TrustOSV: Building Trustworthy Executing Environment with Commodity Hardware for a Safe Cloud

Xiaoguang Wang, Yi Shi*, Yuehua Dai, Yong Qi, Jianbao Ren, Yu Xuan
Dept. of Computer Science, Xi'an Jiaotong University
Xi'an, China
{xiaogw,xjtudso,renjianbao,xuany}@stu.xjtu.edu.cn {shiyi,qiy}@mail.xjtu.edu.cn

Abstract—**The Infrastructure as a Service (IaaS) cloud computing model is widely used in current IT industry, providing the cloud users virtual machines as the executing environment. However, current executing environment the cloud provided is not trustworthy. For a user's executing environment faces threats from malicious cloud users who aim at attacking the underlying virtualization software (virtual machine monitor, VMM, or hypervisor). In this paper, we first make an analysis of the potential threats to a commodity hypervisor, and then propose architecture to build a more trustworthy executing environment for IaaS cloud. The main ideas of our architecture are: removing interaction between hypervisor and the exposed executing environment, enhancing platform data secrecy as well as providing feature rich environment attestation. To prove the effectiveness of our architecture, we build a prototype system, named TrustOSV, which can host multiple trustworthy isolated computing environments on multi-core x86 hardware. The final evaluation shows that TrustOSV can provide enhanced security guarantees to the exposed VMs at modest cost.**

Index Terms—**Safe cloud computing, tiny-hypervisor, trustworthy executing environment.**

I. Introduction

Cloud computing has become the buzzword of IT industry. In the past few years, a lot of IT companies have moved their workloads to the cloud. This trend will become more and more notable in the future [1]. Among the several proposed cloud computing models, IaaS (Infrastructure as a Service) cloud appears to be the most interesting one for customers, as it offers VM hosting as the service. This kind of service enables cloud users to run their applications and services in remote cloud. And it facilitates cloud users to maintain their personalized software stack and host the ever increasing customers' data. However, the lack of trustworthiness of the remote cloud executing environment has limited its further deployment and thus proves to be the major public concern [2].

In IaaS cloud computing model, cloud users who run their workloads have no choice but to trust the cloud providers. A cloud provider normally uses virtual machine monitors to multiplex hardware resources, do server consolidation, etc. Hence, a normally running virtual machine monitor should successfully support the concurrent execution of multiple operating systems (commonly called the Guest Operating System, Guest OS) with the capacity of complete executing isolation. But unfortunately, current attacks [3], [4], [5] and the vulnerability reports [6], [7] show that traditional VMMs (such as Xen [8], VMware [9] and KVM [10]) still have a lot of security problems, which may cause the up-running VMs insecure. This is mainly caused by the large attack surface between the VMM and the Guest OS. In other words, a Guest OS needs to frequently trap to hypervisor to request the privilege services. This brings various interactions between a Guest OS and the hypervisor, which can lead to the violation of VMM executing integrity. Once an intended attacker hijacks the host VMM, all of the other running Guest OSes will be under control. Thus the security sensitive workloads from cloud users may execute untrusted.

To make matter worst, the underlying VMM software itself can not be trusted. The cloud administrator may be genuinely mal-intentioned, or "honest but curious", or simply ignorant of threats posed by malware. With the *VMM-to-guest interactions*, any malicious operations performed by the underlying hypervisor (or a privileged *dom0 VM* [8]) would result in control flow hijacking attack or memory dumping attack in user VMs [11]. At the same time, as all the sensitive data are stored on cloud platform, the adversaries and/or cloud administrators may make unauthorized access to unencrypted sensitive data [12]. For example, Google recently fired two employees for breaching user privacy [13]. Hence IaaS customers have no way to control their remote executing environment, and lack the knowledge of whether their executing environment runs in a correct platform. The lack of "trust proof" makes customers have no choice but to "believe" that the underlying platform works in a good state.

Being aware of such serious situation, current cloud providers may choose to fix those vulnerabilities by promptly patching the underlying software [14], [15]. While before the patches work, those vulnerabilities may still lead to zero-day exploits, raising user privacy leakage or possibility of data loss [2], [16], [17]. Researchers

*Corresponding author: *Yi Shi (shiyi@mail.xjtu.edu.cn)*

This work was supported in part by grants from National Natural Science Foundation of China (Project No.60933003 and No.61272460).

also propose to make formal verification on cloud infrastructure softwares [18], [19]. However, those researches show that it brings tremendous cost on formalizing the prototype software system [18]. For example, G.Klein et.al built the seL4 OS kernel with formal verification method, and it takes *20+* person years to formalize a *8K LOC* (lines of code) operating system kernel. The experiment indicates that it is nearly impossible to formalize commodity hypervisor like Xen, KVM. Most recently, researchers have also made attempts by modifying hardware architecture (e.g. adding hardware registers and special hardware assisted data storage) to enforce secrecy over the running guest OSes [20], [21]. However, these approaches may require large scale hardware modification, and unfortunately current hardware venders haven't adopted their approaches yet.

In this paper, we first analyze why a current commodity hypervisor cannot provide safe executing environment for cloud users. Then we argue the possibility of a retrofitted hypervisor model, and provide architecture which can be used to build trustworthy executing environment for remote cloud users. Finally we demonstrate the possibility of this architecture with a prototype system implementation.

We leverage the currently popular multi-core hardware platform and the hardware-support for virtualization (such as AMD SVM [22], Intel VT-x [23]) to realize such a system. By making Guest OSes explicitly run on pre-allocated cores and RAMs and forcing the Guest OS to use network software instead of emulated I/O device, our system can remove most *hypervisor - guest interactions*. Hence it eliminates the major threat to an executing environment. By leveraging a stackable file system over existing file systems [24], TrustOSV provides application-transparent data encryption, and thus prevents data leakage against potentially malicious cloud administrators.

TrustOSV system also provides hash attestation of the initial executing environment state to remote cloud users, which can ensure the integrity of a trusted environment. Through planting a non-bypassed attestation agent in hypervisor context, TrustOSV provides flexible and feather rich attestation service with various customers' demand. By leveraging a memory-locking technique, TrustOSV provides the runtime integrity on its security guarantees. With all the protection and platform attestation, the users can make sure that they are served by trusted VM instances.

We list the contributions below, which also serves as a roadmap to the paper:

1) We analyze the cause of VMM system software vulnerability, and indicate one important reason why hypervisor cannot provide safe executing environment for cloud users in hardware sharing environment, taking a commodity VMM as an example (§2).
2) We point out the possibility of multi-core based trustworthy VMM architecture for strict resource isolation and feather-rich platform attestation (§3). By illustrating the life cycle of a VM running, we show it provides life long protection over the outsourced VMs (§4).
3) We implement TrustOSV, a prototype system to realize such architecture (§5), and evaluate its effectiveness in both security and performance (§6).

