Scotch: Combining Software Guard Extensions and System Management Mode to Monitor Cloud Resource Usage

Kevin Leach[1], Fengwei Zhang[2], and Westley Weimer[3]

[1] University of Virginia, Charlottesville, VA 22903 kjl2y@virginia.edu
[2] Wayne State University, Detroit, MI 48202 fengwei@wayne.edu
[3] University of Michigan, Ann Arbor, MI 48109 weimerw@umich.edu

Abstract. The growing reliance on cloud-based services has led to increased focus on cloud security. Cloud providers must deal with concerns from customers about the overall security of their cloud infrastructures. In particular, an increasing number of cloud attacks target resource allocation in cloud environments. For example, vulnerabilities in a hypervisor scheduler can be exploited by attackers to effectively steal CPU time from other benign guests on the same hypervisor. In this paper, we present Scotch, a system for transparent and accurate resource consumption accounting in a hypervisor. By combining x86-based System Management Mode with Intel Software Guard Extensions, we can ensure the integrity of our accounting information, even when the hypervisor has been compromised by an escaped malicious guest. We show that we can account for resources at every task switch and I/O interrupt, giving us richly detailed resource consumption information for each guest running on the hypervisor. We show that using our system incurs small but manageable overhead—roughly 1μs every task switch or I/O interrupt. We further discuss performance improvements that can be made for our proposed system by performing accounting at random intervals. Finally, we discuss the viability of this approach against multiple types of cloud-based resource attacks.

1 Introduction

The growing ubiquity of Software- and Infrastructure-as-a-Service has led to an increase in the cloud computing market. Spending on cloud computing infrastructure is projected to reach $38 billion in 2016 [14]. At the same time, the National Vulnerability Database shows that there are 226 security vulnerabilities in Xen, 99 vulnerabilities for VMWare ESX, and 98 vulnerabilities for KVM hypervisors [29]. As a result, there is additional concern over security breaches in cloud environments [20, 26].

Such vulnerabilities have already led to exploits related to the improper allocation of cloud resources. For instance, resource-freeing attacks [35] allow a malicious VM guest to take one resource from a victim VM (e.g., more CPU time). Similarly, vulnerabilities in hypervisor schedulers have been documented [32, 49].

Hypervisor vulnerabilities may permit a malicious customer to acquire cloud resources for free or at the expense of a victim. As a result, there is a need for cloud providers to guarantee levels of service and billing accountability to their customers using their infrastructure [24].

Cloud providers make use of virtualization platforms such as the Xen hypervisor [18]. Resource allocation is performed by the hypervisor according to the provider's configuration corresponding to the customer's service level. For example, a cloud provider might offer more CPU time to a customer that pays more money—this policy would be enforced by the hypervisor's scheduler. However, malicious customers that exploit vulnerabilities in the hypervisor may be able to evade this policy, obtaining more resources than would be dictated by their service levels.

In this paper, we present SCOTCH (Securely Communicating Objective, Transparent Cloud Health), a technique that leverages two x86 features to accurately account for resources consumed by virtual machines: System Management Mode (SMM) and Software Guard eXtensions (SGX). SMM permits transparent access to CPU registers and memory in the underlying operating system, hypervisor, and guests. SGX allows the creation of encrypted regions called enclaves that isolate critical execution from a potentially-compromised hypervisor or operating system. We can use SMM to track the resources consumed by each guest such that 1) potentially malicious guests are unaware, and 2) we can detect previously undetected resource accounting attacks. While SMM asynchronously measures resource usage, this information can be securely conveyed to an individual userspace enclave using SGX. This novel combination of SMM and SGX enables a new method of accurately measuring and securely communicating resource usage information in virtualized environments.

We evaluate a prototype of our technique based on the Xen hypervisor. We show that our technique takes roughly 1μs to check resource usage during each context switch and interrupt. We also show how this fixed 1μs cost can be amortized across multiple context switches and interrupts by randomly choosing intervals in which to check resource consumption. Next, we discuss the tradeoff between the quantity of a resource that can be stolen by a malicious guest compared to the overhead our technique incurs. Finally, we discuss the types of attacks for which SCOTCH is capable of providing accurate resource accounting information where other approaches cannot. We note that SCOTCH does not automatically decide whether malicious activity is occurring; a direct comparative study against such techniques remains future work.

We make the following contributions:

- A technique for accurately and transparently measuring system resources consumed by guest VMs running under a hypervisor,
- A prototype implementation employing the proposed technique for Xen, and
- An experimental evaluation of the prototype measuring accuracy and overhead of the proposed technique.

2 Background

In this section, we discuss three topics relevant to our proposed technique. First, we introduce System Management Mode, a special execution mode built into x86-based CPUs that permits transparent, isolated execution. Second, we discuss the Xen hypervisor and the types of vulnerabilities that could be leveraged by a malicious customer to gain or otherwise misuse cloud resources. Third, we introduce Intel Software Guard eXtensions (SGX), another set of instructions that enable our approach.

2.1 System Management Mode

System Management Mode (SMM) is a CPU mode available in all x86 architecture. It is similar to Real and Protected Modes. Originally designed for facilitating power control, recent work has leveraged SMM for system introspection [28, 43], debugging [45], and other security tasks [44, 46]. In brief, the CPU enters SMM upon a System Management Interrupt (SMI). While in SMM, the CPU executes the System Management Handler (SMI Handler), a special segment of code loaded from the Basic Input/Output System (BIOS) firmware into System Management RAM (SMRAM), an isolated region of system memory [6]. Upon completing executing the SMI Handler, the CPU resumes execution in Protected Mode.

We use SMM as a trusted execution environment for implementing our resource accounting functions. SMM has been available on all x86 platforms since the 386, so it is widely available for usage on commodity systems. In addition, the underlying operating system is essentially paused while the SMI handler executes. This isolated execution provides transparency to the operating system. We trust SMM for two main reasons: 1) SMRAM can be treated as secure storage because it is inaccessible by Protected and Real Modes, and 2) the SMI handler requires only a small trusted code base because it is stored in the BIOS and cannot be modified after booting when properly configured.

