Hindawi
Security and Communication Networks
Volume 2018, Article ID 4216240, 18 pages
https://doi.org/10.1155/2018/4216240

Research Article

Leveraging KVM Events to Detect Cache-Based Side Channel Attacks in a Virtualization Environment

Ady Wahyudi Paundu, Doudou Fall, Daisuke Miyamoto, and Youki Kadobayashi

Laboratory for Cyber Resilience, Nara Institute of Science and Technology, 8916-5 Takayama, Ikoma, Nara 630-0192, Japan

Correspondence should be addressed to Ady Wahyudi Paundu; ady.paundu.ak9@is.naist.jp

Received 25 September 2017; Revised 13 December 2017; Accepted 23 January 2018; Published 25 February 2018

Academic Editor: Wojciech Mazurczyk

Cache-based side channel attack (CSCa) techniques in virtualization systems are becoming more advanced, while defense methods against them are still perceived as nonpractical. The most recent CSCa variant called Flush + Flush has showed that the current detection methods can be easily bypassed. Within this work, we introduce a novel monitoring approach to detect CSCa operations inside a virtualization environment. We utilize the Kernel Virtual Machine (KVM) event data in the kernel and process this data using a machine learning technique to identify any CSCa operation in the guest Virtual Machine (VM). We evaluate our approach using Receiver Operating Characteristic (ROC) diagram of multiple attack and benign operation scenarios. Our method successfully separate the CSCa datasets from the non-CSCa datasets, on both trained and nontrained data scenarios. The successful classification also include the Flush + Flush attack scenario. We are also able to explain the classification results by extracting the set of most important features that separate both classes using their Fisher scores and show that our monitoring approach can work to detect CSCa in general. Finally, we evaluate the overhead impact of our CSCa monitoring method and show that it has a negligible computation overhead on the host and the guest VM.

1. Introduction

Virtualization technology has become a common utility in the current computation world. This technology has many advantages over traditional computing systems, such as lower cost, energy saving, faster provisioning, and application isolation. This isolation property is supposed to be one of the security properties of cloud computing systems. However, within the last decade, academicians and practitioners have discovered that this isolation is not impenetrable [1–4]. One well-known technique to break this isolation feature is a cache-based side channel attack (CSCa). This attack takes advantage of a main characteristic of the virtualization technique which shares physical hardware resources among multiple guest systems to improve server utilization. CSCa is known to be able to gather information such as cryptographic keys, keystroke sequences, coresidency, and website access across multiple CPUs, CPU cores, and even VMs. To help protect security in cloud computing systems, CSCa detection methods have become important.

There are many implementations of CSCa, but they all share one basic operation, which is timing a certain operation on the shared cache of the physical CPU. Two of the well-known CSCa techniques are Prime + Probe and Flush + Reload. In a nutshell, both techniques measure the time to read a certain location in the memory. The read operation will create either cache hit or cache miss events. Both events can be enumerated easily using the Hardware Performance Counter (HPC). Current CSCa detection methods utilize this cache hit and miss irregularity to detect the CSCa. However, the latest CSCa method called Flush + Flush employs an improved technique that does not require reading any memory locations. This improvement eliminates the cache hit-miss information and makes the Flush + Flush attack stealthier than the previous attacks.

On the other hand, to aid in virtualization security, methods for monitoring guest VM activity have also been proposed in many academic papers [5–12]. In general, those proposed VM monitoring methods can be categorized into three common techniques, which are computational metric

monitoring, system-call monitoring, and Virtual Machine Introspection (VMI). However, the effectiveness of a monitoring process in a public cloud is still limited due to the additional layer (virtualization layer) between the observer and the observation object. Furthermore, requirements in a public cloud limit the access of a cloud administrator to internal information from the guest system.

The motivation of this work is to introduce a Kernel Virtual Machine (KVM) events monitoring method for CSCa detection in the virtualization environment and to give proofs that the data features collected from the monitoring can give good detection results for three major variants of CSCa (Prime + Probe, Flush + Reload, and the latest more stealthy Flush + Flush) with a negligible computation overhead. To evaluate our monitoring method, we collected KVM events data inside the host from multiple emulated scenarios of CSCa and non-CSCa operation in the guest VM. Then we applied a Support Vector Machine (SVM) machine learning technique to analyze and classify the KVM events sequences and examined its accuracy using the AUC (Area Under the Curve of Receiver Operating Characteristic) unit.

The contribution of this paper is three-fold:

(1) First, we introduced a new method to monitor guest activity within a virtualization environment using KVM events data. This monitoring technique enables us to gather data with less performance overhead, without guest VM cooperation and without system components modification which are requirements in public cloud operation.

(2) Next, we showed that the proposed KVM events sequence data can be used to differentiate between non-CSCa operation data and CSCa operation data that includes Flush + Flush, the latest stealthier CSCa.

(3) Finally, we performed several empirical evaluations to measure the performance of our detection method. With our evaluation, we are able to answer the following questions:

   (a) Can the KVM events information be used to differentiate the CSCa and non-CSCa operations? How effective is this detection method to classify the trained scenarios and the new (untrained) scenarios?

   (b) Can the results in (a) be generalized, such that using this method with other scenarios or other microarchitectures is still able to give good detection results?

   (c) What is the effect of a noisy background or mimicry attempts by the adversary on the detection and the attack results?

   (d) How big is the impact given by the monitoring operation on the host and guest VM operation?

The remainder of this paper is organized as follows. In Section 2, we define the scope of our work by presenting the threat model and assumptions. In Section 3, we present several previous related works on cache-based side channel attacks, the evolution of the attack, and prevention and detection techniques. In Section 4, we give a theoretical background by providing a brief explanation on how KVM and the cache-based side channel attack work. In Section 5, we explain how our KVM event monitoring approach works. In Section 6, we present our empirical evaluation in detail. In Section 7, we discuss some more related issues and possibilities, and, finally, in Section 8, we conclude the paper.

2. Threat Model and Assumptions

In this section, we will define the scope of our work. In this work, we study the cache-based side channel attacks (CSCa). This set of attacks is a subset of two broader attack classes, side channel attacks, and microarchitecture attacks. We narrow this down to the three most well-known attack types, Prime + Probe, Flush + Reload, and Flush + Flush attack.

We focus further on attacks inside the virtualization environment. The resource sharing characteristic of virtualization technology makes this environment highly vulnerable to CSCa attacks. Since the virtualization environment is a vast and complex environment, it would be hard to cover it in full. To focus our study, within our threat model we assume that the cloud provider, its administrator, and its infrastructures are trusted. We also further assume that the Virtual Machine Monitor (VMM) is safe. However, we assume that one or more cloud tenants are not trusted and might have bad intentions to violate the privacy of the cloud by spying on a certain person's operation, either on their own VM or on the peer's VM. Moreover, our anticipated attack environments are public virtualization environments such as those using the Infrastructure as a Service (IaaS) Cloud model, where the host has limited-to-no authority over its guest system operations.

We set our defensive effort on a detection method. We base this choice on the assumption that the attackers do not know when the victim process will be executed; therefore the attackers have to put a constant probe on the cache before gathering any data from the victim. Furthermore, common CSCa techniques require many repeated bits of data from a victim to be able to extract any useful information. Hence, a CSCa spends most of its time in a loop observing the cache. We propose a detection method for this CSCa probing phase, which can then stop the attack from actually gathering its target information.

3. Related Work

The threat of CSCa attacks, especially in the virtualization environment, and methods of defense against these attacks have been researched since the early 2000s. This section first examines some of the work on CSCa attacks and then focuses on such attacks in the virtualization environment, before looking at research on defense and prevention. Since we propose a new VM monitoring technique, in this section we also describe the previous related works on guest VM observation method. These related studies provide the background for our study.

The idea of observing the cache access time as a side channel medium to spy on the victim process has been around since the early 2000s [13, 14]. The first application of this cache-based timing attack was demonstrated by Osvik et al. in 2006 [15]. The authors introduced two methods called Evict + Time and Prime + Probe. Both methods observe the state of the CPU's memory cache to reveal memory access patterns that later can be used in a cryptanalysis process. The Flush + Reload attack was introduced by Yarom and Falkner in 2014 [16]. This method took advantage of a memory deduplication technique [17] and improved the previous CSCa methods by increasing the speed and granularity of the attack to the cache-line level using the clflush function in the microarchitecture API. The CSCa not only has been proven to work for cryptanalysis purposes, but also can be used to spy on many other daily applications, such as a javascript browser [18], user interface [19], and even a mobile application [20].

In particular in the virtualization environment, an attack on a coresident VM was demonstrated by Ristenpart et al. in 2009 by recovering the keystrokes from a coresident VM in commercial clouds [21]. In 2012, Zhang et al. showed how to recover an El-Gamal decryption key from a coresident VM [22]. Later, the same authors presented ways to use CSCa to attack a peer VM within a Platform using a Service (PaaS) cloud model [23]. İnci et al. in 2016 presented a cache attack to enable bulk key recovery in a commercial cloud [24].

Many research studies have also been conducted on defense against CSCa attacks. One defense idea is to make the attack measurement process more difficult by introducing random variables. Such random variables include random memory-cache mapping, the use of prefetches, random timers, and random cache states [25–28]. Other proposals aimed to strengthen the victim application code to make it less vulnerable to CSCa attacks. This technique can be applied at the Operating System (OS) level [29, 30] or at the application level using sanity verification frameworks [31, 32]. Other approaches prevented cache sharing by distributing the VMs to different partitions in the cache, using either hardware [29, 33] or software [34, 35]. For CSCa in the cloud, the common protection idea is to change the new VM placement policies to reduce the probability of having the attacker VM and the victim VM stay in the same physical host [36–38]. However, cloud providers might find all these approaches less attractive because they require significant modifications to the cloud infrastructure.

Contrary to the many prevention techniques for CSCa attacks, detection methods have not been as widely studied. CSCa techniques are well-known to be very noisy and therefore can be easily detected using the Hardware Performance Counter (HPC). Chiappetta et al. used the HPC data and coupled it with a neural network method to detect CSCa in real time [39]. Zhang et al. went further by implementing CSCa detection in a virtualization environment [40]. They created a handshake system that correlates the signature-based detection of the cryptographic application in the victim VM with the anomaly detection system in the attacker's VM. This method requires cooperation from the victim VM to provide signatures of their cryptographic operation. Other detection methods were presented by Payer [41] and Herath and Fogh [42]. However, the latest development in CSCa introduced a new stealthier variant called Flush + Flush [43]. Since this method does not try to read the memory, no hit and miss events will happen; thus its existence cannot be detected using the HPC. As of the writing of this paper, we have not heard yet any academic paper presented to detect such attacks.

