitative Analysis of Isolation Aspects

Isolation is a fundamental security requirement in microkernel-based systems, particularly in the context of executing untrusted or semi-trusted components. Both of the designs explored in this project: Design 1 using the CAMkES component architecture and Design 2 using multiple rumprun-based applications as user-level processes were aimed at achieving isolation to various extents. Although both designs faced practical implementation challenges that limited their success, seL4’s capability-based architecture provides a strong theoretical foundation for isolation that deserves discussion.

Design 1 attempted to use the CAMkES component-based architecture to isolate different parts of a single rumprun application. The CAMkES framework is built on the principle of statically defined components that interact only through explicitly declared interfaces, with all resource access controlled through capabilities. This design inherently supports fine-grained isolation, as each component can be provisioned with only the capabilities it strictly requires. Theoretically, even within a monolithic application such as one built using rumprun, one could isolate untrusted libraries or subsystems by wrapping them as CAMkES components and strictly controlling inter-component communication via capabilities. However, due to the limitations of rumprun’s linker behavior and its requirement for a unified TLS region issues not compatible with the CAMkES build and linking strategy, this isolation was not realized in practice. Still, the design direction demonstrates how seL4’s capability model can enforce the Principle of Least Privilege even at the component level.

Design 2 took a more coarse-grained approach by running multiple rumprun applications as separate user-level processes, under a common root task. This method aligns more closely with traditional UNIX-like process isolation, where each process is expected to run in its own virtual address space with restricted access to system resources. In seL4, these boundaries are even more strictly defined: each user-level process is given a set of capabilities upon creation, and it cannot access resources or communicate with other processes unless explicitly granted the necessary capabilities.

As shown in Figure 5.1 each Rumprun unikernel instance, upon boot, mounts its virtual file system as shown in the blue-colored output from the Rumprun user-level application. This occurs after the kernel and root task complete their bootstrapping sequence (indicated by the black-colored output). In this setup, each Rumprun process operates with a separate file system namespace, resulting in effective file system isolation. Even though all processes share the same underlying seL4 environment including shared resources like timers and serial devices: seL4’s capability-based access control ensures that these shared resources are used in a controlled and restrictive manner. As a result, multiple Rumprun



```
dinuka@dinuka-Nitro-S: ~/Research/benchmark/build
ELF-loading userland images from boot modules:
size=0x14d0000 v_entry=0x442ff2 v_start=0x400000 v_end=0x18d0000 p_start=0x1f94000 p_end=0x3464000
Moving loaded userland images to final location: from=0x1f94000 to=0xa13000 size=0x14d0000
Starting node #0 with APIC ID 0
Mapping kernel window is done
available phys memory regions: 1
[100000..1ffe0000]
reserved virt address space regions: 1
[ffffff8000100000..ffffff8001ee3000]
Booting all finished, dropped to user space

Rump on seL4
=====

simple_get_extended_bootinfo_length@simple.h:608 simple_get_extended_bootinfo_length not implemented
Copyright (c) 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005,
2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016
The NetBSD Foundation, Inc. All rights reserved.
Copyright (c) 1982, 1986, 1989, 1991, 1993
The Regents of the University of California. All rights reserved.

NetBSD 7.99.34 (RUMP-ROAST)
total memory = 8064 KB
timecounter: Timecounters tick every 10.000 msec
timecounter: Timecounter "clockinterrupt" frequency 100 Hz quality 0
cpu0 at thinair0: rump virtual cpu
root file system type: rumpfs
kern.module.path=/stand/amd64/7.99.34/modules
mainbus0 (root)
timecounter "bmktc" frequency 1000000000 Hz quality 100
mounted tmpfs on /tmp
```

Figure 5.1: File system mount upon startup

instances can run concurrently with strong isolation guarantees, particularly at the file system level. Even though the root task in this design initially lacked the scheduling and capability-passing flexibility required to dynamically manage multiple processes, the seL4 kernel itself guarantees that one process cannot interfere with another unless capabilities have been shared. This represents a formal, enforceable notion of isolation rarely seen in traditional microkernel systems.

