Hardware-Assisted Secure Resource Accounting under a Vulnerable Hypervisor

Seongwook Jin, Jinho Seol, Jaehyuk Huh, and Seungryoul Maeng

Computer Science Department, KAIST

{swjin, jhseol, jhuh and maeng}@calab.kaist.ac.kr

Abstract

With the proliferation of cloud computing to outsource computation in remote servers, the accountability of computational resources has emerged as an important new challenge for both cloud users and providers. Among the cloud resources, CPU and memory are difficult to verify their actual allocation, since the current virtualization techniques attempt to hide the discrepancy between physical and virtual allocations for the two resources. This paper proposes an online verifiable resource accounting technique for CPU and memory allocation for cloud computing. Unlike prior approaches for cloud resource accounting, the proposed accounting mechanism, called Hardware-assisted Resource Accounting (HRA), uses the hardware support for system management mode (SMM) and virtualization to provide secure resource accounting, even if the hypervisor is compromised. Using a secure isolated execution support of SMM, this study investigates two aspects of verifiable resource accounting for cloud systems. First, this paper presents how the hardware-assisted SMM and virtualization techniques can be used to implement the secure resource accounting mechanism even under a compromised hypervisor. Second, the paper investigates a sample-based resource accounting technique to minimize performance overheads. Using a statistical random sampling method, the technique estimates the overall CPU and memory allocation status with 99% $\sim$ 100% accuracies and performance degradations of 0.1% $\sim$ 0.5%.

Categories and Subject Descriptors D.4.8 [*Operating Systems*]: Performance

Keywords resource accounting; cloud; virtualization

VEE '15, March 14–15, 2015, Istanbul, Turkey.
Copyright © 2015 ACM 978-1-4503-3450-1/15/03. . . $15.00.
http://dx.doi.org/10.1145/2731186.2731203

1. Introduction

Although cloud computing provides elastic computing resources based upon service contracts between cloud users and providers, one of the critical concerns for such a utility-based computing model is whether the quantity of computing resources can be guaranteed by the cloud providers. Some cloud providers may choose to provide a fixed amount of resources, regardless of whether cloud users actually consume them or not. A different cost model may charge the cost based on the actual usage of resources. In both cost models, the accurate accounting of resources is critical for cloud computing, to provide fair and verifiable costs for outsourced computation.

Considering the importance of accountable resource allocation, cloud providers make their best effort to support the availability of the computing resources mandated by the service level agreement (SLA). However, compromised system administrators or remote attackers can potentially reduce the computing resources assigned for valid users, and redirect the stolen resources for their own benefit. Most of the current cloud computing systems employ virtualization, and a hypervisor in each physical system is responsible for allocating computing resources to each user VM. However, a compromised hypervisor or compromised administrator can assign an arbitrary amount of computing resources to user VMs, violating SLA. Furthermore, a malicious user may exploit the system vulnerability to steal computing resources from co-tenants.

Among cloud resources, explicit and coarse-grained resources such as a virtual machine (VM) itself or I/O operations can be relatively easily traced by users or guest OSes without significant performance overheads. However, CPU time shares or memory pages allocated for user virtual machines are very difficult to track its availability as cloud users can only observe virtual CPUs and memory, hiding actual allocation of physical resources. An alternative way to account such resources is to occasionally run benchmark applications to verify the resources based on performance outcome of benchmark runs. However, such a benchmark-based approach cannot account fine-grained resource allocation changes accurately. Furthermore, the performance of a benchmark application can be affected by various valid dynamic events including interference with co-tenants.

This paper proposes a resource accounting technique, called HRA (Hardware-assisted Resource Accounting) for CPU time shares and memory pages, even if a hypervisor or individual system administrator is compromised. To detect any violation of resource allocation for CPU time shares and memory pages, instead of running a benchmark application, this study uses a sample-based approach to check the status of CPU and memory allocation by random probing. The probing program, which is protected from the hypervisor, checks the CPU allocation status and the number of memory pages with random time intervals. The execution of such random probing mechanism must be isolated from a potentially compromised hypervisor, and its sampled execution must not be preempted by the hypervisor. We use the system management mode (SMM) for probing execution, which provides a higher privileged execution environment than the hypervisor execution as shown in Figure 1. The probing code checks CPU and memory status randomly in SMM, without any interference from the hypervisor.

This paper investigates two aspects of resource accounting under a vulnerable hypervisor. First, this paper discusses our probing mechanism based on SMM, which is implemented on a commodity system. We show that the probing mechanism incurs little performance interference, while providing accurate resource allocation status. Even system administrators cannot affect the execution of probing program in SMM unless they have accesses to BIOS, which requires rebooting of the system. The integrity of BIOS can be assured by remote attestation based on TPM (Trusted Platform Module). Even if a remote attacker acquires root permission to the hypervisor or management VM, the attacker cannot intervene on the SMM-based probing execution.

Second, we investigate the trade-off between sampling rates and accuracies with statistical modeling to prove that the proposed technique can verify CPU and memory resources with negligible performance impact. CPU and memory status is checked randomly, and by statistical inference, the total CPU time and memory amount are estimated. As probing occurs more frequently, the accuracy of estimation further increases. However, performance overheads can increase to run the probe program frequently. Using a statistical analysis and experimental evaluation on a real system, we show that the proposed sample-based approach does not cause significant performance overheads, while maintaining a high accuracy.

To the best of our knowledge, this is one of the first studies to verify cloud CPU and memory resources in an unobtrusive way, even under a vulnerable hypervisor. Although prior studies discussed the importance and potential challenges for resource accounting in cloud computing [7, 10, 15, 17, 23], solutions require a secure hypervisor and system administrator. However, there have been successful attacks on the security of hypervisors, and furthermore resource appropriation attacks even without compromising the hypervisor [28, 34]. In addition, prior studies incur some overheads due to an additional software layer [7]. Our system can report the accurate CPU and memory accounting for such cases with little performance overheads. Our experimental analysis shows that the proposed system can detect the CPU time share and memory allocation with accuracies of 99% $\sim$ 100% with performance losses of 0.1% $\sim$ 0.5%.

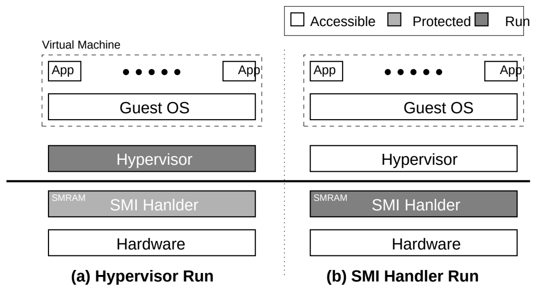

Figure 1: TCB of the proposed HRA system

The rest of the paper is organized as follows. Section 2 presents the motivation for resource verification in cloud computing. Section 3 presents the background for system management mode (SMM). Section 4 discusses our sample-based probing technique and statistical analysis. Section 5 describes the architecture and implementation of the verification mechanism with SMM. Section 6 shows experimental results. Section 7 discusses prior work, and Section 8 concludes the paper.