The SMI handler is stored as part of the BIOS. Typically, vendors ship SMI handler code specific to their platforms. Upon powering the system, the BIOS loads the SMI handler code into SMRAM before loading the operating system. After loading the SMI handler, the BIOS prevents further modifications to the SMI handler by locking down SMRAM. On Intel and AMD platforms, this is implemented using a write-once model-specific register (MSR); upon setting a specific bit, no other changes can be made to SMRAM (or the associated MSR). Thus, even if the hypervisor becomes completely compromised, the underlying SMI handler performing our resource accounting task will remain intact. The SMI handler is, by default, loaded into a 4KB region of memory, called the ASEG segment. We can alternatively load the SMI handler into another segment of memory called TSEG to allocate more space, often as much as 8MB.

Finally, as SMRAM is isolated in hardware (i.e., it cannot be mapped by the MMU unless the CPU is in SMM), a hypothetical DMA attack would not be able to corrupt resource accounting information stored in SMRAM.

2.2 Xen Credit Scheduler and Resource Accounting

Xen [18] is a widely-deployed open source hypervisor. Xen is responsible for multiplexing multiple independent guest virtual machines. In a cloud environment, customers are given access to guest VMs with different configurations according to how much they pay. For instance, a customer may pay more to the cloud provider for a VM configured with more memory, disk space, or nominal CPU time.

Xen uses the Xen Credit Scheduler [1] by default to manage CPU time. The Credit scheduler allocates virtual *credits* to each Virtual CPU (VCPU) that wants CPU time. Each VCPU can be given more or fewer credits depending on the service level paid for. That is, the scheduler can distribute more credits to one customer's VCPU over another's based on how much is billed for CPU time. Every context switch, the scheduler decides which VCPU to run next based in part on the number of credits that VCPU currently has. While there are other schedulers Xen can be run with (Cherkasova et al. [13] provide a comparison), the Credit scheduler is the most commonly deployed scheduler.

Critically, Xen runs a helper function (`burn_credits` in the `sched_credit.c` file) at a regular interval that deducts credits from the currently executing VCPU. In brief, this function approximates CPU usage over time by polling the currently-executing context. Previous research [24, 32, 49] discussed in Section 7 has already explored vulnerabilities related to this approximation. If a malicious guest knows about the interval at which `burn_credits` is executed, the guest can measure time precisely and yield the CPU before the credits are accounted for. In doing so, a malicious attacker can potentially use CPU time without being billed for it.

In addition, Xen maintains credit information (and other metadata) about each guest in memory. Guests that escape the VM [15] could potentially alter such data, yielding incorrect accounting (and later, billing) information. For example, by deducting credits more rapidly from a benign victim guest, the victim's apparent CPU consumption could be made to exceed its real consumption.

2.3 Software Guard eXtensions

Intel SGX is another new set of instructions that permits the creation of *enclaves* in userspace [23]. These enclaves are encrypted regions of memory (code and data) that cannot be accessed from outside of the enclave context. SGX allows computation to occur securely, even if the operating system or hypervisor is malicious.

SGX is intended to secure local computation; I/O instructions are illegal while inside an enclave. Instead, SGX-based applications must call out (via `OCALL`s) to switch to untrusted OS code to execute I/O on behalf of the enclave. SGX applications are therefore unable to monitor other activity happening on the system (e.g., through shared memory or device I/O) securely. In this paper, we use SMM to measure system-wide usage and then report this information to the end user via an SGX enclave application.

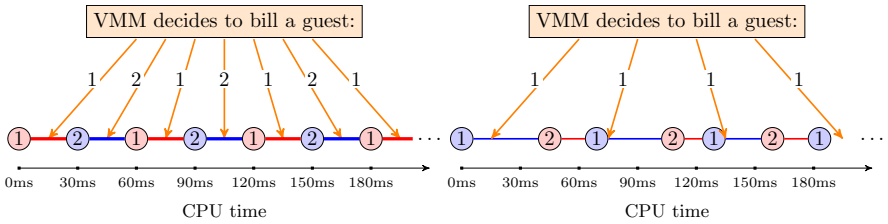
3 Threat Model

In this section, we discuss three types of attacks against which SCOTCH is capable of reliably accounting: 1) scheduler attacks, 2) resource interference attacks, and 3) VM escape attacks. These attacks increase in terms of expressive power and detriment against a hypervisor.

3.1 Scheduler attacks

We consider an attacker capable of exploiting vulnerabilities in the hypervisor's scheduler to acquire system resources for the malicious VM at the expense of a victim VM. This approach allows the attacker to prevent the victim from accessing rightful resources and also allows the attacker to perform expensive computations for free.

Figure 1a shows the non-attack scenario, a potential schedule of two benign CPU-bound VMs competing for CPU time on one physical CPU. Both guests 1 and 2 are given equal time, and when the VMM assesses which VM to bill, each guest is billed for its fair share of CPU time. However, as shown in the attack scenario in Figure 1b, a malicious guest could yield at precise times to avoid when the VMM attempts to assess which guest is running. As a result, a malicious VM could appear to never consume CPU time. Zhou et al. [49] showed that such an attack can consume the vast majority of CPU time under proper conditions.



(a) Non-attack scenario. (b) Attack scenario.

Fig. 1. Resource accounting scenario. A potential schedule of two benign VMs (denoted 1 and 2) with ideal CPU-bound workloads. The orange arrows represent when a VMM would poll which guest is running as part of determining billing. The accounting information inferred is accurate over time. In Subfigure 1b, a malicious guest closely controls CPU usage so that the benign guest (1) appears to use all of the CPU time.

3.2 Resource interference attacks

Resource interference attacks work by exploiting VM multi-tenancy. That is, all VM guests on a single hypervisor will have to share the underlying physical

resources at some point (e.g., there is only one system bus). A clever attacker VM can execute precise, calculated workloads that could impact the performance of other victim VMs or simply improve its own performance. For example, Resource Freeing Attacks [35] work by forcing a victim VM to free up a resource for the attacker to use. For example, the victim might be running a webserver, in which case the attacker can flood requests to the victim, cause it to block on I/O, and free up CPU time for the attacker. In this paper, we consider an attacker capable of degrading victim guest performance in this manner.

