Architecting Trust: Building Secure and High-Performance Confidential VMs

by

Shashvat Srivastava

S.B. in Computer Science and Engineering and in Mathematics
Massachusetts Institute of Technology 2023

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2023

Authored by: Shashvat Srivastava
Department of Electrical Engineering and Computer Science
August 18, 2023

Certified by: Mengjia Yan
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Architecting Trust: Building Secure and High-Performance Confidential VMs

by

Shashvat Srivastava

Submitted to the Department of Electrical Engineering and Computer Science
on August 18, 2023, in Partial Fulfillment of the
Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Recent research in TEE (Trusted Execution Environment) design have focused on the development of *confidential VMs* — virtual machines completely protected by secure hardware. All major CPU vendors have rolled out support for VM based TEEs — AMD created SEV (2017), Intel created TDX (2020), and ARM launched CCA (2021). Confidential VMs are a quite promising new technology as they are significantly more user-friendly, allow existing applications to run without modifications, and have better performance compared to process-based TEE. However, confidential VMs still face two large design challenges: security and performance.

In the first part of this thesis, we propose a secure confidential VM design on the RISC-V platform, which currently has no official confidential VM support. We specifically focus on the task of secure CPU virtualization and build a security monitor that hides the virtual CPU register state from the hypervisor during context switches. To allow the hypervisor to properly handle interrupts and emulate instructions, we summarize a specification listing which registers need to be exposed in specific scenarios.

In the second part of this thesis, we aim to improve the network I/O performance of existing confidential VMs. The hardware protections of TEEs create additional I/O overhead in confidential VMs, and Trusted I/O (TIO) is a promising solution to reduce this overhead. However, TIO has several drawbacks — it relies on hardware support from the I/O device and expands the Trusted Computing Base (TCB) to include these TIO devices. Furthermore, TIO devices will not be commercially available for several years. We aim to create a I/O solution that can reach the performance of TIO without relying on TIO devices. In particular, we present Folio, a system for high-performance network I/O compatible with AMD SEV-SNP. Compared to network I/O in a non-TEE VM, Folio performs only a single extra memory-copy of packet data. Our extensive evaluation shows that Folio performs only 6% worse than the ideal TIO solution.

Thesis Supervisor: Mengjia Yan
Title: Assistant Professor

Acknowledgments

First and foremost, I would like to thank my parents, who have supported me throughout my time at MIT and without whom I would not even be here. I also have to thank all my friends, who accompanied (and distracted) me through countless hours of study and work. Of course, I must also thank Dr. Mengyuan Li, with whom I worked with on both projects contained in this thesis over a span of more than 3 semesters. Towards the end, we spent many nights discussing and resolving issues. Finally, I would like thank Professor Mengjia Yan for supervising this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Trusted execution environments (TEEs) [11] are used to securely run sensitive software. Typically, they guarantee the confidentiality and integrity of the software and data run within. The purpose of a TEE can be motivated by a mistrust of the underlying operating system. Modern operating systems are large and complex; the Linux kernel itself has over 26 million lines of code [48]. It is impossible to guarantee the correctness of an operating system, and any bugs in the OS are prone to exploitation by an adversary. For instance, an adversary may be able to manipulate an existing application, such as a web browser, into exploiting the operating system and compromising another secure application on the computer. One way to model these vulnerabilities is to treat the operating system itself as a malicious, adversarial entity. Using specific hardware features, TEE systems can secure applications and protect them from the OS.

TEEs are becoming increasingly more important in the context of cloud computing. Concerns over the security of cloud data often deter many potential cloud users from adopting cloud services. Typically, the cloud service provider (CSP) has unfettered access to the underlying cloud hardware through hypervisor software and is free to examine sensitive cloud user data. Even if the CSP can be trusted to not access user data, it has been shown that malicious users sharing physical cloud resources are capable of determining sensitive details through side-channel attacks [44, 18]. Consequently, both CSPs and cloud users are challenged to find solutions that ensure the

security of cloud workloads, and TEEs provide a natural solution.

The significant market demand for TEE solutions is reflected in the confidential-computing products launched by major CSPs. AWS, Google, and Microsoft all support AMD's SEV technology [2, 15, 34]. Moreover, Microsoft also supports Intel's SGX technology [33], and AWS even developed their own TEE system, AWS Nitro [1].

The interest in the creation and application of TEEs has lead to a wealth of different TEE designs. Originally, TEE designs could be described as enclaves that protected individual, isolated applications (see *e.g.* Intel Software Guard Extensions (SGX) [49], RISC-V Keystone [27], Penglai [14]). However, the industry has recently shifted its focus away from enclave-based TEEs and towards VM-based TEEs that protect entire virtual machines [23, 22, 4, 7, 19]. All major CPU vendors have rolled out support for VM based TEEs — AMD created SEV (2017), Intel created TDX (2020), and ARM launched CCA (2021) — and Intel has even decided to discontinue SGX support for desktop processors [42]. VM-based TEEs are significantly more user-friendly compared to enclave-based TEEs as they protect the entire virtual machine. This approach allows applications to execute securely without any source code modifications. Furthermore, multiple applications can run and communicate within the same TEE without any hassle or overhead.

Despite the great number of TEE designs, there are still many challenges in the field of creating secure TEE designs for new CPU architectures and creating higher-performance TEEs for existing CPU architectures. In this thesis, we focus on these two challenges.

In the first part of this thesis, we propose an open-sourced and customizable VM-based TEE design for the RISC-V platform[1]. While there are various RISC-V TEE systems that secure single applications, there are currently no systems that secure an entire virtual machine (running on RISC-V hardware)[14, 27]. Securing an entire virtual machine poses a new set of challenges. For instance, unlike Intel and AMD, a RISC-V TEE does not have the option of modifying hardware to aid the TEE; it must make use of existing facilities and hardware mechanisms. A RISC-V confidential VM

[1]Specifically, we focus on the component that protects a VM's virtual CPU.

relying only on existing RISC-V hardware features and primitives will undoubtedly be a great benefit to the RISC-V community.

In the second part of this thesis, we focus on improving the I/O performance of existing VM-based TEEs. Recent work highlights the poor I/O performance of TEE systems [53, 25]. In an ideal world, there would be no performance gap between VMs and confidential VMs. In practice, the additional protections introduced by VM-based TEEs seem to cause non-negligible I/O performance overhead. To combat these problems, both AMD and Intel have published whitepapers on trusted I/O devices (TIO) [5, 20].

To understand the purpose of TIO, consider the interaction between a traditional I/O device with memory encryption, a feature often used to protect memory in confidential VMs. For an I/O device to access data in a confidential VM, that data must first be decrypted, which incurs a performance penalty. More importantly, I/O devices cannot perform DMA operations directly on VM's memory; instead, they must perform DMA operations to and from a special region of shared memory called the *bounce buffer*. When the VM wants write (or read) from the the I/O device, it must first copy data to (or from) the bounce buffer from its own private memory. This memory-copy, as well as the allocation and release of memory from the bounce buffer, incurs a second performance penalty.

This second problem could be mitigated if the I/O device could encrypt data as part of the DMA operation. The solution is currently impossible because peripheral devices, including network I/O cards, are outside the trust boundary and cannot be trusted with encryption or decryption keys. TIO extends the trust boundary from the CPU to include peripheral devices and grants TIO devices the ability to DMA directly to the VM's private memory, eliminating TEE-related I/O overhead.

Trusted I/O devices are promising, but it is unclear if they are immediately useful. TEE solutions depend on hardware, which leads to long product development cycles. For instance, while the Intel TDX whitepaper was published in 2020, there are no commercial implementations as of the time of writing in 2023. Additionally, trusted I/O devices introduce a new threat models where trusted I/O devices are included in

the trusted computing base (TCB). It is risky to extend trust to trusted I/O devices as thorough evaluation of the security of their design requires time from both industry and academia. The security of confidential VM is limited by the weakest link in the TCB, which may be the new I/O device. The addition of trusted I/O devices may create new attack vectors that could undermine the security of the entire confidential VM.

Therefore, in the second part of this thesis, we investigate if it is possible to achieve high I/O performance comparable to trusted I/O devices without sacrificing any security guarantees.

Chapter 2

Background

2.1 RISC-V TEE Designs

2.1.1 RISC-V ISA

RISC-V is an open-sourced ISA following RISC (reduced instruction set computer) principles. RISC-V follows a modular approach, with different standard extensions that add different features to the ISA. For example, the 'M' extension adds basic integer multiplication and division to the ISA, and the 'H' extension adds support for virtualization through hypervisor-specific instructions [50, 51]. Crucially, RISC-V is completely royalty-free, allowing anyone to create processors based on the RISC-V ISA. Moreover, its open-source nature allows the ISA to be customized for different use cases. As such, it is already being used to create specialized microprocessor designs for home appliances, robotics, and autonomous vehicles [16].

Since its creation in 2010 [9], RISC-V has been following a growing trend of adoption and use. Companies such as Samsung, Western Digital, NVIDIA, and Qualcomm use (or plan to use) RISC-V in devices such as SSDs, HHDs, and GPUs [16]. Additionally, companies such as SiFive and Experanto offer RISC-V CPUs. Notably, Esperanto's ET-SoC-1 Chip contains 1000 low-powered RISC-V cores for use in machine learning applications, showcasing RISC-V's use in upcoming technologies [37].

2.1.2 RISC-V Hardware Features

Before we can dive into TEE systems for RISC-V, we must introduce the different RISC-V hardware features used to build these systems. There are two key hardware features than can be leveraged to build a TEE on RISC-V hardware: a higher privilege mode, machine-mode, and physical memory protection.

Machine mode (M-mode) is an even higher privilege mode than supervisor mode (S-mode) and user mode (U-mode) [51]. M-mode is the highest privilege mode on RISC-V systems and controls the access to all physical resources and interrupts and is therefore typically used to write firmware.

Crucially, M-mode is uninterruptible from lower privilege levels, but is capable of trapping exceptions and interrupts to lower privilege levels, making it a useful tool for security purposes. Just as the kernel running in S-mode is protected and isolated from userspace programs, low-level M-mode firmware is protected and isolated from the kernel. Consequently, a secure enclave running in M-mode is protected against the operating system or hypervisor.

M-mode can also be used to implement supervisor binary interfaces (SBI) for the operating system. These SBIs can control and manipulate hardware features not directly available to S-mode. An example of this is the PMU SBI in OpenSBI[1] which is used to manage a RISC-V performance monitor unit [39]. In the context of enclaves, SBIs are used to create, run, resume, and destroy enclaves. They can also be used to facilitate communication between the programs running within the enclave and the operating system.

The second primitive, physical memory protection (PMP), is used to control access to physical memory[51]. Each CPU has several PMP registers (usually 16) that specify which regions of memory can be accessed by the CPUs. Each register specifies read, write, and execute permissions for a region of memory. The PMP registers can only be modified while in M-mode. Thus, M-mode code can be used in conjunction with the PMP registers to prevent the operating system or hypervisor from reading specific memory regions, such as the internal memory of a process or virtual machine.

[1] OpenSBI is an open-source platform for writing RISC-V firmware running in M-mode.

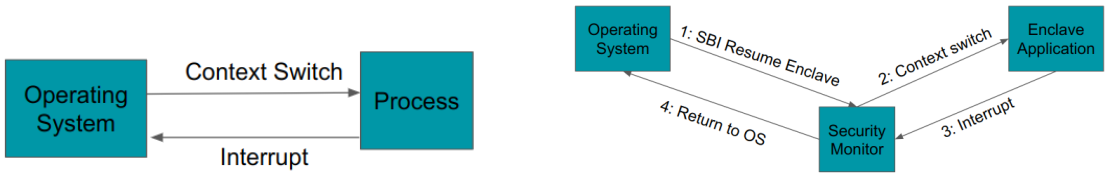
2.1.3 RISC-V Keystone

Keystone is an open framework for creating TEEs on RISC-V hardware [27]. Keystone introduced several novel contributions, including the concept of a customizable TEE. We focus on the core *security monitor* component as it is the component most relevant to us.

The Keystone framework uses a trusted security monitor, running in M-mode, to manage enclaves and enforce enclave security. Each enclave has its own separate physical memory region; the security monitor uses the PMP registers to isolate each memory region and restrict access to the enclave's memory to the enclave's underlying application. These PMP registers are manipulating during context switches to and from the enclave application.

Crucially, the security monitor does not have to manage any resources (*e.g.* memory management), allowing the TCB to remain small. Instead, each enclave is packaged with a special *runtime* running in S-mode that does memory management and more. The application can then run in U-mode unmodified.

Figure 2-1 contrasts context switches in Keystone to traditional context switches [24]. To resume an enclave, the operating system using a special Keystone SBI to trigger the security monitor (1). When this happens, the security monitor manipulates the PMP registers to grant the current CPU access to the enclave's memory. By default, PMP registers are configured to deny access to any enclave's memory. The security monitor then resumes the enclave application (2). And interrupt or traps during the execution of the enclave application are handled by the security monitor *first* (3). Upon such a trap, the security monitor stores the enclave application's execution state (*e.g.* register state, program counter) within its own private memory and reconfigures PMP registers to deny all access to enclave memory. The security monitor finally resumes the execution of the operating system (4). Through this careful process, the security monitor guarantees that enclave memory access is restricted to the enclave application.



(a) Traditional (b) Keystone

Figure 2-1: Context switch comparison.

2.2 AMD SEV

AMD SEV (secure encrypted virtualization) was AMD's first attempt at a trusted execution environment holding an entire virtual machine[23]. AMD SEV initially focused on just protecting and encrypting the virtual machine's memory. Two additional versions, AMD SEV-ES and AMD SEV-SNP add additional crucial security features [22, 46].

AMD SEV provides memory isolation and protection by encrypting all virtual machine data. Each virtual machine is associated with an ASID (identifying) tag. SEV hardware automatically tags all code and hardware with an ASID. Whenever data is written to physical memory, it is encrypted using a secret key according to the ASID. Whenever data is read, it is decrypted according to that secret key. Encryption and key management is performed by the trusted AMD hardware.

This approach prevents the hypervisor from reading a secured virtual machine's memory contents. Although the hypervisor is able to create, run, pause, and otherwise manage virtual machines, it is not allowed to access the encryption keys or modify its own ASID. If the hypervisor tries to read a virtual machine's memory, the AMD hardware will decrypt memory using the hypervisor's decryption key, not the virtual machines. As a result, the hypervisor will only be able to read garbage data.

2.2.1 AMD SEV-ES

The original AMD SEV design does not prevent two crucial attack vectors. Part of the hypervisor's function is to emulate instructions for the guest virtual machines through exceptions and perform page table management, but these functions create

severe vulnerabilities. For example, when the hypervisor handles an exception or interrupt, it can access the entire CPU state of the virtual machine. If the CPU state holds part of some encryption secret key, the hypervisor is capable of learning the entire key.

AMD SEV-ES (encrypted state) adds support to encrypt the guest's CPU state while also allowing the hypervisor to emulate critical instructions. The overall process is summarized in Figure 2-2. When the VM first exits, The AMD hardware raises a special #VC exception, which the guest OS uses to transfer specific registers into the GHCB block. The hypervisor emulates the instruction based on GHCB data. The guest OS then uses the results in the GHCB block to resume after the original instructions.

In general, exits from the VM can be divided into automatic exits and non-automatic exits. On an automatic exit, the AMD CPU automatically encrypts the CPU state. If the hypervisor tries to read the CPU state, they will only see garbage values. In certain cases, the hypervisor must have access to some of the CPU state in order to emulate an instruction. A good example is the CPUID instruction; depending on the value of the eax register, the hypervisor will return different information about the guest machine.

The guest-host communication block (GHCB) is a shared memory region that allows the hypervisor to correctly emulate these instructions. When an instruction must be emulated, a non-automatic exit (NAE) exception is triggered. The guest supervisor will trap this exception, and will copy any relevant registers to the GHCB. The guest supervisor will then trigger an AE, causing the CPU state to be encrypted; the GHCB will remain in plaintext. The hypervisor can then use the register values in the GHCB to properly emulate the instruction.

