Integrating VirtIO and QEMU on seL4 for Enhanced Devices Virtualization Support

Everton de Matos, Conor Lennon, Eduardo K. Viegas
Secure Systems Research Center
Technology Innovation Institute
Abu Dhabi, United Arab Emirates
{everton.dematos, conor.lennon, eduardo.viegas}@tii.ae

Markku Ahvenjärvi, Hannu Lyytinen, Ivan Kuznetsov, Joonas Onatsu, and Anh Huy Bui
Unikie Oy
Tampere, Finland
{markku.ahvenjarvi, hannu.lyytinen, ivan.kuznetsov}@unikie.com,
{joonas.onatsu, anh.huy.bui}@unikie.com

***Abstract*—Virtualization is a crucial technology for consolidating workloads and improving resource utilization in modern computing systems. seL4 is a small TCB (Trusted Computing Base) microkernel that can be used as a hypervisor to provide virtualization features. However, providing standard device interfaces to it remains a significant challenge in achieving secure and high-performance virtualization solutions for critical systems. To address this challenge, this paper proposes an approach that leverages the VirtIO standard through QEMU to provide a secure and efficient device virtualization solution for seL4. The feature takes advantage of seL4's isolation guarantees and enables sharing of complex devices for multiple virtual machines, combined with the efficient communication interface provided by the VirtIO standard. We implemented and evaluated the approach using a set of benchmarks. The proposed approach leverages the VirtIO standard through QEMU on top of the seL4, aiming to offer a virtualization solution that accelerates development speed and enhances architectural flexibility by eliminating the need for native drivers.**

***Index Terms*—Virtualization, VirtIO standard, QEMU, seL4 microkernel, Hypervisor**

I. Introduction

Virtualization is a technology that enables multiple virtual machines (VMs) to operate on a single physical machine, each with its own operating system, applications, and data, while sharing the underlying physical resources of the host machine [1]. This technology allows for more efficient use of hardware resources, improved flexibility, and enhanced security [2]. Virtualization enables the creation of virtual devices that can be used by guest operating systems running on a virtual machine. VirtIO is a technology used in virtualization to provide efficient communication between the guest virtual machine and the hypervisor, allowing for high-performance virtual devices [3]. Hypervisors, which sit between the physical machine and virtual machines, are responsible for managing the physical resources and allocating them to the different VMs.

There are two types of hypervisors: Type-1, or bare-metal hypervisors, run directly on the host machine's hardware, while Type-2, or hosted hypervisors, run on a host operating system [4]. Type-1 hypervisors are generally considered more efficient and secure than Type-2 hypervisors because they have direct access to the hardware and operate independently of the host operating system. However, Type-2 hypervisors are easier to install and manage, making them more suitable for desktop virtualization and other simpler applications [5].

One key difference between Type-1 and Type-2 hypervisors is the size of their Trusted Computing Base (TCB) [6]. Type-1 hypervisors have a small TCB, as they run directly on the hardware and do not rely on a host operating system. In contrast, Type-2 hypervisors have a larger TCB, as they are implemented as a software layer on top of an existing operating system. This larger TCB increases the attack surface and potential vulnerabilities of the hypervisor. Examples of Type-1 hypervisors include Bao [7], Jailhouse [8], and Xen [9]. On the other hand, Type-2 hypervisors include Oracle VirtualBox, VMWare Workstation, and KVM [10]. Notably, when seL4 is used as a hypervisor, it acts as a Type-1 hypervisor due to its small trusted computing base and direct control over the underlying hardware [11].

seL4 is a microkernel-based operating system developed to provide a high-assurance, secure, and efficient platform for deploying critical systems [2] [11]. It is unique in that it is the first operating system to have formal proof of its implementation and properties, providing a high level of confidence in its security and functionality. In addition to its use as an operating system, seL4 can also be utilized as a hypervisor, providing a minimal and scalable infrastructure for virtualized environments. This makes seL4 well-suited for use in embedded and real-time systems and for use in high-assurance security-critical systems that require virtualization.

While seL4's virtualization functionality has proven useful, the seL4 stack possesses gaps at the Virtual Machine Monitor (VMM), userspace, and tooling levels that complicate use case fulfillment for both users and developers [12]. Furthermore, the lack of a publicly accessible or present standard interface for virtual devices on top of seL4 poses limitations for deploying more complex use-cases, inspiring the need for our proposed solution. Even though seL4 supports a limited set of VirtIO devices natively through *libsel4vmmplatsupport*[1] (i.e., VirtIO over PCI, *virtio-console*, and *virtio-net*), it lacks support for all the other possible VirtIO devices that could be used in a system, such as *virtio-blk*, *virtio-gpu*, and VirtIO Balloon, just

[1] https://docs.sel4.systems/projects/virtualization/libsel4vmmplatsupport.html

to name a few [13]. The novelty of the proposed approach sits in providing generic and modern VirtIO device support to systems on top of seL4.

The proposed approach provides VirtIO standard support to devices on top of seL4 hypervisor. It uses QEMU to provide the VirtIO device backends for the guest VMs. Thus, we get support for a wide range of VirtIO-device backends with a minimal amount of source code. Without this approach, we would have to manually create the support for each VirtIO backend and the proper device driver support. Also, accelerated graphics and proper file system support are some examples of features that we get when using Linux VMMs, such as QEMU as our backend. Moreover, the approach enables the rapid development of feature-rich systems on top of seL4. To the best of our knowledge, there is no publicly available support of standard devices in such a scalable way that runs on top of seL4 as a hypervisor.

Our efforts culminate into the main contribution of laying the groundwork for more robust and future-ready systems based on seL4. We design and provide abstractions at both the *libsel4vm*/*libsel4vmmplatsupport* and CAmkES levels. The overall contributions of this paper can be summarized as follows:

- Establish support for VirtIO-based virtualization using seL4 with QEMU, addressing the limitations of existing solutions and providing a novel integration of VirtIO support.
- Introduce an advanced modern VirtIO frontend and comprehensive abstractions for facilitating external MMIO devices, augmenting the efficiency and versatility of seL4-based systems.
- Conducted a comprehensive evaluation of the proposed solution using benchmarks.
- Contributed to the field of virtualization with seL4 as a hypervisor by offering a scalable and flexible solution for critical systems, filling an existing gap in the literature on supporting devices on top of seL4.

The rest of this paper is presented as follows. Section II presents basic concepts related to the proposed approach. Section III presents work related to the proposed approach. Section IV presents our solution, showing its design and components. Section V presents the evaluation results of the proposed approach. Finally, Section VI concludes the paper.