In both designs, seL4’s capability-based access control serves as a theoretical guarantee of isolation. Capabilities in seL4 are unforgeable tokens of authority that govern access to all kernel objects, including threads, address spaces, I/O devices, and memory regions. As such, isolation in seL4 is not merely a design-time or policy-level concept; it is backed by the kernel’s enforcement mechanism and has been subject to formal verification. This makes seL4 particularly well-suited for building secure systems where isolation must be guaranteed, even in the presence of malicious or faulty components.

5.2 Performance Benchmarking of seL4

To establish a baseline for evaluating isolation in unikernel-based systems on seL4, we first conducted performance benchmarks on the microkernel itself. These benchmarks aim to characterize seL4’s efficiency and responsiveness, particularly in handling core system operations such as context switching, IPC, and memory management. By understanding seL4’s performance in these critical areas, we can assess its suitability as a foundation for unikernel deployments and compare it against traditional operating systems.

5.2.1 IPC Performance

IPC is a central component of microkernel-based systems like seL4, where the kernel facilitates communication between independent processes. In our IPC benchmarks, we measured the round-trip message-passing latency between two processes. Figure 5.2 shows the results regarding seL4’s one-way IPC performance was obtained as follows in terms of mean number of cycles by considering 8 rounds of benchmarking.

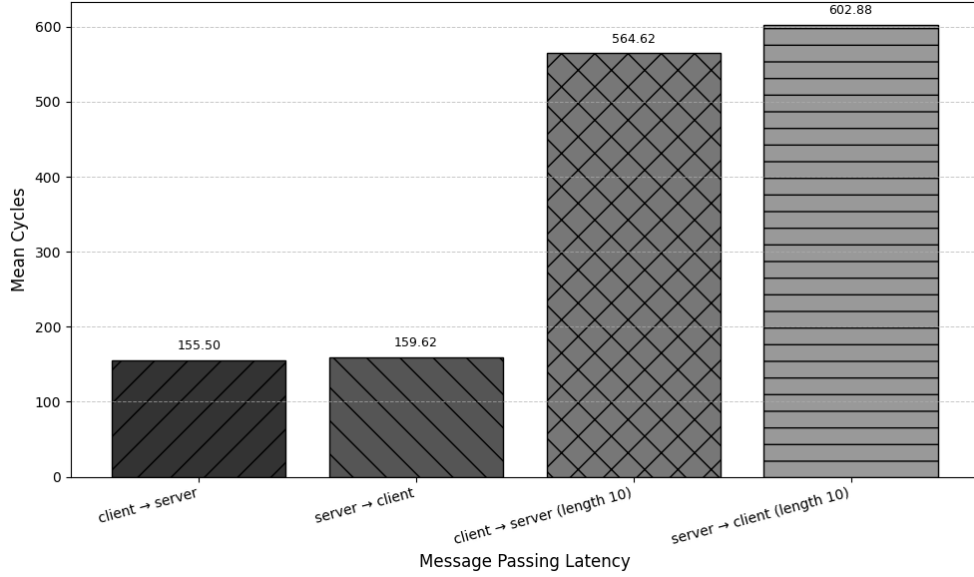


Figure 5.2: Message passing latency

Given the centrality of IPC to seL4’s architecture, this efficient communication is promising for this research, as it suggests that isolated unikernels can communicate without significant overhead, maintaining both performance and security.

5.2.2 Fault overhead

Figure 5.3 shows benchmarking results associated with faults, and measures the time it takes for the system to handle a fault from detection to resolution, including the necessary system overhead for fault handling and recovery. This metric is crucial for understanding how effectively the system can isolate and contain faults without impacting other running components, that aim to ensure the isolation of individual instances.