AMD specifies a GHCB protocol that should be followed by the guest operating system and hypervisor. It summarizes the possible different types of non-automatic exceptions and what register values need to be shared in the GHCB. In addition to encrypting the register state, AMD hardware also automatically calculates a measurement of the register state. This measurement is referenced during subsequent

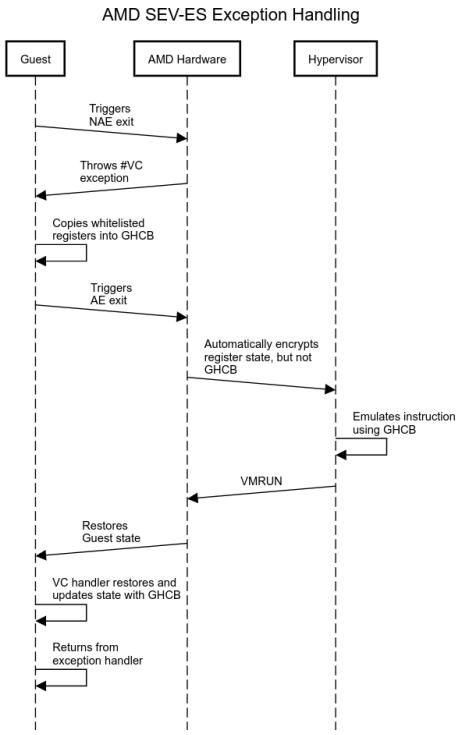


Figure 2-2: Instruction emulation workflow in AMD SEV-ES.

`VMRUN` operations to detect replay attacks and register tampering.

2.2.2 AMD SEV-SNP

AMD SEV-SNP was the next iteration in AMD SEV design, and protects guest VMs against attacks exploiting page table management [46, 31, 35, 36]. SNP mainly addresses the issue of data integrity. Although a malicious hypervisor cannot read encrypted memory, it can still attempt to modify a guest VM's memory (even without knowing the guest's encryption key). This type of attack can potentially compromise an application running inside a VM. For example, such an application could be vulnerable to replay attacks. Even without knowing the encryption key, the hypervisor could try to launch a replay attack on such an application.

The key addition SNP brings is the *Reverse Map Table* (RMP). The RMP is a table holding one entry for each page of physical memory in the system. Each entry holds the owner of the page, whether it be the hypervisor or a specific virtual machine. Crucially, the RMP provides bijectivity: each physical page of memory can only have

one owner. The RMP thus prevents the hypervisor from assigning the same physical page to two different VMs, or from creating a duplicate page table entry between itself and a VM.

The RMP table is referenced whenever the CPU does a page-table walk (called an *ownership check*), specifically checking if the id of the current executor matched the listed owner in the RMP table. The hardware rejects illegal accesses by throwing a special page-fault. If the RMP entry matches, the virtual address is added to the TLB. This process prevents the hypervisor from violating the integrity of the guest VM as the hypervisor is unable to write to guest memory.

SNP also protects against I/O operations using direct memory access (DMA) and interfaces with the IOMMU to allow devices to write to memory when appropriate. The IOMMU also checks the RMP table, preventing I/O devices from corrupting memory. However, there are instances where an I/O device *should* be allowed to write to guest memory. To handle these cases, SNP allows pages to be marked as shared or private. The owner of a page can request that a page be marked as shared, which removes that page's entry from the RMP table. This allows an I/O device to write to the memory region.

Protection Against TLB Poisoning

SEV-ES also suffers from a second, more subtle type of attack known as the TLB poisoning attack [32]. To protect against this attack, SNP makes an additional check during each `VMRUN` that determines if the TLB needs to be flushed, and flushed the TLB if it is necessary.

Removal of `VMEXIT` Measurement

The ownership check feature of SNP also stops a malicious hypervisor from tampering with the encrypted register states of SEV-ES. Thus, measurements of the register state during `VMEXIT`s are not necessary in SNP, and SNP hardware does not make these measurements or checks.

2.3 Networking

There are three commonly used networking techniques used in virtualization: VirtIO, SR-IOV, and DPDK. We also briefly discuss TIO.

2.3.1 VirtIO

VirtIO [45] is a popular networking technology and the default configuration in the Linux KVM (Kernel-based Virtual Machine). In VirtIO, the VM interacts with I/O devices that are emulated by the hypervisor. Figure 2-3 shows a common VirtIO networking path, and also highlights three main sources of overhead: routing within the VM, routing within the host, and emulated interrupts.

When a network application needs to send data, it typically makes a system call, which triggers a context switch to the guest kernel. The guest kernel then performs several layers of packet processing, before finally performing a "virtqueue kick." Each of these actions create overhead, and are cumulatively referred to as overhead from routing within the VM.

After the virtqueue kick action, network packets must go through several layers of emulation. This emulation requires communication between QEMU, which actually emulates the device, and the host kernel, which interacts with the NIC. This cumulative overhead is called overhead from routing within the host.

Finally, notifying the network application requires interrupts, which takes additional overhead. Interrupts cannot be delivered to the VM directly. Instead, the physical interrupt triggers the KVM, which must then emulate and inject that interrupt. During the next `VMRUN`, hardware automatically checks for any pending emulated interrupts and redirects the CPU to the guest VM's interrupt handler if necessary. All of these overheads are cumulatively referred to as emulated interrupt overhead.

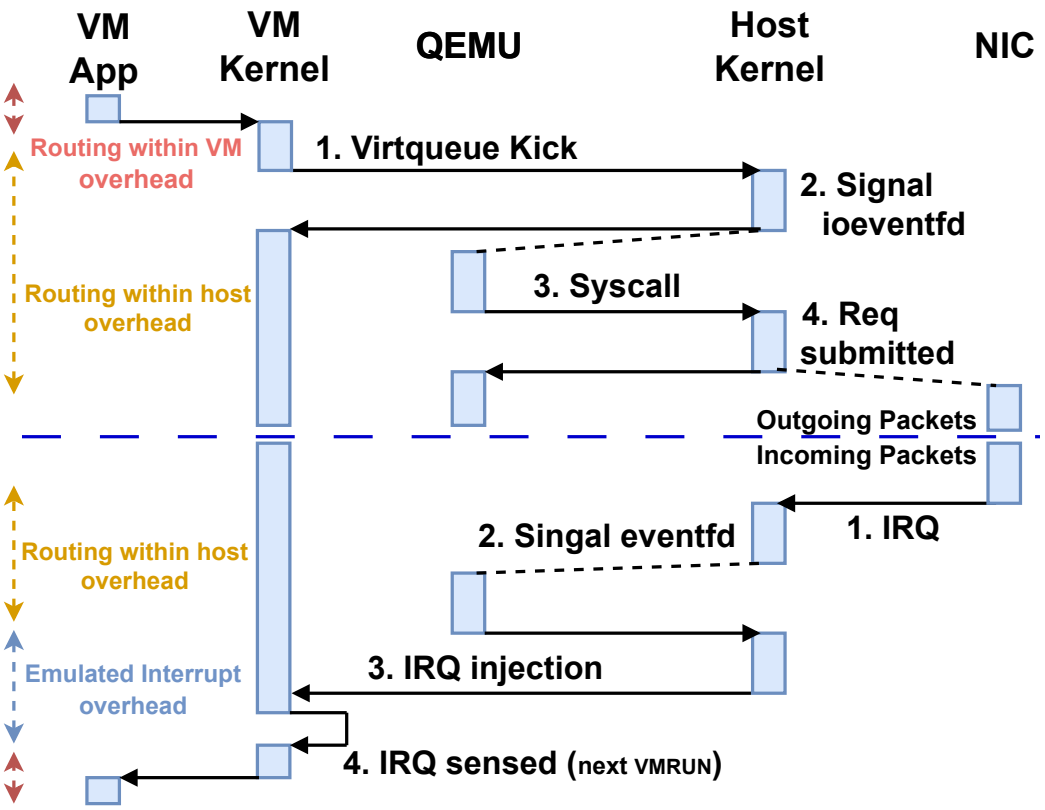


Figure 2-3: Common VirtIO network path.

2.3.2 SRIOV

To avoid the expensive overhead of routing within the host side, hypervisors can directly assign I/O devices to virtual machines using PCI passthrough. Although PCI passthrough removes much overhead, this technique is unfavorable for several reasons. Each virtual machine would need its own physical NIC, which may be expensive and unfeasible in the context of cloud computing. Furthermore, the guest VM would gain complete access to the I/O device, which could be a security vulnerability (*e.g.*, the guest VM could try to maliciously modify the device's firmware).

SR-IOV, an extension of the PCIe standard, helps solve many of these issues. An SR-IOV capable I/O device can split its resources among a Physical Function (PF) and several Virtual Functions (VFs). Each PF and VF appears as its own separate device with its own PCI Express Requester ID, allowing the IOMMU to differentiate their I/O traffic and interrupt signals. Thus, SRIOV allows the hypervisor to bind a VF to the VM directly using PCI passthrough, significantly reducing the VM's I/O

overhead. SRIOV also allows a CSP to service many VMs using a single NIC (*e.g.*, an Intel 82599 10GB NIC can create up to 63 VFs per physical port). Crucially, only the PF is capable of configuring the device, so directly assigning a VF does not create any security vulnerabilities.

2.3.3 DPDK

Intel DPDK is a software framework that boosts network performance by avoiding the overhead of I/O interrupts and bypassing the kernel. DPDK provides a series of userspace APIs through the construction of an Environment Abstraction Layer (EAL) to assist DPDK applications in fast network processing. DPDK applications run in userspace, thus avoiding the overhead of making system calls and switching to kernel space. DPDK libraries also provide Poll Mode Drivers (PMD), which allow DPDK applications to send and receive packets without relying on interrupts. The DPDK library itself is also heavily optimized, employing a zero-copy architecture and several hardware-based optimizations such as using huge pages and cache alignment.

2.3.4 Trusted I/O

Trusted I/O (TIO) devices are specifically designed to reduce I/O overhead from security features of confidential VMs. Confidential VMs can establish a trusted relationship with a TIO device using the TEE Device Interface Security Protocol (TDISP) [5, 20], which grants TIO devices access to the private memory and memory encryption keys of the confidential VM. Thus, TIO devices can execute DMA reads and writes directly and avoid all bounce buffer overhead.

Chapter 3

VM-based TEE for RISC-V

In this chapter, we discuss the design and implementation of the secure CPU virtualization component of a VM-based TEE for RISC-V. This component is the part that protects or hides a virtual CPU from the hypervisor like AMD SEV-ES does. The two other components for a VM-based TEE, memory and page-table protection and secure I/O, were researched by others in parallel.

Key Challenges

With respect to CPU virtualization, there are two core problems that must be solved to securely run a virtual machine. First, the hypervisor must not be able to access and manipulate the guest's CPU state through interrupts and exceptions. However, the hypervisor must retain control of the execution of the virtual machine — it must still be able to run and pause a virtual machine arbitrarily as part of its duties as a hypervisor. We call this the context-switching problem.

Second, the hypervisor must still be able to emulate instructions on behalf of the guest. Notably, this conflicts with the first problem: the hypervisor needs to be able to view certain parts of the guest's CPU state in order to correctly emulate instructions. This is called the instruction emulation problem.

Our Solution

A security monitor can be used to reconcile these two issues. If the security monitor can be used to *intercept* all traps to the hypervisor and all attempts from the hypervisor to resume a guest VM, we can limit the hypervisor's view of the guest's CPU state. This approach is similar to that of Keystone's security monitor (see Figure 2-1b).

At a high level, whenever a trap is taken to the hypervisor, the security monitor should hide most CPU register values by zeroing them out. The hypervisor may need to legitimately access certain register values; the security monitor is responsible for determining the smallest set of registers that must be shared with the hypervisor. When guest execution is resumed, the security monitor should intercept and *restore* the original CPU state, excepting registers that the hypervisor should have modified.

To implement such a security monitor, we need an analog to the AMD GHCB specification [3]: a specification of what CPU registers need be to shared with the hypervisor during instruction traps as well as which CPU registers are allowed to be modified during these instruction traps. We call this specification the RISC-V Register Whitelist, and is discussed in Section 3.1. Then in Section 3.2, we discuss the design and implementation of the security monitor that protects a guest VM's CPU state according to the white-list specification.

Deviations from SEV-ES

Overall, the approach is similar to AMD SEV-ES, save for two small improvements. First, the decision of which registers to reveal to the hypervisor is left to the security monitor, not the guest supervisor as in AMD SEV-ES. Transferring the decision to the security monitor does not significantly change the size of the TCB. The benefit is that it eliminates extra several extra layers of exception handling (see Figure 2-2). Second, there is no shared memory region between the hypervisor and guest supervisor (or security monitor) that serves as an analog to the AMD GHCB. Whitelisted registers simply remain as-is when control is transferred from the security monitor to the hypervisor.

3.1 RISC-V Register Whitelist Specification

In this section, we outline exactly which CPU registers must be shared with the hypervisor during a context switch. This specification is a *whitelist*; by default, all CPU state should be hidden during a context-switch.

Throughout this section, the term "CPU state" refers to the normal registers used during computation, and not other pieces of CPU state such as control status-registers or performance counters. There is one special control-status register that we must pay attention to: the `HGATP` register. We discuss this at the end in Section 3.1.4.

3.1.1 Exceptions

To gather the necessary information, we look at three main sources. First, the RISC-V specification lists all possible causes for interrupts and exceptions [51]. We then looked at the source code of OpenSBI to determine which types of exceptions could conceivably be handled using M-mode code directly [39]. Finally, we looked at the Linux KVM to determine which registers needed to be shared with the hypervisor [26].

We start off by discussing exceptions. At a high level, an exception can be handled in three ways. In the simplest case, the exception can be handled by the security monitor directly and does not need to be redirected to the hypervisor at all. In the second case, the hypervisor handles the exception by redirecting the exception to the guest VM's exception handler. This case is also simple; the security monitor can simply redirect the exception to the guest VM instead of the hypervisor. The third case is the important case; in this case, the hypervisor actually does some relevant computation to handle the exception. This information is summarized in Table 3.1. Fortuitously, most exceptions can be handled by the SM itself or the guest OS. Only a few types of exceptions must be redirected to the hypervisor.

Exception Code	Description	Handler
0	Instruction address misaligned	Guest
1	Instruction access fault	Hypervisor
2	Illegal instruction	Security Monitor
3	Breakpoint	Guest
4	Load address misaligned	Security Monitor
5	Load access fault	Security Monitor
6	Store/AMO address misaligned	Security Monitor
7	Store/AMO access fault	Security Monitor
8	Environment call from U/VU-mode	Guest
9	Environment call from HS-mode	Security Monitor
10	Environment call from VS-mode	Hypervisor
11	Environment call from M-mode	Cannot be thrown by VM
12	Instruction page fault	Guest
13	Load page fault	Guest
14	Reserved	
15	Store/AMO page fault	Guest
16–19	Reserved	
20	Instruction guest-page fault	Hypervisor
21	Load guest-page fault	Hypervisor
22	Virtual instruction	Hypervisor
23	Store/AMO guest-page fault	Hypervisor

Table 3.1: Exceptions codes and their required handlers.

3.1.2 Hypervisor-Necessary Exceptions

There are six exception types that must be handled by the hypervisor: instruction access faults, environment calls, instruction guest-page faults, load and store guest-page faults, and virtual instructions faults. Each of these cases must be handled separately. We give a high-level overview of the different cases in table 3.2 before diving into each exception type in detail.