On the aspect of guest VM monitoring, many methods have been studied. Some of the common techniques to monitor the guest VM in a nonintrusive way are computation metric observation [5, 6], system-call observation [7–9], and Virtual Machine Introspection [10–12].

(i) The computation metric observation approach analyzes metrics such as CPU utilization, memory utilization, and the volume of block device read and write operations inside the guest VM. The main assumption of this approach is that a malicious activity will likely change some considerable amount of computing resources. The shortcoming of this approach is that it is hard to map the workload data to a specific process target inside the guest VM.

(ii) System-call is a set of interfaces that enable user processes to access the services that are provided by the Operating System (OS) kernel. By observing the system-call invocations from user processes to the underlying kernel system, a security agent can try to infer whether the user processes constitute a normal operation or not. However, the addition of a virtualization layer in the system makes this observation method less effective.

(iii) Virtual Machine Introspection (VMI) was first introduced by Garfinkel and Rosenblum (2003) [44]. It works by capturing a snapshot of the memory space used by the guest VM and uses it to reconstruct an exact same picture of the situation inside the guest VM. One minor limitation of the current VMI implementations is their dependency on information from the guest OS to correctly interpret the memory snapshots. Examples of such data are the debugging symbols information file for Windows systems or memory offset information file for Linux systems. Although this requirement is easy to satisfy in a private environment, in other arrangements such as a public IaaS, this approach could be hard to implement.

In this study, we move the research on CSCa detection forward by proposing a monitoring method that can detect even the latest Flush + Flush attack. This monitoring system can also be seen as an improvement over previous guest VM monitoring methods, as it can give clearer information on VM operation without the need of the guest VM operator participation.

4. Background

This section provides a brief explanation of how KVM and cache-based side channel attacks (CSCa) operate.

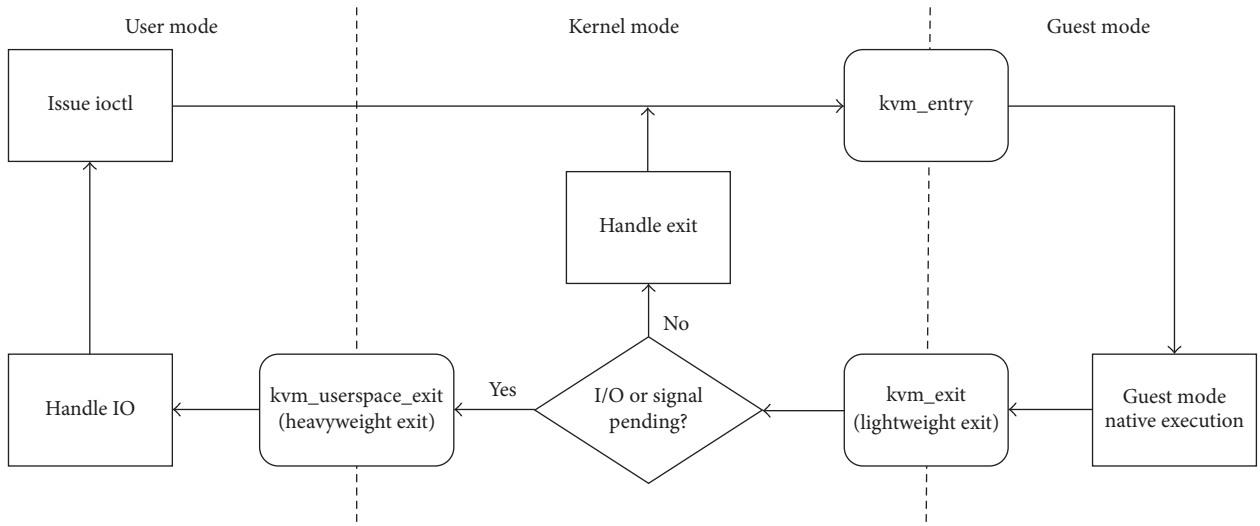

Figure 1: Guest execution loop.

4.1. Kernel Virtual Machine. A Kernel Virtual Machine (KVM) [45] is a virtualization solution that is embedded as a kernel module inside the Linux Operating System. This module enables the Linux system to act as a bare metal Virtual Machine Monitor (VMM) system (also usually being referred as type 1 virtualization). A VMM (or a Hypervisor) is a software that manages the virtualization environment operation, which includes the management of Virtual Machines (VM).

A KVM provides a set of Application Programming Interfaces (API) to utilize the hardware-assisted virtualization functions from the latest CPU architectures, such as Intel VT-x or AMD-V. Even though the hardware-assisted virtualization extensions are not standardized (both Intel and AMD processors have different instruction sets and capabilities), the basic operations are similar:

(i) The processors provide a new operating mode called guest mode, in addition to the previous two modes, the userspace mode and the kernel mode (the basic scheme of guest system operation is given in Figure 1). The guest mode enables the guest system to have all the regular privilege levels of the normal operating modes of a single Operating System. The exceptions of the privileges are several critical operating modes such as the control-sensitive IO operations (operations that have to change the state of system resources) and the handling of external interrupts, exception, and time-outs (scheduling operations are still performed by the host). These exceptions need to be performed by the host.

(ii) The operation switches between the kernel mode and guest mode, which include control registers, segment registers, and instruction pointers are performed by the hardware.

(iii) The hardware reports every exit reason (changes from the guest mode to the kernel mode) so the software can take proper action for the switch.

When it is time to run the guest system, the VMM calls ioctl() to instruct the KVM module to start up the guest system. The KVM then performs the VM entry and lets the guest system directly interact with the processor. If later the guest system is required to perform a critical instruction, it transfers the control to the kernel mode through VM exit (lightweight exit). If VMM intervention is required to execute an IO task, control is further transferred to the VMM userspace mode through KVM exit (heavyweight exit). On the completion of the VM exit handling, control is then given back to the guest mode through the VM entry process.

4.2. Cache-Based Side Channel Attack. A side channel attack is a method to gain information from a victim by eavesdropping through a nonconventional channel. An analogy would be that it is like trying to count the number of people in another room by hearing footsteps on the floor. In the case of a cache-based side channel attack, the floor is analogous to the CPU cache. An attacker measures the time to access a certain memory address to find out if those locations have been accessed (and henceforth cached) by the victim. The access information can then be translated into information about whether a certain operation has been executed or not by the victim. Since all the VMs inside a host share the same set of CPU caches, this technique can be used in the virtualization environment by an adversary to spy on its peer VM. For example, an attacker can spy on his neighbor VMs to detect if a certain user exists [21], or the attacker can spy any key-press on his peer tenant applications [19].

There are three common methods being used for cache-based side channel attacks, Prime + Probe, Flush + Reload, and Flush + Flush attack.

4.2.1. Prime + Probe. As the name suggests, this technique is comprised of two stages. In the Prime stage, the attacker evicts all the victim's data from the targeted cache set by allocating an array of memory blocks into that set. The attacker then

```
(1) procedure PrimeProbe (addr, thr)
(2)    accessed = []
(3)    access (addr)
(4)    while (true) do
(5)        wait ()
(6)        t0 = time ()
(7)        access (addr)
(8)        tx = time () − t0
(9)        if tx > thr then
(10)           accessed.append (1)
(11)       else
(12)           accessed.append (0)
(13)       end if
(14)   end while
(15)   return accessed
(16) end procedure
```

Algorithm 1: Prime + Probe.

```
(1) procedure FlushReload (addr, thr)
(2)    accessed = []
(3)    while (true) do
(4)        flush (addr)
(5)        wait ()
(6)        t0 = time ()
(7)        access (addr)
(8)        tx = time () − t0
(9)        if tx < thr then
(10)           accessed.append (1)
(11)       else
(12)           accessed.append (0)
(13)       end if
(14)   end while
(15)   return accessed
(16) end procedure
```

Algorithm 2: Flush + Reload.

waits for an interval before performing the next step. In the Probe stage, the attacker again reads the memory array and measures the access time. If the access time took longer than a certain time threshold, the attacker assumes that the cache set has been accessed by the victim during the interval. The attacker keeps repeating these Prime and Probe actions to collect the pattern of cache access by the victim which can be used later to extract information about the victim's operation. The method's operation is depicted as pseudocode in Algorithm 1.

4.2.2. Flush + Reload. This method requires that multiple identical processes using different virtual addresses be mapped into the same physical addresses. This mapping mechanism is intended to augment memory density. Two well-known implementations of this mechanism are Kernel Same-Page Merging (KSM) [46] and Transparent Page Sharing (TPS) [47].

The attacker first runs the process he wants to spy for so the process occupies the physical memory and the cache. Henceforth, anytime the victim runs the same process, the Operating System will map the process to the same location used by the attacker. The attacker then selects some specific cache line from the shared pages to be monitored. In the Flush stage, the attacker flushes his targeted cache lines. The attacker then waits for an interval before performing the Reload stage. In the Reload stage, the attacker reloads the memory blocks into the cache and measures the access time. If the access time is shorter than a predefined time threshold, it indicates a cache hit and the attacker will assume that the victim has performed the same instruction during the waiting time. As with the Prime + Probe, the attacker keeps repeating the Flush + Reload stages to collect the victim's instruction execution patterns.

Flush + Reload utilizes the assembly mnemonic *clflush*() that enables the cache flush to operate at the granularity of cache lines. To perform time measurement, this method uses the processor's hardware API, the *rdtsc*(). This Flush + Reload method has higher granularity information compared to the Prime + Probe since the Flush + Reload works at the level of cache lines. This method's operation is depicted as pseudocode in Algorithm 2.

4.2.3. Cache-Based Side Channel Attack Detection. Both Prime + Probe and Flush + Reload measure the access time of the cache. The access time of the cache is highly affected by the existence of the accessed data in the cache. The access time will be shorter if the data already exists in the cache. This is usually called a cache hit situation. In comparison, a cache miss means that the data being accessed is currently not in the cache and needs to be copied from memory, hence the longer access time. Fortunately, both events, the cache-hit and cache-miss, are observable from the processor. Modern microprocessors are equipped with a set of special purpose registers called Hardware Performance Counters (HPC). The HPCs are used to count all the CPU processing events and activities inside the computer system. Therefore, based on the HPC readings, previous CSCa detection methods can spot any CSCa attempts if they read an unusual number of cache-hits or cache-misses. As an example, a Flush + Reload probing process will create a constant high number of cache-miss that can easily be spotted.