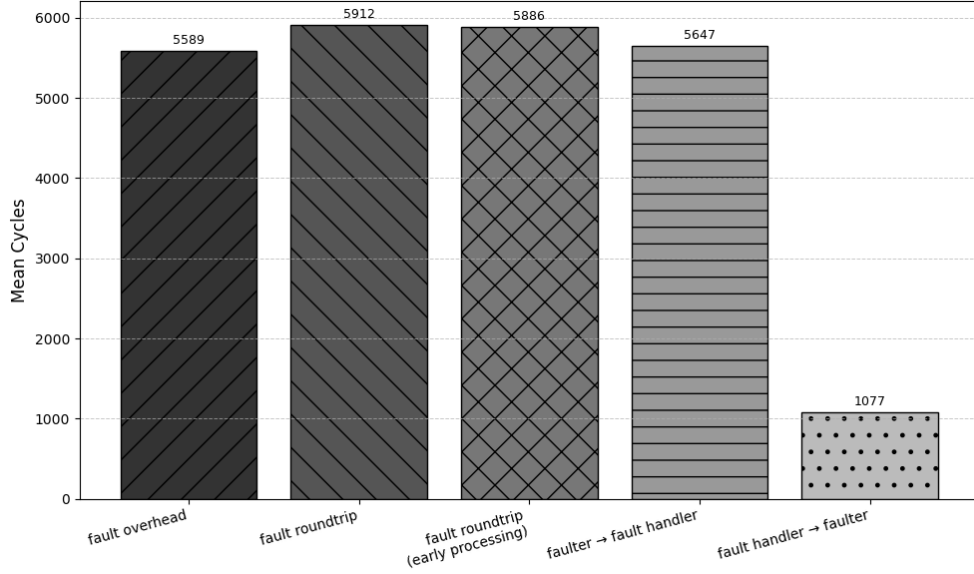


Figure 5.3: Fault overhead

5.2.3 Network throughput

Iperf was used to generate a TCP load, sending packets to the target system while measuring CPU utilization. The results are presented in Figure 5.4. During initial tests, it became evident that CPU utilization was sensitive to the underlying hardware interrupt delivery mechanism. To better understand this effect, we also measured CPU usage under different interrupt delivery models supported in seL4.

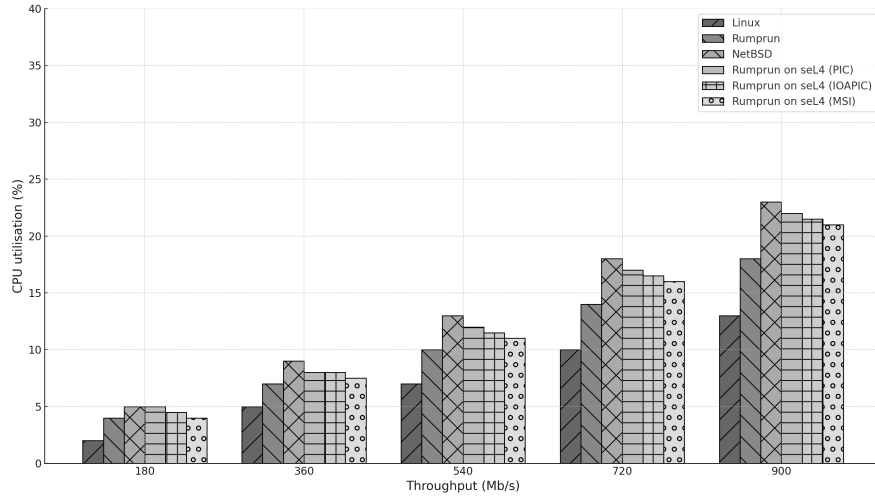


Figure 5.4: CPU utilization (platform wise)

Compared to traditional operating systems such as NetBSD, the Rumprun and seL4 combination showed lower CPU utilization. This is expected, as Rumprun provides a minimalistic runtime environment without features like preemption or memory protection that would otherwise increase system overhead. For comparison, Linux exhibited the lowest CPU utilization overall, although it uses a different network driver and dynamic interrupt throttling, making direct comparison chal-

lenging. The higher variance observed in Linux data points can be attributed to its dynamic adjustment of interrupt throttle rates, whereas NetBSD uses a fixed rate. Within the seL4-based system, we observed a slight decrease in CPU usage when the system avoided the use of the legacy PIC for interrupt management.