2. Motivation

2.1 Cloud Security and Hypervisor Vulnerability

Along with the recent rapid growth in cloud computing, there have been increasing concerns over the security of cloud computing. Although cloud providers may make their best effort to keep the private data protected from remote attackers, cloud computing with multi-tenancy of sharing physical systems among cloud users can have an inherent potential vulnerability, which did not exist in in-house servers. In most of the commercial cloud services, cloud users can access only virtual machines (VMs), and the hypervisor provides an isolated execution environment for each VM. Furthermore, the hypervisor is responsible for allocating resources to VMs as mandated by SLA. Even though the hypervisor is critical for the security of virtualized clouds, hypervisor vulnerabilities have been growing gradually with its increasing code size for better performance and more complex functionality [16, 21, 22]. With such growing vulnerabilities of hypervisors, there have been several reports of attack cases [30, 31]. Furthermore, since the root-permission to the hypervisor or management domain (dom0 in Xen) can potentially provide unlimited accesses to user data in memory or resource allocation, a compromised system administrator can also have a full control over guest VMs.

To mitigate the security problem of hypervisors, there have several recent studies for protecting VMs. In SW-based approaches, support for nested virtualization adds an extra layer of virtualization to perform security-critical functions in the secure bottom-layer of virtualization [11]. An alternative design is to add a hardware-based security layer to provide such security functions [13]. However, in commercial systems, such proposals to improve the security of clouds with hypervisor vulnerabilities have yet to be adopted.

Most of the recent studies for hypervisor security have been focused on providing confidentiality and integrity of user data. However, a neglected aspect of security is the availability of resources. Even in the prior studies for securing hypervisors, the availability issue has not been addressed, since resource management still remains as an important role of hypervisors, and thus cannot be separated effectively from the hypervisors. In addition, the malicious administrator can assign an arbitrary amount of computing resources to a VM, by simply changing the resource allocation policy without breaking the hypervisor integrity.

2.2 Verifiable Resource Accounting

In virtualized clouds, users can only manage virtual CPUs and memory without direct accesses to the allocation of physical resources for their VMs. A VM has an illusion of having a contiguous virtual DRAM and CPU, but the hypervisor may dynamically change VM memory pages or schedule virtual CPUs, hiding the actual allocation of memory pages and CPU shares from guest VMs. Due to the virtualization of CPUs and memory, cloud users cannot directly account the CPU and memory allocation performed by the hypervisor. Since such CPU and memory usage information can only be reported by the potentially vulnerable hypervisor or administrator, the current cloud provider cannot provide secure user verification of resource accounting.

An alternative way to verify the CPU and memory allocation is to occasionally run a test benchmark application to measure the performance. The benchmark may infer any discrepancy between the reported resources and allocated resources based on the difference between the expected and measured performances. To verify the resource allocation, guest users must run the benchmark application occasionally. A major drawback of this benchmark approach is the difficulty of accounting fine-grained resource allocation changes. Furthermore, the performance of a benchmark application can be affected by various dynamic events including interference with co-tenants. Accurately inferring CPU usage or allocated physical memory indirectly from performance is not trivial. The second drawback of the benchmark approach is that a compromised hypervisor may detect the launch of the known test benchmark or even disable the test runs entirely, since it can access the memory of target VMs and modify the scheduled benchmark execution. This drawback can potentially abrupt the entire accounting mechanism by cloud users.

Another way to verify the allocated resources directly is to track application performance continuously. This type of application performance tracking can be used to a certain SLA type, which guarantees a maximum response time for some server workloads. For example, web servers running in clouds may have a target worst-case response time, and the cloud provider must meet the requirement with a very low rate of outliers. However, this type of SLA can be applied, only when the workloads are known a priori and their response time can be defined clearly. Commonly, it can be more easily applicable to platform-as-a-service (PaaS) clouds, than to infrastructure-as-a-service (IaaS). Due to the limited scope of the application-driven performance SLA, our system focuses on the resource-driven SLA, which verifies the amount of resources allocated to user VMs.

2.3 Threat Model

With various vulnerabilities [20, 21, 30, 31], a hypervisor can become malicious by external attacks or malicious insider. A malicious hypervisor or administrator can easily modify or access the private data of VMs and enforce an arbitrary resource management policy to allocate fewer resources to the guest VMs. In protecting guest VMs from these kinds of resource attacks, one of the most important components is to build a safe and privileged layer from the malicious hypervisor. The proposed HRA mechanism uses SMM as a higher privilege layer than the hypervisor execution. With SMM, HRA can exclude the hypervisor from the trusted computing base (TCB) and TCB includes only the hardware and BIOS loading the probing program. Since the system cannot guarantee the integrity of the proposed mechanism with a compromised BIOS, HRA uses Trusted Platform Module (TPM) [27] to verify the BIOS by writing the integrity information of the BIOS into Platform Configuration Register (PCR) when the system boots. With the TPM-based attestation, the client can rely on remote attestation to verify the proposed mechanism before a probing process.

To accelerate virtualization, current hardware manufactures developed hardware-assisted virtualization such as VT-x [12] and AMD-V [1], and most cloud providers use it for better performance. The proposed mechanism assumes that the hardware-assisted virtualization is available. The cloud providers manage all of the physical machine components including even the proposed mechanism. We assume that cloud providers, as an enterprise, are trustworthy, and they make their best effort to protect user VMs. Once provider attacks are detected, it has a fatal influence on their reputation and business. In opposition to the providers, some of individual administrators can be malicious. Each administrator has enough power to compromise the hypervisor of a subset of systems. Such malicious administrators can deallocate the resources by controlling the vulnerable hypervisor, allowing resource attacks even without altering hypervisor binaries. The administrators can steal the resources of VMs

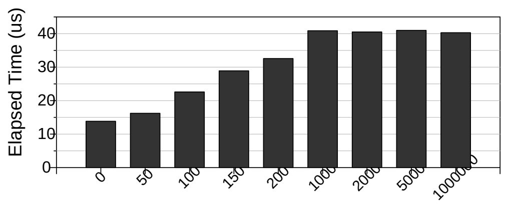

Figure 2: The Cost of Invoking SMM with intervals

by just altering VM configurations such as CPU allocation time or total memory.

However, the proposed mechanism does not provide memory protection to VMs. The previous studies [3, 4, 11, 13] can protect VMs against the malicious hypervisor for guest memory protection, and the proposed mechanism can be combined with them. The prior work for VM security protects the guest VM memory by adding an extra layer of hypervisor to perform security-critical functions, or requires a new architectural support for HW-based security operations. However, this work is based on the currently available hardware support for SMM and virtualization, and thus does not rely on an extra SW hypervisor, which can also be vulnerable and controlled by the administrator. However, we do not consider hardware attacks such as accessing DRAM after power-off or probing external buses, assuming the hardware system is protected by the cloud providers.

3. System Management Mode

To implement the proposed verifiable accounting mechanism under a vulnerable hypervisor, we use SMM to execute a resource probing program in an isolated execution environment protected even from the hypervisor. SM M [1, 12] is a special execution mode designed for processing maintenance functions in urgent circumstances. When the system software cannot handle abnormal system events such as high temperatures or hardware errors, SMM is invoked and execute the handlers for the abnormal events.

The x86 architectures use a special interrupt-driven mechanism to enter SMM. When a special interrupt called System Management Interrupt (SMI) is issued by predefined events or external requests, the current processor state is automatically saved in System Management RAM (SMRAM), part of memory configured for storing SMM states. SMRAM is inaccessible in any other modes except SMM. To process an SMI, a processor switches the current running task to the SMI handler located in SMRAM. After completing the SMI handling, the processor restores the saved processor state in processor registers by using a special instruction called *RSM* to resume the prior task.