4.2.4. Flush + Flush. The Flush + Flush method [43] is the latest variation of the Flush + Reload attack. It enhances the attack by removing the Reload stage of the spy process. Instead of measuring the time needed for the Reload stage, this method simply measures the time needed to execute the clflush(). The idea is that a flushing process will require less time if the address that needs to be flushed is not in the cache. Since there is no memory access in this attack, there is no cache miss which makes the previous detection technique almost impossible. Another advantage of Flush + Flush is that it gives higher resolution information because it works faster than the Flush + Reload attack. The Flush + Flush operation is depicted as pseudocode in Algorithm 3.

```
(1) procedure FlushFlush (addr, thr)
(2)    accessed = []
(3)    while (true) do
(4)       t0 = time ()
(5)       flush (addr)
(6)       tx = time() − t0
(7)       if tx > thr then
(8)          accessed.append (1)
(9)       else
(10)         accessed.append (0)
(11)      end if
(12)      wait ()
(13)   end while
(14)   return accessed
(15) end procedure
```

Algorithm 3: Flush + Flush.

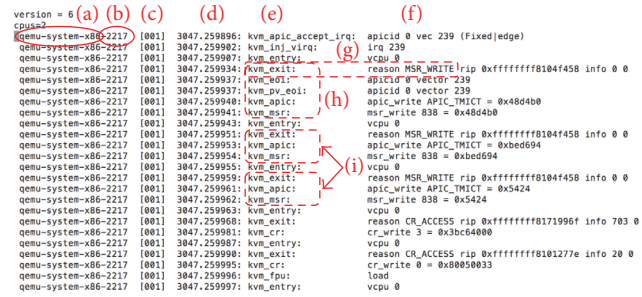

Figure 2: A snapshot example of trace-cmd output for KVM events. The preprocessing procedure to transform the text format into a vector input is explained in Section 5.2. (a) Process name. (b) Process/thread ID. (c) CPU ID. (d) Timestamps. (e) KVM event name. (f) KVM event information. (g) An example of one KVM_exit event and its exit reason. In this case we log the reason attribute. (h) An example of one KVM exit session that we used as one data (sequence) type. (i) An example of two sequences that belong to one sequence type.

We performed a simple test using *perf* tool ("*perf kvm stat -e cache-misses, cache-references -p* PID") on a VM running each of Prime + Probe, Flush + Reload, Flush + Flush, and a VM running web application. The average over 10 tries were 94%, 97%, 12%, and 18% (the percentage represents the ratio of cache misses over cache references) for Prime + Probe, Flush + Reload, Flush + Flush, and web application, respectively.

5. Monitoring System Design

This section describes our approach to detecting CSCa.

5.1. KVM Events. In computing world terms, an event can be defined as "a change of state." The same definition will be used in this paper, where KVM events are the changes of states inside the KVM module during kernel mode operation (see Figure 1). In our implementation, we introspected the KVM events that are instrumented by a standard Linux kernel tracing utility called *ftrace* [48]. Ftrace was built directly into the Linux kernel and thus brings the ability to see what is happening inside the kernel. We have three reasons to utilize this default Linux KVM instrumentation instead of creating our own user defined instrumentation. First, it allows us to target the generic hardware environment. Microarchitecture attacks depend on the type of hardware being used. To add a new probe, we would have to consider every possible hardware combination, which would increase the complexity of our study. Therefore, we decided to utilize the default set of probes that are provided by Linux and use a machine learning process to decide which events should be used for the classification process. Second, by not changing the default set of trace points, we wanted to ensure ease of implementation and make it applicable in a production environment. Finally, by using the built-in Linux function, we expected a lesser cost in computation. To ease the ftrace tracing process, we used the *trace-cmd* tool. Trace-cmd is a user-space front-end for ftrace that automates the process of accessing multiple files when directly working with ftrace itself.

The basic trace-cmd command that we used to capture KVM events from the host is *"trace-cmd record -e kvm -P xxx"* (where *xxx* is the process/thread ID of the guest KVM VCPU). This command pins data collection to one specific process/thread that represents the VCPU of the VM, thus enabling us to specify which guest VM to observe. An example of the output of this tool is given in Figure 2. It gives us the list of KVM events sequences that occurred during kernel mode operation (Section 4.1). The information gathered from this tool is the process name, process or thread ID, CPU ID, time information, KVM event name, KVM event information, and the sequence of the events.

5.2. Data Transformation. The raw data format is a text file that contains a list of KVM operation events in a chronological order. This raw data also gives additional information such as the name and parameters of each event. Figure 2 shows an example of the raw data.

We defined our data unit as the number of KVM event sequences within one monitoring time unit (e.g., 1 second). A KVM event sequence is a list of ordered KVM events that occurred between one VM exit to the next VM entry (one kernel mode session). For each KVM event, we only captured its name, with an exception for VM exit events where we captured its exit reason information. Having more features from the KVM events might increase the detection results; however, to minimize complexity, we decided to start simple and only increase the information level if it is deemed as necessary.

We formalize a data unit $X = \{Y_1, Y_2, Y_3, \ldots, Y_n\}$, where Y_i is the number of ith KVM event sequences in observation X and n is the total number of unique KVM event sequences in the dataset.

For an illustration, the observation example in Figure 2 contains five KVM event sequences:

(1) MSR_WRITE - kvm_eoi - kvm_pv_eoi - kvm_apic - kvm_msr

(2) MSR_WRITE - kvm_apic - kvm_msr

(3) MSR_WRITE - kvm_apic - kvm_msr

(4) CR_ACCESS - kvm_cr

(5) CR_ACCESS - kvm_cr - kvm_fpu

We simplified the data presentation by converting them into sequence IDs. The observation example in Figure 2 gives four sequence IDs (note that sequences which are pointed to by (i) belong to the same ID):

(i) ID1: MSR_WRITE - kvm_eoi - kvm_pv_eoi - kvm_apic - kvm_msr

(ii) ID2: MSR_WRITE - kvm_apic - kvm_msr

(iii) ID3: CR_ACCESS - kvm_cr

(iv) ID4: CR_ACCESS - kvm_cr - kvm_fpu

After having transformed all the sequences into IDs, we then counted how many times each ID showed up in an observation. Again, for illustration, having an input of Figure 2, the output would be freq(X) = freq(ID1, ID2, ID3, ID4) = (1, 2, 1, 1). We use this bag of KVM event sequence data as the input for the machine learning process to detect a CSCa attack.

6. Evaluation

6.1. Setup

6.1.1. Computation Environment. We setup one host on a Dell Poweredge 860. This machine was equipped with one Intel Xeon Dual core 3040 1.86 GHz (Conroe), 64 KB L1 (32 KB L1d + 32 KB L1i), 2 MB L2, and 8 GB system memory. Inside the host we setup eight VMs (for the scalability evaluation later). All the VMs had one virtual CPU, 512 MB memory, and 20 GB disk size. For the OS in the host and guest VM we used Ubuntu LTS 14.04 Linux (kernel version: 3.13.0-24-generic). We also setup one external computer as the web workload generator.

6.1.2. Scenarios. We collected data from multiple scenarios that represent the cache-based side channel attacks and common operations in the public cloud. We categorized the scenarios into two main classes, a positive class which contains all CSCa scenarios and a negative class which contains all non-CSCa scenarios (Table 1).

For the positive class, we collected five datasets of CSCa attacks:

(1) Three CSCa implementations from Gruss [43] (https://github.com/IAIK/flush_flush/tree/master/sc). These are Prime + Probe, Flush + Reload, and Flush + Flush attacks to eavesdrop for function calls of key-press on a Linux User Interface that utilized the libgdk library.

(2) The original Flush + Reload implementation from Yarom that spies on GnuPG's RSA implementation [16] (https://github.com/defuse/flush-reload-attacks/tree/master/flush-reload/original-from-authors).

(3) Another Flush + Reload implementation from Hornby that spies on the victim's browsing destinations [19] (https://github.com/defuse/flush-reload-attacks).

Table 1: List of all collected scenarios for evaluation.

Positive class	Negative class	
	Standard Op.	CPU Intensive Op.
Prime + Probe (Gruss)	Idle	Stress CPU
Flush + Reload (Gruss)	RUBiS 20 clients	Stress memory
Flush + Flush (Gruss)	RUBiS 200 clients	Binary tree
Flush + Reload (Yarom)	RUBiS 2000 clients	Lucas-Lehmer
Flush + Reload (Hornby)	Mail server	Urandom generator

For the negative class, we collected ten datasets of non-CSCa operation:

(1) Idle scenario: in this scenario, the VM just did nothing (with the exception of standard Linux daemons in the background). We needed to include this in our evaluation since every guest VM would go through this scenario at some time in its life-cycle.

(2) Web application scenario: we decided to use web scenario workloads under the assumption that web operations are being run the most in the public cloud system. Approximately 25% of IP addresses in Amazon's EC2 address space hosted a publicly accessible web server [21]. Web server operations also allowed us to experiment with multiple normal workload profiles for our evaluation purpose. We used RUBiS application [49] to emulate this web application scenario. RUBiS is a prototype of an auction site that was built to evaluate web application server scalability. RUBiS allowed us to easily scale the workload and generate dynamic web traffic. We used the *workload_number_of_clients_per_node* attribute to control the application workload. We collected the KVM events for three web application scenarios with different workloads, which are 20, 200, and 2000 clients.

(3) Mail server scenario: we set up a Postfix mail server system in a VM with 100 dummy users. We generated the load data from an external machine using the *postal* application (https://doc.coker.com.au/projects/postal/). For this scenario, the options that we used were as follows:

(i) Maximum size of message body: 10 Kilobytes

(ii) Number of threads that should be created for separate connection attempts: 10

(iii) Number of messages per SMTP connection: 100

(iv) Maximum number of messages per minute: 1000.

(4) CPU and memory stress test scenario: our decision to include this scenario class was intended to possibly maximize the number of false positives that our test scenarios can generate. The high intensity usage on the CPU and memory by the CSCa might not be observed in a standard VM operation (such as a web server). Therefore, we needed to introduce

several scenarios that uniformly and highly utilized the computer's CPU or memory to give a good upper false positive threshold. We collected five datasets for this scenario:

(a) Linux CPU and memory stress test: we used the standard *stress* tool from the Linux.