Exception Code	Description	Overview
1	Instruction access fault	Should currently never be thrown
10	Environment call from VS-mode	Hide based on ecall type
20	Instruction guest-page fault	Hide all registers
21	Load guest-page fault	Hide depending on MMIO read or not
22	Virtual instruction	Hide depending on instruction
23	Store guest-page fault	Hide depending on MMIO write or not

Table 3.2: High-level overview of which registers to hide.

Instruction access faults and instruction guest-page faults are the simplest cases to handle. For instruction guest-page faults, the hypervisor needs no CPU state; thus, all registers should be hidden. Currently, instruction access faults are not actually handled by the hypervisor, meaning that the VM will crash if this exception type is thrown. For now, all registers should be hidden for this exception type as well.

Virtual Instruction Faults

Certain instructions may trigger a virtual instruction fault, including instructions that manipulate CSRs. Currently, the only other type of instruction that triggers an virtual instruction fault is the wait-for-interrupt (WFI) instruction. These are documented in Table 3.3.

Hypervisor Environment Calls

Similar to how an application may make system calls to the supervisor, the guest virtual machine may also make environment calls to the hypervisor using the `ecall` instruction. The usage the `ecall` instruction is identical in both types of environment calls. Per convention, the `a7` register is used to specify the system call number or binary interface extension number, and the register `a6` is optionally used to provide a function id. The parameters to the system call are stored in registers `a0` – `a5`, and the return value is stored in `a0`.

By default, we require that the security monitor only reveals registers `a0` - `a7` and

Instruction	Registers Read	Registers Wrote
CSRRW	RS1	RD
CSRRS	RS1	RD
CSRRC	RS1	RD
CSRRWI	RS1	RD
CSRRSI	RS1	RD
CSRRCI	RS1	RD
WFI	None	None

Table 3.3: All instructions emulated by hypervisor and registers needed for emulated. Note that RS1 and RD are specific bits in the instructions encoding. For example, in the instruction **add** a0, a1, a2, RD is a0 and RS1 is a1.

only allows register a0 to be written to. As an extension, the security monitor can use knowledge of the different binary interfaces to place further restrictions on the whitelisted registers and return values. For example, if the security monitor knows that a specific type of environment call does not give a return value, it can restrict the modification of the a0 return register. We do not include these details as part of the specification because different systems may have different binary interfaces, and these interfaces are liable to change over time.

Load and Store Guest-Page Faults

The last type of exception are load and store guest-page faults. These types of exceptions can be further broken down into two types: ordinary page faults, and MMIO instructions.

RISC-V handles MMIO (memory-mapped I/O) by reading and writing from special virtual addresses that are not mapped to physical memory. Attempts to read and write from these addresses trigger page faults, allowing the hypervisor to emulate MMIO instructions.

For an ordinary page fault, the hypervisor does not need to access *any* CPU registers. For MMIO store instructions, the hypervisor must access one register corresponding to the stored value. For MMIO load instructions, the hypervisor writes

to one register corresponding to the loaded value.

An overview on the whitelist for page faults is shown in Table 3.4. There are two complications that must be dealt with. First, there is no simple way for the security monitor to distinguish between MMIO loads and stores and regular load and store instructions. Second, specifically for MMIO instructions, the hypervisor uses the `HGATP` control-status register to access the faulting instruction. As we discuss in section 3.1.4, the hypervisor is no longer allowed to access the `HGATP` register. Therefore, we must provide an alternative way for the hypervisor to look-up the faulting instruction.

The issue is resolved by the security monitor, which should look up the faulting instruction instead and stores this instruction into the `a1` register. The hypervisor no longer needs to rely on the `HGATP` register, instead using the value provided by the security monitor.

Page Fault Type	Values Read	Values Wrote
Normal Load	None	None
Normal Store	None	None
MMIO Load	INSN (`a1`)	RD (RD)
MMIO Store	INSN (`a1`), `RS2` (`a0`)	None

Table 3.4: All page-fault cases and data needed to emulate instruction. Note that the faulting instruction itself must be stored into the `a1` register. Also note that for MMIO stores, reading the register encoded by `RS2` consists of several sub-cases that we do not describe for simplicity. For simplicity, rather than revealing the actual encoded register, the security monitor should store the relevant value in the `a0` register (*e.g.*, if `RS2` encoded the `a7` register, the value of `a7` should be stored in `a0` register and not the `a7` itself).

3.1.3 Interrupts

Interrupts are a much simpler case than exceptions. Currently, the hypervisor never needs to access guest CPU state during an interrupt. Thus, for interrupts, all registers should be hidden in all cases.

3.1.4 Hypervisor-mode Address Translation and Protection (HGATP) Register

The HGATP register is a special control-status register used by hypervisors and guest VMs for page-table management. Specifically, it stores the root for the hypervisor's second-stage page table for the guest VM, the guest's VM ID, and additional page-table metadata (*e.g.* addressing mode and protection bits). At a high-level, the hypervisor uses `HGATP` to keep track of which VM is running and to perform top-level page table management for the guest VMs.

The hypervisor can use special *hypervisor-load* and *hypervisor-store* instructions to arbitrarily read and write from guest VM memory. If allowed to run, these instructions would trivially circumvent the protection of the security monitor.

The hypervisor-load and hypervisor-store instructions depend on the value stored inside the `HGATP` register to execute. Therefore, in every trap, all bits of the `HGATP` register except for the VM-ID bits should be hidden (zeroed out), preventing the hypervisor from correctly executing hypervisor-load and hypervisor-store instructions.

It is important to note that hiding bits of the `HGATP` register from the hypervisor still results in a functional design. The *memory-protection*[1] component carefully controls the hypervisor's ability to manipulate the page table. As such, the hypervisor no longer needs to use the `HGATP` register while the VM is running, save for except for the VM-ID bits. The only instance we found where the hypervisor used the hypervisor-load and hypervisor-store instructions were when it needed to handle MMIO stores and loads. This case is already addressed; thus, hiding the `HGATP` register in the way described results in a functional design.

3.2 Secure Context-Switching

In this section, we discuss the implementation details of the security monitor and how it achieves the intended white-list behaviour. For a starting point, we forked

[1]We give a reminder that the memory protection component is covered in other, simultaneous work.

OpenSBI [39] and added three new SBIs:

- `sm_resume_cpu`: the primary interface, used to resume a virtual cpu
- `sm_create_cpu`: the interface used to initialize a virtual cpu
- `sm_prepare_mmio`: a special interface used to deal with the complexity of MMIO instructions

Figure 3-1 shows the overall execution flow of the system.

3.2.1 Creating and Resuming Virtual CPUs

The starting point for the system is `sm_create_cpu`; the hypervisor must be modified to call this interface before attempting to run the virtual CPU for the first time. This interface takes the initial virtual CPU state and stores it securely within the security monitor's memory. In a full VM-based TEE, this interface would also be combined with a secure attestation component. However, the main purpose is to just store the CPU state, allowing future calls to `sm_resume_cpu` to function.

The main interface in the system is `sm_resume_cpu`; the hypervisor must be modified to use this interface instead of the normal procedure to resume a VM. It takes as input the virtual CPU index to resume and the new CPU register state. The security monitor must *merge* this state with the CPU state prior to the trap (see Figure 3-2) as per the white-list specification. This merging process is necessary for two reasons: (1) it prevents a malicious hypervisor from unnecessarily modifying registers during traps in an attempt to glean information and (2) as per the white-list specification, most registers in the hypervisor's view will be 0; merging simply brings these registers back to original state from before the trap.

The security monitor must also ensure that it handles any traps before the hypervisor. By zeroing out the `MIDELEG` (machine interrupt delegation) and `MEDELEG` (machine exception delegation) registers, the security monitor guarantees that the next trap to supervisor mode would instead trap to machine mode (the security monitor) instead.

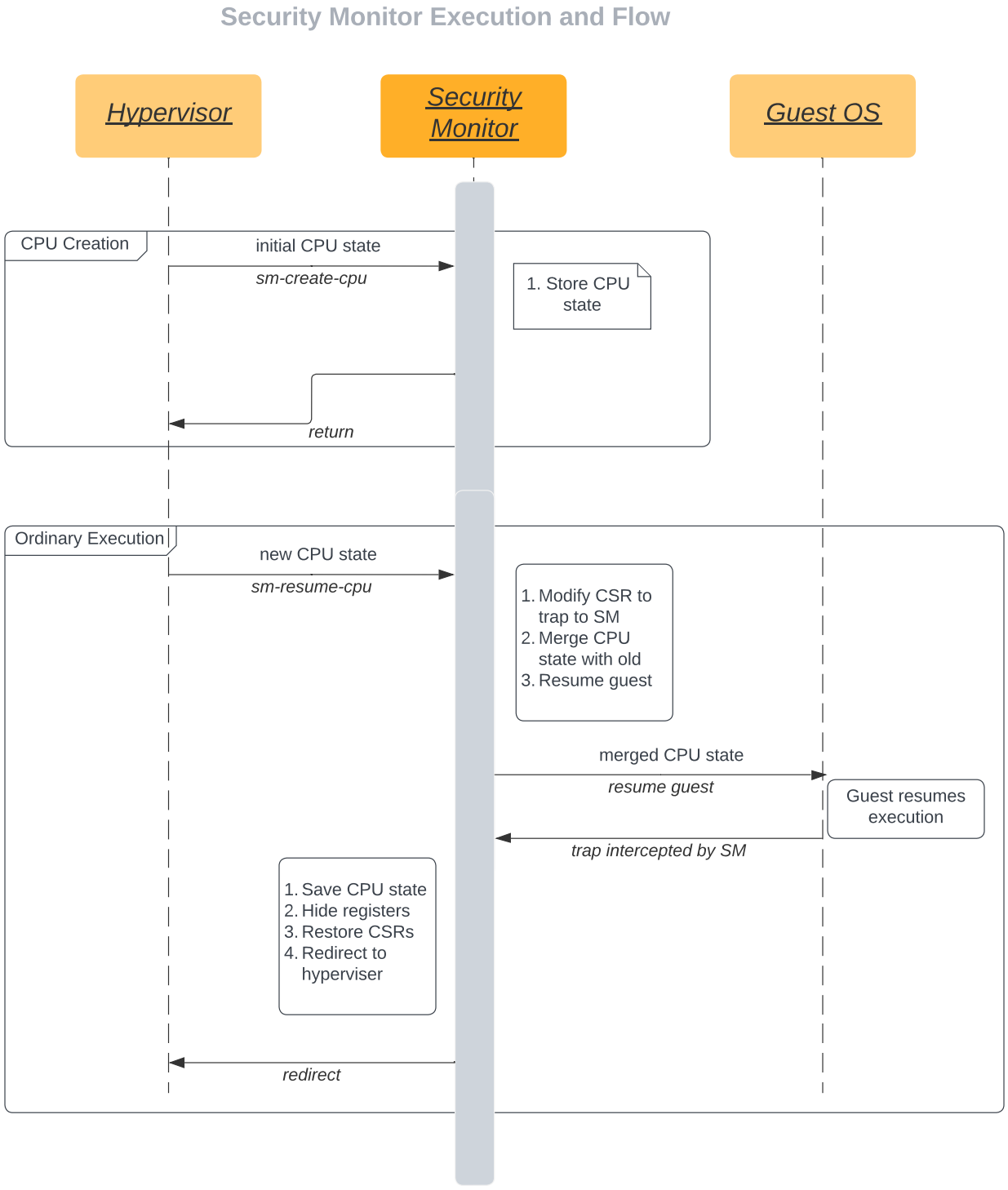


Figure 3-1: Overall execution flow.

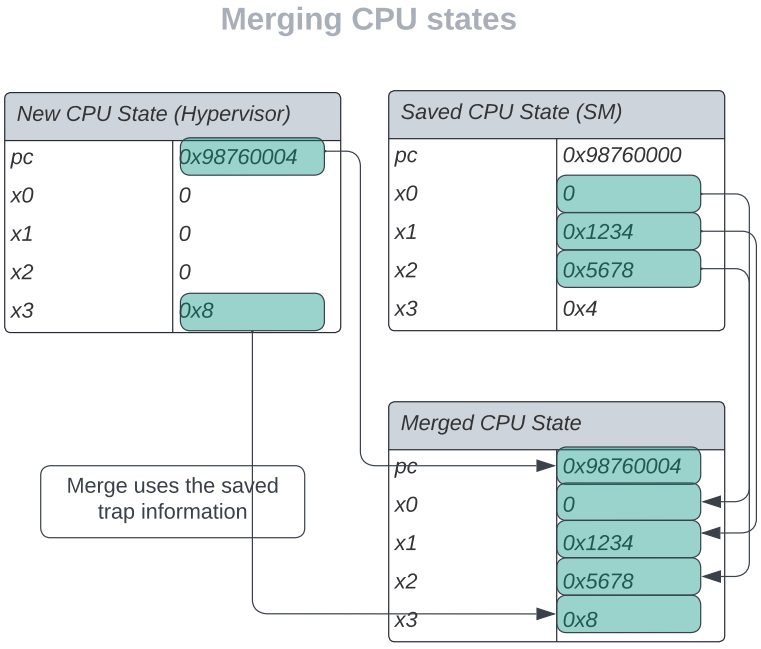


Figure 3-2: Merging CPU states.

Finally, the security monitor must restore the `HGATP` register. Once this is done, the security monitor resumes the guest vCPU.

Eventually, a trap is triggered and intercepted by the security monitor. The security monitor must first save a copy of CPU state and trap details in its private memory; this information is necessary for merging CPU states after `sm_resume_cpu`. After this, registers and `HGATP` are zeroed out according to the whitelist, and `MIDELEG` and `MEDELEG` CSRs are restored to their original state. Finally, the security monitor redirects the trap to the hypervisor[2].

Inputs to `sm_create_cpu` and `sm_resume_cpu`

Both `sm_create_cpu` and `sm_resume_cpu` take as input the entire vCPU state and a virtual CPU index. Implementing this behaviour for an environment call is tricky as traditional environment calls conventionally take only a few registers as input.

[2]In cases where the guest OS can handle the trap, most of these steps are skipped and the security monitor simply redirects to the guest OS trap handler

Theoretically, the input for `sm_resume_cpu` could be modified to instead be a list of registers that were modified and their new values, but this approach would require additional changes to the hypervisor. Additionally, determining which registers were modified during an exception and converting the information to a shorter format could introduce significant overhead during the context switch.

Fortuitously, environment calls only take a few registers as input by convention only. Thus, to create a `sm_create_cpu` or `sm_resume_cpu` environment call, the hypervisor can load guest VM register values as if it were resuming the guest VM normally (*i.e.*, the hypervisor loads the guest VM's `a5` register into the `a5` register). However, to actually trigger the correct environment call, the `a7` register must hold the environment call id and another register (we use `a1`) must hold the virtual CPU index. To resolve this issue, we reuse two special control-status registers, `STVAL` and `SCAUSE`, to hold the guest VM's `a1` and `a7` registers respectively. These control-status registers are only used to hold special trap handling information, and their values are irrelevant when the hypervisor needs to resume a guest VM. It is thus possible to transmit all inputs necessary for these two environment calls.

The use of the `STVAL` and `SCAUSE` may seem ad-hoc, but we believe it is currently the best solution. Other approaches could be considered, but they add complexity and overhead. A simple solution could be to use two environment calls instead of a single call, with each call transmitting half the CPU state; this solution obviously creates much extra overhead when resuming a guest VM. The conventional method would be to give the security monitor a pointer to the hypervisor's view of the guest CPU state instead. While the security monitor is capable of reading from the hypervisor's memory, doing so takes significantly longer than reading from the security monitor's own memory and again creates extra overhead. While crude, the approach we use may be the most effective. An additional benefit is that the procedure for creating the two environment calls is nearly identical to the original procedure for resuming a guest VM.

3.2.2 Handling MMIO Instructions

The third SBI interface, `sm_prepare_mmio`, is used to deal with a special edge case. Currently, both guest VM page faults and guest MMIO page faults trigger the same exception codes, and there is no reasonable method for the security monitor to distinguish between the two cases. However, according to the white-list specification, MMIO page faults require certain registers to be revealed, while normal page faults require no register information. The security monitor could simply treat all page faults as MMIO page faults, but this approach reveals unnecessary information.