SMRAM is normally configured by BIOS before the system software boots. After configured, SMRAM cannot be changed or disabled, protecting SMRAM from any software including the hypervisor which runs in a non-SMM mode. The system software cannot preempt the execution of SMM, until SMM releases the current SMI handling, because all incoming interrupts or exceptions are pending in SMM. SMM is similar to a real mode which has full accesses to hardware resources. SMM can access the main memory and hardware devices with I/O port, being able to access even the memory area where the hypervisor and kernel are located. The proposed probing program resides in SMRAM protected from the hypervisor, and the hypervisor cannot know when the probing program is invoked.

The current SMM is designed for supporting management tasks such as power management and system hardware control tasks. These types of tasks require making all the current running tasks frozen. In addition to the lack of ability to control each core individually, there are several limitations in the current SMM design to consider for the implementation of the proposed secure accounting mechanism.

4GB Address Space Limit: SMM has privileged access permission to the main memory but it is restricted to $0 \sim 4$GB memory space. Accessing the area beyond the 4GB limit leads to an undefined system behavior. To address this limitation in our implementation, the SMI handler modifies the SMM state-save area, replacing the saved context with stub codes. The stub codes are in charge of accessing the high memory region above the 4GB address limit, and run in the normal mode. Even if the stub code is running in the normal mode, all interrupts are disabled at the first SMI handling, so the stub code execution cannot be interrupted. After the execution of stub code for accessing the high memory region, the execution comes back to the SMI handler to re-enable interrupts.

All Cores SMM invocation: When an SMI occurs, all of the cores in a system must enter SMM synchronously. The HRA system checks the status of all cores simultaneously for resource verification operations to enter SMM together in all cores. However, some management operations such as creating and deleting VMs in our design need only one core to run the SMI handler, but the rest of cores also have to enter SMM due to the hardware restriction. This limited design causes performance degradations, but such management operations are very infrequent compared to resource probing operations.

Flushing write-back caches: An indirect performance overhead of entering SMM is the cost for flushing modified cache blocks. Before entering SMM, the modified write-back data in all the caches must be flushed to the main memory for coherence. To assess the performance overhead, Figure 2 shows the cost of invoking SMM with various intervals. To isolate the effect of cache flushing, the SMI handler immediately calls *RSM* instruction to exit as soon as entering SMM in this experiment. When SMIs are consecutively issued without intervals, it takes about 14μs per SMI. However, as the interval increases, the SMI handling time increases significantly to 40μs, even for the same operation. The reason for the increasing latency is that with a longer

Resource	Time	Space
CPU	Allocation Time ✓	Number of CPU ✓
Memory	Bandwidth	Total Size ✓
Network	Latency, Throughput	Total Traffic
Storage	Latency, Throughput	Total Size

HRA verifies checked resources

Table 1: Resource Classification

interval until 1000 us, more cache blocks become modified, and the flushing operation takes a longer latency of 40μs. However, as the cache capacity is limited, the cost for flushing dirty blocks does not increase further beyond 40μs.

4. Sample-based Resource Accounting

This section describes our sample-based resource accounting mechanism for CPU and memory resources. To reduce the overhead of invoking SMM for probing resource allocation status, sampling rates should be reduced while a high accuracy is maintained. In this section, we describe a random sampling method, and analyze the accuracy and sampling rate trade-offs in the method.

4.1 Verifiable Resources

In cloud systems, all the resources are virtualized with time and space sharing by the hypervisor, although each guest VM has an illusion of dedicated resources. Time sharing is a virtualization of a resource by multiplexing virtual resource requests into a shared physical resource with fine-grained scheduling. Space sharing partitions resources in space and assigns each partition to a VM. Table 1 presents a classification of cloud resources and how they can be shared either by time and space partitioning. Among the resources with different sharing types, this paper focuses on two resources, which are difficult to account with the current virtualization techniques, CPU and memory resources.

CPU Resources: A hypervisor creates and schedules virtual CPUs (vCPU) for VMs, multiplexing vCPUs to limited physical cores by scheduling. The vCPUs share physical cores by time and space sharing, but the guest OS in each VM observes only the vCPUs without any direct knowledge of scheduling. Due to such CPU virtualization, a cloud user or guest OS is hard to identify the actual amount of CPU allocation by the hypervisor. Without any reliable hardware support in the current processors which cannot be altered by the hypervisor, the hypervisor can provide incorrect accounting information to the guest OS about CPU scheduling.

Memory Resources: The entire available memory is partitioned and allocated to VMs with space-sharing. To virtualize the memory, the current virtualization technique uses an additional address translation from guest physical address to machine address, for a contiguous guest-physical memory to each VM. The hypervisor can dynamically adjust the allocated memory pages of a VM by memory swapping without any permission from guest OSes. However, the hypervisor cannot directly adjust memory bandwidth due to the lack of HW interface for bandwidth scheduling, even if its privilege level is high enough. Furthermore, common SLAs for clouds mandate only the capacity of memory allocation, without any guarantee on memory bandwidth due to the same lack of architectural support for adjusting memory bandwidth for a particular VM. In this paper, the proposed HRA mechanism reports the capacity of memory assigned for each VM.

I/O Resources: Unlike CPU or memory, I/O operations can be counted by the guest OS, since the guest VM generates I/O operation requested explicitly by the guest OS system call. Therefore, accounting I/O operations is relatively straightforward, unlike CPU and memory hidden by virtualization. However, one key problem can be the protection of accounting data from the potentially malicious hypervisor. The verifiable accounting system may provide an interface to guest VMs to record I/O operations securely. However, this paper focuses only on CPU and memory accounting, leaving the secure recording interface as future work.

4.2 Random Sampling based Resource Accounting

Our accounting system resides in SMM protected from the hypervisor. However, it cannot trace every context switch for CPUs or every allocation and deallocation of memory pages for two reasons. First, the accounting system in SMM is self-initiated not to be interfered by the hypervisor. In the current system architecture, system management interrupts (SMI) cannot be generated automatically by the hardware system for each context switch or memory page change. Therefore, the SMM-based accounting system should be launched by time-based scheduling to check the allocation status. Second, although the accounting program can quickly check the allocation status, there are non-negligible overheads for invoking SMM. Due to the overheads, there is a trade-off between sampling rates and accuracies for accounting. In this study, instead of tracking every context switch or page change, we use a sample-based accounting mechanism. The probing program is initiated by an SMI with random intervals, and checks the allocation status. In this section, we will discuss and analyze the random sampling method with an analytical model and simulation.

In this section, to simulate the effect of sampling with accuracy-overhead trade-offs, we use a scheduler simulator for examining estimation of the CPU resources under a potentially malicious hypervisor. The simulator emulates the Xen Credit Scheduler [32]. The simulator uses micro second as the minimum time unit and simulates vCPU scheduling based on VM traces collected from busy-waiting applications. To mimic the behavior of a malicious hypervisor, the simulator models that the hypervisor can force a context switch from a running VM any time. In real execution scenarios, the hypervisor can steal CPU resources after voluntary or forced VM exits any time. For example, even though CPU intensive applications are processed, the VM frequently exits and needs hypervisor interceptions for sys-

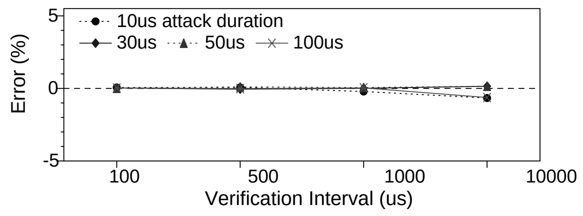

Figure 3: The Random Sampling Results on Simulation

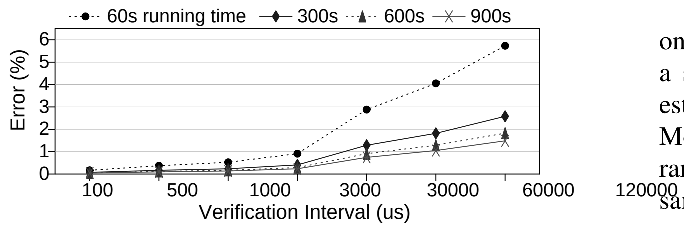

Figure 4: The Error on Various Running Time

tem calls and privileged executions. The malicious hypervisor can schedule the attacker VM as soon as the victim VM exits. Even exploiting Inter-Processor Interrupt (IPI) or timer interrupts, the hypervisor forces the victim VM to exit to schedule the attacker VM on any time. We have emulated such capability of the hypervisor stealing the CPU resources into the simulator.