(b) Standard Linux random number generator: we chose the *urandom* device from Linux that use "unlimited" nonblocking random source. We performed the following: cat /dev/urandom > /dev/null. This operation is another well-known stress test for CPU.

(c) Another two mathematical operations.

(i) A python operation to solve Lucas-Lehmer prime test equation. This problem is used by many benchmark tools for stressing the CPU operation.

(ii) A binary tree operation to fully create perfect binary trees. This program stretched memory utilization by allocating, walking, and then deallocating nodes of a big binary tree. The process of allocating and deallocating memory page will mimic the cache access operation of a CSCa.

We collected data from all the scenarios exclusively. This means that there were no other operations being run at the same time we executed and collected each scenario's data. The adversary also will try to operate in an exclusive environment as much as possible to increase the CSCa effectiveness. Our evaluations on the obfuscation attempts by an attacker are given in a separate section (Section 6.4).

We evaluated our data in batches. That means, instead of evaluating them one by one in real time as the data came in, we collected the data in groups and evaluated them all together (offline). One observation data unit is a collection of all KVM events that were captured in one second. We ran each of the scenarios in turn inside the guest while collecting the KVM events inside the host. For each scenario, we collected 500 units of observation data.

For further research on this topic, our dataset can be accessed at http://iplab.naist.jp/research/CSCaD.

6.1.3. Machine Learning Setup. We applied a machine learning approach for the classification process. Microarchitectural and Operating System domain data consist of a high number of variables and parameters which are hard to observe manually. Furthermore, not all information about those variables and parameters is available to the virtualization operators. Therefore, we believe that a machine learning approach is the best option for a real world detection operation. In the evaluation phase we used the Support Vector Machine method [50] with a Radial-Based Function (RBF) to perform binary classification (CSCa or non-CSCa). We chose this supervised approach for its ease of use, while allowing us to observe in detail the differential aspect of the monitoring data between the benign scenarios and the CSCa scenarios. We utilized Scikit-learn libraries [51] for the machine learning implementation.

It is important to emphasize that our evaluation was not meant to benchmark the machine learning engine. Our chosen machine learning algorithm was selected only by its common use in classifying high dimensional data. Instead, we wanted to benchmark the ruleset, which in this case describes the characteristics and formats of the KVM event sequences from our monitoring data. Therefore, the identification of false positives and false negatives is still required even though we only used one machine learning method in our evaluation.

To avoid any confusion, we define the following quantities:

(i) True positive, CSCa data classified into the CSCa class

(ii) False positive, non-CSCa data classified into the CSCa class

(iii) True negative, non-CSCa data classified into the non-CSCa class

(iv) False negative, CSCa data classified into the non-CSCa class.

We conducted a small scale Grid Search experiment to find the best γ value for our SVM function. We found the value of 0.0003 for γ and used this value throughout this evaluation process.

For preprocessing the data, we first applied a standardization process that converted the data into standard normally distributed data: Gaussian with zero mean and unit variance. The second preprocessing step was simple removal of all the features with low variance. This second step was needed because there were a lot of sequences that appear only rarely (most of its occurrence value was 0) and can be seen as exceptions. Our filter was arbitrarily set up at 0.9, such that we removed all features that contained at least 90% similar values. The initial number of features (unique sequences of events) in the raw data was 271. After the preprocessing stage, the number of features was reduced to 69.

It is preferable to have multiple pairs of learning-test datasets to make sure that the results are not dependent on one particular random choice of learning datasets. One way to create multiple learning and test datasets is by applying a *k-fold cross validation*. In this study, as we have 500 data units for each dataset, we applied a 5-fold cross validation. In our evaluation, we calculated the average score of the 5-fold results as the final detection score.

For the detection measurement unit, we used the Area Under the Curve (AUC) value of the Receiver Operating Characteristic (ROC). The AUC value can be interpreted as the expectation that a uniformly drawn random positive sample is consistently ranked before or after a uniformly drawn random negative samples. Thus, the AUC can be seen as the separation score between two sample classes, which ranges from .50 (both classes datasets cannot be separated, fully random) to 1.00 (both classes datasets are fully separated). For the binary classification process, ROC has the advantage of being able to show the outputs from all possible positive-negative discrimination thresholds and therefore has the ability to depict relative trade-offs between

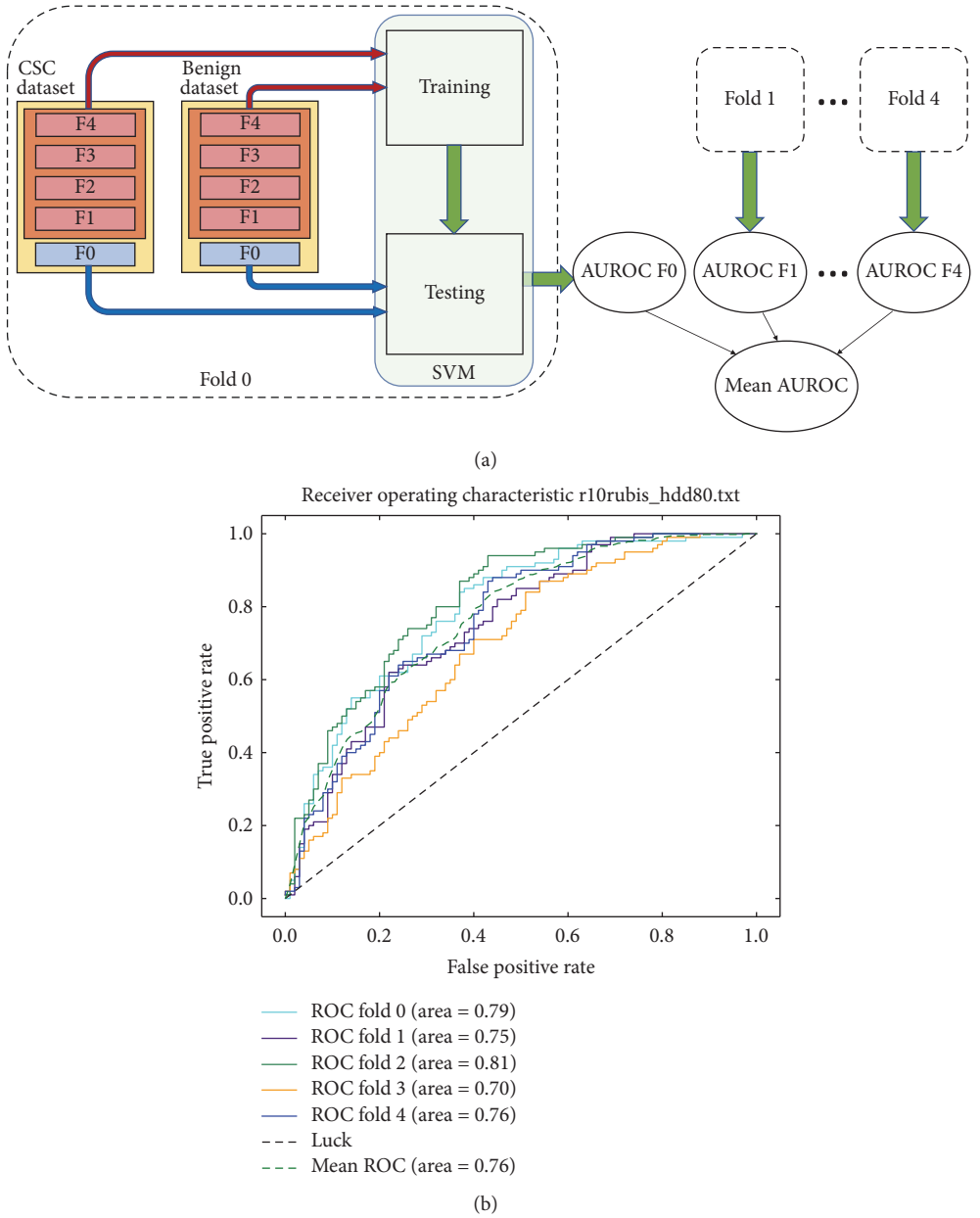

Figure 3: (a) Dataset distribution using the k-fold cross-validation technique to find an AUC value from a pair of CSCa and non-CSCa datasets for the SVM classification. (b) An example of a 5-fold cross-validation ROC graph.

the number of true positives (benefits) and false positives (costs). To create the ROC graph, instead of using the binary nonprobabilistic output of the SVM model, we used the distance of data point to the SVM model decision boundary as the input for ROC.

The scheme for dataset treatment and an illustration of its outputs are shown in Figure 3.

6.2. The Binary Class Classification for the CSCa Detection. The reason for a machine learning implementation is to use all the information one can get in the learning process. A server in the cloud is most likely performing only a small set of tasks, such as a web server, file server, or mail server. This means that having training data samples for the negative class (non-CSCa scenario) in real life is not difficult. We used this assumption to evaluate our dataset in a binary class classification approach by providing both positive class and negative class datasets for the training stage.

To evaluate this approach, we created two superset classes called the trained class and the untrained class. The trained class was the set of scenarios that were already known by the system and would be used for the training phase. The untrained class was the collection of scenarios that were not known previously by the system; therefore they were not used in the training process and would only be used in the test phase. We divided the scenarios of the positive class into

Table 2: The arrangement of scenario datasets for the binary SVM evaluation.

Trained class		Untrained class	
Positive class	Negative class	Positive class	Negative class
Prime + Probe (Gruss) and Flush + Reload (Gruss) and Flush + Reload (Yarom) scenarios are combined into one dataset	Idle and RUBiS 20 clients and RUBiS 200 clients and stress CPU and stress memory scenarios are combined into one dataset	Flush + Flush (Gruss) Flush + Reload (Hornby)	RUBiS 2000 clients Mail server Urandom generator Lucas-Lehmer Binary tree

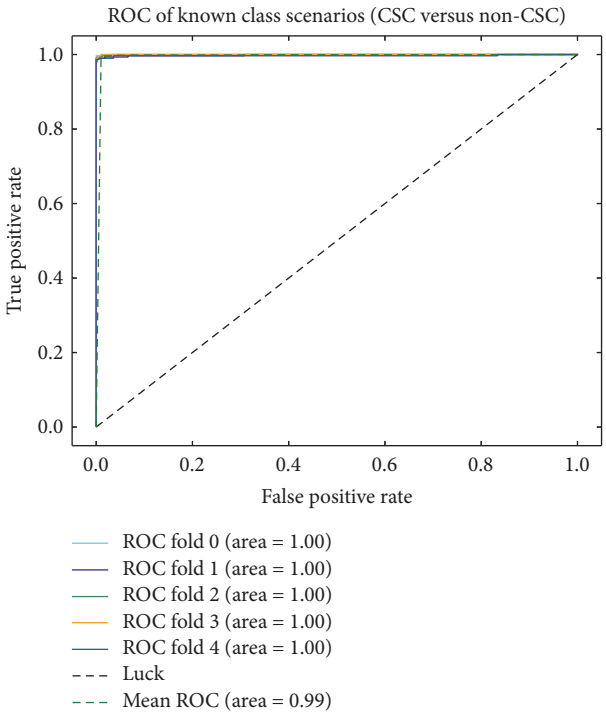

Figure 4: The classification ROC of the trained CSCa scenario test data and the trained non-CSCa scenario test data.