3.3 VM Escape attacks

Virtualization technologies, such as Xen, nominally isolate guest VMs from one another. Indeed, with full hardware virtualization, each guest believes it has control of an entire system. However, vulnerabilities inevitably find their way into hypervisors that allow malicious guests to escape out of the virtualization environment and execute arbitrary code within the hypervisor context [15, 27]. Naturally, such attacks can have a devastating impact on cloud providers, potentially exposing private or valuable data to the attacker. In this paper, we consider an attacker capable of escaping the guest context, and taking over the VMM.[4]

In this paper, we do not assume VM escape attacks that completely disable the system. For instance, it is very possible that a VM escape attack could compromise the hypervisor and stop executing all guests, or an attacker could attempt to disable network communications in the SMI handler with the Remote System. These sorts of denial-of-service (DoS) attacks can often be detected with timeouts and are out of scope for this work. Instead, we consider escape attacks where the attacker is capable of corrupting data structures related to resource usage.

4 Architecture

The goal of the Scotch architecture is to provide accurate and transparent resource accounting for cloud computing systems. This is done via resource accounting code that measures resources consumed by each guest residing on a hypervisor during every task switch and interrupt. We take advantage of hardware support to provide incorruptible accounting code and data storage as well as tamper-proof event-based invocation.

Figure 2 illustrates our system architecture. We have two or more systems in our approach. First, one or more Protected Systems run Virtual Machine Monitor (VMM) software capable of hosting multiple benign or malicious VM guests. Each Protected System reliably collects resource consumption information about each guest, periodically reporting this information to an SGX enclave. The SGX

[4] We assume the attacker can gain ring 0 (i.e., kernel) privilege after escaping the guest VM environment.

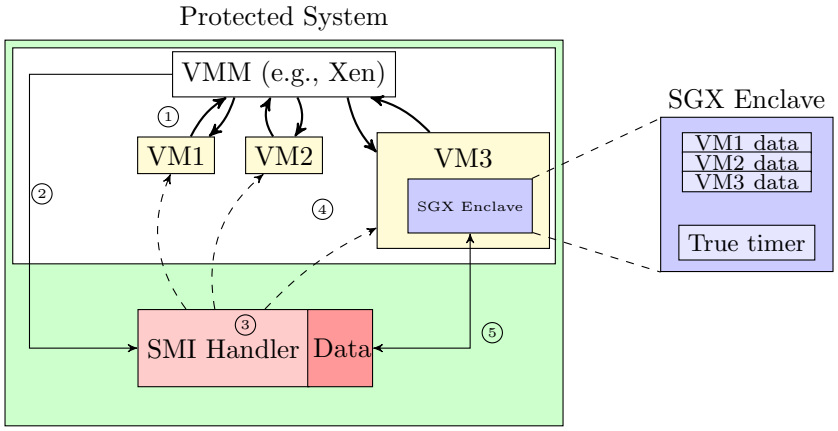


Fig. 2. High level overview of SCOTCH. The system contains one Protected System running VMM software containing a number of benign and malicious guests. One of the benign guests has an SGX enclave application running that receives accounting information from our tamper-resistant resource monitoring code. The annotations ①–⑤ correspond to the order of events in an indicative workflow. We assume benign guests are motivated to know their resource consumption.

enclave stores all of the resource consumption information from the VMs on the Protected System for further processing or analysis in a way that cannot be read or tampered with by a malicious guest, operating system, or hypervisor. In our implemented prototype of SCOTCH, we consider one Protected Machine with one SGX enclave.

4.1 Resource Accounting Workflow

The Protected Machine described in Figure 2 is responsible for collecting reliable and tamper-resistant resource consumption information about each VM guest whether it is malicious or benign. To accomplish this goal, we will discuss five steps (marked ①–⑤ in Figure 2) taken by the Protected System to ensure the integrity of the resource accounting information.

In step ①, the VMM is engaged by a VM guest through preemption or a hypercall to service an I/O request. Using hardware support (q.v. Section 5), we capture all such events, and execute our custom resource accounting code (denoted step ②). Note that the VM guest could be malicious or benign—we make no distinction in our approach because we are simply computing accurate and tamper-resistant resource accounting so that benign customers are eventually notified of the resources actually consumed.

During a context switch, step ② invokes an SMI, causing our accounting code to run in the SMI handler. Using further hardware support, we can convert certain types of I/O and event interrupts into SMIs. For instance, when a VM's time quantum elapses, a timer raises an interrupt telling the VMM to

switch guests. In SCOTCH, we change such interrupts to invoke SMIs instead. Invoking an SMI is critically important to the continued reliability of accounting information provided by our system.

In step ③, our accounting code records which VM guest will run next as well as the time elapsed since the last time our code executed (i.e., the last context switch event). This information is recorded in an isolated region of system memory, inaccessible from the hypervisor (or guest) context. For I/O events, we record information about what type of I/O is being done. For recording resource consumption besides CPU time, capturing these I/O events allows us to reason about whether a guest is consuming disk or network.

In step ④, our accounting code finishes executing and transfers control back to the guest. We do not pass control back to the hypervisor because a compromised hypervisor may change the result of a task switch event (cf. time-of-check-to-time-of-use attacks). For example, during a context switch, the hypervisor scheduler will select a new guest to run. If one were to perform resource accounting before the hypervisor finalizes the scheduling decision, a compromised hypervisor could spoof which guest will run next, perform accounting, and then run a different guest. Instead, in SCOTCH we invoke the resource accounting code right before control would have been transferred to the guest. After our accounting code completes, control flows directly to the correct guest.

Finally, step ⑤ represents a task that is completed occasionally. It is possible that a malicious guest that escapes to the hypervisor could corrupt data. In particular, if such an attacker is trying to hide the resources they consume, they might corrupt timers on the hypervisor that we use to measure the amount of time each guest spends consuming a resource. In such cases, we could use the SMI handler code (Step ②) to occasionally request time information from a trusted remote server (cf. Spectre [43]).

Cost of Accounting Recall that our approach invokes SMIs to reliably execute our resource accounting code. The invocation of the SMI and the resource accounting code itself both incur overhead on the hypervisor. This, in turn, affects the performance of the guests on the system, even if no malicious guests are running. For example, assuming a CPU-bound workload in which all guests consume all of their allocated time quanta, adding our resource accounting code essentially increases the amount of time taken to complete a context switch. Thus, deploying SCOTCH means accepting an associated performance loss in order to gain high accuracy, tamper-resistant resource accounting information.

As we discuss in Section 6, we also consider an alternative scenario to mitigate performance impact by invoking our code at different intervals. Ideally, we would invoke our accounting code on every possible task switch and I/O interrupt event. However, we could instead elect to invoke our code every x such events, where x is some random interval from 1 to some maximum interval. Essentially, every time an interrupt or task switch occurs, we flip a coin to decide whether to invoke our resource accounting code. This requires adding such decision code to the hypervisor, which could be noticed (or altered) by malicious, escaped

guests. However, we propose this approach as a means to significantly improve performance on diverse workloads. This option allows a cloud provider to trade off resource accounting granularity and overhead.