`sm_prepare_mmio` is used to resolve this issue, as shown in Figure 3-3.

By default, the security monitor treats all guest page faults as normal (non-MMIO) page faults, hiding all registers. If the hypervisor determines that the page fault is an MMIO page fault and that it needs extra information from the security monitor, the hypervisor uses `sm_prepare_mmio` to inform the security monitor that the *next exception* for that vCPU will be an MMIO page fault. The security monitor verifies that the most recent trap was, in fact, a guest page fault before marking the next exception down as an MMIO page-fault. Afterwards, the hypervisor resumes the guest VM *at the same instruction.*

This approach causes the same page fault exception to trigger again. The security monitor, using the information from the previous SBI call, realizes that this page fault is a guest page fault and includes the registers necessary for an MMIO page fault. The hypervisor can then handle the page fault, and execution resumes as normal.

Note that we cannot simply create an SBI for the hypervisor to request the register information it is missing. Such an interface could easily be misused to fetch information during traps that are not MMIO page faults. Moreover, by convention, environment calls only return values through the `a0` register and by Table 3.4, the hypervisor needs two pieces of information to correctly handle the page-fault.

We note that, although this approach is quite inefficient, it is sufficient for an initial implementation.

3.2.3 Security Discussion

At first glance, the system seems to rely on the hypervisor willingly using the provided SBI interfaces to create and run the vCPUs. In practice, secure attestation and memory protection would prevent the hypervisor from circumventing the use of these interfaces. During the secure attestation phase for VM startup, the VM would simply query the security monitor if the hypervisor had called `sm_create_cpu` and `sm_resume_cpu` at least once. If the hypervisor had not, the attestation would fail and the VM would refuse to continue.

Once the regular cycle of calling `sm_resume_cpu` begins, the hypervisor is forced to continue using the `sm_resume_cpu` interface to resume the vCPU. As the hypervisor's view of the vCPU state is hidden, it cannot resume the vCPU with the proper registers. The hypervisor could try to resume the vCPU anyways with arbitrary data for the CPU registers. Even then, the memory protection component would prevent the CPU from reading from the VM's memory as the VM was launched without use of the security monitor. Very quickly, an exception would be triggered without the hypervisor learning anything.

There is one small weakness relating to MMIO page faults. Despite the careful design of the `sm_next_mmio` interface, there is currently no way for the security monitor to verify if the instruction is indeed an MMIO instruction. Consequently, it is possible for the hypervisor to lie and attempt to glean trace amounts of information during guest page faults.

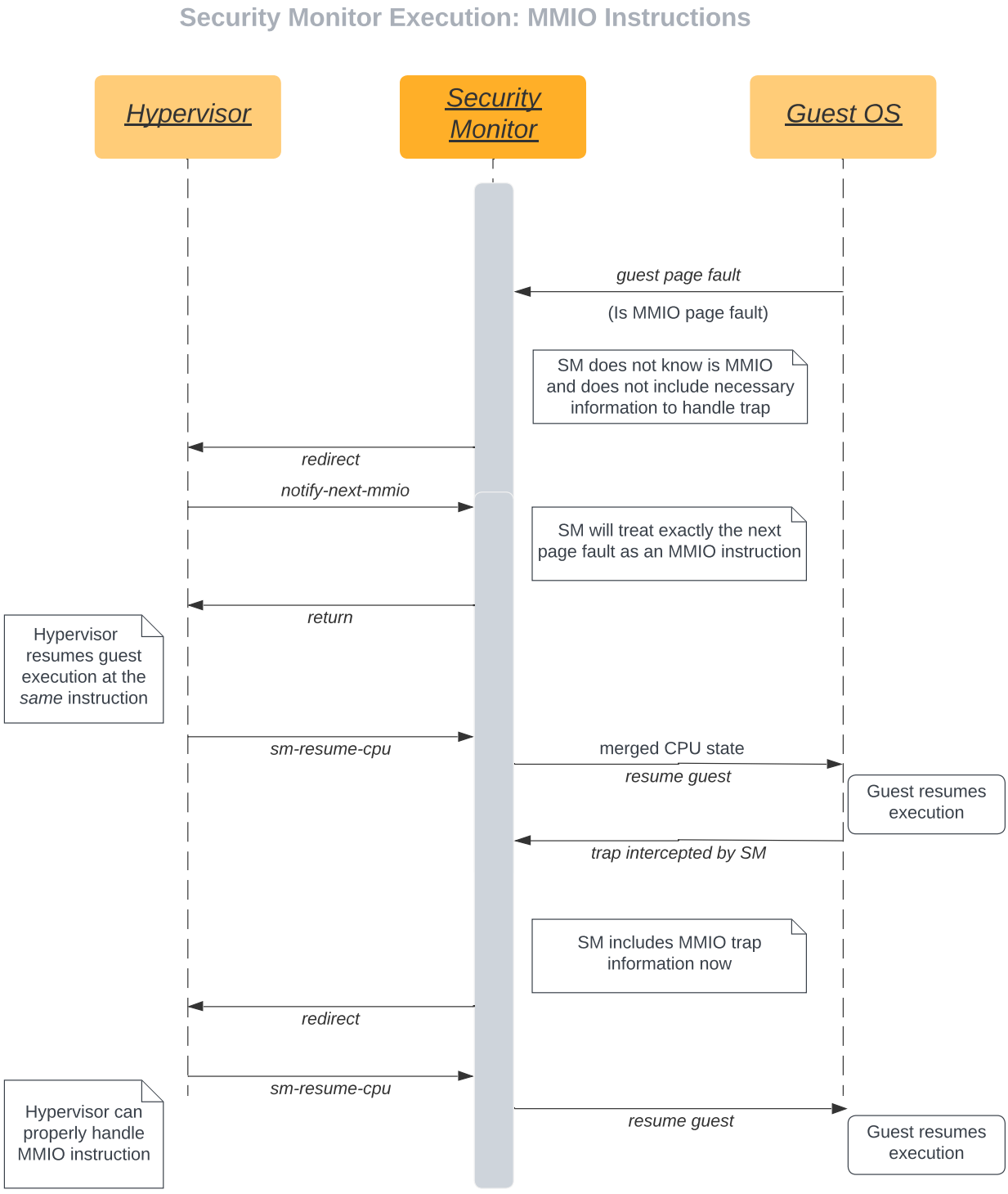


Figure 3-3: Special case for MMIO instructions.

Chapter 4

I/O Performance of VM-Based TEEs

In this chapter, we discuss the network I/O performance of VM-based TEEs. Our primary goal is to address the following question:

> *Is it possible to achieve the same level of I/O performance without trusted I/O devices and sacrificing the existing security guarantees?*

We focus on the network I/O performance of AMD SEV-SNP because it is the only available VM-based TEE as of the time of writing.

We start off in Section 4.1 with a comprehensive analysis on the sources of I/O overhead in a VM-based TEE. The sources of overhead can generally be broken into two different categories: VM-related overhead, and TEE-related overhead. The purpose of TIO is to reduce TEE related I/O, but our initial findings indicate that bounce buffer-related overhead — the main source of TEE-related overhead — only creates approximately $1-2\%$ additional overhead. This initial finding is encouraging, as small TEE-related overhead indicates that it should be feasible to approach the performance of a TIO solution even without using TIO.

Our analysis finds that using DPDK within a VM minimizes VM-related overhead, making TEE-related overhead the most impactful. If an SNP VM could use DPDK and meet the performance of an ordinary VM using DPDK, TIO devices would not be necessary. This implication would also apply to lower-performing configurations such as VirtIO or plain SRIOV because these configurations have more VM-related

overhead, decreasing the impact of TEE-related overhead[1]. Unfortunately, SNP VMs cannot use DPDK out of the box because of memory encryption protections.

In Section 4.2, we design Folio (Fast Opaque user-Level I/O), a solution that allows use of DPDK within an AMD SEV-SNP VM. The primary goal of Folio is to approach or match the performance of a (future) TIO network solutions without sacrificing any security or changing any security assumptions. Folio meticulously manages two interfaces: the one between the I/O device and the confidential VM, and the one between the DPDK library and the DPDK application. This approach allows existing DPDK applications to run without modifications while continuing to guarantee the security of the application and the VM itself. We conduct an extensive evaluation comparing the performance of Folio to a non-TEE VM (as a substitute for a VM using TIO) in Section 4.3. Our results indicate that the performance gap between Folio and a future TIO VM is less than 6% in terms of throughput and latency. We conclude with a comparison of Folio to TIO solutions in Section 4.4 and a brief discussion of related works in Section 4.5.

4.1 Network Bottlenecks in Confidential VMs

As a first step, we conduct an in-depth experiment to analyze the importance of various sources of I/O overhead.

4.1.1 Methodology

There is a limited set of factors that can contribute to a VM's network performance, and the presence of these factors can be controlled by the configuration of the VM. Table 4.1 shows various VM configuration and which factors contribute to network overhead. To estimate the impact of a factor, we simply need to compare the performance of two VM configurations differing only by that factor. To compare performance, we measure the latency (both average round-trip latency and tail latency)

[1]Albeit, with some caveats for VirtIO as that method requires frequent `VMEXIT`s, which creates additional sources of TEE-related overhead.

between a simple UDP echo server running on a VM and a desktop client on the local network.

We note that this comparison-based method diverges from more established benchmarking methods such as simply measuring the percentage of time spent executing each type of overhead. For our use case, the comparison-based method actually has several key benefits.

(1) The comparison-based method can estimate the impact of factors that are impossible to measure traditionally. For instance, consider the SNP ownership check, in which SNP hardware verifies ownership of a piece of memory against the RMP table. The ownership check is performed for *every* page table walk, making its performance impact exceedingly difficult to measure directly.

(2) The comparison-based method reduces inaccurate results caused by the frequent reading on timestamps within the VM. To read a timestamp within a VM, an instruction like `RDTSCP` would typically need to be executed. This instruction forces a `VMEXIT` event, triggering the hypervisor and making it emulate the instruction. Even within a normal VM, frequently reading timestamps in such a manner would slow down any application being benchmarked and incorrectly amplify the impact of any factors being measured.

In the context of an SEV-SNP (or even SEV-ES) VM, the amplification is even more inaccurate because of the `VC` handlers. Recall Figure 2-2; a `VC` handler must be invoked both during the original `VMEXIT` event and after the hypervisor finishes emulating the timestamp instruction. These overheads would lead to even greater inaccuracies for SEV VMs.

(3) The comparison-based method can still easily isolate the impact of a single factor. For example, as shown in Table 4.1, the presence of a bounce buffer is the only difference between the "non-TEE VM with SR-IOV" configuration and the "SEV VM with SR-IOV" configuration. By comparing the performance of these factors, we can still estimate the performance impact of the bounce buffer.

SEV's Impacts on Non-TEE VMs.

Before diving into our experimental setup and the results, we must note that certain SEV features also impact the performance of non-TEE VMs. While these impacts are small and do not significantly impact our results or understanding, it is important to discuss them now.

First, Secure Memory Encryption (SME) [23] is enabled by default when using SEV. Thus, the memory pages of the hypervisor and all non-TEE VMs are encrypted automatically by hardware, and our results do not account for the raw cost of hardware encryption. This cost is small and does not significantly impact the results of our findings.

Second, when SNP is enabled, the hardware must also perform an RMP ownership check for all hypervisor writes. Non-TEE VMs experience some small ownership check overhead as a result, but this overhead has been shown to be minimal [30].

4.1.2 Experimental Setup

Our experimental setup consists of a SNP-supported workstation and a desktop client. The SNP-supported workstation has an AMD EPYC 7313 16-Core Processor, 64GB DRAM, 1TB disk, an Intel I350 Gigabit NIC for internet connection, and an Intel 82599ES-based 10Gb NIC (Silicom PE210G2SPI9) for supporting DPDK and SR-IOV. The desktop client has an Intel i5-12400F Processor, 32GB DRAM, 500GB disk, an Intel I219-V NIC for sharing internet with the workstation, and another Intel 82599ES-based 10Gb NIC for supporting DPDK and SR-IOV. The two Intel 82599ES NICs are connected by a MokerLink 8 Port 10Gbps switch. The original host kernel (`sev-snp-iommu-avic_5.19-rc6_v4` branch), QEMU (`snp-v3` branch), and OVMF (`master` branch) were directly obtained from the `sev-snp-devel` repository [6] (Commit: `fbd1d07628f8a2f0e29e9a1d09b1ac6fdcf69475`). The desktop client simply runs an unmodified Ubuntu 22.04.1 LTS with a kernel of `5.19.0-38-generic`.

The VMs were configured as 4-virtual CPUs (vCPUs), 8-GB memory, 30GB disk storage. The guest kernel used to support the SNP feature is forked and built from

the same repository and commit version as the host kernel. For "non-TEE VM" and "SEV VM" setup, the VMs use the default virtio-net-pci device suggested by AMD's official script used for launching SEV VM [6]. For all SR-IOV and DPDK setups, one NIC's virtual function (VF) is directly assigned to the VM via QEMU PCI pass-through configuration. The network server applications running inside the VMs are two simple UDP echo servers. One of them uses normal socket APIs, and the other is only used in "VM with DPDK" setup and is configured using DPDK APIs. On the client side, we reuse a client application from an existing open-sourced project that focuses on measuring tail latency [21]. We configure the packet rate for the client application to be 5000 packets per second (pps).

4.1.3 Results

Table 4.1 shows the performance of various VM configurations, which we use to make several observations.

Observation-1: VM-related overhead is the dominant source of overhead impacting performance.

Table 4.1 shows that, when focusing in non-TEE VMs, enabling SR-IOV reduces routing latency by more than 50 times. This result indicates that the cost of emulating an I/O device on the host side is the dominant factor in performance. Furthermore, when comparing 'non-TEE VM with SR-IOV" to "non-TEE VM with DPDK," we see that DPDK usage results in roughly 3 times lower latency, indicating that the kernel network stack is a critical factor in performance.

Observation-2: Overhead due to ownership checks and TLB checks in SNP is smaller than that of `VMEXIT` measurement checks in ES. The VC handler might be the key factor that impact performances for SEV-ES and SEV-SNP VMs.

	Overhead Description	non-TEE VM			Existing SEV Configurations				Proposed Configurations		
		non-TEE VM	VM with SRIOV	VM with DPDK	SEV VM	SEV with SRIOV	ES VM with SRIOV	SNP VM with SRIOV	SNP VM with TIO	SNP VM ⋆ (TIO/ DPDK)	SNP VM (Folio)
Common	Routing within the VM	Y	Y	N	Y	Y	Y	Y	N	N	N
	Routing within the host	Y	N	N	Y	N	N	N	N	N	N
	Emulated I/O interrupt	Y	Y	N	Y	Y	Y	Y	Y	N	N
	Other factors	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
SEV-only	Encrypted memory overhead	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
	Register encryption	Y	Y	N+	Y	Y	Y	Y	Y	N+	N+
	Bounce buffer allocation	N	N	N	Y	Y	Y	Y	N	N	N
	Bounce buffer copy	N	N	N	Y	Y	Y	Y	N	N	Y
	VMEXIT check overhead	N	N	N	N	N	Y	N	N	N	N
	VC handler overhead	N	N	N	N	N	Y	Y	Y	N*	N*
	Ownership check overhead	N−	N−	N−	N−	N−	N−	Y	Y	Y	Y
	TLB check overhead	N	N	N	N	N	N	Y	Y	N+	N+
	Standardized Latency (5000 pps)										
	Mean latency	>50x	1.00 (75.1 µs)	0.37x	>50x	1.01x	1.17x	1.16x	N/A	N/A	N/A
	Median latency	>50x	1.00 (75.3 µs)	0.36x	>50x	1.02x	1.18x	1.13x	N/A	N/A	N/A
	p95 tail latency	>50x	1.00 (83.8 µs)	0.33x	>50x	1.00x	1.20x	1.18x	N/A	N/A	N/A
	p99 tail latency	>50x	1.00 (123.6 µs)	0.31x	>50x	0.78x	0.94x	2.19x	N/A	N/A	N/A

Table 4.1: Factors affecting network performance under different VM configurations and the standardized round trip latency. With DPDK means the VM is using DPDK to operate a SR-IOV device. With TIO means the VM is using trusted I/O devices with SR-IOV. ⋆ means the future idealized solution. Other factors refer to the impact of physical devices and other software settings on performance, which are all configured the same. **Y** implies that there is this overhead and **N** implies not. - indicates that the ownership check only happens for write access. * indicates that VC handler overheads due to the scheduler or interrupts are minimized. + indicates that the number of `VMEXIT` and thus the overhead is minimized due to the polling mode. For standardized latency, the performance of "non-TEE VM with SR-IOV" is chosen as the standard with their value represented in parentheses. x represents the ratio of latency under different configurations compared to the standard. N/A means this configuration is not supported yet.