Figure 3 shows the errors of random sampling on stealing 10% the CPU resources from the victim. The errors represent differences between the actual CPU allocation and one estimated by the sampling. In theory without considering the overheads of frequent context switches, the hypervisor may attempt to steal CPU resources with a very short duration not to be detected. To consider such extreme cases, fine-grained attack durations from 10μs to 100μs are selected in the experiment. In the figure, each curves shows different attack durations of the range. The x-axis represents different average verification intervals used by our random sampling method.

In the result, the random sampling correctly detects the hypervisor attack with the average 100μs verification interval. Even if the interval becomes longer, the random sampling generates relatively accurate results within 1% error. The error becomes as large as 1% in 10ms interval and 10μ attack duration, because the number of samples is not enough to generate correct results.

4.3 Statistical Analysis

With random sampling, it is critical to know the number of samples to have a certain level of confidence for the accuracy of estimated results. Figure 4 shows the number of samples must be large enough to produce correct CPU usage. In this figure, the x-axis represents different average verification intervals used by the probing program. Each curves shows different total execution times from 60 to 900 seconds. The error of 120000μs sample interval is as large as 6% with 60s, since with the 120000μs interval, it generates

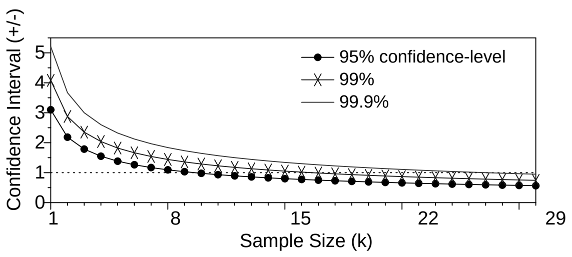

Figure 5: CPU Confidence Interval

only 500 samples during the 60s running time. With such a small number of samples, the random sampling cannot estimate CPU usages accurately. In this section, we use the Monte Carlo method for modeling the confidence of the random sampling and for inferring the sufficient quantity of samples [18].

First, we define an averaged CPU usage as the expected value of a random variable X, such as $\mu = \mathbb{E}(X)$. Each X is an independent sample from the same distribution. We calculate our estimation of μ by aggregating all of the generated random value X_i and dividing by the number of samples n: $\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$. In the Monte Carlo method, true variance δ^2 is usually unknown. However, it can be estimated from the sample values. The estimates of δ^2 are $s^2 = \frac{1}{n-1}\sum_{i=1}^{n}\left(X_i - \bar{X}_n\right)$. The s^2 will be very close to true variance δ^2 if the number of samples is large enough. According to Central Limit Theorem (CLT), $\bar{X}_n$ - μ has nearly a normal distribution with mean 0 and variance δ^2 [8]. Therefore, the random sampling has a confidence interval which can be estimated as:

$$\bar{x}_n \pm z_{\alpha/2}\frac{s}{\sqrt{n}} \tag{1}$$

For a 95% confidence interval, we use $\alpha = 0.05$, and therefore set $z_{\alpha/2}$ = $z_{0.025}$ = 1.96. According to the confidence interval equation, the length of confidence interval tends to be 0 as the number of sample n gets extremely huge. By rearranging the confidence interval equation, we can generates the minimum sample size n under the length of 100 (1 - α)% confidence interval $2d$ for true mean μ as follows:

$$n \geq \left(z_{\alpha/2}\frac{s}{d}\right)^2 \tag{2}$$

To show the correlation between sample size and the length of confidence interval, we run the simulator emulating that hypervisor steals 10% of victim CPU resources with average intervals of 60ms. The Figure 5 presents the length of confidence interval associated with the sample size. The confidence interval is likely to drop sharply until 8000 samples. To narrow 1% length confidence interval, we needs about 16,000 samples. It indicates that the random sampling can have at least 1% error in 16,000 samples. If we sample more, we can more assure our results and the error can be more reduced.

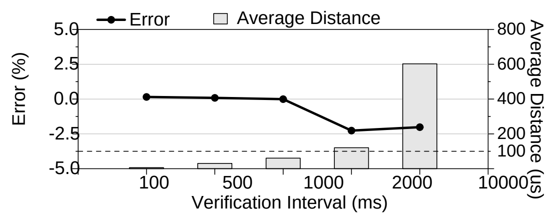

Figure 6: The Error on Long Intervals

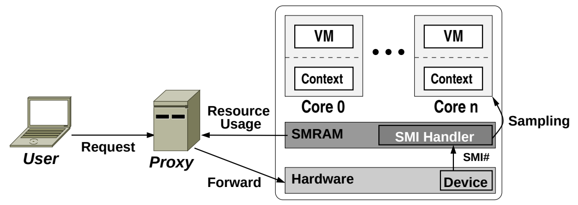

Figure 7: HRA Overview

However, the Monte Carlo method can generate misleading results if samples do not reflect a whole distribution. In random sampling, samples need to be uniformly distributed and cover whole VM scheduling stochastically. Suppose the sampling rate is $\frac{the\ number\ of\ samples}{time}$, and the interval can be defined as $\frac{time}{the\ number\ of\ samples}$. Longer intervals may miss more chances to detect hypervisor attacks, increasing the coverage error of sampling. With long sampling intervals, the random sampling does not cover the resource status occasionally in spite of a large number of samples.

With higher sampling rates, samples are more densely located within the same time interval. The average distance of samples is represented as follows:

$$Average\ Distance = \frac{Interval}{The\ Number\ of\ Samples} \quad (3)$$

Figure 6 shows that the random sampling does not produce correct CPU usage even with a large number of samples for long sampling intervals. In the experiment, the attack duration and the attack interval are 10μs and 100μs respectively. A malicious hypervisor steals 10% of CPU resources from the victim. The sampling interval is extremely increased from 100ms to 10000ms. The random sampling can produce correct results until 1000ms interval, but the error of 2000ms and 10000ms interval is beyond 2%. Nearly 50% of resource attacks cannot be detected with the long intervals, because the error is close to 2.5% and the average distance of samples is longer than the 100μs attack interval. In addition to the quantity of sample, the following equation for the restriction on sample distances must be satisfied to avoid large coverage errors.

$$Attack\ Interval \geq Average\ Distance \quad (4)$$

5. Architecture

5.1 Overview

This section describes the overall architecture of our resource accounting system, which is based on the random sampling described in the previous section. The probing program resides in SMRAM, and runs in SMM protected from the hypervisor. The system reports the amount of allocated CPU and memory resources even under a malicious hypervisor or administrator attempting to report false accounting. Figure 7 shows the overall architecture of the proposed accounting system. The system consists of a proxy system and computing machines. The proxy forwards accounting requests from users to a target system, issuing an SMI to generate the accounting summary. To be able to initiate SMIs by requests from a remote proxy, the computing machine must have an SMI-capable network device. The prototype system uses a serial device to invoke SMIs for the probing program from the proxy server. The serial device is excluded from the available devices for the hypervisor, which prevents any possible hypervisor intervention for the serial communication. For better bandwidth and scalability, the proposed accounting platform can use an out-of-band network channel. Most commercial servers are equipped with Ethernet as an out-of-band channel for IPMI (Intelligent Platform Management Interface). This out-of-band channel securely communicates with the proxy under a malicious hypervisor or administrators.