II. Background

This section provides an overview of the concepts and technologies that serve as the foundation for the proposed approach. It covers topics such as virtualization, hypervisors, and the seL4 microkernel. Additionally, it discusses VirtIO, a standard for providing virtual devices to virtual machines, and QEMU, a popular open-source emulator that supports VirtIO.

A. Virtualization and Hypervisors

Virtualization is the ability to run multiple independent operating systems on a single physical machine. This technology enables the sharing of hardware resources such as CPU, memory, and storage, making it possible to consolidate workloads and increase efficiency [2]. Virtualization is achieved through the use of a hypervisor, which is a software layer that abstracts physical resources and presents virtual resources to guest operating systems [14].

Hypervisors can be classified into two main types: Type-1 and Type 2. Type-1 hypervisors, also known as native or bare-metal hypervisors, run directly on the host machine's hardware and have direct access to physical resources. They provide a high degree of isolation between guest operating systems, resulting in better performance and security. Usually, taking into account the architectural point of view, the Type-1 hypervisors run at a higher privilege level than the applications and at a lower privilege level than the firmware [15]. For instance, in ARM the Type-1 hypervisor is usually placed at EL2, and in RISC-V, at HS. Type 2 hypervisors, also known as hosted hypervisors, run as applications within a host operating system. They provide less isolation between guest operating systems and are less performant than Type-1 hypervisors but are easier to install and manage [4] [16]. The hypervisor acts as a mediator between the physical hardware and the virtual machines, managing both the Virtual Machine Monitors (VMMs) and Virtual Machines (VMs) to ensure optimal performance and resource allocation [17].

The VMM is a key component of virtualization that provides the interface between the physical hardware and the guest operating systems running on top of it [18]. Usually, architecture-wise, it runs at the same privilege level as the applications. The VMM abstracts physical hardware resources and presents virtual resources to the guest operating systems, enabling multiple guest operating systems to run on a single physical machine. The VMM also provides mechanisms for managing virtual resources, such as allocating CPU time, memory, and storage to each VM [19]. It ensures that each VM operates independently and securely by enforcing isolation and resource allocation policies. A VM is an emulation of a computer system that runs as a software program on top of a VMM [20]. It includes a complete set of virtualized hardware resources, such as CPU, memory, and storage, and can run a complete operating system and associated applications. Each VM is isolated from other VMs running on the same host, ensuring that each guest operating system can run securely and independently.

Virtualization has become a widely adopted technology in data centers, cloud computing, and other IT environments due to its flexibility, scalability, and cost-effectiveness. It has enabled the rapid deployment of new applications, simplified management of IT resources, and increased utilization of hardware. In recent years, virtualization has become increasingly popular in the field of embedded systems [10] [21]. Traditionally, embedded systems were designed to run a single application or operating system on dedicated hardware. However, this approach has limitations, such as difficulty in upgrading hardware and software, inflexibility, and potential security vulnerabilities. Virtualization allows multiple operating systems and applications to run on a single hardware

platform, providing greater flexibility, scalability, and isolation. Moreover, virtualization enables the development of critical systems with varying levels of safety and security requirements, such as those found in automotive, aerospace, and medical devices [22] [23]. Therefore, virtualization has become an essential tool for embedded system developers, enabling them to create more efficient, secure, and adaptable systems.

B. seL4 as a hypervisor

seL4 is a microkernel-based operating system designed to provide high assurance security and reliability for critical systems [24]. Unlike monolithic kernels, which run all of their services and drivers in kernel space, microkernels adopt a minimalist approach, only running the bare essentials in kernel space and pushing everything else into user space [25]. This design approach provides several benefits, including increased security, fault isolation, and flexibility. Because each service or driver runs in its own isolated address space, any bugs or errors that may arise in one component will not affect the operation of the entire system, providing fault tolerance and resilience.

The seL4 microkernel implements a capability-based access control model, providing a robust mechanism for secure resource management in embedded systems. This capability-based design enables fine-grained control over access rights, allowing the system to grant or revoke permissions for specific resources, such as memory regions or hardware devices, with precision. This structure permits secure delegation of access rights while maintaining a clear view of the system's security policy. The separation of concerns in seL4's design ensures that the principle of least privilege is upheld, thus minimizing the potential attack surface and facilitating fault containment. Furthermore, the modular architecture of seL4 enables efficient isolation between system components, which is crucial for maintaining system integrity and security in safety-critical applications [26].

seL4 takes this design approach a step further by implementing formal verification techniques to prove the correctness of the kernel [27]. This means that seL4 is mathematically proven to be free from certain types of bugs, such as buffer overflows, *null* pointer dereferences, and use-after-free errors, which are common sources of security vulnerabilities.

In addition to its use as a standalone operating system, seL4 can also be used as a hypervisor, providing isolation and virtualization capabilities for guest operating systems [12]. seL4 as a microkernel, with its small trusted computing base, makes it well-suited for use in embedded systems where memory and processing power are limited. Using seL4 as a hypervisor enables a wide range of use cases, including partitioning a single system into multiple virtual machines, providing secure and isolated execution environments for different applications or operating systems, and enabling mixed-criticality systems where different parts of the system have different safety or security requirements.

C. VirtIO Standard and QEMU

The VirtIO standard is a widely adopted virtualization interface that provides a common framework for communication between a hypervisor and virtual machines [13]. It defines a set of devices and device drivers that can be used by virtual machines to communicate with the underlying hypervisor, allowing for efficient and flexible communication between the virtual and physical components of a system.

Prior to VirtIO, virtualized I/O was typically implemented using custom, proprietary interfaces that were specific to particular hypervisors or operating systems. This made it difficult to create portable virtual machines that could run on different hypervisors or be migrated between different physical hosts [28]. VirtIO, on the other hand, allows virtual machines to use the same set of devices and drivers, regardless of the underlying hypervisor or physical hardware. This not only simplifies the development of virtual machines but also makes it easier to migrate virtual machines between different physical hosts, since the same set of devices and drivers can be used on all hosts [29].

The VirtIO standard includes a set of devices such as network interfaces, storage devices, console devices, and memory ballooning devices [30]. Each device is accompanied by a set of drivers that can be used by the guest operating system to communicate with the device. The devices are designed to be simple and efficient, with a minimal set of features that can be extended as needed. This allows for easy implementation and reduces the overhead of virtualizing I/O.

Overall, the VirtIO standard has become a *de facto* standard for virtualized I/O in the industry, and is widely adopted by hypervisors and operating systems. Its flexibility and portability make it an ideal choice for embedded systems that require virtualization capabilities [31].

