Doctoral Dissertation

A Dynamic Analysis with Static Source Code Instrumentation for the Guest Monitoring Problem in Virtualization Environments

Ady Wahyudi Paundu

December 13, 2019

Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Ady Wahyudi Paundu

Thesis Committee:

Professor Youki Kadobayashi	(Supervisor)
Professor Kazutoshi Fujikawa	(Co-supervisor)
Professor Hajimu Iida	(Co-supervisor)

A Dynamic Analysis with Static Source Code Instrumentation for the Guest Monitoring Problem in Virtualization Environments*

Ady Wahyudi Paundu

Abstract

Even though the cloud technology shows the trend of rapid development and decreasing cost, it still has not been fully embraced by organizations and industries around the world. This reluctance mostly stems from the cloud system security issues. One of the main potential attack vectors in a cloud system is the guest Virtual Machine (VM). Therefore, it is necessary to provide a system to monitor the guest VM operations. In the public cloud model, there are several operational requirements for the monitoring program. First, the monitoring system must work separately outside of the guest VM. Separating the monitoring process and the monitored system can deny any malicious processes in the monitored system from compromising the monitoring agent. However, this separation requirement could lead to the semantic gap problem. A good monitoring system is expected to choose the observation data that preserve the semantic information as much as it can. Second, the monitoring system must be able to work without any cooperation from the guest VM. The guest VM should not even realize the existence of the monitoring program. Therefore, for the third requirement, the monitoring program should cost, in terms of computation resources usage, as efficient as possible.

In this thesis, we investigate a guest VM monitoring method that can work independently outside the monitored guest VM, without losing much of the semantic information and without high computation cost for either the host and

*Doctoral Dissertation, Graduate School of Information Science,
Nara Institute of Science and Technology, December 13, 2019.

the guest VM. We propose a method that embeds multiple tracepoints inside the source code of the hypervisor (Static Instrumentation). During the hypervisor operation, we collect the tracepoints execution data to dynamically monitor the operational flow of a guest VM (Dynamic Source Code Analysis). Since the instrumentation was carried out within the underlying process of the instances of guest VM, we believe that the dynamic pattern of the tracepoints sequences can indirectly describe the operations of the VM.

We first applied this dynamic source code analysis with a static instrumentation method to the user space of the Qemu-KVM hypervisor. We captured the tracepoints from the Qemu operation and used it for an Anomaly Detection System. We emulated a web server VM and multiple attack scenarios, such as DDoS for network-based attack and Flush-Reload attack for virtualization-based attack. We factored in the mimicry attack scenario. We compared several machine learning algorithms for the monitoring data analysis process. Finally, we compared our detection result with system-call data analysis. Our evaluation showed that monitoring guest VM using dynamic source code analysis with the static instrumentation method gave better detection results compared to the system-call data, with minimum computation cost. However, we had subpar results when trying to detect malicious activities that work upon the host CPU. That is because, on Qemu-KVM combination, CPU operations are performed natively through the KVM kernel module.

We investigated further this dynamic source code analysis with the static instrumentation method at the kernel layer by instrumenting the KVM module. We used this method to implement a signature-based intrusion detection system and try to detect multiple variants of Cache-based Side-Channel Attack (CSCA) including a new stealthier variant called Flush+Flush attack. In our evaluation phase, we showed that our proposed approach is the first successful attempt to detect this Flush+Flush attack in the virtualization environment.

Keywords:

virtualization technology, cloud security, hypervisor operation, machine learning, program analysis, virtual machine monitoring

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Cloud computing is a new paradigm in the IT industry that offers many advantages over the conventional computing model. The advantages range from the lower overall cost to the operational convenience and improvements. However, aside from its rapid development and decreasing cost, the cloud computing system has not been fully embraced by organizations and industries around the world. In Europe, for example, only 26% of EU enterprises used cloud computing in 2018, mostly for email hosting and file storage only [eur18]. In the case of public cloud IaaS, only 18% of financial services firms in Europe broadly implementing IaaS for production applications today [fin19].

This reluctance to the cloud computing adoption mostly stems from the cloud security concerns [VB16, SN17, XX12, RANR15]. Most of the security concerns are related to the loss of control to the data and the low of trust to the cloud provider. In its recent report on cloud security [isc19], the International Information Systems Security Certification Consortium ((ISC)^2) disclose that 93% of organizations are moderate to extremely concerned about cloud security. Moreover, they revealed that data security (29%) and general security risks (28%) are the top two barriers holding back cloud adoption in organizations.

Overall, it is important to emphasize that the cloud security teams must reassess their security posture and strategies to address their shortcomings in protecting the evolving cloud computing environments.

1.1 The Rogue Tenant: A Problem in the Cloud

One of the main potential security threats in a cloud system is the guest Virtual Machine (VM) [BFS+16, SL16, XX12]. An attack from the VM is considered as a cloud-specific vulnerability since it is intrinsic to or prevalent in a core cloud

computing technology [GWS11]. In their Special Publication series number 800-125A, the National Institute of Standards and Technology (NIST) identifies the threats emanating from rogue or compromised VMs as one of the primary sources of threats to a hypervisor platform [Cha18]. These threats can arise through channels such as shared hypervisor memory and virtual network inside the hypervisor host. The rogue VM threats can manifest through the breach of process isolation, the breach of network isolation, or the denial of service attack. Ristenpart et al. demonstrated a well known working example on how to use a guest VM to attack the peer VM and the hypervisor [RTSS09]. The demonstration showed that key risks could arise from sharing physical infrastructure between mutually distrustful users, even when their actions are isolated through machine virtualization. Not only for attacking their peers, but a malicious VM has also become an attractive option for the criminals as a powerful low-cost tool for cybersecurity attacking devices [HSJ, Kol15].

Since the guest VMs are one of the prime sources of threats to the hypervisor, continuous monitoring of the state of VMs is necessary. In regards to VM monitoring, the NIST Special Publication: *Security Recommendations for Hypervisor Deployment on Servers* gives two recommendations [Cha18].

Security Recommendation HY-SR-14 : There should be a mechanism for security monitoring, security policy enforcement of VM operations, and detecting malicious processes running inside VMs and malicious traffic going into and out of a VM.

Security Recommendation HY-SR-15 : Solutions for Security Monitoring and security policy enforcement of VMs should be based outside of VMs and leverage the virtual machine introspection capabilities of the hypervisor.

1.2 Motivation and Problem Statement

A common approach to monitoring a computation system is by planting a monitoring agent (or process) inside the system, e.g., an antivirus program. All methods that adopt similar approach are categorized as Host-based Intrusion Detection System (HIDS). However, numerous past studies and experiences sug-

gest that we should separate the monitoring agent and the observed objects on different planes [GR03, MKA+13, Cha18, JWX07, SL16, PZH13]. The purpose is to deny attempts from any malicious processes in the observed system to compromise the monitoring program. In the case of the cloud system, especially in public IaaS, there is another requirement of non-intrusiveness. In this non-intrusiveness requirement, the monitoring process, from outside of the monitored guest VM, cannot depend on the guest VM to provide any monitoring data. The host administrator cannot just ask the guest VM users to install some agent programs or send specific files to the host administrator. The monitoring process should be invisible from the guest VM perspective. In both separation and non-intrusiveness requirements cases, the implementation of traditional HIDS is not feasible. Furthermore, both requirements will increase the logical distances between the observer and the observed object, thus reduce the information quality. Such a problem is known as the semantic gap problem.

Another monitoring category called Network-based Intrusion Detection System (NIDS) monitors network activities to detect any threats to the system. However, the cloud system, with its virtualization technology, has many characteristics that enable the offenders to use mediums other than the network to launch attacks. Examples are the family of side-channel attacks that use cache access time to spy peer VMs and the fuzzing attacks that try to probe the hypervisor weaknesses. The NIDS approach is inadequate against these types of attacks.

Virtual Machine Monitor (VMM)-based IDS were introduced to tackle the limitations of HIDS and NIDS in the virtualization environment. There are several techniques that have been proposed to monitor the guest VM from the hypervisor. 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). Figure 1.1 shows the variety of monitoring points of observation inside a Qemu-KVM hypervisor.

Computation metric monitoring analyzes the performance metric of the guest VM. Examples of such metrics are CPU utilization, memory utilization, and the volume of hard-drive read and write operation. The primary assumption of this approach is that malicious activity will likely change a considerable amount of computing resources. For example, a substantial amount of memory utilization

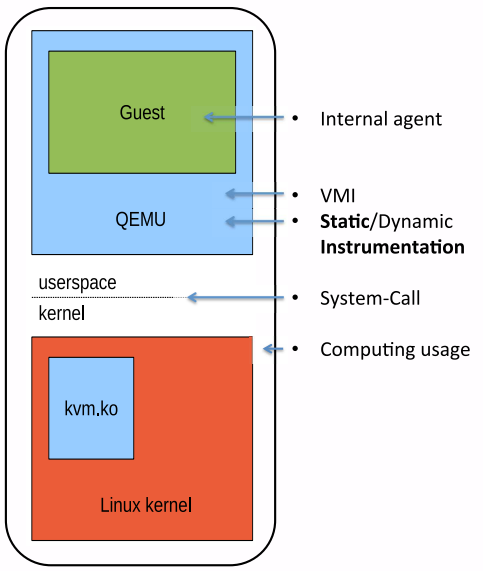


Figure 1.1: Multiple options for observation points inside Qemu-KVM

might indicate a memory leak incident while a sudden increase in network traffic might indicate a network-based Denial of Service (DoS) attack. This monitoring approach, however, can only be practical if implemented inside the observed system. This is because most of the resources for guest VM are pre-allocated and not allocated on an as-needed basis. The process of memory space allocation and other processes inside the guest VM are hidden from the host system. To perform an effective VM monitoring, the virtualization operator usually utilizes agents inside the guest VM in order to collect the usage data. Ganglia* is an example of the monitoring system that collects resource metrics from cloud infrastructure at any scale.

Another standard VM-based monitoring utilized the System Call. System-call is a set of interfaces that enable user processes to get access to services that are provided by the Operating System (OS) kernel. Examples of the services provided by the OS kernel are the I/O operations, such as reading from or writing to the CPU, the main memory or any other computer devices. By observing the system-calls invocation from user processes to the underlying kernel system, a security agent can try to make conclusions whether the user processes constitute

*http://ganglia.info

a normal operation or not. In the context of the virtualization environment, the process of system-call monitoring is not trivial. The hypervisor as an intermediate system between the guest OS and the host OS adds another layer of system-call simulation. Hence, the system-calls that are collected in the host are, for the most part, not the direct representations of the internal guest OS system-calls.

The latest and more sophisticated approach to monitoring the guest VM operations is Virtual Machine Introspection (VMI). It works by capturing a snapshot of the memory space used by the guest VM. With this snapshot, a monitor can reconstruct an exact same picture of the situation inside the guest VM. This technique of "inspect a VM from the outside to asses what is happening on the inside" was first introduced by Garfinkel et al. (2003) [GR03]. Some examples of information that can be captured by VMI are the list of running processes and the status of the file system in the guest OS. The features of VMI shows that the VMI approach is currently the best option to monitor the guest VM without having any inside agents. For its clear visibility, VMI can be used by a host system to detect malware and rootkit scanners, also to perform integrity checking and forensic analysis in guest VMs. This capability is usually offered as an additional service on demand by the cloud provider (Security as a Service). One minor limitation of the current VMI implementations is their dependency on some specific data from the guest OS to correctly reconstruct the raw memory snapshots into useful information. As instances of such data, we can cite the debugging symbols information files for Windows systems or memory offset information file for Linux systems. Since these data are different for each OS type and OS version, they should be copied from the guest OS to the monitoring program in the host system. This requirement is easy to satisfy in a private environment. However, in a particular arrangement such as public IaaS, this approach could be hard to implement.

The explanation of three common VM-based monitoring above shows that there are still room for improvements. 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 [FL13]. Furthermore, requirements in a public cloud limit the access of a cloud administrator to internal information from the guest system. Therefore, the motivation of our work is to find

a monitoring data source within a hypervisor system that can give high-quality information for a Virtual Machine Monitor (VMM)-based monitoring system and can be collected efficiently without any cooperation from the guest VM.

1.3 Contribution

In this work, we study a new method to monitor the operation of a guest VM within a virtualization environment, especially to identify if that monitored VM is operating normally or running some malicious operations. The challenge of this work is to find a monitoring data source that can offer high-quality information for a Virtual Machine Monitor (VMM)-based monitoring system and can be collected efficiently without any cooperation from the guest VM.

We give two main contributions in this work.

First, we promote the use of KVM-Qemu static instrumentation tracepoints as a novel way to monitor the operation of a guest VM. We performed this promotion with empiric evaluations of multiple facets of the virtualization monitoring issue:

- Two Layer of Hypervisor Modules. In the KVM-Qemu combination hypervisor, each software serves a different purpose. Qemu works at userspace to emulate the hardware while the KVM works at kernel space to enable Qemu takes advantage of hardware virtualization extensions present in modern Intel and AMD CPUs for safely executing guest code directly on the host CPU. We study the dynamic analysis of the source code both at Qemu and KVM level.

- Multiple Data Analysis Methods. The data that is collected in our monitoring method is in the form of big text data. This data would be hard to be processed manually by human operators. Therefore, we chose to implement a Machine Learning approach to process the monitoring data. In this work, we study three types of Machine Learning approach. First, we used the Bag of Tracepoint approach, where we use the pattern of each tracepoint quantities. Second, we used the Tracepoint sequence pattern. Third, we used the combination of both the tracepoint sequence and the pattern of its quantities. Since each of those Machine Learning approaches require a

specific form of input, we also study the pre-processing method for each of the Machine Learning approaches to maximize the quality of data analysis results.

- Multiple Malicious and Anomaly Operations. Previous studies on anomaly detection in the cloud mostly evaluated their work using anomaly or malicious scheme that significantly change the pattern of computing resources usages, such as high CPU operations or busy I/O transactions. These approaches are indeed useful for detecting volume-based attacks. However, in reality, many attacks on the computer system do not change the patterns of resource usage data and hence are harder to detect. Higher semantic information is needed to detect those non-volume based attacks. To show that our monitoring method gives a rich semantic data, we evaluate them using multiple types of attacks, ranging from network-based attack and host-based attacks to cloud-based (virtualization-specific) attacks. A vulnerability or an attack is cloud-specific if it is intrinsic to or prevalent in a core cloud computing technology [GWS11].

Our second contribution is specifically aimed at Cache-based Side Channel attack (CSCa) detection. While promoting the use of KVM-Qemu static instrumentation tracepoints as a novel way to detect an attempt of CSCa within a virtualization environment, we introduce a first method to detect the Flush+Flush variant of CSCa. As the most recent CSCa variant, Flush+Flush, showed that the previous CSCa detection methods, which heavily depend on Hardware Counter, could be easily bypassed. Not only we were able to show a high degree of detection accuracy using our method, we were also able to offer an insight into why our classification works by extracting the set of most important features that separate both CSCa classes and Normal class and further show that our monitoring approach can work to detect CSCa in general.

1.4 Thesis Structure

The remainder of the thesis is structured as follows:

Chapter 2 : offers basic information on the Virtualization technology. First, we

give some fundamental notions of the Cloud Computing system and how it all related to Virtualization technology. Next, we dive into more detail about the Hypervisor (or Virtual Machine Monitor), what is it and how it works. Finally, we give an overview of the hypervisor we use in this thesis, the Qemu-KVM hypervisor.

Chapter 3 : presents details about the program analysis technique. The chapter first explains the classification of the program analysis technique. Using the classification information, we continue to explain how the program analysis method that we use in this thesis, the Dynamic Analysis with Static Source Code Instrumentation, works. This chapter also gives background information on how program analysis has been used in the field of Computer Security.

Chapter 4: details the methodologies we propose to detect anomaly in guest VM using the tracepoints we collected from the Qemu static instrumentation approach. We introduce three machine learning algorithms to analyze the monitoring data for an Anomaly Detection System. For each machine learning analysis, we give their detail works and their anomaly detection results. Our test also includes the scalability of the monitoring system and how much the monitoring affect the guest VM performance.

Chapter 5 : details the methodologies we propose to detect Cache-based Side Channel attack (CSCa) operation inside a guest VM using the tracepoints we collected from the KVM kernel module static instrumentation approach. We give a detail explanation of how our CSCa detection system works, the way we evaluate it and the results. This chapter also includes a preview of what is Cache-based Side Channel attack and how it works.

Chapter 6 : The future work is detailed in this chapter. Besides extensions of the work we accomplished in this dissertation, we propose different distinct projects that can be good avenues towards the security of cloud computing in particular and computer systems in general.

Chapter 7 : This chapter concludes the thesis.

2 Virtualization: The Nuts and Bolts

2.1 Cloud Computing and Virtualization Technology

Once a simple iconic symbol to represent the internet and cyberspace, the cloud has been transformed within the last decade into a more specific idea of Cloud Computing. There are many definitions that have been given to the term Cloud Computing; however the most cited one was coined by the U. S. National Institute of Standards and Technology (NIST):

> "Cloud computing is a model for enabling ubiquitous, convenient, on demand network access to a shared pool of configurable computing resources (e.g. network, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [MG11], p.2.

The NIST document defines five essential characteristics of Cloud Computing, which are: on-demand self-service, broad network access, resource pooling, rapid elasticity and measured service. In specific to resource pooling characteristics, the Cloud Computing model requires the computing resources to be pooled to serve multiple users in a multi-tenant model dynamically. This requirement is accomplished using Virtualization technology.

Virtualization is a computation concept that provides a virtual version, instead of the original physical version, of a device or resource, be it a server, a desktop, a storage device, an operating system or network resources, to the users. Human users, devices and applications can interact with the virtual resources as if it were

a real single physical resource. This technique allows a single physical resource to be pooled and accessed by multiple users and hence makes the computing system more scalable, efficient and economical.

Virtualization is achieved through the use of a software called Hypervisor. A Hypervisor connects directly to the system hardware. It allows splitting one system into separate, distinct, and secure environments known as virtual machines (VMs).

2.2 Hypervisor

A hypervisor is a computer function to create and manage virtual machines. It provides an efficient, isolated duplicate of the physical machines for virtual machines. The hypervisor is also sometimes referred to as Virtual Machine Monitor (VMM). A hypervisor can be implemented in software or hardware form. The physical machines running the hypervisor are called the Host, while the virtual machines running on top of the Host are usually called the guest VM.

There are several types of hypervisor. Based on how they provide the hardware to their guests, the hypervisors can be categorized into three types:

- Binary Translation Based Hypervisors.

  These hypervisors translate the binary code of the process inside the guest OS to another binary to be processed by the hypervisor. This translation means that the input contains a full instruction set, but the output is a subset thereof and contains the innocuous instructions only [AA06]. These hypervisors include the interpreter for the binary translation. The binary interpretation function in this hypervisor type is similar to the emulation techniques, where the hypervisor provides emulated hardware functions to the guest OS. It relies on binary translation to trap and virtualize the execution of sensitive, non-virtualizable instructions sets. The performance of binary translation based hypervisors is dependent on the instructions to be translated.

- Paravirtualized Hypervisors.

These hypervisors accelerate the communication process between the guest OS to the infrastructure layer by modifying the guest OS. The modified guest OS uses a different set of APIs to communicate with the hypervisor layer, henceforth removing the binary translation process and improve access to the infrastructure layer. However, this efficiency does come at the cost of flexibility and security. Since the operating system must be modified to run with the paravirtualization, a particular OS or distribution may not be readily available for the solution and reduce flexibility. Furthermore, as the guest OS has much closer control of the underlying hardware, the risk of impacting the lower hardware level is slightly increased, which could lead to impacting all guest systems on the host. Since this modified OS can only be used in the paravirtualization environment, therefore, the guest VM users (or at least the system administrators) are aware that their environment has been virtualized. Applications running atop the altered OS do not have to be changed.

