Article

Not peer-reviewed version

HyperShield: An Automated Evaluation Platform for Security and Performance Trade-Offs in Virtual Systems

Faiz Alam *, Mohammed Mubeen Mifthak, Sahil Purohit, Md Shadab, Gregory T Byrd *, Khaled Harfoush *

Posted Date: 9 March 2026

doi: 10.20944/preprints202603.0551.v1

Keywords: virtual machines; containers; cloud computing; IOMMU; kernel space

 doi:10.20944/preprints202603.0551.v1

Article

HyperShield: An Automated Evaluation Platform for Security and Performance Trade-Offs in Virtual Systems †

Faiz Alam [1,*], Mohammed Mubeen Mifthak [1], Sahil Purohit [1], Md Shadab [2], Gregory T. Byrd [1,*] and Khaled Harfoush [1,*]

[1] North Carolina State University, USA
[2] Arizona State University, USA
* Correspondence: falam3@ncsu.edu (F.A.); gbyrd@ncsu.edu (G.T.B.); kaharfou@ncsu.edu (K.H.)
† This paper is an extended version of our paper published in Alam, F.; Mifthak, M.M.; Purohit, S.; Shadab, M.; Byrd, G.T.; Harfoush, K. VirtShield: A Security Evaluation Framework for Virtualized and Containerized Systems. In Proceedings of the IEEE Consumer Communications & Networking Conference 2026, Las Vegas, NV, USA, 9-12 January 2026 [1].

Abstract

Virtualization is the building block of modern cloud computing infrastructure. However, it remains vulnerable to a range of security threats, including malicious co-located tenants, hypervisor vulnerabilities, and side-channel attacks. These threats are generally mitigated by developing and deploying advanced and complex security solutions that incur significant performance overhead. Prior work on Virtual Machines (VMs) and containers has mainly evaluated basic security solutions, such as firewalls, using narrow performance metrics and synthetic models within limited evaluation frameworks. These studies often overlook advanced security modules in both user and kernel space, lack flexibility to incorporate emerging features, and fail to capture detailed system-level impacts. We address these gaps with HyperShield, an open-source framework for unified security evaluation across VMs and containers that mimics a realistic cloud infrastructure. HyperShield supports advanced security modules in both user and kernel space, providing rich system-level performance metrics for comprehensive evaluation. Our performance evaluation shows that containers generally outperform VMs due to their lower virtualization overhead, achieving a throughput of 9.38 Gb/s compared to 1.98 Gb/s for VMs for our benchmarks. However, VM's performance is comparable for kernel space deployments, as Docker uses the shared kernel space of the Docker bridge, which can result in packet congestion. In latency-sensitive workloads, VM access latency of 14.91 ms is comparable to Docker's 12.86 ms. In storage benchmarks, FIO, however, VMs outperform Docker due to the overhead of Docker's layered, copy-on-write file system, whereas VMs leverage optimized virtual block devices with near-native I/O performance. These results highlight performance dependencies on benchmark choice, trade-offs in deploying security workloads between user and kernel space, and the choice of containers and virtual machines as virtualization environments. Therefore, HyperShield provides a comprehensive evaluation toolkit for exploring an optimal security module deployment strategy.

Keywords: virtual machines; containers; cloud computing; IOMMU; kernel space

1. Introduction

Virtualization has transformed cloud computing by enabling efficient resource sharing, allowing providers to deliver scalable, on-demand services to multiple tenants. Platforms such as Amazon EC2 provide Infrastructure as a Service (IaaS), which forms the backbone of modern computing, ranging from web hosting to large-scale data analytics [2,3]. As cloud infrastructure scales, ensuring both performance and security is critical, as multi-tenant cloud environments become increasingly complex and vulnerable to both external and internal threats. While virtualization enables secure resource

 doi:10.20944/preprints202603.0551.v1

sharing across tenants, shared hardware resources and hypervisors can be easily exploited through vulnerabilities and side channels [4], facilitating sophisticated attacks [5]. To mitigate such risks, additional isolation and security mechanisms are continuously developed and deployed to maintain the integrity of these services [6].

As security threats become increasingly sophisticated, security researchers and practitioners continually innovate and design new security solutions. Similarly, organizations with proprietary and sensitive intellectual property are compelled to implement robust protections to prevent costly IP leakage [7,8]. However, prior studies have primarily evaluated basic security mechanisms, such as firewalls, in VMs and containers using narrow, benchmark-driven metrics [9,10]. These efforts often overlook complex security modules that operate in both user and kernel space, and they lack the flexibility to incorporate emerging features for comprehensive analysis. Furthermore, most prior work evaluates security solutions in isolation or using synthetic models rather than within realistic multi-layered deployments. They rely solely on coarse-grained metrics, such as throughput and latency, and ignore detailed system-level effects, including address translation overhead, CPU utilization, CPU scheduling, core activity, and CPU Microarchitecture Performance. Additionally, they do not provide an open, extensible framework that others can adapt for reproducible experimentation [11,12].

To better understand and evaluate trade-offs between security and performance, we developed **HyperShield**. HyperShield is an open-source, configurable framework designed for VMs and containers to study end-to-end performance characterization of emerging security modules. HyperShield emulates a real cloud system, providing a platform with a native Linux host and precise network topology for studying the impact of security modules across the microarchitecture, OS, and networking layers. As a case study, we evaluated Suricata, a user space security module, alongside a lightweight packet sniffer deployable in both user and kernel space. Our results show that Docker generally outperforms VMs in throughput-sensitive benchmarks, sustaining 9.38 Gb/s compared to 1.98 Gb/s for VMs, primarily due to the high virtualization overhead of VMs. However, when security modules are deployed in kernel space, the throughput of VMs and Docker is comparable, as Docker incurs significant packet traversal overhead when multiple containers share the Docker bridge in kernel space. In latency-sensitive workloads, VM access latency (14.91 ms) is close to that of Docker (12.86 ms). In storage benchmarks (FIO), VMs outperform Docker because Docker's layered filesystem introduces copy-on-write and traversal overhead, whereas VMs use optimized virtual block devices with near-native I/O performance. Overall, VMs demonstrate competitive performance in kernel-space deployments and storage benchmarks, narrowing the performance gap with containers.

In summary, this paper makes the following key contributions:

(1) Open-source Testbed: HyperShield provides an end-to-end performance characterization testbed of any security modules deployed in user and kernel space for both containers and VMs. It can capture a range of hardware, software, and networking performance metrics.

(2) Flexible and Modular Architecture: It decouples the client, server, and security layers and includes parametrized representative benchmarks for networking, databases, and storage. It enables the extension and integration of new workloads and novel security solutions for performance studies in a selected region of interest within the entire system stack.

(3) Realistic and Reproducible System Environment: HyperShield operates directly on real systems running a native Linux OS and networking stack, avoiding reliance on synthetic models or simulations. Its automated deployment, benchmarking, and result collection tools ensure reproducible experiments and enable fair comparisons across diverse environments.

(4) Use-cases and Extensibility: We demonstrate HyperShield's utility through the evaluation of an open-source and widely used security engine, Suricata [13]. It also includes a generic, extendable packet sniffer module deployable in both user and kernel space, providing a foundation for extending HyperShield to study emerging security mechanisms.

The rest of this paper is organized as follows: Section 2 provides a brief background of VM and Containers technologies. Section 3 reviews related work and outlines the motivation for this study.

 doi:10.20944/preprints202603.0551.v1

Section 4 describes the HyperShield framework in detail. Section 5 presents our experimental setup and benchmarks. Section 6 discusses the evaluation results under different configurations. Finally, Section 8 concludes with key insights and directions for future work.

2. Background

This section provides the necessary background to understand and interpret the rest of the paper.

2.1. Virtualization Fundamentals

Virtualization is an abstraction that turns physical hardware, such as CPUs, GPUs, networks, memory, and storage, into a pool of resources that can be shared across multiple tenants (users). Virtualization enables resources to be either time-shared or space-shared, which makes the system more adaptable, scalable, and efficient. CPU virtualization allows numerous virtual CPUs (vCPUs) to run concurrently on a limited number of physical cores. Similarly, memory virtualization provides the illusion of infinite virtual memory for applications, which are dynamically mapped to the finite physical memory. IO virtualization enables virtual devices and interfaces to access physical I/O resources, allowing multiple virtual machines or containers to share hardware safely and efficiently. Hypervisor, a software layer, manages various pieces of the virtual system and manages virtual to physical translations. Virtualization technologies, such as Intel VT-x and AMD-V, provide hardware support for the hypervisor. The hypervisor maintains separate execution contexts per VM and rapidly switches between them with minimal overhead. VM scheduling, which maps virtual systems to physical systems (e.g., vCPUs to physical cores), uses scheduling policies such as credit-based scheduling [14,15]. GPU virtualization extends this model to accelerators, allowing multiple VMs or containers to share a physical GPU. The APIs allow sharing of physical GPU or full device passthrough. Pass through [16] provides tenants with direct access to a physical GPU with near-native performance. [17].

2.2. Memory Virtualization in Virtual Machines

Memory virtualization is critical to enable virtualization, creating the illusion that any process has access to an infinite memory space, called a virtual address space. The virtual address is then translated into a limited set of physical addresses. Virtual address provides each process with private memory space, which provides security and isolation. It also allows processes to use simple programming model and contiguous memory model, even when physical memory is limited or fragmented. The virtual to physical address translation is done through several key hardware and software components which is part of the Memory Management Unit (MMU). The MMU uses a multi-level page table to translate virtual addresses to physical addresses. The page table, a software structure stored in main memory, contains virtual-to-physical address translations. The page table is maintained per process and is managed by the OS. The logic to fetch translations from the page table is called the Page Table Walker (PTW), which can be implemented in software or hardware. There is a special cache called the Translation Lookaside Buffer (TLB) that caches recently used translations [18,19].

In a virtualized system, this address-translation process becomes even more complex because virtual machines introduce an additional layer of indirection. Each guest VM maintains its own page table to translate guest virtual addresses to guest physical addresses. The guest physical address is still a virtual address, which is then translated by the hypervisor-managed page table into a system physical address. Hence, the modern system contains a nested page table for address translation in VM. Intel refers to this as Extended Page Tables (EPT), while AMD calls it Nested Page Tables (NPT) [20,21]. If a memory access needs translation, the MMU performs a two-level translation, first walking the guest OS managed page tables to obtain a guest-physical address, and then consulting the hypervisor-managed nested page tables to translate that address into an actual host-physical location. Hardware-managed TLBs are extended to cache both levels of translation, reducing the cost of nested page walks. Despite these optimizations, nested address translation remains a major contributor to virtualization overhead in memory-intensive workloads [22–24].

 doi:10.20944/preprints202603.0551.v1

2.3. Device Virtualization in Virtual Machines

Similar to memory virtualization, device virtualization enables multiple virtual machines to share the physical I/O devices while maintaining strong isolation, performance, and correctness guarantees. Hypervisors use hardware and software mechanisms to virtualize devices, employing three main techniques: full device emulation, paravirtualized drivers, and hardware-supported virtualization via an I/O Memory Management Unit (IOMMU). First, full emulation allows a VM to interact with a virtual hardware device implemented in software that mimics the behavior of a real physical device (e.g., an Intel E1000 NIC or an AHCI controller), as shown in Figure 1. The hypervisor intercepts guest device requests and returns responses from the emulated hardware. Second, paravirtualized drivers such as VirtIO eliminate the need for expensive device emulation by exposing a lightweight interface to the guest OS. These drivers eliminate the need for full device emulation and instead communicate with the physical device via hypervisor backends with shared memory rings, delivering near–native performance. Third, hardware-assisted I/O virtualization through the IOMMU (e.g., Intel VT-d, AMD-Vi) enables secure sharing of a physical device from a guest VM as shown in Figure 2. The IOMMU provides hardware support for address translation for device virtual devices performing DMA. IOMMU translates DMA access to device virtual addresses to host physical addresses using per-device or per-VM page tables (e.g., Intel VT-d, AMD-Vi) [25–27]. The device passthrough allows a physical device to be directly attached to a VM, providing flexibility and native performance.

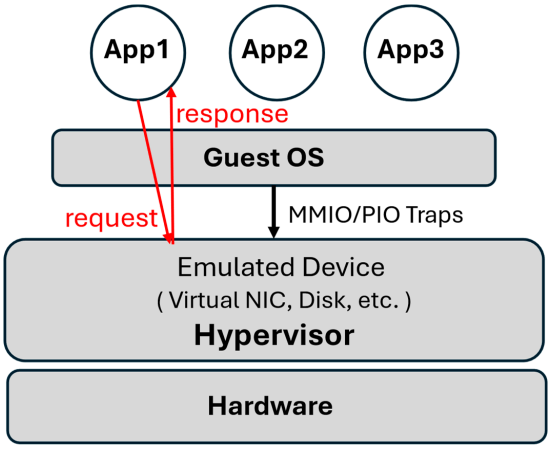

Figure 1. Full Device Emulation.