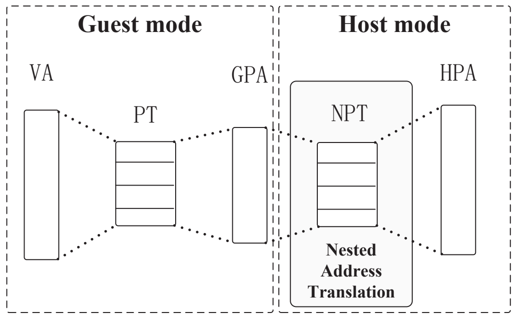

Fig. 1: Nested paging address translation diagram

II. Background and Security Weakness of Commodity Hypervisor

Hypervisor in cloud computing environment has been widely used in hardware resource multiplexing, application mobility, server consolidation, and task isolation, etc. A normally running virtual machine monitor should successfully support the concurrent execution of multiple operating systems. The VMM realizes those characteristics by multiplexing processor cores and physical memory, emulating I/O devices, etc. But these can also bring security concerns. In the following section, we will demonstrate this kind of security weakness caused by hypervisor-guest interaction. The goal of the analysis is to answer three questions: 1) what does current hardware virtualization support? 2) Is commodity virtualization software secure enough to provide trustworthy guest environment? 3) If not, does this kind of hypervisor vulnerability exist in real world?

What does current hardware virtualization support? Current processor supports hardware-assisted virtualization and a new processor mode – guest mode. Normally, a guest operating system runs in guest mode for most of the time. When a privilege instruction is emitted, the hardware saves the guest state to a virtual machine control block (VMCB) structure, and turns to host mode executing.

Modern hardware-support for virtualization also provides nested page table (NPT) address translation technique. A guest's view of its physical address space could be different from the actual layout of the CPU's physical address space. Therefore, the host needs to translate the guest physical addresses to the CPU's physical address (also called the host physical address). Figure 1 illustrates it.

Is commodity virtualization software secure enough to provide trustworthy guest environment? By multiplexing processor cores and physical memory, commodity hypervisor could directly control the executing flow of a

Guest OS. That means an operating system in the cloud should trust the underlying virtualization software. Unfortunately, hypervisor in cloud computing environment (especially in a multi-tenant cloud) cannot be trusted. This is because that any malicious operation either exploited by a malicious guest VM or mis-operated by an administrator could potentially lead to an untrusted executing environment. For example, one important function a hypervisor provides is the I/O emulation. By emulating I/O device with an emulator (normally a QEMU emulator [25]), the Guest OS runs as if it owns the physical device. When a Guest OS emits an I/O instruction, the hypervisor capture this event, followed by processor guest-to-host mode switching together with complicated event handling and sophisticated software stack saving and restoring. This over complex *guest - hypervisor interaction* may make a guest executing environment much more vulnerable since it brings a larger attack surface.

We call the operations or instructions that switch the system from guest mode executing to host mode executing the *VM_EXIT events*. Other system behaviors in virtualization environment would also cause VM_EXIT, such as Nested Paging page fault, operation on specific registers, processor HLT instruction, etc.

We capture the VM_EXIT events on a KVM hypervisor for a 10 seconds period and record the type of VM_EXIT event and its corresponding appearance count.

Table I shows the type of each VM_EXIT event and its distribution in a KVM hypervisor. We configure a single core, 1GB RAM virtual machine as the Guest OS. We manually assign different workload, record the VM_EXIT events numbers and analyze the corresponding reasons. The table shows that with normal workload such as compiling Linux kernel (kernel built), the NPT fault rate could be rather high. And the result also shows that even for the empty workload, the frequency of VM_EXIT event is also remarkable.

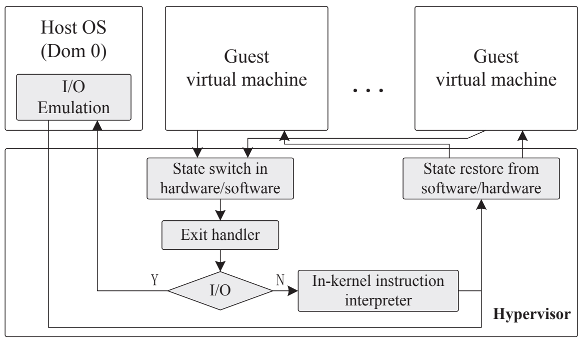
Host OS
(Dom 0)
I/O Emulation
Guest virtual machine
. . .
Guest virtual machine
State switch in hardware/software
State restore from software/hardware
Exit handler
Y
I/O
N
In-kernel instruction interpreter
Hypervisor

Fig. 2: Operations in KVM to handle VM_EXIT event

Beside the high frequency of VM_EXIT events, the event handling procedure is also complex. To further demonstrate the complexity of *hypervisor-guest interaction*, we analyze the VM_EXIT event handling procedure in KVM hypervisor (Shown in Figure 2). During the complex host-guest state switch, malicious attacker would be able to hijack hypervisor's control flow.

TABLE II: Vulnerability categories and their occurrence counts in Commodity Hypervisors

$Categorie$	Xen	KVM	$VMware$
Crash/DoS	43	12	14
Access Host Resource	6	2	9
Gain Privilege	7	4	14
Others	2	1	15

Does this kind of hypervisor vulnerability exist in real world? We manually examined hypervisor vulnerabilities from NVD database [26] in the last two years [1], and checked out 129 vulnerabilities covering from mainstream hypervisors like Xen, KVM and VMware. We found that more than 98 out of the 129 vulnerabilities were caused by hypervisor-guest interactions, and 69 of them could cause a host or running user VM crashing, leading to deny of service attacks. We found 25 out of the 129 vulnerabilities made the guest obtain host privilege. Table II shows the result.

III. System Design

A. Threat model, Goals and Assumption

In this paper, we address two types of threat originated from *hypervisor-Guest OS interaction*: attacks from malicious Guest OS to hypervisor (malicious cloud users may trigger vulnerabilities of the hypervisor through the guest-to-hypervisor interaction); and the contrary attacks from malicious underlying environment to the up-running user computing environment. Although TrustOSV focuses on how to provide cloud user a trustworthy computing environment (the second type), we should take the first threat into account. Because a compromised hypervisor could easily cause an untrusted execution of users' VMs.