- Hardware Assisted Hypervisors.

  These hypervisors accelerate the communication process between the guest OS to the infrastructure layer, especially to the hardware layer, through additional functionality included in the CPU. These additional functions work in an execution mode called guest mode, which is dedicated to the virtual instances [AA06, Dre08]. These hypervisors require specific hardware to operate, such as the virtualization technology which has been integrated on X86 processors (Intel VT-x and AMD-V) that allow the execution of privileged instructions directly on the processor, even though it is virtualized. Besides the handful of new instructions, these processor level virtualization functions also introduce a new privilege level. The hypervisor can now run at "Ring -1"; so the guest operating systems can run in Ring 0. As the results, there is no need for paravirtualization, the hypervisor does less work, and the performance hit is reduced.

Another popular classification classifies the hypervisors based on how the hypervisor connects to the hardware.

- Type 1 hypervisors.

These hypervisors are usually called native or bare-metal hypervisors. They are the hypervisors that are run directly above the physical hardware. These hypervisors act as the operating system for the physical hardware. The infrastructure is dedicated solely for the virtualization operations. Since these hypervisors can interrupt and access the physical hardware directly, their performance is much better than the Type 2 hypervisors.

- Type 2 hypervisors.

  These hypervisors are usually called a hosted hypervisor. They are the hypervisors that are run on top of a host operating system. These hypervisors work in the userspace of the hosting OS. This virtualization mode adds another abstraction layer for hardware access and hence penalize the guest OS performance. However, this virtualization mode enables the infrastructure (the physical hardware and the host OS) to serve other functions besides the virtualization.

The distinction between type 1 and type 2 hypervisor do not always clear. Certain solutions, such as the Qemu-KVM system, can be categorized in either type of hypervisors. Qemu is run on top of the Linux system and works by emulating the hardware to be used by its guest OSes and therefore is classified as type 2 hypervisor. However, the Qemu in its KVM accelerated mode can execute the operation in the guest OS natively by leveraging the KVM module, which is an integral component of the kernel of the host OS. Since the host OS kernel handles the guest executions, this approach is qualified as a type 1 hypervisor.

Besides all the above types of hypervisor, there is also another type of virtualization called *Containerization*. Containerization is OS-level virtualization. Unlike the other virtualizations above, this virtualization mode emulates an operating system rather than the underlying hardware. A container relies on virtual isolation to deploy and run applications that access a shared operating system (OS) kernel without the need for virtual machines (VMs). Figure 2.1 depict the comparison between Type I Hypervisors and Containers.

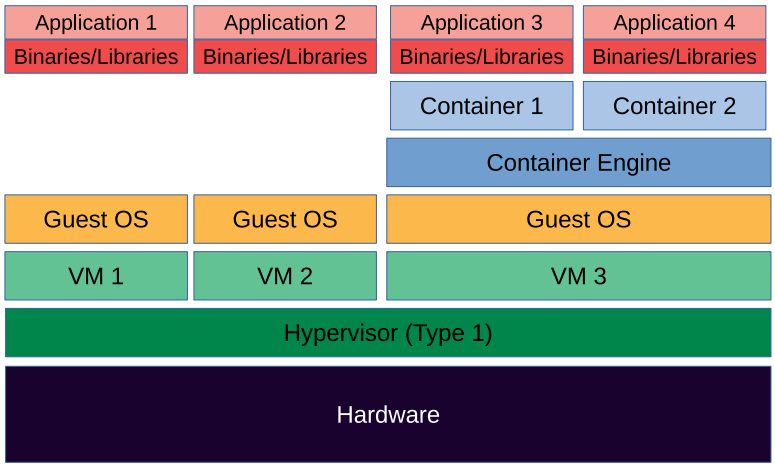


Figure 2.1: Diagram of comparison between the Type I hypervisor and the Container

2.3 The Qemu-KVM Hypervisor

There are four hypervisors provide up to 93% of the commercial market, VMware, Hyper-V, XEN, and KVM [PBSL13]. For the biggest IaaS providers, Xen and KVM are the hypervisors being used the most. Xen offers several advantages over KVM, such as the efficiency of paravirtualization, which exceeds what is available in KVM due to the closer access Xen has to the physical hardware. Xen has been used as the primary hypervisor by Amazon Web Service (AWS). However, since 2017, AWS has announced its new instance package, coded C5, uses a new KVM-based home-brewed hypervisor. Along with that, the AWS also announced that their future development on their instances would be based on the KVM [aws17]. KVM, on the other hand is the leading hypervisor used by Google Cloud and Digital Ocean. Since the future of type I hypervisors is shifting towards the Qemu-KVM, in this study, we use the Qemu-KVM hypervisor.

2.3.1 Qemu

Qemu is an open-source emulator to virtualize hardware in a computation system. Qemu can emulate multiple computing architectures, such as x86, PowerPC, SPARC (Scalable Processor Architecture) and ARM (Advanced Reduced

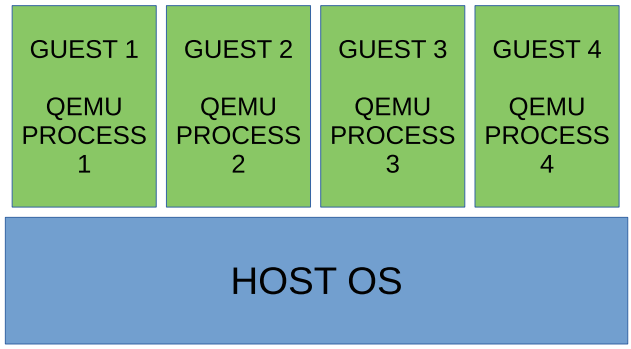


Figure 2.2: Each VM is served using separate userspace process

instruction set computing Machine) architecture. On its own, Qemu is a Type 2 Hypervisor (hosted virtual machine monitor) since it runs on top of a conventional operating system (OS). However, working in tandem with KVM, Qemu-KVM can be considered as a Type 1 Hypervisor (bare metal virtual machine monitor). That is because KVM is an internal part of Linux OS kernel, therefore the functions of Qemu-KVM can be seen as an integral function of the OS.

The core of QEMU is based on event-driven architecture. Event-driven architecture means that Qemu reacts to events by running a main loop that dispatches to event handlers. However, Qemu also combines the event-driven architecture with threaded architecture to take advantage of the multi-core computer that is common nowadays. Threaded (or parallel) architecture means that Qemu splits work into processes or threads that can be executed simultaneously.

Qemu have two mechanisms to execute user code, which are Tiny Code Generator (TCG) and KVM. TCG emulates the guest using dynamic binary translation. This technique is also known as Just-In-Time (JIT) compilation or the emulation mode. TCG transforms target instructions to TCG operators which are then transformed into host instructions. KVM or the acceleration mode, on the other hand, takes advantage of hardware virtualization extensions present in modern Intel and AMD CPUs for safely executing guest code directly on the host CPU.

In Qemu, each VM is served using a separate userspace process (see Figure 2.2.). When a guest shuts down, the Qemu process exits. Guest RAM is allocated when the Qqemu starts up. The RAM is mapped into the Qemu processes address space and acts as the physical memory for the guest. The host kernel schedules

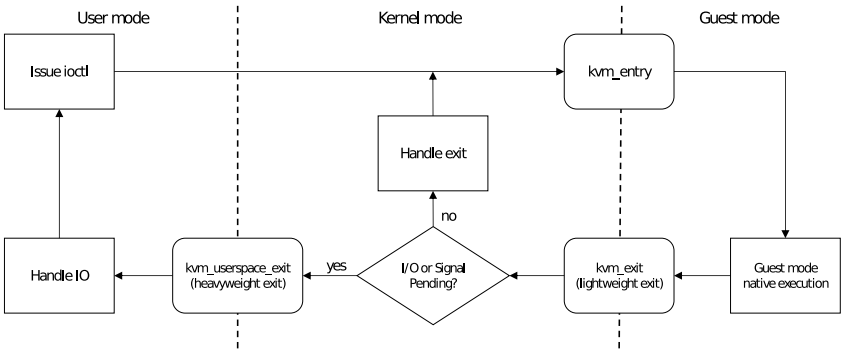


Figure 2.3: Guest VM execution loop

```
open("/dev/kvm")
ioctl(KVM_CREATE_VM)
ioctl(KVM_CREATE_VCPU) for (;;) {
        ioctl(KVM_RUN)
        switch (exit_reason) {
                case KVM_EXIT_IO: /* ... */
                case KVM_EXIT_HLT: /* ... */
        }
}
```

Algorithm 1: The pseudocode of basic KVM operation

Qemu like a regular process. Multiple guest VMs run alongside without the knowledge of each other. Applications like Firefox or Apache also compete for the same host resources as each guest VMs (i.e., Qemu process) although resource control operations can be used to isolate and prioritize Qemu. Since Qemu system emulation provides a full virtual machine inside the Qemu userspace process, the details of what processes are running inside the guest are not directly visible from the host.

2.3.2 KVM

A Kernel Virtual Machine (KVM) [KKL+07] 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 (type-1 virtualization).

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:

- 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 2.3). 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.

- the operation switches between the Kernel mode and Guest mode, which include control registers, segment registers and instruction pointers, are performed by the hardware.

- 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 Qemu calls KVM_RUN 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 Qemu intervention is required to execute an IO task, control is further transferred to the Qemu 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. This basic flow of a guest CPU is given as pseudocode in Algorithm 1.

3 Program Analysis Techniques

Programming a software is not a simple thing to do. Many aspects need to be considered to produce a good program, such as resource consumption efficiency, error-free operation and operational security. The dynamic and diversity of programming environments and programming tools exacerbate the complexity of one control over the program. Therefore, any mean of monitoring over such a program can be invaluable. One method to have a deeper understanding of a computer program is called Program Analysis technique.

Program analysis is a process to analyze the behavior of a given computer program. The result of the analysis process can be used to either improving the program performance with efficient computing resources or to ensure that the program does what it is supposed to do.

This dissertation is focused mainly on the aspect of program correctness, which is to monitor and try to infer what is the program, in our case, the Virtual Machine is currently doing. In this chapter, we will give further information on program analysis and how it has been used in the computer security field.

3.1 Types of Program Analysis

There are several methods to perform program analysis. In this section, we cover the common taxonomy of program analysis, which can be classified based on the form or based on the state of the objects being analyzed.

3.1.1 Based on the Form of Objects

The form of the program can either be source code or a binary form. Source code is a set of instructions and statements in plain text that is created by a programmer using a particular computer programming language. This code is later will

be used to create the program. Since a computer cannot directly recognize the source code, the code needs to be translated to a form known by the computer. A binary is the form of the code after compilation or interpretation that can be directly understood by a computer.

Source Code Analysis.

Source code analysis performs its operation over the source code objects. An example of source code analysis is the way a compiler works. A compiler is a program that translates the source code into a machine-readable format. The analysis in this category is not only directed to the source code but also can be performed to any program representations that are derived directly from the source code, such as control-flow graphs. The features of these analyses are commonly in the form of programming language constructs, such as functions, statements, expressions, and variables.

These analyses are dependent on the chosen programming language. Even though performing one similar function, multiple programs that were written with different programming languages will require multiple program analyses. However, these analyses is compiler, platform and operating system agnostic. An advantage of source code analysis is that this analysis has access to high-level information, and hence can be easily maintained and understood by unsophisticated users.

Binary Analysis.

In a binary analysis, the objects are in the form of compiled code. Not only works with raw binary machine code forms, but this category also includes the executable intermediate representations, such as byte-codes, which run on a virtual machine. The features of these analyses are commonly in the form of machine entities, such as procedures, instructions, registers and memory locations.

In contrary to the source code analysis above, binary analyses are programming language independent but platform-specific. Therefore, different machines or different operating systems will require different analysis programs. The main advantage of binary analysis is that one does not need to have the source code of the program and all the third-party libraries to perform the analysis.

3.1.2 Based on the State of Objects

There are two states of the program where it can be analyzed, which are in a static state or a dynamic state. A static state refers to the state where the program is not running. The dynamic states refer to the state where the program is executed and part or the whole of the programs are copied to the memory.

Static Analysis

In static analysis, the program is analyzed without running [Bin07]. This approach scans the application source code and examines all possible execution paths without executing the application. Static analysis processes are fast, repeatable and not dependent on the fact whether the program can be executed or not. A compiler is an example of the tool performing static analysis. Common static analysis operations include analyses for correctness, such as type checking, and analyses for optimization, which identify valid transformations to improve performance. A static analysis tool can be used to debug the program's source code or can be used to help visualize the code. The static analysis tool only needs the text of the source code to perform the program analysis. The main advantage of static analysis is its ability to consider all execution paths in the program. However, since the analysis works with simulated value, the validity of analysis depends on the completeness of the testing scenario.

Dynamic Analysis

Dynamic analyses are performed during the program runtime [Bal99]. Examples of classes that perform dynamic analysis are the profilers, checkers and execution visualizers. Since the analysis processes are performed in real operation and work with real values, the analysis results are considered more precise than the static analysis [DMD+02]. The downside is, this analysis can only consider one execution path at a time [Ern04].

Dynamic analyses work by instrumenting the program. Instrumentation means the acts to add monitoring codes, called instrumentation codes or analysis code or probes or tracepoints, into a program. The tracepoints can be embedded inline of the program or it can include external routines that will be called when the

tracepoints are fired. The tracepoints will not change the way program works but may add extra work to the host. Every tracepoint contains information of analysis state or metadata that heavily used in the analysis process.

There are two methods of instrumentation for the program's dynamic analysis. These methods are classified based on when the instrumentation takes place.

Static Instrumentation. This instrumentation occurs before the program is run, either by modifying the source code or by modifying the binary code. The modifications are saved to the disk. This approach does not need any additional program at runtime and hence will not affect the main program performance. This instrumentation can cover all paths of the program's code without any dependency on the program input. However, this approach requires the source code or the binary code of the program, which are not always available. Moreover, since this approach is performed during the static state of the program, this approach cannot be applied to the system that is already deployed.

Dynamic Instrumentation. This instrumentation occurs during runtime. The analysis codes or the tracepoints are injected either by a program grafted onto the client process or by an external process. The main advantage of this instrumentation approach is that we do not need to have the source code of the application that we need to analyze. Moreover, the analysis can be performed even when the application is already running. However, this instrumentation method is not able to achieve full code coverage. Furthermore, the fact that there is a need to embed an external process to instrument the original one will add the computation requirement and make the monitored object run slower.

3.2 Program Analysis for Computer Security

One purpose of the program analysis process is to observe the behavior of a program to ensure that the program works as it supposed to do. This function is in line with the purpose of functions in the field of computer security. Therefore it is common to find efforts in the field of computer security, either in the industry or in academics, that use the program analysis approach. The research to utilize

program analysis in the computer security field is categorized into two categories, static analysis and dynamic analysis. Both static analysis and dynamic analysis are complementary to their own merits and demerits [Ern04].

Static code analysis is performed on a model of the program to be analyzed. There are various techniques and mechanisms developed to detect security vulnerabilities in a program model. For example, two approaches to assess access control vulnerabilities, the Stack-based Access Control (SBAC) and Role-based Access Control (RBAC). SBAC systems ensure that only programs that satisfy a set of permission requirements gain access to restricted resources [BN05, LT13]. RBAC is an access control mechanism based on operations instead of resources. Therefore, in order to restrict access to sensitive data all operations accessing such data need to be identified [FK09, NBL+10]. To assess the information flow of a program, static analyses perform static check to the flow of information between variables in a program to see whether they are consistent with the security labeling of variables [PCFY07,SRK06]. Another use of static analysis is to assess API conformance of a program by using trivial syntactic code scanning or some more complex methods [PJAG12, SIH14].

From the explanation of static analysis above, it is clear that this approach can be applied as early as the development phase, which can help programmers to find and fix bugs and security lapses early. The main issue of the static analysis approach for computer security is the fact that the observation surface, i.e., all the possible scenario to be analyzed, is very vast, even in the face of current computing capability [And12]. That leads to the facts that most of the results of this approach still contain considerable high false positives and false negatives [WD12].

Most current software architecture is constructed as a collection of dynamically linked libraries. This architecture is making rendering static analysis imprecise. Moreover, static analysis is found to be ineffective in a dynamic environment that use features like dynamic binding, polymorphism and threads. On the other hand, dynamic analysis has the benefit of examining the actual domain of program execution [Bal99]. In general, dynamic analysis involves the recording of a program's dynamic state. Generally, a dynamic analysis technique involves these three steps. First, program instrumentation, second, profile/trace generation,

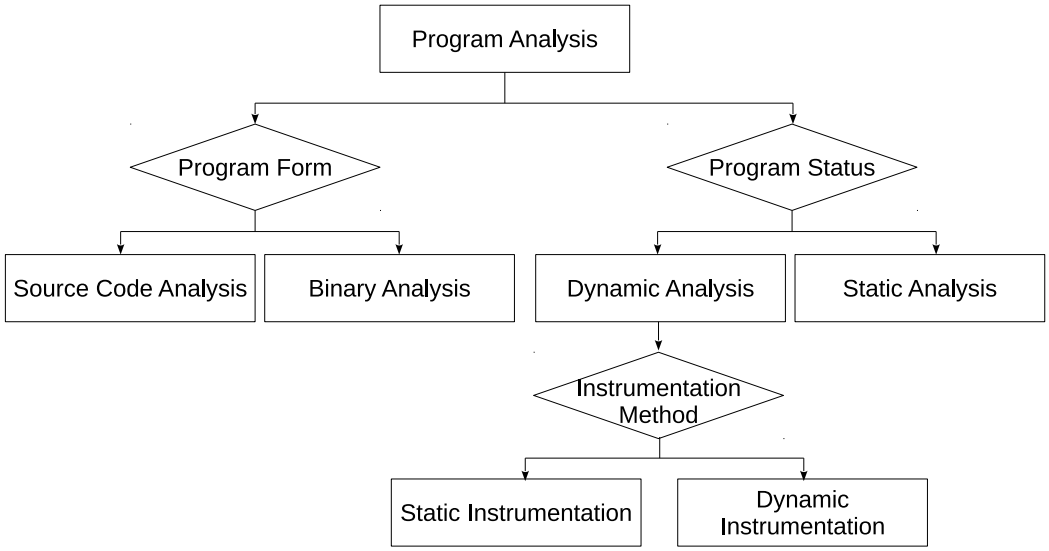


Figure 3.1: Program analysis taxonomy

and finally, analysis or monitoring. There are many study tools for an efficient and accurate dynamic analysis. Valgrind [NS07] is an instrumentation framework for building dynamic analysis tools. It can automatically detect many memory management and threading bugs, and profile a program in detail. Pin [SDC+10] is a tool for dynamic binary instrumentation of programs in the Microsoft Windows environment. Diduce [HL02] is a popular tool for online invariant detection. It works by dynamically hypothesizes invariants at each program point and only presents those invariants which have been found to satisfy a property. Caffeine [GDJ02] helps a maintainer to check conjectures about Java programs, and to understand the correspondence between the static code of the programs and their behavior at runtime. Another popular dynamic program analysis tool is Decaf [HPY+14]. Decaf provides a framework to introspect a virtual machine using taint propagation rules over the QEMU Tiny Code Generator (TCG) intermediate representation.

Since static analysis and dynamic analysis are complementary, both approaches can be combined into one framework [Che11, SA15]. The illustration of program analysis taxonomy is given in Figure 3.1.

3.3 Dynamic Analysis with Static Source Code Instrumentation

In this section, we will explain the program analysis that we used in this dissertation. In terms of the time of analysis, our work can be categorized as a dynamic analysis method. The analysis (or the observation) of the program is performed when the program is running. We intend to observe the dynamic behavior of the Virtual Machine, which is hard to achieve in the static analysis method. Another main consideration for choosing dynamic analysis is because we only want to observe the VM under a small set of well-defined scenarios and therefore, do not have to consider all possible execution paths.