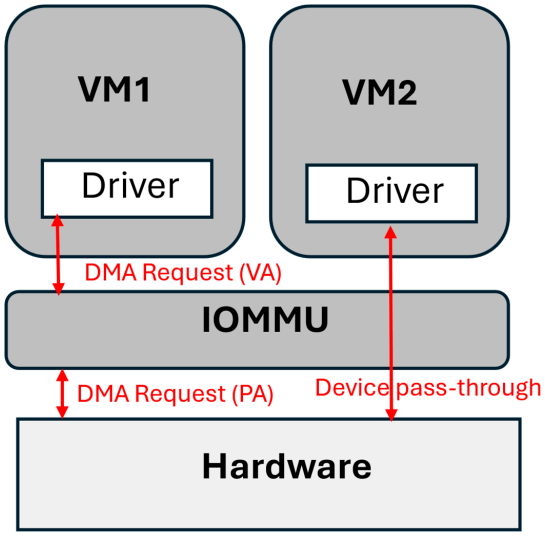

Figure 2. IOMMU based virtualization.

 doi:10.20944/preprints202603.0551.v1

2.4. VM Architecture

VMs rely on several key components to deliver flexibility, isolation, and efficient resource utilization among co-located tenants. Figure 3 shows a typical virtualized system running multiple VMs. Each VM operates with its own independent guest operating system (OS), e.g., Linux, Windows, or macOS, which is managed by the hypervisor. The guest OS interacts with emulated hardware in isolation while the hypervisor manages a fair share of physical hardware resources among multiple VMs [28].

The emulated hardware, such as virtual CPUs (vCPUs), is allocated to VMs by the hypervisor, which schedules these vCPUs across physical CPUs using the MMU. Similarly, virtual I/Os are used to share physical I/Os, such as network interface cards and storage devices, using the IOMMU [29]. Likewise, network-attached storage (NAS) enables efficient storage sharing among multiple clients. These components allow VMs to achieve strong isolation, flexible resource allocation, and robust security, making them the backbone of modern computing infrastructure.

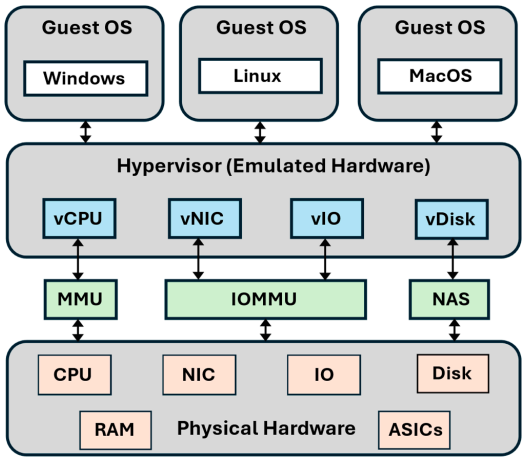

Figure 3. Architecture of Virtual Machine.

2.5. Container Architecture

A containerized system (e.g., Docker) provides a lightweight, isolated application environment through operating-system-level virtualization. Multiple containers can run on the operating system kernel, while the underlying hardware stacks are not partitioned. Figure 4 shows the architecture of a Docker system, which is built on a host-client architecture. The Docker client provides a command-line interface or APIs for interacting with the host. The Docker daemon on the host listens to API requests from the client. The Docker host contains the Docker image, which consists of layers of a read-only, packaged file system containing applications, libraries, and other dependencies. The Docker host can create multiple instances of Docker images called Docker containers, which run on top of the OS. The containers provide an abstraction similar to a process and are managed by a Cgroup through the execution driver API. Cgroups provide a unified interface for process control and resource management in the Linux kernel, regulating CPU, memory, I/O, and network bandwidth. In combination with namespaces, which isolate global resources (e.g., process and user IDs), they ensure containers run independently without interference.

VMs are considered more secure due to their strict partitioning, but they have higher overheads. Containers are faster because they rely on the same kernel and do not partition hardware; however, this also means they have a larger attack vector [30].

 doi:10.20944/preprints202603.0551.v1

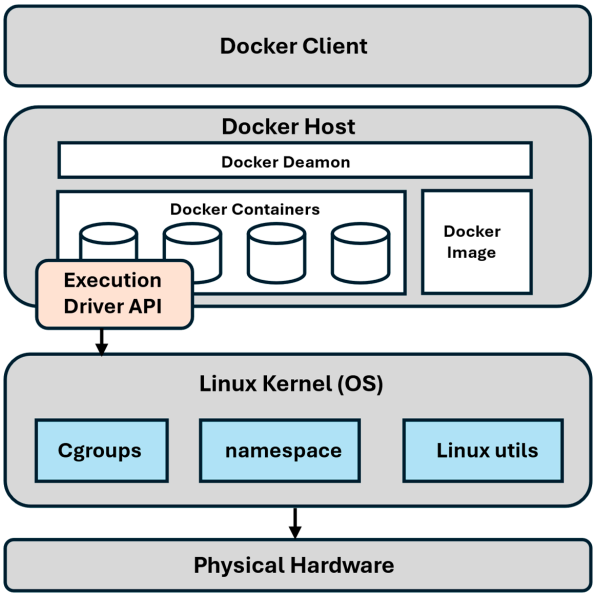

Figure 4. Architecture of Container (e.g., Docker).

2.6. Comparison of Container with a Virtual Machine

While VMs and containers provide isolation for applications running on shared hardware, their approaches and trade-offs differ significantly. In Table 1, we summarize the key differences between VMs and containers across various metrics.

Table 1. Comparison of Virtual Machines with Containers.

Metric	Virtual Machines	Containers (Docker)
Performance	Low	Medium
Security	High	Low
Resource Allocation	Hard Limit	Soft Limit
Deployment	Hard	Easy
Migration	Hard	Easy
Version Control	Hard	Easy

VMs and Docker containers have distinct strengths and weaknesses, and performance depends on the specific use case. VMs partition the entire software and hardware stack, incurring higher overhead but achieving near-native performance once initialized. Docker uses OS-level virtualization and offers faster setup by simplifying tasks such as process control, page table instantiation, namespace configuration, and cgroup configuration.

VMs provide stronger isolation by partitioning the entire system stack, ensuring that distrusting users are not co-located, which enhances security. Docker, on the other hand, shares the kernel space and Docker bridge. Resource allocation in VMs is fixed and difficult to adjust dynamically, whereas Docker allows flexible resource management. Docker images are smaller than VM images, as they do not include the entire hardware and software stack, making them easier to deploy. However, Docker migration can be complex, requiring the transfer of OS states such as process control blocks and page tables, and the technology in this area is still evolving. In contrast, VM migration, which involves transferring memory pages, thread state, and other execution contexts, benefits from mature technologies such as VMware vCenter, which can automate the process.

 doi:10.20944/preprints202603.0551.v1

Finally, Docker supports version control more effectively through its immutable image layers, enabling efficient cloning and change tracking. VMs, in contrast, have less efficient version control because user changes are decoupled from the images.

2.7. Security Challenges in Virtualization

The virtualization stack of the virtual machines and containers provides strong isolation and security guarantees. However, they remain unsafe, and there is substantial evidence of emerging security issues. In this section, we highlight some notable security threats. The hypervisor, which manages co-located VMs, poses a critical security risk itself since it controls CPU scheduling, device access, and memory. Any vulnerability will result in compromise of the co-located VMs or the host [31,32]. Similarly, container escape is a threat arising from vulnerabilities in OS-level virtualization. Any flaws in namespaces, cgroups, or the runtime can be exploited by an adversary to launch attacks or execute code with elevated privileges [33,34]. Moreover, microarchitectural side channels, such as cache timing attacks, branch predictor leaks, and speculative execution vulnerabilities, allow adversaries to infer sensitive information across VM or container boundaries even without direct access [35,36]. The management of VMs, such as context switches in the co-located environment, can result in microarchitectural changes such as TLB invalidation. These microarchitectural traces can enable various attacks, as discussed in the prior works [37–39]. The VMs are also vulnerable to DMA-related attacks, in which an amicable or compromised device can read or modify arbitrary host memory, thereby bypassing IOMMU isolation [40–42]. These challenges highlight that, although virtualization provides a baseline security guarantee, the attack surface is evolving; hence, tools such as HyperShield can enable the design and development of new security solutions and the analysis of overheads, including CPU, core scheduling, address translation, and microarchitectural overheads.

3. Motivation and Prior Art

The cloud environment has become an integral part of modern computing infrastructure, making it essential to guarantee security while also maintaining high performance and energy efficiency. Virtual machines and containers, two of the most widely adopted virtualization technologies with distinct architectural paradigms, significantly influence the performance and effectiveness of deployed security solutions. However, existing methodologies for evaluating these solutions often lack the flexibility and precision necessary to capture detailed performance impacts across diverse workloads and configurations.

Most prior studies have examined VMs and containers in isolation, with limited attention to unified platforms that can analyze the trade-offs between the two. Evaluating advanced and evolving security mechanisms such as deep packet inspection, intrusion detection, and traffic filtering also requires sophisticated tools that integrate seamlessly with virtualized environments. These existing tools lack the modularity necessary to incorporate emerging technologies, and they also fail to provide the level of isolation required to accurately evaluate individual security components.

Firewalls remain a standard security solution in cloud platforms, and significant research has focused on analyzing their performance in both native hardware systems [10,11,43] and virtualized settings [9,44,45]. However, most of the firewalls studied are very basic and lack advanced capabilities. Another study [44] evaluated a firewall implemented as a virtualized network function within the Open Platform for Network Functions Virtualization (OPNFV) on commercial off-the-shelf servers, demonstrating its elastic scalability under varying traffic demands. Similarly, [10] explored the deployment of virtual firewalls in OpenStack, which indicates that placement and configuration decisions have a significant impact on performance. While these studies provide valuable insights, they remain predominantly VM-centric and leave essential gaps in understanding containerized environments such as Docker, where firewall behavior under diverse configurations has not been extensively characterized.

Beyond firewall-specific research, a substantial body of literature compares VMs and containers. Some studies focus exclusively on container performance [12,46], others concentrate on VMs [47],

 doi:10.20944/preprints202603.0551.v1

and several attempt direct comparisons [48,49]. However, most of these efforts rely on high-level performance indicators, which are important but do not apply to the needs of modern security solutions, which require a comprehensive understanding of low-level trade-offs between performance and security.

Our research addresses these gaps through HyperShield, a modular, configurable, and extensible platform designed to evaluate security solutions in virtualized environments. HyperShield enables systematic comparisons between VMs and containers, offering insights into the trade-offs involved when deploying security defenses in cloud infrastructures. It allows detailed analysis of how security modules impact system-level parameters, including CPU utilization, core scheduling, address translation overhead, microarchitectural performance, network throughput, and latency, under various workloads. We evaluate Suricata, a feature-rich user-space security engine and generic packet sniffer module deployable in both user and kernel space. These case studies highlight the performance implications of different design choices, while the framework provides a flexible foundation for testing emerging security features. HyperShield aims to guide research and development of optimized security solutions tailored to the performance and security demands of modern cloud platforms.

4. HyperShield Framework

In this section, we describe different components of HyperShield. HyperShield is designed to enable the realistic performance evaluation of security modules in cloud systems. It supports both container and VM-based platforms, providing a unified and configurable framework for systematic security testing and analysis. The core components of HyperShield are the Client, Server, and Security Module, with detailed configurations for VM and container-based deployments described in sections 4.3 and 4.4.

4.1. HyperShield Design Overview

HyperShield decouples the client, server, and security layers, enabling fine-grained analysis of performance and security trade-offs. The client generates realistic workloads using built-in representative benchmarks for networking, database, and storage, reflecting applications commonly found in cloud systems. These benchmarks can be parameterized for various factors, including packet sizes, query lengths, read/write ratios, and storage demands, ensuring flexibility for different use cases.

The generated traffic traverses the whole system and networking stack on a native Linux OS. It is passed through the security module, which can be deployed in either user space or kernel space. The security module enforces policies, such as encryption, access control, packet filtering, packet inspection, or traffic shaping, before forwarding packets to the server that has the workload endpoints. Each of the client, server, and security modules contains an automated script to extract system and performance metrics, allowing for an understanding of the overheads and bottlenecks in the region of interest. This architecture enables HyperShield to capture a wide range of system-level metrics, including CPU and memory utilization, network throughput, latency, and cache behavior, under various deployment scenarios.

4.2. Security Modules in HyperShield

HyperShield is designed to treat the security layer as a modular and extensible component. In our evaluation, we focus on three case studies:

Suricata: A user-space security module developed by the Open Information Security Foundation (OISF) is widely used in commercial cloud platforms such as Amazon Web Services (AWS). It provides advanced capabilities, including deep packet inspection (DPI), intrusion detection and prevention (IDS/IPS), and application-layer protocol analysis. Suricata highlights the overhead of feature-rich defenses [13].