Our primary goal is to prevent hypervisor directly interacting with a computing environment ***runtime*** via intercepting privileged I/O operation, handling memory mapping, etc. Meanwhile, the Guest OS is precluded from triggering vulnerabilities in hypervisor via those VM_EXIT events. At the same time, we prevent the malicious or "honest-but-curious" administrator from dumping sensitive information in memory or leaking data from the permanent storage. Moreover, we provide cloud users *trust* information about the underlying environment and their own cloud executing environment (such as guest initial state, immutable running state, etc.).

We assume to run the system in a commodity x86 hardware, which means it is equipped with hardware assisted virtualization like AMD SVM or Intel VT-x extension. And we also assume the platform is equipped with TPM chips to guarantee load-time integrity. At last, the system is assumed to run on a platform with correct hardware configuration, and any physically destructive actions should be prevented.

[1]The vulnerabilities we examined are from Jan. 1, 2012 till Sept 31, 2013

TABLE I: The occurrence count of each VM_EXIT event in a KVM Hypervisor

10 seconds VM execution	**VM_EXIT event categories**				
	Total VM_EXIT events count	***I/O ports access***	***Nested page fault***	***Virtual CPU related events***	***Others***
Empty workload	20072	6707	7326	6038	1
Linux compilation	33049	11836	15848	5364	1

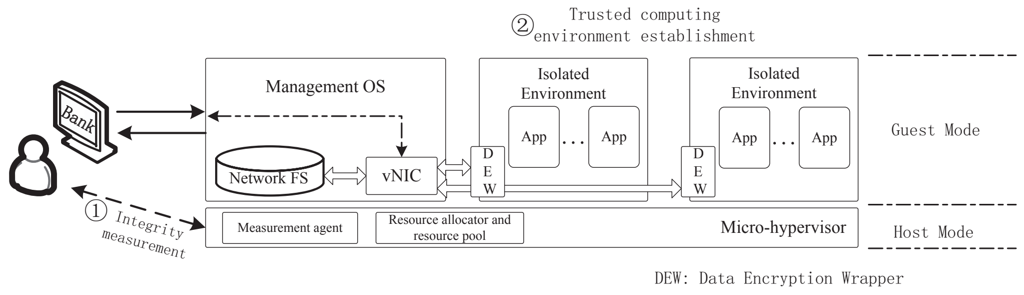

Fig. 3: Overall architecture of the executing environment isolation system

B. Architecture overview

Figure 3 shows the architecture of our system, and we will demonstrate in the rest of this paper how this architecture can provide an isolated and trustworthy environment to execute the security-sensitive applications.

The overall system contains four main components: an isolated and strictly protected micro-hypervisor, a *management operating system* for controlling cloud executing environment, several isolated trustworthy *executing environment* to run security-sensitive applications, and a *hypervisor-embedded attestation agent* that attests the underlying environment to remote users. Our system can create multiple copies of trustworthy executing environment on a hardware-sharing platform.

The central part of the system is a micro-hypervisor. It boots first after the hardware has warmed up. And it will collect the system hardware information and then boot the first operating system for platform management. The central job for micro-hypervisor is to partition hardware resources and allocate them to each executing environment. Once it begins to run, the micro-hypervisor isolates itself from all the running OSes by leveraging existing hardware protection mechanisms (For example, write protection bit in page table, nested page table, I/O MMU, etc.).

With the help of an embedded attestation agent, the micro-hypervisor also enables cloud users to attest system information, the initial integrity of the up-running operating system memory image, and any other targets specified by cloud users. To prevent potential attackers from compromising the platform attestation (e.g. modifying attestation logic or bypassing the attestation logic), the micro-hypervisor will lock its memory code pages down immediately after it boots up. The details will be shown in the following part of this section.

The management operating system runs directly on the pre-allocated hardware with few interactions to micro-hypervisor (except for some commands like, *start a VM*, *show running VMs*, etc.). It has only one interface which notifies the micro-hypervisor to set up the executing environment. The management OS contains a virtual network card for user VMs' network connection and is responsible for routing users' packages to the outside world.

After setting up the management operating system, the whole system is ready to create trustworthy executing environment. When a secure executing environment is needed, the management OS would notify hypervisor to boot an executing environment. The isolated executing environment is also booted by micro-hypervisor, but it has no interaction with the hypervisor immediately after the boot-up procedure.

In the rest of the section, we show the design principles to build a trustworthy executing environment in TrustOSV system.

C. Design principles

In order to prevent the safe executing environment from being destroyed by untrusted infrastructural softwares, we should ensure: 1) initial state correctness of the executing environment; 2) no runtime hypervisor intervention for the executing environment; 3) data secrecy for customers' permanence storage.

In our system design, cloud user would first attests the integrity of the initial executing environment state before using it. Customers would first send the attestation request (with a challenge key) to TrustOSV hypervisor, in order to attest the integrity of the kernel images and underlying system state. TrustOSV calculates the hash, seals it with other system information with the platform signing key, and sends it back. Customers unseal it with the public key from trusted third party servers, and then verify the attestation result. In this way, TrustOSV ensures the customers the system runs in correct state, and could also prevent the initial kernel image faking attack [27], [28].

To follow the second design principle, the interaction between guest and host hypervisor should be removed. From the analysis in section II, we can remove the interaction by eliminating the VM_EXIT in a hypervisor. We have to mention that we could not simply clear the corresponding bit in VMCB, since a real NPT fault without a handler will crash the isolated executing environment.

For the third design principle, we need to provide data secrecy for customers' permanent storage. While in order to provide permanent storage secrecy, TrustOSV should not sacrifice the transparency and flexibility of the cloud storage system. In other word, we should not let the customers to identify their sensitive data and encrypt them before send to the cloud.

In the rest of this section, we show in detail how these design principles come to work.

1) VM_EXIT avoided resource allocation: Nowadays the hypervisor dynamically manages processor cores and system memory. By dynamically managing the system resource, hypervisor can provide more virtual resources than the physical machine owns. But this can also bring frequent hypervisor - Guest OS interaction, which is a large attack surface to an ideal isolated environment. In our prototype system, we statically pre-allocate processor cores and physical memory to the executing environment before it is launched.

In a traditional hypervisor (such as KVM), a Guest OS uses page tables of its own to translate guest-virtual address to guest-physical address (GVA->GPA). And the hypervisor translates guest-physical address to host-physical address (GPA->HPA). In our experiment, more than 45% of the VM_EXIT events occur in the above steps and more than 80% VM crashes are resulted of the exit events. Also traditional hypervisor leverages on-demand paging technique to complete address translation, which may also bring large attack vector for intruders.

On platform which implements nested paging in hardware, the MMU consults both the guest and host page tables to perform a 2D (GVA->GPA->HPA) address translation. If we prepare the nested page table in advance, there will be no interaction with virtualization software. Thus, by pre-allocating memory ranges for an isolated executing environment and preparing NPT in advance, we can eliminate 45% of the VM_EXIT events and observably improve the safety of the executing environment.

