A brief overview of bare-metal hypervisors

Marek Moravcik
Faculty of Management Science and Informatics
University of Zilina
Zilina, Slovakia
Marek Moravcik@fri.uniza.sk

Martin Kontsek
Faculty of Management Science and Informatics
University of Zilina
Zilina, Slovakia
martin.kontsek@fri.uniza.sk

Pavel Segec
Faculty of Management Science and Informatics
University of Zilina
Zilina, Slovakia
pavel.segec@fri.uniza.sk

Lubica Zidekova
Faculty of Management Science and Informatics
University of Zilina
Zilina, Slovakia
lubica.zidekova@fri.uniza.sk

***Abstract*—Virtualization technologies have become a fundamental component of modern computing infrastructures, enabling efficient resource utilization, scalability, and workload isolation. Hypervisors play a critical role in this context by managing virtual machines and providing abstraction between hardware and operating systems. This paper presents an overview of hypervisor architectures with particular attention to security-relevant aspects such as privilege rings, the trusted code base, and the distinction between monolithic and microkernel architectures. It then provides a descriptive comparison of three widely used hypervisors, VMware ESXi, KVM (Kernel-based Virtual Machine), and Xen, with respect to their architecture, virtualization techniques, resource and I/O management, and security features. Finally, the main advantages and disadvantages of the compared hypervisors are summarized to support the selection of a suitable platform.**

***Index Terms*—Hypervisors, VMware ESXi, KVM, Xen**

I. Introduction

Today, virtualization forms the natural basis of modern computing and cloud infrastructures. It enables efficient use of physical resources, simplifies environment management and creates space for flexible deployment of various systems and services.

Today, virtualization forms the natural basis of modern computing and cloud infrastructures, enabling efficient use of physical resources and flexible deployment of various systems and services. The key component of every virtualization infrastructure is the hypervisor, which abstracts the physical hardware and mediates the access of virtual machines to the processor, memory and input/output devices.

Hypervisors differ not only in their position relative to the hardware (Type 1 and Type 2), but also in the virtualization technique they use and in the internal organization of their subsystems. In particular, the distinction between monolithic and microkernel architectures determines which components run at the highest privileged level, and therefore how large the trusted code base and the attack surface of the platform are [5].

Existing comparisons of VMware ESXi, KVM and Xen focus mainly on experimental performance evaluation [1], [2], [6], while their architectural concepts are discussed in more general surveys [3], [4], [7]. The aim of this paper is to provide a concise overview of hypervisor architectures and a descriptive comparison of these three hypervisors with respect to their architecture, virtualization techniques, resource and I/O management, and security-relevant properties.

The rest of the paper is organized as follows. Section II explains the classification of hypervisors and their architectural aspects from a security perspective. Sections III, IV and V describe VMware ESXi, KVM and Xen, respectively. Section VI concludes the paper and summarizes the compared hypervisors in Table I

II. Hypervisors as the basis of virtualization infrastructure

A hypervisor is the basic layer of virtualization infrastructure that allows virtual machines to be created and run on top of physical hardware. Its role is to mediate access of virtual machines to the processor, memory, and input/output devices so that multiple separate virtual environments can operate simultaneously on a single physical system. The hypervisor thus abstracts physical resources and creates a basis for their controlled sharing between individual virtual machines.

From the perspective of virtualization system architecture, not only the existence of a hypervisor is important, but also the method of its implementation. The differences between hypervisors are reflected in their location relative to the hardware, in the method of virtualization implementation, and in the degree of integration of individual support mechanisms. These properties subsequently affect the performance, isolation of virtual machines, and the security characteristics of the entire environment [1], [2].

979-8-3195-3729-4/26/$31.00 ©2026 IEEE

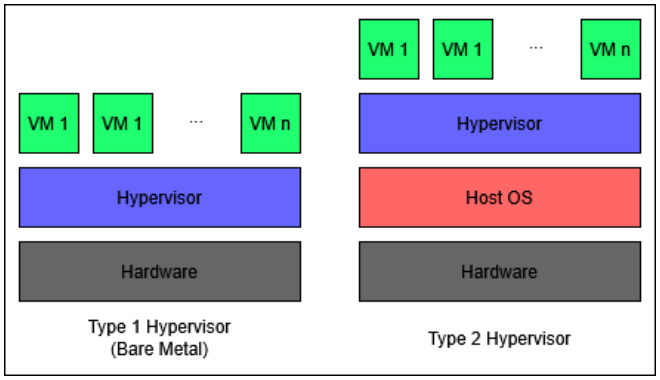


Fig. 1. Comparison of Type 1 and Type 2 hypervisors

A. Types of hypervisors and their implementation models

The differences between hypervisors cannot be reduced only to their functional properties, but primarily to the way in which they are integrated into the computing environment. From an architectural point of view, it is crucial where the virtualization layer is located in relation to the physical hardware and what extent of privileged code it executes [3], [4].