5 Implementation

In this section, we discuss how we implement our approach on a real system. Recall there are five steps in our workflow from Figure 2:

1. Capture interrupts and task switch events,
2. Redirect interrupts to invoke resource accounting code,
3. Compute resource usage impact of the current event,
4. Transfer CPU control to next guest, and
5. Relay accounting information into a trusted SGX enclave running within a VM guest.

Capturing these interrupts depends on features from Intel's Virtualization (VT-x) extension. In particular, we use VT-x's intercept capability, which allows us to control what happens as a result of a diverse array of events that can happen during execution, including task switching and interrupts. VT-x supports intercepting other events such as when a guest executes certain instructions, but we do not use this feature in SCOTCH. After intercepting task switches and I/O interrupts, we execute our resource accounting code.

We use System Management Mode (SMM) to implement our resource accounting code. We invoke a System Management Interrupt (SMI), which causes the CPU to save its state and transfer control to the SMI handler. The SMI handler is stored in the BIOS and loaded into the special SMRAM memory region upon booting the system. SMRAM is only addressable by SMM, and so any hypervisor or guest code running in Protected or Long Mode are not capable of reading or writing our SMI handler code. We overwrite the SMI handler with custom resource accounting code, which is then executed every time we assert an SMI.

SMIs can be asserted in several ways according to the platform's chipset. For our prototype, we use the AMD 800 series chipset. This platform supports invoking SMIs by writing to the legacy I/O port `0xb0` [5]. By executing `outb` instructions, we can invoke SMIs. Alternatively, we can also write to offset `0x9b` of the SMI control register of the advanced configuration and power interface (ACPI) MMIO configuration space.[5] Writes to this address causes an SMI to occur. Once an SMI is asserted, the CPU switches to SMM and begins executing the SMI handler at a fixed offset. Finally, we can also assert SMIs by configuring timing registers to deliver SMIs at configurable intervals.

We wrote a custom SMI handler that locates the VM guests residing on the system, identify which one was executing when the SMI occurred, and updates resource account information about that guest. On x86 machines, the control

[5] On our platform, the specific physical address was `0xfed8029b`.

register CR3 contains a pointer to the physical location of the page directory associated with a process—in Xen, the CR3 value can uniquely identify guests. We maintain a map of CR3 register values to VM guest IDs. We can also compute the location of the Virtual Machine Control Structure (VMCS) of each guest, which contains information about virtualized timers (and other information related to VM guest context). In our prototype, we have two guest VMs executing on one physical core—this setup simplifies identifying which guest is currently executing.

Recall that our SMI handler is invoked for one of two reasons: task switching or interrupt servicing. During a task switch, the VMCS page contains a pointer to the next guest that will run after the task switch completes. In other words, we know which guest will run next but not the guest that just completed running. Nonetheless, we can record current timestamp t_1 using the `rdtsc` instruction. Then, when the next task switch occurs, we can get another timestamp t_2, and use the difference $t_2 - t_1$ to estimate the amount of CPU time consumed by the guest that was previously executing. For interrupts, we can determine which IRQ was involved using the VMCS, from which we can determine the device that caused the interrupt. For our current prototype, we track the number of interrupts and associated IRQs corresponding to each guest.

After our resource accounting SMI handler completes, it switches back to Protected Mode to resume normal execution. Executing an `RSM` instruction restores the previous state and configuration registers. Ultimately, in our prototype, this transfers control of the CPU to the next guest task to execute without any space for the VMM to execute any instructions. Thus, even if the hypervisor is compromised, it does not have an opportunity to change the results of a task switch or interrupt event after we have completed our accounting code. This approach allows a highly granular and accurate view of resource consumption of each guest.

Next, we relay our accounting information to the SGX enclave, which stores data for later analysis in an isolated space. We cannot use SGX-related instructions while in SMM [23]. Instead, we perform several steps to get the data into the SGX enclave. First, we create a normal userspace stub program in the virtual machine guest containing the SGX enclave. This stub program contains a page of memory for arbitrary data, and code to marshall that data into the SGX enclave (via `EENTER`). We use the SMI handler to check the integrity of the stub program to detect potential tampering. Next, we note the physical address of this starting page, and the SMI handler writes its accounting data into that location. We configure the SMI handler to transfer control to the stub code after exiting SMM (by changing save state). The stub code (executing in Protected Mode at ring 3) then places that data into the enclave. This approach allows us confidence that the accounting data is securely relayed to user space.

Finally, we implement a network card driver in the SMI handler to communicate with the Remote System for accurate, external timing information. A similar approach was used in Spectre [43] and MalT [45]. We use symmetric key encryption with a key stored in SMRAM transmitted by the Remote

System as the BIOS is booting the Protected System. This ensures that the key is stored securely before the Protected System has an opportunity to load potentially-compromised hypervisor code.

6 Evaluation

In this section, we evaluate SCOTCH. We present experimental results and discussion. We seek to answer the following research questions:

RQ1 Can we perform accurate resource accounting during scheduler attacks?
RQ2 What is the overhead of our accounting approach on benign workloads?
RQ3 Can we accurately account resources during resource interference attacks?
RQ4 Can we perform accurate resource accounting during VM escape attacks?
RQ5 How do our CPU-based techniques apply to other resources?

6.1 Experimental Setup

Our experiments were carried out on an Intel Core i7-7700HQ 2.8GHz CPU with 32GB of memory. We ran two identical Ubuntu 15.04 guests, each given 256MB of memory and 1CPU core. We recorded the physical memory addresses of each guest's Virtual Machine Control Structure (VMCS) to ease experimentation. For ground truth data, we used Xen's built-in instrumentation, xentrace [3]. Xentrace behaves similarly to perf in that it can monitor for certain events and record resource usage. For some research questions, we developed our own attacks to mimic the behavior of possible attacks that would occur in the wild. Those implementations are detailed in the appropriate sections that follow.