VirtIO drivers in Linux are implemented as kernel modules in the guest operating system and communicate with the backend in the host operating system through a VirtIO interface [13]. QEMU can provide the backend for VirtIO devices, enabling communication between the guest and the host operating systems, and allowing for handling of I/O operations. QEMU (Quick EMUlator) is an open-source emulator and virtual machine monitor (VMM) that can emulate a wide range of hardware devices and can run various operating systems as guests. It is widely used in the virtualization ecosystem as it supports different types of virtualization, such as full system virtualization, hardware virtualization, and container virtualization [32].

By using the VirtIO standard with QEMU as the backend, VMs can achieve high levels of performance and scalability, making it a popular choice for virtualization solutions. This approach is especially relevant for embedded systems, where resource utilization is critical, and hardware configurations vary greatly.

III. Related Work

The application of VirtIO in various hypervisors has been extensively studied in recent years, and several research efforts

have focused on improving its performance and capabilities. This section reviews the existing literature on using VirtIO in different hypervisors. We also discuss how the proposed approach of integrating VirtIO and QEMU in seL4 leverages standard virtualization of devices for embedded systems.

Park et al. [33] propose a new IO virtualization technique called "Ambient Virtio" for the seamless integration and access of devices in ambient computing environments. The authors implemented virtio-ambient device that virtualizes the physical device and enables ambient access of VMs. The proposed technique is based on the VirtIO standard and leverages the advantages of virtualization to provide a flexible, scalable, and secure way of accessing devices in the ambient computing environment. The paper describes the architecture of Ambient VirtIO and evaluates its performance and overhead using a prototype implementation. The results show that Ambient VirtIO can provide low latency and high throughput while maintaining a low overhead, making it a promising solution for IO virtualization in ambient computing. The authors made use of KVM as a hypervisor to enable their solution.

Li et al. [34] introduce microverification as a new approach for verifying the security properties of large, multiprocessor commodity systems. The microverification approach reduces the proof effort for a commodity system by retrofitting the system into a small core and a set of untrusted services. MicroV, a framework for verifying the security properties of multiprocessor commodity systems, further reduces proof effort by providing a set of proof libraries and helper functions. The authors used MicroV to prove the security properties of the Linux KVM hypervisor by retrofitting it into a small, verifiable core, KCore, and a rich set of untrusted hypervisor services, KServ. They proved that any malicious behavior of the untrusted KServ using KCore's interface could not violate the desired security properties, including VM confidentiality and integrity. Their approach supports standardized VirtIO virtualization with vhost kernel optimizations.

Patel et al. [35] introduce Xvisor, a lightweight, open-source Type-1 hypervisor that provides flexible and portable virtualization. Xvisor supports ARM virtualization extensions, enabling both full virtualization and para-virtualization through optional VirtIO-compatible drivers. It allows guest interrupts to be managed directly without hypervisor intervention and uses ARM's virtualization support to ensure memory isolation between the hypervisor, guests, and guest applications. Xvisor has around 440K lines of code in its kernel and outperforms KVM ARM guest and Xen ARM DomU in terms of CPU overhead and memory bandwidth [21].

Oliveira et al. [36] discuss how traditional virtualization may not be suitable for embedded systems and highlights the need for inter-partition communication mechanisms to enable cooperation between different OS classes. It mentions various virtualization solutions that have implemented such mechanisms and the adoption of VirtIO as the transport abstraction layer for communication mechanisms. The paper presents the implementation of a standardized inter-partition communication mechanism in a TrustZone-assisted hypervisor using VirtIO as the transport layer.

Li et al. [37] discuss the use of embedded virtualization in IoT solutions, particularly in industrial and automotive scenarios. The article presents ACRN, a lightweight, scalable, and open-source embedded hypervisor designed for IoT development. ACRN provides a secure and efficient solution for real-time virtualization and supports spatial and temporal isolation. ACRN offers rich I/O virtualization interfaces, including VirtIO, full emulation, mediated pass-through, pass-through, and so on. To develop VirtIO backend driver, users can use ACRN's VirtIO backend service (VBS) framework.

Currently, seL4 natively supports three VirtIO devices: VirtIO over PCI, *virtio-console*, and *virtio-net* [38]. VirtIO over PCI is a virtualized interface that allows the operating system to interact with virtual devices, such as storage devices and network adapters. The *virtio-console* provides a mechanism for input and output communication between the guest and host, while *virtio-net* offers a network interface. However, it is worth noting that adding support for new VirtIO devices natively to seL4 can be a complex and time-consuming task (e.g., *virtio-blk*, VirtIO RNG, VirtIO Balloon), requiring careful design and verification due to seL4's focus on security and correctness.

Compared to other hypervisors that support a wide range of VirtIO devices natively or through a framework, seL4 currently offers limited VirtIO support. While hypervisors like Xvisor, KVM, and ACRN have successfully provided virtualization capabilities for a broad range of devices, seL4 currently only natively supports VirtIO over PCI, *virtio-console*, and *virtio-net*. Given the limited native support for VirtIO devices in seL4, our solution steps up to bring a wider variety of VirtIO devices into the picture. We adopt the modern VirtIO standard to make the system more flexible and adaptable. Our approach provides generic VirtIO interfaces by using QEMU as our backend. This way, we make it easier to support more VirtIO devices and speed up their development and deployment. As a result, we offer an efficient way to extend VirtIO device support in seL4, cutting down the complexity and time usually required to add native support, and we do this without compromising the security and correctness that seL4 is known for.

IV. Proposed Approach

In this section, we present our proposed approach to enhance seL4's support for VirtIO devices, specifically focusing on the addition of new VirtIO devices. Our proposed approach aims to overcome the limitations of natively adding support for new VirtIO devices in seL4. To achieve this, we leverage the VirtIO backend in QEMU to provide a more practical solution to support additional VirtIO devices. By using user-level drivers, we can achieve more rapid development and deployment of new device support. We aim to improve seL4's flexibility and scalability in virtualizing devices while also maintaining its high level of security and correctness. We provide an overview of the architecture of our approach, including the design of user-level drivers and their interaction with seL4's existing VirtIO implementation.

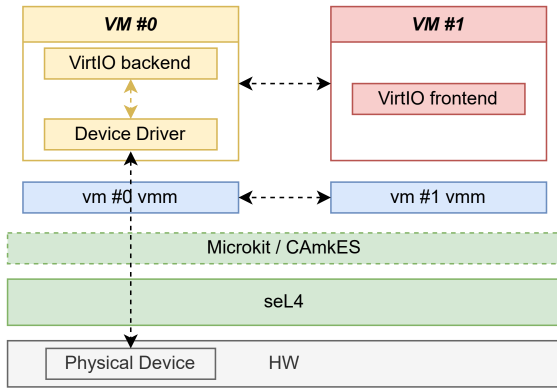

Fig. 1. VirtIO driver support for seL4

Figure 1 presents a high-level view of our proposed approach to enable VirtIO driver backends and frontends on top of seL4. This approach serves as the basis for understanding the proposed approach in this paper. The VirtIO interfaces can be connected to various open-source technologies, including QEMU, crosvm, and Firecracker, among others. In this setting, the open-source technologies execute in the user space of a different VM than the one utilizing the device. This approach facilitates reusability, portability, and scalability. Figure 1 depicts this approach, which involves a scenario where a *VM #1* utilizes services provided by a backend *VM #0*.