In terms of the analysis objects, our work can be categorized as a source code analysis. It is because we used the tracepoints that were embedded at the source code, instead of the byte code or the binary code. Our primary consideration for choosing the source code analysis approach is the availability of the Qemu-KVM source code. Having the source code, we should have the opportunities to observe the detail operation of the Qemu-KVM hypervisor. However, the dimension of the Qemu-KVM program is enormous. There are more than 6000 *.c* object file that each contains a various number of methods. Instrumenting every function would be an arduous and inefficient process. An ideal approach would be to select only a small set of functions to be instrumented, based on a specific set of operations that need to be observed. That ideal requires a deep understanding of how each component of Qemu work. However, even though the Qemu-KVM system is among the most used open-source hypervisor for the public cloud, its official documentation lacks any information on its inner operational work. To resolve this issue, we applied a blind introspection approach by utilizing all the default debugging tracepoints provided by Qemu. The other reason for considering all default tracepoints is because we want to use the data for anomaly detection and not some specific attack signatures.

An example of a tracepoint is shown at Algorithm 2. It shows a function called bdrv_open_common that is part of the block.c class from Qemu. The block.c class contains the library for accessing the emulated block drive that is created by Qemu. The bdrv_open_common method describes the function to open a

```
...
static int bdrv_open_common(BlockDriverState *bs, BlockDriverState *file,
        QDict *options, int flags, BlockDriver *drv, Error **errp)
{
  ...
  trace_bdrv_open_common(bs, filename ?: "", flags, drv->format_name);
  ...
}
```

Algorithm 2: An example of static tracepoint implementation

```
26) proctor qemu:memory_region_ops_write: { cpu_id = 0 }, { vpid = 1772 }, {
65) proctor qemu:virtio_queue_notify: { cpu_id = 0 }, { vpid = 1772 }, { vde
84) proctor qemu:kvm_vcpu_ioctl: { cpu_id = 0 }, { vpid = 1772 }, { cpu_inde
17) proctor qemu:virtqueue_pop: { cpu_id = 1 }, { vpid = 1772 }, { vq = 0x7F
70) proctor qemu:from_qemu_sendv_packet_async: { cpu_id = 1 }, { vpid = 1772
42) proctor qemu:virtqueue_fill: { cpu_id = 1 }, { vpid = 1772 }, { vq = 0x7
44) proctor qemu:virtqueue_flush: { cpu_id = 1 }, { vpid = 1772 }, { vq = 0x
25) proctor qemu:virtqueue_pop: { cpu_id = 1 }, { vpid = 1772 }, { vq = 0x7F
95) proctor qemu:from_qemu_sendv_packet_async: { cpu_id = 1 }, { vpid = 1772
50) proctor qemu:virtqueue_fill: { cpu_id = 1 }, { vpid = 1772 }, { vq = 0x7
58) proctor qemu:virtqueue_flush: { cpu_id = 1 }, { vpid = 1772 }, { vq = 0x
91) proctor qemu:virtqueue_pop: { cpu_id = 1 }, { vpid = 1772 }, { vq = 0x7F
17) proctor qemu:from_qemu_sendv_packet_async: { cpu_id = 1 }, { vpid = 1772
64) proctor qemu:virtqueue_fill: { cpu_id = 1 }, { vpid = 1772 }, { vq = 0x7
93) proctor qemu:virtqueue_flush: { cpu_id = 1 }, { vpid = 1772 }, { vq = 0x
54) proctor qemu:from_tap_send: { cpu_id = 1 }, { vpid = 1772 }, { s = 0x7F3
33) proctor qemu:from_qemu_deliver_packet: { cpu_id = 1 }, { vpid = 1772 },
65) proctor qemu:virtqueue_pop: { cpu_id = 1 }, { vpid = 1772 }, { vq = 0x7F
42) proctor qemu:virtqueue_fill: { cpu_id = 1 }, { vpid = 1772 }, { vq = 0x7
91) proctor qemu:virtqueue_flush: { cpu_id = 1 }, { vpid = 1772 }, { vq = 0x
32) proctor qemu:virtio_notify: { cpu_id = 1 }, { vpid = 1772 }, { vdev = 0x
```

Figure 3.2: An example of a Qemu tracepoints observation log content.

specific block for IO operation. Inside this function, there is a line that calls another function named trace_bdrv_open_common(). This trace function, upon being called, will write the name of its host function name to an output stream (i.e., a log file) along with several other information about the host function. An example of the content a log file is given in Figure 3.2. The log consists of multiple metadata on multiple sequenced tracepoints within an execution flow of a Qemu's Virtual Machine (VM). The metadata not only includes the VM level data, such as the function name and parameter, execution time and CPU ID, it also gives the function level data, such as memory location, memory size, execution flag and other function-specific data.

We used the static instrumentation method to reduce the complexity of instrumentation process and also to reduce the hefty computation overhead that is common in the dynamic instrumentation approach.

This program analysis technique works by monitoring the patterns (either the frequency or the sequence) of function calling and events within the hypervisor during the Virtual Machine life operations. To identify each function and each event, we utilize multiple tracepoints that were distributed across multiple po-

sitions within the hypervisor source code. Therefore, every time a function is called or an event is triggered, the ID of the function or event is recorded in a log file. The information within the log file will be used as an input to a particular machine learning procedure to classify the status of the Virtual Machine.

We believe that the log data of the called tracepoints during the VM operation contains rich semantic information and therefore is a good alternative for observation data into a VM.

4 Qemu Layer Introspection for Anomaly Detection System

In the Qemu-KVM hypervisor combination, the Qemu part works in the userspace of the host system. Consequently, the Qemu is the closest component of the hypervisor to the guest VM and therefore, can provide better semantic data on the guest VM operation. Moreover, instrumenting a userspace application will not require us to make any modification to the host operating system. This instrumentation method, in turn, will simplify the monitoring operation and do not add additional cost to the host OS operation. It is, therefore, a logic action to start this research at the Qemu layer.

In this chapter, we focus our work on monitoring guest VM to detect anomaly operation using the operational pattern of tracepoints from Qemu source code.

4.1 Data Collection

We perform the guest VM operation by statically instrumenting Qemu source code and then collecting the log data of Qemu runtime for analysis.

We performed the static instrumentation using *ust* (userspace tracer) backend from the LTTng userspace tracing (LTTng-UST) library *. We used the *trace-events* file from the Qemu standard installation as our list of monitored tracepoints. The list contain around 1200 tracepoints inside the Qemu. We collected multiple observations for every scenario we used in the evaluation. We defined an observation as a single data unit which contains a collection of all the executed tracepoints in a VM in one second. An example of a monitoring data is given in Figure 3.2. In the figure, each line is one data unit (one observation).

*http://lttng.org

The interval between each data unit is one second. Each file constitute a scenario that compiles time-based sequence of an average 1000 data units.

4.2 Analysis of Qemu Introspection Data

The data that is collected in our monitoring method is in the form of big text data. This data would be hard to be processed manually by human operators. Therefore, we chose to implement Machine Learning approach to process the monitoring data. Within this work, we study three type of Machine Learning approach. First, we used the Bag of Tracepoint approach, where we use the pattern of each tracepoint quantities. Second, we used the Tracepoint sequence pattern. Since the core of QEMU is based on event-driven architecture, each process by the qemu will leave a unique sequence footprints. Each of both previous approach has its own advantage and drawbacks, so for our third attempt we used the combination of both tracepoint sequence and the pattern of their quantities.

4.2.1 Bag of Tracepoints Approach

We started with one common feature extraction, the *'bag of'* approach. For every observation dataset, we collect the occurrence frequency of each unique tracepoints. Therefore, the form our next dataset is a vector $D = |N| \times |M|$, where N is the set of observations, M is the set of unique tracepoints and $D_{n,m}$ is the frequency of tracepoint M_m at observation N_n. A unique tracepoint is represented only by its name.

Feature Reduction Process

Qemu comes with around 1200 default tracepoints. For a real-time anomaly detection, this amount of tracepoints is still very big to observe. Furthermore, from the machine learning point of view, not all of those tracepoints contain significant information. Some of the tracepoints can even act as noises. Therefore, we need to extract just small enough feature without compromising the final anomaly prediction quality.

Ideally, we can minimize the number of feature using a complete understanding on how internal hypervisor works, which is usually called the white-box approach. The clear picture of hypervisor inner works can enables us to identify what functions are related to the processes we want to observe. However, in this preliminary research, we do not assume any knowledge of hypervisor internal operation. As a solution, we adopt the black-box approach, where we compare each input to its output and tries to figure out their relation. In specific, we decided to use memory, I/O and network process signature as our basic profile. Many well-known and common malicious activities can be directly related to abnormal system performance [ATJ+10], for instance the Denial-of-Service, password dictionary attack and fuzz testing.

The purpose of our feature reduction is to select a subset of variables that can highly explain the change in memory usage, I/O read-write frequency or network send-receive volume. Hence, we need to use dimension reduction technique that considers class information. For that reason, we use Linear Discriminant Analysis (LDA) technique. LDA seeks to reduce data dimensionality while preserving as much of the class discriminatory information as possible, such that maximizing between-class to within-class covariance. For two class analysis, linear discriminant is defined as the linear function $y = w^T x$ that maximizes criterion function

$$J(w) = \frac{|\widetilde{\mu}_1 - \widetilde{\mu}_2|^2}{\tilde{s}_1^2 + \tilde{s}_2^2}$$

where $\widetilde{\mu}_i$ is mean value for class i and $\tilde{s}_i^2$ is class i's scatter value along w projection. The output of LDA is linear combinations of its variables input.

We collected five datasets, each with 100 unit data. Those datasets represented the following scenarios: idle, stress-memory, stress-I/O, stress-disk and stress-network. We used "stress" Linux application for generating memory, I/O and disk data while for network we use ping with flood option. Next we paired the idle dataset with each of stress dataset which resulted in four pair scenarios. For each pair we applied Linear Discriminant Analysis to extract tracepoints that best separate each scenario. Using R [R C14, ODCM14], Table 4.1 gives the best tracepoint with its associative scenario and its LDA score.

Table 4.1: List of selected tracepoints with its associative scenario and LDA score

idle vs	TP name	LDA score
stress network	tap_send qemu_deliver_packet	0.9913
stress IO	bdrv_aio_flush	0.9869
stress memory	virtqueue_fill	0.5795
stress hd	memory_region_ops_write	0.9992

Anomaly Prediction Process

In our anomaly prediction process we implemented a semi-supervised anomaly detection system. This system introduces only one class model, which is the normal profile. Any data input that does not conform to this model will be considered as an anomaly. This approach is also known as one-class novelty detection system as all the learning data to create the prediction model come from one single class. We used this approach under the assumption that a VM in the cloud works for one specific service, for example a web server, an application server or a mail server. Since this kind of servers create almost homogenous operation, it is easier to provide a sound training dataset.

We used the One-Class Support Vector Machine (OCSVM) as our prediction engine. This predictor uses several advantageous properties of Support Vector Machine (SVM) where it is more robusts to noise and can easily work with high dimensional data while allowing smooth and flexible nonlinear mappings. Data points that cannot be separated in their original space dimension are mapped to another higher dimension feature space where there is a straight hyperplane that separates one class to another. When the resulted hyperplane is projected back into the original input dimension, it would form some non-linear curve.

Scenarios for Normal Class

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 [RTSS09]. Web server operations also allowed us to experiment with multiple normal workload profiles for our evaluation purpose. We used RUBiS application [CMZ02] 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.

Scenarios for Anomaly Class

There were already many kinds of attacks that have happened in the cloud. However, one of our primary concern in this research is semantic gap information for the non-intrusive VM monitoring. Therefore we tried to imitate multiple attack scenarios that can represent either the processes that extensively use computer resources and can be easily detected without semantic context or the processes that in contrary happened in higher layer which makes it very hard to detect. For that reason we utilized these four attacks:

1. Synchronous-packet flood attack. This scenario was emulated using "hping3" tool and targeting port 80.

2. Password brute force attack. Using "ncrack" tool we try to crack an http basic authentication procedure.

3. Slow HTTP attack. We emulated this scenario using "httpslowtest" tool with attack in the body option.

4. Port scan attack. We emulated this attack using "nmap" tool and executed vertical port-scan attack.

For each type of attacks above, we performed both 'source attack' where the attack originates from the monitored VM and 'target attack' where monitored VM is the intended target of the attack. It resulted in total of eight anomaly type. We collected 500 unit data for each anomaly types. Just like normal data collection, one unit data was the list of variable occurrence within two-seconds of observation. We did the same amount of collection for the three data sources

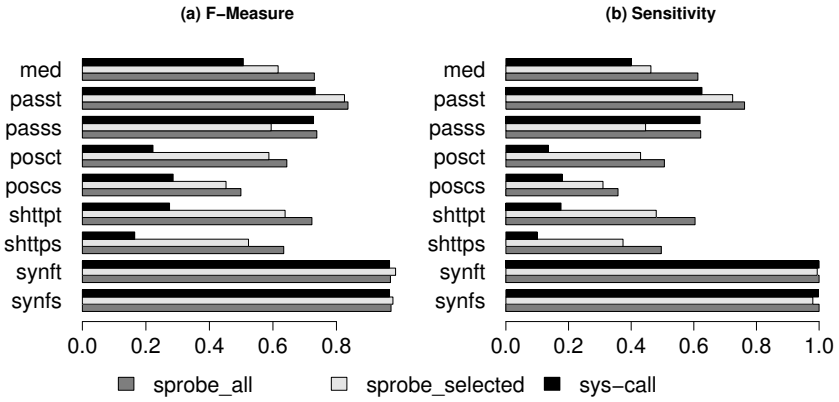


Figure 4.1: Prediction results for three monitoring approaches and eight anomaly scenarios

(static probe instrumentation using all feature, static probe instrumentation using selected feature and system-call) we want to compare. In total we evaluated 24 scenarios.

Evaluation of Detection Effectiveness

We emulated several attack scenarios as anomaly data and then observed if the predictor was able to identify them as anomaly or not.

To measure effectiveness, we used F-measure metric. F-measure defined as the harmonic-mean of sensitivity and precision.

$$Sensitivity = \frac{TP}{P}; Precision = \frac{TP}{TP + FP}; Fmeasure = \frac{2}{\frac{1}{Precision} + \frac{1}{Sensitivity}}$$

We divided 500 unit data for each scenario into five subsets. We then paired each subset of normal and anomaly, and feed them into one-class SVM outlier predictor. Over the five subsets, we calculated the averages of true-positives (TP), false-positives (FP), true-negatives (TN) and false-negatives (FN). Using these values, we calculated f-measure value of each scenario. We summarize the results of this evaluation in Figure 4.1.a.

From Figure 4.1.a. we note that, aside from synflood scenario, our static-probe instrumentation approach and system-call approach still give poor predic-

tion values, which are below 90%. The median of all scenario f-measure value for static-probe using all tracepoints, static-probe using selected tracepoints and system-call monitoring is 73%, 62% and 50% respectively. It is also useful to look in detail to sensitivity value. This metric measure how accurate the system is in detecting real positive data (anomaly data) only. The sensitivity results of this evaluation are given in Figure 4.1.b. The median of all scenario sensitivity value for static-probe using all tracepoints, static-probe using selected tracepoints and system-call monitoring is 61%, 46% and 40% respectively. Syn-flood scenario can be easily recognized because it directly affects the volume of send and receive data. However, since slowhttp scenario and portscan scenario do not change transfer rate, prediction is proved more difficult. Due to the fact that our approach works only by monitoring frequency of function-call, it can only detect changes in volume. It cannot however detect changes in sequence pattern for example, which might help increase accuracy. This Figure 4.1.b also shows, that in all of scenario test, system-call prediction value was lower than static probe instrumentation prediction.

4.2.2 Sequence Analysis Approach

In terms of decision making, the bag of tracepoint features is similar to the more conventional computation usage features. The anomaly decisions are being given only by how 'noisy' the monitoring objects are. The lack of semantic information might leads to a high number of false alarms. Therefore we believe that we can increase the detection quality if we put some effort to extract more semantic information. One of the method to extract this information is by analysing sequence.

The logical architecture of this sequence analysis is given in Figure 4.2. Following the approach of bag of tracepoints above, we use the semi-supervised one class anomaly detection method. We used only normal class dataset for training phase. First we randomly create a subset of normal class dataset using Hiddem Markov Model (HMM) fitting method to create the model of normal class. Then, we used the normal class HMM model and another random subset of normal class dataset to create a Likelihood Score (LLS) of the normal class. We used the LLS of the normal class as a reference in the evaluation phase or in the production

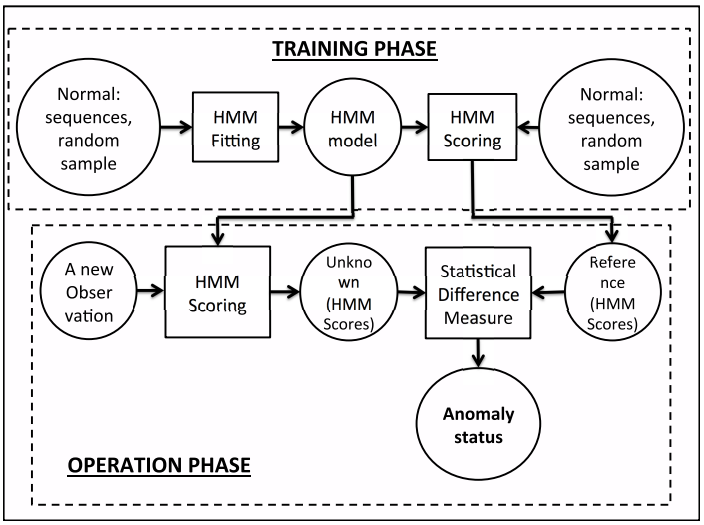


Figure 4.2: Logical framework of the proposed Sequence-based Anomaly Detection System

phase to decide if the new incoming monitoring data units are coming from a normal or from an anomaly operation in the guest VM. Detail explanation of this framework are given in the following sections.

Extracting the Sequences

One unit of monitoring data is the collection of caught tracepoints within a fixed pre-defined period. However, even though the collection time for each data unit is the same through out the monitoring process, the number of captured tracepoints for each data unit might be different. On the other hand, the sequence inputs for the modelling function had to have equal lengths. To accommodate this requirement, we convert each observation data unit into lists of smaller sequence of tracepoints using the sliding window approach. It is common practice to extract smaller sequences out of longer test sequences and stated the outlier score of the whole sequences as combination of its smaller sequences [Agg13]. We denote the observation O by a series of elements $\{p_i\}_{i=1}^{m}$ where p_i element $[n]$ and m is the number of trace-points within an observation. A sequence $S(x, y) = \{p_i, i\epsilon[x, y]\}$ is the set of all observation elements between p_x and p_y, inclusively. A sliding window is defined as $W = \{S(t, t + N - 1)|0 \leq t \leq m - N + 1\}$, where N is a

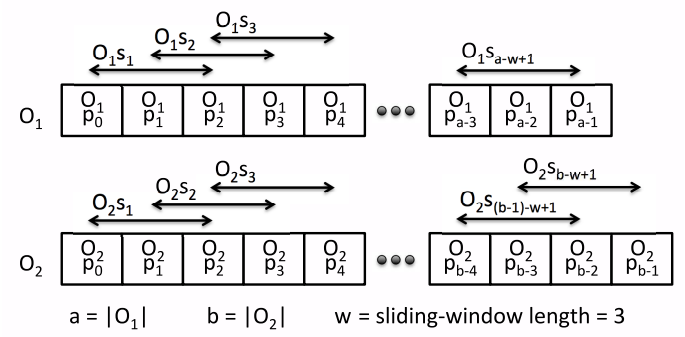


Figure 4.3: An illustration of sliding window method

predefined parameter of the sliding window's length.