6.2 RQ1: Scheduler Attack

Our first research question asks whether our system is capable of accurately recording CPU time consumption when a malicious guest uses a scheduler attack to steal CPU time. For this experiment, we have one malicious guest VM and one victim guest VM competing for the same amount of CPU time on a physical core. We wrote ten variants of the Xen credit scheduler, each of which gives the malicious VM an increasing amount of CPU time by influencing credit allocation in the scheduler. This is similar to the pseudo-attack implemented in [24], though we effect changes in the credits assigned to each guest over time to achieve changes in CPU time.

The ten scheduler variants are meant to represent varying degrees severity of a given attack—during each accounting period, each variant will randomly decide whether to deduct credits from the attacker VM, with variant n being $4n\%$ likely to skip credit deduction. That is, scheduler variant 10 is 40% likely to skip the deduction of credits on the attacker VM. This means that, over time, the attacker will have more credits and thus more time to get scheduled.

We ran several benchmark applications in both guests using each of the ten scheduler variants: pi, gzip, and PARSEC [11]. Computing pi represents a

Table 1. Ratio of attacker VM CPU time to guest VM CPU time.

	Scheduler attack severity level										
	Benign	1	2	3	4	5	6	7	8	9	10
SCOTCH	1.00	1.04	1.07	1.10	1.13	1.17	1.21	1.26	1.31	1.36	1.41
ground truth	0.99	1.05	1.09	1.12	1.15	1.17	1.20	1.25	1.30	1.35	1.39

highly CPU-bound workload, while gzip on a large random file represents a more mixed CPU and I/O-bound workload. The PARSEC benchmark suite has been used previously in the area of cloud performance and economics [37, 39]. Under benign circumstances, each guest should get 50% of the CPU time regardless of workload. When the attack variants are considered, an increasing amount of CPU time should be allocated to the attacker.

Table 1 shows the results of this experiment. We ran each benchmark program for five minutes measuring the CPU time allocated. We report the ratio between the attacker VM and victim VM CPU time for both SCOTCH and xentrace [3]. Furthermore, we average the results of all benchmarks. We note that, under benign circumstances, SCOTCH and xentrace both report a ratio of 1.0. However, as the attack becomes more severe, the attacker VM gets a higher ratio of CPU time, again validated against xentrace. This pattern is consistent across all workloads. Overall, SCOTCH performs accurate resource accounting even in the face of severe scheduler attacks.

6.3 RQ2: Overhead

We note that executing our isolated SMI handler resource accounting code takes additional time during each context switch and interrupt. Our SMI handler code takes 2248 ± 69 cycles to execute. On our 2.8GHz platform, that corresponds to about 1μs. However, acquiring granular resource accounting information means this 1μs cost must be incurred every context switch and every interrupt. In contrast, a typical VM switch takes roughly 20,000 cycles, or roughly 7.1μs. Adding our resource accounting code thus increases context switching time 14%. However, in purely CPU-bound workloads, Xen uses a 30ms default quantum per guest. Thus, the context switching time is amortized into the 30ms runtime per quantum. In other words, every 30ms of useful work requires a total of 8.1μs overhead in SCOTCH, compared to 7.1μs overhead in default systems. Thus, we can estimate the additional system overhead incurred by SCOTCH on CPU-bound workloads with:

$$\frac{|8.1\mu\text{s} - 7.1\mu\text{s}|}{(30\text{ms} + 7.1\mu\text{s})} = 33 \times 10^{-6}\text{additional overhead}$$

That is, our system incurs an additional .0033% overhead by using our system. As I/O operations typically take much longer in comparison to CPU-bound computation, this overhead reasonably approximates the worst-case overhead incurred by SCOTCH.

However, for the complete picture, we must also consider more realistic mixed CPU- and I/O-bound workload. Using gzip, we compressed a large randomly-generated file for a total of 5 minutes. The file was twice the size of guest system memory preclude caching the entire file and force operations to go all the way to disk. We measured the amount of CPU time and the amount of time spent servicing disk requests using our approach. In five minutes, there were 8070 context switches in which 214.59 seconds of CPU time were consumed. Thus, we can estimate the amount of CPU time consumed after each context switch with:

$$\frac{214.59\text{s}}{8070\text{switches}} = 26.6\text{ms},$$

which is reasonable (for reference, recall the standard quantum in Xen is 30ms): gzip would be spending some amount of time executing CPU-bound compression code. Using the formula above, we get an additional overhead of 0.0038%.

In contrast, there were 1371 interrupts going to disk, which took a total of 85.42 seconds. This corresponds to 62.3ms per interrupt. Using a similar formula above, we can estimate the additional overhead incurred on disk-bound interrupt events. For interrupts, this additional overhead is 0.0016%. Both values represent a significant improvement over existing SMM-based work [24]. While part of this improvement is due to faster SMI-handler code, much of the overhead depends on the underlying capability of the CPU to switch into SMM. Previous work has found SMI handler code takes on the order of 10μs [43, 45]. That said, even with a 100-fold increase in execution time of our SMI handler code, we still incur an overhead below 1%.

Note that we can further improve performance using an interval-based approach. Instead of invoking our code on every task switch or I/O interrupt, we can instead invoke our code after x such events, where x is a random number between 1 and some maximum interval. This random interval approach prevents transient resource attacks from going unnoticed because such attacks cannot learn a pattern for our resource accounting invocation. Thus, in the long run, such an approach maintains high accuracy with regard to resource accounting, but features a lower overhead. That said, spreading out the interval does create an opportunity for a sophisticated attacker to hide malicious activity; such an attacker could risk a certain amount of detection (determined by the measurement interval) by attempting to steal resources and counting on not be measured with our approach. Ultimately, the end user must decide the level of granularity of resource accounting information they need in comparison to the amount of overhead incurred by Scotch.

6.4 RQ3: Resource Interference Attacks

We also consider accounting in the face of resource interference attacks [35]. Scotch is capable of maintaining accurate resource accounting information even in the presence of such attacks. Because Scotch is invoked on every task switch and I/O interrupt, we maintain an accurate picture of resource consumption by construction. For example, as discussed in Section 3.2, a resource freeing attack

may work by causing a victim to block on I/O and thus free up CPU time for the attacker—but they still involve standard task switching and I/O interrupts. Thus, in such an attack, SCOTCH will accurately report that one guest is blocked on I/O and that the other is using the CPU.

We note that resource interference attacks often rely on an attacker's knowledge of a victim's workload. We reiterate that SCOTCH does not detect or prevent such an attack per se (although an analyst may do so by inspecting the resource accounting information). Instead, SCOTCH provides a guarantee about the quality and accuracy of resource accounting information our system delivers, even in the face of such attacks. This represents an improvement over previous approaches [12, 24], which neither detect nor prevent nor accurately account for resource usage in the presence of such attacks.