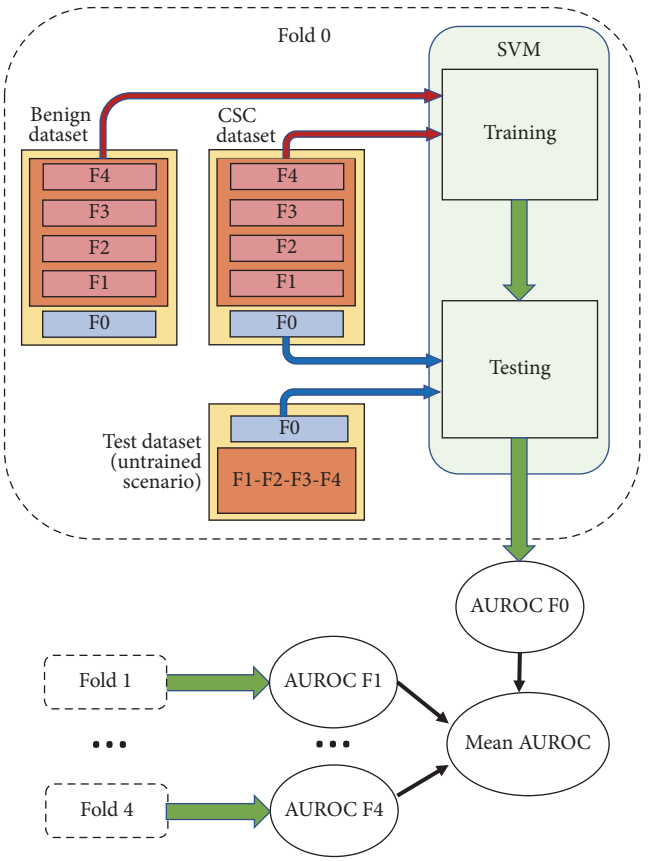

Figure 5: The evaluation scheme for each of the untrained scenario.

the trained-positive class and the untrained-positive class. We also divided the scenarios in the negative class into two, the trained-negative class and the untrained-negative class. The arrangement of all collected scenarios for use in this evaluation process is given in Table 2.

6.2.1. Test of the Trained Scenario. Our first test deals with the data that belong to the trained scenario class but not included in the training process. The aim is to see if the trained model was able to represent the trained scenario class in general. The procedure of the test is given in Figure 3(a). In this test, we do not yet use the untrained class scenarios of Table 2. The results of this test are given in Figure 4.

The results show that the detection system can successfully classify the data from all the scenarios that have been trained into CSCa and non-CSCa classes (0.99 AUC). This further shows that there are differentiable patterns of KVM event sequences between the trained CSCa scenarios and non-CSCa scenarios.

6.2.2. Test of the Untrained Scenario. In this second test, we wanted to see if the trained model was able to represent both classes, the positive class and negative class, in general. Therefore, we used the scenarios from the untrained class for the test phase. To achieve the concept of a signature-based detection system, in the test phase, the untrained class scenarios were compared against the trained-positive class dataset. The procedure of this test is given in Figure 5. The expected results should give a low AUC value (around 0.50 AUC) for the untrained-positive class scenarios and high

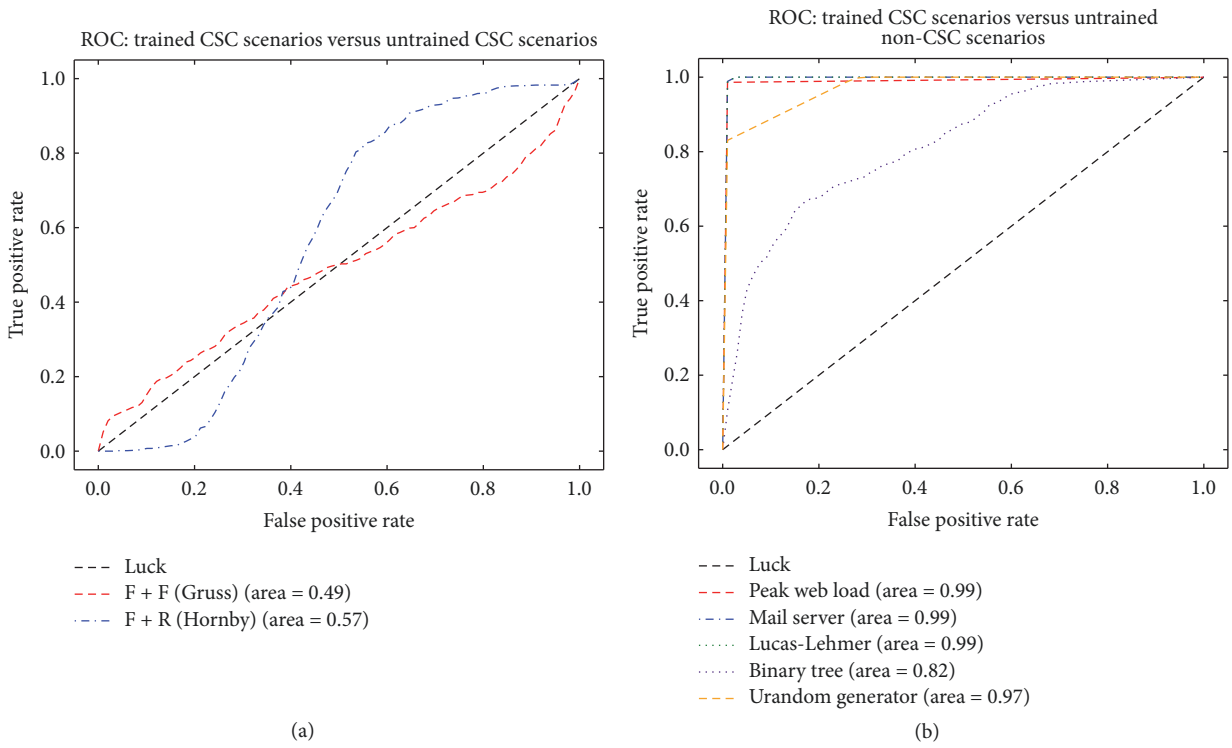

Figure 6: (a) Binary classification results for the Trained Positive Class versus each of the Untrained Positive Class (CSC) test scenarios. (b) Binary classification results for the Trained Positive Class versus each of the Untrained Negative Class (non-CSC) test scenarios.

AUC value (around 0.99 AUC) for the untrained-negative class scenarios. The actual results are given in Figure 6.

Figure 6(a) shows near to 0.50 AUC score for Flush + Flush scenario data (=0.49) and Flush + Reload Hornby scenario data (=0.57). This shows that the machine learning trained model cannot differentiate between the known CSCa scenario dataset and the unknown CSCa scenario dataset. This is the expected result as it means that the model created by the SVM training process was able to capture the common features of all CSCa and therefore will have low false negative rate detections.

On the other hand, Figure 6(b) shows near to 0.99 AUC score for the peak web workload scenario data (=0.99), mail server operation scenario data (=0.99), Lucas-Lehmer Test scenario data (=0.99), Binary Tree Operation scenario data (=0.82), and urandom generator scenario data (=0.97). This shows that the detection system was able to differentiate between the known CSCa scenarios and the unknown non-CSCa scenarios. This further means that the KVM event sequence training model was able to capture the generic differentiable features between CSCa operation and non-CSCa operation, which leads to low false positive rate detection.

In our test case, the binary tree scenario gave a smaller separation score in comparison to the other non-CSCa scenarios. We believe the reason for this score is the lesser number of arithmetic operations within the binary tree program. An in-depth explanation of the generic differentiable features between CSCa operation and non-CSCa operation is given in the next section.

6.3. Generalizing the Classification Results. In the previous sections, we showed that our monitoring system worked successfully against the scenarios that we prepared. Even though we showed that our system also works for the scenarios that were not yet trained, we still need to show that our solution can work in general for all other possible scenarios. To explain the separation between the CSCa class scenarios and the non-CSCa class scenarios, we made the effort to identify the exact KVM event sequences that separate CSCa operation and non-CSCa operation. First, we divided the non-CSCa scenarios into three different operation types: regular operations, CPU intensive operations, and Memory intensive operations. Then, we used the Fisher Score [52] approach to look for the most important features that separated each non-CSCa operation type dataset from the CSCa dataset. Fisher score comparison is a well-known method to find the optimal features, so that the distances between data points in the same class are minimized and the distances between data points of different classes are maximized. Even though the discrimination process between the SVM method and the Fisher score are not the same, we believe the results of this Fisher Score evaluation can give basic insight on the class discriminatory features. The results of this evaluation are given in Table 3.

Table 3 lists only five of the highest Fisher score features for each non-CSCa operation type dataset when compared to the CSCa dataset. Besides the Fisher scores, we also listed the median, average and standard deviation value of each feature to give a basic statistical perspective of the separation.

Table 3: Five features with the highest Fisher score for each non-CSCa scenario operation type compared to the CSCa scenarios. Note: ":" symbol represent delimiter.