Once a user requests resource verification to a proxy while the user's application is running, the proxy enables the sample probing program in the computing machine, which schedules sample probing runs with random intervals set by the proxy. After aggregating samples, the probing program infers the actual usage of CPU and memory, and reports the result to the proxy. With a relatively minor overhead as will be discussed in the result section, the resource accounting system can be continuously running without noticeable performance degradation. The client can use the reported accounting information in different ways depending on the SLA. It can be used to verify whether the contracted amount of resources are actually allocated, or to verify the cost of CPU and memory, if the provider charges the cost based on the actual resource usage.

The accounting system depends on not only SMM but also hardware-assisted virtualization. For the hardware-assisted virtualization, the Virtual Machine Control Block (VMCB) [1] has an area to save the VM context information as well as the vCPU configurations. In our implementation, the VMCB is used to identify a VM and find the vCPU configurations. When a CPU enters SMM, the context of a running VM is saved in the SMM save-state area. The saved context contains the physical location of VMCB and indicates whether the running task belongs to the hypervisor or VM. In addition, the accounting system can identify which VM runs on a particular core by inspecting its VMCB.

Algorithm 1 Random sampling's algorithm

```
Proxy: Invoking VERIFY CPU at random
1: for $count \geq n$ do
2:     $interval \leftarrow rand(configuredvalue * 2)$
3:     Invoking Verify CPU via SMI
4:     $sleep(interval)$
5:     $n \leftarrow n + 1$
6: end for
```

Algorithm 2 Verify CPU's algorithm

```
Probing: Verifying CPU
1: $save\_state \leftarrow SMRAM + STATE\_OFFSET[core\_id]$
2: if $save\_state.svm\_state$ is not host then
3:     $vmcb \leftarrow save\_state.vmcb$
4:     $vcpu \leftarrow rb\_tree\_search(vmcb.nCR3)$
5:     $vcpu.run\_count \leftarrow vcpu.run\_count + 1$
6: end if
7: $total\_count \leftarrow total\_count + 1$
8: $RSM$
```

5.2 Basic Operations

HRA supports several basic operations for managing VMs and estimating the resource allocation. Currently, HRA supports six operations. *Create VM* and *DeleteVM* are used for tracking VM life cycles. *Create vCPU* and *Delete vCPU* are provided for managing vCPUs. These management operations maintain mapping information between the physical entity such as the nCR3 register value and the virtual entity such as the VM name. Mapping information are necessary for *Verify CPU* and *Verify MEM*, which are used for verifying resource allocation status. The detailed explanation of operations will be described in the following section.

5.3 VM Identification

To distinguish each VM, our accounting system selects the value of nCR3 register as a unique key reflecting VM identification. The nCR3 register value represents the VM address space, pointing the top translation page for nested address mapping. To register a VM to the probing program, the proxy must request to add a VM to the probing program by a *create VM* operation. A user requests a new VM to the proxy, and then the proxy forwards this request to the hypervisor for creating a new VM. After creating a new VM, the hypervisor informs the proxy the about completion of the request. The steps are similar to the conventional cloud systems. In addition, the proxy invokes *Create VM* and *Create vCPU* on the probing program to discover the nCR3 register value of the new VM. With *Create VM* and *Create vCPU*, the proxy maintains the mapping information between the VM name and nCR3 values. Once the mapping information between a user VM and its nCR3 value is established, the HRA system uses the nCR3 value as a key for the VM.

Algorithm 3 Verify MEM's algorithm

```
Probing: Verifying MEM
1: $save\_state \leftarrow SMRAM + STATE\_OFFSET[core\_id]$
2: if $save\_state.svm\_state$ is not host then
3:     $vmcb \leftarrow save\_state.vmcb$
4:     $mem \leftarrow rb\_tree\_search(vmcb.nCR3)$
5:     $mfn\_list \leftarrow npt\_walk(vmcb.nCR3)$
6:     $mem.count \leftarrow npt\_verify(mfn\_list)$
7: end if
8: $RSM$
```

5.4 Estimation of Resources

Algorithm 1 describes the invocation of *Verify CPU* with a random interval. The proxy generates a random interval with a configured value. The random intervals are uniformly distributed between 0 and $2 \times configured\ value$. After issuing an SMI to call *Verify CPU*, the proxy waits for the next sampling run. Generating random intervals in the proxy is to mitigate the performance overhead for high quality random number generation in computing nodes and to create random numbers isolated completely from the hypervisor.

Algorithm 2 shows how to verify the CPU allocation from the probing program. The saved state area includes the last context information before the SMI invocation, automatically created on SMI. The *svm_state* fields of the saved state represents whether the core runs on the host mode or guest mode. If the core runs on the guest mode, the probing program retrieves the VMCB address from the saved state and the nCR3 value of VMCB is used for identifying which VM was running on the core. To quickly look up the vCPU list created by *Create vCPU*, the probing program maintains a red-black tree as an index structure. The red-black tree provides the worst-case guarantees, which is critical in a low latency system such as the proposed accounting system. After looking up the vCPU list, the probing program increases the run count of the vCPU data, and then *RSM* instruction is called to resume the execution of the VM. The accounting systems count the number of samples each vCPU is running, and infer the CPU allocation with the ratio of the count of each vCPU over the total number of samples.

For memory resources, the probing program also uses the nCR3 register to estimate the total allocated memory of a VM. The nCR3 value points the nested page table referenced on every page translation. As the nested page table contains mapping information from the guest physical address to machine address space, the total allocated memory can be estimated by traversing the nested page table. Algorithm 3 shows the estimation of memory resources. After the probing program accesses the physical location of VMCB by inspecting the saved state, it traverses the nested page table pointed by nCR3 and measures the total allocated memory of the VM.

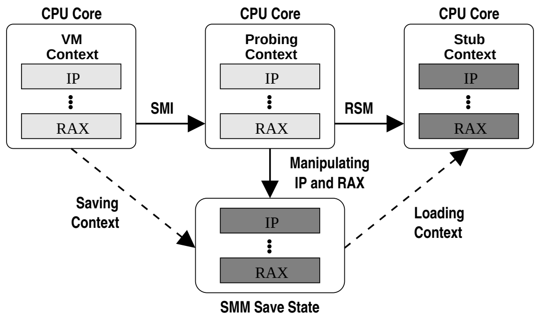

Figure 8: Snatching VM Context

To reduce latency of *Verify MEM*, the probing program only traverses addressable entries of nested page table. For example, if a VM has up to 1GB memory, the probing program just traverses the first l3 entries. Because the l3 entries cover the 1GB address space. The malicious hypervisor can share the same machine memory with multiple VMs. To detect these types of sharing attacks, the probing program inspects nested page tables of all VMs and identifies whether duplicated machine memory exists. The probing program also checks present bits in entries of nested page table due to memory swapping. The memory verification latency can be much longer than the CPU verification, and it may increase with larger memory capacities and small page sizes. We can reduce latency of *Verify MEM* by applying a spatial random sampling. The probing program randomly traverses a fixed number of entries, instead of traversing all of entries. Adopting random sampling, *Verify MEM* always generates the same latency regardless of memory configuration. In the result section, we discuss the effect of spatial sampling of memory allocation.