The traditional division of hypervisors is based on their location in the system. Hypervisors referred to as Type 1 (bare-metal) are implemented directly on top of the physical hardware and represent the lowest software layer of the system. They do not require a host operating system and themselves provide processor scheduling, memory management and mediation of input-output operations. This model is used mainly where a high level of isolation and efficient resource management is required.

On the other hand, Type 2 hypervisors are implemented as applications running on top of an existing operating system. In this case, the virtualization layer is dependent on the host kernel, which provides communication with the hardware. This model simplifies implementation and is suitable for development or testing environments, but from an architectural perspective it introduces a wider range of code running in privileged mode and a greater dependency on the host system. In the Figure 1, there is comparison of Type 1 and Type 2 hypervisors.

However, such a binary division is not sufficient to understand the implementation differences between individual platforms. In addition to the location of the virtualization layer, the method of implementing the virtualization itself is also crucial. Historically, full virtualization techniques were used, in which the hypervisor captured and emulated privileged instructions, as well as paravirtualization, which involved modifying the guest operating system so that it could interact with the hypervisor through specific interfaces.

Modern implementations use hardware support for processor virtualization, which minimizes the need for emulation and reduces performance overhead.

From an architectural perspective, the method of implementing virtualization has a direct impact on the extent of the trusted code base (TCB) and the structure of the privileged layers of the system. Models that minimize code running in the most privileged mode of the processor reduce the potential attack surface and simplify system security analysis.

In contrast, centralized or monolithic implementations may include extensive device management and network stack subsystems directly in the hypervisor, which increases complexity.

From the perspective of designing virtualization environments, it is therefore appropriate to evaluate hypervisors not only according to their Type 1 or Type 2 classification, but also according to their implementation model, the way in which the processor and memory are virtualized, as well as the degree of integration with the virtualization environment's control mechanisms.

B. Architectural aspects of hypervisors from a security perspective

The architecture of the hypervisor significantly affects the security properties of the virtualization environment. When analyzing it, it is necessary to take into account not only the method of deploying the virtualization layer, but also the organization of privileged processor modes, the extent of the trusted computing base, and the internal division of the hypervisor subsystems.

1) Security Rings and privileged modes: One of the basic elements of the hypervisor architecture is the distribution of components between individual security levels called security rings. A security ring represents a hierarchical level of authorization within the system. The levels are arranged from the highest (Ring 0) to the lowest (Ring 3), with each of them determining the scope of access to system resources [5].

Ring 0 is the innermost and most privileged level. Traditionally, it is reserved for the operating system kernel, which directly communicates with the processor, memory, and other hardware components. In the context of virtualization, this level is occupied by the hypervisor, which controls the access of virtual machines to physical resources.

Ring 1 and, if used, Ring 2 represent modes with higher privileges than user mode, but lower than the system kernel. These levels may contain privileged subsystems, such as device drivers or virtual machine kernel mechanisms. Within the virtualization model, Ring 1 is used to separate the kernel space of a virtual machine from the hypervisor itself.

Ring 3 is known as user mode. This is the least privileged level in which application code executes. Programs in this mode do not have direct access to hardware or system resources, and their operations are mediated through higher privilege levels.

Security rings restrict the capabilities of executing code by controlling access to memory and shared resources. Each memory segment is assigned a privilege level, and a gate mechanism ensures that less privileged code cannot directly access resources intended for higher levels. Switching between levels requires a context switch, which affects system performance. However, from a security perspective, this mechanism represents a protective barrier - an error or compromise at

a lower level has less impact on system stability than a disruption at the highest privileged layer.

Minimizing the amount of software executing in Ring 0 is therefore a fundamental security goal. The fewer components that have direct access to the system core, the less risk there is of destabilizing or compromising the entire virtualization platform.

2) Trusted Code Base: Every virtualization system contains a layer that must be trusted unconditionally. This is the part of the architecture that ensures the isolation of virtual machines, the enforcement of security rules, and the control over access to physical resources. This layer is called the Trusted Code Base (TCB) [5].