User space Packet Sniffer: A lightweight module developed as part of HyperShield, deployed in the user space. It intercepts incoming packets from the client and applies security rules before transmitting them to the server. This type of security module is best suited for users or applications

 doi:10.20944/preprints202603.0551.v1

using the cloud infrastructure and requires protection against other users or the rest of the system. Although it operates at a lower privilege level, it is easier to develop, debug, and deploy the security module for the application or the cloud service provider. Depending on the application, it may incur additional overhead due to system calls and context switching.

Kernel space Packet Sniffer: A high privilege security module integrated as a kernel module into the kernel of HyperShield. It operates directly within the kernel networking stack and intercepts packets at the higher privilege level. It enables early enforcement of security policies before they reach user processes. This type of module is primarily implemented by the cloud service provider into the OS or the hypervisor. It offers stronger isolation guarantees and comprehensive visibility into system activity. The kernelspace deployment of a security module may provide lower latency and reduced overhead by avoiding system calls. However, developing, debugging, and maintaining the entire system is more complex.

4.3. Supporting Virtual Machines in HyperShield

Figure 5 shows HyperShield configured for virtual machines. Each component, client, server, and security module, is provisioned as an independent VM connected through a virtual bridge on the host system. The bridge operates as a software-defined switch, enabling inter-VM communication and external connectivity via Network Address Translation (NAT). Traffic from the client VM is directed through the security module using static Address Resolution Protocol (ARP) entries, ensuring inspection occurs before it reaches the server VM. The security module contains a parametrized buffer that can cache incoming packets and apply appropriate security policies before retransmitting them to workload endpoints in the server. Any additional security measure can be easily deployed as kernel objects or executables without altering the base security module, therefore providing flexibility and extensibility.

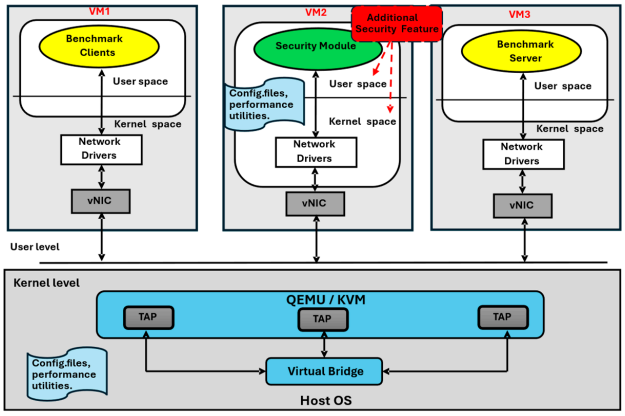

Figure 5. HyperShield configuration for virtual machines.

4.4. Supporting Containers in HyperShield

The configuration of a fully decoupled container deployment is more challenging due to differences between container and VM architectures. To ensure that all client traffic is routed through the security module, HyperShield employs a dual-subnet design. As illustrated in Figure 6, three Docker containers are interconnected via a Docker bridge through virtual Ethernet (veth) pairs. Container 1 hosts benchmark clients, Container 2 deploys security modules, and Container 3 runs the benchmark server. Each container has an isolated user space and its own TCP/IP stack enforced via Linux namespaces, but all containers share the same underlying Linux kernel. The eth0 interface within each container maps to a veth endpoint on the host, which connects to the Docker bridge in the shared kernel space that does packet forwarding. Each container is packaged with configuration scripts and utilities to facilitate workload generation and metric collection, preserving reproducibility while maintaining low overhead.

 doi:10.20944/preprints202603.0551.v1

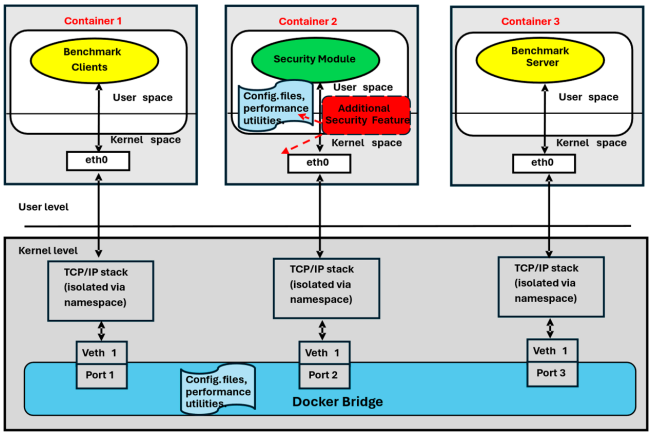

Figure 6. HyperShield configuration for containers.

4.5. Extensibility and Future Research

The modular design of HyperShield enables it to extend beyond the currently studied security module, allowing for the evaluation of new or experimental security primitives. Researchers can incorporate custom modules, explore emerging defenses, or analyze performance under specialized workloads. It provides a realistic execution environment with reproducible experiments, which establishes a foundation for advancing the study of trade-offs between security and performance in cloud systems.

5. Methodology

In this section, we explain our methodology for designing and developing HyperShield.

5.1. Experimental Setup

The HyperShield for VM is built using QEMU/KVM-based virtualization with Ubuntu 22.04 LTS as the guest OS for each of the client, server, and security module VMs. The container uses docker engine 24.0.0 for the client, server, and security module. The host machines that run the HyperShield framework use an 11th Gen Intel Core i9-11900 processor with 16 cores, clocked at 2.50 GHz. The main memory size is 1TB, and the network interface has a bandwidth of 1 Gbps. Table 2 summarizes our hardware and software configurations, as well as the benchmarks used.

Table 2. System Configuration of HyperShield.

Hardware Specifications	
Processor	11th Gen Intel® Core™ i9-11900 @ 2.50GHz × 16
Memory	32 GB DDR4
Storage	1 TB SSD (NVMe)
Network Interface	1 Gbps Ethernet
Software Setup	
Operating System	Host: Ubuntu 20.04 LTS Guest: Ubuntu 22.04 LTS (VMs)
Virtualization	QEMU/KVM with Virtio NICs
Docker Version	Docker Engine 24.0.0, Compose 2.18.1
Security Module	Suricata (user mode), Packet-sniffer (user/kernel)
Benchmarks Used	
Network	Iperf, Netperf, Nuttcp
Database	Redis, MySQL
Storage	FIO

 doi:10.20944/preprints202603.0551.v1

We utilize various Linux-based performance utilities and scripts to gather system statistics, which are open-sourced as part of our HyperShield testbed. The measurements are taken from the security module to quantify the overheads of different security modules. All benchmarks are run for a fixed time to compare the overhead across different configurations.

5.2. Benchmarks

We evaluate system performance with security modules enabled using a suite of benchmarks covering networking, databases, and storage. Networking benchmarks such as Iperf, Netperf, and Nuttcp measure throughput, highlighting how security rules affect bandwidth and packet handling. Redis and MySQL serve as representative benchmarks for databases. Redis captures the latency of in-memory accesses, while MySQL shows query response and transaction performance in client–server environments. For storage testing, FIO simulates diverse read/write patterns to measure latency and assess the security module's impact on disk I/O. These benchmarks provide a comprehensive view of how security modules affect performance across the system stack.

5.2.1. Network Performance

(1) Iperf: Iperf is a widely used network performance testing tool that measures bandwidth between two hosts across a network. It creates TCP or UDP data streams, allowing for the evaluation of throughput under various network conditions. Iperf measures the impact of firewall configurations on data transfer rates between the client and server VMs. It enables detailed analysis of latency and jitter, thereby quantifying how a security module may throttle or process different types of network traffic [50].

(2) Netperf: Netperf is another network benchmarking tool, ideal for measuring various networking aspects, such as unidirectional throughput (TCP and UDP) and request-response performance. In our setup, Netperf can assess the responsiveness of network services across different security module settings, providing insights into how packet filtering rules affect overall performance, including the server's ability to handle network requests under varying loads [51].

(3) Nuttcp: Nuttcp is a network performance measurement tool similar to Iperf but focusing on simplicity and efficiency. It tests TCP and UDP performance between two systems and provides detailed measurements of bandwidth and network latency. It is beneficial for evaluating how security modules affect TCP/UDP performance, enabling you to fine-tune configurations to maximize network throughput and minimize packet loss [52].

5.2.2. Database Performance

(1) Redis: Redis is an in-memory data structure store, often used as a cache, message broker, or database. It is susceptible to latency and network performance, making it an excellent benchmark for assessing the impact of security module rules on application-level services in real-world use cases. By running Redis in the virtualized environment, we can evaluate how well security module configurations can handle the high volume of requests and responses in a high-throughput, low-latency application [53].

(2) MySQL: MySQL is a widely used open-source relational database management system. It is a useful benchmark for evaluating how security module configurations affect database operations, especially in client-server architectures where secure, reliable communication is crucial. Running MySQL queries while testing firewall rules allows us to measure the impact on database response times and transaction throughput, a key performance indicator in a networked environment [54].

5.2.3. Storage Performance

(1) FIO (Flexible I/O Tester): FIO is a flexible I/O benchmarking tool used to assess disk I/O performance. FIO helps evaluate the performance of networked storage systems under different firewall configurations. It simulates various read/write workloads, enabling us to measure latency and

 doi:10.20944/preprints202603.0551.v1

throughput under different conditions, thereby revealing the potential impact of the security module on storage performance [55].

5.3. Evaluation Configuration

Table 3 summarizes the eight configurations (B1–B8) evaluated in our experiments. Each configuration corresponds to either a Docker-based or VM-based environment and is tested under four security settings: no security module, Suricata deployed in user space, a packet sniffer deployed in user space, and a packet sniffer deployed in kernel space.

Table 3. Security configurations in HyperShield evaluation.

Docker		VM	
Label	**Description**	**Label**	**Description**
B1	No Security Module	B5	No Security Module
B2	Suricata (User Space)	B6	Suricata (User Space)
B3	Packet Sniffer (User)	B7	Packet Sniffer (User)
B4	Packet Sniffer (Kernel)	B8	Packet Sniffer (Kernel)

6. Evaluation

This section presents HyperShield characterization results measured from the security module under various configurations, as described in Table 3. Figure 7 presents the performance of networking benchmarks in terms of throughput, and Figure 8 shows the performance of storage and database benchmarks in terms of access latency, evaluated across eight configurations.

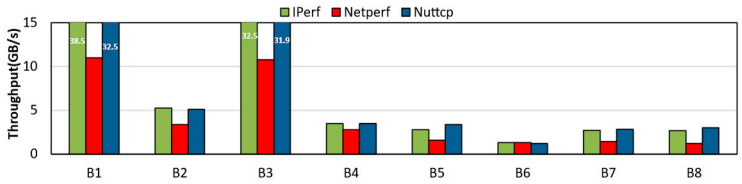

Figure 7. Throughput of network benchmarks (Gbps).

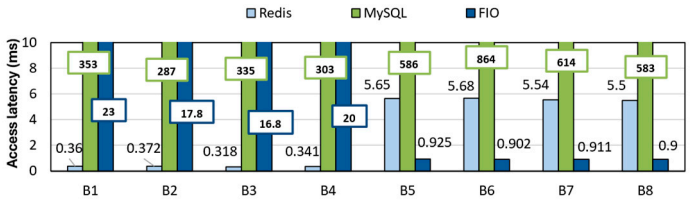

Figure 8. Latency of database and storage benchmarks (ms).

6.1. HyperShield Performance

Figure 7 shows throughput-sensitive networking benchmarks. The average throughput for the VM environment is 1.98 Gb/s, as compared to 9.38 Gb/s for Docker, which is nearly 5× higher. It is essential to note that although the physical NIC supports 1 Gb/s, the system packets never reach the physical NIC when a single host is partitioned into abstracted client, server, and security modules. Instead, these modules communicate through software (virtual) bridges. Therefore, the system throughput is primarily determined by how fast data is copied across these abstracted modules. In a container, a module is just an OS process running on the host. A transaction between the client and the security module involves data transfer between the client process and the security module via inter-process communication (IPC). In a shared-memory abstraction, this would just mean passing an address pointer between processes, which can be significantly faster. On the other hand, a VM provides a strict hardware-level abstraction, so packet transfers between the client VM and the security module VM trigger VM exits and context switches, as the hypervisor manages the transition between the Guest

 doi:10.20944/preprints202603.0551.v1

OS and the host. This creates a CPU-bound bottleneck that significantly degrades VM throughput compared to Docker. Our results demonstrate that Docker's (B1-B4) throughput is greater than that of VMs (B5-B8) because VMs partition the entire software and hardware stack, whereas Docker partitions only the software stack. The throughput of Packet Sniffer in kernel mode in Docker (B4) is similar to that of VM (B8). This is because Docker bridge shares kernel space among the client, server, and security module, thereby creating congestion.