An illustration of this partition method is given in Figure 4.3. This example shows two observations, O_1 and O_2 with different length, a and b. The sliding window's length (w) is 3. The number of sequences in O_1 is $a - w + 1 = a - 2$ sequences, while the number of sequences in O_2 is $b - w + 1 = b - 2$ sequences.

Modelling the Sequences: Hidden Markov Model Method

In the training phase, we modeled our training dataset using the Hidden Markov Model (HMM) method. A HMM is a statistical model where the system being modeled is assumed to be a Markov process with unobserved, hidden states. This HMM model consists of a list of hidden state transition probabilities A, a list of observed variable emission probabilities B and a list of starting probabilities of each state π that most likely create the training data. In a formal form, given the observation sequence O, we try to adjust the model parameters $\lambda = (A, B, \pi)$ to maximize $P(O|\lambda)$, where:

- A is a hidden state transition matrix with size $N \cdot N$, row stochastic (the sum of each row is equal to 1). In this work we also specifically use ergodic probability chain (there is a positive probability to pass from any state to any other state in one step). N is the number of hidden states.

- B is the output emission probability matrix $N \cdot M$, and represents the chance of each observable variable to occur from each hidden state. This matrix is also row stochastic. M is the number of observable variables.

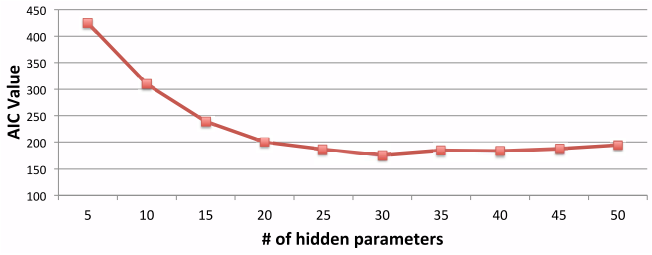


Figure 4.4: AIC value for HMM model created using 5 - 50 hidden states

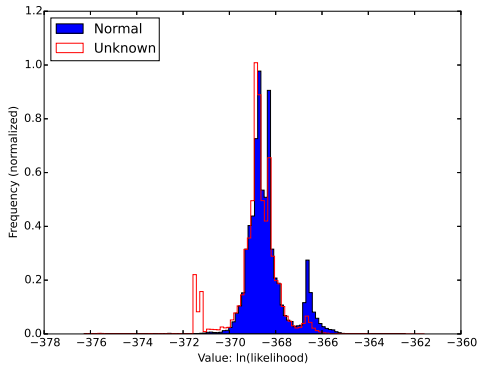
- π is the N-sized list of probabilites for each hidden state to become an initial state.

The main issue for the model creation process in HMM is deciding the number of hidden states. In many cases, this problem is not as straightforward as stating the number of observable variables, especially if there are no physical attributes that correlate to the hidden state parameters. Presenting a bigger number of hidden states will create a fitter model. On the other hand, that big number of hidden states will introduce an overfitting problem. There is still no definitive and recognized solution to the problem of choosing the optimal number selection for hidden states in HMM. Instead, researchers usually use some information-based techniques, such as AIC (Akaike Information Criterion), BIC (Bayesian Information Criterion) or LRT (Likelihood Ratio Test). These Information Criterions (IC) are a measure of the relative quality of statistical models for a given set of data and work by introducing a penalty for additional free parameters. Given a collection of models for the data, IC estimates the quality of each model, relative to each of the other models. Hence, IC can be used to provide a means for model selection. Within our work, we use Akaike Information Criterion (AIC) as suggested in [CdA10].

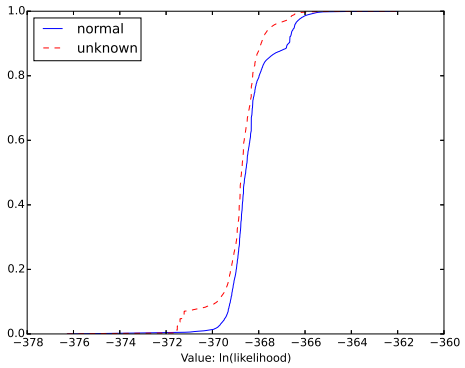
$$AIC = -2\ln(p) + 2k$$

where p is the highest likelihood value from the model and k is the number of free parameters. Varying the value of k as the number of hidden states, we select one preferred model by choosing the lowest AIC score.

Results of AIC value for $0 < k \leq 50, k\%5 = 0$ are given in Figure 4.4.



(a) Comparison in histogram format



(b) Comparison in CDF format

Figure 4.5: LLS comparison between a reference and an unknown observation

Deciding Anomaly: Comparing Data Distributions

In the HMM evaluation phase, the output of the HMM scoring function for an observation is a list of continuous values of likelihood score (LLS). Using the binning technique, we can visualize this LLS using a histogram. To decide whether a new observation is an anomaly or not, we compare its LLS with an LLS from a reference. This reference LLS is created from the normal scenario dataset during the ADS training phase. An example of reference's LLS and the LLS of a new observation is given in Figure 4.5a.

There are many known measures to compare two sets of continuous values. In general, these measures can be categorized into two main groups:

- *Comparing location.* Here comparison is based on the individual values within the list. Hence, the term 'location' here refers to its histogram position along the X-axis. An example of this is the comparison of means, medians, or tails.

- *Comparing shape.* In this method, instead of comparing individual values, we compare the relation between values. This relationship can be represented by its probability density function (PDF), so the shape here refers to the shape of its PDF. Some examples of this method are the Jansen-Shannon Distance and the Bhattacharya Distance.

In this work, we will compare three types of comparisons, namely the Mean Comparison, the Jensen-Shannon Distance (JSD) and the 2-Sample Kolmogorov-Smirnov Test (2S-KST).

1. *Mean Comparison.* Here we compare the absolute difference between the LLS average of an observation with the LLS average of the reference.

$$M(P,Q) = |mean(P) - mean(Q)|$$

   This method will represent methods that compare LLS locations.

2. *Jensen-Shannon Distance (JSD)* [OV03]. JSD is a metric commonly used to measure the difference between two probabilty distributions. It uses the Kulback-Leibler divergence formula that measures information difference, gain or loss, when mapping one distribution function to another. JSD is measured by:

$$JSd(P,Q) = \sqrt[2]{\frac{1}{2}(D(P||R) + D(Q||R))}$$

   where $R = \frac{1}{2}(P+Q)$ is the mid-point measure and $D(\cdot||\cdot)$ is the Kullback-Leibler divergence. This method will represent methods that compare LLS shape.

3. *Two Sample Kolmogorov-Smirnov Test (2S-KST).* 2S-KST quantifies the distance between two empirical distributions using their Cumulative Distribution Function (CDF). It seeks the maximum vertical distance between both CDFs. For two given CDFs $F_1(x)$ and $F_2(x)$, the 2S-KST statistic is given by

$$D(F_1, F_2) = sup|F_1(x) - F_2(x)|$$

   where *sup* is a supremum function. An example of two compared CDFs is given in Figure 4.5b. It is one of the most useful methods to compare two samples as it takes into account both their location and shape.

Evaluation Metric: Receiver Operating Characteristic (ROC) Curve

In most cases of binary classification systems such as ADS, the probability distribution area of both classes are overlapped. In such cases, the quality of an ADS

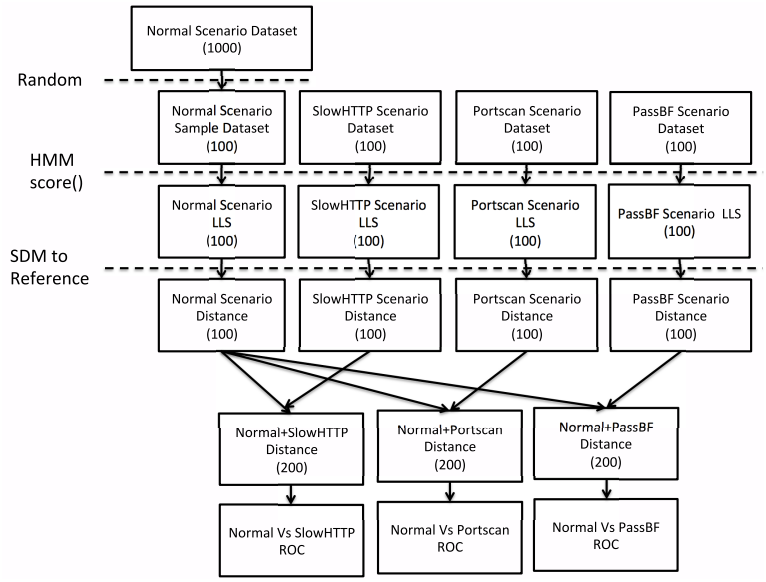
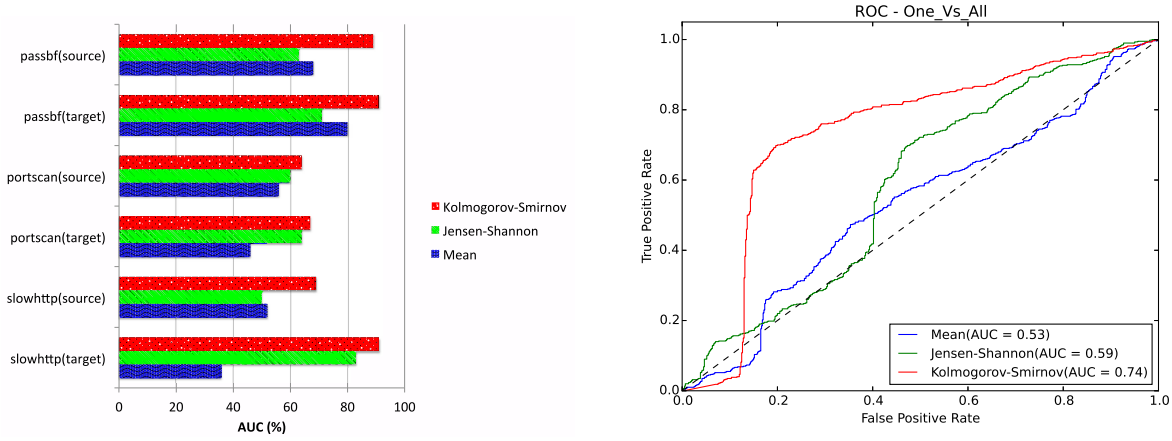


Figure 4.6: Steps taken to create ROC/AUC

predictor not only depends on an instance measure of sensitivity or accuracy, but also depends on what constitutes a normal or anomalous instance in the dataset. As the decision of classification is decided using a threshold value, it is preferrable to see all possible outputs to be able to select optimal models and discard sub-optimal ones. An ROC curve visualizes how well a system separated both classes by evaluating all possible thresholds. It depicts relative trade-offs between true positive (benefits) and false positive (costs). To quantify the system's effectiveness, we measure the size of Area Under the Curve (AUC), where $0 \leq AUC \leq 1$. An AUC value that close to 1.0 indicates a good model to distinguish classes.

Within this work we used ROC to evaluate models. For each ROC graph we randomly selected 100 out of 1000 observations of normal scenarios and 100 observations from the dataset of each anomalous scenario. Next we compute List of Likelihood Score (LLS) for every observation. Each LLS consists of the HMM scores of all sub-sequences (sliding windows) within the observation. The LLS is then converted into a distance value by calculating its separation to a certain reference LLS. Finally, the list of distance values of each test scenario are paired with the list of distance values of the normal scenario to create ROCs and their respective AUC value. Graphical representation of of ROC's calculation processes



(a) AUC comparison of individual scenarios

(b) Combining all scenario into one data

Figure 4.7: Comparison results of three statistical distance measures.

are given in figure 4.6.

Comparison Between Statistical Distance Measure Methods

In our framework, SDM is used to measure the similarity between our reference of normal data and a new observation data. We chose three commonly used SDMs, which are the Mean Comparison (MC), the Jensen-Shannon Distance (JSD) and the 2-Sample Kolmogorov-Smirnov Test (2S-KST) and compared ADS prediction results using each of the SDMs. The results are represented by the AUC value given in figure 4.7. From this bar-chart, we can see that 2S-KST consistently outperformed the other two methods, MC and JSD, by giving better separation between LLS from normal dataset and LLS from the anomalous dataset.

We believe the explanation of this result lies in the relation between the characteristic of the data and how each method computes the distance. First, Kolmogorov-Smirnov (KS) is a statistical approach that converts both the to-be-compared Probability Density Functions (PDF) A and B into Cumulative Distribution Functions (CDF) and finds a point x where $abs(CDF_A(x) - CDF_B(x))$ is the biggest. Therefore KS is a 'local-maxima', which means that it's value is decided by one specific value x. Kullback-Leibler family (KL) on the other hand is an entropy-based approach. In general, it uses the sum of logarithmic differ-

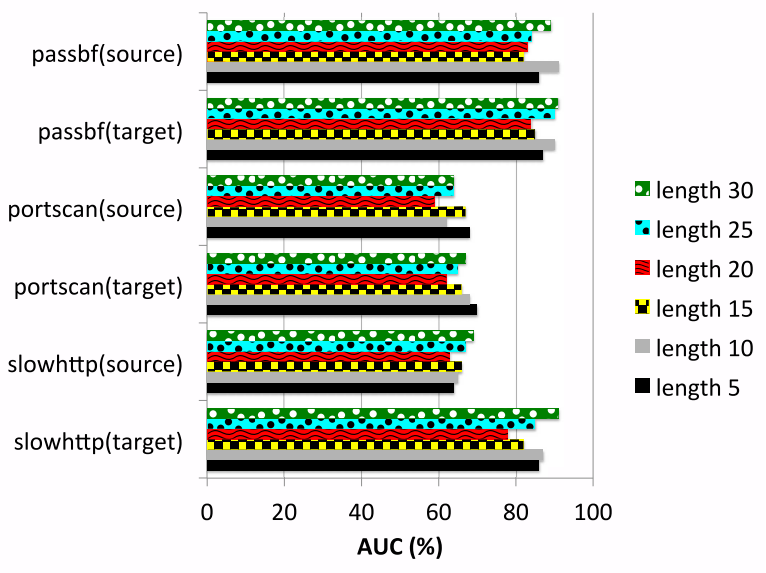


Figure 4.8: AUC results for different sequence lengths.

ence between two PDFs. Unlike KS test, the KL divergence is 'non-local', which means that its value is the result of processing all data in the problem domain. Intuitively, the local-maxima distance (KS) will give a bigger distinction value between two PDFs compared to the non-local distance (KL). To put this in the context of an ADS, KS will give a better True Positive Rate value compared to KL. In retrospection, KS is more susceptible to False Positive than KL. Secondly, on the data part, since we use only one specific normal scenario data in the experiment, the pattern of normal scenario is nearly homogenous. In other words, we did not have much of a problem to find the True Negatives. The real problem is to detect the True Positives. We assume that both the distance computation method and data characteristic are the reason why KS works better than KL in our case. Furthermore, if normal data scenario is more dispersed, KS will give lesser performance. However, as we stated in the Discussion section, in the HMM approach, multi normal scenarios will each require a different model.

Comparison Between Length of Sequence Windows

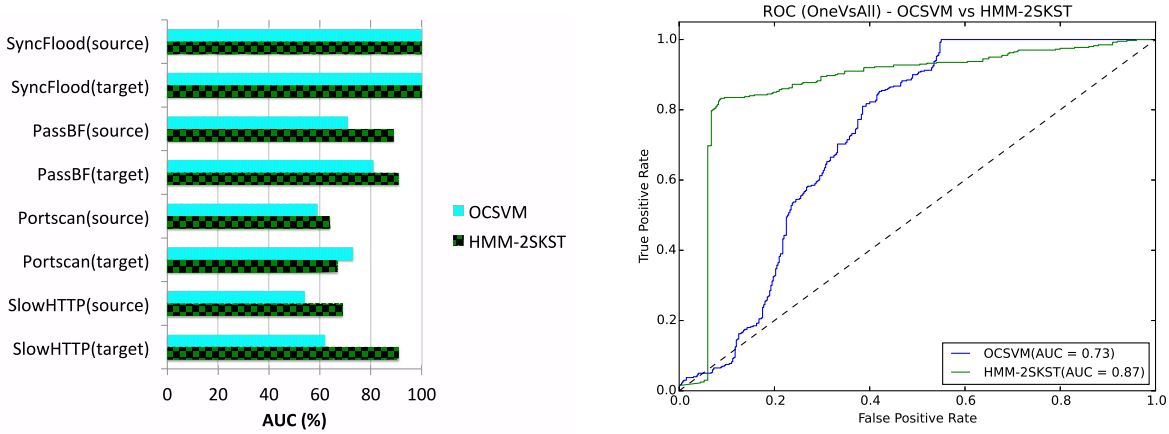
We wanted to see any relation between chosen sequence length and prediction result for each anomaly scenario. We compared various HMM models based

Table 4.2: Sequence patterns with support of 95% in normal scenario dataset

length	pattern	support
14	tap_send - qemu_send_packet_async_with_flags - qemu_deliver_packet - virtio_net_receive - tap_send - qemu_send_packet_async_with_flags - qemu_deliver_packet - virtio_net_receive - virtqueue_pop - virtqueue_fill - virtqueue_flush - virtio_notify - memory_region_ops_write - kvm_vm_ioctl	95%
43	cpu_set_apic_base - kvm_run_exit - memory_region_ops_read - kvm_vcpu_ioctl - cpu_set_apic_base - kvm_run_exit - memory_region_ops_read - kvm_vcpu_ioctl - cpu_set_apic_base - kvm_run_exit - memory_region_ops_read - kvm_vcpu_ioctl - cpu_set_apic_base - kvm_run_exit - memory_region_ops_read - kvm_vm_ioctl - apic_report_irq_delivered - kvm_vm_ioctl - apic_report_irq_delivered - memory_region_ops_read - kvm_vcpu_ioctl - cpu_set_apic_base - kvm_run_exit - memory_region_ops_read - kvm_vcpu_ioctl - cpu_set_apic_base - kvm_run_exit - memory_region_ops_read - kvm_vcpu_ioctl - cpu_set_apic_base - kvm_run_exit - kvm_vm_ioctl - apic_report_irq_delivered - memory_region_ops_read - kvm_vcpu_ioctl - cpu_set_apic_base - kvm_run_exit - memory_region_ops_read - kvm_vcpu_ioctl - cpu_set_apic_base - kvm_run_exit - memory_region_ops_write - kvm_vcpu_ioctl	95.1%

on sequence length for each anomalous scenario we prepare. We used a sliding window of 5, 10, 15, 20, 25 and 30. Working with a longer sequence is not feasible within our computing environment. All the other variables were set constant, where the number of hidden states was 30, the modelling iteration was 10 and number of training sequences was 20000. In this evaluation we used 2S-KST to check observations similarity. The results are given in figure 4.8.

The pattern for sequence length results look random. However, we can see that it formed two maxima at either side of length 20. To try to explain these results, we applied a data mining approach from [FVWGT14] to find the longest pattern



(a) AUC comparison of individual scenarios

(b) Combining all scenario into one data

Figure 4.9: Detection results of OC-SVM (frequency-based) and HMM (sequence-based).

to appear in normal dataset scenario. We looked within a 95% support value (which means the pattern should at least appear in 95% of the observations). The results were two sequence patterns with the length of 14 and 43. The details of the sequence's pattern are given in table 4.2. This finding in our normal dataset might explain why the AUC results for several sequences length between 5 to 30 formed a graph with two peaks. Between these two sequence length values, 5 is preferable as it requires lower computation resources.

Comparison to the Bag of Tracepoints Approach

Static probe instrumentation data inside hypervisor is mostly used for hypervisor debugging purposes. To the best of our knowledge, [POKY16a] was the first attempt to apply static instrumentation data for host-based anomaly detection purposes. The authors attempted to use occurrence frequency of trace-points (well known as the Bag of Trace-Point approach) as the basis for anomaly detection. Their results showed that their approach can only recognize attacks that change computation volume (volume-attack), such as sync-packet flood attacks. However, when dealing with non-volume-based attacks, their system still worked poorly.