The TCB is not just the hypervisor itself as a program module. It includes all the components that contribute to ensuring the basic functionality and protection of the system. In the case of a bare-metal hypervisor, this set includes elements that provide access to hardware, device drivers, control mechanisms, and software layers that mediate resource abstraction. If any of these elements fail or are compromised, the isolation of virtual machines can be violated.

The term Trusted Code Base is taken from the field of information systems security assessment, where it refers to all parts of the system responsible for enforcing security policy and maintaining the isolation of objects on which protection is based. The scope of the TCB thus determines the boundary beyond which the system can no longer be considered securely isolated.

From a hypervisor design perspective, it is crucial that this trusted layer contains only those components that are essential to maintaining the security and proper functioning of the system. Each additional element placed in the TCB increases the complexity of the architecture and expands the set of code that must be considered trustworthy. Therefore, minimizing the Trusted Code Base is considered a fundamental principle of secure design of virtualization platforms.

3) Monolithic and microkernel hypervisor architecture: The difference between hypervisors is not only that they mediate hardware virtualization, but also in the way their internal subsystems are organized. From an architectural point of view, it is crucial which mechanisms remain part of the hypervisor itself and which are moved to separate support layers. It is this division that affects the scope of privileged code, the size of the trusted code base, and the resulting attack surface of the virtualization platform. From this perspective, hypervisors are often described as monolithic or microkernel architectures. In the monolithic model, the hypervisor kernel, interfaces, drivers, and virtualization stack are concentrated into a single entity in the highest privileged layer, while in the microkernel model, only the kernel remains in the hypervisor and other support mechanisms are moved to a separate control partition [5]. In the Figure 2, there is comparison of Monolithic and microkernel hypervisor architecture.

4) Micro-kernel architecture: The microkernel architecture is designed to minimize the amount of code placed directly in the hypervisor space. Only the hypervisor kernel remains at

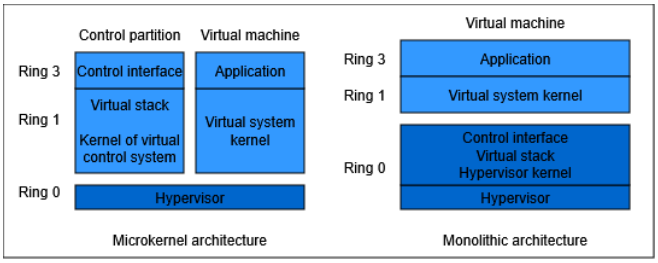


Fig. 2. Comparison of Monolithic and microkernel hypervisor architecture

the highest privileged level, which is responsible for isolating virtual machines and controlling access to physical resources. Support mechanisms, such as handling network communication, storage, hypercalls, and administrative interfaces, are moved to a separate management partition, which contains its own operating environment and is considered part of the Trusted Code Base (TCB).

Such a division means that part of the trusted code base is located outside Ring 0, but still has access to system resources through the hypervisor. The management partition therefore acts as an intermediary between the virtual machines and the hypervisor kernel. If a virtual machine issues a request to physical hardware, the hypervisor traps it and redirects it to the control partition. The control partition then processes it using the virtualization stack and the appropriate device drivers. The system architecture is thus divided between a minimized hypervisor kernel and a control plane running outside the privileged kernel itself. Xen and Hyper-V are typical examples of the microkernel approach [7], although their specific implementations differ in some details.

5) Monolithic architecture: Monolithic architecture approaches the design differently. In this model, the hypervisor core, its supporting subsystems, interfaces, and drivers are integrated into a single, continuous layer. The entire hypervisor resides in Ring 0, the highest privileged level, and this layer also contains a significant portion of the trusted code base. Compared to the microkernel model, this is an architecture with a larger internal footprint, as it includes not only the hypervisor core, but also its support mechanisms and management interfaces.

Unlike the microkernel architecture, the monolithic model does not use a separate management partition. Virtual machines are placed directly above the hypervisor, and requests directed to the hardware are processed within the same privileged layer. Administrative and management operations are performed through interfaces that are part of the hypervisor itself. This arrangement simplifies the communication path between the virtual machine and the virtualization mechanisms, but at the same time concentrates a larger scope of code into the highest level of permissions. VMware ESXi is a representative example of the monolithic architecture [7].