The second group is SEV series VMs. When comparing "ES with SR-IOV" and "SNP with SR-IOV", we found that SNP's performance is better than ES despite SNP introducing ownership checks and TLB checks[2]. This improvement could be attributed to the significant performance impact caused by the `VMEXIT` check in ES. At the same time, it also implies that the overhead of ownership and TLB checks is not as substantial. When comparing "ES/SNP VMs with SR-IOV" and "SEV VM with SR-IOV", we noticed that their performance differs by more than 10%. This finding suggests that the VC handler might be the primary reason for the performance decline in the ES and SNP configurations.

Observation-3: The impact of the bounce buffer is actually quite small, much smaller than previously thought.

[2]Recall AMD SEV-ES and Protection Against TLB Poisoning

Comparing "non-TEE VM with SR-IOV" with "SEV with SR-IOV", we see that the performance difference is only $1 - 2\%$. The only difference between these two configurations is that "SEV with SR-IOV" has additional overhead from using bounce buffers, suggesting that the performance impact of the bounce buffer itself is small.

4.2 Folio

The observations in Section 4.1.3 indicate that even if bounce-buffer related overhead cannot be eradicated without use of TIO devices, such overhead is not actually a limiting factor in the network performance of confidential VMs. This observation lead us to create Folio, a software solution that enables the usage of DPDK and DPDK application within SNP VMs. Compared to the ideal future configuration ("SNP VM with TIO/DPDK"), Folio only requires a single packet buffer copy overhead.

Challenges: Designing Folio in SNP poses several challenges in terms of *security*, *efficiency*, and *functionality*:

- *Security:* Folio must ensure security in terms of maintaining a strict boundary between shared memory and private memory and preventing the VM from leaking any secrets to the I/O device. The task is complicated by the fact that DPDK and its applications run in userspace and cannot easily access the features of the kernel.

- *Efficiency:* Folio must maintain DPDK's performance, even when it has to deal with the extra complexity of shared and private memory regions.

- *Functionality:* TIO might offer additional cryptography offload capabilities that cannot be replicated in software alone. As only the SoC is trusted, it is impossible to offload any sensitive cryptographic operations without compromising security. Folio finds a way to implement similar functionality at the cost of additional CPU resources.

In the remainder of this section, we discuss our threat model and design goals (Section 4.2.1) and a brief design overview (Section 4.2.2) before jumping into the specific design details (Section 4.2.3).

4.2.1 Threat Model and Design Goals

FOLIO uses the same threat model as traditional VM-based TEEs [23, 19]. Specifically, the TCB consists only of the SoC and the software running within the SNP VM. We assume that any potential adversary has complete control over the rest of the server and any I/O devices. To communicate with I/O devices, SNP VMs need to use a shared memory region, and we assume that any data within this shared region can be accessed or modified at will. Finally, we do not consider denial of service attacks just like VM-based TEEs.

The goals of FOLIO are summarized as follows:

G1. End-to-end Security: FOLIO must achieve the same level of security as existing confidential VMs using VirtIO. We assume that applications in these confidential VMs use some software-based encryption such as TLS/SSL or IPsec to achieve end-to-end-security. Note that we cannot compare the security FOLIO to TIO, which we discuss in Section 4.4.1.

G2. Comparable Network Performance: FOLIO should achieve high networking performance that is comparable to the future performance of TIO solutions.

G3. On-core acceleration of offloading tasks: We suspect that confidential VMs may be able to offload sensitive network tasks, such as packet encryption, to TIO devices when they are made available[3]. FOLIO should find a way to make offloading possible or find an alternative way to enhance performance.

G4. Code Compatibility: FOLIO should be mostly compatible with existing DPDK applications; these applications should be able to run within a confidential VM with little to no changes. Obviously, the primary benefit of this goal is to let existing DPDK applications to easily run within SNP VMs. An additional benefit is that

[3]Of course, packet encryption and decryption can be offloaded to ordinary I/O devices as well, but I/O devices cannot be trusted with this task in the context of confidential VMs

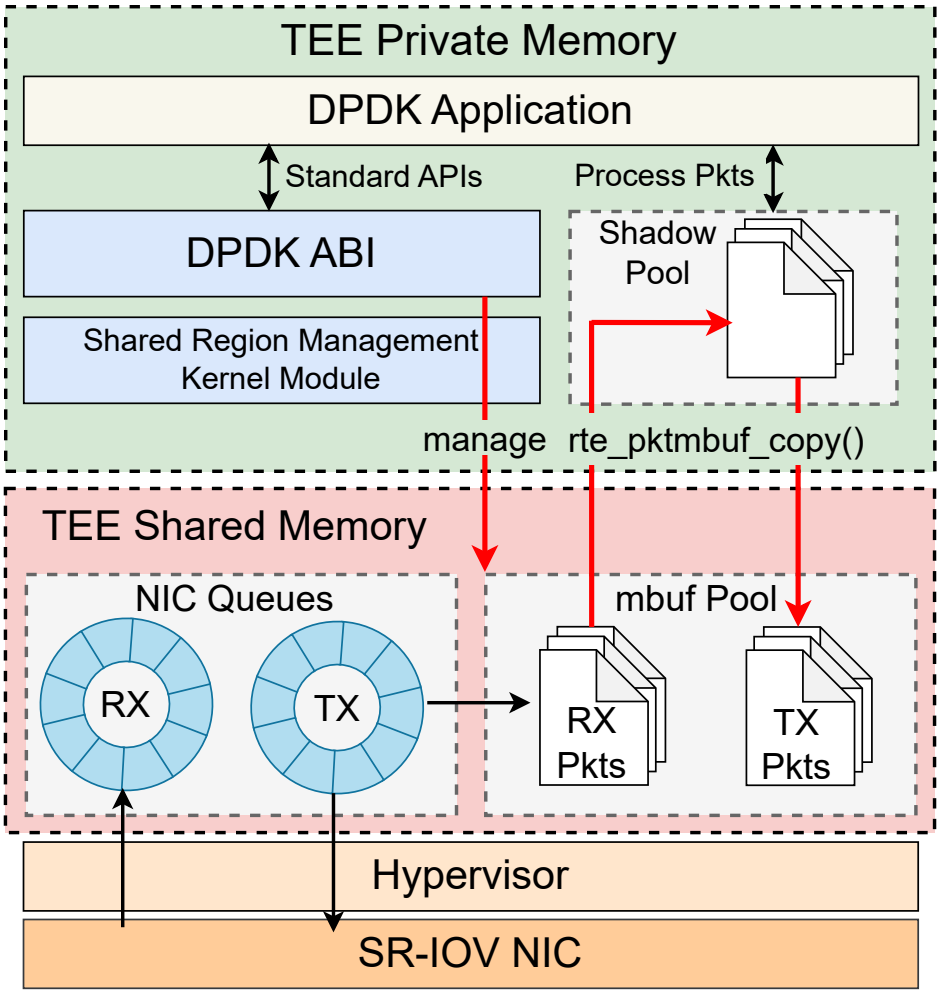


Figure 4-1: FOLIO overview. All components within the TEE private memory are considered trusted. Meanwhile, data within the shared memory, the hypervisor, and the NIC are untrusted. The DPDK library and shared region management module strictly control the interaction with shared memory.

developers can develop and run their applications with FOLIO and effortlessly switch to TIO devices once they are available.

4.2.2 FOLIO Design Overview

Of the design goals we outlined, the hardest goal to satisfy is **G1.**. For DPDK to function within an SNP VM, it must be modified to establish a shared memory region to communicate with the I/O device. There are currently no tools for creating shared regions in userspace within SNP VMs. Even if the challenge of allocating a shared

memory region is solved, the system must take care to handle this memory very carefully; no true computations should be done using this shared memory.

To achieve **G1. End-to-end Security**, FOLIO breaks this considerable task into two essential aspects that ensure security: *Constrained VM-I/O interaction interfaces* and *Constrained DPDK-App interaction interfaces*. These aspects ensure security between the VM and hypervisor (and I/O devices), and the security between the DPDK library and DPDK application.

As shown in Figure 4-1, the interaction between VMs and I/O devices is managed by a combination of a Shared Region Management kernel module and a modified DPDK library. This setup rigorously manages the shared memory region, ensuring that only necessary data is exposed and tightly controlling all network packets and metadata that could be exposed to the untrusted environment.

Regarding the interaction between the DPDK library and the DPDK applications, we employ a *shadow network buffer pool* design. This design ensures that only data structures within the private memory region can be accessed by DPDK applications. This protection mechanism prevents sensitive secrets or information from being accidentally exposed due to misutilizations of the shared region. Additionally, FOLIO also takes care of secure memory recycling and fault handling.

To achieve **G2. Comparable Network Performance**, FOLIO tries to minimize overhead due to different factors to the greatest extent. This includes using a modified version of DPDK to mitigate most factors associated with SEV-specific protection and using preallocated memory segments to mitigate bounce buffer allocation overhead. These measures enable FOLIO to achieve near non-TEE-protection's performance in the evaluation section.

For **G3. On-core acceleration of offloading tasks**, FOLIO focuses on ensuring end-to-end secure network communication with a specific emphasis on crypto-operation offload support. FOLIO leverages AES-NI instructions to accelerate crypto operations, supporting both look-aside offload mode and a CPU-enabled emulated inline mode[4]). To evaluate the performance of network offloading, we also devel-

[4]See Hardware-accelerated End-to-end Encryption

oped an IPsec performance testing tool to enable a comprehensive evaluation of the performance of CPU-based network offloading.

To handle **G4. Code Compatibility**, FOLIO follows the original DPDK ABI, allowing the existing DPDK application to be compiled and executed directly within SEV-SNP VMs.

4.2.3 FOLIO Design Details

FOLIO is a modification of DPDK that allows secure and efficient network I/O from within an SNP VM. This section outlines the design details of FOLIO.

Constrained VM-I/O Interaction Interfaces

To properly interact with the I/O device, FOLIO needs to use shared memory. To properly restrict and control this shared memory, FOLIO uses a *shared region management* module to allocate and manage regions of shared memory and follows a *limited exposed metadata* principle to ensure only necessary and secure data are exposed.

Shared region management FOLIO introduces a shared region management kernel module to strictly control the contents that need to be placed in shared memory. Specifically, any DPDK-related content that needs to be placed in shared memory must explicitly notify this kernel module during DPDK initialization. This module then interacts with the hypervisor to synchronize the corresponding physical addresses of shared regions for later I/O purposes. When shutting down the DPDK application, the kernel module zeros out all contents placed in these shared regions, recycles memory pages, and resets them to private memory.

Limited exposed metadata FOLIO explicitly and manually examines all data structures exposed to the host. It turns out that only two types of data region need to be shared. The first is the receive/transmit (RX/TX) descriptor rings, which are also used in the default VirtIO to facilitate the communication between VM and I/O devices. These rings contain metadata information about network packets, such as

the memory location, length, or status of network packets. The second type of region is the DPDK-specific RX/TX memory buffers, which hold the actual incoming and outgoing network packets and were originally designed to be directly accessible by DPDK applications. In the manual examination procedure, we conduct a thorough comparison between the data exposed by the default VirtIO and FOLIO to ensure that there is no inadvertent information leakage. The most exposed data mainly comprises meta-data associated with packets, which is handled either in an on-core write-only manner or protected by software-based encryption. Additionally, we pay extra attention to data structures resembling pointers. For pointers that do not necessarily need to be exposed, such as some DPDK-specific pointer design, we keep them in private memory. For pointers that must be exposed, like addresses in the descriptor rings, we check whether the pointed-to addresses are within the shared region when performing operations.

Constrained DPDK-APP Interaction Interfaces

To ensure the security of DPDK applications, FOLIO follows a copy-before-processing principle and proposes a shadow packet buffer pool design to ensure that applications only operate on private memory. Additionally, FOLIO also conceals shared memory from DPDK applications by hiding vulnerable information (*e.g.*, address of the shared region or exposed data) inside the Environment Abstraction Layer (EAL), and includes special fault handling mechanisms to provide additional protection.

Copy before processing. FOLIO strictly adheres to the principle of handling network packets as outlined in VirtIO, *i.e.*, copying the network packet and the associated network buffer data structures from the shared region to the private region before any software begins processing it, and vice versa. Once the network packets are placed in private memory, SEV-SNP's memory protection mechanism explicitly ensures the confidentiality and integrity of the network data packets and prevents the device or hypervisor from modifying the packet content or associated data structures of the network packets while an application is using them. Note that we want to

emphasize that even encrypted network packets should be copied to the private region before network applications begin processing them. Some existing applications may attempt to decrypt the packets directly in their original memory addresses or read unencrypted information, such as header information, during packet processing. These actions can enable the untrusted hypervisor to manipulate the VM's intended control flow or provide incorrect data.

Shadow packet buffer pool FOLIO introduces a shadow packet buffer pool design to provide a secure interface for DPDK applications to process network data packets. Specifically, packet buffer pools are the memory pools used by DPDK applications to handle network packets. These pools are created during DPDK initialization, and each pool consists of a fixed number of data structure objects known as `rte_mbuf`. These `rte_mbuf` objects are responsible for storing network packets and are accompanied by a set of preceding metadata. The metadata includes information such as the message type, length, a region to store some application-specific private data, a pointer to the address of the raw network packet and, if necessary, a pointer to the next `rte_mbuf` object in case a single object is insufficient to hold an entire network packet.