As already described earlier, this work utilized sequence-based analysis using HMM. We argued that this approach can give better prediction results, even for some non-based volume attack. The basic intuitions of our arguments are:

- In our experiment we use static probe instrumentation data, which are related to function call information. By the nature of this data, we believe that the relation between each data-point (the trace-points) is best described by their sequence (the chain of function call). As a comparison, in an NIDS experiment that uses, for example, ip-address or port number (source/destination) as data-points, the relation between each data-points would be better described by occurrence frequency.

- A change in data frequency will always (to some extent) change data sequence, and therefore can be detected by both frequency and sequence analysis. On the other hand, a change in data sequence does not always change frequency, and therefore can only be detected by sequence analysis.

- In the occurrence frequency analysis approach, each data point is composed of multiple discrete data (frequency of each trace-point). In the sequence analysis approach, each data point is composed of multiple continuous data (likelihood of each sub-sequence). Therefore, inherently one data-point in sequence analysis approach contains more information than one data-point in occurrence frequency analysis approach. The compensation lies in the computation requirement to collect and process the data-point. The sequence-based approach requires much heavier computation compared to the occurrence frequency approach.

To have empirical proof, we perform an experiment to compare detection quality of both frequency-based analysis and sequence-based analysis. We emulated the same set of attack dataset, which are sync-flood, port-scan, password brute force and slow-http attacks, both as attack source and target. We then compared the prediction results against each individual test attack scenario (figure 4.9a) and prediction results against the whole test attack scenario (figure 4.9b).

Figure 4.9b shows that analyzing sequences of trace-points give 20% better prediction results than frequency-based analysis. In figure 4.9a, aside from minor

Table 4.3: Specific sequence we use to explain performance of HMM framework in Section 4.3.

op.	pattern
receive	qemu_send_packet_async_with_flags - qemu_deliver_packet - virtio_net_receive
send	qemu_sendv_packet_async - tap_receive_iov - tap_write_packet

lesser value in portscan (target) attack and equal results for sync flood attack, all other results show improved performance, with a notable improvement for slowhttp (target) attack. However, portscan attack still cannot be detected properly. The AUC values around 60 for portscan attack means that prediction results are still random.

To explain these results, we ran one minor experiment. We focused on the network operation for each of the scenarios to find their characteristics. We looked for the sequences list in table 4.3 for send and receive operations. We counted the average occurrence of these sequences in an observation, as we believe this feature is related to HMM's score Probability Density Function. For simplicity, we just used anomalous scenario for attacked VM and the results are given in table 4.4.

The ratio row in the table represents the proportion between the number of receive-sequence to send-sequence. We can use this ratio to guess the probability of two scenarios having the same sequence pattern. Two scenarios having a very different ratio value will have a higher chance of having different combination between their items. In contrast, two scenarios having almost similar ratio values will have a higher chance of having similar combination between their items. The difference in sequence patterns will lead to a difference in PDF and makes them easy to distinguished each other. The same explanation applies for all other scenarios. For example, to differentiate SyncFlood scenario from normal scenario we can use the ratio of network sequences over non-network sequences. The ratio between network-receive operations and read-block operations for normal, portscan, password-bf and syncflood scenarios are 686, 1074, 2853 and 80965

Table 4.4: Occurrence number of specific send and receive sequence in different scenarios.

	Normal	SlowHTTP	Portscan	Pass.BF	SyncFlood
Send	394.641	42.56	1198.46	4297.93	28138.42
Receive	318.277	234.26	1292.46	2711.18	31576.67
Ratio	0.806	5.504	1.078	0.631	1.122

respectively. Portscan scenario was hard to differentiate from the normal scenario because the frequency ratio among any different operation from both datasets was almost similar.

On the other hand, One-Class SVM only works by using trace-point frequency to create a proximity model in a high dimensional space. This is however only effective for anomalous scenarios that drastically change certain resource utilization.

There were many other anomaly detection systems previously introduced. However, direct comparison is difficult to do as observation data and evaluation scenario were different. For example, Sha et al. [SZCH15] applied Multi-order Markov chain to detect anomalies in cloud server systems. For evaluation purposes, they used system-call information from DARPA's 1998 Intrusion Detection Evaluation dataset. In their paper however, there was no specific information on what the quantified results on their anomaly detection scheme were. Alarifi et al. [AW13] implemented HMM for the same purpose with this paper. They collected system-call in the host system to monitor guest VM. For anomaly scenario, they used a non-malicious stress test in guest VM to emulate what they called an "over-committed migration" attack, and reported a 100% detection rate with a 5.66% false positive rate. Their malicious data type was similar to our synchronous packet flood attack which are both volume-based attacks. However, we do not know how their system fares with any non-volume-based attack.

4.2.3 Bag of Sequence Approach

Each of both previous approach, the Bag of Tracepoints and the Sequence of Tracepoints has its own advantage and drawbacks. In short, the Sequence of

	Feature 1: Unseen sequences	Feature 2: Sum of all 'weak' sequences	Feature 3: Unique Sequence 1	Feature 4: Unique Sequence 2	...	Feature n: Unique Sequence n-2
Observation 1					...	
Observation 2					...	
Observation 3					...	

Figure 4.10: Bag of Sequence data format

Tracepoints (SoT) gives better detection results compared to the Bag of Tracepoints (BoT) approach. However, the SoT analysis are more expensive (in term of computation resources and computation time needed) compared to the BoT. Therefore, for our third attempt we used the combination of both tracepoint sequence and the pattern of its quantities. Our intention is to acquire the accuracy of SoT with the cost of BoT.

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

Feature Extraction

Just like the previous two approaches, we extract the tracepoint name from the monitoring data. Then we split the tracepoint sequence of an observation into multiple sequence windows with equal length (see Section 4.2.2). Then we listed every unique sequence and count their appearance frequency. This data will be used for our chosen machine learning algorithm.

Feature Reduction

As we will use One Class Classification technique for our ADS, we need to keep all the features (e.g. all sequences) in our datasets. Feature reduction will decrease the ADS ability to recognize any new sequence patterns. Yet, a sliding window with a length of w and p unique tracepoints will create a possibility to have up to p^w unique sequences, which can create a very big dimension of data. To minimize the problem dimension, instead of removing the 'weak' features from the learning class, we opt to merge all the weak features into one single feature. The weak features are all the sequences that have lower appearance frequency than a threshold t in the training dataset. Lowering the t will increase the data

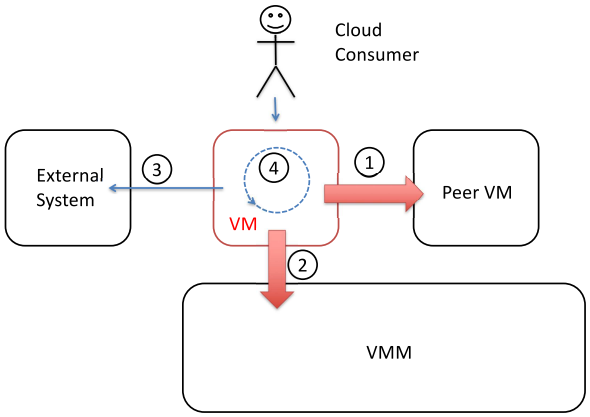
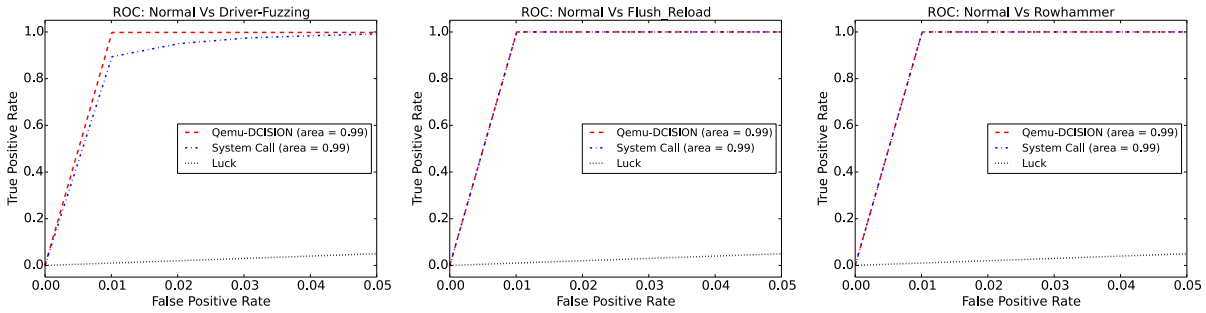


Figure 4.11: Attack vectors of our threat model.

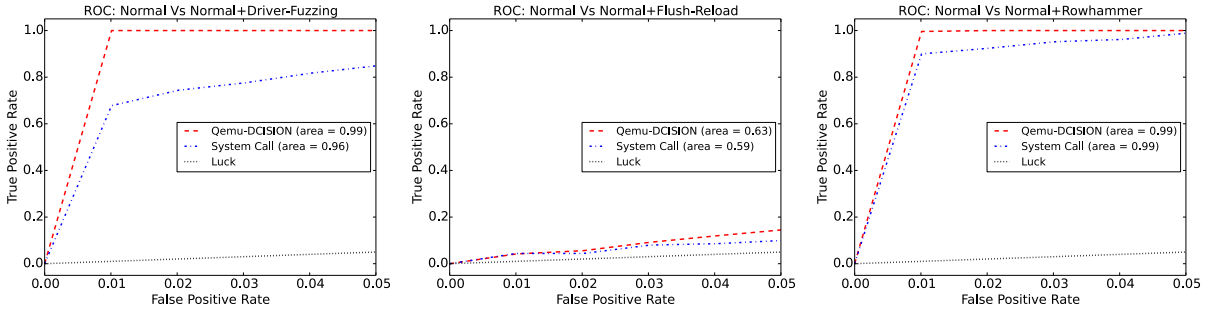
dimension while a higher t will decrease the data dimension.

The advantage of this approach is two folds. First, we will have a more compact datasets without removing any features. This can leads to a better computation performance with a minimum cost of information lost. Second, it will give more weights to the sequences that appear more regularly in the normal scenario dataset and hence will increase the discrimination factor between the normal scenario dataset and the other class datasets.

The format for our dataset is given in Figure 4.10. The first attribute is reserved for all the sequences that were never appeared in the training dataset. Therefore, in the normal class dataset, this column values should be an integer close or equal to zero. The second attribute recorded the sum of appearance number of all the 'weak' sequences. Finally, all the remaining attributes represent each individual 'strong' sequence's appearance frequency. As opposed to the weak sequences, the strong sequences are those that have higher occurrence frequency than the threshold t in the training dataset. By using this format, we can use the first column to observe any change in the sequence pattern while the other columns are used to observe the change of intensity.



(a) without noise



(b) with noise

Figure 4.12: The ROC graph for Qemu dataset and system-call dataset.

Threat Model

We slightly changed the threat model of this evaluation. Instead of using network attack scenario, we evaluate this bag of sequence approach using several cloud environment based attacks. We are not considering the attacks from guest VM to other systems outside the host (Figure 4.11 point 3), or vice-versa. Those attacks will require a certain network connection and thus can be handled by the vast array of contemporary Network Intrusion Detection Systems (NIDS). We also exclude the scenarios where a certain guest VM user wants to attack his/her own VM (Figure 4.11 point 4). We believe that this type of attacks should be handled internally inside the VM using Host Intrusion Detection System (HIDS).

Evaluation of Detection Quality

We evaluated two environment scenarios, noiseless and noisy environment.

Table 4.5: The statistical comparison between the regular workload dataset and each real attack scenario dataset.

		(a)	(b)	(c)	Additional note
Noiseless scenario	Flush-Reload	0.3	0.4	0.43	in Flush-Reload strong sequences: all values are very small and almost all of them are zero
	Driver- Fuzzing	12.6	3.5	0.28	
	Rowhammer	971.8	267.9	0.43	in Rowhammer strong sequences: all values are very small and almost all of them are zero
Noisy scenario	Flush-Reload	0.9	1.2	0.02	
	Driver-Fuzzing	39.1	28.3	0.53	
	Rowhammer	1431.5	214.5	0.19	

note: (a) Average of the previously unseen sequences; (b) Ratio of the weak sequences; (c) Difference of strong feature vectors

Noiseless Environment Results of the Qemu Bag of Tracepoint Sequence based ADS against real attack are given in Figure 4.12. In the scenario where the attack is the sole main workload within the VM (noiseless scenario, Figure 4.12.a.), all three attack scenarios, the rowhammer, flush-reload side-channel and driver fuzzing attack have the AUROC value of 0.99. That means that the anomalies can be fully distinguished and identified as anomalies.

The flush-reload attack result is easy to explain. Since it only affects the CPU cache, its operations cannot be detected by this Qemu Bag of Tracepoint Sequence based ADS. Therefore, its intensities are roughly counted as zero. A near zero change of intensity is easily distinguished from the intensities of regular workload scenario. However, to explain the other two attacks, we have to dig into the raw data.

Table 4.5 shows some statistics of comparison between each anomaly dataset and the regular workload dataset for Qemu Bag of Tracepoint Sequence analysis.

- *The Average of unseen sequence* column lists the average of the first column values in the anomaly dataset. It gives the average number of sequences from the anomaly scenario observations that never appear in the regular workload scenario. The average value for this column in the normal dataset is 0.

- *The ratio of the weak sequences* column values come from the average of the second column values of the anomaly dataset divided by the average of the second column values of the normal dataset. This will show how much the frequency of weak sequences changes in the anomaly dataset.

- The strong features are all the sequences that appeared frequently in the normal scenario dataset (see Section 4.2.3). The *difference of the strong feature vector* column is meant to quantify the difference between the intensities of the strong sequences (in respect of normal scenario dataset) in normal scenario dataset and the anomaly scenario dataset. The procedure to calculate this value is given in Equation (1).

Let us define normal dataset as ND and abnormal dataset as NA. Next we define $max_x = max(ND_x, NA_x)$ as a maximum value of feature x from both ND and NA. We can then calculate $diff$ as the difference value between ND and NA:

$$diff = \frac{\sum_{x=3}^{n} abs(\frac{\sum_{y=1}^{a} \frac{ND_{x,y}}{max_x}}{a} - \frac{\sum_{y=1}^{b} \frac{NA_{x,y}}{max_x}}{b})}{n} \tag{4.1}$$

where n is the number of features, a is the number of observations in ND, and b is the number of observations in NA. We remove the first two element of x because both features, the average of unseen sequence and the ratio of weak sequences, were evaluated separately.

Looking at the Rowhammer scenario raw data, we saw a high number of unseen sequences (an average of 972). The value of infrequent sequence (in the second column) of the Rowhammer dataset is also 270 times higher than the

regular workload dataset. The other important fact is in the other features of Rowhammer data, all values are very small, where most of them are zero. All of this gives strong indication that the Rowhammer scenario produces a different sequence pattern of tracepoints compared to the regular workload scenario. The Driver-Fuzzing scenario gives almost similar results as the Rowhammer scenario, but with a smaller intensity.

Noisy Environment In the noisy scenarios, the attack operations were executed while the normal web application was also running with regular workload. Our results showed that the detection quality did not decreased significantly, except for the noisy Flush-Reload attack (Figure 4.12.b.). In the noisy Flush-Reload attack, the detection score was significantly reduced where the quality dropped to the level of random detection (0.63 for Qemu tracepoint data and 0.59 for System-Call data). This can be explained by the fact that in monitoring guest VM, Qemu tracepoints data cannot be used to detect any changes in CPU operation. Therefore, comparing between regular scenario workload and the noisy flush-reload workload is like comparing two similar objects. As a hindsight on the Flush-Reload attack, the adversaries are unlikely to perform their attack in a noisy environment since it will also decrease the effectivity of the attack.

Comparison of Anomaly Detection Results: Software Instrumentation Data Vs System Call Data

Our results in Figure 4.12 shows that even though the Qemu Bag of Tracepoints Sequence approach consistently gave better results compared to the System-Call data, the difference is not significant. Therefore, in terms of the detection of Rowhammer, Flush-Reload and Driver-Fuzzing attacks, we can conclude that Qemu ttracepoint data and System-Call data gives similar results.

4.2.4 Computation Cost of the Monitoring Process

Cost to the Host: Efficiency and Scalability

We used the Linux's *perf* tool to measure CPU usage and the time needed to capture one unit data for certain number of VMs. For scalability evaluation

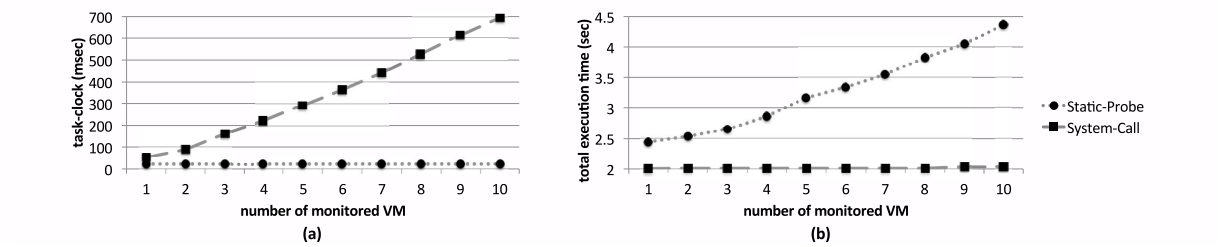


Figure 4.13: Performance value when capturing one unit data as the number of monitored VM was increased.

purpose, we used one until ten VMs. All multiple data collections were executed concurrently. The average results of twenty measurements for Qemu tracepoints and system-calls data collection are given in Figure 4.13.

Figure 4.13.a shows that the tasks of collecting system-call data required longer CPU time than the tasks to collect Qemu tracepoints data. As the number of monitored VMs were increased, the additional rate of task clocks consumed by system-call data collection was also became higher. Meanwhile, the required task clocks for Qemu tracepoints data collection in all VM number scenarios were almost remaining constant.

On the other hand, the results of measuring total execution time show that Qemu tracepoints data collection took longer to finish compared to the system-call data collection. As the number of monitored VMs were increased, the additional time required for collecting Qemu tracepoints data was increased at a much higher rate than system-call data collection. We believe that this is due to the different way each tool, the *LTTng* for Qemu tracepoints data collection and the *strace* for system-call data collection, process the collected data and writes the results to the disk.

Impact to the Guest VM

We compared the CPU and database performance of a guest VM when it is being monitored from the host. For the measurement process, we used the *sysbench* tool. In the CPU benchmark (Figure 4.14.a.), we recorded the total execution time of one thread to calculate the first 10 million prime numbers. In the database benchmark (Figure 4.14.b.), we counted how many database transactions can be

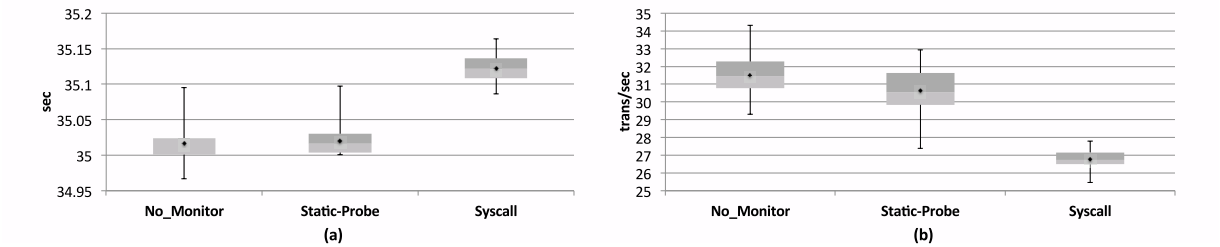


Figure 4.14: The impact of VM monitoring to the guest VM's CPU performance and database operation.

processed every second. Again, in both of these evaluations, we compared the performances of Qemu tracepoints data collection and system-call data collection. Figure 4.14 presents the averages from twenty benchmarking results.