Figure 8 shows the access latency of database and storage benchmarks, which are latency-sensitive. The average access latency of VMs is 14.91 ms, comparable to 12.86ms for containers. MySQL exhibits the highest latency due to SQL parsing, query planning, and transaction overhead, whereas Redis and FIO benefit from direct raw I/O. In storage benchmarks (FIO), VMs outperform Docker because Docker's layered filesystem introduces copy-on-write and traversal overhead, whereas VMs use optimized virtual block devices with near-native I/O performance. The latency for the security module in Docker (B2, B3, B4) is lower than without it, since packets bypass the regular kernel path and instead leverage optimized mechanisms. These modules exploit batching and core-pinning, resulting in reduced measured latency.

In the following section, we analyze the key performance metrics used in our evaluation, including CPU utilization, CPU scheduling behavior, core activity, address-translation overheads, and microarchitectural performance.

6.2. CPU Utilization Metrics

CPU metrics capture the internal behavior of processors across multiple subsystems, including instruction pipelines, CPU frontends, CPU backends, Cache, and TLB structures. The CPU utilization can be measured either in user space or in kernel space; it can also be idle while instructions wait for memory access or I/O. In Figure 9, we plotted the percentage of CPU utilization in the user space across all configurations for the networking benchmark. In Figure 10 we plotted CPU utilization for the remaining benchmarks.

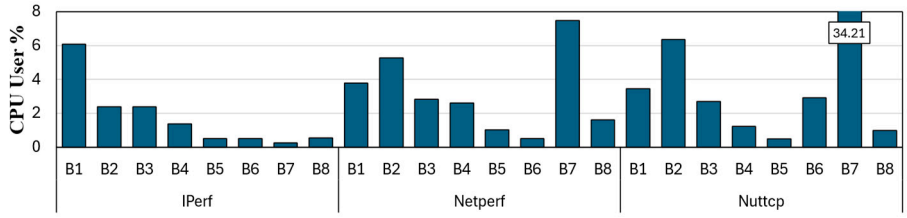

Figure 9. CPU usage in user space (in %) for Network benchmarks.

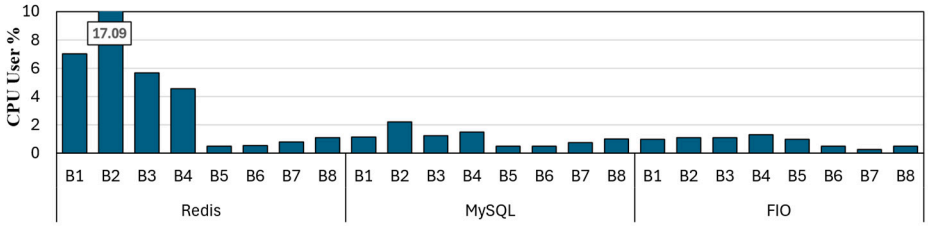

Figure 10. CPU usage in user space (in %) for the remaining benchmarks.

The variation in CPU utilization in user space across benchmarks and models primarily reflects where the bulk of packet processing or request handling occurs, which is either user space or kernel space. Across all benchmarks, CPU utilization in user space is less than 10%, except for 17.09% when running user space Suricata in Docker and Nuttcp running user space packet sniffer in a VM. As expected for networking benchmarks such as Iperf, Netperf, and Nuttcp, the kernel space security modules, such as B4 and B8, exhibit the lowest user space usage because most packet filtering and forwarding occur within the kernel. The user space models B2, B3, B6, and B7 perform more

 doi:10.20944/preprints202603.0551.v1

computation in user space, especially for Nuttcp in B7, which is more CPU-intensive, leading to higher CPU utilization in user space. Suricata B6 and B2 exhibit intermediate or elevated user space utilization on some benchmarks (e.g., Redis) because it performs deep packet inspection and pattern matching in user space. The non-networking benchmarks Redis, MYSQL, and FIO itself dominates in CPU cycles, so the security modules do not add much of an overhead except for Redis in Docker. Redis is single-threaded, and its CPU behaviour is sensitive to the processing speed of the network packets through the security module in Docker. The delay introduced by the security module along the network path causes more work to accumulate in a single execution thread.

The CPU system metric measures the percentage of CPU time spent in kernel space, which indicates system-level overhead such as context switches, I/O handling, and kernel operations. In Figure 11, we plotted the percentage of CPU utilization in kernel mode for networking benchmarks. In Figure 12, we plotted the same for the remaining set of benchmarks. The CPU system utilization reflects how intensively each benchmark interacts with the kernel, including network stack processing, system calls, packet handling, and I/O servicing. In general, networking benchmarks such as IPerf, Netperf, and Nuttcp show higher system usage in baseline configurations B1 and B2. This is mainly because the Linux network stack, socket operations, network system calls, and interrupt handling dominate the execution time. The security module that uses the kernel space security modules e.g., B8 exhibits noticeable spikes in system usage, such as 35.33% for Iperf and 25.41% for netperf. The Kernel space security module B4 in Docker does not show a similar spike because Docker shares the entire kernel space and essentially runs as a user space process.

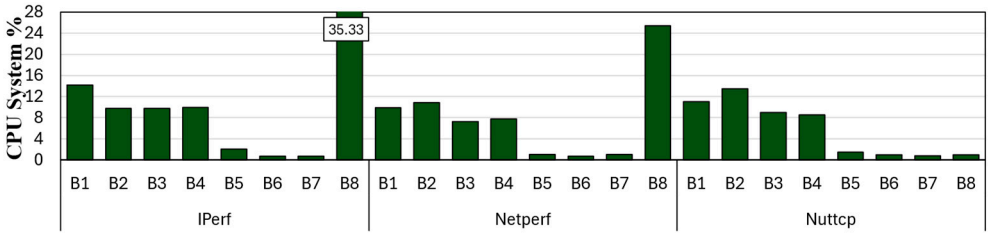

Figure 11. CPU usage in kernel space (in %) for Network benchmarks.

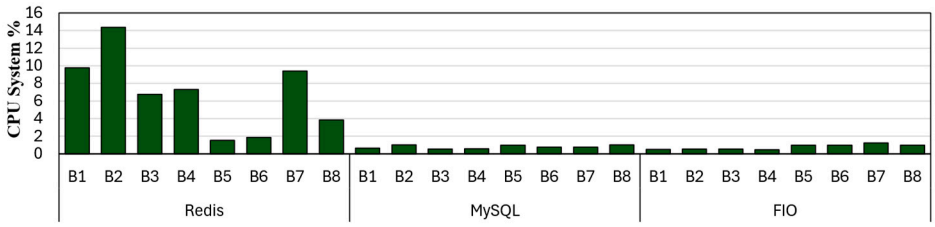

Figure 12. CPU usage in kernel space (in %) for the remaining benchmarks.

The user space models, such as B2, B3, and B7, reduce kernel transitions and therefore exhibit lower system utilization, especially when packet processing or filtering is offloaded to user-level logic. In benchmarks such as Redis and MySQL, system usage is small mainly because these are application centric workloads that mix networking, memory operations, and system calls. Therefore, the system time is primarily influenced by security module related overhead rather than the workload itself. Storage benchmarks like FIO consistently show low system usage because they rely heavily on user space I/O submission with minimal kernel activity. and their model variants exhibit similar behavior, except for minor fluctuations due to differing security instrumentation.

In Figure 13, we plotted the percentage of CPU idle time for networking benchmarks. In Figure 14, we plotted the CPU idle time for the rest of the benchmarks. CPU idle time reflects the amount of CPU processing available while processing each configuration of the security module. The B1 and B2 configurations, which lack a security module, exhibit the highest idle times in most benchmarks. It is

 doi:10.20944/preprints202603.0551.v1

slightly higher for VM in B5 than for Docker in B1. This is because Docker is not fully partitioned, so it is always a running process. The B2 shows a minimum idle time because Suricata is a computationally intensive user space module, which keeps the CPU busy when running as a Docker process. The Redis for B2 drops to 52.16%, demonstrating very high CPU usage. The workloads, such as MySQL and FIO, exhibit consistently high idle percentages greater than 90% across all configurations. This indicates they are less affected by security module overhead or network processing demands.

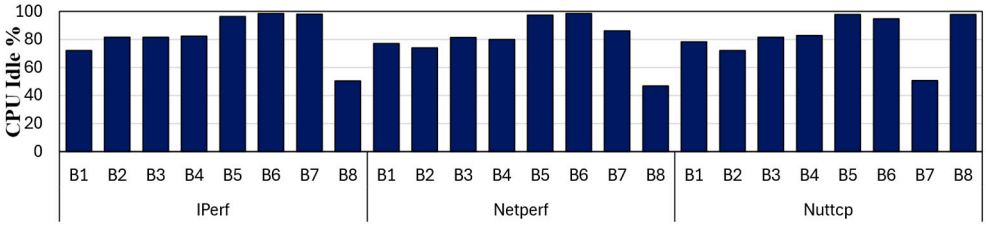

Figure 13. CPU idle (in %) for Network benchmarks.

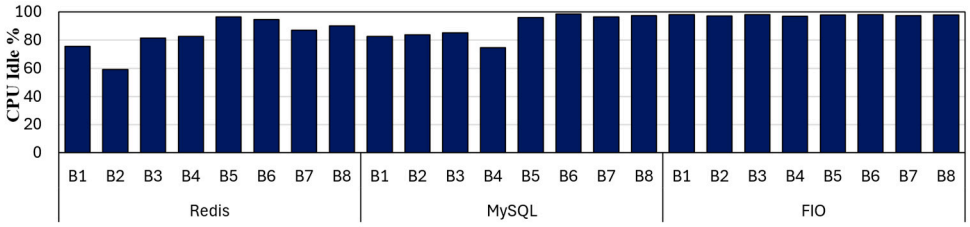

Figure 14. CPU idle (in %) for the remaining benchmarks.

In Figures 15 and 16, we have plotted the breakdown of CPU time into user space usage, kernel space usage, IO wait time, idle time, and other processing delays. As we have discussed extensively regarding most of these breakdowns, IO wait time accounts for a significant share of CPU time in MySQL Docker operations. MySQL shows high I/O wait in Docker because it is a disk-intensive workload with frequent synchronous writes.

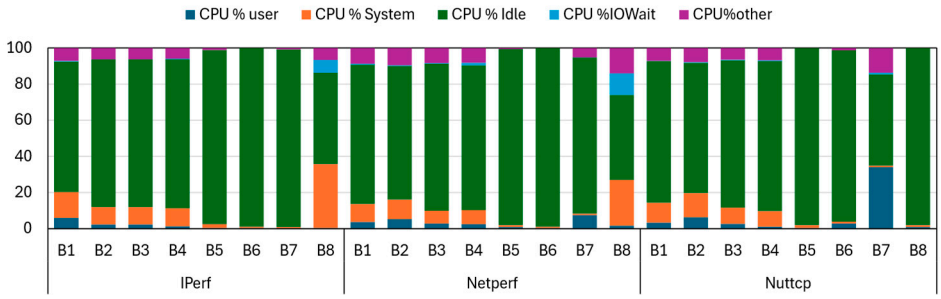

Figure 15. CPU Performance breakdown for the Network benchmarks.

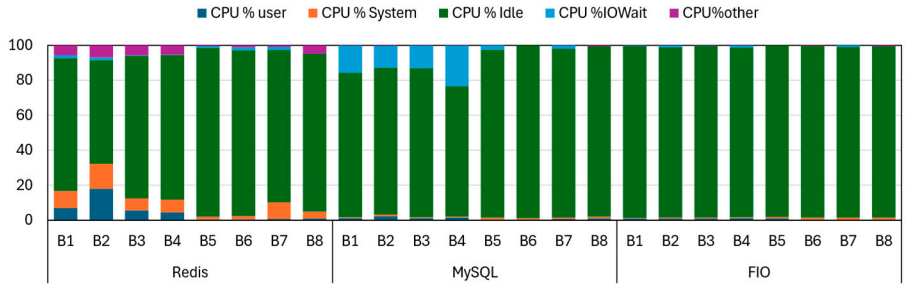

Figure 16. CPU Performance breakdown for the remaining benchmarks.

 doi:10.20944/preprints202603.0551.v1

6.3. CPU Scheduling and Core Activity

In this section, we present the measurements related to CPU Scheduling and Core Activity. We discuss metrics such as TLB shootdown, cPU migration, and context switch, and their impact on virtualization and containerized execution. TLB shootdowns occur when one worker invalidates a translation that must be flushed across all cores that may be caching it locally. It starts with an update to the page table which results in cross-core coordination via inter-processor interrupts (IPIs). CPU migration occurs when a process is rescheduled to a different core, which impacts performance by flushing TLBs, private caches, and core pipeline registers, increasing both user and system overhead. A context switch enables multiple processes to run concurrently on a CPU. This would result in sustained overhead from TLB and cache flushes, and from register spilling and filling. We studied these metrics to gain insight into the performance impacts arising from repeated or concurrent use of the security module across various security domains.