The shadow packet buffer pool design enables the guest VM and DPDK applications to retain critical metadata internally while maintaining the ability to efficiently communicate with I/O devices. As shown in Figure 4-2, FOLIO creates three packet buffer pools during DPDK initialization, which we call the *shared* pool, the *shadow* pool, and the *temporary* pool. Among these three packet buffer pools, the *shared pool* is the only memory pool allocated in shared memory. It serves as the memory pool used by the untrusted NIC for reading TX/RX packets and is strictly restricted from direct use by the SNP VM. The *temporary* memory pool is used by the device driver inside the SNP VM to communicate with the device, where the network packet fields of *temporary* pool point to shared memory while other data structures or metadata are stored safely in private memory. This arrangement allows the device to read and write packets from these memory buffers while keeping other data such as the

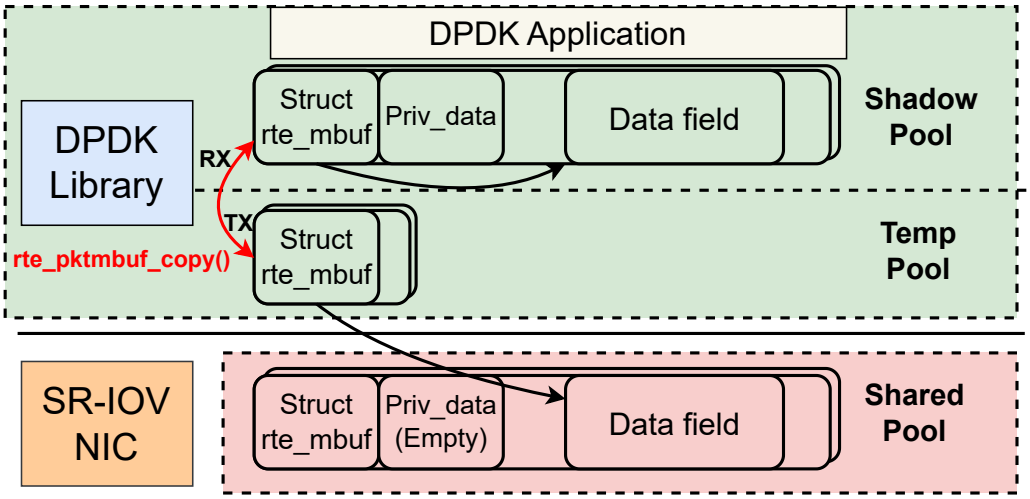


Figure 4-2: Shadow packet buffer pool design.

application's private data and pointers inaccessible to the untrusted devices. The *shadow* pool is the memory pool that is fully protected by the private memory and used by the application. As a result, DPDK applications can maintain their existing code behaviors without having to worry about source-code modification or any data leakage, including performing in-place packet processing directly in the data field. The memory copy operation between the *shadow* memory pool and the *temporary* memory pool is embedded in TX/RX-related functions.

Concealed shared region and fault handling

In the context of SEV with VirtIO, network applications can only interact with network devices through restricted interfaces such as system calls. In these system calls, the guest VM's kernel then formats or parses the raw network packets, and crucially copies them from/to the shared region, guaranteeing that the network application never interacts with or uses shared memory. By bypassing the kernel networking layer, DPDK eschews all protections provided by the guest VM's kernel. Thus, FOLIO must address the loss of these security protections and ensure that DPDK network applications do not mistakenly access the shared region. To address this concern, all virtual addresses pointing to shared regions are hidden from the network application, concealed within the Environment Abstraction Layer (EAL) of the DPDK library. This approach prevents network applications from inadvertently accessing the shared

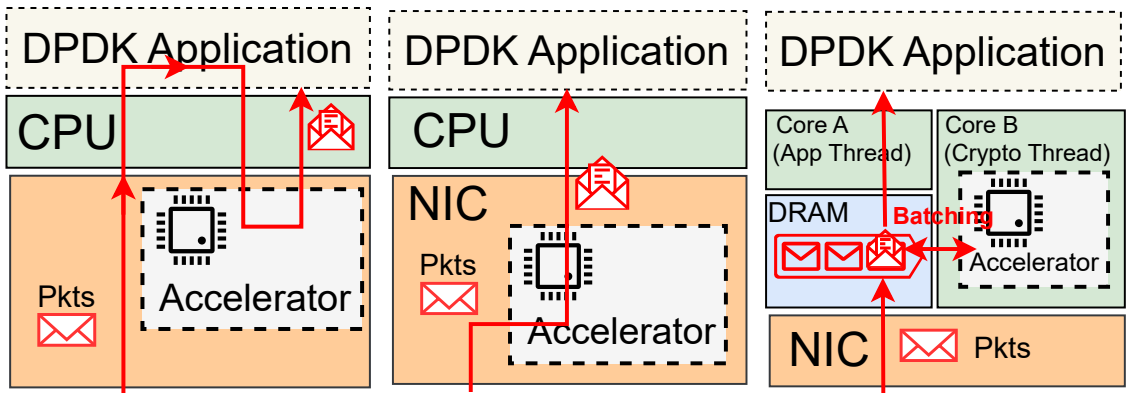
region. In the case of an application crash (potentially by a malicious hypervisor or device), improperly managed shared memory could compromise VM security if it were to be reused by another application. In FOLIO, the shared region management kernel module works with the guest VM OS to ensure that shared pages of memory are not reused until the management module clears the region and makes it private again.

Efficient I/O Event Handling

FOLIO naturally inherits the efficient optimizations of I/O event handling provided by DPDK, including the use of polling mode to avoid interrupt overhead and the use of huge pages to reduce memory lookup costs. Polling mode has additional performance benefits in the context of SNP; by avoiding frequent retrieval of timestamps and other special instructions that would trigger a `VMEXIT`, polling mode avoids both the expensive cost of the `VMEXIT` and cost of all VC handler overhead. By preallocating the shadow pool, FOLIO significantly reduces the overhead of bounce buffer operations to a single memory-copy of the packet data.

Hardware-accelerated End-to-end Encryption

Although FOLIO cannot offload cryptographic operations to the untrusted NIC, it can still leverage CPU crypto instructions to accelerate packet encryption. These crypto instructions remain non-interceptable and secure in the SNP setup, enabling FOLIO to securely and efficiently boost packet encryption and decryption. FOLIO supports two crypto offload modes: look-aside mode and inline mode. Look-aside mode is an offload mode in which the DPDK application must actively trigger hardware-accelerated decryption, as shown in Figure 4-3a. FOLIO can naturally support such mode by using DPDK's implementation of IPsec and only needs to specify CPU crypto instructions as the accelerator. In the original inline mode (Figure 4-3b), the NIC actively decrypts the packet upon receiving it. To support this mode without trusting the NIC, FOLIO introduces a CPU-based emulated inline mode (Figure 4-3c). In emulated inline mode, the application thread collaborates with a reserved crypto



(a) Look-aside Mode. (b) Inline Mode. (c) Emulated inline.

Figure 4-3: Three crypto offload methods.

thread to share the TX/RX queues. This crypto thread performs cryptographic tasks for data packets in real-time through polling and batching, efficiently offloading crypto tasks from the application thread. Both modes supported by FOLIO guarantee the secure storage of encryption keys within the confidential VM's vCPU side, thereby eliminating the risk of key leakage.

Code Compatibility

FOLIO makes no breaking changes to the DPDK API and is compatible with existing DPDK applications. Specifically, if the application uses the default method `rte_pktmbuf_pool_create` to initialize and use memory pools, *no* additional source-code changes are necessary. If the application instead allocates memory pools using a custom method, we provide a simple API to create the shared and temporary packet buffer pools; memory copies between shared and private regions are done automatically. Furthermore, for network applications using CPU-based cryptography offload as an hardware accelerator, no source-code modifications are required. However, to use CPU-based inline mode, a series of function calls need to be injected into the source code. These function calls are responsible for setting up and launching the crypto thread.

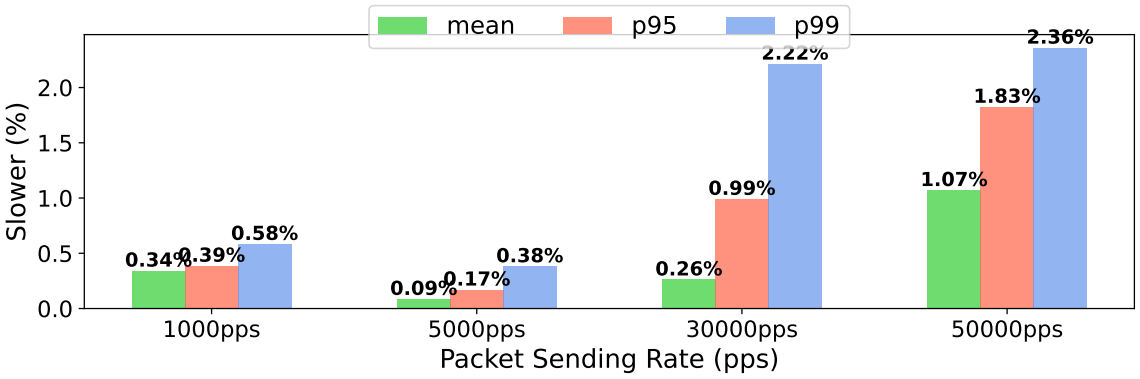
4.3 Evaluation

Again, our goal is to demonstrate that FOLIO can perform at levels comparable to the future idealized solution, SNP VMs running DPDK and TIO. We cannot directly benchmark the performance of TIO solutions because TIO devices are not available yet; we instead compare the performance of FOLIO to non-TEE VMs using plain DPDK and SR-IOV. The performance of non-TEE VMs should be an upper bound on the performance of SNP VMs using TIO, so comparing the performance of FOLIO to non-TEE VMs should provide a reasonable estimate or bound on the performance gap between FOLIO and SNP VMs using TIO. For the rest of this section, note that the term "performance of non-TEE VM" refers to the performance of a non-TEE VM using DPDK with a virtual function (VF) from SR-IOV and that the term "performance of SNP VM" refers to the performance of FOLIO inside an SNP VM using the same VF.

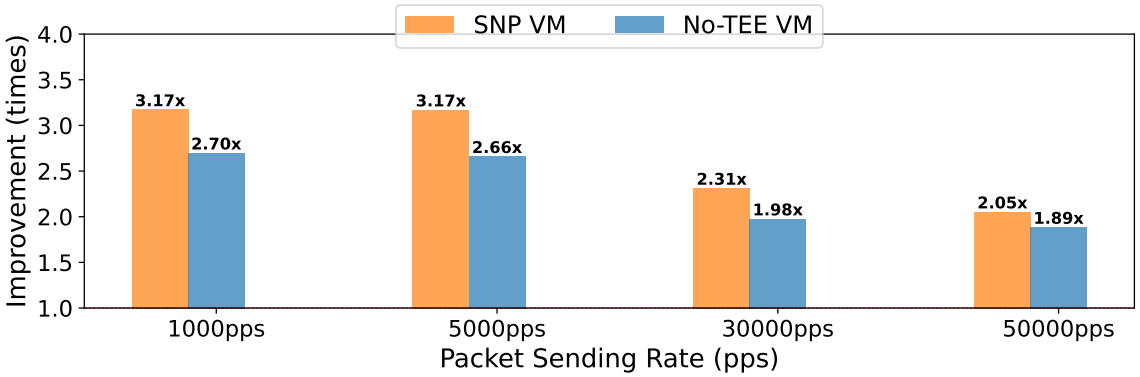
We evaluate FOLIO in four general categories: performance of a simple UDP server (Section 4.3.1), performance against a general network testing tool (Section 4.3.2), performance for IPsec data streams, and performance of a real-world DPDK application (Section 4.3.4). For all categories, our testing configuration is the same as described in Section 4.1.2.

4.3.1 Simple UDP Echo Server

We first repeated the experiments in Section 4.1 to get a baseline understanding of the performance differences between FOLIO and other configurations. Figure 4-4a compares the tail latency of FOLIO to the non-TEE VM for various packet sending rates. In this experiment, FOLIO demonstrated an average latency overhead of less than 1% (compared to the non-TEE VM); even the p99 tail latency overhead was within only 2.5%. Figure 4-4 compares the latency improvement of FOLIO to the latency improvement of a non-TEE VM — latency improvement refers to how much faster the system is compared to a corresponding configuration using just SR-IOV. FOLIO performs with an impressive latency improvement of about $2\times - 3\times$, negatively corre-



(a) SNP's percentage of slowdown compared to non-TEE VM.



(b) Improvement compared to SR-IOV only.

Figure 4-4: Tail latency between SNP and non-TEE VMs.

lated to the packets-per-second; in fact, FOLIO enhances SNP VM performance more than DPDK enhances a non-TEE VM using SR-IOV. These results are consistent with expectations as FOLIO only introduces a single memory-copy overhead.

4.3.2 Generalized Network Testing Tool

We used dperf, a well-established[5] DPDK-based benchmarking tool, to thoroughly assess the performance of FOLIO [10]. dperf was designed to benchmark network performance for a variety of network loads and has several useful features for our purposes. dperf can be configured with a multitude of configuration options — including packet size, packet rate, packet type, number of threads, number of concurrent connections, and connection live time — granting us a versatile testing environment. dperf

[5]dperf has more than 3.3k stars in GitHub.

also provides detailed statistics and metrics, including packet counts, throughput, and round-trip time (RTT), that offer valuable insights and information necessary for a comprehensive analysis of FOLIO's performance. Most importantly, dperf is capable of simulating heavy network traffic on the level of multiple Gigabits-per-second and tens of thousands of concurrent connections. This traffic volume is enabled by leveraging DPDK libraries on both the client and the server side. Finally, thanks to FOLIO's code compatibility with the original DPDK, dperf can be compiled and used in SNP VMs without any source-code modifications.

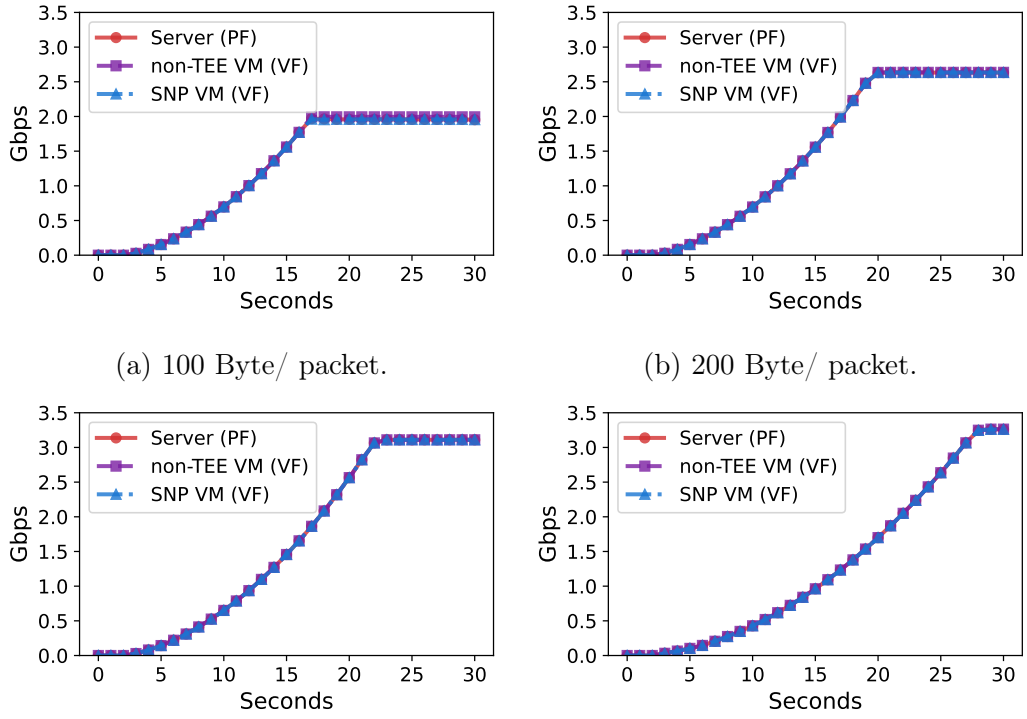
Throughput

We first compare the throughput of FOLIO and the non-TEE VM. Note that we also conducted tests in a non-virtualized environment (*i.e.*, running dperf directly on the server) using the network card's Physical Function (PF). These tests, marked as "server (PF)," serve as a control and represent the theoretical maximum bandwidth of the system (which is constrained by the NIC's performance and bandwidth).

In our throughput evaluation, we measured client throughput for various packet sizes (100 bytes, 200 bytes, 500 bytes, 1000 bytes). The client sent UDP packets due to its higher speed. Specifically, the client sent 1000 packets every second for each client connection to the server; to test the limits of the network throughput, the client incrementally increased the number of concurrent connections per second (CPS) until packet loss occurred. Every second, the client would increase its CPS by 5% of the maximum CPS, which we define as $\frac{8\text{ Gbps}}{N\text{byte/ packet}\times 1000\text{packets/ second}}$. The throughput is then simply the number of bits received by the server every second.