Vs CSCa	KVM event sequence	Fisher score	Non-CSCa Seq. Stat.			CSCa Seq. Stat.		
			Med.	Mean	St. dev.	Med.	Mean	St. dev.
Regular operation	MSR_WRITE:kvm_apic:kvm_msr:	28.4	132	133.5	20.7	1	1.3	4.5
	HLT:kvm_eoi:kvm_pv_eoi:kvm_apic_accept_irq: kvm_inj_virq:	26.7	45	45.1	8.6	0	0	0
	HLT:kvm_inj_virq:	23.8	87	87.3	16.1	0	0	0
	MSR_WRITE:kvm_apic:kvm_apic:kvm_apic_ipi: kvm_apic_accept_irq:kvm_msr:	19.5	109	108.2	21.6	0	0.1	2.8
	HLT:kvm_eoi:kvm_pv_eoi:kvm_inj_virq:	17.2	312	305.8	67.3	0	0	0
CPU-intensive operation	EXCEPTION_NMI:kvm_fpu:	9.3	9	9.2	2.2	0	0.5	0.6
	EXTERNAL_INTERRUPT:kvm_fpu:	4.7	7	7.7	3.1	2	1.6	0.9
	CR_ACCESS:kvm_cr:kvm_fpu:	0.8	0	0.4	2.4	3	2.6	0.9
	PENDING_INTERRUPT:kvm_inj_virq:	0.4	2	3.2	6.6	167	106.5	84.3
	EXTERNAL_INTERRUPT:kvm_apic_accept_irq: kvm_inj_virq:	0.3	255	255.4	6.1	94	152.8	82.6
Memory-intensive operation	EXCEPTION_NMI:kvm_page_fault:kvm_emulate_insn:	434.9	1004	1002.1	15.7	0	0.7	18.3
	EXCEPTION_NMI:kvm_page_fault:kvm_inj_exception:	87.1	5058	4949.2	233.6	0	5	189.8
	EXCEPTION_NMI:kvm_page_fault:kvm_apic_accept_irq: kvm_inj_virq:	22.9	22	22.2	4.2	0	0	0.4
	EXCEPTION_NMI:kvm_page_fault:	19.3	5128	5630.6	983.2	0	7.9	283.8
	EXCEPTION_NMI:kvm_page_fault:kvm_emulate_insn: kvm_apic_accept_irq:kvm_inj_virq:	9.5	11	11.2	3.6	0	0	0

6.3.1. Regular Workload. In the case of a regular workload, such as web server operation and mail server operation, Table 3 shows there were a high number of VM exits on the Model Specific Register (MSR) writing operation to access the Advanced Programmable Interrupt Controller (APIC) chip in the non-CSCa scenario. This shows that, in comparison with the CSCa operation, the regular workload scenario operation produced more software and hardware interrupts. Another important VM exit shown in the table is HLT. *hlt* is an instruction to halt the CPU until it receives the next external interrupt requesting its service. The table shows that the regular scenario operations in the guest were not using the CPU intensively and therefore fired more *hlt* instructions to save the CPU power usage and heat output. The CSCa, on the other hand, were using the CPU extensively, hence the rare *hlt* calls.

However, a quick look at the entire raw data of the regular workload operation scenario is enough to easily discriminate the CSCa and non-CSCa data. There are several other features (KVM event sequences) besides the five listed in Table 3 that can be used to differentiate CSCa and non-CSCa operation. We believe this is because the regular non-CSCa operation works with diverse workload types and resources and therefore creates many different KVM event sequences, while the CSCa operations work uniformly with only a small set of suboperations (timing operation, read or write specific memory addresses and cache flushing). With knowledge of the difference in patterns of KVM event sequences between our regular operation scenario and the CSCa, we can safely extrapolate that the classification results would be the same for other regular operations within the public guest VM.

6.3.2. CPU Intensive Workload. Manual observation of the raw data shows an almost similar pattern between the CSCa scenarios and the CPU intensive non-CSCa scenarios. Table 3 for CPU-intensive operation shows that only two of the five features listed (EXCEPTION_NMI - kvm_fpu and EXTERNAL_INTERRUPT - kvm_fpu) can actually be useful for classification (the Fisher Scores are higher than 1). Both of these sequences are related to the use of the Floating Point Unit (FPU). In comparison to the CSCa attack, common CPU intensive non-CSCa operations are usually related to complex mathematical-related operations. On the other hand, CSCa does not need any complex mathematical operations and therefore can be discriminated from the CPU intensive non-CSCa operation using the sequence of FPU utilization. Examples of CPU intensive workload are cryptography operations.

6.3.3. Memory Intensive Workload. We also checked the discriminatory features between the CSCa scenarios and the memory intensive non-CSCa scenarios. All the features in Table 3 on memory intensive operation show high Fisher scores, which means that the CSCa operations can easily be separated from the non-CSCa memory intensive operation. The table shows that the memory intensive non-CSCa scenarios create a lot more page fault exceptions than the CSCa operations. Page fault exceptions may happen for

Table 4: Fisher score for the evaluation on another host with different microarchitecture.

	Sequence	Fisher score
Regular load	HLT:kvm_inj_virq:	26.74
	EXCEPTION_NMI:kvm_page_fault: kvm_inj_exception:	26.18
	MSR_WRITE:kvm_apic:kvm_msr: kvm_apic_accept_irq:	24.43
	HLT:kvm_eoi:kvm_pv_eoi:kvm_inj_virq:	20.65
	CR_ACCESS:kvm_cr:	18.79
CPU-intensive load	EXCEPTION_NMI:kvm_fpu:	9.13
	EXTERNAL_INTERRUPT:kvm_fpu:	6.76
	EXTERNAL_INTERRUPT: kvm_apic_accept_irq: kvm_inj_virq:	0.52
	EXTERNAL_INTERRUPT: kvm_apic_accept_irq:	0.51
	PENDING_INTERRUPT:kvm_inj_virq:	0.29
Memory intensive load	EXCEPTION_NMI:kvm_page_fault: kvm_inj_exception:	91.42
	EXCEPTION_NMI:kvm_page_fault: kvm_emulate_insn:	75.46
	EXCEPTION_NMI:kvm_page_fault:	43.69
	EXCEPTION_NMI:kvm_page_fault: kvm_inj_exception:kvm_apic_accept_irq: kvm_inj_exception:	15.31
	EXCEPTION_NMI:kvm_page_fault: kvm_apic_accept_irq:kvm_inj_virq:	12.84

two reasons, either because there is no translation for the memory address or because there is no access right for the specified address. In short, the high number of page fault exceptions in the memory intensive non-CSCa scenarios points to diverse memory address access, in contrast to the CSCa scenarios that focused on accessing only a small set of memory addresses.

6.3.4. Evaluation on Different Microarchitecture. As a part of the microarchitecture attack class, CSCa are characterized by the type of the CPU architecture of the physical host. To check the impact of different types of CPU architecture on the results of our monitoring method, we performed the same Fisher score evaluation as above on a host with a different microarchitecture. We set up the host on a Dell Poweredge R910 machine which is equipped with two Intel Xeon Quad-Core E7520 1.86 GHz (Nehalem), 36 MB L3 cache, and 32 GB system memory. We choose this specification as it has a different Last Level Cache (LLC) layer and different chipset architecture compared to our main evaluation setup (Section 6.1.1). Table 4 lists the five highest Fisher score features from each of the non-CSCa operation type datasets compared to the CSCa dataset on the Nehalem-based host.

Table 3 (Conroe setup) and Table 4 (Nehalem setup) show that the two highest Fisher score features that differentiate the CSCa scenario dataset and non-CSCa CPU intensive scenario dataset in the Conroe setup and Nehalem setup are the same. The similarity of the higher Fisher score set also happened in the case of the non-CSCa memory intensive dataset differentiation (4 out of 5 similar features). This shows that the operational characteristics of the non-CSCa CPU-intensive scenario and memory intensive scenario on both microarchitectures are similar and thus can be captured through KVM events observation.

On the other hand, for the regular operation datasets in the Conroe and the Nehalem setups, there were four out of five different features in the set of the five highest Fisher score features that differentiated between the CSCa scenario and the non-SCSa regular operation datasets. We believe this result could be expected since there are many features that can be used to differentiate these operations and their Fisher scores might change slightly with each evaluation, thus changing the Fisher score ranking. However, the high Fisher scores show that even though the order of ranking is different, the regular non-CSCa operation scenario and the CSCa scenario can still be easily differentiated.

6.4. On the Case of Noisy Environments and Mimicry Attempts. In this evaluation part, we examine the performance of our approach against two types of attack evasion scenarios. First is having to detect CSCa within a noisy environment. In this scenario, the adversaries try to run their CSCa attack, while, either intentionally or unintentionally, there are other benign operations running in the VM (e.g., web server transactions). Second is having to detect a modified CSCa process that tries to mimic benign operation to evade any detection process.

(1) Noisy environment: we collected another dataset of the positive class (CSCa class). This time, we ran the CSCa in the guest VM while at the same time processing a significant web application workload.

(2) Mimicry attempt: we collected several new datasets from a modified CSCa that slightly altered its behavior to obfuscate its signature characteristics.

   (a) We reduced the spy frequency by increasing the waiting interval between cache access timing. We modified the Gruss's Flush + Reload implementation by increasing the number of yield operations between each timing process (Algorithm 4). We tried 100 and 1000 yield repetition values.

   (b) We added a diversion function inside the real CSCa code. We added a read and write file operation between cache access timing operations

```
...
start = rdtsc ();
While (1){
      flush_flush (addr + offset);
      for (int i=0; i<1000, ++i)
            sched_yield ();
}
...
```

Algorithm 4: An example of a mimicry attempt by reducing the spy frequency.

```
...
start = rdtsc ();
While (1) {
      flush_flush (addr + offset);
      diversion_func ();
      sched_yield ();
}
...
```

Algorithm 5: An example of a mimicry attempt by introducing a diversion function.

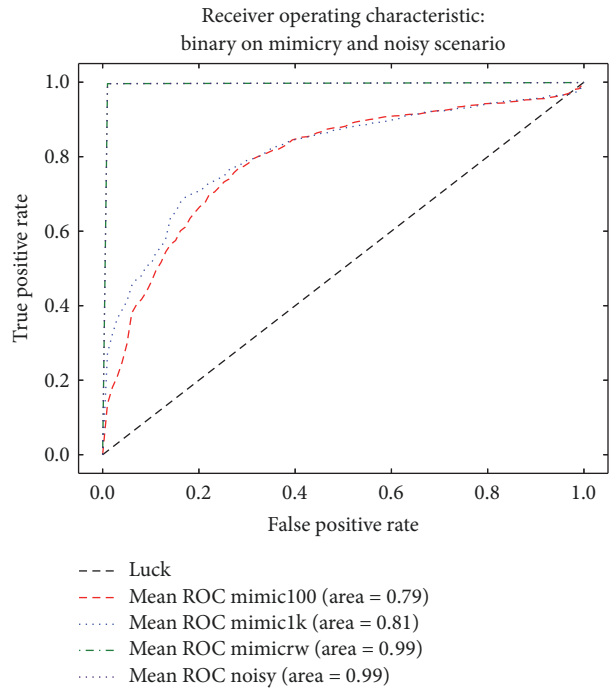

Figure 7: ROC of several mimicry attack and noisy case scenario.

in the Gruss's Flush + Reload implementation (Algorithm 5).