In Figures 17 and 18, we plotted the TLB shootdown rate for networking and the rest of the benchmarks, respectively. The TLB shootdown rate is the number of shootdowns per cycle. The TLB shootdowns are generally low, with the highest being 0.019 for Netperf B5, which does not have any security module running. In general, Docker runs B1 to B4 show slightly higher TLB shootdown activity because containers share the host kernel and therefore participate more directly in page-table updates triggered by the security modules. VMs running B5 to B8 typically exhibit lower shootdown rates, such as B6 for iperf, B6 for netperf, B6 for nuttcp, and B6 for Redis, because guest kernels isolate most page-table changes within the VM and reduce cross-core coordination at the host level. Benchmarks with heavier memory-management activity such as Redis and MySQL shows higher shootdowns in both the environments, reflecting frequent page faults, allocator updates, and updates to shared structures. The kernel space security models B4 and B8 tend to induce slightly higher shootdowns than user space models because they interact more directly with kernel memory structures. In summary, the data indicates that Docker amplifies host-level TLB shootdown events due to the shared kernel model, whereas VMs naturally reduce them through nested or per-VM page table management.

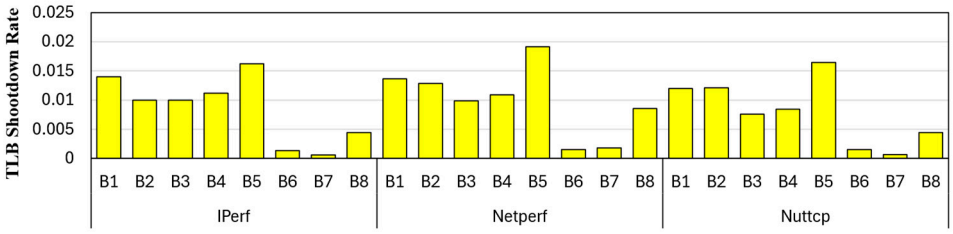

Figure 17. TLB Shootdown Rate for network benchmarks.

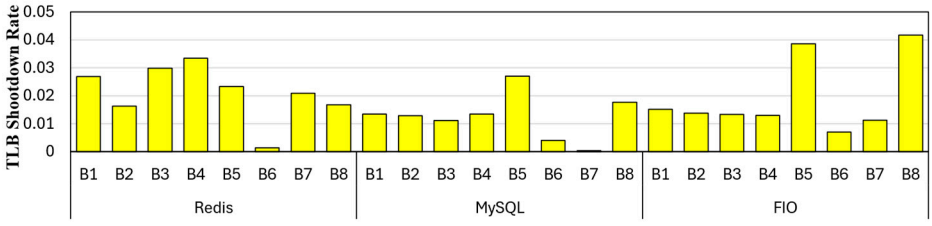

Figure 18. TLB Shootdown Rate for remaining benchmarks.

In Figures 19 and 20, we plotted the context switches per second for networking and the rest of the benchmarks, respectively. Docker measurements do not show context switches because container-level tools did not expose host-level context-switch counts; however, VM runs report measurable context switches consistently, since each VM maintains dedicated kernel threads on the host. The network benchmarks for VM configurations B5 to B8 show modest and stable context-switching between 55 and 355, reflecting lightweight packet-processing loops and efficient scheduling. In contrast, Redis

 doi:10.20944/preprints202603.0551.v1

and MySQL reveal substantially higher context-switch rates in specific VM configurations, especially 6678 for B7 and 24,351 for B8 in Redis. MySQL, on the other hand, shows 2891 for B8, which is mainly driven by multi-threaded request processing, I/O waits, concurrency, lock contention, and internal worker scheduling.

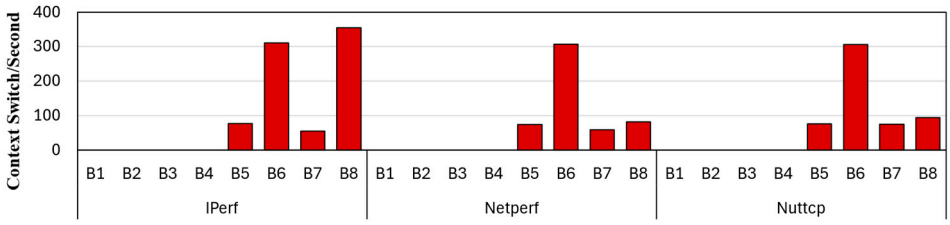

Figure 19. Context Switches per second for network benchmarks.

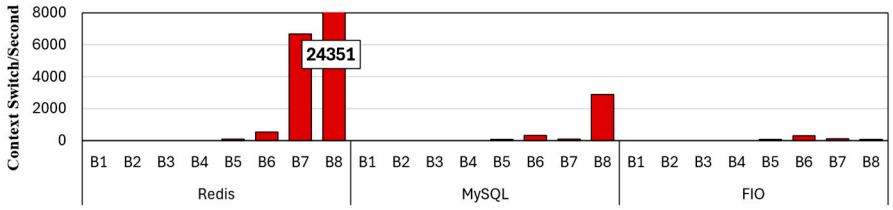

Figure 20. Context Switches per second for remaining benchmarks.

In Figures 21 and 22, we plotted the context switches per second for networking and the rest of the benchmarks, respectively. The CPU migration counts remain zero for the Docker configurations between B1 and B4. This is because Docker containers run as regular Linux processes within the host kernel, which is typically bound to a single core. The Linux Completely Fair Scheduler tends to keep long-running, CPU-intensive processes pinned to their initial core to preserve cache locality and avoid unnecessary TLB and cache flush costs. As a result, Docker workloads exhibit stable core affinity and rarely migrate across cores, leading to the observed zero CPU migration.

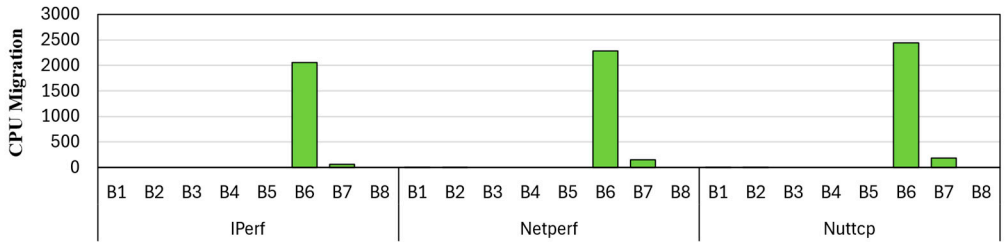

Figure 21. CPU Migration for network benchmarks.

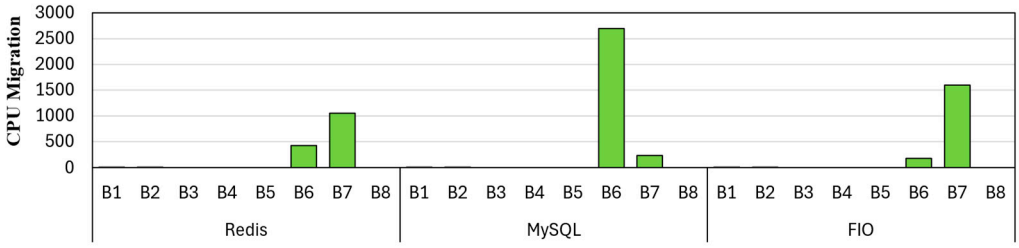

Figure 22. CPU Migration for remaining benchmarks.

In contrast, each vCPU inside a VM behaves like an independent kernel thread from the host's perspective. The host scheduler often migrates these vCPU threads across cores when there is load imbalance, NUMA pressure, I/O activity, interrupts, or background kernel tasks. Suricata (B6) and

 doi:10.20944/preprints202603.0551.v1

the user space model (B7) exhibit severely elevated migration counts, often in the hundreds or even thousands (e.g., 2,061–2,446 for network workloads, 2,695 for MySQL, and 1,597 for FIO). These migrations primarily arise from the VM execution stack: guest scheduling within the VM interacts with host scheduling at the hypervisor level, effectively doubling scheduling pressure and increasing the likelihood of tasks bouncing between cores. This two-level scheduling effect, combined with security-module activity (e.g., packet capture in user space), amplifies CPU-migration frequency. Kernel space model execution (B8) shows near-zero migration, highlighting that bypassing user space processing avoids scheduler churn. Overall, the results indicate that VM-based deployments, especially with user space packet-processing stacks, significantly increase CPU migration, negatively impacting performance through TLB flushes, cache invalidations, and pipeline disruptions.

6.4. Address Translation Overhead

Address translation is a significant overhead in managing a virtualized environment. On one hand, virtual memory provides a simplified programming model, efficient utilization of physical addresses, and strong isolation and security among domains. On the other hand, the overheads are considerable. Each virtual-to-physical address must be translated using structures such as the TLB and multi-level page tables. This process becomes even more complex with the presence of nested page tables. Furthermore, due to memory oversubscription, many virtual pages are not mapped to physical pages, a situation that triggers operating system intervention to allocate pages, update page tables, and reload TLB entries.

In Figures 23 and 24, we plotted the data TLB miss rates per second for networking and the rest of the benchmarks, respectively.

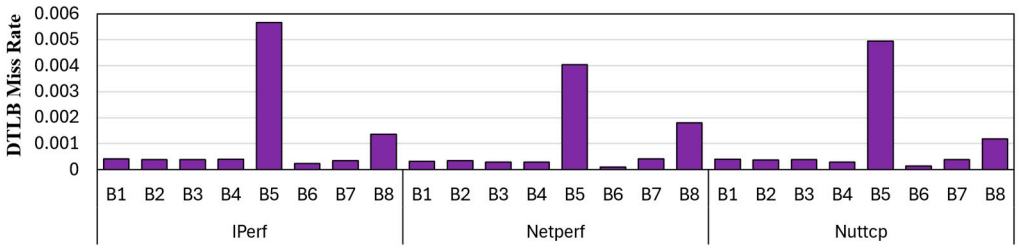

Figure 23. Data TLB Miss Rates for network benchmarks.

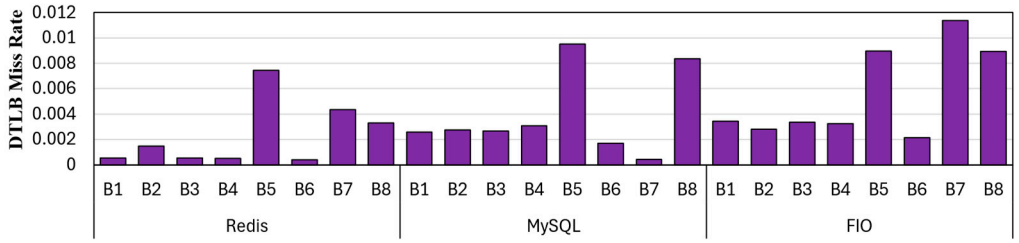

Figure 24. Data TLB Miss Rates for remaining benchmarks.

The TLB miss rate measures how often the system fails to find a virtual-to-physical address translation in the cache. A miss in the TLB results in expensive multiple memory accesses to retrieve the translation from multi-level page tables. Among all configurations, the DTLB miss rate is highest for B5. B5 do not implement a security model that intercepts and scrutinizes network packets, which leads to higher throughput. This increased throughput drives networking structures such as NIC rings and socket buffers, resulting in a greater number of memory accesses and more misses per second in our study. In contrast, Docker relies on the host kernel for networking, introducing additional processing through namespaces and virtual functions. This design reduces packet processing frequency, lowers the memory footprint, and consequently results in lower DTLB miss rates.

 doi:10.20944/preprints202603.0551.v1

In Figure 25 and Figure 26, we present the raw page fault data for TLB misses in networking and the remaining benchmarks, respectively. Page faults occur when a program accesses a memory page that is not currently mapped to physical memory, requiring operating system intervention to establish the mapping between virtual and physical pages. A minor page fault occurs when the physical page is already present in the main memory but has not yet been mapped to the virtual page. In contrast, a major page fault occurs when the page to be mapped resides on disk or was never allocated.

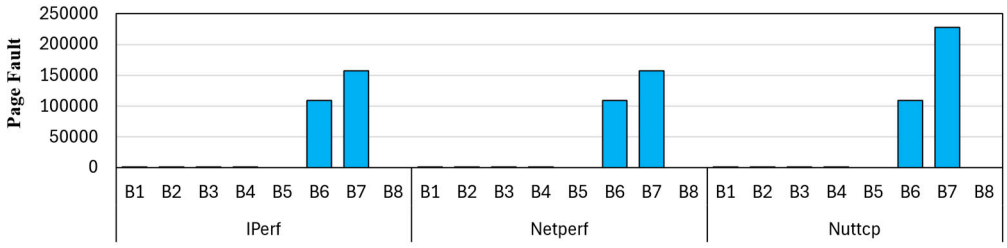

Figure 25. Number of Page Faults for network benchmarks.