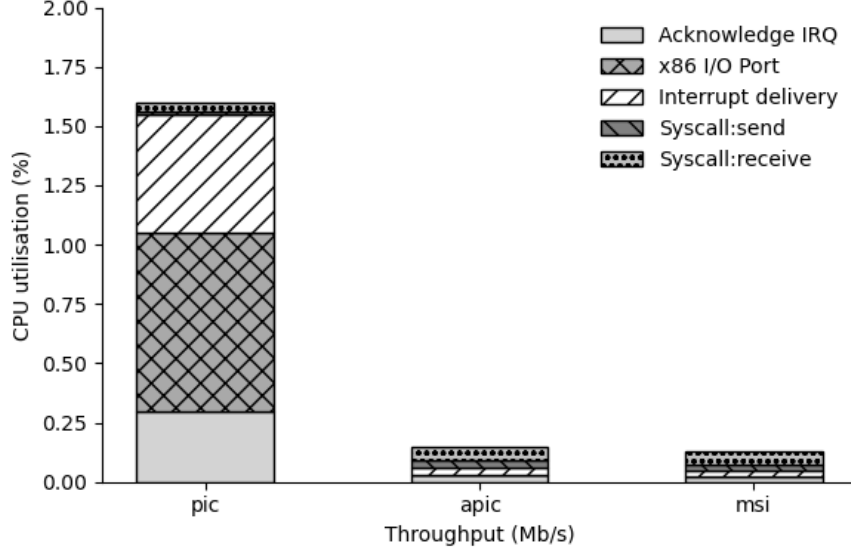


Figure 5.5: CPU utilization (method wise)

Figure 5.5 provides a detailed breakdown of the kernel overhead observed in the seL4 environment. In configurations relying on the PIC, the most time-consuming task was handling IO Port operations, which require privileged access through the kernel. In our case, IO Ports were used primarily for programming periodic timers. Beyond this, interrupt receipt, delivery to the user-level network stack, and the subsequent acknowledgment accounted for approximately 50% of the kernel overhead. To optimize the interrupt path, we also experimented with using HPET timers in combination with IOAPIC and MSI-based interrupt delivery, both of which use MMIO instead of IO Ports. These configurations significantly reduced CPU utilization, lowering it to under 0.3% of total CPU resources. The remaining system overhead was primarily due to sending and receiving system calls that provide synchronization between user processes and the kernel.

6 Conclusion & Future Work

This project explored the isolation capabilities of the seL4 microkernel in the context of unikernel-based systems, with a particular focus on leveraging Rumprun applications. The findings suggest that while seL4 offers strong theoretical guarantees for isolation backed by its formally verified, capability-based access control model, the practical integration of unikernels like Rumprun into seL4’s architecture remains a non-trivial challenge. Both design attempts demonstrated potential but encountered limitations related to build tooling, runtime constraints, and capability management. Despite these setbacks, the overall isolation potential of seL4 for running unikernels remains high, especially when compared to traditional operating systems or microkernels lacking formal verification.

6.1 Limitations

While this research demonstrates the feasibility of running a Rumprun unikernel on the seL4 microkernel, several significant limitations have emerged that restrict its broader applicability and real-world deployment potential.

One of the primary limitations is that the current system design supports only a single Rumprun application running on seL4 at any given time. This constraint stems from the inherent architecture of Rumprun, which is designed to build the entire application and its minimal operating environment into a single monolithic image. When integrated with seL4, which relies on static system configuration and compile-time resource allocation, this tight coupling becomes even more rigid. As a result, launching multiple Rumprun-based applications within a single seL4 instance is not currently feasible without substantial modifications to the build and runtime systems of both platforms.

Because only one application is supported at a time, no traditional process scheduling or context switching between multiple applications is implemented in the current setup. In most general-purpose operating systems or multi-application unikernel platforms, a scheduler is responsible for allocating CPU time between processes or threads, providing multitasking and responsiveness. However, in this setup, seL4 acts more like a static hypervisor, where each component is pre-defined and launched at the system start. There is no scheduler in the user space to manage application switching, nor is there any such mechanism in the kernel. This means the current system lacks dynamic process management, a core capability required for hosting more than one active application or service concurrently.

To implement true multitasking support, core components of a process scheduler would need to be introduced at the kernel level, including context-switch mechanisms and scheduling hooks. Meanwhile, the scheduling policies should ideally be implemented in user space as a separate scheduling service. This would preserve the microkernel principle of moving policy out of the kernel, maintain-

ing seL4’s formal verification guarantees. However, implementing such a design introduces considerable complexity. It requires a redesign of application lifecycle management, inter-process communication, and the capability distribution model, all of which are statically defined in the current seL4 system.