To remove the hypervisor-guest OS interaction, TrustOSV relies on a static resource allocator and the resource pool. In particular, TrustOSV hypervisor contains a resource pool for customer VM creation. The resource pool maintains a list of platform idle CPUs and physical memories. On receiving the starting VM requests, TrustOSV resource allocator allocates the CPU cores and multiple ranges of memory according to customers' demands. The resource allocator is also responsible for boundary checks. For example, resource allocator checks the physical pages' ownership in order to avoid illegal memory double mapping. [2]

2) Stackable encryption filesystem over distributed protocol: As described in section II, emulated I/O device could be another major source of threat to isolated environment. How to eliminate the security problem brought by emulated I/O device while still maintain the I/O functionality is an open problem. One way to remove the emulated I/O device is to dedicate physical I/O device to the isolated executing environment. However, this solution does not scale.

In the architecture, we replace the I/O device with distributed protocol. Since there is no need for a mouse, keyboard, or printer in cloud environment, the only thing needed is the network connection and the storage for the executing environment. We provide a virtual network interface card (NIC) for network connection to the executing environment. In our virtual NIC design, we use the shared memory for network package transmission, and leverage inter-processor interrupt (IPI) for package arriving notification. The management operating system is configured as a router to transmit packages from executing environment to the outside world. More details of our virtual NIC design are described in paper [29].

Another important part of I/O operations is accessing storage system. In our protocol system design, we provide the isolated executing environment with network file system (NFS). It can offer the executing environment storage service, while without using I/O device emulation. And this can greatly eliminate the MMIO VM_EXIT event.

However, using an existing network file system alone cannot provide secrecy for customers' data, especially permanent file data on cloud servers. Since an "honest-but-curious" cloud administrator may wish to view customer data with privileged authority. One way to solve this problem is to identify every sensitive files by cloud users themselves, encrypt each file and send it to the cloud. But this user involved approach may sacrifice the flexibility and convenience of the cloud. Therefore, we decided to plant a stackable file system as building blocks to add desired encryption functionality over standard network file system interfaces. In this way, the security of the data exchanged between clients and servers is ensured.

Stackable file systems are layers that intercept VFS-layer requests, probably modify the associated data or metadata and forward them further. They can be mounted over existing file systems, directories, or files. In TrustOSV design, we insert an encryption wrap file system between VFS layer and the network file system, which provides data secrecy for TrustOSV users.

Figure 4 shows a simple stackable file system diagram which receives requests from VFS, encrypts data blocks and passes them to the underlying file systems.

Through the design approaches described above, we

[2]One purpose for double mapping check is to prevent a physical memory page from accessed by two or more guest OS (or hypervisor itself). By checking illegal double mapping, TrustOSV can provide a strictly isolation for guest executing environment. However, the only one legal double mapping in TrustOSV design is used for virtual network card buffering. We analyze this scenario in Section VI.

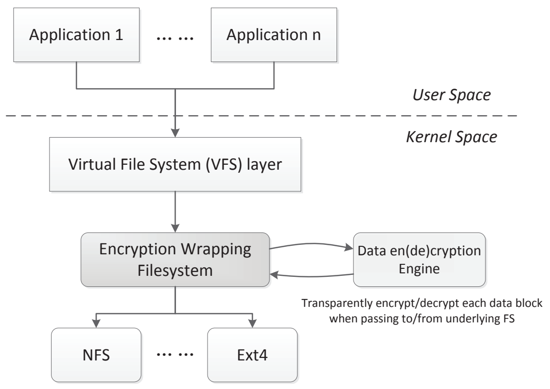
Application 1
… …
Application n
User Space
Kernel Space
Virtual File System (VFS) layer
Encryption Wrapping Filesystem
Data en(de)cryption Engine
Transparently encrypt/decrypt each data block when passing to/from underlying FS
NFS
… …
Ext4

Fig. 4: A stackable filesystem diagram for transparent data encryption

could not only remove the VM_EXIT events caused by I/O emulation but also provide data secrecy against potential malicious cloud administrators.

3) Self-locked attestation agent: To provide a feather rich platform attestation, TrustOSV also provides an in-context, feather rich platform measurement agent. Although the boot-up integrity can be measured by secure co-processors (like TPM), attestation based on hardware chips may still has limitations. For example, a hardware based platform could be difficult to break the semantic gap between attestator and the attestation objects, like the page mapping of hypervisor or the resource pool for guest VMs.

In TrustOSV design, we plant a tiny attestation agent in hypervisor context, which facilitates cloud users to attest the underlying system condition. However, the problem arises when the hypervisor itself is compromised (such as suffering from a buffer overflow attack, etc.); the in-context measurement agent will be bypassed. To solve this problem, we adapt a non-bypassable memory lockdown technique [30] for attestation agent. Although it is not new, we first apply it to TrustOSV hypervisor and use it to give full protection over attestation agent module. In the rest of this section, we make a brief review of the no-bypassable memory lockdown technique and then we will provide more details about its application scenario – the memory-locked attestation agent – in TrustOSV design.

In commodity x86 architecture systems, the paging-based system protection divides the virtual address space into pages, and physical memory into frames of the same size. When paging mode is enabled, the x86 memory management unit (MMU) translates virtual address to physical memory address mapping by consulting an in-memory page table. Therefore, by modifying the page table content, system software can map the virtual address to any physical address needed.

The commodity x86 hardware also provides write-protection and non-executable for a page attribute [22]. By setting the corresponding page table entry write-protected, the x86 hardware enforces the page as read only. Therefore, when TrustOSV starts up, it prepares the page mapping and sets the pages which contains attestation agent [3] read-only. Commonly, the physical page of the page table itself is set writable. However, this facilitates attackers from directly mapping the physical address (e.g. by a buffer overflow attack) to its own address space. Hence the code and data of the TrustOSV kernel may suffer from memory modification attacks. And all the hypervisor independent resource allocation and in-context attestator would be bypassed.

In TrustOSV design, we adopt the approach proposed by Wang Z. et. al [30]. By setting the physical pages which host the page table read-only, TrustOSV remarkably prevents malicious page table update. By checking the update intention, TrustOSV accommodates benign page table updates. Fortunately, due to the simple program logic, TrustOSV will not update its own page table during the runtime and its page table will be locked immediately after TrustOSV sets up.

IV. Lifecycle of a trustworthy executing environment

As discussed in previously sections, the trustworthy running of an executing environment depends on a secure TrustOSV hypervisor. The assurance of boot up integrity could be achieved by leveraging a hardware root of trust, like a TPM chip. The use of TPM chip has been well studied and various libraries have been developed to provide boot up integrity [31], [32]. We now review the lifecycle of an outsourced job on TrustOSV platform and highlight how TrustOSV provides a trustworthy executing environment.