Figure 2 shows how our system architecture is able to provide support for VirtIO devices using QEMU on top of the seL4 hypervisor. The ultimate goal of this architecture is to bring seL4 on par with the other hypervisors in terms of device support and flexibility for development. To tackle the device support, we leveraged the existing Linux VMMs, such as QEMU, to provide the VirtIO device backends for the guest VMs. The approach has some benefits, such as: (i) getting support for a wide range of VirtIO-device backends with a minimal amount of code, (ii) enabling the development of feature-rich multimedia capable systems on top of seL4, and (iii) developers get the familiar user experience which helps adoption of seL4. Without using the existing Linux VMMs, we'd have to manually create the support for each VirtIO backend on top of seL4, which is a non-trivial task with a large amount of work.

The proposed architecture runs seL4 as a hypervisor. On top of it, it can use either CAmkES or seL4 Microkit to define the system and assign the resources/capabilities. The CAmkES project is a framework for executing virtualized Linux guests on seL4 for ARM and x86 architectures. Its *camkes-vm* module functions as a virtual machine monitor (VMM) server, which enables the initialization, booting, and runtime management of guest operating systems [39]. The project offers a straightforward method for running various virtualization scenarios with one or more virtual machines and different applications. Moreover, it allows device pass-through in such environments. However, a disadvantage of this framework is its inability to support dynamic virtual machines, requiring predefined VM configurations during the design phase. As a different option from CAmkES, the seL4 community has developed the seL4 Microkit, also known as Microkit [40] [41] [42] [43]. The Microkit intends to be less complex and more dynamic than CAmkES. The Microkit provides abstractions such as protection domains (PDs), communication channels (CCs), memory regions (MRs), and notifications and protected procedure calls (PPCs). A virtual machine is a special type of PD with additional attributes specific to virtualization. Internally, a virtual machine appears as a single PD to other PDs, with its internal processes hidden from view. A Microkit program, which is an ELF (Executable and Linkable Format) file containing both code and data, runs within a PD and is exposed as memory regions that are mapped into the PD.

In order to provide VirtIO support through QEMU, our architecture has two kinds of VMs: *service-vm* and *user-vm*[2]. It allows the system to have multiple isolated *service-vms* managing their own guests (e.g., isolated functionality, or criticality domains). Figure 2 shows an architectural view of our architecture with one *service-vm* and one *user-vm*.

The *service-vm* is a Linux VM providing guest VM management interface and VirtIO backends provided by the Linux VMM (i.e., QEMU). Frameworks such as CAmkES or seL4PC can be used to configure the VM in various manners, such as the amount of RAM, and the device's resources. The VM is then created according to the configuration. The *service-vm* Linux kernel is loaded with a kernel driver module, *sel4-virt*, which handles the interaction towards the hypervisor VMM (e.g., Microkit or CAmkES). The driver provides an API for Linux VMM.

There can be multiple *service-vms* in the system, given that there are enough physical hardware resources available for each of the *service-vms* (e.g., RAM, block devices etc.). Typically a *service-vm* is launched by the root task (i.e., it is a static VM). In theory, a *user-vm* could act as a *service-vm*, and be a host for its guests. However, for performance reasons, it hardly makes sense in practical systems. The term *user-vm* refers to a virtual machine that functions as a guest within the context of a *service-vm*. Specifically, the *user-vm* is reliant upon the paravirtualized devices that are hosted by the *service-vm* in order to operate. The *user-vm* has the frontend instance of the VirtIO drivers.

The VirtIO device backends, provided by the Linux VMM (i.e., QEMU), use the physical devices of the hardware platform. This means that in order to use a given device type, such as *virtio-gpu*, the physical device must be passed through to the *service-vm*. The filesystem for the guests is allocated from the *service-vm* disk as regular files, for example, using *qcow2* format. The *virtio-blk* is used as a VirtIO backend for the filesystem.

The *qemu-sel4-virtio*[3] repository holds the source code of the *sel4-virt-glue*, as shown in Figure 2. This implementation serves as the primary interface between the seL4 and the

[2]The terminology of *service-vm/user-vm* was adopted from the ACRN project [37].

[3]https://github.com/tiiuae/qemu-sel4-virtio

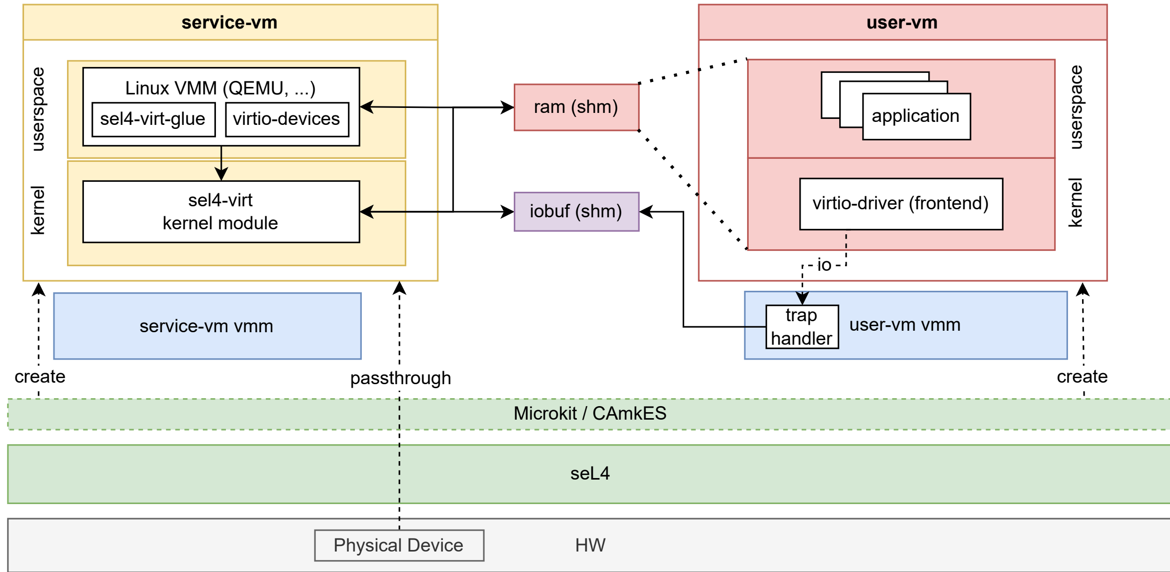

Fig. 2. Overview of the proposed architecture

emulated hardware provided by the QEMU environment. It brings together various components, including the interrupt handling mechanism, virtual device management via VirtIO, and the implementation of the main event loop for the system. It initiates a sequence of events, including the initialization of seL4 and its utilities, and the creation and setup of VirtIO devices. Following the setup, the system enters an infinite loop where it continually waits for and responds to hardware interrupts.