5.5 Avoiding Impersonating Attacks

A malicious hypervisor can allocate a duplicated nCR3 register value or dynamically change the nCR3 register at runtime. To detect such attacks, our accounting system maintains all the nCR3 register values of VMs in the system. However, the malicious hypervisor can use a more sophisticated attack impersonating a victim VM, switching the nCR3 values between the victim and attacker. For the attacker VM to run with the nCR3 value of the victim VM, the attacker VM must run on the address space of the victim. If the hypervisor simply extends the address space by altering the nested page table, the accounting system can detect this attack, inspecting entries of the nested page table. However, replacing memory contents with the attacker code is hard to detect without verifying the integrity of memory contents.

To detect this attack, our system uses a preliminary sample-based approach, which runs a stub code occasionally instead of running user VMs. This approach is similar to GINGER [24]. The system snatches a VM context switch and runs a stub code instead of the expected user VM. The stub code leaves integrity watermarks on its memory space and registers. The hypervisor, without knowing whether the current context is for a guest VM or the stub code, may attempt to compromise the memory content or registers to run an attacker VM. If a malicious hypervisor attacks the running stub context, our system can detect the integrity violation. The stub context works as a trap for attacks and runs a simple job consuming allocated CPU resources. Since it wastes CPU cycles, it can be used only for a limited number of sampled periods. Figure 8 shows snatching a VM context switch on an SMI, the VM context is saved and the probing program copies the stub code and changes the instruction pointer and RAX register to run the stub code. Then the probing program calls the RSM instruction to return to the prior context. The current implementation of stub code just leaves known simple values on memory and registers in random locations to detect any changes by the malicious hypervisor. A more sophisticated implementation is our future work.

5.6 Limitation

CPU-intensive workloads mostly consume all the CPU resources allocated to users. However, CPU and I/O mixed workloads voluntarily release assigned CPU shares before consuming all CPU shares waiting for I/O completion. Our accounting system just samples the current CPU usage and cannot identify whether the sampled result are caused by voluntary release or preemption by hypervisor. The accounting system reports only how much CPU share each VM actually received from the hypervisor, but does not identify the reason for not using CPUs, if the measured CPU share is lower than the expected value.

How to use the CPU and memory accounting information depends on the cloud service model and SLA. If the cloud service must always guarantee a certain amount of CPU and memory without work-conserving scheduling, the estimated accounting results must match the resource amount mandated by SLA. In commercial public clouds, such non-work-conserving model is commonly used [28, 34]. An alternative cloud model is to charge the cost based on the actual usage of resources. In such models, the accurate accounting is also critical to verify the cost from resource consumption.

6. Evaluation

6.1 Experimental Setup

Our prototype HRA platform consists of a proxy machine and a computing machine. The computing machine has a single AMD hexa-core 2.8GHz CPU supporting the hardware-assisted virtualization and 8GB DDR3 Memory. The proxy machine has a single Intel quad-core 3.4GHz CPU and 8GB DDR3 Memory. The proxy does not affect the performance of the computing system due to the limited role in forwarding user requests. We use Xen 4.0.1 [32] as the hypervisor to run VMs and use Ubuntu 10.04 for the management VM and user VMs. The proposed mechanism can be applicable to not only a variant of Xen but also other hypervisors with

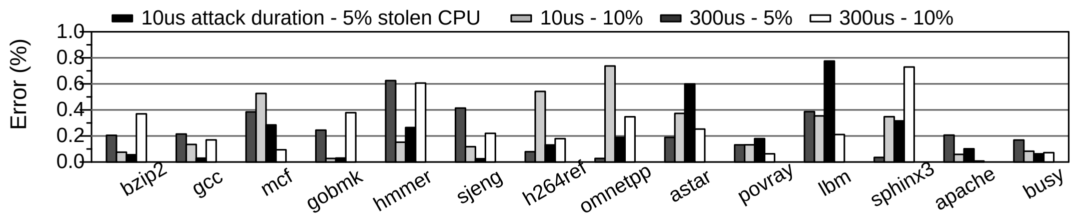

Figure 9: The CPU Verification Results on Real Environment

small modifications, since the HRA system does not modify the hypervisor or guest OS directly. A customized BIOS is installed on the computing machine to apply our probing program into SMM. The probing program has 16MB size in SMRAM where its codes and data reside. For communication between the proxy and computing systems, we use a serial device, which is excluded from the available devices to the hypervisor.

We run diverse experiments for evaluating the proposed mechanism. First, we run micro-benchmark tests to evaluate the performance of six basic operations of the proposed mechanism. HRA manages the data structures related to VMs through basic operations whenever a VM or vCPU is created. The experiment runs each operation 100 times, and the latencies are averaged. To estimate the CPU resources, HRA uses the aforementioned random sampling technique. We compare the actual CPU resources usage with the one estimated by the random sampling to show the accuracy and performance of the random sampling method.

As mentioned in Section 3, the overall system performance can be degraded because the running tasks have to stop when SMM is invoked. As a simple benchmark in each VM exercising CPUs, we first use a busy-waiting application, which constantly consumes CPU cycles. Furthermore, to investigate the trade-offs between the accuracy and performance, we use diverse workloads to reflect complex characteristics caused by multi-tenancy of clouds. The workloads consist of apache compile [2], SPECjbb 2005 [26], SPECCPU 2006 [25] and RUBis [19]. The apache compile is a CPU intensive workload compiling apache web server 2.2219. SPECjbb 2005 consumes a large amount of memory and is a typical memory intensive workload evaluating the server side java. SPECCPU 2006 is widely used in business and academia, measuring compute-intensive performance with a range of workloads. RUBis simulates an auction site for performance scalability. RUBis generates many TCP/IP packets, emulating various web interactions.

6.2 Micro-benchmark

HRA uses four operations as management operations, which are used for tracking and managing VM life cycle and vCPUs. As invoked at VM management, these operations are not frequently used during a VM life cycle and do not affect the performance of VM workloads significantly. Table 2 shows that these operations take between 53μs and 63μs. During these operations, a half of the elapsed time is spent for entering and exiting SMM as discussed in Section 3. The rest of the time is spent on performing the actual operations. Considering these operations are rarely executed, the overhead of these operations is negligible.

Operation	Time (μs)
Create / Delete VM	63.3 / 57.6
Create / Delete vCPU	61.7 / 58.1
Verify CPU	52.0
Verify MEM (1GB)	897
Verify MEM (20 entries)	111

Table 2: The Elapsed Time for Basic Operations

Verify CPU identifies which VM runs on the current core by inspecting the saved context. It takes 52μs for *Verify CPU*, therefore it must be negligible overhead as a single operation. However, to estimate the CPU usage, a large number of the operations should be called, which can cause performance degradation in the system. As mentioned in Section 4.3, we have to call at least 16000 *Verify CPU* operations to keep error below 1%. It takes 8.3s for calling consecutively 16000 *Verify CPU*. We will discuss an effect of CPU verification on system performance in Section 6.5. *Verify MEM* measures the total allocated memory to VM. It takes about 897μs for *Verify MEM 1GB*, accessing all of entries in the nested page table. The performance of *Verify MEM* with various memory sizes will be discussed in Section 6.4.