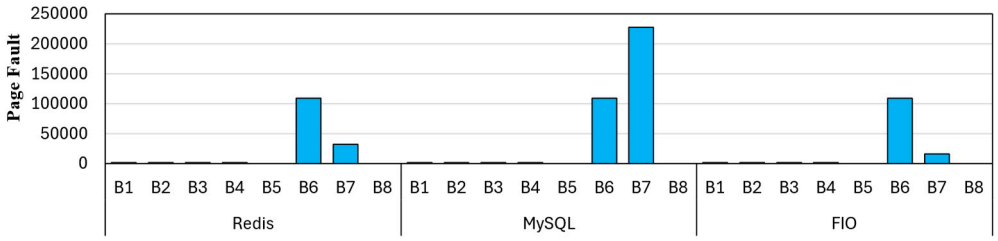

Figure 26. Number of Page Faults for the remaining benchmarks.

Page faults are a major contributor to memory management overheads, particularly in memory and I/O intensive benchmarks. Docker B1 to B4, the number of page faults is minimal due to shared kernel initialization. Since Docker is initialized on top of the host kernel, most page allocations and mappings are completed during the initialization phase. Similarly, VMs show negligible page faults in configurations without a security model, as there are no changes in data transmissions or buffer allocations, resulting in a stable working set. VMs with kernel-mode security models also exhibit minimal page faults, since the guest OS has already faulted and established its working set during boot time. In contrast, VM configurations running user-space Suricata B6 and the user-space security model B7 experience substantially higher page faults, ranging from 30k to 227k. This increase is attributed to large user-space memory allocations, dynamic buffer creation, and security rule initialization.

6.5. CPU Microarchitecture Performance

CPU Microarchitecture Performance provides key insights into how efficiently a workload utilizes the underlying CPU microarchitecture. The Instructions per cycle (IPC) reflects the number of instructions the CPU pipeline retires each cycle. The higher IPC indicates better Instruction-level parallelism (ILP), fewer pipeline stalls, and better use of CPU resources. The cache miss rate indicates how well the cache is designed to capture the memory access pattern of benchmarks.

In Figures 27 and 28, we plotted the IPC of networking and the rest of the benchmarks, respectively. IPC results reveal how security modules affect CPU efficiency in Docker and VM environments. Across all modes, Docker maintains higher IPC than VMs, benefiting from reduced virtualization overhead. The baseline modes without security B1 and B5 generally show high IPC. However, it can be lower due to the streamlining of packets through the optimized security module. The user and kernel packet sniffer in VMs B7 and B8 is significantly lower compared to Docker B3 and B4 due to the high setup overhead in VMs.

 doi:10.20944/preprints202603.0551.v1

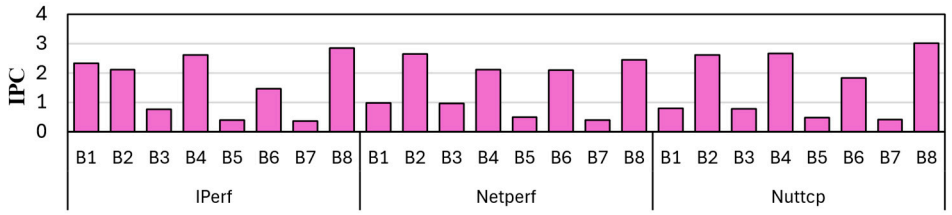

Figure 27. Instructions per cycle (IPC) for network benchmarks.

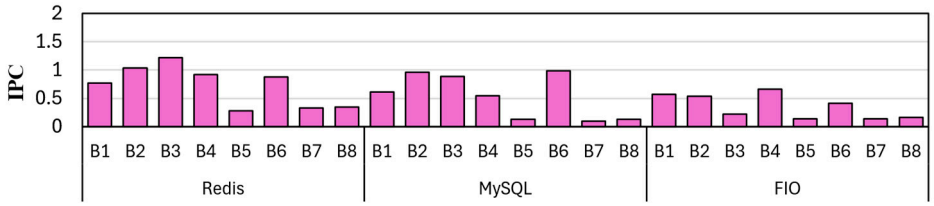

Figure 28. Instructions per cycle (IPC) for remaining benchmarks.

In Figures 29 and 30, we plotted the cache misses of networking and the rest of the benchmarks, respectively. Cache miss behavior across the evaluated modes reflects the trade-offs introduced by different security modules in Docker and VM environments. In general, VMs experience elevated cache stress due to emulation, virtual NICs, and additional address translations, which further exacerbate cache stress. The baseline configurations without any security module (B1 for Docker and B5 for VM) exhibit the lower cache miss rates, except for MySQL and Iperf, due to the batching effect introduced by the security module. Suricata in user space B2 and B6 moderately increases cache misses in most cases. Packet sniffers in user space B3 and B7 further raise cache activity, as frequent context switches and memory copies between kernel and user space stress caches and disrupt locality. Kernel space sniffers B4 and B8 exhibit significant cache miss rates, as their inline security hooks interfere with fast-path optimizations, disable NIC offloads, and introduce irregular memory access patterns.

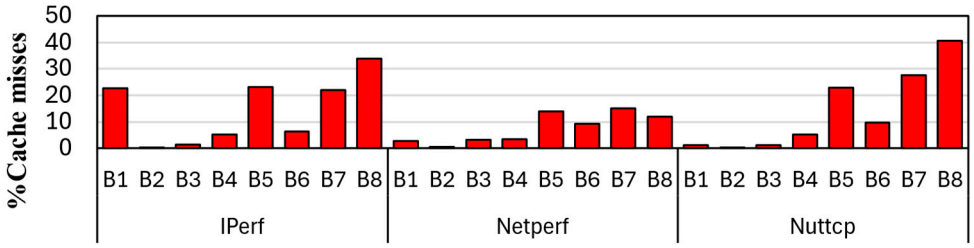

Figure 29. Cache misses (in %) for network benchmarks.

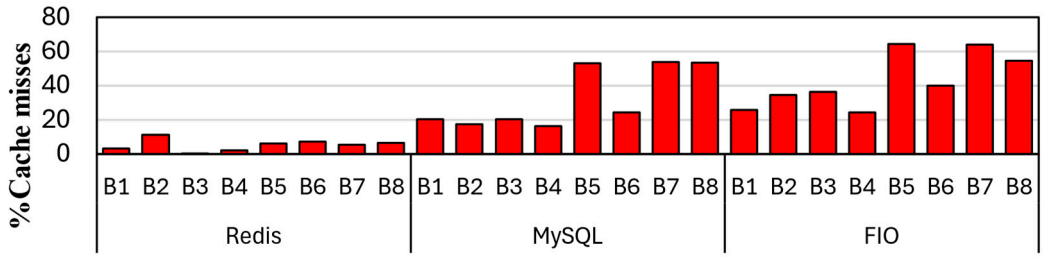

Figure 30. Cache misses (in %) for the remaining benchmarks.

6.6. Network and Interconnect Activity

A Network Interface Card (NIC) is the hardware responsible for sending and receiving packets between the host system and the network. NICs maintain a dedicated transmitter (TX) and receiver

 doi:10.20944/preprints202603.0551.v1

(RX) that buffer packets awaiting transmission or processing. The depth of these queues provides valuable insights into system health. Shallow queues imply smooth packet flow, whereas consistently deep queues may indicate congestion, CPU bottlenecks, or slow packet processing by higher-level software. In Figures 31 and 32, we plotted the average NIC queue size for networking and the rest of the benchmarks, respectively. Docker models B1 to B4 has no queue data because the measurement point is inside the container, where NIC hardware queues are abstracted away. The transmitter path has higher occupancy as compared to the receiver, indicating transmission bound. The packet processing by the security models does not seem to introduce significant bottlenecks. The queue size of user space models B6 and B7 is smaller than that of kernel space models B8 and B5. The application layer tends to dequeue packets in smaller batches but at a higher frequency, which prevents large queues at the NIC. In contrast, kernel space processing is faster per operation but involves additional overhead from repeated transitions between user space to kernel space. These additional steps introduce latency resulting in packet accumulation and hence longer NIC queues.

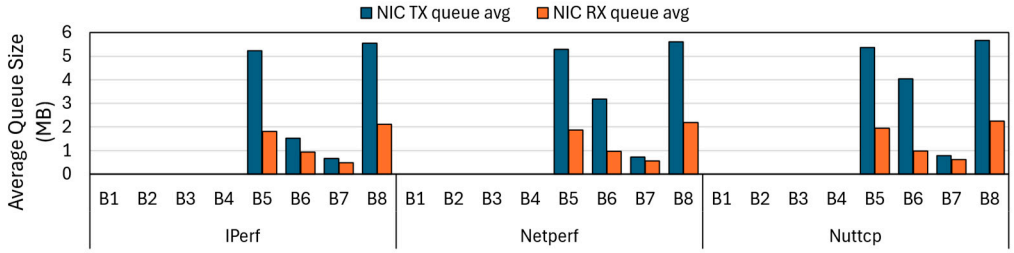

Figure 31. Average Queue size for the network benchmarks.

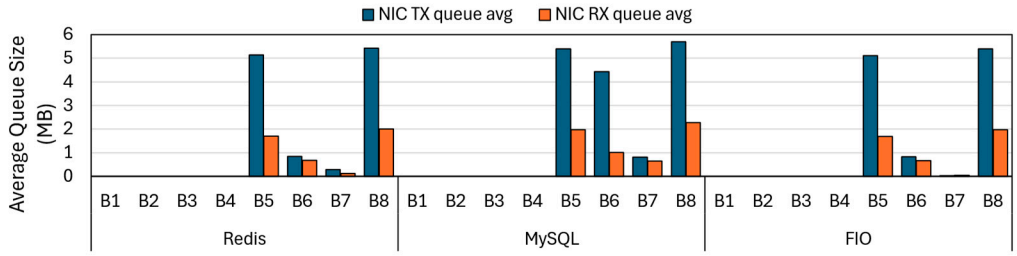

Figure 32. Average Queue size for the remaining benchmarks.

6.7. Energy and Power Overheads

In this section, we describe the system's impact on energy and power. Since energy and power are important considerations in the system performance. The power and energy data have been extracted by leveraging hardware-level telemetry, specifically Intel's Running Average Power Limit (RAPL) counters, to capture cumulative consumption across the Package, Core, and DRAM domains. Figures 33 and 34 describe the energy and power consumption, respectively, for networking benchmarks. Similary Figures 35 and 36 presents the non-networking benchmarks. The energy and power trends align because total energy equals power multiplied by duration. Since the benchmarks were run for fixed intervals, energy numbers are linearly proportional to the average power draw.

B1 and B5 have the lowest energy and power consumption across all benchmarks, clearly indicating that the security module incurs energy and power overheads. The energy and power numbers are mostly constant across the benchmarks. The energy and power consumed by the core, package, and RAM dominate the trend, while the type of benchmarks and the security module have minimal influence, except for MySQL and FIO. The MySQL and FIO benchmarks exhibit significant energy fluctuations primarily due to interaction with the virtualized disk and I/O. Redis, which is an in-memory database, does not show a similar trend as it does not interact with the disk.

 doi:10.20944/preprints202603.0551.v1

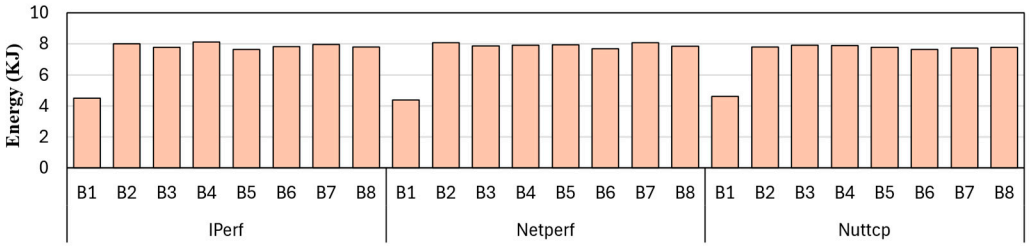

Figure 33. Energy consumption (kJ) for the network benchmarks.

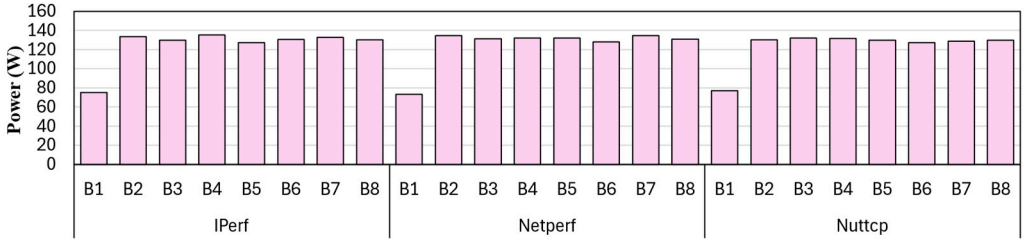

Figure 34. Average power consumption (W) for the network benchmarks.