6.5 RQ4: VM Escape Attacks

Next, we discuss the viability of using SCOTCH even when the hypervisor has been compromised completely. Attacks such as Venom [15] or CloudBurst [27] allow a malicious VM guest to exploit vulnerabilities in the underlying hypervisor to escape the virtualized environment and execute arbitrary code in the hypervisor context. These are particularly dangerous attacks because they have the potential to compromise all of the other VM guests on the hypervisor. Additionally, such attacks are capable of changing resource allocation arbitrarily, potentially influencing ultimate billing for benign customers. In such cases, SCOTCH can provide accurate resource accounting information that can be used to provide accurate billing for all customers.

Recall that our resource accounting code is stored in isolated SMRAM. Even if an attacker is allowed ring 0 privilege in the underlying hypervisor, there is not a way for such an attacker to either 1) change previously-collected accounting information, or 2) change the accounting code itself. While ring 0 code could influence configuration registers and invoke spurious SMIs, a cursory analysis of the data transmitted to the Remote System would reveal such behavior. Additionally, such an attacker is not able to change SMM-related configuration registers because they are locked before the BIOS transfers control to the hypervisor.

However, malicious ring 0 code could alter kernel structures (Direct Kernel Object Manipulation [30]) or sensitive registers to influence accounting information before it is seen by the SMI handler. An attacker could, for instance, write the TSC register so that it appears a particular guest has consumed fewer cycles than it actually has, leading to an accounting discrepancy. In such cases, we could employ an instruction-level instrumentation approach similar to MALT [45] while kernel code executes to detect TSC writes or other malicious DKOM activity.

6.6 RQ5: Beyond CPU Time

RQ1 discusses experiments related to CPU time as a resource. However, SCOTCH is also capable of accurately recording VM guests' consumption of other system resources as well. First, by invoking our code on every I/O interrupt as well as

every task switch, we have the opportunity to examine consumption of peripheral devices (e.g., network and disk). As discussed in Section 5, VT-x allows us to gather information about the cause of the interrupt via the VMCS. Second, we do not give the hypervisor an opportunity to execute any code after the interrupt occurs—instead, after our resource accounting code executes, we transfer control to the next guest VM that was supposed to run after the interrupt completed. In doing so, there is no opportunity for a compromised hypervisor to alter the results of an interrupt to make it appear as though a different resource had been consumed.

6.7 Threats to validity

SCOTCH is a system meant to provide accurate resource accounting information in the cloud so that end customers have greater assurance that they are billed correctly according to the resources they really consume. While we have conducted experiments validating the high accuracy and low overhead of our approach, we discuss some assumptions we have made in conducting this evaluation.

First, we did not experiment using a test in the wild. For example, we implemented a resource-based attack by directly modifying the scheduler's behavior. We favored this approach because it admits controlled experimentation: it allowed us to vary how much of the CPU time was being stolen. We believe this represents different modalities of attackers with varying goals—some attackers may wish to operate more stealthily for longer periods of time, while others might operate more blatantly. We believe a controlled attack such as the one we have created is reasonably indicative of a variety of attacker behavior. Similarly, the benchmark workloads we evaluated on may not generalize. We attempted to mitigate this threat by including both microbenchmarks (CPU-bound and mixed) as well as the PARSEC [11] benchmarks which have been previously used in the area of cloud performance.

Second, invoking SMIs may cause perturbations in the behavior of certain caching mechanisms. For instance, the instruction cache might be cleared, and different chipsets and CPUs may perform other tasks while switching to SMM. Attacks abusing knowledge of this low-level detail have been documented [41,42]. In this paper, we assume that the hardware is trusted and that hardware-level bugs that admit such attacks are out of scope.

Third, while DMA attacks would be unable to affect the integrity of data stored in SMRAM or within the SGX enclave, there is a potential opportunity for an attacker to compromise data while it is being marshalled into the enclave from SMM. In SCOTCH, we configured the system to immediately transfer control to the enclave entry code after resuming from SMM. Depending on the platform's RSM implementation, there may be a small window to corrupt that marshalled data.

Finally, modifying the SMI handler to enable SCOTCH requires some degree of trust in the hardware vendor's BIOS code. Several attacks against SMM and related firmware have been discovered [17, 25]; such attacks could compromise the resilience of data collected by SCOTCH. We can mitigate such concerns by

using open source firmware where available, such as Coreboot [16] as used in SPECTRE [43] and MALT [45]. This would allow evaluating the firmware before deployment while trusting a restricted set of closed-source vendor code.

6.8 Evaluation Conclusions

Unlike previous approaches, SCOTCH was able to perform accurate resource accounting in the face of scheduler attacks, producing results that were within 2% of the ground truth. SCOTCH increases the cost of each context switch by 14%, which corresponds to a .0033% overhead for CPU-bound workloads and a .0016% overhead on more mixed workloads. This can be mitigated by accounting at random intervals, trading off granularity for overhead. By construction, SCOTCH provides accurate accounting in the face of resource interference attacks, since such attacks still use standard task switching and I/O interrupts. SCOTCH also provides accurate accounting in the presence VM escape attacks, since even the hypervisor cannot tamper with SMRAM or SMI handler code. In addition to accurately measuring CPU time, techniques in SCOTCH can address resources such as disk and network I/O that are processed through interrupts. Overall, SCOTCH provides transparent and accurate resource accounting for virtual machine guests.

7 Related Work

In this section, we discuss four main areas of related work: 1) Resource accounting techniques that propose helping cloud providers guarantee a particular service level to their customers, 2) SMM-based system protection techniques, 3) SGX-based system protection techniques, and 4) other multi-tenancy virtualization studies.

7.1 Resource Accounting

Chen et al. [12] propose Alibi, a system for verifiable resource accounting. It places a reference monitor underneath the service provider's software platforms (i.e., nested virtualization). Jin et al. [24] propose another verifiable resource accounting mechanism for CPU and memory allocation even when the hypervisor is compromised. Similar to our system, their approach also uses SMM as a trusted execution environment to account the resource usage. However, our system differs from previous work in the following ways:

1. By invoking our resource accounting code every context switch and interrupt, we can derive a granular resource usage report for each guest. This allows a rapid identification of discrepancies in resource usage. By contrast, Jin et al. employ a polling technique that requires running the analysis for a long time before a conclusion can be made—if an attacker is trying to be stealthy by stealing fewer resources, our approach can be used to more quickly identify such behavior, possibly within a few context switches, depending on the workload.