Initialing TrustOSV platform. When the system boots up, the boot up protocol will first load TrustOSV kernel with hardware integrity protection. Then TrustOSV sets the memory mapping for itself and reserves a private memory range to isolate itself from other running guest OS. After that, TrustOSV sets up several empty page mappings for guest OSes and clear the guest physical memory ranges by writing all 0s.

Starting a VM. Once customers start a running VM, TrustOSV allows customers to attest their remote kernel images integrity to prevent an image modification attack. On starting a VM, customers send remote platform a random challenge value (a nonce) to prevent record-and-replay attack.

On receiving the starting request, TrustOSV first generates the hash of guest OS kernel images (kernel and initrd) and the hash of the protection over TrustOSV kernel (including attestation agent, resource allocator, and memory lockdown logic). Then, the TrustOSV hypervisor signs the hash values as well as the VMID and the nonce with a hypervisor reserved signing key. [4]. This signature (SIG_{hash}) would be returned to system customers. The customers get the signature and may retrieve the

[3]Indeed, besides the attestation agent code, any other kernel code pages can be protected by setting non-writable. For example, the resource allocator code and even the boot-up code.

[4]Note: On platforms equipped with hardware based secure chips - like a TPM - it could be replaced by the hardware reserved signing key for stronger secrecy

encrypted values from a trusted third party server to verify the correctness.

$$SIG_{hash} = SIGN_{priv\_key}(NC||hash(guest\ kernel\ images)||hash(protection\ logic)||VMID) \tag{1}$$

VM_EXIT avoided runtime executing environment. To ensure guest OS running safely and isolatedly, TrustOSV provides a VM_EXIT avoided runtime for guest execution. As described in previous section, TrustOSV pre-allocates the hardware resource for the guest executing environment. To avoid the MMIO VM_EXIT events while still provide storage service, TrustOSV leverages existing network filesystem for the running guest. The security of the remote storage is guaranteed by a stackable file system layer, making applications transparently encrypt the data at the modest cost (Describe in Section VI).

Moreover, due to the *resource pre-allocation* and *distributed protocol instead of I/O*, a guest OS is no longer trapped to TrustOSV. Thus any unexpected traps will be simply ignored by TrustOSV handlers (e.g. unexpected hypercalls). And we should also note that TrustOSV is not responsible for the executing environment hangs caused by the guest OS crashes. Such problems can be addressed by either the reliable OS kernel techniques [33], or the verification of OS design [18].

VM Termination. When the VM is terminated, TrustOSV collects the corresponding resource back to the resource pool. At the same time, TrustOSV clears all the memory ranges a guest environment has used by writing the memory ranges with all 0s. With that, TrustOSV prevents in-memory data leakage via cold memory dumping.

V. Implementation

We implement the prototype system based on a homegrown efficient light-weight hypervisor named OSV [34]. The original purpose of OSV is to provide high performance computing. In this paper, we focus on presenting how to build the trustworthy computing environment based on OSV for cloud users.

In TrustOSV implementation, the first 128M physical memory is reserved for TrustOSV kernel. TrustOSV protects this memory range by removing it from all the NPTs of guest VMs and clearing the corresponding bits in device exclusive vector (DEV) to prevent DMA based memory occupation.

We manually modify the link script to locate a continuous physical memory range, which starts from `0x400000`, for TrustOSV kernel's page table, and the 32MB memory behind it to serve as guest address mapping pool. We also adjust the link script to separate the code pages from data pages. This facilitates the runtime code memory lockdown as we described in Section III-C.

When TrustOSV boots up, it first sets up its own page mapping. Then TrustOSV turns on the WP protection in `CR0` register, and sets the code pages as write protected. TrustOSV also enforces the "Write$\oplus$eXecute" (W$\oplus$X) protection over all its physical memory space, to prevent code injection attack. After all those preparations, TrustOSV locks down the physical pages which host the corresponding page table entries as read-only to prevent page table modifications by malicious attackers. Because of the small code size and the simple program logic, TrustOSV uses a `1:1` memory mapping and will not change the page map during its execution.

The TrustOSV implementation leverages normal Ubuntu distribution as platform management OS [5], and uses NFS backed Linux kernel to serve for the trusted computing environment. We slightly modify the kernel configuration in order to load file system from network. To prevent information leakage through NFS server, we insert ecryptfs [35] stackable filesystem between VFS layer and the network filesystem, which can transparently encrypt/decrypt customers' data from/to the running applications.

As described in section III-B, we provide the network connection between the trusted computing environment and the outside world. We configure the management OS as the router for package transmitting. To simplify our implementation, we leverage the existing network control software, such as VNC viewer and SSH client, to view and control the executing environments.

VI. Evaluation

In this section, we present the evaluation results of our trusted cloud executing environment. First, we make an analysis of the security guarantee provided by TrustOSV. After that, we report the performance overhead on executing environment with several benchmark experiments.

A. Security Analysis

The trustworthiness of the initial VM (exposed computing environment) is guaranteed by the underlying layer of micro-hypervisor. The micro-hypervisor provides the integrity proof of the VMs and underlying VMM to remote cloud users. TrustOSV removes the possibility of an administrator starting the platform in an untrusted state, e.g., by starting a rogue hypervisor. Similarly, any malicious modifications to the execution image before the VM is launched will be recorded and sent to cloud users. Note that an attacker cannot pre-store the valid hash value and reply it during the check, since we use a challenge-response process with a user indicated random challenge key. After this stage, the cloud users can be sure that the remote underlying environment is in safe state.

The TrustOSV system can also provide the isolated runtime of the computing environment. By detaching the VM's physical memory from the management OS's control scope, TrustOSV can efficiently avoid the privacy sniffing (sensitive memory sniff) from malicious cloud administrator. Furthermore, the system pre-allocates the physical memory and the processor cores to each running

[5]Note that although TrustOSV owns a management OS, it has limited ability to access and leakage customers data. As we will analyze in Section VI-A, it cannot access guest environment memory, and may only access the encrypted customers data.

TABLE III: Source line of code in TrustOSV's TCB

$TrustOSV component$	$SLOC$
Micro-hypervisor kernel	6917
Virtual NIC	810
Attestation verifier	1052
Crypto library	1380
Total	***10159***

VM before its boot stage. Thus it can remarkably reduce the *hypervisor-guest* interaction, which contributes the major attack surface of the running VMs. In fact, apart from a few VM management hypercalls (like, start a VM, list running VMs), TrustOSV has no other interaction with the running VMs.