The *kmod-sel4-virt*[4] repository contains the Linux kernel module to manage seL4 guest VMs, as shown in Figure 2 as *sel4-virt kernel module*. It utilizes VirtIO for effective I/O operations between the host (i.e., *service-vm*) and guest (i.e., user-vm) VMs. It enables guests to get high-performance I/O operations in a virtualized environment. The kernel module defines an API for managing VMs. It includes the capability to handle I/O requests from the VMs and perform operations when I/O is completed. The kernel module uses file descriptors and file operations such as *ioctls* to manage a VM, it also supports multiple VMs. It is worth mentioning that this implementation is highly influenced by ACRN and KVM approaches.

V. Evaluation

In this section, we assess the performance of VirtIO within the context of seL4, focusing on networking (i.e., *virtio-net*) and storage (i.e., *virtio-blk*). VirtIO performance is crucial in virtualized environments as it directly influences overall system performance. The evaluation aims to determine the QEMU VirtIO's performance on seL4 and examine the effects of seL4's security properties on VirtIO performance. The outcome of this evaluation will offer valuable insights into the viability of utilizing seL4 as a secure and efficient platform for virtualized environments.

The experimental setup for this study comprises a Raspberry Pi 4 4GB board running seL4 as a hypervisor. Our source code is publicly accessible on GitHub[5], enabling result reproduction, contributions, and further research [44]. The hypervisor offers a secure and isolated environment for virtual machines to function. The setup features a *service-vm* that operates the QEMU Linux VMM at the user level, supplying the VirtIO backend for virtual machines. In addition, the *service-vm* manages virtual machine images and conducts I/O operations on behalf of the *user-vm*. The *user-vm* runs test applications and the VirtIO front end at the user level. It is important to note that both *service-vm* and *user-vm* runs in the same processor core of the Raspberry Pi 4 platform, as the support for multicore VMs/VMMs is not available in seL4 mainline repositories. Details on the architectural view of the evaluated setup can be found in Figure 2. Communication between the VirtIO front end and the VirtIO backend provided by the *service-vm* enables access to the virtual machine's devices. In essence, this experimental setup offers a platform for executing tests and examining virtual machine performance in a controlled environment.

In this paper, we propose a solution that offers key benefits for building feature-rich systems on top of seL4 while leveraging existing technology and minimizing codebase complexity. Our primary objectives include facilitating the faster development of such systems and enhancing their potential security through seL4's isolation guarantees. We contend that our solution provides better security compared to existing options, such as KVM, due to the inherent isolation properties of seL4. Furthermore, this implementation offers enhanced flexibility

[4]https://github.com/tiiuae/kmod-sel4-virt

[5]https://github.com/tiiuae/tii_sel4_build

compared to the native VirtIO devices implementation present in seL4. The native seL4 implementation is limited by its support for only three VirtIO devices, and the process of developing new support is both complex and labor-intensive.

A. *virtio-net*

The network interface is one of the most crucial components in a system, especially in modern distributed systems. The *virtio-net* is a widely used network device model that provides a high-performance virtual network interface for virtual machines [45]. It is designed to be agnostic to the underlying physical network hardware, which allows virtual machines to be easily migrated between physical hosts with different network setups. The *virtio-net* has become the *de facto* standard for virtualized network devices, and it is widely supported by various hypervisors.

The *iPerf3* is a widely used, open-source network performance measurement tool that facilitates comprehensive testing of various network parameters, including bandwidth, latency, and packet loss [46]. This versatile and robust utility is suitable for evaluating the performance of *virtio-net* in the context of our experimental setup. The *iPerf3* enables both TCP and UDP tests, providing a means to analyze the behavior and throughput of these protocols within the virtualized environment. By utilizing *iPerf3* as a benchmark for *virtio-net*, we aim to generate a detailed understanding of the networking performance within the proposed seL4-based virtualized system.

In our evaluation, we separately assessed the performance of the *user-vm* and the *service-vm* using *iPerf3*. Each virtual machine was configured to communicate with a distinct machine on the same network, rather than operating in a client-server configuration between the *user-vm* and *service-vm*. This approach allowed us to analyze the performance of each virtual machine independently, offering a more accurate representation of their individual networking capabilities within the seL4-based virtualized system. This environment encompasses the communication from the Raspberry Pi 4 (RPi4) to a Personal Computer (PC) and vice-versa. The network is supported by a 1 Gbit router.

Under the evaluation tests of the *user-vm* we ran the tests with two different setups: (a) with *vhost* and (b) without *vhost*. The *vhost* emerges as a pivotal mechanism engineered to improve the performance of virtual devices such as network and block devices within VMs. This enhancement is achieved by adeptly offloading the processing of *virtqueues*, a fundamental component in the VirtIO standard for network and disk device drivers within a virtual environment [47].

The *vhost* backend, located in the kernel module, efficiently manages I/O requests and *virtqueue* processing, minimizing the overhead from context switching between user and kernel space. The *vhost-user*, operating in user space, similarly oversees I/O requests and *virtqueue* processing, contributing to enhanced performance in virtual environments. The *vhost* frontend, such as QEMU, manages the initiation of VirtIO and feature negotiation with the backend [48]. After completing these tasks, it hands over essential configurations and descriptors for smooth operation, including memory region configuration, *virtqueue* configurations, and event file descriptors for asynchronous event notifications. In the execution phase, the *vhost* backend, embodying the actual device, processes *virtqueues* and manages the transfer of network packets to and from the guest VM, while staying responsive to notifications from the guest.

To accurately evaluate the *service-vm* performance, we conducted tests considering the passthrough to the actual network device of the RPi4 board. This method enabled a direct and clear assessment of the maximum performance achievable by the *user-vm* using our VirtIO backend implementation. We acknowledge the possibility of further enhancing these performance metrics through additional passthrough optimizations. This specific configuration, distinct from traditional approaches, is crucial. It provides a comprehensive understanding, unveiling both the potential and constraints of the *service-vm* performance with our network device passthrough implementation.