6) The architectural significance of the difference: The difference between the microkernel and monolithic models therefore lies primarily in which components remain in Ring

0 and which are moved to lower layers of the system. The microkernel architecture tries to minimize the content of the privileged layer and separate the control and support mechanisms from the hypervisor core itself. The monolithic architecture, on the other hand, concentrates all major subsystems into a single privileged layer. This difference has direct significance from a security perspective, as it affects the extent of the trusted code base, the structure of the attack surface, and the way in which critical operations of the virtualization environment are implemented.

However, the theoretical division of hypervisors into monolithic and micro-kernel architectures only becomes practical when analyzing specific implementations. Individual platforms implement these models differently, with differences mainly manifested in the organization of the privileged layer, processing of input-output operations, and the coupling between the hypervisor, virtual hardware, and control components of the virtualization environment.

III. VMware ESXi

VMware ESXi is a bare-metal Type 1 hypervisor implemented in a monolithic architectural model. The virtualization layer is implemented through a custom kernel called the VMkernel, which manages physical resources and allocates them to virtual machines.

Unlike solutions that use a separate management domain or host operating system, ESXi centralizes all the main virtualization mechanisms directly in the hypervisor. The VMkernel provides processor scheduling, memory management, and processing of network and I/O operations without the presence of an external operating system [4], [6].

A. VMkernel and compute resource management

Each virtual machine in an ESXi environment is represented as an isolated entity with its own virtual processors (vCPUs), memory, and devices. The VMkernel provides the mapping of these virtual resources to physical hardware.

CPU scheduling occurs directly in the hypervisor, which allocates physical cores to individual virtual processors. Processor virtualization is supported by hardware extensions (Intel VT-x, AMD-V), which allow the separation of privileged operations of guest systems from the hypervisor.

Memory management in the ESXi environment is implemented in the VMkernel, which provides the mapping and allocation of memory resources to individual virtual machines.

B. Processing I/O operations

The VMkernel contains the subsystems needed to handle network and storage operations, as well as physical device drivers. This means that virtual machines communicate with physical hardware directly through the hypervisor.

A virtual machine communicates with an abstract virtual device that is implemented in the hypervisor. Requests are then processed by the VMkernel and routed to the physical device. Unlike micro-kernel models, it does not use a separate control domain or the host operating system as an intermediary for I/O operations.

C. Simplified architecture without a host operating system

The ESXi architecture removes the original service console that was present in earlier versions of ESX. ESXi is implemented as a standalone hypervisor without a full-fledged host operating system.

This:

- reduced system complexity
- limited the scope of code outside the hypervisor itself
- centralized resource control into the VMkernel

The hypervisor thus forms the main privileged layer between virtual machines and physical hardware.

D. ESXi as part of the VMware vSphere platform

ESXi is part of the VMware vSphere virtualization platform, which in addition to the hypervisor also includes tools for managing and accessing virtual servers, such as vCenter or vSphere Client. This means that ESXi is not used in isolation as a virtualization layer itself, but as part of a broader management environment.

IV. Kernel-based Virtual Machine (KVM)

Kernel-based Virtual Machine (KVM) is a virtualization mechanism implemented as part of the Linux kernel. After activating the appropriate modules, the Linux operating system behaves like a hypervisor, with virtualization mechanisms integrated directly into its kernel. KVM does not act as a separate software layer separate from the host system, but extends the existing operating system infrastructure to support virtualization [3], [7], [8].

The emergence of KVM is closely related to the introduction of hardware support for virtualization in modern processors. The Intel VT-x and AMD-V mechanisms enabled more efficient capture of privileged operations at the processor level, which significantly reduced the need for software emulation and at the same time enabled the launch of unmodified guest operating systems. KVM is therefore designed as a solution based on full virtualization with the use of hardware assistance.

A. Architectural structure

The KVM architecture is divided between kernel space and user space. The kernel space contains the kvm module along with architecture-specific extensions for individual processor platforms. This module provides for the creation and management of virtual machines, processing events related to the processor virtualization mode, and switching between the guest system and the host kernel context.

Each virtual machine is represented as a regular process in the Linux system. Processor scheduling, memory allocation, and isolation are therefore implemented through existing Linux kernel mechanisms. Virtualization is thus directly connected to the internal subsystems of the operating system.

The QEMU process runs in the user space, which provides for the initialization of virtual hardware and device emulation. Communication between QEMU and the kernel module takes place through defined interfaces. The division of the virtualization stack between the kernel and user space means that part

of the functionality is located outside the highest privileged level of the system.

B. Virtualization model

KVM uses a full hardware-assisted virtualization model. The guest operating system does not need to be modified, as the processor in virtualization mode allows sensitive operations to be captured and subsequently processed by the virtualization layer. When a privileged operation is performed, control is temporarily transferred to the host environment, after which execution can be returned to the guest system.