Using the previous SVM Binary Class Classification, the results are given in Figure 7. We can see that, in both cases, the noisy environment and mimicry attempts, the AUC values are high (0.79 and 0.81 for frequency alteration and 0.99 for both noisy environment and R/W mimicry attempt). These results point to high false negative detections. This shows that our detection method is still vulnerable to the scenarios of a noisy environment or mimicry attack. The poor results on detecting the mimicry CSCa are actually a common consequence for any indirect observation. Since we are not directly observing the target, the adversary can always create a diversion to hide their true acts.

However, looking from a different perspective, we believe that working in a noisy environment will also significantly decrease the CSCa effectiveness, making it impractical, and therefore would be avoided by the attacker. The same thing would happen in the mimicry attack. CSCa is actually a highly focused operation and requires a high level of information granularity. An attempt to obfuscate its procedure will highly reduce the granularity of the collected information. This is especially true for the Flush + Flush attack where the timing differences of clflush() hits and misses are very small. These requirements will limit the type and amount of obfuscation an adversary can use [53–56].

To evaluate the impact of a noisy environment and mimicry attempts on the CSCa output, we performed a Flush + Reload attack against an AES implementation of OpenSSL (as attempted in [43]) with four conditions: clean implementation, noisy environment, R/W mimicry attempt, and reduced probing frequency scenarios. Figure 8 shows the comparison of the cache lines visible pattern in the case of k_0 = 0xf_ between a clean Flush + Reload implementation and a frequency-reduced Flush + Reload implementation. We highlighted all cache line entries that were hit at least 99% times the number of encryptions. The number of encryptions that were required to produce less than 2% pattern errors are given in Table 5.

Table 5 shows that a noisy environment, R/W mimicry attempt, or reduced probing frequency will decrease the effectiveness of the CSCa attacks. In our case, the noisy environment and mimicry attack scenarios reduced the accuracy to 25% and 20%, respectively. In the case of reduced probing frequency, we could not capture the cache-line pattern with less than 2% error after up to 10000 trial encryptions. The high standard deviation for the Noisy scenario shows that the load fluctuation in the background will affect detection accuracy. Finally, the mimicry attack will add to the computational load of the spy process and lead to some additional processing time, reducing the CSCa resolution timers and increasing the probability of missing the real encryption events from the victim.

Basically, while noisy environments and mimicry may obfuscate the CSCa signatures, these also make the CSCa less effective. We did not study the way to tackle this noisy and mimicry problem within this work as we believe this problem is quite big and challenging for another future work of its own.

6.5. Performance Impact of the Monitoring Process to the Host and Guest VM. We also tested the scalability of our monitoring approach by increasing the number of monitored VMs from 1 up to 8 guest VMs and measured the time needed

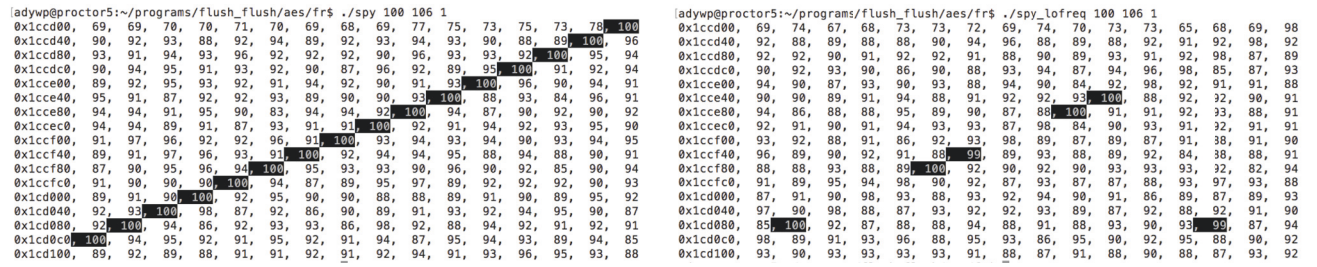

Figure 8: Cache line pattern of k_0 = 0xf_ (a) for clean CSCa implementation (b) for the CSCa with a reduced probe frequency.

Table 5: Comparison between clean, noisy, mimicry, and reduced probe frequency scenarios.

Scenario	(a)	(b)	(c)
Clean	44	7.8	75.93
Noisy (200 CU)	178	82.3	78.86
Mimicry R/W	209	10.5	134.22
Reduced Probing freq.	NA	NA	761.83

(a) Number of encryptions needed to create a cache line pattern of the upper 4 bits of k_0 with less than 2% error (average of 10 attempts); (b) standard deviation of (a); (c) CPU task-clock needed to find the pattern for 100 encryptions (average of 10 attempts).

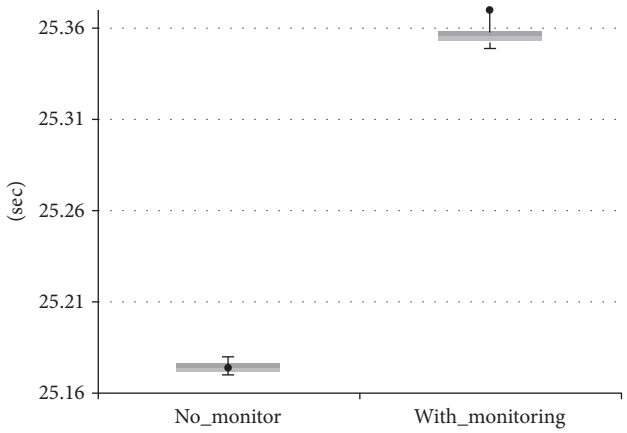

Figure 9: The comparison of time that was needed to calculate 10000 first prime numbers in the guest VM when the KVM events at the host was monitored and when it was not monitored.

to collect 1 unit of observation data. We used Linux's *perf* tool and collected the task-clock data (the CPU time). We found out that the trace-cmd KVM tracing process did not increase CPU utilization even if the number of monitored VMs was increased (at least up to 8 guest VMs in our experiment). The task clock for collecting data remained constant with an average of 0.0443 msec and standard deviation of 0.00176 msec.

Next, we compared the CPU performance of a guest VM with no monitoring and when it is being monitored by the host. For this measurement process, we used the sysbench tool. For this benchmark, we recorded the total execution time of one thread to calculate the first 10000 prime numbers. Figure 9 presents the averages from twenty benchmarking results.

The boxplot shows that the monitoring process in the host had a small impact on the computation performance of the guest VM. In this experiment, there was an increase of 0.7% in the time to complete the task in the guest system when it was monitored from the host using our approach (KVM event observation).

7. Discussion

7.1. Considerations for Implementation in Operational Environment. The procedures used in this study were set up for experimentation purpose. To have a working operational system, we need to determine the explicit threshold for positive-negative decision and the explicit number of positive results threshold to decide when to fire the alarm.

7.1.1. Positive-Negative Discrimination Threshold. The Receiver Operating Characteristic (ROC) curve shows the whole spectrum of possible discrimination thresholds and therefore is useful for selecting the optimal criterion (maximize the true positive rate and minimize the false positive rate). Theoretically, the optimum threshold that maximizes the trade-off between the true positive rate and false positive rate can be derived from the ROC using the Youden Index [57]. The Youden Index J is formulated as

$$J = \text{Sensitivity} + \text{Specificity} - 1, \quad (1)$$

where Sensitivity refers to the true positive rate and Specificity refers to the true negative rate. Graphically, the index can be explained as a single operating point of the ROC with the maximum distance from the chance (diagonal) line.

In practice, the optimum threshold from the Youden Index is not always applicable. That is because the Youden Index gives both false positive and false negative the same

weight (cost). In real-life operation, the operator might apply a different weight to the false positives and false negatives. If we apply a different weight to FP and FN as α and β, respectively, we can write a cost function C as follows:

$$\begin{aligned} C &= \mathrm{FPR}\alpha\left(1-p\right) + \mathrm{FNR}\beta p \\ &= \mathrm{FPR}\alpha\left(1-p\right) + \left(1-\mathrm{TPR}\right)\beta p, \end{aligned} \tag{2}$$

where p is the ratio of positive events from the total events:

$$\begin{aligned} p &= \frac{\mathrm{TP}+\mathrm{FP}}{\mathrm{TP}+\mathrm{FP}+\mathrm{TN}+\mathrm{FN}} \\ \mathrm{FPR} &= \frac{\mathrm{FP}}{\mathrm{FP}+\mathrm{TN}} \\ \mathrm{TPR} &= \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}. \end{aligned} \tag{3}$$

The gradient of the cost function C is any line with coefficient $c = \alpha(1-p)/\beta p$. The optimal threshold that produces a minimum cost is the intersection of the line with coefficient c and the ROC.

7.1.2. Alarm Threshold. Raising an alarm based on only one unit of observation is not suggested as it has a high probability of introducing false alarms from outlier events. We argue that, by assessing the detection status in groups of sequenced observation data (a decision window), the accuracy can be increased. For example, having a decision window of 10, we might choose to raise the alarm anytime it contains 7 positive observations. The proper value for the window size and positive data threshold can be varied for different types of implementations. Finding the optimum value of the observation window size and the threshold value for positive data is a good new research subject.

7.2. Potential Use of KVM Event Data. In the evaluation section, we have shown that even though the KVM event sequences are not directly related to internal CSCa functions, this dataset can still be used to differentiate between CSCa operations and non-CSCa operations. This leads us to believe that the KVM event sequence information can also be used for other more generic monitoring functions, such as an Anomaly Detection System. We can use the same approach we used in Section 6.4, but instead of comparing the incoming data (or the test data) with a specific attack patterns, we can compare it with the benign class scenario which will make this system work as an anomaly detection system.

8. Conclusion

This work is a feasibility study of using KVM events information to detect the cache-based side channel attacks (CSCa). Within this paper, we have shown that CSCa create several unique patterns of KVM event sequences. These patterns can be used to detect the existence of any CSCa variants, including the Flush + Flush attack, within a guest VM. The monitoring system which collected the KVM events does not need any host or guest VM modification. It can work inside the host without guest participation. Furthermore, it only has a small impact on the guest performance and almost zero impact on the host performance which can lead to a highly scalable monitoring system.