As shown in Figure 4-5, the results indicate that the difference in throughput between FOLIO and the non-TEE VM was minimal. For the different packet sizes (100, 200, 500, and 1000 bytes), FOLIO's throughput compared to the non-TEE throughput was 98.1%, 99.9%, 99.9%, and 99.8%, respectively. Furthermore, we also note that FOLIO's performance also matched the control experiments (direct execution on the server). Virtual environments are still affected by scheduling (as the hypervisor may choose to interrupt the VM arbitrarily), so the close performance is

notable.



(a) 100 Byte/ packet.

(b) 200 Byte/ packet.

(c) 500 Byte/ packet.

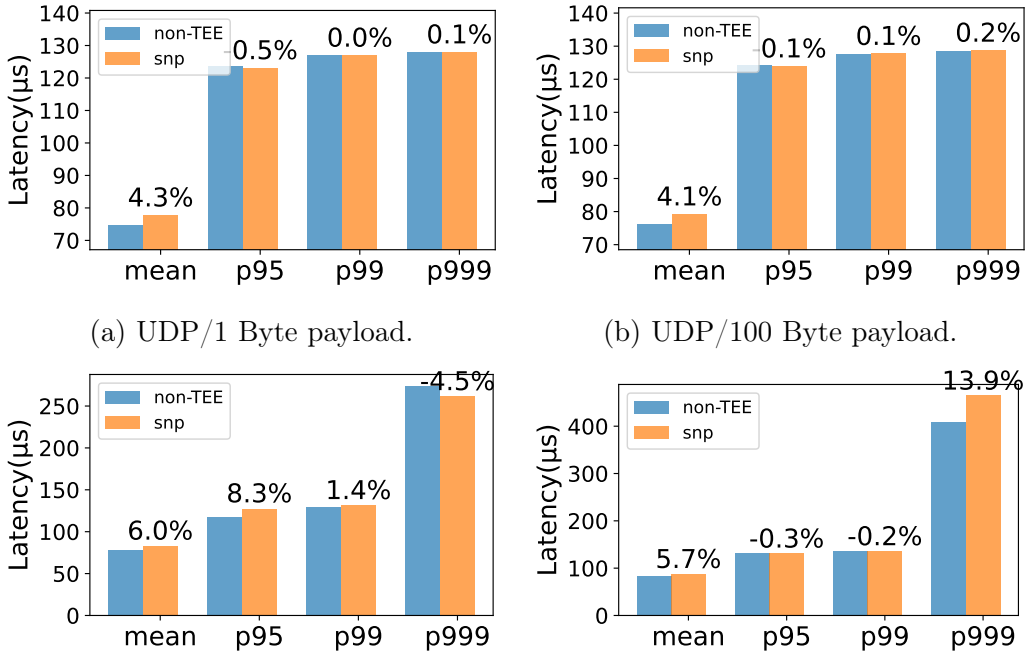
(d) 1000 Byte/ packet.

Figure 4-5: Throughput under different packet sizes.

Latency

Latency naturally represents the server's actual data processing time. We measured latency for UDP and TCP configurations, conducting tests at different stress levels. To simulate different stress levels, the client sent a packets of a variety of different sizes. We measured the round-trip time from the client side, which is specifically the duration between the client initiating a new connection and it receiving the first response from the server. For the UDP configuration, the client gradually increases the number of new connections per second until it reaches 20000, approximately 1/3 of the total port number 65535, and then maintains this level of connections for a stable period of 30 seconds. Under the TCP configuration, the client follows the three-way handshake principle to establish connections, and the server transmits the primary data payload during the third handshake. Every connection is closed after a

single round of communication. Figure 4-6 and Figure 4-7 show that FOLIO's average latency is consistently less than 6% slower than the non-TEE latency across different packet sizes. Moreover, a comparison of UDP and TCP packets shows that TCP demonstrates discernible tail latency variations. Specifically, the cases of 1 byte and 100 byte payloads exhibit a tail latency disparity of approximately 25% at the 99% significance level.



(a) UDP/1 Byte payload.

(b) UDP/100 Byte payload.

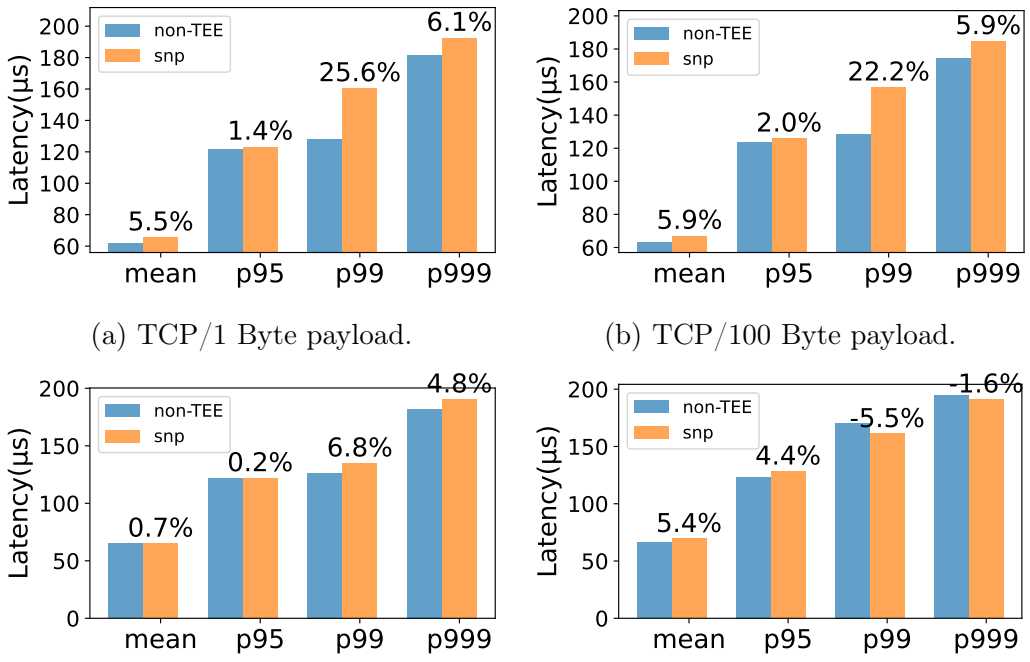
(c) UDP/500 Byte payload.

(d) UDP/1000 Byte payload.

Figure 4-6: Mean and tail latency for UDP workload.

Multi-thread

Finally, we tested the performance FOLIO in a multi-threaded system using RSS (Receive Side Scaling). When RSS is enabled, all received packets are automatically spread across the different threads and cores in the dperf server. Using the same latency testing strategy and environment from before, we measured the latency of UDP connections with a payload size of 1000 bytes for dperf servers with both one or two threads. As shown in Figure 4-8, multi-threading slightly increases the latency, most likely from the overhead of the RSS feature. However, FOLIO's latency remains



(a) TCP/1 Byte payload.

(b) TCP/100 Byte payload.

(c) TCP/500 Byte payload.

(d) TCP/1000 Byte payload.

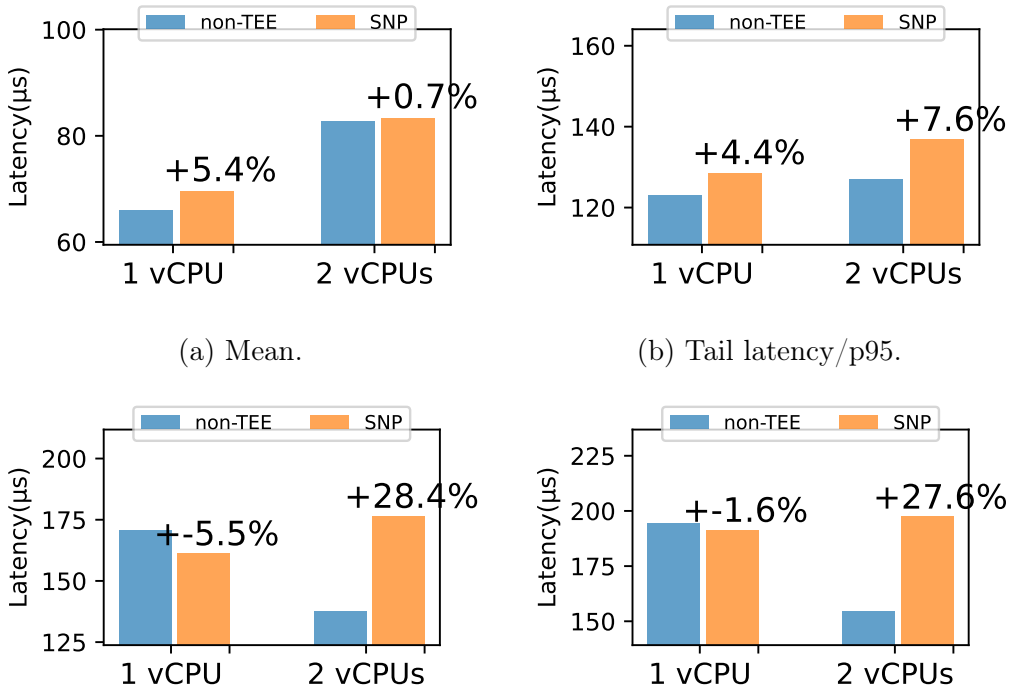
Figure 4-7: Mean and tail latency for TCP workload.

within within 6% of the non-TEE system.

4.3.3 IPsec Performance

To evaluate FOLIO's performance with IPsec, we enhanced the dperf to send and receive IPsec data streams. We focused on analyzing the performance impact of encrypting and decrypting data end-to-end. Therefore, all packets were sent through a single transport-mode IPsec connection (in which source and destination addresses are not encrypted for routing purposes) and the IPsec secret keys were hardcoded on both server and client sides. Each packet was 1000-bytes and encrypted or decrypted using 128-bit AES in GCM mode.

Regrettably, due to hardware limitations with the network card, the SR-IOV Virtual Functions used in our experiments did not support IPsec offload inline mode, even in non-virtualized environments. Thus, we were unable to benchmark this mode. Instead, we used a CPU look-aside mode (marked as "Server (PF)") running in a non-virtualized environment with the PF directly as the reference for upper bound.



(a) Mean.

(b) Tail latency/p95.

(c) Tail latency/p99.

(d) Tail latency/p999.

Figure 4-8: Comparison between 1 vCPU and 2 vCPUs.

CPU look-aside mode

To evaluate CPU look-aside mode, we ran throughput and latency tests configured similarly to tests from Section 4.3.2. Figure 4-9 compares IPsec throughput. Once again, FOLIO nearly matches the non-TEE VM and the native environments, reaching 99.8% of the non-TEE throughput. Predictably, IPsec did decrease overall throughput, resulting in a 9.85% decrease. Figure 4-10 shows the difference in percentage between non-TEE VM and the server, and between FOLIO and non-TEE VM. On average, FOLIO exhibited a latency 6.96% higher than the non-TEE VM.

CPU-enabled emulated-inline Mode

Evaluating the emulated inline mode is trickier than the previous tests because it requires reserving an extra vCPU. In most real network applications, it would be simpler and wiser to just run the application with an additional CPU thread. Such an approach eliminates overheads from shared queues and communication between

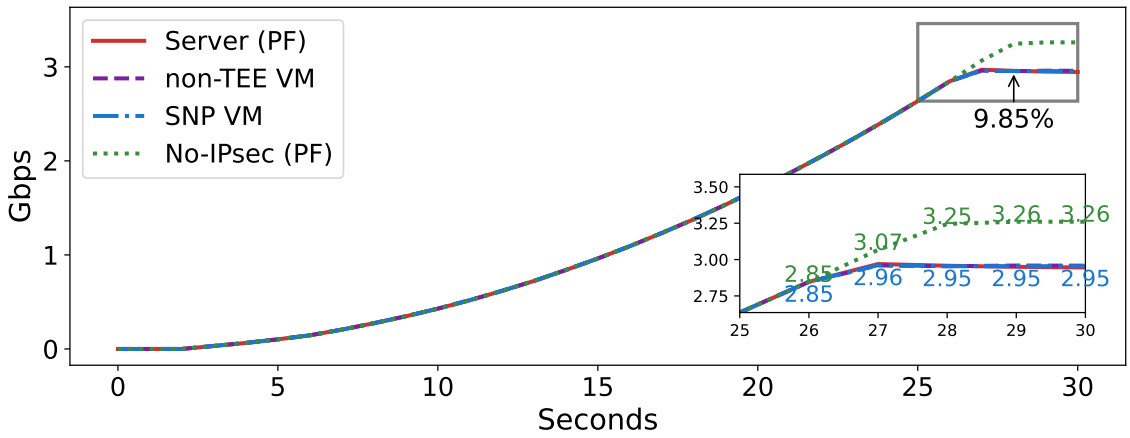


Figure 4-9: Throughput when enabling IPsec.

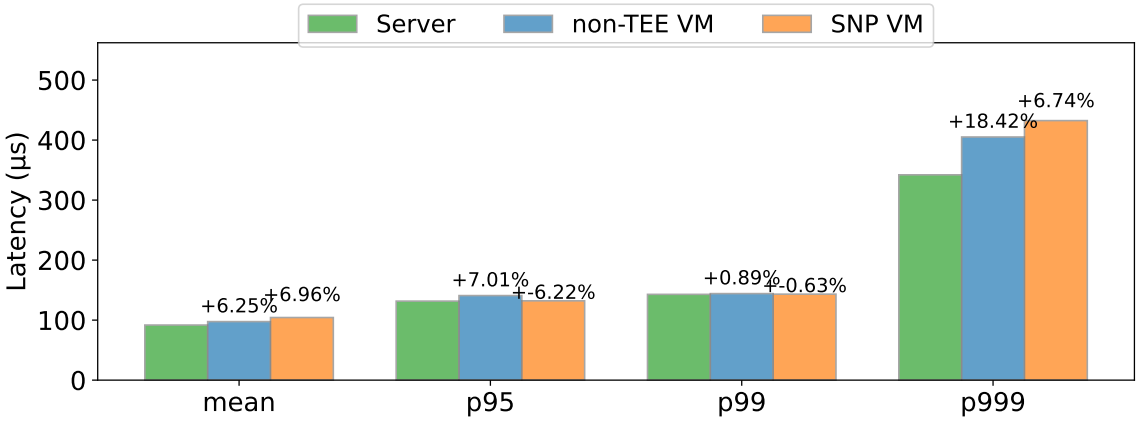


Figure 4-10: IPsec latency.

the application thread and the extra cryptography thread. However, emulated inline mode may still be useful for single-threaded network application that suffer from CPU pressure.

To accurately simulate such conditions, we introduced an additional per-packet processing delay that simulates a more realistic post-decryption workload, and measure the average latency for different processing delay times. Indeed, emulated mode is capable of alleviating pressure from the application thread, as shown in Figure 4-11. At a workload of 135 milliseconds, the latency of CPU mode spiked by about a factor of 10. Emulated mode, however, only experience packet loss at a workload of 145 milliseconds.