This mechanism allows processor virtualization in close cooperation between hardware extensions and the software implementation in the Linux kernel. KVM thus combines hardware-assisted processor virtualization with device emulation or paravirtualization, depending on the virtual hardware model used.

C. Memory management

The KVM memory subsystem is built on existing Linux memory management mechanisms. The mapping of guest memory space to host memory is supported by the processor's hardware virtualization mechanisms and the Linux kernel's memory management. Thus, KVM does not implement a separate memory manager separate from the host system, but integrates the virtualization mechanisms into the existing kernel infrastructure.

D. I/O model

The processing of input/output operations is divided between the kernel module and the user process QEMU. QEMU provides emulation of devices such as disk controllers or network interfaces. Operations generated by the guest system are redirected to the appropriate implementation in user space and then use the services of the host operating system.

To reduce the overhead of emulation, paravirtualization drivers, such as virtio, are used, which reduce the need for full device emulation and allow more efficient communication between the guest system and the host.

E. KVM's place in the wider virtualization environment

The open nature of KVM and its integration into the Linux kernel make it a virtualization solution that integrates well with other mechanisms of the Linux ecosystem, especially process, memory, and device management. This feature supports its use in environments where openness of implementation and binding to existing Linux infrastructure are important.

V. Xen Hypervisor

Xen is a bare-metal (Type 1) hypervisor that is implemented as a separate virtualization layer running directly on top of physical hardware. It does not use a host operating system but takes control of the processor, memory, and interrupts early in the system boot process. Architecturally, it is a microkernel-oriented hypervisor, since the hypervisor core itself is minimized and most support mechanisms are separated into a separate control domain [3], [6], [7].

This proposal reduces the scope of functionality concentrated directly in the hypervisor kernel, as part of the control and I/O mechanisms are moved to the privileged domain Dom0.

A. Xen architecture

After the system is initialized, the Xen hypervisor itself is first started, gaining control over the physical hardware. Its role is to manage processor scheduling between individual virtual machines, ensure memory isolation, and enforce security boundaries between domains. The hypervisor is located at the highest privileged layer of the system and does not contain full-fledged device drivers.

A special privileged domain, called Dom0 is then created above the hypervisor. This is a full-fledged operating system that contains physical device drivers and tools to manage the virtualization environment. Dom0 mediates communication between unprivileged virtual machines and physical hardware. Dom0 also contains backend mechanisms for network and disk requests coming from DomU domains, while these domains use frontend interfaces to access virtualized devices. The backend in Dom0 thus mediates communication with the physical hardware and creates the impression of a dedicated device for guest domains.

Virtual machines of users are referred to as DomUs. They do not have direct access to physical devices, and their I/O requests are mediated through the virtualization layer and the privileged domain Dom0. The guest kernel executes at a less privileged level than the hypervisor itself, while the application code runs in user mode.

B. Virtualization modes

Xen originally implemented a paravirtualization model. In this mode, the guest operating system kernel is modified to use explicit hypervisor calls instead of executing privileged instructions. This approach reduces the need for emulation and allows for more efficient communication between the virtual machine and the hypervisor.

With the expansion of hardware support for processor virtualization, Xen was supplemented with a full virtualization mode (HVM). In this case, an unmodified operating system can be run. Privileged operations are captured by the hardware and processed by the hypervisor via the VM exit mechanism. This model increases compatibility with proprietary systems and expands deployment options in heterogeneous environments.

C. Processor and memory management

Processor scheduling in Xen is implemented through an internal scheduler that allocates computing time to individual virtual processors based on defined weights. This mechanism makes it possible to control the share of resources between individual domains.

Memory is strictly isolated between domains. The hypervisor controls the mapping of memory pages and ensures that individual virtual machines cannot access the memory space of

Hypervisor	Advantages	Disadvantages
VMware ESXi	High stability and reliability; easy management through vCenter; advanced features such as vMotion, High Availability (HA), Distributed Resource Scheduler (DRS), and Fault Tolerance; broad hardware and enterprise application support; comprehensive documentation and vendor support.	High licensing costs; proprietary and closed-source platform; limited customization options; dependence on the VMware ecosystem.
KVM	Open-source and integrated into the Linux kernel; no licensing fees; high performance with low virtualization overhead; flexible and highly customizable; compatible with management frameworks such as Libvirt, OpenStack, and Proxmox VE.	Requires advanced Linux administration skills; less intuitive management compared to VMware ESXi; enterprise support depends on the selected distribution; some advanced features require additional configuration.
Xen Hypervisor	Open-source solution with a mature architecture; supports both paravirtualization and full virtualization; strong security through domain isolation; suitable for cloud computing and hosting environments; low hypervisor overhead.	More complex deployment and administration; smaller community compared to KVM; less common in contemporary enterprise environments; certain features require specialized tools or distributions.