This single-application limitation also affects scalability and deployment models. In scenarios where multiple isolated services or microservices need to coexist presents a major bottleneck. The only current workaround is to deploy each Rumprun application on a separate instance of the seL4 microkernel, essentially creating multiple isolated microkernel environments. While this may be acceptable in some virtualized contexts, it leads to increased memory overhead, redundant kernel instances, and duplication of system services, which negates many of the benefits of using a lightweight microkernel and unikernel approach in the first place.

Furthermore, the tightly integrated nature of Rumprun and its reliance on its build system limits the modularity and composability of components. Unlike traditional CAMkES components, which can be instantiated and connected flexibly through Interface Definition Language and system description files, Rumprun applications must be compiled and linked as a whole. This hampers the ability to treat them as reusable or independently schedulable units within a larger system architecture.

6.2 Future Directions

A promising direction moving forward is the exploration of hybrid approaches. Specifically, combining minimal CAMkES components to manage critical services or security boundaries, while deploying Rumprun applications as standalone components within those boundaries, could provide a more modular and manageable architecture. Such an approach may better align with seL4’s strengths, such as static resource allocation and strict capability separation while avoiding the pitfalls of trying to shoehorn existing monolithic unikernel toolchains directly into a CAMkES workflow.

Future work should begin by addressing the integration and debugging challenges between CAMkES and Rumprun. Understanding and resolving issues related to linking TLS management, and component startup logic will be critical to making this architecture viable. In parallel, a systematic benchmarking of isolation versus performance trade-offs is essential to evaluate the practicality of deploying such systems in real-world scenarios. Finally, the exploration of alternative unikernel frameworks such as Solo5, which offers a more modular and minimalist approach to unikernel development could reveal better compatibility with seL4’s design principles and may serve as a better fit for secure, isolated execution in microkernel environments.

List of References

- Albinet, A., Arlat, J. & Fabre, J.-C. (2004), Characterization of the impact of faulty drivers on the robustness of the linux kernel, *in* ‘International Conference on Dependable Systems and Networks, 2004’, pp. 867–876.
- Alnaim, A., Alwakeel, A. & Fernández, E. (2019), A pattern for an nfv virtual machine environment, pp. 1–6.
- Atlidakis, V., Andrus, J., Geambasu, R., Mitropoulos, D. & Nieh, J. (2016), Posix abstractions in modern operating systems: the old, the new, and the missing, *in* ‘Proceedings of the Eleventh European Conference on Computer Systems’, EuroSys ’16, Association for Computing Machinery, New York, NY, USA.
- Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K. & Ustuner, A. (2006), ‘Thorough static analysis of device drivers’, *SIGOPS Oper. Syst. Rev.* **40**(4), 73–85.
- Barik, R. K., Lenka, R. K., Rao, K. R. & Ghose, D. (2016), Performance analysis of virtual machines and containers in cloud computing, *in* ‘2016 International Conference on Computing, Communication and Automation (ICCCA)’.
- Blair, W., Robertson, W. & Egele, M. (2023), ‘Threadlock: Native principal isolation through memory protection keys’.
- DiBona, C. & Ockman, S. (1999), *Open sources: Voices from the open source revolution*, ” O’Reilly Media, Inc.”.
- Drebes, R. & Nanya, T. (2010), Analysis of inter-module error propagation paths in monolithic operating system kernels, pp. 175–184.
- Elphinstone, K. & Heiser, G. (2013), From l3 to sel4 what have we learnt in 20 years of l4 microkernels?, *in* ‘Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles’, SOSP ’13, Association for Computing Machinery, New York, NY, USA, p. 133–150.
- Gu, J., Wu, X., Li, W., Liu, N., Mi, Z., Xia, Y. & Chen, H. (2020), Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication, *in* ‘2020 USENIX Annual Technical Conference (USENIX ATC 20)’, pp. 401–417.
- Hakamian, A. & Rahmani, A. (2015), ‘Evaluation of isolation in virtual machine environments encounter in effective attacks against memory’, *Security and Communication Networks* **8**.
- Heiser, G. & Elphinstone, K. (2016), ‘L4 microkernels: The lessons from 20 years of research and deployment’, **34**(1).
- Hevner, A. (2007), ‘A three cycle view of design science research’, *Scandinavian Journal of Information Systems* **19**.