Moreover, TrustOSV protects customers' permanent data in the cloud. As we have described in Section V, an ecryptfs is used to transparently encrypt customers' sensitive data. With ecryptfs, only data users who satisfy the identity authentication can access the plaintext data. And the identity authentication can be realized by either promoting cloud user a password input or providing an authentication key from cloud customers. We would also emphasize that a malicious cloud administrator cannot intercept the password transition, both because of the SSL protection for data transfer and the strict memory isolation in TrustOSV design.

Another important factor for secure system is the small TCB code size, since a small TCB code size can make the formal verification procedure possible [18]. Table III displays the code size of TrustOSV on its different parts, as measured by the SLOCCount [36] tool. The micro-hypervisor kernel contains 6,917 lines of C and assembly code. The virtual network card for VM connection has 566 LOC. The verifier for user attestation contains 1,052 LOC, while the cryptography library contributes 1,380 LOC. This totals the size of the TCB to only 10,159 LOC.

To better understand the trustworthiness provided by TrustOSV, we examine several real-world vulnerabilities from NVD [26] . Since some of these vulnerabilities rely on specific VMM implementation, we only make analysis on the vulnerability mechanism and demonstrate that TrustOSV is free from those vulnerabilities.

The first vulnerability we examined is CVE-2008-3687, a heap-based buffer overflow in Xen hypervisor. This could allow unprivileged domain users (Xen domU) to execute arbitrary code via the flask_op hypercall. While for TrustOSV system, we remove the *hypervisor-Guest OS* interactions, and provide a hypervisor-independent computing environment for cloud users. So the underlying VMM may not easily be modified. Another vulnerability we examined is CVE-2012-0045, which happens in Linux kernel before 3.2.14. The KVM x86 emulator under this version does not properly handle 0f05 opcode, which allows a malicious guest to crash the host via a crafted application. As for TrustOSV, we remove the device emulation by using virtual network card and network protocols like network file system. Therefore, the users' executing environment will not suffer from this kind of attack.

B. Performance Evaluation

We evaluate the performance of trusted executing environment provided by TrustOSV. By running multiple benchmarks and comparing the performance overhead with the same configuration guests from original OSV hypervisor and commodity hypervisors like KVM (as well as native Linux), we show TrustOSV has modest performance overhead.

The evaluation environment is based on a Dell R715 server. The machine is equipped with two 2GHz 8-core AMD Opteron 6128 processors with SVM extension, NPT support and 32GB physical memory. The network card is Broadcom NetXtreme 5722. We use an Ubuntu 10.04 as the management OS, and boot multiple Linux 3.1.5 kernel to serve for the trusted computing environment.

1) Micro-benchmarks: First, we measure the verifying procedure of TrustOSV. Then we will show several system call latencies of the executing environment and compare it with a same configured KVM guest.

a) measurement cost: : TrustOSV provides integrity proofs of the underlying system information. After secure loading the kernel into memory, TrustOSV measures the code memory pages and the guest images with the in-memory attestor. We measure both cases, that is with/without TrustOSV self memory lock and attestation on platform, and find the attestation procedure adds no noticeable latency. In our experiment, we found it took $\tilde{1}5$ seconds from system start till the login screen for both the cases. To be precise, the time used by SHA-1 hash function for kernel images and underlying VMM's static code and data checksum is quite short ($\tilde{0}.9$ seconds), which proves that the attestor is applicable to use in real cloud.

b) Lmbench: : Table IV displays the OS-related results of lmbench benchmark on TrustOSV guest, KVM guest and native Linux. Except for `Bcopy` (block memory copy) test, all test cases treat smaller value as better results. Our measurements indicate that both hypervisors have some performance degradation compared to native Linux. For example, KVM has 12.2% performance loss in `ctxsw` test (context switch among 16 processes, each 64 KB in size) and TrustOSV has 6.6% performance loss in `Mmap Latency` (latency to map a file into a process). TrustOSV has greater performance loss on I/O intensive operations. This is because TrustOSV replace I/O emulation by using distributed protocols like NFS, in order to remove VM_EXIT events from I/O emulation.

It should be noted that TrustOSV has no distinct performance drop for `fork, exec, double operations` test suits even compared to native Linux. Because we pre-allocate the CPU cores to each guest environment, and pre-fill the NPT items of each guest, there will be no need for hypervisor intervention.

2) Overall performance: In this part, we will demonstrate the runtime efficiency for an isolated computing environment by running SPECint 2006 benchmark suite

TABLE IV: Lmbench Latency Results on Native Linux, KVM, OSV, TrustOSV with native execution, and TrustOSV with data encryption enabled

Config.	**Native Linux**	**KVM**	**OSV**	**TrustOSV**	
				native	data encryption
Null (ms)	0.08	0.08	0.09	0.09	0.16
Fork Process (ms)	206	216	204	217	221
Exec Process (ms)	626	779	611	788	791
Ctxsw(16p/64K, ms)	8.09	9.08	11.2	11.1	14.3
Mmap Latency (ms)	12600	13414	14500	15700	15100
Main mem (ns)	48.2	95.1	48.2	48.2	48.2
Double add (ns)	2	2.01	2	2	2
Double div (ns)	11.4	11.4	11.3	11.4	11.4
Bcopy(libc) (MB/s)[a]	3320.1	2129.1	3740.3	2347	2120.3

[a] Larger is better in BCopy test case, while for other test cases smaller is better.

[37]. We do our experiment and compare the result with original OSV guest as well as a KVM guest with same configuration. Each guest is configured with four cores and 4GB memory.

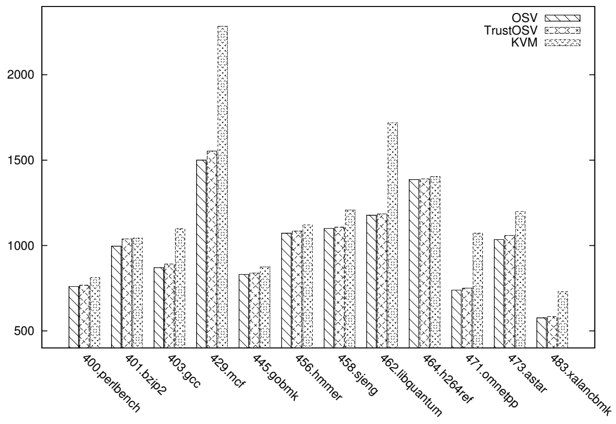

Fig. 5: SPECint2006 performance result of TrustOSV against KVM, lower is better

Shown in Figure 5 are the results of our experiments. The overall performance of TrustOSV is comparable with original OSV hypervisor. And the slight performance loss (on average 3%) is mainly deal to latency when test data is loaded through the stackable file system layer.