TABLE I
COMPARISON OF VMWARE ESXI, KVM, AND XEN HYPERVISORS

other domains. This ensures a basic level of isolation between individual domains sharing the same physical server.

VI. CONCLUSION

This paper provided an overview of hypervisor architectures and a descriptive comparison of three widely used hypervisors, VMware ESXi, KVM and Xen, from the perspectives of architecture, virtualization techniques, resource and I/O management, security, and typical deployment scenarios. The comparison showed that each hypervisor offers specific strengths and limitations, so none of them can be considered a universally optimal solution.

VMware ESXi follows a monolithic model in which the main virtualization mechanisms are concentrated in the VMkernel. It is a mature enterprise-grade solution with advanced management tools and high reliability, although these advantages come with licensing costs and dependence on the VMware ecosystem. KVM integrates virtualization directly into the Linux kernel, which makes it a flexible and cost-effective open-source platform, particularly suitable for Linux-based environments and cloud infrastructures. Xen follows a microkernel-oriented design with a minimized hypervisor and a separate control domain. Together with its support for both paravirtualization and hardware-assisted virtualization, this makes it an attractive choice for security-critical deployments.

The selection of an appropriate hypervisor should therefore be based on the intended application, performance and security requirements, administrative expertise, licensing constraints and existing infrastructure. Table I summarizes the key advantages and disadvantages of the compared hypervisors as a basis for such a decision.

Future work should focus on experimental benchmarking under identical hardware and workload conditions to quantify differences in performance, resource utilization, scalability and virtualization overhead. Such measurements would complement the descriptive comparison presented in this paper.

ACKNOWLEDGMENT

Co-financed by the European Union through the Slovakia Programme under project No. NFP401101C360: Research and development of advanced artificial intelligence solutions for the detection of cyber threats and defense against sophisticated attacks.

REFERENCES

[1] M. Satalino, H. Wu and Y. Pang, "Performance Evaluation of Hypervisors: A Comprehensive Benchmarking Study on VMware, Xen, and KVM," 2026 3rd International Conference on Computer and Multimedia Technology (ICCMT), Sanming, China, 2026, pp. 71-78, doi: 10.1109/ICCMT69755.2026.11608177.

[2] S. Varrette, M. Guzek, V. Plugaru, X. Besseron and P. Bouvry, "HPC Performance and Energy-Efficiency of Xen, KVM and VMware Hypervisors," 2013 25th International Symposium on Computer Architecture and High Performance Computing, Porto de Galinhas, Brazil, 2013, pp. 89-96, doi: 10.1109/SBAC-PAD.2013.18.

[3] S. Lozano, T. Lugo, and J. Carretero, "A Comprehensive Survey on the Use of Hypervisors in Safety-Critical Systems", 2023. DOI: 10.1109/ACCESS.2023.3264825.

[4] A. Desai, R. Oza, P. Sharma, and B. Patel, "Hypervisor: A survey on concepts and taxonomy", in International Journal of Innovative Technology and Exploring Engineering, vol. 2, no. 3, 2013.

[5] J. Shropshire, "Analysis of monolithic and microkernel architectures: Towards secure hypervisor design", in Proceedings of the Annual Hawaii International Conference on System Sciences, 2014. DOI: 10.1109/HICSS.2014.615.

[6] H. Fayyad-Kazan, L. Perneel, and M. Timmerman, "Benchmarking the Performance of Microsoft Hyper-V server, VMware ESXi and Xen Hypervisors", in Journal of Emerging Trends in Computing and Information Sciences, Volume 4, Issue 12, 2013.

[7] A. Binu and G. S. Kumar, "Virtualization techniques: A methodical review of XEN and KVM", in Communications in Computer and Information Science, 2011. DOI: 10.1007/978-3-642-22709-7_40.

[8] G. Motika and S. Weiss, "Virtio network paravirtualization driver: Implementation and performance of a de-facto standard", in Comput. Stand. Interfaces, Volume 34, Issue 1, 2012, DOI: 10.1016/j.csi.2011.05.002.