Ten iterations of the *iPerf3* benchmark were performed for each of the aforementioned scenarios, with each iteration lasting 60 seconds. The first 5 seconds of each iteration was omitted to avoid the TCP slow start phase. Table I presents a summary of the average results obtained for the tests on seL4. In Table I, MB represents a megabyte, which is equal to 2^{20} bytes[6].

TABLE I
AVERAGE THROUGHPUT FOR *iPerf3* BENCHMARK ON SEL4

Scenario	Direction	Sender (Mbps)	Receiver (Mbps)
service-vm	RPi4 to PC	455.2	455.2
service-vm	PC to RPi4	662.7	662.7

Scenario	Direction	vhost	Sender (Mbps)	Receiver (Mbps)
user-vm	RPi4 to PC	No	213.2	213.1
user-vm	PC to RPi4	No	392.7	392.6
user-vm	RPi4 to PC	Yes	317.1	317.0
user-vm	PC to RPi4	Yes	410.0	410.0

In the tested scenario involving the seL4 *service-vm*, *iPerf3* was utilized to assess the passthrough implementation performance. This evaluation delineates the maximum achievable performance for the *user-vm*. In the evaluation conducted from the *service-vm* on RPi4 to PC, the *iPerf3* tests demonstrate a stable network throughput. The results register no packet retransmissions across all tests. A consistent average bitrate is observed, ranging from approximately 421 Mbits/s to 503 Mbits/s. The congestion window size remained stable, further highlighting the network's efficiency. Considering the scenario

[6]The International Electrotechnical Commission (IEC) has established the binary prefix "mebibyte" (MiB) to represent 2^{20} bytes, while megabyte (MB) traditionally denotes 10^6 bytes. However, in many contexts, including computer memory and storage, MB is still commonly used to represent 2^{20} bytes.

of PC to RPi4 *service-vm*, the average bitrate predominantly hovering around 605 Mbits/sec to 610 Mbits/sec.

Considering the seL4 *user-vm* experiments, the results differ in two distinct scenarios: no *vhost* and with *vhost*.

In the case of scenario of a *user-vm* with no *vhost*, considering the *iPerf3* from RPi4 to PC, the average bitrate was consistently registered within the range of 205 to 223 Mbits/sec. The congestion window size, a crucial parameter indicating the robustness of the network setup [49], demonstrated an increasing trend in the experiments. This consistent increase in the congestion window size, along with the zero packet retransmission observed, emphasizes the network's reliability and efficiency in managing data transfers under various load conditions. This performance is further mirrored in the transfer sizes, which ranged from 12.5 MBytes to 27.7 MBytes within one-second intervals in different tests, and the overall data transferred in each test, with figures such as 1.43 GBytes to 1.56 GBytes being recorded. In the evaluation of data transmission from a PC to RPi4 running the *user-vm* without *vhost*, the data underscore a stable average bitrate of approximately 392-395 Mbits/sec. The consistent congestion window sizes, typically oscillating between 892 KBytes and 938 KBytes. The total data transferred predominantly lies around 2.73 to 2.76 GBytes, accentuating the uniformity in network performance. One of the observations is the nearly nonexistent packet retransmissions.

Considering the scenario with *vhost*, in the execution of *iPerf3* from RPi4 to PC, the results consistently demonstrate a robust and stable connection, evidenced by a lack of packet retransmissions in all test scenarios. The tests reveal an average bitrate ranging from 242 Mbits/sec to 343 Mbits/sec, with the majority of the tests maintaining a bitrate of approximately 317 Mbits/sec. The congestion window size predominantly hovers around 1.77 MBytes, indicating the network's ability to maintain a stable throughput without encountering significant congestion, further corroborated by the consistent data transfer of approximately 2.21 GBytes in each test. The overall stability in the average bitrate and the absence of packet retransmissions highlights the efficiency and reliability for data transmission from *user-vm* to PC. In the evaluation of the PC to *user-vm* network performance using *vhost*, the results consistently displayed a bandwidth ranging from 397 Mbits/sec to 438 Mbits/sec, with an average of approximately 410 Mbits/sec. The consistent high bandwidth and low retransmission rate shows the efficacy of *vhost* in maintaining robust network communication between the PC and *user-vm* at RPi4.

In the comparison between scenarios utilizing *vhost* and those without, a notable enhancement in network performance is evident. Specifically, the data transmission from *user-vm* RPi4 to PC exhibits an approximately 48.7% improvement in the average bitrate. Simultaneously, the PC to *user-vm* RPi4 direction demonstrates a more modest, but still significant, improvement of around 4.4%. The use of *vhost* clearly mitigates network congestion and optimizes bandwidth utilization, thereby ensuring more robust and efficient data communication in diverse transmission scenarios.

A related approach using VirtIO devices on top of KVM was evaluated on an R120-T34 (1U Server) [50]. In this approach, the authors evaluated the *virtio-iommu* using *iPerf3* benchmark and obtained an average of 420 Mbps for the transmission (i.e., tx) with *vhost off* scenario and 464 Mbps with *vhost on*. In a similar effort, Alonso et al. [51] analyzed the Xen hypervisor performance on the network using *iPerf3* benchmark. In the measurements on network connection over Petalinux running on Xen DomU with paravirtualized network device, the authors obtained an average of 500 Mbps for the sender and 644 Mbps for the receiver.

B. *virtio-blk*

The *virtio-blk* is a virtualization technology that provides an efficient and standardized way for virtual machines to access block storage devices [30]. It is designed to improve I/O performance by reducing overhead and eliminating the need for additional software or drivers. The *virtio-blk* is widely used in virtual environments as it enables guests to access host storage directly, without compromising security or performance. This technology allows for the easy migration of virtual machines across different hypervisors and operating systems while ensuring compatibility and optimal performance. Overall, the *virtio-blk* is a key component of virtualization infrastructure, providing efficient and reliable block storage access for virtual machines.

The Flexible I/O (*fio*) benchmark is a widely-used and versatile tool designed for measuring and characterizing the performance of storage subsystems, such as disk drives, storage area networks, and network-attached storage devices [52]. The *fio* benchmark provides detailed metrics and statistics, which facilitate a comprehensive understanding of the storage system's performance, enabling comparisons across different systems and configurations. In this paper, the *fio* benchmark was employed to evaluate the use of *virtio-blk* on top of seL4 by the proposed approach, shedding light on its performance characteristics and overall efficiency.