Both boxplots in Figure 4.14 show that the system-call data collection process gave the bigger negative impact to the monitored VM than the Qemu tracepoints data collection process. In the CPU benchmark, system-call data collection process increased the benchmark time of no-monitoring scenario by 0.3%, while Qemu tracepoints data collection process only increased it by 0.007%. Similarly, in the database benchmark, the number of database transaction per second in the guest VM were decreased more by the system-call data collection process compared with the Qemu tracepoints data collection process. system-call data collection process and Qemu tracepoints data collection process reduced the number of database transaction per second by 15% and 2.9% respectively. Both results show that system-call data collection decreases the guest VM performance, more than Qemu tracepoints data collection.

4.3 Previous Works on Anomaly Detection System in Virtualization Environment

The previous methods for monitoring guest Virtual Machine (VM) are basically can be categorized into three classes, the computation metric monitoring system, the system call monitoring system and the Virtual Machine Introspection (VMI) method.

Several previous research on computation metrics monitoring method for the

virtualization system are summarized in the first five rows of table 4.6. The main assumption of this approach is that a malicious activity will likely change a considerable amount of computing resources usage. However, the usage data do not contain any semantic information. Both benign and malicious processes may have similar computing usage pattern, and therefore hard to be distinguished from one to the other. Furthermore, monitoring computer metric data can only be effective if it is implemented inside the observed system because most of the resources for guest VM are pre-allocated and not allocated on an as-needed basis.

The second VM monitoring method uses the system-call data. In the context of virtualization operation, the process of system-call monitoring is not trivial. The hypervisor as an intermediate system between the guest OS and the host OS adds another layer of system-call simulation. Hence, the system-calls that are collected in the host are mostly not the direct representations of the internal guest OS system-calls and therefore generate low quality semantic data. We summarized several previous studies on system-call monitoring for the virtualization system in the last three rows of table 4.6.

All the works mentioned above evaluate their work using anomaly scheme that significantly change the pattern of computing resources usage, such as CPU, memory or I/O. These approaches are indeed useful for detecting volume based attacks. In reality however, many attacks on computer system do not change these resources data, hence are harder to detect. To detect those non-volume based attacks, higher semantic information is needed and for that researchers usually monitor internal operation of observed VMs. Our work were focused on trying to extract clearer semantic information without requiring to interrupt guest VM operation. That is why we choose to evaluate our framework using varied anomaly scenarios, from volume based attack scenarios such as Syncflood attack to non volume based attack such as Slow HTTP attack.

The latest and more sophisticated approach to monitor guest VM operations is Virtual Machine Introspection (VMI). It works by capturing a snapshot of the memory space used by the guest VM. With this snapshot, a monitor can reconstruct an exact same picture of the situation inside the guest VM. This technique was first introduced by Garfinkel et al. (2003) [GR03]. Several notable VMI frameworks are Virtuoso [DGLZ+11], VM Space Traveller (VMST) [FL13],

LibVMI [†] and Volatility[‡]. One minor limitation of the current VMI implementations is their dependency to a certain data from the guest OS. This data is needed to correctly reconstruct the raw memory snapshots into a useful information. Examples of such data are the debugging symbols information file for the Windows systems and memory offset information file for the Linux systems. This requirement is easy to satisfy in a private environment. However, in a certain arrangement such as public IaaS, the VMI approach could be hard to be implemented.

4.4 Summary and Outlook

We introduced Qemu tracepoints as a new data source for a nonintrusive monitoring process to the guest Virtual Machine (VM) operation. This monitoring process leverages static probe data within the Qemu Virtual Machine Monitor software. Next, we used the Qemu tracepoints data to investigate three Anomaly Detection Systems (ADS) for predicting the status of guest VM operations. Each of the ADS uses different Machine Learning approaches, which are: Bag of Tracepoints approach, Sequence of Tracepoints approach and Bag of Sequence approach. Finally, we successfully investigated the strength and weaknesses of our Qemu tracepoint monitoring method and each of the ADS approaches through several empirical evaluations. We evaluated the effectiveness of the ADS'es against multiple virtualization anomaly scenarios, includes host-based and network-based anomalies. We investigated its computational cost to the host and its impact on the guest VM's computational load. We also compared the results of those evaluations to the same ADS using system-call data.

Our observation at anomaly detection results shows that Qemu tracepoints data gives an excellent overall performance on detection anomaly events, be it the host-based or the network-based anomaly. In comparison with the system call-based monitoring data, our Qemu monitoring method gives better anomaly detection results, scaled better and gives a lesser impact to the guest VM operation.

Our Machine Learning comparison shows that analyzing the sequence of tra-

[†]http://libvmi.com
[‡]http://www.volatilityfoundation.org

cepoints gives better anomaly detection compared to simply count the number of each the tracepoints. However, the sequence analysis approach is more expensive in terms of computation time and resources needed. Our third approach, the bag of sequence pattern gives better overall results for both the anomaly detection process and the computation cost.

Despite the sound overall detection results, we found that not all anomaly scenario that we tested can be detected satisfyingly. The most notable case was detection against processor-based anomalies, such as the Flush-Reload cache-based side-channel attack scenario. Against such a scenario, our Qemu tracepoint-based ADS gave close to random results. This result shows that QEMU instrumentation works poorly against CPU-based anomaly. Further studies are needed to overcome this shortcoming.

As the attacks that come from internal tenant VMs are becoming more complex, Virtual Machine Monitor (VMM)-based IDS will be needed more than the traditional Host-based IDS and Network-based IDS. Furthermore, several studies have shown that VMM-based can even detect host-based and network-based attacks in the virtualization environment [HZZ+13, SSKK14, ACL15, LTXZ14]. Our study using this hypervisor source-code tracepoints can detect SlowDoS attack [POKY16b], a case that is considered difficult in the NIDS domain. Therefore we argue that VMM-based IDS will have an important part in the future of virtualization security. Moving forward, we want to study the possibility to implement a hybrid monitoring for Qemu and KVM. Our Qemu monitoring implementation only works within Qemu, hence it cannot collect the CPU information. By concurrently instrumenting KVM operation, we might have better information about guest VM operation.

Table 4.6: Summary of several studies that use computation usage metric and system-call data for ADS in cloud system

Author	Data type	Normal scenario	Anomaly scenario	Results
Dean et al. (2012) [DNG12]	computation usage metric	RUBiS, IBM system S, Hadoop	Memleak, CPUleakNethog	up to TPR: 97% FPR: 2%
Huang et al. (2013) [HZZ+13]	computation usage metric	web service	malicious port scan	up to TPR: 70%, FPR: .4%
Doelitzscher et al. (2013) [DKRC13]	computation usage metric	simulation data	simulation data	Detection error rate 0.01375%
Silvestre et al. (2014) [SSKK14]	computation usage metric	Mongo DB operation	packet loss, network latency, misuse of memory	98% accuracy for supervised, 75% accuracy for unsupervised
Vallis et al. (2014) [VHK14]	computation usage metric	Twitter production data	injected anomaly	Precision 100%, Recall and F-Measure 99%
Alarifi et al. (2012) [AW12]	system-call	ERP system	'over-committed migration' attack	up to TPR: 100% FPR: 0%
Sha et al. (2015) [SZCH15]	system-call	DARPA dataset	DDoS DARPA dataset	average one step transition probability 10^{-5}
Abed et al. (2015) [ACL15]	system-call	Java application	SQL Injection attack	up to TPR: 100% FPR: .58%

5 KVM Layer Introspection: Detecting Cache-based Side Channel Attack

When the Qemu works with KVM mode, its CPU operations are not simulated. The CPU operation are runs natively on top of physical processors with the help of the KVM kernel module. As a consequence, we cannot monitor the CPU operation from the Qemu instrumentation. We need to find other data sources to detect any CPU-based anomaly operation inside the guest VM.

In this chapter, we introduce a new class of virtualization-based attack called Cache-based Side-Channel attack (CSCa) and propose a method to distinguish it from the normal operation using KVM-based tracepoint data. One of the CSCA scenarios we were able to detect was the Flush+Flush, a new, more stealthy CSCA.

5.1 Cache-based Side-Channel Attack

One of the advantage that is offerred by virtualization technology over the traditional computation method is the Isolation property. 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 [GYCH16, Sze18, BGN17, eni17]. 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, co-residency and website access across multiple CPUs, CPU cores, even across VMs. To help protect security in cloud computing systems, CSCa detection methods have become important.

A side-channel attack try to gain information from a victim by eavesdropping through a non-conventional 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 [RTSS09], or the attacker can spy any key-press on his peer tenant applications [Hor16].

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 [Hu92, Ber05]. The first application of this cache-based timing attack was demonstrated by Osvik et al. in 2006 [OST06]. 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 cryptoanalysis process. The Flush+Reload attack was introduced by Yarom et al. in 2013 [**?**]. This method took advantage of a memory deduplication technique [SIYA11] 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 has been proven to work not only for cryptoanalysis purposes, but also can be used to spy on many other daily applications, such as a javascript browser [OKSK15], user interface [Hor16] and even a mobile application [LGS+16].

In particular in the virtualization environment, an attack on a co-resident VM was demonstrated by Ristenpart et al. in 2009 by recovering the keystrokes from

a co-resident VM in commercial clouds [RTSS09]. In 2012, Zhang et al. showed how to recover an El-Gamal decryption key from a co-resident VM [ZJRR12]. Later, the same authors presented ways to use CSCa to attack a peer VM within a Platform using a Service (PaaS) cloud model [ZJRR14]. Inci et al. in 2016 presented a cache attack to enable bulk key recovery in a commercial cloud [İGI+16].

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

5.1.1 Prime+Probe

```
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 3: 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 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 pseudo-code in Algorithm 3.

5.1.2 Flush+Reload

```
 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 4: 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) [AEW09] and Transparent Page Sharing (TPS) [VC09].

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 pre-defined 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 pseudo-code in Algorithm 4.

5.1.3 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 need 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.