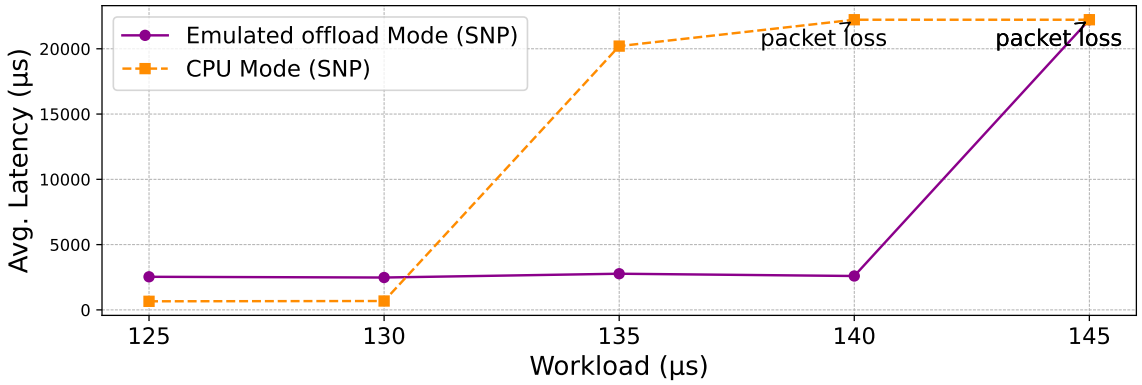


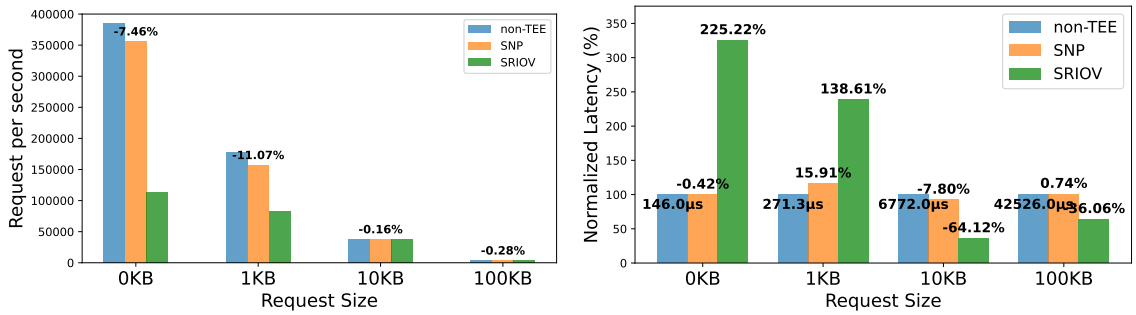
Figure 4-11: Performance of emulated-inline mode.

4.3.4 Real World DPDK Applications

To showcase the performance of FOLIO in real-world applications, we evaluated the performance the performance of a fork of nginx [38] based on F-Stack [13] (`Commit:dd27d06`). F-Stack is an open-sourced network framework based on DPDK, providing POSIX APIs (Socket, Epoll, Kqueue), a user-space TCP/IP stack (port FreeBSD), a programming SDK (Coroutine), and application interfaces (nginx) that assist network applications in benefiting from DPDK. We focus on comparing the performance of F-Stack-provided nginx on FOLIO against non-TEE VM. As a control, we also show the performance of an standard version of nginx (obtained from `apt-get`) running in a non-TEE VM with SR-IOV. We note that this evaluation also showcases the code compatibility of FOLIO with existing DPDK applications.

Following official guidance [43], we measured the requests-per-second (RPS) and latency following official guidance. The client side ran a HTTP benchmarking tool called wrk [52]. We collected each metric against various requested file sizes. For the RPS test, we ran 12 independent wrk to gather the maximum RPS. Each instance opened 50 HTTP connections for 1 minute. For the latency test, we ran 1 wrk instance for 5 minutes to get more stable results.

Figure 4-12a shows the total RPS. While there is a performance gap of 11.07% for the 1KB test, this gap shrinks to negligible as the request size increases. Figure 4-12b shows the average latency. Curiously, when the request size exceeds 10KB, the original version of nginx shows better performance. We speculate that this discrepancy is due



(a) RPS Comparison.

(b) Latency Comparison.

Figure 4-12: Nginx Performance.

to optimization differences between the different versions of nginx; the standard version could include additional optimizations for large files, or the F-stack version may not have considered such use cases and refrained from implementing them. Therefore, for the larger request sizes, we primarily focus on the performance variance between the identical F-stack nginx code executed in the SNP VM (using FOLIO) and in the non-TEE VM. Although there is a higher performance difference shown in these results, that difference is the combination of the performance difference of both the core program execution and all network I/O operations. The former depends highly on the implementation of the application and cannot be mitigated through TIO solutions. Crucially, our experiments show that FOLIO can be reasonably applied to complex existing network frameworks.

4.4 Comparison of FOLIO with TIO Solutions

4.4.1 Security Comparison

Due to distinct threat models, FOLIO and TIO solutions attain varying security levels and TCB sizes.

FOLIO: end-to-end security

FOLIO shares the same threat model and accomplishes the same end-to-end security level as the default VirtIO solution. They both exclusively rely on the CPU side,

thereby necessitating the employment of software-based encryption for ensuring comprehensive I/O security. Various software-based solutions can be utilized to ensure end-to-end security, such as IPsec at the network layer or SSL/TLS at the application layer. Considering the performance of SSL/TLS may vary based on the application being used, we use IPsec in this paper to evaluate the potential throughput when software-based encryption is enabled. Furthermore, it is important to mention that malicious or vulnerable DPDK applications are beyond the scope of this paper's discussion as SNP's threat model assumes that all software components within SNP VMs are trustworthy.

TIO: VM-to-NIC security

TIO solutions achieve a different VM-to-NIC security. The different threat model adopted by TIO, where the device and VM both reside within the TCB, fosters mutual trust between devices and VMs, allowing them to operate within a private memory region protected from the untrusted hypervisor. This, together with encryption on the PCIe buses, ensures the security of VM-to-NIC communication, safeguarding it from the hypervisor. This design also reduces some attack surfaces during I/O communication. For instance, the driver interfaces now reside completely in private memory, mitigating attacks that attempt to explore vulnerable driver implementations [17]. Even with the TIO solution, software-based encryption may still be necessary to prevent potential leakage during network routing.

TCB size

FOLIO only slightly increases the TCB size compared to the original software stack whereas TIO solutions significantly increase TCB size by including the I/O device in the trust boundary. FOLIO introduced approximately 2K lines of code modifications to the DPDK library, and the shared region management kernel module added 0.5K lines of code. Additionally, we added 1.2K lines of code to dperf to support IPsec benchmarking. Apart from these changes, we made less than 5 lines of code changes to the F-stack source code, primarily focusing on fixing inconsistent kernel headers,

to ensure successful compilation with the kernel version running in our VMs.

4.4.2 Performance Comparison

FOLIO can only benefit DPDK-compatible applications and has extra memory overhead compared to TIO solutions.

Network application without DPDK

Although FOLIO can achieve the same-level performance compared to the idealized solution, where both DPDK and TIO are utilized, network applications without DPDK support cannot directly benefit from FOLIO. On the contrary, with TIO support, network applications using POSIX APIs can also potentially benefit from the overhead mitigated by the TIO device. Luckily, FOLIO can work together with some software-based network frameworks, like F-stack [13], to alleviate SNP-specific overhead for applications using POSIX APIs. These frameworks embed DPDK to provide POSIX APIs or other network socket interfaces, such as user-space TCP/IP stacks, to help network applications benefit from DPDK.

Additional Memory Overhead

One limitation of FOLIO is the additional memory overhead. To avoid the extra allocation delay during runtime, FOLIO allocates all data structures required by the shadow memory pool design during DPDK initialization. By default, this setting approximately doubles the memory pool size used, as we configure the shared memory pool to be the same size as the shadow memory pool. In situations of excessive memory overhead, FOLIO offers a special configuration option for a constant-sized memory overhead. In practical scenarios, each device can only accommodate a fixed number of P packets for TX and RX, significantly smaller than the number of reserved memory buffers. Thus, the shared and temporary memory pools only need to be large enough to store P packets at a time. FOLIO supports the application to adjust such size through additional configurations.

4.5 Related Work

During the development of this research, we became aware of another project [29] that was also investigating network performance in confidential VMs. Despite the similarities in the research area, our work significantly differs from theirs in several ways: (1) Overhead identification: our work also comprehensively analyze the individual impacts of all known types of network overhead. (2) Research vision: while this project focuses primarily on the future of high-performance I/O with SR-IOV and DPDK support, their focus was on improving the focus of QEMU-based emulated I/O devices. Both techniques are necessary and cater to different types of use-cases for confidential VMs. (3) Optimization methods: Li *et al.* using various methods to reduce bounce buffer overhead, including packet pre-processing and combining encryption and memory-copy operations during I/O payload bouncing. In contrast, our approach centers on eliminating all factors within an SNP configuration that lead to poor network performance.

Other related TEE network performance optimizations [8, 40, 41] primarily target enclave-based TEE (Intel SGX [12]) and involve using extra I/O threads to achieve exitless I/O communication with SGX. Rkt-I/O [47] also utilizes DPDK to enhance network performance by embedding it into LibOS as a hidden network driver. However, a significant distinction between our work and theirs is that our approach fully utilizes DPDK without the LibOS layer, motivating us to provide a security-oriented set of interfaces distinct from theirs, while ensuring end-to-end IPsec and maintaining code compatibility. Remarkably, our research also reveals that existing SNP VMs, with the support of Folio based on a secure and efficient DPDK, can also achieve the same level of performance as future TIO solutions.

TEE I/O security Hetzelt *et al.* [17] develop a dynamic fuzzing tool for testing device interfaces in confidential VM, identifying 50 bugs in various device drivers. Their work holds significant value in real-world TEE setups, as all code running inside the confidential VM is considered trusted within its thread model. Therefore, driver bugs, such as buffer overflows, can easily undermine the security guarantees

offered from internal. Lefeuvre *et al.* [28] discuss the requirement for fast confidential I/O, primarily focusing on the proper boundaries between the host and TEE I/O.

Chapter 5

Conclusion

In this thesis, we presented two different contributions in the field of confidential VMs.

In Chapter 3, we discussed the design of a RISC-V confidential VM, focusing on the task of secure CPU virtualization. Using the RISC-V specification and the source code of the Linux-KVM and OpenSBI, we first created a precise specification detailing which registers the hypervisor needed to properly handle interrupts and exceptions. This specification served as an analog for the AMD GHCB. We then discussed the design and implementation details of our security monitor, which concealed CPU register state according to this specification.

In Chapter 4, we researched the performance of network I/O in confidential VMs. We first conducted an extensive evaluation of all possible sources of I/O overhead in a confidential VM. Using our evaluation as a guide, we created Folio, a system for high-performance I/O using DPDK in AMD SEV-SNP. Unlike TIO devices, Folio does not sacrifice any security assumptions. Our evaluation of Folio showed that it performed within 6% of the ideal network I/O performance.

Bibliography

[1] Amazon. The Security Design of the AWS Nitro System. https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.html?did=wp_card&trk=wp_card, 2022.

[2] Amazon. Amazon EC2 now supports AMD SEV-SNP. https://aws.amazon.com/about-aws/whats-new/2023/04/amazon-ec2-amd-sev-snp/, 2023.

[3] AMD. Sev-es guest-hypervisor communication block standardization. https://developer.amd.com/wp-content/resources/56421.pdf.

[4] AMD. SEV secure nested paging firmware API specification. *API Document*, 2020.

[5] AMD. AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization. *White paper*, 2023.

[6] AMD. AMDSEV/sev-snp-devel branch. https://github.com/AMDESE/AMDSEV/tree/sev-snp-devel, 2023.

[7] ARM. ARM CCA Security Model 1.0, 2021.

[8] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[9] Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[10] Baidu. dperf. https://github.com/baidu/dperf, 2023.

[11] Confidential Computing Consortium. Confidential Computing Consortium Members. https://confidentialcomputing.io/members/, 2022.

[12] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[13] f-stack. F-stack. `https://github.com/F-Stack/f-stack`, 2023.

[14] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the {PENGLAI} enclave. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 275–294, 2021.

[15] Google. Introducing Google Cloud Confidential Computing with Confidential VMs. `https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms`, 2020.

[16] Samuel Greengard. Will risc-v revolutionize computing? *Communications of the ACM*, 63(5):30–32, 2020.

[17] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. Via: Analyzing device interfaces of protected virtual machines. In *Annual Computer Security Applications Conference*, pages 273–284, 2021.

[18] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. *Cryptology ePrint Archive*, 2015.

[19] Intel. Intel trust domain extensions whitepaper. `https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-final9-17.pdf`, 2020.

[20] Intel. Intel TDX Connect TEE-IO Device Guide. *White paper*, 2023.

[21] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[22] David Kaplan. Protecting vm register state with sev-es. *White paper*, 2017.

[23] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.

[24] keystone-enclave. Keystone security monitor. `https://github.com/keystone-enclave/sm`, 2023.

[25] Awais Khan, Arnab K Paul, Christopher Zimmer, Sarp Oral, Sajal Dash, Scott Atchley, and Feiyi Wang. Hvac: Removing i/o bottleneck for large-scale deep learning applications. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 324–335. IEEE, 2022.

[26] KVM. Code — kvm,, 2015. [Online; accessed 19-July-2023].

[27] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[28] Hugo Lefeuvre, David Chisnall, Marios Kogias, and Pierre Olivier. Towards (Really) Safe and Fast Confidential I/O. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 214–222, 2023.

[29] Dingji Li, Zeyu Mi, Chenhui Ji, Yifan Tan, Binyu Zang, Haibing Guan, and Haibo Chen. Analysis and optimization of network i/o tax in confidential virtual machines. In *Proceedings of the 2023 USENIX Conference on Usenix Annual Technical Conference*.

[30] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1541–1541. IEEE Computer Society, 2022.

[31] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. CROSSLINE: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV. *arXiv preprint arXiv:2008.00146*, 2020.

[32] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. Tlb poisoning attacks on amd secure encrypted virtualization. In *Annual Computer Security Applications Conference*, pages 609–619, 2021.

[33] Michael McReynolds. Azure announces next generation Intel SGX confidential computing VMs. `https://techcommunity.microsoft.com/t5/azure-confidential-computing/azure-announces-next-generation-intel-sgx-confidential-computing/ba-p/2839934`, 2021.

[34] Microsoft. Azure and AMD announce landmark in confidential computing evolution. `https://azure.microsoft.com/en-us/blog/azure-and-amd-enable-lift-and-shift-\confidential-computing/`, 2021.

[35] Mathias Morbitzer, Manuel Huber, and Julian Horsch. Extracting secrets from encrypted virtual machines. In *9th ACM Conference on Data and Application Security and Privacy*. ACM, 2019.

[36] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's virtual machine encryption. In *11th European Workshop on Systems Security*. ACM, 2018.

[37] RISC-V Community News. Accelerating ml recommendation with over 1,000 risc-v/tensor processors on esperanto's et-soc-1 chip | david r. ditzel, esperanto technologies inc. https://riscv.org/blog/2022/07/accelerating-ml-recommendation-with-over-1000-risc-v-tensor-processors-on-esperan 2022.

[38] Nginx. NGINX: Advanced Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com/, 2023.

[39] OpenSBI. RISC-V Open Source Supervisor Binary Interface (OpenSBI). https://github.com/riscv-software-src/opensbi, 2023.

[40] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 238–253, 2017.

[41] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host OS interface for trusted execution. *arXiv preprint arXiv:1908.11143*, 2019.

[42] Anil Rao. Rising to the Challenge—Data Security with Intel Confidential Computing. https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Rising-to-the-Challenge-Data-Security-with-Intel-Confidential/post/1353141, 2022.

[43] Amir Rawdat. Testing the Performance of NGINX and NGINX Plus Web Servers. https://www.nginx.com/blog/testing-the-performance-of-nginx-and-nginx-plus-web-servers/, 2017.

[44] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, 2009.

[45] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[46] AMD SEV-SNP. Strengthening vm isolation with integrity protection and more. *White Paper, January*, 2020.

[47] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. Rkt-io: A direct i/o stack for shielded execution. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 490–506, 2021.

[48] torvalds. linux. https://github.com/torvalds/linux, 2023.

[49] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[50] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanović. The risc-v instruction set manual volume i: Unprivileged isa. `https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf`, 2019.

[51] Hauser Waterman, Asanovic. The risc-v instruction set manual volume 2: Privileged architecture version 1.12. Technical report, University of California at Berkeley Berkeley United States, 2021.

[52] WG. wrk - a http benchmarking tool. `https://github.com/wg/wrk/`, 2023.

[53] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, et al. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 1042–1057, 2022.