We conducted a series of *fio* benchmarks to evaluate the performance of various storage configurations under the scenario utilizing the seL4 hypervisor on a Raspberry Pi 4 with 4GB of RAM. The *fio* benchmark was executed within the *user-vm*. The *fio* tool was configured with the *posixaio* I/O engine and a random write workload. Seven block sizes were utilized: 4KB, 16KB, 64KB, 256KB, 1024KB, 2048KB, and 4096KB. A 64MB file was used for each block size with a single job running for 60 seconds. The benchmark was set to be *time-based* to maintain a consistent workload intensity throughout the test duration. To ensure the accurate measurement of I/O operations, the benchmark was set to be time-based, and *end_fsync* was used, causing *fio* to synchronize file system buffers with storage devices at the end of each test. The results of the performed *fio* benchmark can be found in Table II.

The conducted evaluation revealed a decline in IOPS from 3842 to 15 as block size increased, coupled with a consistent improvement in bandwidth, peaking at 60.1 MiB/s for 1024k block size. However, a corresponding increase in latency was

TABLE II
fio BENCHMARK RESULTS ON THE *user-vm* WITH SEL4 HYPERVISOR

Block Size	IOPS	BW (MiB/s)	Avg. Latency (usec)	99.99th Percentile (usec)	CPU (%)	Disk Utilization
4 KB	3842	15.0	212.68	24773	usr=11.44, sys=25.99	52.46%
16 KB	2155	33.7	367.96	28705	usr=5.90, sys=23.36	59.94%
64 KB	768	48.0	795.45	18482	usr=2.20, sys=27.57	68.50%
256 KB	230	57.7	2529.93	94897	usr=1.12, sys=26.64	64.72%
1024 KB	60	60.1	10329.42	32900	usr=1.38, sys=23.07	62.19%
2048 KB	30	60.6	20380.61	46400	usr=1.35, sys=23.18	62.54%
4096 KB	15	61.4	40144.14	73925	usr=1.65, sys=22.95	62.92%

observed, reaching 40144.14 usec for 4096k block size. At lower block sizes, like 4k and 16k, a more balanced performance was observed with IOPS of 3842 and 2155, and average latencies of 212.68 usec and 367.96 usec respectively. The results generally demonstrated a trade-off among the evaluated metrics as the block size was adjusted.

The optimal block size configuration depends on the application requirements and constraints. For applications prioritizing higher IOPS and lower latency, smaller block sizes proved a more suitable performance profile. On the other hand, for bandwitdth-intensive applications where higher latency is acceptable, larger block sizes could be more appropriate.

VI. CONCLUSION AND FUTURE WORK

This paper presented the first publicly available implementation of the VirtIO standard for virtual machines running on top of the seL4 hypervisor. This implementation aims to enhance architectural flexibility and accelerate development speed by leveraging the efficient communication interface provided by the VirtIO standard and QEMU, eliminating the need for native drivers. As a Type-1 hypervisor, seL4 offers unique advantages in terms of security and isolation, which are critical for certain use cases that demand a small trusted computing base. The proposed feature paves the way for future work to optimize the performance of seL4-based virtualization solutions without compromising its core benefits in security and isolation. By continuing to develop and refine seL4's virtualization capabilities, it is possible to create a more robust and efficient platform for consolidating workloads and improving resource utilization in critical systems.

Currently, our proposed architecture defines the characteristics of virtual machines, such as the amount of RAM and resources, in a static manner. However, as future work, we aim to make these characteristics dynamic and customizable during runtime. This will provide greater flexibility and adaptability to the system, allowing for better resource allocation and management. By allowing the configuration of VMs to change dynamically, the system can adapt to different workload requirements and optimize the usage of available resources. We plan to explore different approaches to achieve this goal, including dynamic memory allocation and resource sharing between VMs. Moreover, it is essential to note that the current inter-VM implementation holds room for optimization, and we firmly believe that making these enhancements will significantly bolster the system's overall performance. While this paper introduces a novel approach to implementing the VirtIO standard for virtual machines on the seL4 hypervisor, it is an intermediate step in our ongoing research. This study serves as a cornerstone, providing crucial groundwork and insights upon which subsequent advancements and versions will be constructed, inching us closer to our ultimate objective of a dynamic, robust, and efficient virtualization solution.

REFERENCES

[1] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39–47, 2005.

[2] G. Heiser, "The role of virtualization in embedded systems," in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, ser. IIES '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 11–16. [Online]. Available: https://doi.org/10.1145/1435458.1435461

[3] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, "Emerging trends, techniques and open issues of containerization: A review," *IEEE Access*, vol. 7, pp. 152 443–152 472, 2019.

[4] A. Iqbal, C. Pattinson, and A.-L. Kor, "Performance monitoring of virtual machines (vms) of type i and ii hypervisors with snmpv3," in *2015 World Congress on Sustainable Technologies (WCST)*, 2015, pp. 98–99.

[5] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, "Breaking up is hard to do: Security and functionality in a commodity hypervisor," in *2011 SOSP: ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 189–202. [Online]. Available: https://doi.org/10.1145/2043556.2043575

[6] T. Shimada, T. Yashiro, N. Koshizuka, and K. Sakamura, "A real-time hypervisor for embedded systems with hardware virtualization support," in *2015 TRON Symposium (TRONSHOW)*, 2015, pp. 1–7.

[7] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems," in *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, ser. OpenAccess Series in Informatics (OASIcs), M. Bertogna and F. Terraneo, Eds., vol. 77. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, pp. 3:1–3:14. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/11779

[8] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look mum, no vm exits! (almost)," 2017. [Online]. Available: https://arxiv.org/abs/1705.06932

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 164–177. [Online]. Available: https://doi.org/10.1145/945445.945462

[10] C. Dall and J. Nieh, "Kvm/arm: The design and implementation of the linux arm hypervisor," *SIGPLAN Not.*, vol. 49, no. 4, p. 333–348, feb 2014. [Online]. Available: https://doi.org/10.1145/2644865.2541946

[11] G. Heiser, G. Klein, and J. Andronick, "Sel4 in australia: From research to real-world trustworthy systems," *Commun. ACM*, vol. 63, no. 4, p. 72–75, mar 2020. [Online]. Available: https://doi.org/10.1145/3378426

[12] E. d. Matos and M. Ahvenjärvi, "sel4 microkernel for virtualization use-cases: Potential directions towards a standard vmm," *Electronics*, vol. 11, no. 24, 2022. [Online]. Available: https://www.mdpi.com/2079-9292/11/24/4201

[13] M. S. Tsirkin and C. Huck, "Virtual i/o device (virtio) version 1.1," *OASIS Committee*, 2018.

[14] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in *2015 IEEE International Conference on Cloud Engineering*, 2015, pp. 386–393.