5.1.4 Flush+Flush

The Flush+Flush method [GMWM16] 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 pseudo-code in Algorithm 5.

```
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 5: Flush+Flush

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 ratio of cache misses over cache references were 94%, 97%, 12% and 18% for Prime+Probe,

Flush+Reload, Flush+Flush and web application respectively.

5.1.5 Defence Methods

Many research studies have 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 [WL08, WL07, LL14, ZR13]. 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 [LGY+16,ZRZ16] or at the application level using sanity verification frameworks [IES16, DFK+13]. Other approaches prevented cache sharing by distributing the VMs to different partitions in the cache, either using hardware [DJL+12, LGY+16] or software [KPMR12, SSCZ11]. 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 [HACL13, MSR15, ZLB+12]. However, cloud providers might find all these approaches less attractive because they require significant modifications to the cloud infrastructure.

In 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). Chiapetta et al. used the HPC data and coupled it with a neural network method to detect CSCa in real time [CSY16]. Zhang et al. went further by implementing CSCa detection in a virtualization environment [ZZL16]. 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 [Pay16] and Herath et al. [HF15]. The latest development in CSCa introduced a new stealthier variant called Flush+Flush [GMWM16]. 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.

5.2 Threat Model

The focus of our effort is detecting the Cache-based Side-Channel attack. 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. Further threat model are similar to the threat model we describe in Section 4.2.3.

Moreover, 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.

5.3 KVM Introspection Data Collection

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 2.3). In our implementation, we introspected the KVM events that are instrumented by a standard Linux kernel tracing utility called *ftrace* [Ros14]. Ftrace was built directly into the Linux kernel, 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 tracepoints, we wanted to ensure ease of implementation and make

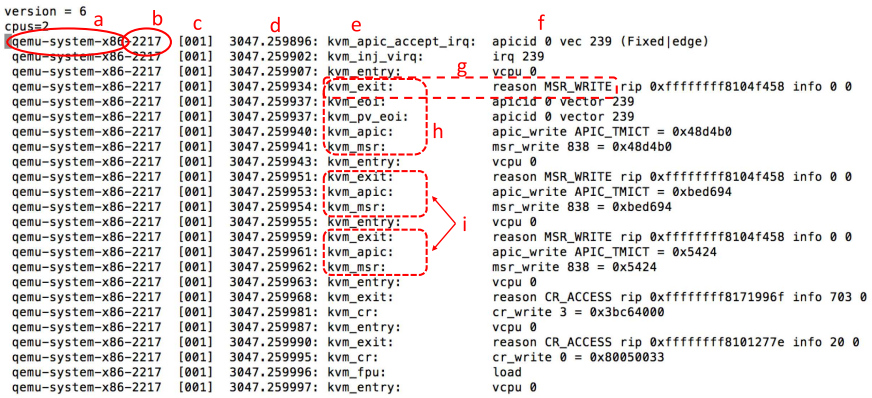


Figure 5.1: A snapshot example of trace-cmd output for KVM events.

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 5.1. It gives us the list of KVM events sequences that occurred during kernel mode operation (Section IV.A). The information gathered from this tool are the process name, process or thread ID, CPU ID, time information, KVM event name, KVM event information and the sequence of the events. Figure 5.1.g shows an example of one KVM_exit event and its exit reason. In this case we log the reason attribute. Figure 5.1.h shows an example of one KVM exit session that we used as one data (sequence) type. Figure 5.1.i shows an example of two sequences that belong to one sequence type. The pre-processing procedure to transform the text format into a vector input is explained in the next section.

Previous attempts and studies to monitor guest VM was described in Section 4.3.

5.4 Data Presentation

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 5.1 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 $X = \{Y_1, Y_2, Y_3, ..., Y_n\}$, where Y_i is the number of i-th KVM event sequence 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 5.1 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 5.1 gives four sequence IDs (note that sequences which are pointed to by (i) belong to the same ID)

- ID1: MSR_WRITE - kvm_eoi - kvm_pv_eoi - kvm_apic - kvm_msr
- ID2: MSR_WRITE - kvm_apic - kvm_msr

Table 5.1: 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

- ID3: CR_ACCESS - kvm_cr
- ID4: CR_ACCESS - kvm_cr - kvm_fpu

After having transformed all the sequences into ID's, we then counted how many times each ID showed up in an observation. Again, for illustration, having an input of Figure 5.1, 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.

5.5 Evaluation

We evaluate several aspects of Cache-based Side-Channel attack (CSCa) detection in this section, such as the effectiveness of trained scenario detection, the effectiveness of zero days scenario detection, the comparison of different microarchitecture data, the effect of noisy environment and mimicry attack and the effect of monitoring process to the host and the guest VM.

5.5.1 Evaluation Setup

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 5.1).

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

1. Three CSCa implementations from Gruss [GMWM16]. 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 [YF14].

3. Another Flush+Reload implementation from Hornby that spies on the victim's browsing destinations [Hor16].

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). This scenario is needed since every guest VM would go through this scenario at some time in its life-cycle.

2. Web application scenario. We used RUBiS the same way described in Section 4.2.1. We collected the KVM events for three workload scenarios, 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.

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. 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' non-blocking random source. We performed: cat /dev/urandom > /dev/null. This operation is another well-known stress test for CPU.

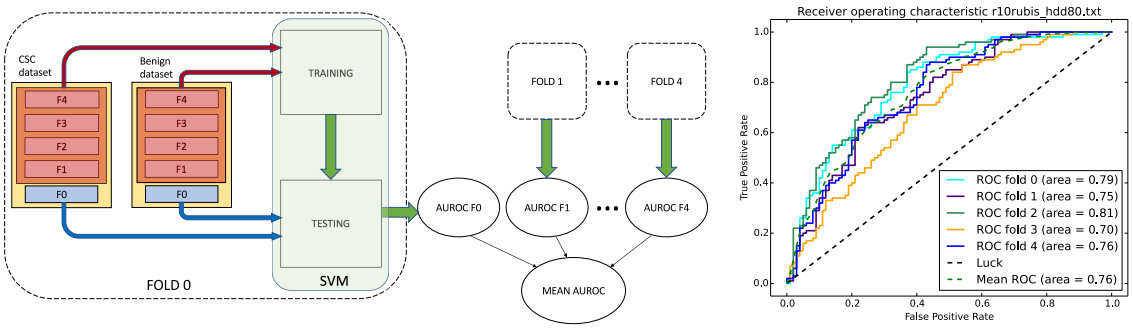


Figure 5.2: (a) Dataset distribution using the k-fold cross-validation technique. (b) An example of a 5-fold Cross Validation ROC graph

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, then deallocating nodes of a big binary tree.

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

Machine learning setup

In the evaluation phase we used the Support Vector Machine method [BGV92] with a Radial-based Function (RBF) to perform binary classification (CSCa or non-CSCa).

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

Table 5.2: 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)	RUBiS 2000 clients
		Flush+Reload (Hornby)	Mail Server
			Urandom generator
			Lucas-Lehmer
			Binary Tree

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. The initial number of features (unique sequences of events) in the raw data was 271. After the pre-processing stage, the number of features was reduced to 69.

To make sure that the evaluation results are not dependent on one particular random choice of learning datasets, we applied 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.

The scheme for dataset treatment and an illustration of its outputs are shown in Figure 5.2.

5.5.2 The Binary Class Classification for the CSCa Detection

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. However, this assumption is not always the case in the real life operation. New

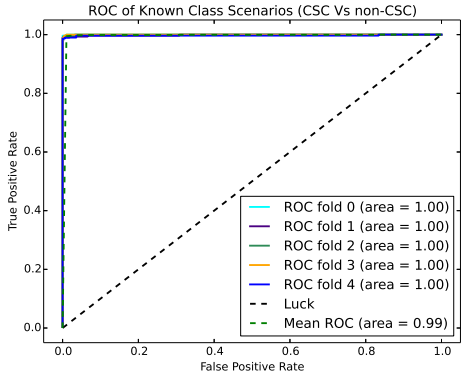


Figure 5.3: ROC of the trained CSCa scenario Vs the trained non-CSCa scenario

scenarios, or the zero days scenarios might still have chances to pop-up sometime. To factor in these zero days scenarios in our evaluation, 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 (the zero days scenario), therefore it was not used in the training process and would only be used in the test phase. The arrangement of all collected scenarios for use in this evaluation process is given in Table 5.2.

Evaluation 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 5.2.(a). In this test, we do not yet use the Untrained Class scenarios of Table 5.2. The results of this test are given in Figure 5.3.

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.

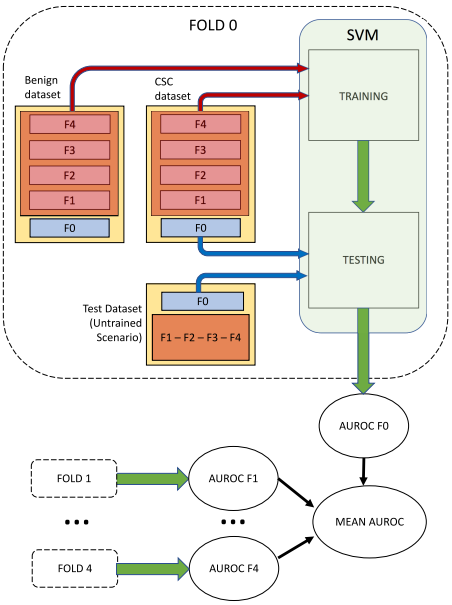


Figure 5.4: The evaluation scheme for each of the untrained scenario

Evaluation 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 (i.e. not only the trained scenario class). 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 5.4. The expected results should give a low AUC value (around 0.50 AUC) for the Untrained-Positive class scenarios and high AUC value (around 0.99 AUC) for the Untrained-Negative class scenarios. The actual results are given in Figure 5.5.

Figure 5.5.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

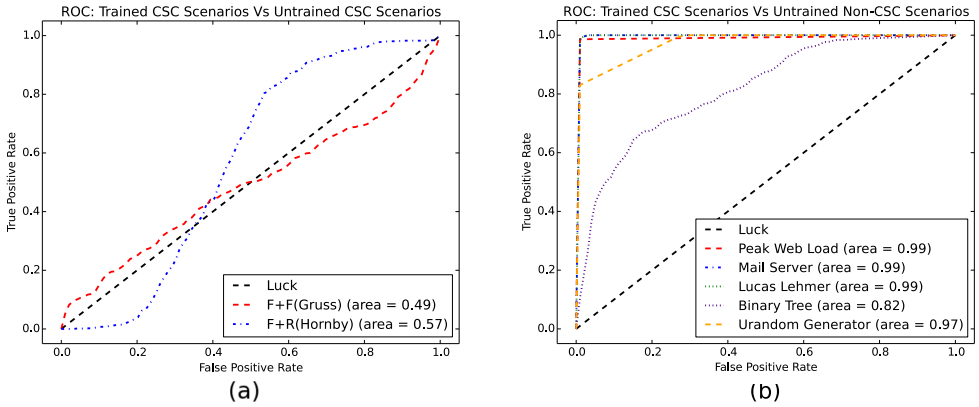


Figure 5.5: (a) Trained Positive Class Vs Un-trained Positive Class (CSC). (b) Trained Positive Class Vs Un-trained Negative Class (non-CSC).

negative rate detections.

On the other hand, Figure 5.5.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.

5.5.3 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 [Fis36] 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 is 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 5.3.

Table 5.3 lists only three 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.

Regular Workload

In the case of a regular workload, such as web server operation and mail server operation, Table 5.3 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.

Table 5.3: Three features for each non-CSCa scenario with the highest Fisher score when compared to the CSCa scenarios.

Vs CSCa	KVM event Sequence	FS	Non-CSCa Seq. Stat.			CSCa Seq. Stat.		
			Md	Mn	SD	Md	Mn	SD
RO	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
CPU	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
Mem	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

RO: Regular Operation, CPU: CPU Intensive Operation, Mem: Memory Intensive Operation, FS: Fisher Score, Md: Median, Mn: Mean, SD: Standard Deviation

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 three listed in Table 5.3 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 sub-operations (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.

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 5.3 for CPU-intensive operation shows that only two of the three 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.

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 5.3 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

Table 5.4: Fisher Score for the Evaluation on Nehalem 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

page fault exceptions than the CSCa operations. Page fault exceptions may happen for 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.

5.5.4 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 compared the Fisher score of two hosts with different microarchitecture. For our default setup above, we used Intel Xeon Dual core 3040 1.86 GHz (Conroe), 64KB L1 (32KB L1d + 32KB L1i), 2MB L2 and 8GB system memory. For comparison, 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), 36MB L3 cache and 32GB 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 VI.A.1). Table 5.4 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 5.3 (Conroe setup) and Table 5.4 (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.

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

(a)

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

(b)

Figure 5.6: (a) A mimicry attempt by reducing the spy frequency (b) A mimicry attempt by introducing a diversion function.

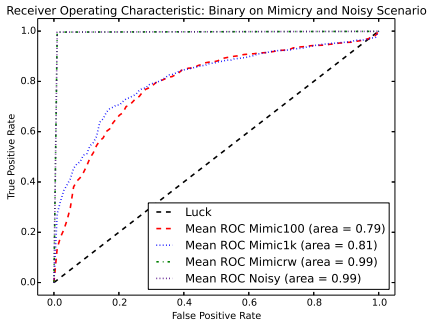


Figure 5.7: ROC of several mimicry attack and noisy case scenario.

5.5.5 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 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 (Figure 5.6.a.). 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 in the Gruss's Flush+Reload implementation (Figure 5.6.b.).

Using the previous SVM Binary Class Classification, the results are given in Figure 5.7. 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 is 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 [IIES15, Fog16, IIES14, IES15]. The results of our own attempt to show the effect of the noisy environment and mimicry attempts on the CSCa output are given in Figure 5.7 and Table 5.5.

Figure 5.8 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

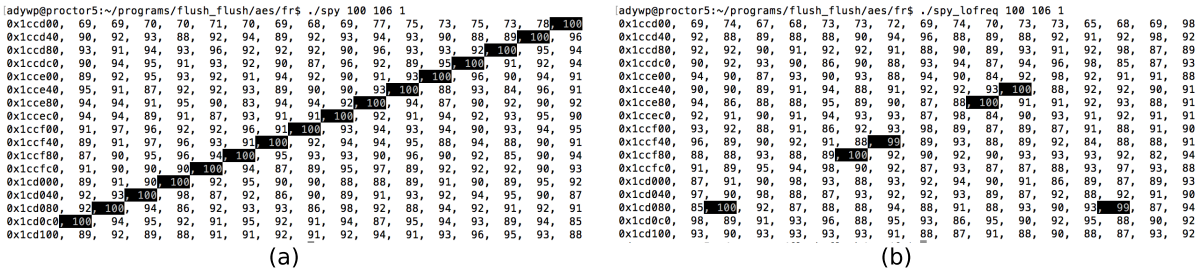


Figure 5.8: Cache line pattern of k_0 = 0xf_ (a) in clean CSCa implementation (b) in the CSCa with a reduced probe frequency.

in Table 5.5.

Table 5.5 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, lead to some additional processing time, reducing the CSCa resolution timers and increasing the probability of missing the real encryption events from the victim.

In short, while noisy environments and mimicry may obfuscate the CSCa signatures, these also make the CSCa less effective.

5.5.6 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 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

Table 5.5: 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)

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 5.9 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).

5.6 Summary and Outlook

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 creates 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

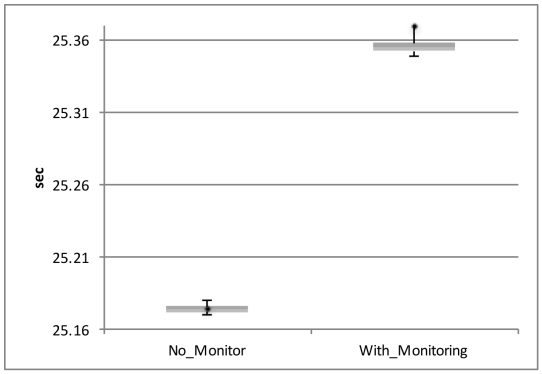


Figure 5.9: Comparison of performance impact in guest VM when the KVM events at the host was monitored and when it was not monitored.

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 were 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 a 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.

6 Discussions, Recommendations and Future Work

During our work, we encountered several interesting topic to be discussed or to be given as recommendation for current virtualization operational and the future works.

6.1 The Approach for the Other Hypervisors

The static instrumentation method we promote in this thesis requires two things. First, the hypervisor has to be an open source one, so that the administrator can modify to add tracepoints. Most of the well-known hypervisors come with default tracepoints for performance and debugging tracing. However, for the VM monitoring purpose, the administrator usually needs to add their own tracepoints. For example in Qemu, there are no default tracepoints for network operations. To add a network monitoring capability into the Qemu framework, we have to add several new user-defined tracepoints in the source code. This requirement will be hard to satisfy by the proprietary hypervisor, such as Microsoft Azure.

Since a hypervisor is used to host multiple guest VMs, the second requirement is the ability of the hypervisor to isolate one guest VM information from the other. In Qemu and Xen, each vCPU thread is represented by one user process and can be identified by its Process ID (illustration is given in Figure 2.2).

Another useful requirement is the availability of an efficient data collection framework. The monitoring process will consume an additional portion of host system computation process. Therefore, we need a data collection tool that is able to collect enough information by using the least resources from the host. In this approach, we use LTTng tool. In Xen system, the user can use the combination

of Xentrace and Xenalyze tools.

6.2 Development of an Operational Monitoring System

As the works in this thesis is still in an experimental phase, a direct future direction for this work is a development of an operational monitoring system to be deployed in a production environment. There are several discussion points that we found during our experimental phase that might be useful in the development phase.

6.2.1 Going deep into the tracepoints data.

In all our experimental work, we only utilise the tracepoint name as our data. On the other hand, a tracepoint actually can consists of multiple information (for examples, see Algorithm 2 and Figure 3.2). Additional information from each tracepoints to be used as features in this VM monitoring system can reduce the sematic gap issue and increase detection accuracy.

6.2.2 Utilising the Decision Confidence

Most of the machine learning implementation applies scoring detection output instead of just giving a binary decision. The detection scores usually depict the distance of data to the classifier border. This feature can be beneficial to introduce decision confidence for the output status from the ADS. For instance, an administrator can setup a high level alarm that can only be triggered if the decision confidence is raised or lowered to a certain threshold.

In our proposed method, the confidence score should be given in three layer.

Tracepoints Sequence Layer

The decision making in this layer should decide based on its detection score whether a sequence of tracepoints is in positive class or in negative class. In our work, we did not choose a specific threshold but instead used all possible

threshold to draw the Receiver Operating Characteristic (ROC) curve. However, in operational mode, we have to pick a certain threshold that maximise the trade-offs between true positive (benefits) and false positive (costs).

One possible option is the Youden's Index [You50]. Youden Index J is formulated as:

$$J = Sensitivity + Specificity - 1$$

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 ROC with the maximum distance to the chance (diagonal) line.

However, in practice, this approach is rarely used. First, in a real life operation, an administrator will not likely have the luxury of having enough data to create an ROC diagram. To create an ROC we need to have an anomaly dataset, which is a rare commodity in reality. Second, an administrator will most likely put more emphasis to minimize false alarms, and therefore prefers to use a maximum value of accepted false positives.

Observation Layer

The decision making in this layer should decide based on the number of detected positive and negative sequence in an observation, whether the observation is in positive observation class or in negative observation class. A simple example would be to set a 10% threshold, where an observation will be deemed as an anomaly if the number of positive sequences exceeds a tenth of the total sequences inside the observation. Another approach would be utilizing the time series of the sequence class in the observation.

Final Decision Layer

We argue that we should not only use a single observation status to make the decision, such as whether to raise the alarm or not. Therefore we should add another final decision making layer. The decision making in this layer should decide whether the monitoring system should raise the alarm or not based either

on the number of detected positive and negative sequence in a series of observation or based on the time series of the observation class.

The size of the observation series to use for an Anomaly Detection System (ADS) had been proposed by several previous studies, such as length of 6 in [HFS98] and length of 10 in [AW12]. We argue that these values are unique to each case, so further studies could be taken to decide a proper window size to improve the anomaly detection results.

6.3 A Repository of VM Image Dataset

There are many previous works that studied and tried to improve the security aspects of cloud system. Some of the topic for detection methods have been described in Section 4.3 and Table 4.6. However, the research variables are highly varied from one research to the other. That makes it hard to measure the advancement of research in this field, as there is no standard for comparison. In their paper, Litchfield and Shahzad argued that the lack of dataset leads to lack of consistent risk assessment process and mitigation [LS17]. Indeed, it is hard to suggest one standard research environment considering the vast landscape of cloud computing system. A simple solution would be in a form of an open repository of standardised Virtual Machine (VM) dataset. The purpose of this repository are to allow validation of results and make the peer review-easier, also to ensure the continuation of any research work in this field.

Within this work, we have shared three datasets for each of our experiments. However, those are not raw datasets, as they are already a results of a specific data collection and pre-processing. For a similar VM operation scenario, other researcher might want to propose a different data collection method. Therefore, in the future, we would like to see a repository of VM scenario datasets in the form of VM image. By using the VM image format, the dataset can retain the syntactical integrity of both the VM structure and any individual individual values within, so that automated analysis is possible.

6.4 A Further Research In the Side Channel Attack Detection

The duel between the attacker and the defender team in cybersecurity field can often seen as a race to the lowest layer of computation. From the apps to the API to the OS userspace to the kernel space and now to the hardware. Whoever command the lowest layer holds the ground.

The strongest case to the race analogy above is the rise of processor and cache-based vulnerability within the last five years. It started with the emergence of Cache-based Side Channel attack and many of its variants at the start of 2000s. This phenomena is starting to gain recognition in early 2018 when two vulnerabilty, the Spectre (CVE-2017-5753 and CVE-2017-5715) and the Meltdown (CVE-2017-5754) were introduced to the public.

In short, both Spectre and Meltdown vulnerability leverage the main feature of modern CPU to speed up operation, such as instruction pipelining and speculative execution. The features may leave residual effect that can be observed to reveal patterns of private data to the attackers. For example, the attacker can use these vulnerabilities to read the privilige domains of the memory that can lead to the attack such as cross-VM side channel and hypervisor escape.

Since the vulnerabilities are based on hardware (processor and cache) operation, there is not much all security mechanism in application and OS layer (include the separation methods in virtualization environment) can do to prevent it. This problem can lead to an interesting research topic, such as to detect if a VM is hosting a processor vulnerability-based attack against its peer VM or against its host system.

6.5 Program Analysis Approach for Software Defined Networking Security

Software defined networking (SDN) has established a new method for creating and administering networks, but has also changed the attack surface that is presented by networks. SDN adds a software-based control and management layer above

the hardware layer (data layer) to simplify the network operations. As the side effect, SDN introduces new vulnerabilities, especially software-based threats, such as software virusses and malwares that are not present in traditional networks.

The controller in the control plane is the fundamental element used for all operations of SDN data plane management. This controller is a software applications that usually implemented as a dedicated server or a VM in the cloud. Since the controller can acts as a single point of failure on the network, it is necessary to defend the controller against all the possible attacks. The attacks may varied, that includes the network-based attack such as network DDoS attack, the host-based or VM-based attack such as the malwares and SDN-specific attack such as the state manipulation attack [XHH+17] and the crosspath attack [CLX+19].

The detection for attacks against SDN controller can be a usefull feature. It is an interesting topic to study how to leverage the program analysis techniqueto be used to detect any attack against the SDN controller.

7 Conclusion

In this thesis, we investigate a guest VM monitoring method that can work independently outside the monitored guest VM, without losing much of the semantic information and without high computation cost for either the host and the guest VM. We propose a method that embeds multiple tracepoints inside the source code of the hypervisor or the Virtual Machine Monitor (VMM) (Static Instrumentation). During the hypervisor operation we collect the tracepoints execution data to dynamically monitor the operational flow of a guest VM (Dynamic Source Code Analysis). Since the instrumentation was carried out within the underlying process of the guest VM's instances, we believe that the dynamic pattern of the tracepoints sequences can indirectly describe the operations of the VM.

We first applied this dynamic source code analysis with a static instrumentation method to the user space of the Qemu-KVM hypervisor. We captured the tracepoints from the Qemu operation and used it for an Anomaly Detection System. We emulated a web server VM and multiple attack scenarios, such as DDoS for network-based attack and Flush-Reload attack for virtualization-based attack. We factored in the mimicry attack scenario. We compared several machine learning algorithms for the monitoring data analysis process. Finally, we compared our detection result with system-call data analysis. Our evaluation showed that monitoring guest VM using dynamic source code analysis with a static instrumentation method gave better detection results compared to the system-call data, with minimum computation cost. However, we had subpar results when trying to detect malicious activities that work upon the host CPU. That is because, on Qemu-KVM combination, CPU operations are performed natively through the KVM kernel module.