6.3 Measuring CPU Resources

Figure 9 shows the accuracy of CPU verification as compared with the actual CPU usage. There are four attack types in the experiment according to the attack duration and amount of stealing CPU resources. The prefix of each legend is the attack duration and the postfix of each legend presents how much hypervisor steals the CPU resources from the victim. For example, 10us - 5% indicates that hypervisor steals the CPU resource as much as 5% from the victim with 10μs attack duration. Random intervals with an average 60ms are used for verifying CPU resources in this experiment. SPECCPU 2006, apache and busy-waiting workload are used in this experiment. The experiment results are averaged from five runs.

This experiment shows that the random sampling method produces a good accuracy with below 1% error on every workload. These results are similar to the estimated result from the simulation experiments as shown in Figure 3, with

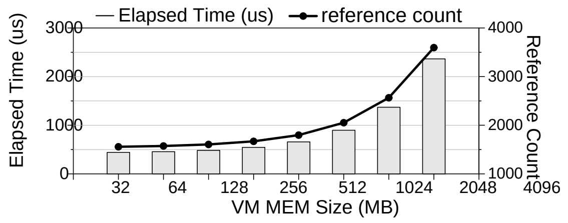

Figure 10: The Elapsed Time for Memory Verification

minor variations. There is no clear correlation among errors, workloads, and attack types. Even with a short 10μs attack duration, random sampling can still produce a good accuracy with the average 60ms interval. A very short attack duration requires that attack must be frequently occurred as much as the reduced attack duration. In addition, such a short attack durations slow down system performance due to frequent VM switching which pollutes system locality. 10μs attack duration slow down 20% of system performance on the gcc workload of SPECCPU 2006. The slowdown for the short attack duration is caused by the attacking hypervisor, not by the probing program which is invoked only with 60ms average intervals.

6.4 Measuring Memory Resources

Figure 10 shows the total elapsed time and memory references for memory verification with various memory sizes. Unlike the prior experiments, we use DSL Linux 4.4 [9] to run a VM with a small OS memory footprint, to measure the latency for the *verify MEM* from 32MB to 4GB memory sizes. The baseline Linux has too large a memory footprint to fit in the small 32MB memory. It takes 442μs for memory verification on 32MB memory size and there is a very little time increase of latency until 256MB memory size. The reason for minor increases is that the cost for traversing nested page tables does not increase significantly. Since the hypervisor assigns memory with a large page unit (2MB) mostly, a small number of page table entries can cover a large memory, reducing the number of memory references during memory verification. An exception is that the current Xen hypervisor allocates the first 6MB memory with 4KB page unit, which requires about 1500 memory references. Due to the small pages for the first 6MB, it takes a non-negligible latency for 32MB memory.

The number of memory references increase significantly when the memory size increases beyond 1024MB, with the considerable increase of the elapsed time with 4096MB. Memory is mainly shared by space but a malicious hypervisor can steal memory from victim time sharing. Frequent memory verification can detect this attack but may slow down system performance. To deal with this constraint, we use a random spatial sampling for memory. In memory verification, our system just traverses 20 entries by random, which consumes only 111μs. Figure 11 shows the accuracy of memory verification as compared with the actual allocation of memory. Four bars present the errors when 5%, 10%,

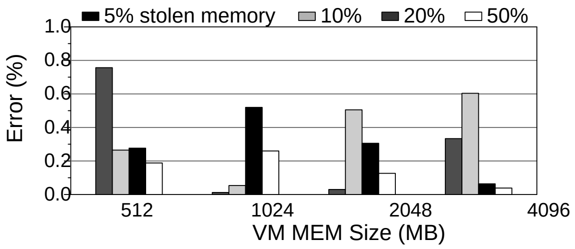

Figure 11: The Random Sampling for Memory Verification

20% and 50% of the victim memory stolen by hypervisor. In this experiment, we enable the balloon driver of guest VM to mimic hypervisor attacks. Balloon driver repeats the allocation and deallocation memory every 5 seconds. The random sampling of memory verification produces a good accuracy even with a small amount of stolen memory. The error does not exceed 1% on any attack case. Unlike CPU verification, only 150 samples are needed to be produce reliable accuracy in Monte Carlo calculation. The performance of memory verification with random sampling will be discussed in the follow section.

6.5 Performance of Workloads

To evaluate HRA on a real environment, we run various workloads with different sampling intervals. Each guest VM has 3072MB memory and a virtual CPU. Workloads consist of SPECCPU 2006, apache compilation, SPECjbb and RUBis workload. The inverse elapsed time is used as the performance metric in the SPECCPU 2006 and apache compilation. For the SPECjbb, the performance score is used as the metric and we use the averaged throughput as the metric for the RUBis.

Figure 12 shows the normalized performance with random CPU verification runs against non-CPU verification runs. The performance degradation appears on every workload when CPU verification is launched on an average interval of a very short 3ms interval. The performance of the apache compilation, SPECjbb and RUBis decrease by 7%, 8% and 3% respectively. The performance of SPECCPU 2006 decreases broadly. All cores are forced to stop on every sampling interval, which takes about 60μs as shown in Table 2. Especially the entire dirty cache lines must be flushed on entering SMM, which affects the locality of running workloads and causes performance degradations. These effects stand out on the CPU intensive workloads. SPECjbb consumes not only a huge memory but also CPU resources, so the performance degradation is severe in SPECjbb as well as SPECCPU 2006.

For the average 30ms interval, the performance of most workloads decreases only by 1% $\sim$ 2% due to less verification overheads than 3ms. For the 60ms, the performance degradation is negligible and there is no performance difference between 60ms interval and non-CPU verification. In addition, some workloads of SPECCPU 2006 show better performance against non-CPU verification. The CPU verifi-

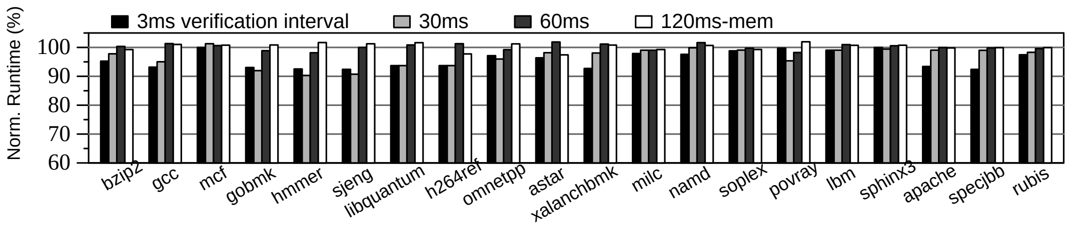

Figure 12: The Normalized Performance on Diverse Workloads

cation which runs randomly with an average of 60ms minimizes the performance degradations with less 2% and provides users with resource verification accurately with less than 1% error. The 120ms-mem legend shows normalized performance with random memory verification against non-CPU verification. Memory verification also shows negligible performance overheads with an average interval of 120ms. Such random sampling for memory verification with negligible performance overhead incurs less than 1% error.