2. In addition, the manner in which our resource accounting code is invoked guarantees that we do not miss transient events—other techniques that employ polling for resource auditing may miss malicious guests that learn their polling behavior. For instance, Wang et al. [38] provides a systematic analysis of evasion attacks (i.e., transient attacks) for polling-based SMM systems. In such attacks, the adversary can circumvent the defense mechanisms by studying their polling behavior. With SCOTCH, if a malicious guest wants CPU time, control must transfer to it at some point, at which point our SMI handler will be invoked.
   However, this guarantee comes at the price of performance. As noted in Section 6, our resource accounting code incurs an additional 1μs per task switch and I/O event. We can tune this depending on the end-user's needs, instead invoking our code on random intervals to amortize the 1μs cost. Ultimately, the 1μs cost corresponds to a worst-case additional overhead of .0033%, which may be low enough for most applications.
3. SCOTCH requires no nested virtualization and retains a small Trusted Code Base (TCB) within SMM. In contrast, Alibi [12] incurs a higher overhead, roughly 6% CPU and 700% I/O, much of which is due to nested virtualization. Additionally, Alibi incorporates the KVM codebase, significantly increasing the TCB.
4. Finally, SCOTCH is capable of reporting accurate accounting information in the presence of a malicious guest capable of escaping the virtualization environment. An escaped guest might be able to change resource usage information recorded by the hypervisor (e.g., credits consumed in the Xen scheduler to hide oddities in consumed CPU time). However, as we store this information in SMRAM, we can derive an accurate report of resource usage without relying on data structures stored in the hypervisor.

In addition to works from academia, several industrial systems have been introduced for resource accounting [4, 31, 36]. For instance, Amazon AWS provides a tool called CloudWatch [4], which is a monitoring service for AWS cloud resources that provides system-wide visibility to resources consumed by cloud applications.

7.2 SMM-based Approaches

To the best of our knowledge, only Jin et al. [24] have proposed an SMM-based cloud resource accounting technique. Their approach is called Hardware-Assisted Resource Accounting (HRA). This technique is limited by its dependency on random polling. By sampling which VCPU (and therefore which VM guest) is currently executing, HRA relies on a large sample size to approximate a sort of Gantt chart of VM running time. Additionally, HRA relies on data structures in the hypervisor to coarsely approximate memory consumption. In contrast, by measuring resource consumption every context switch and interrupt, SCOTCH can rapidly determine accurate resource consumption information.

Additionally, there are several other SMM-based systems that are not directly used in securely reporting hypervisor resource consumption. These systems instead focus on detecting malicious activity [43], hiding keystrokes from the OS [44], and securing peripheral devices [46]. Furthermore, systems like HyperCheck [47] and HyperSentry [8] have been used to verify the integrity of a running hypervisor. Finally, MalT [45] proposed a transparent, remote debugging framework for use in analyzing stealthy malware or attacks capable of escaping a VM or rooting a system. Besides using SMM for defense, attackers use it for malicious purposes like implementing stealthy rootkits [19, 33]. For example, the National Security Agency (NSA) uses SMM to build advanced rootkits such as DeityBounce for Dell and Ironchef for HP Proliant servers [2].

7.3 SGX-based Approaches

Previous SGX-based systems such as Haven [10] ported system libraries and a library OS into an SGX enclave, which forms a large TCB. Arnautov et al. [7] proposed SCONE, a secure container mechanism for Docker that uses SGX to protect container processes from external attacks. Hunt et al. [21] developed Ryoan, a SGX- based distributed sandbox that enables users to keep their data secret in data-processing services. These two papers did not propose techniques to reduce the attack surface of computation inside enclaves or reduce the performance overhead imposed by SGX paging. Schuster et al. [34] developed VC3, an SGX-based trusted execution environment to execute MapReduce computation in clouds.

7.4 Other VM Multi-tenancy Studies

Zhang et al. [48] presented a class of memory denial-of-Service attacks in multi-tenant cloud servers, showing that a malicious VM may cause significant performance degradation of the victim VM by causing contention in storage-based and scheduling-based resources. Bates et al. [9] discussed using side-channel attacks to recover private information about co-resident VM guests. Similarly, Inci et al. [22] exploited side-channel information to acquire RSA keys from victim guests. Scotch does not address these sorts of attacks. We instead focus on scenarios in which attackers actively attempt to consume more resources for themselves at the expense of victim guests.

8 Future Work

In Section 3, we discussed three classes of attacks where Scotch can provide accurate resource accounting information. However, we also discuss *transplantation* attacks in which an escaped VM guest moves malicious code into a victim guest so that the victim computes and accesses resources on behalf of the malicious guest. Scotch and similar accounting systems are not currently capable helping detect such attacks or otherwise automatically deciding whether malicious

activity occurs. Even with perfectly accurate resource consumption information, the victim VM in this case would appear as though it were consuming resources as normal, and so the victim would end up being billed for work initiated by the attacker. We believe that such attacks would require detecting either the escape itself (i.e., detecting the vulnerability or exploit causing the guest to escape the virtualized environment) or detecting disparities from the normal workload performed by the benign guest. In the future, we would like to incorporate such detection into SCOTCH.

Additionally, we see SCOTCH as seeding the development of a general approach to securing interrupts and peripheral I/O. Currently, SGX does not support any form of secure communication outside the scope of the enclave. Existing work such as SGXIO [40] has investigated trusted I/O paths with peripheral devices. SCOTCH can target a similar application—by interacting with peripheral devices in SMM, we have the opportunity to attest firmware on potentially malicious devices, whereas SGXIO requires trusting a hypervisor containing a driver. We intend to explore securing I/O using SCOTCH's combination of SMM and SGX.

9 Conclusion

The growing popularity of cloud-based virtualization services, coupled with the increasing number of security vulnerabilities in hypervisors, presents a compelling need for accurate and transparent virtual machine resource accounting. We introduce SCOTCH, an architecture that uses System Management Mode on x86-based systems to carry out resource accounting and store information in an isolated manner that cannot be tampered with by a compromised guest or hypervisor. By accounting for resources at every task switch and I/O interrupt, our system is accurate in the presence of certain classes of attacks, such as scheduler attacks and resource interference attacks, by construction. SCOTCH produced results that were within 2% of the ground truth, while incurring a .0016% overhead on indicative workloads. Because SMRAM is isolated, SCOTCH can even provide accurate information in the face of VM escape attacks. Overall, SCOTCH provides transparent and accurate resource accounting for virtual machine guests.