Compared with KVM guest, we saw a range from 1% to 30% performance improvement across the board. The `mcf`, `libquantum`, and `omnetpp` test cases have almost 30% performance improvement. We contribute the performance improvement to the removal of VM_EXIT by using SVM and NPT hardware.

VII. Discussions and Future Works

The setup of trusted executing environment depends on the TrustOSV hypervisor, and we add the desired security by removing the *hypervisor-guest interaction.* As a result, the flexibility of the virtualization is removed, which means we cannot virtualize more resource than the actual physical resource. However, we believe the TrustOSV system can still have applying space for the following reasons: 1) users pay the money for computing resources, therefore they should have the corresponding resources. That is what cloud users expect to receive and not the unpredictable performance and extra side channels security threats. 2) TrustOSV provides users with strict resource isolation and better trustworthiness of the running computing environment. Therefore, the cloud providers could charge for extra fee for VIP users who want to receive a more trustworthy computing environment service. In particular, real-time cloud based systems with security requirement, such as secure smart grid system [38] or real-time wireless base station infrastructures [39], can greatly benefit from TrustOSV design.

Although TrustOSV system has demonstrated the feasibility of providing trustworthy executing environment for remote cloud users, there is still exploring space behind its implementation. Next, we discuss several possible extensions to TrustOSV and list our future works.

Verification: Most of the TrustOSV system is built on a tiny micro-hypervisor, which provide the essential resource isolation utility. The code base of the micro-hypervisor is quite small. Thus it is possible to perform a formal verification [18] of our system design. For example, to guarantee the design requirement functions equal to system implementation.

Transparent security sensitive workload protection: In our paper, we provide an isolated executing environment to protect security-sensitive applications in cloud. Existing works [40], [41] attempted to use code insertion to protect part of the sensitive code, while it still needs lots of manual interactions. We propose to build transparent workload protection schema on TrustOSV without any extra manual operations.

VIII. Related Works

The related work can be classified into the following three areas: the micro-hypervisor system, enforcing the VMM and securing the cloud, and splitting large of hardware resource.

The micro hypervisor systems: Apart from the commonly used hypervisors such as Xen, KVM, VMware, there still exist other kinds of micro-hypervisor for special uses. SecVisor [42] is a small hypervisor for preventing operating system executes malicious user level code, and it provides a novel way to prevent kernel-level rootkit. BitVisor [43] is a hypervisor that intercepts device I/O to implement OS-transparent data encryption and intrusion detection. TrustVisor [40] uses a small hypervisor to isolate part of the program execution, and it aims to protect a running program from compromised by any malicious software (even including OS) on the same platform. However, the micro-hypervisor systems mentioned above aim at protecting operating systems (or single process) from compromised by malicious OS users, thus they do not meet the design goals of TrustOSV.

NOVA system [44] replaces commodity hypervisor with a small hypervisor (around 9 KLOC) in order to minimize the TCB. However, NOVA is still responsible for several management tasks (e.g. address space management, processor scheduling). Thus it still leaves potential security vulnerabilities for safe cloud executing environment.

Enforcing the VMM and securing the cloud: Several attempts focus on enforcing the hypervisor to secure the cloud [45], [46], [47], [20], [21], [48]. One is CloudVisor system [45], CloudVisor leverages nested virtualization technology to host Xen hypervisor and multiple Guest OSes on a tiny hypervisor. And the tiny hypervisor will intercept each VM_EXIT event and check exist reasons before passing it to Xen. Although CloudVisor also aims at preventing malicious cloud operator from stealing cloud user's privacy contents, the frequent interception on commodity VMM degrades the overall performance. Moreover, as an introduced layer, the security of CloudVisor itself is not guaranteed (as we lock the memory pages in TrustOSV).

NoHype [46] eliminates the interaction between Guest OS and Xen hypervisor, but it still leaves Host OS (Xen's Dom0) interacting with hypervisor. While actually, in a real cloud environment, the host may still be compromised and becomes one important security concern [2], [45]. In TrustOSV system, all operating systems run symmetrically on a micro-hypervisor. In other words, except for a few of management commands (i.e. start an isolated environment command), the management OS can also avoid interacting with hypervisor. At the same time, TrustOSV system also provides memory-locked platform attestation, as well as permanent data secrecy for permanence storage, which are not appeared in NoHype design.

More recently, researchers also proposed a DEHYPE system to de-privilege the hypervisor [47]. By moving most hypervisor code into user mode and requesting privileged service with a *HypeLet* layer [47], DEHYPE system can remarkably reduce system TCB. However, as a privileged code piece in host OS, HypeLet is still confronted with security vulnerabilities and thus lead to a privilege escape attack.

Splitting large hardware resource into smaller parts: Another related work is the massive resource management in large-scale heterogeneous computing environment. FOS [49] is an operating system for cloud environment, it manages multi-core platform to provide efficient cloud service. Another prototype is multi-kernel system sponsored by Microsoft Barrelfish project [50]. But they all focus on heterogeneous hardware management, while our system focuses on partitioning hardware resource for a secure executing environment.

IX. Conclusion

Nowadays, virtualization technology plays an important role in cloud computing. However, virtualization on a hardware-sharing platform has become a major security concern. Thus the executing environment of cloud user has become a potential target for attackers. It is emergent to provide security sensitive workload with a trustworthy executing environment. In this paper, we present an architecture which aims to provide cloud users a safe isolated executing environment, and supply cloud users integrity proof on their executing environments. We achieve our goals by leveraging the VM_EXIT avoided resource allocation, application transparent data encryption, as well as platform attestation to remote users. At last, a prototype system is built to demonstrate the feasibility of our design architecture, and the experiments show that it has optimistic performance.

Acknowledgment

We would like to thank Pilar Howard, Cancan Wang and all other anonymous reviewers for their comments and suggestions.

References

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[2] Vaquero, Luis M., Luis Rodero-Merino, and Daniel Morlén. Locking the sky: a survey on iaas cloud security. *Computing*, 91(1):93–118, 2011.

[3] K. Kortchinsky. Cloudburst: Hacking 3d (and breaking out of vmware. *Black Hat USA*, page 6, 2009.

[4] R. Wojtczuk, J. Rutkowska, and A. Tereshkin. Xen 0wning trilogy. In *Black Hat conference*, 2008.

[5] N. Elhage. Virtualization under attack: Breaking out of kvm. *DEF CON*, 19.

[6] Secunia.com. Xen multiple vulnerability report. http://secunia.com/advisories/44502/, Accessed: Nov-16-2013.