- Khajehei, K. (2014), ‘Role of virtualization in cloud computing’, *International Journal of Advance Research in Computer Science and Management Studies* .
- Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. & Winwood, S. (2009), sel4: formal verification of an os kernel, *in* ‘Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles’, SOSP ’09, Association for Computing Machinery, New York, NY, USA, p. 207–220.
- Kolyshkin, K. (2006), ‘Virtualization in linux’, *White paper, OpenVZ* **3**(39), 8.
- Kuz, I., Liu, Y., Gorton, I. & Heiser, G. (2007), ‘Camkes: A component model for secure microkernel-based embedded systems’, *Journal of Systems and Software* **80**(5), 687–699. Component-Based Software Engineering of Trustworthy Embedded Systems.
- Liedtke, J. (1995), ‘On micro-kernel construction’, *Proceedings of the fifteenth ACM symposium on Operating systems principles* .
- Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S. & Crowcroft, J. (2013), ‘Unikernels: Library operating systems for the cloud’, *ACM SIGARCH Computer Architecture News* **41**(1), 461–472.
- Matos, E. d. & Ahvenjärvi, M. (2022), ‘sel4 microkernel for virtualization use-cases: Potential directions towards a standard vmm’, *Electronics* **11**(24).
- Mi, Z., Li, D., Yang, Z., Wang, X. & Chen, H. (2019), Skybridge: Fast and secure inter-process communication for microkernels, *in* ‘Proceedings of the Fourteenth EuroSys Conference 2019’, EuroSys ’19, Association for Computing Machinery, New York, NY, USA.
- Miao, H. (2011), ‘Analysis of practicality and performance evaluation for monolithic kernel and micro-kernel operating systems’, *Hui Miao International Journal of Engineering* **5**, 2011–277.
- Mutia, R. I. (2010), Inter-process communication mechanism in monolithic kernel and microkernel.
- Narayanan, V. & Burtsev, A. (2023), ‘The opportunities and limitations of extended page table switching for fine-grained isolation’, *IEEE Security amp; Privacy* **21**(03), 16–26.
- Obenland, K. M. (2000), The use of posix in real-time systems , assessing its effectiveness and performance.
- Rashid, R., Julin, D., Orr, D., Sanzi, R., Baron, R., Forin, A., Golub, D. & Jones, M. (1989), Mach: a system software kernel, *in* ‘Digest of Papers. COMPCON Spring 89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage’, pp. 176–178.

- Raza, A., Unger, T., Boyd, M., Munson, E. B., Sohal, P., Drepper, U., Jones, R., De Oliveira, D. B., Woodman, L., Mancuso, R., Appavoo, J. & Krieger, O. (2023), Unikernel linux (ukl), *in* ‘Proceedings of the Eighteenth European Conference on Computer Systems’, EuroSys ’23, Association for Computing Machinery, New York, NY, USA, p. 590–605.
- Rodríguez-Haro, F., Freitag, F., Navarro, L., Hernández-sánchez, E., Farías-Mendoza, N., Guerrero-Ibáñez, J. A. & González-Potes, A. (2012), ‘A summary of virtualization techniques’, *Procedia Technology* **3**, 267–272.
- Sultan, S., Ahmad, I. & Dimitriou, T. (2019), ‘Containers’ security: Issues, challenges, and road ahead’, *IEEE Access* **PP**, 1–1.
- Sung, M., Olivier, P., Lankes, S. & Ravindran, B. (2020), Intra-unikernel isolation with intel memory protection keys, *in* ‘Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments’, VEE ’20, Association for Computing Machinery, New York, NY, USA, p. 143–156.
- Tanenbaum, A. S. & Woodhull, A. S. (2005), *Operating Systems Design and Implementation*, Pearson Prentice Hall, Division of Simon and Schuster One Lake Street Upper Saddle River, NJ, United States.
- The seL4 Authors and Contributors (2024), *seL4 Reference Manual*, seL4 Foundation. Version 13.0.0.
- Torvalds, L. (1997), ‘Linux : a portable operating system’.