The networking benchmarks are predominantly interrupt-bound, but storage workloads, such as FIO, incur significant energy and power overhead due to virtualization, as every I/O request triggers a VM exit to the hypervisor for hardware emulation. This results in the VM environment B5 consuming nearly five times the energy of container B1. Docker maintains a low footprint by directly accessing the host's filesystem cache, whereas a VM environment incurs overhead from constant hypervisor interception. The User-space Packet Sniffer (B7) incurs the highest penalty due to the combination of disk-copying overhead with frequent context switching. The lower energy draw in Suricata (B6)indicates that the security modules throttle I/O throughput, reducing total consumption by limiting the frequency of expensive storage operations.

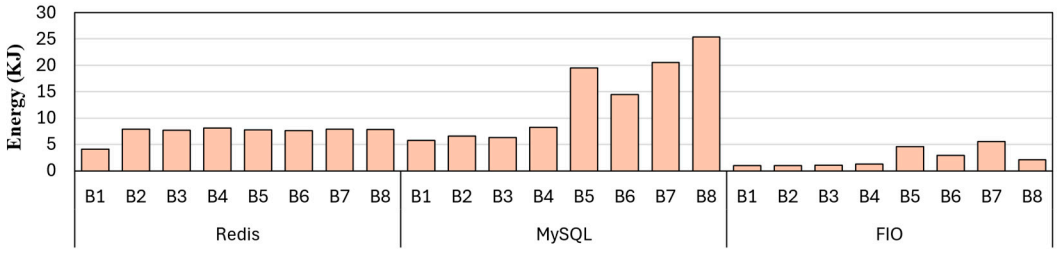

Figure 35. Energy consumption (kJ) for the remaining benchmarks.

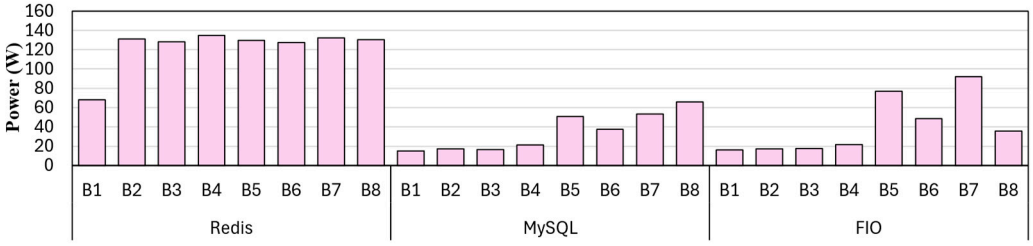

Figure 36. Average power consumption (W) for the remaining benchmarks.

Similarly, MySQL results demonstrate a similar trend due to the virtualization overhead of the disk. The VM incurs a significant penalty due to the combined costs of virtualized network interrupts and disk I/O. The higher energy in B8 (Kernel) compared to B7 (User) is driven by throughput, as the

 doi:10.20944/preprints202603.0551.v1

kernel-space sniffer enables MySQL to process more transactions per second, thereby increasing CPU utilization.

7. Discussion

7.1. Architectural Implications for Multi-Node Scalability

The results presented in this work are based on a single-node environment, where the client, server, and security module run on a single host, connected via virtual bridges and I/O. The systems running on these modules are native Linux, which are generally deployed in cloud infrastructure. However, modern cloud architectures are usually multi-node, with the client, server, and security layers running on different nodes and connected via a distributed fabric. These environments feature a wide variety of compute architectures and network topologies. A common example of such a compute architecture is NUMA (Non-Uniform Memory Access), in which compute cores are connected to distributed memory via high-speed interconnects [56]. These interconnects can be organized in mesh, tree, or fat-tree topologies using east-west or north-south routing algorithms, which are optimized to better serve Quality of Service (QoS), load balancing, and energy efficiency [57,58].

HyperShield enables the study of the overheads of any new security module when deployed either in a container or a VM. Since a security module would generally reside on a given node, the evaluation remains applicable to multi-hop systems. However, in distributed environments, the security module can be deployed across the nodes. It would often encounter cumulative propagation delay and jitter introduced by the network stack and physical switching layers. The framework provides modularity and extensibility, enabling users to tune the architecture to closely match their deployed system. In cases where security modules are present on multiple nodes, such as for stateful packet re-encryption [59], distributed Deep Packet Inspection (DPI) [60], or consensus-based policy enforcement, there would be additional overheads, including inter-node synchronization and packet encapsulation. These complex interactions must be modeled within HyperShield to accurately capture the impact on the system's latency, throughput, and total energy footprint.

7.2. Future Research Directions

The modular design of HyperShield enables it to extend beyond the currently studied security module, allowing for the evaluation of new or experimental security primitives. Researchers can incorporate custom modules, explore emerging defenses, or analyze performance under specialized workloads. The HyperShield can be modified to explore a multi-hop system, where the security module is distributed and can reside on multiple nodes simultaneously. Similarly, HyperShiled can also contain new emerging benchmarks, such as latency-sensitive Large Language Model (LLM) inference or energy-constrained benchmarks for Extended Reality (XR) [61]. It can also be used to deploy emerging cloud security modules, such as Regex and rule-based security Modules, Key-Value (KV) cache side-channel-resistant design, or legacy security modules such as Snort.

8. Conclusion

In this paper, we present HyperShield, an open-source, modular framework for evaluating the performance impact of security modules in cloud infrastructures. We evaluated Suricata and packet sniffers across Docker-based containers and KVM-based VMs, while capturing key performance metrics. Our results show that containers generally outperform VMs in throughput-sensitive workloads due to their lower virtualization overhead. Meanwhile, VMs achieve competitive performance in kernel-space deployments by avoiding Docker's packet congestion due to the shared Docker bridge. In storage benchmarks, VMs further outperform Docker by leveraging optimized virtual block devices that deliver near-native I/O. These findings highlight trade-offs between isolation and efficiency, guiding informed deployment of security modules in modern cloud environments.

 doi:10.20944/preprints202603.0551.v1

Appendix A. Artifact Appendix

Appendix A.1. Abstract

This artifact provides scripts and configuration files to reproduce the experimental results presented in the HyperShield paper. HyperShield is a framework for evaluating security modules in both virtual machines and containers. The artifact includes automated setup scripts for deploying client, server, and security module components, along with comprehensive benchmarking tools for network, database, and storage workloads. The user can reproduce the evaluation results shown in section 6.

Appendix A.2. Artifact Check-List

- **Program**: Setup scripts for VM and container deployments, security modules (Suricata, packet sniffer in user/kernel space), performance measurement utilities
- **Compilation**: GCC 9.4.0 or higher, Make, Linux kernel headers (for kernel module compilation)
- **Run-time environment**:
  - VM setup: Ubuntu 22.04 LTS (guest), Ubuntu 20.04 LTS (host), QEMU/KVM
  - Container setup: Ubuntu 20.04 LTS (host), Docker Engine 24.0.0, Docker Compose 2.18.1
- **Hardware**: 11th Gen Intel Core i9 (or similar), 64 GB RAM (minimum), 1TB storage.
- **Metrics**: Network throughput (Gbps), access latency (ms), CPU utilization (%), IPC, cache miss rate, etc.
- **Output**: Benchmark results, system statistics logs, performance traces etc.
- **How much disk space is required (approximately)?**: 50 GB for VM images, 20 GB for container images and logs
- **How much time is needed to complete experiments (approximately)?**: Less than hour per configuration (8 configurations total: B1-B8)
- **Publicly available?**: Yes

Appendix A.3. Description

Both scripts and configuration files for HyperShield VM and container configurations are available in our public repository.

Appendix A.3.1. How to Access

The artifact can be accessed from the GitHub repository at: https://github.com//faizalam87/HyperShield.

The repository contains:

- `setup/`: Scripts for VM and container setup
- `HyperShield/`: Security module source code (kernel_space.c, user_space_model.c, packet_queue.c/h)
- `Benchmarks/`: Individual benchmark scripts
- `performance/`: Performance profiling utilities
- `README.md`: Comprehensive setup and usage instructions

Appendix A.3.2. Hardware Dependencies

The artifact requires:

- x86-64 processor with virtualization support (Intel VT-x or AMD-V)
- Minimum 64 GB RAM (128 GB recommended)
- 1 TB storage (SSD recommended for VM disk images)
- 1 Gbps Ethernet interface
- Linux kernel 5.4 or higher (for kernel module compilation)

Appendix A.3.3. Software Dependencies

VM-based setup:

 doi:10.20944/preprints202603.0551.v1

- Host OS: Ubuntu 20.04 LTS or later
- Guest OS: Ubuntu 22.04 LTS
- QEMU 4.2 or higher
- KVM enabled
- libvirt (optional, for VM management)

Container-based setup:

- Host OS: Ubuntu 20.04 LTS or later
- Docker Engine 24.0.0 or higher
- Docker Compose 2.18.1 or higher

Common dependencies:

- GCC 9.4.0 or higher
- Linux kernel headers (matching running kernel version)
- Make
- Performance monitoring tools: perf, sysstat, iptables
- Benchmarks: pre-installed iperf3, netperf, nuttcp, redis, mysql, fio.

Appendix A.3.4. Workloads

The artifact includes three categories of benchmarks as described in Section 5. All benchmarks are pre-configured and can be executed via the provided scripts.

Appendix A.4. Installation

Appendix A.4.1. VM-Based Setup

The HyperShield environment comprises a host machine configured with QEMU, a virtual network bridge, and three VMs (Client, Model, Server) for isolation and performance evaluation. The setup includes preparing the host, installing Ubuntu on each VM, and deploying the security modules. The complete step-by-step installation and configuration instructions are available in the project's GitHub repository at https://github.com//faizalam87/HyperShield.

Appendix A.4.2. Container-Based Setup

HyperShield also supports a Docker-based topology comprising Client, Model, and Server containers interconnected via isolated networks to evaluate isolation guarantees. The setup script automatically builds the images, configures networks, and deploys the security modules inside the Model container. The complete build, execution, and troubleshooting steps are available in the project's GitHub repository at https://github.com//faizalam87/HyperShield.

Appendix A.5. Evaluation and expected results

The artifact should reproduce results similar to section 6.

References

1. Alam, F.; Mifthak, M.M.; Purohit, S.; Shadab, M.; Byrd, G.T.; Harfoush, K. VirtShield: A Security Evaluation Framework for Virtualized and Containerized Systems. In Proceedings of the 2026 IEEE 23rd Consumer Communications & Networking Conference (CCNC), 2026, pp. 1–6. https://doi.org/10.1109/CCNC65079.2026.11366513.
2. Susnjara, S.; Smalley, I. What is Infrastructure as a Service? https://www.ibm.com/topics/iaas, 2024. Accessed: 2024-10-13.
3. Microsoft Azure. What is IaaS? https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-iaas, 2024. Accessed: 2024-10-13.
4. Ge, Q.; Yarom, Y.; Cock, D.; Heiser, G. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of cryptographic engineering* **2018**, *8*, 1–27.