7. Related Work

There have been recent studies on attacks on resource allocation for clouds. Varadarajan et al. showed a new class of attack called RFA (Resource-Freeing Attack) in commercial cloud services. RFA compels a victim VM to free up resources by causing interference [28]. Zhou et al. demonstrated scheduler vulnerabilities in commercial cloud services [34]. Exploiting timing-based manipulation, a malicious customer breaks fair allocation of CPU resources and consume resources allocated to valid users. To defend these types of attacks fundamentally, Chen et al. introduced ALIBI, which monitors resource allocation underneath the cloud provider platform with nested virtualization [7]. ALIBI supports verifiable resource accounting to users, tracking guest memory and CPU-cycle consumption. However, it relies on nested virtualization with non-negligible overheads.

There have been several studies about SLA in cloud services. Baset analyzed and compared diverse SLAs of current public IaaS, and then indicated that the provider does not support performance SLA and users have to produce the evidence of SLA violations [5]. Lango introduced new SD-SLA (Software-Defined SLAs) which guarantees various SLOs (Service-Level Objectives) and satisfies minimum service requirements of users by runtime VM reconfiguration [14]. Bouchenak et al. investigated various tools and methods which can verify cloud services. They discussed the properties which must be satisfied on verifying cloud services [6]. They also introduced a new cloud model to guarantee performance SLA under a reliable hypervisor and management VM.

There has been studies about not only new designs of SLA but also possible detection of SLA violation [10, 15, 17]. CASVid [10] detects SLA violation at application layer by allocating and monitoring resources. Macias et al. introduced a policy of SLAs for better QoS, classifying clients according to the relationship with the provider and payment [15]. They preferentially guarantee the SLA of client regarded as a high priority in the cloud. Maurer et al. introduced the resource management framework which supports automatic resource allocation and guarantee SLAs by VM reconfiguration with minimizing violations of SLA and maximizing resource utilization [17]. These prior studies [10, 15, 17] depend on the integrity of administrator and privileged software such as a hypervisor. If administrators or privileged software is malicious, these studies cannot guarantee SLAs. However, our mechanism does not rely on the integrity of administrators or hypervisor.

There are several prior studies [3, 4, 29, 33] using SMM. Most of the studies focus on protecting execution environments. HyperCheck [29] and HyperSentry [3] verify the integrity of hypervisor with SMM. When SMI is invoked, the SMI handler checks the integrity of hypervisor to identify cleanness of system. SPECTRE [33] is an introspection framework detecting memory-based stealthy malware via SMM. To detect heap spray, heap overflow, and rootkit, SPECTRE introspects the operating system kernel, including its code and data. SICE [4] provides an isolated execution environment with hardware supports such as SMM without relying on any software of legacy host. SICE can extend a piece of software into VM level as isolated execution environment, protecting the VM from a malicious hypervisor.

8. Conclusions

This paper proposed a hardware-based secure resource verification framework for cloud computing. The framework can verify CPU and memory resources even under a vulnerable hypervisor, and it reduces the tracking overhead by sample-based measurements. In consolidated virtualized cloud systems with data-center scale deployment, it will become more difficult or inefficient to verify the resource allocation for each individual user without a low-overhead framework. The proposed framework showed the feasibility of hardware-oriented secure resource verification.

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (No. NRF-2013R1A2A2A01015514) and by the IT R&D program of MSIP/IITP [10041313, UX-oriented Mobile SW Platform].

References

[1] Advanced Micro Dvices. AMD64 Architecture Programmer's Mannual: Volume 2: System Programming, 2007.

[2] Apache HTTP Server. `http://httpd.apache.org`, 2011.

[3] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of 17th ACM Conference on Computer and Communications Security, CCS 2010*.

[4] A. M. Azab, P. Ning, and X. Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security, CCS 2011*.

[5] S. A. Baset. Cloud SLAs: Present and Future. *ACM SIGOPS Operating Systems Review*, 46(2):57–66, 2012.

[6] S. Bouchenak, G. Chockler, H. Chockler, G. Gheorghe, N. Santos, and A. Shraer. Verifying Cloud Services: Present and Future. *ACM SIGOPS Operating Systems Review*, 47(2):6–19, 2013.

[7] C. Chen, P. Maniatis, A. Perrig, A. Vasudevan, and V. Sekar. Towards Verifiable Resource Accounting for Outsourced Computation. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2013*.

[8] K. L. Chung. *A Course In Probability Theory*. Academic press, 2001.

[9] Damn Small Linux. `http://www.damnsmalllinux.org/`, 2014.

[10] V. C. Emeakaroha, T. C. Ferreto, M. A. S. Netto, I. Brandic, and C. A. De Rose. CASViD: Application Level Monitoring for SLA Violation Detection in Clouds. In *Proceedings of the 36th IEEE Computer Software and Applications Conference, COMPSAC 2012*.

[11] H. C. Fengzhe Zhang, Jin Chen and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles, SOSP 2011*.

[12] Intel Corporation. Software Developer's Mannual vol. 3: System Programming Guide, 2009.

[13] S. Jin, J. Ahn, S. Cha, and J. Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011*.

[14] J. Lango. Toward Software-defined SLAs. *Communications of the ACM*, 57(1):54–60, 2014.

[15] M. Macias and J. Guitart. Client Classification Policies for SLA Enforcement in Shared Cloud Datacenters. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012*.

[16] D. Magenheimer. Xen developer's mailing list: `http://secunia.com/advisories/26986/`, 2010.

[17] M. Maurer, I. Brandic, and R. Sakellariou. Self-adaptive and Resource-efficient SLA Enactment for Cloud Computing Infrastructures. In *Proceedings of the 5th IEEE International Conference on Cloud Computing, CLOUD 2012*.

[18] R. Y. Rubinstein and D. P. Kroese. *Simulation and the Monte Carlo method*, volume 707. John Wiley & Sons, 2011.

[19] RUBiS Benchmark. `http://rubis.ow2.org`, 2008.

[20] Secunia Vulnerability Report. `http://secunia.com/advisories/15863/`, 2010.

[21] Secunia Vulnerability Report. `http://secunia.com/advisories/25985/`, 2010.

[22] Secunia Vulnerability Report: Xen 3.x. `http://secunia.com/advisories/product/15863/`, 2010.

[23] V. Sekar and P. Maniatis. Verifiable Resource Accounting for Cloud Computing Services. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop, CCSW 2011*.

[24] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking Proof-based Verified Computation a Few Steps Closer to Practicality. In *Proceedings of the 21st USENIX Conference on Security Symposium, Security 2012*.

[25] SPECCPU2006 Benchmark. `http://www.spec.org/cpu2006`, 2005.

[26] SPECjbb2005 Benchmark. `http://www.spec.org/jbb2005`, 2005.

[27] Trusted Platform Module. `http://www.trustedcomputinggroup.org/developers/trusted_platform_module`.

[28] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift. Resource-freeing Attacks: Improve Your Cloud Performance. In *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS 2012*.

[29] J. Wang, A. Stavrou, and A. K. Ghosh. HyperCheck: A Hardware-Assisted Integrity Monitor. In *Proceedings of 13th International Symposium on Recent Advances in Intrusion Detection, RAID 2010*.

[30] R. Wojtczuk. Subverting the Xen Hypervisor, 2008.

[31] R. Wojtczuk and J. Rutkowska. Xen 0wning trilogy, 2008.

[32] Xen Hypervisor. `http://www.xen.org/`, 2010.

[33] F. Zhang, K. Leach, K. Sun, and A. Stavrou. SPECTRE: A Dependable Introspection Framework via System Management Mode. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2013*.

[34] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram. Scheduler Vulnerabilities and Coordinated Attacks in Cloud Computing. In *Proceedings of the 10th IEEE International Symposium on Networking Computing and Applications, NCA 2011*.