[7] Secunia.com. Vulnerability report: Vmware esx server 3.x. http://secunia.com/advisories/product/10757/, Accessed: Nov-16-2013.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[9] J. Sugerman, G. Venkitachalam, and B.H. Lim. Virtualizing i/o devices on vmware workstationąŕs hosted virtual machine monitor. In *USENIX Annual Technical Conference*, pages 1–14, 2001.

[10] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[11] Samuel T. King and Peter M. Chen. Subvirt: Implementing malware with virtual machines. In *2006 Security and Privacy*, pages 14–pp. IEEE, 2006.

[12] Mohammed A AlZain, Ben Soh, and Eric Pardede. A survey on data security issues in cloud computing: From single to multi-clouds. *Journal of Software (1796217X)*, 8(5), 2013.

[13] Techspot. http://www.techspot.com/news/40280-google-fired-employees-for-breaching-user-privacy.html, Accessed: Nov-16-2013.

[14] W. Zhou, P. Ning, X. Zhang, G. Ammons, R. Wang, and V. Bala. Always up-to-date: scalable offline patching of vm images in a compute cloud. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 377–386. ACM, 2010.

[15] xen.org. Submitting xen patches. http://wiki.xen.org/wiki/Submitting_Xen_Patches, Accessed: Nov-16-2013.

[16] InformationWeek.com. 10 massive security breaches. http://www.informationweek.com/security/attacks/10-massive-security-breaches/229300675, Accessed: Nov-16-2013.

[17] S. Sengupta, V. Kaulgud, and V.S. Sharma. Cloud computing security–trends and research directions. In *Services (SERVICES), 2011 IEEE World Congress on*, pages 524–531. IEEE, 2011.

[18] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[19] G. Heiser and B. Leslie. The okl4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, pages 19–24. ACM, 2010.

[20] Jakub Szefer and Ruby B Lee. Architectural support for hypervisor-secure virtualization. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 437–450. ACM, 2012.

[21] Y. Xia, Y. Liu, and H. Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *Proceedings of The 19th IEEE International Symposium on High Performance Computer Architecture*. IEEE, 2013.

[22] A.M. Devices. Amd64 architecture programmerąŕs manual volume 2: System programming, 2006.

[23] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), 2006.

[24] Erez Zadok and Ion Badulescu. A stackable file system interface for linux. In *LinuxExpo Conference Proceedings*, volume 94, 1999.

[25] F. Bellard. Qemu, a fast and portable dynamic translator. USENIX, 2005.

[26] nvd.nist.gov. National vulnerability database. http://nvd.nist.gov/, Accessed: Nov-16-2013.

[27] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 1–16. ACM, 2005.

[28] S. Bugiel, S. NÍźrnberger, T. Poppelmann, A. R. Sadeghi, and T. Schneider. Amazonia: when elasticity snaps back. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 389–400. ACM, 2011.

[29] J. Ren, Y. Qi, Y. Dai, and Y. Xuan. Inter-domain communication mechanism design and implementation for high performance. In *Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on*, pages 272–276. IEEE, 2011.

[30] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 380–395. IEEE, 2010.

[31] Trusted boot – open source software. http://tboot.sourceforge.net/, Accessed: Nov-16-2013.

[32] Peiqiang Chen. Software behavior based trustworthiness attestation for computing platform. *Journal of Software (1796217X)*, 7(1), 2012.

[33] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[34] Yuehua Dai, Yong Qi, Jianbao Ren, Yi Shi, Xiaoguang Wang, and Xuan Yu. A lightweight vmm on many core for high performance computing. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 111–120. ACM, 2013.

[35] ecryptfs.org. ecryptfs. http://ecryptfs.org/, Accessed: Nov-16-2013.

[36] D.A. Wheeler. Sloccount, http://www.dwheeler.com/sloccount. Technical report, Accessed: Nov-16-2013.

[37] John L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[38] Hairong Qi, Xiaorui Wang, Leon M Tolbert, Fangxing Li, Fang Z Peng, Peng Ning, and Massoud Amin. A resilient real-time system design for a secure and reconfigurable power grid. *Smart Grid, IEEE Transactions on*, 2(4):770–781, 2011.

[39] Anna Tzanakaki, Markos P Anastasopoulos, Georgios S Zervas, Bijan Rahimzadeh Rofoee, Reza Nejabati, and Dimitra Simeonidou. Virtualization of heterogeneous wireless-optical network and it infrastructures in support of cloud and mobile cloud services. *Communications Magazine, IEEE*, 51(8), 2013.

[40] J.M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *2010 IEEE Symposium on Security and Privacy*, pages 143–158. IEEE, 2010.

[41] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 279–292. USENIX Association, 2006.

[42] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 335–350. ACM, 2007.

[43] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, et al. Bitvisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130. ACM, 2009.

[44] U. Steinberg and B. Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222. ACM, 2010.

[45] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, 2011.

[46] E. Keller, J. Szefer, J. Rexford, and R.B. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 350–361. ACM, 2010.

[47] Chiachih Wu, Zhi Wang, and Xuxian Jiang. Taming hosted hypervisors with (mostly) deprivileged execution. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[48] Louise E Moser, P Michael Melliar-Smith, and Wenbing Zhao. Building dependable and secure web services. *Journal of Software (1796217X)*, 2(1), 2007.

[49] D. Wentzlaff, C. Gruenwald III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An operating system for multicore and clouds: mechanisms and implementation. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 3–14. ACM, 2010.

[50] A. Baumann, P. Barham, P.E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.

Xiaoguang Wang received his B.S. degree from Northwestern Polytechnical University, China in 2010. He is currently working towards his Ph.D. degree in computer science at Xi'an Jiaotong University, China. His current research interest includes operating systems, VMM, cloud computing and system security.

Yi Shi received her PhD in computer software and theory from Xi'an Jiaotong University in 2008. She is a lecturer in the School of Electronic and Information Engineering, Xi'an Jiaotong University. Her research interests include operating system, network security and cloud computing.

Yuehua Dai received his B.S. degree from Xi'an Jiaotong University in 2004. He is currently a PhD candidate in computer science at Xi'an Jiaotong University. His research interests include operating systems, VMM, and system security.

Yong Qi received his PhD degree in computer software and theory from Xi'an Jiaotong University in 2001. He is currently a professor in the School of Electronic and Information Engineering, Xi'an Jiaotong University and the director of the Institute of Computer Software and Theory. His research interests include operating system, distributed system, pervasive computing, and security in cloud computing.

Jiaobao Ren received his B.S. degree from Xi'an Jiaotong University in 2009. He is currently a PhD candidate in computer science at Xi'an Jiaotong University. His research interests include operating systems, VMM, cloud computing and system security.

Yu Xuan received his B.S. degree in computer software and theory from Xi'an Jiaotong University in 2011. He is currently a Master student in computer science at Xi'an Jiaotong University.