[15] S.-W. Li, J. S. Koh, and J. Nieh, "Protecting cloud virtual machines from hypervisor and host operating system exploits," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1357–1374. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/li-shih-wei

[16] C. Moratelli, R. Tiburski, S. F. Johann, E. Moura, E. De Matos, and F. Hessel, "Mips and risc-v: Evaluating virtualization trade-off for edge devices," in *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)*, 2022, pp. 1–6.

[17] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, p. 412–421, jul 1974. [Online]. Available: https://doi.org/10.1145/361011.361073

[18] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39–47, 2005.

[19] O. Tickoo, R. Iyer, R. Illikkal, and D. Newell, "Modeling virtual machine performance: Challenges and approaches," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 3, p. 55–60, jan 2010. [Online]. Available: https://doi.org/10.1145/1710115.1710126

[20] F. Xu, F. Liu, H. Jin, and A. V. Vasilakos, "Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions," *Proceedings of the IEEE*, vol. 102, no. 1, pp. 11–31, 2014.

[21] R. T. Tiburski, C. R. Moratelli, S. F. Johann, E. de Matos, and F. Hessel, "A lightweight virtualization model to enable edge computing in deeply embedded systems," *Software: Practice and Experience*, vol. 51, no. 9, pp. 1964–1981, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2968

[22] A. Aguiar and F. Hessel, "Embedded systems' virtualization: The next challenge?" in *Proceedings of 2010 21st IEEE International Symposium on Rapid System Protyping*, 2010, pp. 1–7.

[23] K. Han, S. Al Blooshi, N. Alnuaimi, E. Al Nuaimi, E. de Matos, and R. Psiakis, "Improving drone mission continuity in rescue operations with secure and efficient task migration," in *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)*, 2022, pp. 1–6.

[24] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "Sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 207–220. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[25] J. Shropshire, "Analysis of monolithic and microkernel architectures: Towards secure hypervisor design," in *2014 47th Hawaii International Conference on System Sciences*, 2014, pp. 5008–5017.

[26] A. Lyons, K. McLeod, H. Almatary, and G. Heiser, "Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190539

[27] G. Heiser, "The sel4 microkernel–an introduction," 2020. [Online]. Available: https://sel4.systems/About/seL4-whitepaper.pdf

[28] Y. Chen, J. Wu, and S. Yang, "Virtio-based i/o virtualization on arm platforms," in *Proceedings of the 2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 2017.

[29] J. Hao, X. Huang, W. Chen, and M. Chen, "A high-performance virtio implementation for kvm," in *Proceedings of the 2015 ACM SIGPLAN Conference on Virtual Execution Environments*. ACM, 2015.

[30] R. Russell, "Virtio: Towards a de-facto standard for virtual i/o devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, p. 95–103, jul 2008. [Online]. Available: https://doi.org/10.1145/1400097.1400108

[31] S. Bandara, A. Sanaullah, Z. Tahir, U. Drepper, and M. Herbordt, "Enabling virtio driver support on fpgas," in *2022 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2022, pp. 1–8.

[32] F. Bellard, "QEMU, a fast and portable dynamic translator," in *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, apr 2005. [Online]. Available: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator

[33] S. Park, K. Kim, and H. Kim, "Ambient virtio: Io virtualization for seamless integration and access of devices in ambient computing," *IEEE Systems Journal*, pp. 1–12, 2022.

[34] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Zhuang Hui, "A secure and formally verified linux kvm hypervisor," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1782–1799.

[35] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi, "Embedded hypervisor xvisor: A comparative analysis," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015, pp. 682–691.

[36] A. Oliveira, J. Martins, J. Cabral, A. Tavares, and S. Pinto, "Tz- virtio: Enabling standardized inter-partition communication in a trustzone-assisted hypervisor," in *2018 IEEE 27th International Symposium on Industrial Electronics (ISIE)*, 2018, pp. 708–713.

[37] H. Li, X. Xu, J. Ren, and Y. Dong, "Acrn: A big little hypervisor for iot development," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 31–44. [Online]. Available: https://doi.org/10.1145/3313808.3313816

[38] seL4 Project, "Virtualisation on seL4," 2023. [Online]. Available: https://docs.sel4.systems/projects/virtualization/

[39] ——, "CAmkES VMM," 2022. [Online]. Available: https://docs.sel4.systems/projects/camkes-vm/

[40] seL4 Project, "The seL4 Microkit (was "seL4 Core Platform")," 2022. [Online]. Available: https://sel4.atlassian.net/browse/RFC-5

[41] B. Leslie and G. Heiser, "The seL4 Core Platform," 2022. [Online]. Available: https://trustworthy.systems/projects/TS/sel4cp/2011-draft-spec.pdf

[42] ——, "Evolving seL4CP Into a Dynamic OS," 2022. [Online]. Available: https://trustworthy.systems/projects/TS/sel4cp/2203-report-dynamic.pdf

[43] seL4, "seL4 Microkit," https://github.com/seL4/microkit, 2023, [Accessed 27-Sep-2023].

[44] tiiuae, "GitHub - tiiuae/tii_sel4_build," https://github.com/tiiuae/tii_sel4_build, 2023, [Accessed 01-Apr-2023].

[45] G. Motika and S. Weiss, "Virtio network paravirtualization driver: Implementation and performance of a de-facto standard," *Computer Standards & Interfaces*, vol. 34, no. 1, pp. 36–47, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920548911000559

[46] esnet, "GitHub - esnet/iperf: iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool," https://github.com/esnet/iperf, 2023, [Accessed 01-Apr-2023].

[47] V. Maffione, L. Rizzo, and G. Lettieri, "Flexible virtual machine networking using netmap passthrough," in *2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, 2016, pp. 1–6.

[48] J. Tan, C. Liang, H. Xie, Q. Xu, J. Hu, H. Zhu, and Y. Liu, "Virtio-user: A new versatile channel for kernel-bypass networks," in *Proceedings of the Workshop on Kernel-Bypass Networks*, ser. KBNets '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 13–18. [Online]. Available: https://doi.org/10.1145/3098583.3098586

[49] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, "An argument for increasing tcp's initial congestion window," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 3, p. 26–33, jun 2010. [Online]. Available: https://doi.org/10.1145/1823844.1823848

[50] E. Auger, "viommu/arm: full emulation and virtio-iommu approaches," in *KVM Forum*, 2017.

[51] S. Alonso, J. Lázaro, J. Jiménez, L. Muguira, and A. Largacha, "Analysing the interference of xen hypervisor in the network speed," in *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)*, 2020, pp. 1–6.

[52] J. Axboe, "Fio's documentation," https://fio.readthedocs.io/en/latest/index.html, 2017, [Accessed 01-Apr-2023].