 doi:10.20944/preprints202603.0551.v1

5. Szefer, J.; Lee, R.B. A Case for Hardware Protection of Guest VMs from Compromised Hypervisors in Cloud Computing. In Proceedings of the 2011 31st International Conference on Distributed Computing Systems Workshops, 2011, pp. 248–252. https://doi.org/10.1109/ICDCSW.2011.51.
6. Shea, R.; Liu, J. Understanding the impact of Denial of Service attacks on Virtual Machines. In Proceedings of the 2012 IEEE 20th International Workshop on Quality of Service, 2012, pp. 1–9. https://doi.org/10.1109/IWQoS.2012.6245975.
7. Lederer, I.; Mayer, R.; Rauber, A. Identifying Appropriate Intellectual Property Protection Mechanisms for Machine Learning Models: A Systematization of Watermarking, Fingerprinting, Model Access, and Attacks. *IEEE Transactions on Neural Networks and Learning Systems* **2024**, *35*, 13082–13100. https://doi.org/10.1109/TNNLS.2023.3270135.
8. Vahldiek-Oberwagner, A.; Elnikety, E.; Duarte, N.O.; Sammler, M.; Druschel, P.; Garg, D. ERIM: secure, efficient in-process isolation with protection keys (MPK). In Proceedings of the Proceedings of the 28th USENIX Conference on Security Symposium, USA, 2019; SEC'19, p. 1221–1238.
9. Mauricio, L.A.; Rubinstein, M.G.; Duarte, O.C. Proposing and evaluating the performance of a firewall implemented as a virtualized network function. In Proceedings of the 2016 7th International Conference on the Network of the Future (NOF). IEEE, 2016, pp. 1–3.
10. Xia, F.; Hu, J. Application of virtual firewall in computer network security. In Proceedings of the 2020 IEEE Conference on Telecommunications, Optics and Computer Science (TOCS). IEEE, 2020, pp. 42–48.
11. Lyu, M.R.; Lau, L.K. Firewall security: Policies, testing and performance evaluation. In Proceedings of the Proceedings 24th Annual International Computer Software and Applications Conference. COMPSAC2000. IEEE, 2000, pp. 116–121.
12. Struye, J.; Spinnewyn, B.; Spaey, K.; Bonjean, K.; Latré, S. Assessing the value of containers for NFVs: A detailed network performance study. In Proceedings of the 2017 13th International Conference on Network and Service Management (CNSM). IEEE, 2017, pp. 1–7.
13. Foundation), O.O.I.S. Our Story - OISF, 2024. Accessed: 2024-12-10.
14. Vogel, P.; Marongiu, A.; Benini, L. Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU. *IEEE Transactions on Computers* **2019**, *68*, 510–525. https://doi.org/10.1109/TC.2018.2879080.
15. Lv, C.; Zhang, F.; Gao, X.; Zhu, C. LA-vIOMMU: An Efficient Hardware-Software Co-design of IOMMU Virtualization. In Proceedings of the 2022 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), 2022, pp. 246–253. https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom57177.2022.00038.
16. Tian, K.; Dong, Y.; Cowperthwaite, D. A full GPU virtualization solution with mediated pass-through. In Proceedings of the Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USA, 2014; USENIX ATC'14, p. 121–132.
17. Fu, H.C.; Wang, P.H.; Yang, C.L. Active Forwarding: Eliminate IOMMU Address Translation for Accelerator-rich Architectures. In Proceedings of the 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), 2018, pp. 1–6. https://doi.org/10.1109/DAC.2018.8465921.
18. Ausavarungnirun, R.; Landgraf, J.; Miller, V.; Ghose, S.; Gandhi, J.; Rossbach, C.J.; Mutlu, O. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In Proceedings of the 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2017, pp. 136–150.
19. Lee, J.; Lee, J.M.; Oh, Y.; Song, W.J.; Ro, W.W. SnakeByte: A TLB Design with Adaptive and Recursive Page Merging in GPUs. In Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2023, pp. 1195–1207. https://doi.org/10.1109/HPCA56546.2023.10071063.
20. Oracle Corporation. Nested Paging and VPIDs — Oracle VM VirtualBox Administrator's Guide (Release 6.0). https://docs.oracle.com/en/virtualization/virtualbox/6.0/admin/nestedpaging.html, 2020. Accessed: 2025-12-10.
21. Intel Corporation. What Intel® Xeon® Scalable Processors Support Virtualization Features? (Article ID: 000058162). https://www.intel.com/content/www/us/en/support/articles/000058162.html, 2025. Accessed: 2025-12-10.
22. Hoang, G.; Bae, C.; Lange, J.; Zhang, L.; Dinda, P.; Joseph, R. A Case for Alternative Nested Paging Models for Virtualized Systems. *IEEE Computer Architecture Letters* **2010**, *9*, 17–20. https://doi.org/10.1109/L-CA.2010.6.
23. Stojkovic, J.; Skarlatos, D.; Kokolis, A.; Xu, T.; Torrellas, J. Parallel virtualized memory translation with nested elastic cuckoo page tables. In Proceedings of the Proceedings of the 27th ACM International Conference

 doi:10.20944/preprints202603.0551.v1

on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA, 2022; ASPLOS '22, p. 84–97. https://doi.org/10.1145/3503222.3507720.

24. Zhang, J.; Jia, W.; Chai, S.; Liu, P.; Kim, J.; Xu, T. Direct Memory Translation for Virtualized Clouds. In Proceedings of the Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, New York, NY, USA, 2024; ASPLOS '24, p. 287–304. https://doi.org/10.1145/3620665.3640358.
25. Peleg, O.; Morrison, A.; Serebrin, B.; Tsafrir, D. Utilizing the IOMMU Scalably. In Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 15), Santa Clara, CA, 2015; pp. 549–562.
26. Morgan, B.; Alata, É.; Nicomette, V.; Kaâniche, M. Bypassing IOMMU Protection against I/O Attacks. In Proceedings of the 2016 Seventh Latin-American Symposium on Dependable Computing (LADC), 2016, pp. 145–150. https://doi.org/10.1109/LADC.2016.31.
27. Amit, N.; Ben-Yehuda, M.; Research, I.; Tsafrir, D.; Schuster, A. vIOMMU: Efficient IOMMU Emulation. In Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC 11), Portland, OR, 2011.
28. Cloud, A. I/O Virtualization in Cloud Computing, 2024. Accessed: 2024-09-19.
29. Alam, F.; Lee, H.; Bhattacharjee, A.; Awad, A. CryptoMMU: Enabling Scalable and Secure Access Control of Third-Party Accelerators. In Proceedings of the 2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO), 2023, pp. 32–48.
30. Sultan, S.; Ahmad, I.; Dimitriou, T. Container Security: Issues, Challenges, and the Road Ahead. *IEEE Access* **2019**, *7*, 52976–52996. https://doi.org/10.1109/ACCESS.2019.2911732.
31. Gkortzis, A.; Rizou, S.; Spinellis, D. An Empirical Analysis of Vulnerabilities in Virtualization Technologies. In Proceedings of the 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2016, pp. 533–538. https://doi.org/10.1109/CloudCom.2016.0093.
32. Ghoshal, D.; Canon, R.S.; Ramakrishnan, L. I/O performance of virtualized cloud environments. In Proceedings of the Proceedings of the Second International Workshop on Data Intensive Computing in the Clouds, New York, NY, USA, 2011; DataCloud-SC '11, p. 71–80. https://doi.org/10.1145/2087522.2087535.
33. Mattetti, M.; Shulman-Peleg, A.; Allouche, Y.; Corradi, A.; Dolev, S.; Foschini, L. Securing the infrastructure and the workloads of linux containers. In Proceedings of the 2015 IEEE Conference on Communications and Network Security (CNS), 2015, pp. 559–567. https://doi.org/10.1109/CNS.2015.7346869.
34. Liu, P.; Ji, S.; Fu, L.; Lu, K.; Zhang, X.; Lee, W.H.; Lu, T.; Chen, W.; Beyah, R. Understanding the Security Risks of Docker Hub. In Proceedings of the Computer Security – ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part I, Berlin, Heidelberg, 2020; p. 257–276. https://doi.org/10.1007/978-3-030-58951-6_13.
35. Kocher, P.; Horn, J.; Fogh, A.; Genkin, D.; Gruss, D.; Haas, W.; Hamburg, M.; Lipp, M.; Mangard, S.; Prescher, T.; et al. Spectre Attacks: Exploiting Speculative Execution. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), 2019, pp. 1–19. https://doi.org/10.1109/SP.2019.00002.
36. Liu, F.; Yarom, Y.; Ge, Q.; Heiser, G.; Lee, R.B. Last-Level Cache Side-Channel Attacks are Practical. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, 2015, pp. 605–622. https://doi.org/10.1109/SP.2015.43.
37. Tatar, A.; Trujillo, D.; Giuffrida, C.; Bos, H. TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, 2022; pp. 989–1007.
38. Kim, T.; Park, H.; Lee, S.; Shin, S.; Hur, J.; Shin, Y. DevIOus: Device-Driven Side-Channel Attacks on the IOMMU. In Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP), 2023, pp. 2288–2305. https://doi.org/10.1109/SP46215.2023.10179283.
39. Gras, B.; Razavi, K.; Bos, H.; Giuffrida, C. Translation Leak-aside Buffer: Defeating Cache Side-Channel Protections with TLB Attacks. In Proceedings of the Proceedings of the 27th USENIX Conference on Security Symposium, USA, 2018; SEC'18, p. 955–972.
40. Balogh, S.; Mydlo, M. New possibilities for memory acquisition by enabling DMA using network card. In Proceedings of the 2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2013, Vol. 02, pp. 635–639. https://doi.org/10.1109/IDAACS.2013.6663002.
41. Rani, A.; Pai, A.; Naware, B.; Yang, Z.H.; Huang, T.Y. Direct Memory Access Remapping for Thunderbolt, Feature Deployment at Platform Level. In Proceedings of the 2020 IEEE International Conference for Innovation in Technology (INOCON), 2020, pp. 1–5. https://doi.org/10.1109/INOCON50539.2020.9298289.
42. Sang, F.L.; Nicomette, V.; Deswarte, Y. I/O Attacks in Intel PC-based Architectures and Countermeasures. In Proceedings of the 2011 First SysSec Workshop, 2011, pp. 19–26. https://doi.org/10.1109/SysSec.2011.10.

 doi:10.20944/preprints202603.0551.v1

43. Gouda, M.G.; Liu, A.X. A model of stateful firewalls and its properties. In Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05). IEEE, 2005, pp. 128–137.
44. Waheed, M.S.; Al Mufarrej, M.; Sobhieh, M.; Al Barrak, A.; Baig, A.; Al Mazyad, A. Implementation of virtual firewall function in SDN (software defined networks). In Proceedings of the 2017 9th IEEE-GCC Conference and Exhibition (GCCCE). IEEE, 2017, pp. 1–9.
45. Mauricio, L.A.F.; Rubinstein, M.G.; Duarte, O.C.M.B. Proposing and evaluating the performance of a firewall implemented as a virtualized network function. In Proceedings of the 2016 7th International Conference on the Network of the Future (NOF), 2016, pp. 1–3. https://doi.org/10.1109/NOF.2016.7810127.
46. Suo, K.; Zhao, Y.; Chen, W.; Rao, J. An analysis and empirical study of container networks. In Proceedings of the IEEE INFOCOM 2018-IEEE Conference On Computer Communications. IEEE, 2018, pp. 189–197.
47. Piao, J.T.; Yan, J. A network-aware virtual machine placement and migration approach in cloud computing. In Proceedings of the 2010 Ninth International Conference on Grid and Cloud Computing. IEEE, 2010, pp. 87–92.
48. Li, W.; Kanso, A. Comparing containers versus virtual machines for achieving high availability. In Proceedings of the 2015 IEEE International Conference on Cloud Engineering. IEEE, 2015, pp. 353–358.
49. Zhang, Q.; Liu, L.; Pu, C.; Dou, Q.; Wu, L.; Zhou, W. A comparative study of containers and virtual machines in big data environment. In Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). IEEE, 2018, pp. 178–185.
50. Tirumala, A.; Qin, F.; Dugan, J.; Ferguson, J.; Gibbs, K. Iperf. http://dast.nlanr.net/Projects/Iperf/, 2005. Accessed: 2024-10-13.
51. Jones, R. Netperf 2.4.3. http://www.netperf.org/netperf/, 2003. Accessed: 2024-10-13.
52. Developers, N. Nuttcp - A Network Performance Measurement Tool, 2024. Accessed: 2024-12-10.
53. Labs, R. *Redis Benchmarking Guide*, 2024. Accessed: 2024-10-13.
54. Harris, J. *MySQL Performance*, 2008. Accessed: 2024-10-13.
55. Axboe, J. *fio - Flexible I/O tester*, 2024. Accessed: 2024-10-13.
56. Liu, M.; Li, T. Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads. In Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), 2014, pp. 325–336. https://doi.org/10.1109/ISCA.2014.6853224.
57. Ge, Y.; Zhu, Q. Trust Threshold Policy for Explainable and Adaptive Zero-Trust Defense in Enterprise Networks. In Proceedings of the 2022 IEEE Conference on Communications and Network Security (CNS), 2022, pp. 359–364. https://doi.org/10.1109/CNS56114.2022.9947263.
58. Syed, N.F.; Shah, S.W.; Shaghaghi, A.; Anwar, A.; Baig, Z.; Doss, R. Zero Trust Architecture (ZTA): A Comprehensive Survey. *IEEE Access* **2022**, *10*, 57143–57179. https://doi.org/10.1109/ACCESS.2022.3174679.
59. Xu, Q.; Miano, S.; Gao, X.; Wang, T.; Murugadass, A.; Zhang, S.; Sivaraman, A.; Antichi, G.; Narayana, S. State-compute replication: parallelizing high-speed stateful packet processing. In Proceedings of the Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation, USA, 2025; NSDI '25.
60. Biyouki, A.; Lotfipour, S.; Haghi, B. An enhanced deep learning framework for intrusion classification enterprise network using multi-branch CNN-attention architecture. *Scientific Reports* **2025**, *16*. https://doi.org/10.1038/s41598-025-34166-1.
61. Lee, E.S.; Shin, B.S. Enhancing the Performance of XR Environments Using Fog and Cloud Computing. *Applied Sciences* **2023**, *13*, 12477. https://doi.org/10.3390/app132212477.