References

1. Credit scheduler. http://wiki.xensource.com/xenwiki/CreditScheduler.
2. NSA's ANT Division Catalog of Exploits for Nearly Every Major Software/Hardware/Firmware. http://Leaksource.wordpress.com.
3. Xentrace. http://linux.die.net/man/8/xentrace.
4. Amazon AWS. Amazon CloudWatchamazon cloudwatch. https://aws.amazon.com/cloudwatch.
5. AMD. Amd rs800 asic family bios developer's guide, 2010.
6. AMD. Amd64 architecture programmer's manual, volume 2: System programming, 2013.

7. S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. OKeeffe, M. L. Stillwell, et al. Scone: Secure linux containers with intel sgx. In *12th USENIX Symp. Operating Systems Design and Implementation*, 2016.
8. 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 the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010.
9. A. Bates, B. Mood, J. Pletcher, H. Pruse, M. Valafar, and K. Butler. Detecting coresidency with active traffic analysis techniques. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, pages 1–12. ACM, 2012.
10. A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.
11. C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.
12. 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'13)*, 2014.
13. L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three cpu schedulers in xen. *SIGMnfluencingformance Evaluation Review*, 35(2):42–51, 2007.
14. L. Columbus. Roundup of cloud computing forecasts and market estimates, 2016. http://www.forbes.com/sites/louiscolumbus/2016/03/13/roundup-of-cloud-computing-forecasts-and-market-estimates-2016/, March 2016.
15. Common Vulnerability Database. VENOM: CVE-2015-3456, Xen 4.5 VM escape attack, 2015.
16. Coreboot. Open-Source BIOS. http://www.coreboot.org/.
17. C. Domas. The memory sinkhole. 2015.
18. B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *In Proceedings of the ACM Symposium on Operating Systems Principles*, 2003.
19. S. Embleton, S. Sparks, and C. Zou. SMM rootkits: A New Breed of OS Independent Malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks (SecureComm'08)*, 2008.
20. A. Garcia. Target settles for $39 million over data breach. http://money.cnn.com/2015/12/02/news/companies/target-data-breach-settlement/, December 2015.
21. T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: a distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 533–549. USENIX Association, 2016.
22. M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar. Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. Technical report, IACR Cryptology ePrint Archive, 2015.
23. Intel. Intel software guard extensions programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014.

24. S. Jin, J. Seol, J. Huh, and S. Maeng. Hardware-assisted Secure Resource Accounting under a Vulnerable Hypervisor. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'15)*, 2015.
25. C. Kallenberg and X. Kovah. How many million bioses would you like to infect? 2015.
26. L. Kelion. Apple toughens iCloud security after celebrity breach. http://www.bbc.com/news/technology-29237469, September 2014.
27. K. Kortchinsky. CLOUDBURST: A VMware Guest to Host Escape Story. In *Black Hat USA*, 2009.
28. K. Leach, C. Spensky, W. Weimer, and F. Zhang. Towards transparent introspection. In *23rd IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2016.
29. National Institute of Standards, NIST. National Vulnerability Database. http://nvd.nist.gov. Access time: 2016.05.10.
30. A. Prakash, E. Venkataramani, H. Yin, and Z. Lin. Manipulating semantic values in kernel data structures: Attack assessments and implications. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.
31. G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro*, 2010.
32. H. Rong, M. Xian, H. Wang, and J. Shi. *Information and Communications Security: 15th International Conference, ICICS 2013, Beijing, China, November 20-22, 2013. Proceedings*, chapter Time-Stealer: A Stealthy Threat for Virtualization Scheduler and Its Countermeasures, pages 100–112. Springer International Publishing, Cham, 2013.
33. J. Schiffman and D. Kaplan. The SMM Rootkit Revisited: Fun with USB. In *Proceedings of 9th International Conference on Availability, Reliability and Security (ARES'14)*, 2014.
34. F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 38–54. IEEE, 2015.
35. V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift. Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 281–292. ACM, 2012.
36. VMware Inc. vCenter Chargeback Manager. https://www.vmware.com/products/vcenter-chargeback.
37. H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, and L. Zhou. Distributed systems meet economics: Pricing in the cloud. *HotCloud*, 10:1–6, 2010.
38. J. Wang, K. Sun, and A. Stavrou. A Dependability Analysis of Hardware-Assisted Polling Integrity Checking Systems. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*, 2012.
39. L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 488–499. IEEE, 2014.
40. S. Weiser and M. Werner. Sgxio: Generic trusted i/o path for intel sgx. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 261–268, New York, NY, USA, 2017. ACM.

41. R. Wojtczuk and J. Rutkowska. Attacking Intel Trust Execution Technologies. http://invisiblethingslab.com/resources/bh09dc/Attacking2009.
42. R. Wojtczuk and J. Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning, 2009.
43. F. Zhang, K. Leach, K. Sun, and A. Stavrou. SPECTRE: A Dependable Introspection Framework via System Management Mode. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, 2013.
44. F. Zhang, K. Leach, H. Wang, and A. Stavrou. Trustlogin: Securing password-login on commodity operating systems. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 333–344. ACM, 2015.
45. F. Zhang, K. Leach, H. Wang, A. Stavrou, and K. Sun. Using hardware features for increased debugging transparency. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.
46. F. Zhang, H. Wang, K. Leach, and A. Stavrou. A framework to secure peripherals at runtime. In *Computer Security-ESORICS 2014*, pages 219–238. Springer, 2014.
47. F. Zhang, J. Wang, K. Sun, and A. Stavrou. HyperCheck: A Hardware-assisted Integrity Monitor. In *IEEE Transactions on Dependable and Secure Computing*, 2013.
48. T. Zhang, Y. Zhang, and R. B. Lee. Memory dos attacks in multi-tenant clouds: Severity and mitigation. *arXiv preprint arXiv:1603.03404*, 2016.
49. F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram. Scheduler vulnerabilities and coordinated attacks in cloud computing. *Journal of Computer Security*, 21(4):533–559, 2013.