We showed that, by using the KVM event sequences for the Support Vector Machine classification method, the separation score of our trained CSCa scenarios and trained non-CSCa scenarios was 0.99 AUC (Area Under the Curve of Receiver Operating Characteristic). The separation score between the trained CSCa scenarios and the untrained CSCa scenarios, which includes the Flush + Flush attack, was close to 0.50 AUC, while the separation score between the trained CSCa scenarios and the untrained non-CSCa scenarios was close to 0.99 AUC. These results show that the KVM events monitoring method can provide low false negatives and low false positives for a CSCa detection system. To strengthen our claim, we performed Fisher score evaluation and successfully identified the KVM event sequences that generalize the separation of the CSCa and non-CSCa operation dataset.

Our further investigation on false negatives showed that our detection method still did not address evasion techniques such as the noisy environment and mimicry attack scenarios. However, we also showed that both scenarios negatively affected the CSCa effectiveness, thus limiting these options for the adversary.

Finally, we evaluated the computation overhead impact of our CSCa monitoring approach and showed that it has a negligible overhead on the host and the guest VM operations.

We believe the results of these experiments are useful to broaden the understanding of CSCa in particular and the operation of CPU caches in general. Our findings can benefit future research in this field to help identify ways to detect CSCa.

We identify several research direction to move forward:

(i) We would like to design an operational version of this detection system. This is not a trivial task because there are many different functions to adapt from the current experimental implementation, such as real-time data collection, preprocessing, and analysis, along with developing a process to find the proper threshold for a positive or negative detection decision.

(ii) An interesting case is to find the solution of CSCa monitoring for other processor architecture, such as the ARM processors which has gained more popularity recently.

(iii) Another challenging problem to be solved is detecting any effort to obfuscate the CSCa in noisy environments or with mimicry operations. A potential approach would be by using the combination of multiple monitoring techniques such as Hardware Performance Counter (HPC), KVM events, and another probing point available from the VMM.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, 2016.

[2] J. Szefer, "Survey of microarchitectural side and covert channels, attacks, and defenses. Cryptology ePrint Archive," Report 2016/479, 2016.

[3] A. K. Biswas, D. Ghosal, and S. Nagaraja, "A survey of timing channels and countermeasures," *ACM Computing Surveys*, vol. 50, no. 1, article no. 6, 2017.

[4] Security aspects of virtualization, "The European Union Agency for Network and Information Security," Technical Report, February 2017.

[5] D. J. Dean, H. Nguyen, and X. Gu, "UBL: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in *Proceedings of the 9th ACM International Conference on Autonomic Computing, (ICAC '12)*, pp. 191–200, USA, September 2012.

[6] F. Doelitzscher, M. Knahl, C. Reich, and N. Clarke, "Anomaly detection in IaaS Clouds," in *Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science, (CloudCom '13)*, pp. 387–394, UK, December 2013.

[7] S. S. Alarifi and S. D. Wolthusen, "Detecting anomalies in IaaS environments through virtual machine host system call analysis," in *Proceedings of the 7th International Conference for Internet Technology and Secured Transactions, (ICITST '12)*, pp. 211–218, December 2012.

[8] A. S. Abed, T. C. Clancy, and D. S. Levy, "Applying bag of system calls for anomalous behavior detection of applications in linux containers," in *Proceedings of the IEEE Globecom Workshops, (GC Wkshps '15)*, IEEE, USA, December 2015.

[9] W. Sha, Y. Zhu, M. Chen, and T. Huang, "Statistical learning for anomaly detection in cloud server systems: A multi-order Markov chain framework," *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, 2015.

[10] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Leveraging forensic tools for virtual machine introspection," Technical Report GT-CS-11-05, Georgia Institute of Technology, 2011.

[11] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy, (SP '11)*, pp. 297–312, USA, May 2011.

[12] Y. Fu and Z. Lin, "Bridging the semantic gap in virtual machine introspection via online kernel data redirection," *ACM Transactions on Information and System Security*, vol. 16, no. 2, article no. 7, 2013.

[13] W.-M. Hu, "Lattice scheduling and covert channels," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 52–61, May 1992.

[14] D. J. Bernstein, "Cache-timing attacks on aes," Technical Report, The University of Illinois at Chicago, 2005.

[15] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Proceedings of the 2006 The Cryptographers' Track at the RSA conference on Topics in Cryptology, (CT-RSA'06)*, vol. 3860 of *Lecture Notes in Computer Science*, pp. 1–20, Springer, Berlin, 2006.

[16] Y. Yarom and K. Falkner, "Flush + reload: a high resolution, low noise, l3 cache side-channel attack," in *Proceedings of the 23rd USENIX conference on Security Symposium, (SEC '14)*, pp. 719–732, August 2014.

[17] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest OS," in *Proceedings of the 4th Workshop on European Workshop on System Security, (EUROSEC '11)*, Austria, April 2011.

[18] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in JavaScript and their implications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, (CCS '15)*, pp. 1406–1418, USA, October 2015.

[19] T. Hornby, *Side-channel attacks on everyday applications*, Black hat, USA, 2016.

[20] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard, "Practical Keystroke Timing Attacks in Sandboxed JavaScript," in *Proceedings of the 25th USENIX Security Symposium*, vol. 10493 of *Lecture Notes in Computer Science*, pp. 549–564, Springer International Publishing.

[21] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*, pp. 199–212, November 2009.

[22] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security, (CCS '12)*, pp. 305–316, USA, October 2012.

[23] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in PaaS clouds," in *Proceedings of the 21st ACM Conference on Computer and Communications Security, (CCS '14)*, pp. 990–1003, USA, November 2014.

[24] M. S. İnci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache attacks enable bulk key recovery on the cloud," in *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems, (CHES '16)*, vol. 9813, pp. 368–388, August 2016.

[25] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *Proceedings of the 2008 - 41st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-41*, pp. 83–93, IEEE, Italy, November 2008.

[26] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture, (ISCA '07)*, pp. 494–505, ACM, USA, June 2007.

[27] F. Liu and R. B. Lee, "Random Fill Cache Architecture," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, (MICRO '14)*, pp. 203–215, UK, December 2014.

[28] Y. Zhang and M. K. Reiter, "Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, (CCS '13)*, pp. 827–837, ACM, November 2013.

[29] F. Liu, Q. Ge, Y. Yarom et al., "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proceedings of the 22nd IEEE International Symposium on High Performance Computer Architecture, (HPCA '16)*, pp. 406–418, IEEE, Spain, March 2016.

[30] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security, (CCS '16)*, pp. 871–882, Austria, October 2016.

[31] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Mascat: Stopping microarchitectural attacs before execution," *Cryptology ePrint Archive*, 2016.

[32] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: a tool for the static analysis of cache side channels," in *Proceedings of the 22nd USENIX conference on Security, (SEC '13)*, vol. 18, pp. 431–446, 2013.

[33] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, article no. 35, 2012.

[34] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: system-level protection against cache-based side channel attacks in the cloud," in *Proceedings of the 21st USENIX conference on Security symposium*, August 2012.

[35] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops, (DSN-W '11)*, pp. 194–199, China, June 2011.

[36] Y. Han, T. Alpcan, J. Chan, and C. Leckie, "Security games for virtual machine allocation in cloud computing," in *Proceeding of the 4th International Conference on Decision and Game Theory for Security, (Gamesec '13)*, vol. 8252 of *Lecture Notes in Computer Science*, pp. 99–118, November 2013.

[37] S.-J. Moon, V. Sekar, and M. K. Reiter, "Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, (CCS '15)*, pp. 1595–1606, USA, October 2015.

[38] Y. Zhang, M. Li, K. Bai, M. Yu, and W. Zang, "Incentive compatible moving target defense against VM-colocation attacks in clouds," in *Proceedings of the IFIP International Information Security Conference on Information Security and Privacy Research, (SEC '12)*, pp. 388–399, Springer, 2012.

[39] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.

[40] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *Proceedings of the Research in Attacks, Intrusions, and Defenses. (RAID '16)*, vol. 9854 of *Lecture Notes in Computer Science*, pp. 118–140, Springer.

[41] M. Payer, "HexPADS: A platform to detect "stealth" attacks," in *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems, (ESSoS '16)*, vol. 9639, pp. 138–154, Springer-Verlag.

[42] N. Herath and A. Fogh, *These Are Not Your Grand Daddy'S cpu Performance Counters*, Black hat, USA, 2015.

[43] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush + Flush: A fast and stealthy cache attack," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, (DIMVA '16)*, vol. 9721, pp. 279–299, Springer-Verlag, July 2016.

[44] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of The 10th Annual Network and Distributed System Security Symposium*, 2003.

[45] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Ligauri, "Kvm: the linux virtual machine monitor," in *Proceedings of the 2007 Ottawa Linux Symposium - OLS07*, 2007.

[46] A. Arcangeli, I. Eidus, and C. Wright, *Increasing Memory Density by Using Ksm*, Red Hat Inc, 2009.

[47] G. Venkitachalam and M. Cohen, "Transparent page sharing on commodity operating systems," *Patent US7500048 B1*, 2009.

[48] S. Rostedt, "Ftrace kernel hooks, more than just tracing. LINUX Plumbers Conference".

[49] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "Performance comparison of middleware architectures for generating dynamic web content," in *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference*, June 2003.

[50] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "Training algorithm for optimal margin classifiers," in *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory (COLT '92)*, pp. 144–152, ACM, July 1992.

[51] F. Pedregosa, G. Varoquaux, and A. Gramfort, "Scikit-learn: machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[52] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, vol. 7, pp. 179–188, 1936.

[53] G. Irazoqui, M. S. IncI, T. Eisenbarth, and B. Sunar, "Know Thy neighbor: crypto library detection in cloud," in *Proceedings on Privacy Enhancing Technologies*, vol. 2015.

[54] A. Fogh, "Cache side channel attacks: Cpu design as a security problem. Hack In The Box".

[55] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! A fast, cross-VM attack on AES," in *Proceeding of the Research in Attacks, Intrusions and Defenses, (RAID '14)*, vol. 8688, pp. 299–319, Lecture Notes in Computer Science, 2014.

[56] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing - And its application to AES," in *Proceedings of the 36th IEEE Symposium on Security and Privacy, (SP '15)*, pp. 591–604, USA, May 2015.

[57] W. J. Youden, "Index for rating diagnostic tests," *Cancer*, vol. 3, no. 1, pp. 32–35, 1950.