We investigated further this dynamic source code analysis with a static instrumentation method at the kernel layer by instrumenting the KVM module.

We used this method to implement a signature-based intrusion detection system and try to detect multiple variants of Cache-based Side Channel Attack (CSCA) including a new stealthier variant called Flush+Flush attack. In our evaluation phase, we showed that our proposed approach could give good classification results for the Cache-based Side Channel attack. Our attempt is the first successful attempt to detect this Flush+Flush attack in the virtualization environment.

Acknowledgements

First and foremost, I would like to praise and thank God, the Almighty, who has granted countless blessings, knowledge, and opportunity to me, so I am able to complete this thesis.

This thesis appears in its current form due to the assistance and guidance of many people. I would therefore like to offer my sincere thanks to all of them.

I want to express my deepest gratitude to my supervisor, Professor Youki Kadobayashi. His unending trust in me, even at the time when I lose it myself, is my main power for accomplishing this task. I thank him further for his support during the whole period of the study, for his wisdom and insight and especially for his patience during the writing process.

I am grateful to Professor Kazutoshi Fujikawa and Professor Hajimu Iida for kindly reserving their invaluable time to give their thorough review and constructive suggestions as my thesis committee. Their help has considerably improved the quality of my work.

My special thank you to all members of the informal VINE-IPLAB workgroup, Associate Professor Takeshi Okuda, Assistant Professor Doudou Fall and Dr. Marius Liviu Georgescu. As a group, I miss our heated discussion time, where I was able to ponder all the available venues for my work. In person, each of them is my IPLab's guardian angel, the constant source for support, in and out of the laboratory.

I would like to extend my thank you to Assistant Professor Shigeru Kashihara, Associate Professor Daisuke Miyamoto, Assistant Professor Hiroaki Hazeyama, Associate Professor Yuzo Taenaka for all their kind help, advice and encouragement.

I owe my appreciation to Professor Michael Dean Barker for sparing his time from his busy schedule to improve the English of my journal manuscripts.

Thank you to IPLab members past and present during my study, Jema David Ndibwire, Louis Zamora, Christopher Yap, Adlizan "Ben" Bin Ibrahim, Sirikarn Pukkawanna, Wataru Tsuda, Kazuya Okada and all. Because of them, the IPLab becomes not only a great place to conduct research but also an enjoyable place to be, for cooperation and friendship.

I would like to express my appreciation to the IPLab's present and past administrative staff members — Haruna Okumura, Natsue Tanida, Yukika Nishitouge, Naoko Omori, Yoko Inada - for their indefatigable aid.

My extended appreciation to NAIST International Student Affair Section staff members for all their help to me and my family during our stay in NAIST.

I wish to thank the NAIST International Scholarship Programme for providing me the opportunity to study in Japan through its excellent scholarship.

I want to acknowledge the help provided by Iida Foundation, through Professor Masashi Kawaichi and Professor Yasumasa Bessho, who accepted me in Ayameike apartment during one time of my study.

My appreciation to all faculty members of the Department of Informatics, Universitas Hasanuddin, Indonesia, especially Dr. Amil Ahmad Ilham as the Head of Department and Dr. Muhammad Niswar as the Head of UbiCoN laboratory for all the supports and encouragement during the final phase of this Dissertation process.

There are no words that can testify my gratitude for both my parents Jodius Paundu and Siti Fatimah Mosseng, for their endless love, support and encouragement. To my lovely wife Stella and the boys Abia and Asa, you guys are the source of my strength through the testing time. Thank you for putting up with me. To my big family Opa and Oma Ben, Budi, Mimo, 'aunty' Nita, Palen, Malen and the Jak's Geng, thank you for all your support throughout the entire process.

Finally, I would like to dedicate this special space to my late supervisor Professor Suguru Yamaguchi, to whom I owe my greatest gratitude for giving me the chance to start all of this in the first place. I dedicate this work to his memory.

References

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS XII Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13. ACM New York, October 2006. ISBN 1-59593-451-0. doi:10.1145/1168919.1168860.

[ACL15] Amr S. Abed, T. Charles Clancy, and David S. Levy. Applying bag of system calls for anomalous behavior detection of applications in linux containers. In *Globecom Workshops (GC Wkshps)*. IEEE, 2015. doi:10.1109/GLOCOMW.2015.7414047.

[AEW09] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. *Red Hat Inc.*, 2009. URL `https://www.kernel.org/doc/ols/2009/ols2009-pages-19-28.pdf`.

[Agg13] Charu C. Aggarwal. *Outlier Analysis*. Springer International Publishing, 2013. ISBN 978-3-319-47577-6. doi:10.1007/978-3-319-47578-3.

[And12] Paul Anderson. Measuring the value of static-analysis tool deployments. *IEEE Security & Privacy*, 10(3):40–47, June 2012. ISSN 1540-7993. doi:10.1109/MSP.2012.4.

[ATJ+10] Alberto Avritzer, Rajanikanth Tanikella, Kiran James, Robert G. Cole, and Elaine J. Weyuker. Monitoring for security intrusion using performance signatures. *WOSP/SIPEW*, pages 93–104, 2010. doi:10.1145/1712605.1712623.

[AW12] Suaad S. Alarifi and Stephen D. Wolthusen. Detecting anomalies in iaas environment through virtual machine host system call analysis. In *The 7th International Conference for Internet Technology and Secured Transactions (ICITST 2012)*, 2012. ISBN 978-1-908320-08-7.

[AW13] Suaad S. Alarifi and Stephen D. Wolthusen. Anomaly detection for ephemeral cloud iaas virtual machines. In *7th International Conference on Network and System Security*. Springer, Berlin, Heidelberg, 2013. ISBN 978-3-642-38631-2. doi:10.1007/978-3-642-38631-2_24.

[aws17] Aws adopts home-brewed kvm as new hypervisor. URL `https://www.theregister.co.uk/2017/11/07/aws_writes_new_kvm_based_hypervisor_to_make_its_cloud_go_faster/`, 2017. Accessed: 2019-03-30.

[Bal99] Thoms Ball. The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes*, 24(6):216–234, November 1999. doi:10.1145/318774.318944.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, The University of Illinois at Chicago, 2005. URL `https://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[BFS+16] Jon-Michael Brook, Scott Field, Dave Shackleford, Vic Hargrave, Laurie Jameson, Michael Rosa, and Victor Chin. The treacherous 12: Cloud computing top threats in 2016. Technical report, Cloud Security Alliance, 2016.

[BGN17] Arnab Kumar Biswas, Dipak Ghosal, and Shishir Nagaraja. A survey of timing channels and countermeasures. *ACM Computing Surveys (CSUR)*, 50, April 2017. doi:10.1145/3023872.

[BGV92] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational Learning Theory - COLT92*, pages 144–152. ACM, 1992. doi:10.1145/130385.130401.

[Bin07] David Binkley. Source code analysis: A road map. In *FOSE '07 2007 Future of Software Engineering*, pages 104–119. IEEE Computer Society, May 2007. ISBN 0-7695-2829-5. doi:10.1109/FOSE.2007.27.

[BN05] Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, March 2005. doi:10.1017/S0956796804005453.

[CdA10] Michele Costa and Luca de Angelis. Model selection in hidden markov models: a simulation study. *Quaderni di Scienze Statistiche Dipartimento "Paolo Fortunati", Universita di Bologna*, (7):17, 2010. URL `https://EconPapers.repec.org/RePEc:bot:quadip:wpaper:104`.

[Cha18] Ramaswamy Chandramouli. Security recomendations for hypervisor deployment on servers. Technical Report 800-125A, NIST - National Institute of Standards and Technology, January 2018. doi:10.6028/NIST.SP.800-125A.

[Che11] Qichang Chen. *An integrated static and dynamic program analysis framework for checking concurrency-related programming errors*. PhD thesis, University of Wyoming Laramie, WY, USA, 2011. ISBN 978-1-124-88165-2.

[CLX+19] Jiahao Cao, Qi Li, Renjie Xie, Kun Sun, Guofei Gu, Mingwei Xu, and Yuan Yang. The crosspath attack: Disrupting the sdn control channel via shared links. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 19–36, Berkeley, CA, USA, August 2019. USENIX Association. ISBN 978-1-939133-06-9. URL `http://dl.acm.org/citation.cfm?id=3361338.3361341`.

[CMZ02] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of ejb applications. In *OOPSLA '02 Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 246–261, 2002. doi:10.1145/582419.582443.

[CSY16] Marco Chiapetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, December 2016. doi:10.1016/j.asoc.2016.09.014.

[DFK+13] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: a tool for the static analysis of cache side channels. In *Proceedings of the 22nd USENIX conference on Security, SEC'13*, pages 431–446. USENIX, USENIX Association, 2013. ISBN 978-1-931971-03-4. URL `https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev`.

[DGLZ+11] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP '11*, pages 297–312. IEEE Computer Society, 2011. doi:10.1109/SP.2011.11.

[DJL+12] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8, January 2012. doi:10.1145/2086696.2086714.

[DKRC13] Frank Doelitzscher, Martin Knahl, Christoph Reich, and Nathan Clarke. Anomaly detection in iaas clouds. In *IEEE International Conference on Cloud Computing Technology and Science*, 2013. doi:10.1109/CloudCom.2013.57.

[DMD+02] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher. Deli: A new run-time control point. In *Proceedings of the 35th Annual Symposium on Microarchitecture (MICRO35)*, pages 257–270. IEEE, November 2002. ISBN 0-7695-1859-1. ISSN 1072-4451. doi:10.1109/MICRO.2002.1176255.

[DNG12] Daniel J. Dean, Hiep Nguyen, and Xiaohui Gu. Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *ICAC '12 Proceedings of the 9th international conference on Autonomic computing*, 2012. doi:10.1145/2371536.2371572.

[Dre08] Ulrich Drepper. The cost of virtualization. *Queue - Virtualization*, 6(1):28–35, February 2008. doi:10.1145/1348583.1348591.

[eni17] Security aspects of virtualization. Technical report, The European Union Agency for Network and Information Security (ENISA), February 2017. URL `https://www.enisa.europa.eu/news/enisa-news/enisa-study-on-the-security-aspects-of-virtualization`. doi:10.2824/955316.

[Ern04] Michael D. Ernst. Invited talk static and dynamic analysis: synergy and duality. In *PASTE '04 Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, page 35. ACM New York, June 2004. ISBN 1-58113-910-1. doi:10.1145/996821.996823.

[eur18] Ict usage in enterprises in 2018. *eurostat newsrelease*, 193, December 2018.

[fin19] *Multi-Cloud Fundamental to Financial Services Transformation.* Canonical, January 2019.

[Fis36] Ronald Aylmer Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, pages 179–188, 1936. doi:10.1111/j.1469-1809.1936.tb02137.x.

[FK09] David F. Ferraiolo and D. Richard Kuhn. Role-based access controls. *CoRR*, abs/0903.2171, 2009. URL `http://arxiv.org/abs/0903.2171`.

[FL13] Yangchun Fu and Zhiqiang Lin. Bridging the semantic gap in virtual machine introspection via online kernel data redirection. *ACM*

Transactions on Information and System Security (TISSEC), 16-2(7), September 2013. doi:10.1145/2505124.

[Fog16] Anders Fogh. Cache side channel attacks: Cpu design as a security problem. URL `https://conference.hitb.org/hitbsecconf2016ams/sessions/cache-side-channel-attacks-cpu-design-as-a-security-problem/`, 2016.

[FVWGT14] Philippe Fournier-Viger, Cheng-Wei Wu, Antonio Gomariz, and Vincent S. Tseng. Vmsp: Efficient vertical mining of maximal sequential patterns. In *Advances in Artificial Intelligence: 27th Canadian Conference on Artificial Intelligence*, volume 8436 of *Lecture Notes in Computer Science*, pages 83–94. Springer, May 2014. ISBN 978-3-319-06483-3. doi:10.1007/978-3-319-06483-3_8.

[GDJ02] Yann-Gael Gueheneuc, Remi Douence, and Narendra Jussien. No java without caffeine: A tool for dynamic analysis of java programs. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*. IEEE, September 2002. ISBN 0-7695-1736-6. ISSN 1938-4300. doi:10.1109/ASE.2002.1115000.

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan 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, DIMVA2016*, volume 9721, pages 279–299. Springer-Verlag, July 2016. doi:10.1007/978-3-319-40667-1_14.

[GR03] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *The 10th Annual Network and Distributed System Security Symposium*, 2003.

[GWS11] Bernd Grobauer, Tobias Walloschek, and Elmar Stocker. Understanding cloud computing vulnerabilities. *IEEE Security and Privacy Magazine*, 9(2):50–57, May 2011. doi:10.1109/MSP.2010.115.

[GYCH16] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 2016. URL `https://eprint.iacr.org/2016/613.pdf`. doi:10.1007/s13389-016-0141-6.

[HACL13] Yi Han, Tansu Alpcan, Jeffrey Chan, and Christopher 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 2013*, volume 8252 of *Lecture Notes in Computer Science*, pages 99–118, November 2013. doi:10.1007/978-3-319-02786-9_7.

[HF15] Nishad Herath and Anders Fogh. These are not your grand daddy's cpu performance counters. URL `https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-Grand-Daddys-CPU-Performance-Counters-CPU-Hardware-Performance-Counters-For-Security.pdf`, 2015.

[HFS98] Stefen A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system call. *Journal of Computer Security*, 6(3), 1998. ISSN 0926-227X. URL `http://dl.acm.org/citation.cfm?id=1298081.1298084`.

[HL02] Sudheendra Hangal and Monica Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02 Proceedings of the 24th International Conference on Software Engineering*, pages 291–301. ACM New York, May 2002. ISBN 1-58113-472-X. doi:10.1145/581339.581377.

[Hor16] Taylor Hornby. Side-channel attacks on everyday applications. URL `https://www.blackhat.com/docs/us-16/materials/us-16-Hornby-Side-Channel-Attacks-On-Everyday-Applications-wp.pdf`, 2016.

[HPY+14] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Hujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *International Symposium on Software Testing and Analysis (ISSTA'14)*, July 2014. doi:10.1145/2610384.2610407.

[HSJ] Alice Hutchings, Russell G. Smith, and Lachlan James. Criminals in the cloud: Crime, security threats, and prevention measures. In Russell G. Smith, Ray Chak-Chung Cheung, and Laurie Yiu-Chung Lau, editors, *Cybercrime Risks and Responses*, Palgrave Studies in Cybercrime and Cybersecurity, pages 146–162. Palgrave Macmillan, London. ISBN 978-1-349-55788-2. doi:10.1057/9781137474162_10.

[Hu92] Wei-Ming Hu. Lattice scheduling and covert channels. In *IEEE Symposium on Security and Privacy*, pages 52–61, 1992. ISBN 0-8186-2825-1. doi:10.1109/RISP.1992.213271.

[HZZ+13] Tian Huang, Yan Zhu, Qiannan Zhang, Yongxin Zhu, Dongyang Wang, Meikang Qiu, and Lei Liu. An lof-based adaptive anomaly detection scheme for cloud computing. In *IEEE 37th Annual Computer Software and Applications Conference Workshops (COMPSACW)*, 2013. doi:10.1109/COMPSACW.2013.28.

[IES15] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing and its application to aes. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy, SP'15*, pages 591–604, May 2015. doi:10.1109/SP.2015.42.

[IES16] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Mascat: Stopping microarchitectural attacs before execution. *IACR Cryptology ePrint Archive*, page 1196, 2016. URL `https://eprint.iacr.org/2016/1196`.

[İGI+16] Mehmet Sinan İnci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery

on the cloud. In *Proceeding of the International Conference on Cryptographic Hardware and Embedded Systems, CHES2016*, pages 368–388, August 2016. doi:10.1007/978-3-662-53140-2_18.

[IIES14] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-vm attack on aes. In *Proceeding of the Research in Attacks, Intrusions and Defenses, RAID 2014*, volume 8688 of *Lecture Notes in Computer Science*, pages 299–319, 2014. doi:10.1007/978-3-319-11379-1_15.

[IIES15] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Know thy neighbor: Crypto library detection in cloud. In *Proceedings on Privacy Enhancing Technologies*, volume 2015, 2015. doi:10.1515/popets-2015-0003.

[isc19] 2019 cloud security report. *Cybersecurity Insiders*, 2019.

[JWX07] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *CCS'07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 128–138. ACM, 2007. doi:10.1145/1315245.1315262.

[KKL+07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. Kvm: the linux virtual machine monitor. In *Proceedings of the 2007 Ottawa Linux Symposium - OLS07*, 2007.

[Kol15] Daan Kolthof. Crime in the cloud: An analysis of the use of cloud services for cybercrime. In *23rd Twente Student Conference on IT*, June 2015. URL `https://pdfs.semanticscholar.org/9ecb/a6d0edfeb65ce68e722daa68056c290d6331.pdf`.

[KPMR12] Taesoo Kim, Marcus Peinado, and Gloria 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, Security'12*. USENIX, August 2012. ISBN 978-931971-95-9.

[LGS+16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *Proceeding of the 25th USENIX Security Symposium*, pages 549–564. USENIX, August 2016. ISBN 978-1-931971-32-4. URL `http://dl.acm.org/citation.cfm?id=3241094.3241138`.

[LGY+16] Fangfei Liu, Qien Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016. doi:10.1109/HPCA.2016.7446082.

[LL14] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-47*, pages 203–215. IEEE, December 2014. doi:10.1109/MICRO.2014.28.

[LS17] Alan Litchfield and Abid Shahzad. A systematic review of vulnerabilities in hypervisors and their detection. In *Proceeding of the Twenty-third Americas Conference on Information Systems - Information Systems Security and Privacy (SIGSec)*, 2017. ISBN 978-0-9966831-4-2.

[LT13] Xin Li and Hua Vy Le Thanh. A formal framework for access rights analysis. *CoRR*, abs/1307.2964, 2013. URL `http://arxiv.org/abs/1307.2964;https://dblp.org/rec/bib/journals/corr/LiT13`.

[LTXZ14] Kai Luo, Shouzhong Tu, Chunhe Xia, and Dan Zhou. Detecting compromised vm via application-aware anomaly detection. In *Proceedings of the 10th International Conference on Computational Intelligence Security*, 2014. doi:10.1109/CIS.2014.109.

[MG11] Peter Mell and Timothy Grance. The nist definition of cloud computing. Technical Report 800-145, NIST - National Institute of Standards and Technology, September 2011. URL `https://www.nist.gov/publications/nist-definition-cloud-computing`.

[MKA+13] Aleksandar Milenkoski, Samuel Kounev, Alberto Avritzer, Nuno Antunes, and Marco Vieira. On benchmarking intrusion detection systems in virtualized environments. Technical Report SPEC-RG-2013-002, SPEC RG IDS Benchmarking Working Group, 2013. URL `https://arxiv.org/abs/1410.1160`.

[MSR15] Soo-Jin Moon, Vyas Sekar, and Michael 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*, pages 1595–1606. ACM, October 2015. doi:10.1145/2810103.2813706.

[NBL+10] Qun Ni, Elisa Bertino, Jorge Lobo, Carolyn Brodie, Clare-Marie Karat, John Karat, and Alberto Trombeta. Privacy-aware role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 13(3), July 2010. doi:10.1145/1805974.1805980.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary in- strumentation. *ACM SIGPLAN Notices - Proceedings of the 2007 PLDI conference*, 42(6):89–100, June 2007. doi:10.1145/1273442.1250746.

[ODCM14] Jorge Orestes Cerdeira, Pedro Duarte Silva, Jorge Cadima, and Manuel Minhoto. *subselect: Selecting variable subsets*, 2014. URL `http://CRAN.R-project.org/package=subselect`, R package version 0.12-4.

[OKSK15] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos 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*, pages 1406–1418. ACM, October 2015. doi:10.1145/2810103.2813708.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran 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, pages 1–20. Springer-Verlag, February 2006. doi:10.1007/11605805_1.

[OV03] Ferdinand Osterreicher and Igor Vajda. A new class of metric divergences on probability spaces and its applicability statistics. *Annals of the Institute of Statistical Mathematics*, 55(3):639–653, September 2003. ISSN 1572-9052. doi:10.1007/BF02517812.

[Pay16] Mathias Payer. Hexpads: A platform to detect stealth attacks. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems, ESSoS 2016*, volume 9639, pages 138–154. Springer-Verlag, April 2016. doi:10.1007/978-3-319-30806-7_9.

[PBSL13] Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 International Workshop on Security in Cloud Computing (Cloud Computing'13)*, pages 3–10, 2013. doi:10.1145/2484402.2484406.

[PCFY07] Marco Pistoia, Satish Chandra, Stephen J. Fink, and Eran Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2):265–288, 2007. ISSN 0018-8670. doi:10.1147/sj.462.0265.

[PJAG12] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking api protocol conformance with mined multi-object specifications. In *ICSE 2012 Proceedings of the 34th International Conference on Software Engineering*, pages 925–935. IEEE, IEEE Press, June 2012. ISBN 978-1-4673-1067-3. URL `http://dl.acm.org/citation.cfm?id=2337223.2337332`.

[POKY16a] Ady Wahyudi Paundu, Takeshi Okuda, Youki Kadobayashi, and Suguru Yamaguchi. Leveraging static probe instrumentation for vm-based anomaly detection system. In *Information and Communications Security*, Lecture Notes in Computer Science. Springer International Publishing Switzerland, 2016.

[POKY16b] Ady Wahyudi Paundu, Takeshi Okuda, Youki Kadobayashi, and Suguru Yamaguchi. Sequence-based analysis of static probe instrumentation data for a vmm-based anomaly detection system. In *2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud)*, June 2016. doi:10.1109/CSCloud.2016.51.

[PZH13] Michael Pearce, Sherali Zeadally, and Ray Hunt. Virtualization: Issues, security threats, and solutions. *ACM Computing Surveys*, 45(2), February 2013. doi:10.1145/2431211.2431216.

[R C14] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014. URL `http://www.R-project.org/`.

[RANR15] Muhammad Hassan Raza, Adewale Femi Adenola, Ali Nafarieh, and William Robertson. The slow adoption of cloud computing and it workforce. In *Procedia Computer Science*, volume 52 of *3rd International Workshop on Survivable and Robust Optical Networks (IWSRON)*, pages 1114–1119. Elsevier, 2015. doi:10.1016/j.procs.2015.05.128.

[Ros14] Steven Rostedt. Ftrace kernel hooks, more than just tracing. LINUX Plumbers Conference 2014, 2014.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *16th ACM Conference on Computer and Communication Security*, pages 199–212, November 2009. doi:10.1145/1653662.1653687.

[SA15] P.V. Shijo and Salim Abdul Azeez. Integrated static and dynamic analysis for malware detection. In *Proceedings of the International Conference on Information and Communication Technologies, ICICT 2014*, volume 46 of *Procedia Computer Science*, pages 804–811. Elsevier, 2015. doi:10.1016/j.procs.2015.02.149.

[SDC+10] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, and Moshe Bach. Dynamic program analysis of microsoft windows applications. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, March 2010. doi:10.1109/ISPASS.2010.5452079.

[SIH14] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live api documentation. In *ICSE 2014 Proceedings of the 36th International Conference on Software Engineering*, pages 643–652. Association for Computing Machinery (ACM), June 2014. ISBN 978-1-4503-2756-5. doi:10.1145/2568225.2568313.

[SIYA11] Kuniyasu Suzaki, Kengo Ijima, Toshiki Yagi, and Cyrille Artho. Memory deduplication as a threat to the guest os. In *Proceedings of the Fourth European Workshop on System Security, EUROSEC'11*, April 2011. doi:10.1145/1972551.1972552.

[SL16] Daniele Sgandurra and Emil Lupu. Evolution of attacks, threat models, and solutions for virtualized systems. *ACM Computing Surveys*, 48(3), February 2016. doi:10.1145/2856126.

[SN17] Solutionary-NTT. Global threat intelligence report, 2017.

[SRK06] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(4):410–457, October 2006. doi:10.1145/1178625.1178628.

[SSCZ11] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu 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, DSNW'11*, pages 194–199. IEEE, June 2011. doi:10.1109/DSNW.2011.5958812.

[SSKK14] Guthemberg Silvestre, Carla Sauvanaud, Mohamed Kaaniche, and Karama Kanoun. An anomaly detection approach for scale-out storage systems. In *IEEE 26th International symposium on Computer Architecture and High Performance Computing*, 2014. doi:10.1109/SBAC-PAD.2014.42.

[SZCH15] Wenyao Sha, Yongxin Zhu, Min Chen, and Tian Huang. Statistical learning for anomaly detection in cloud server systems: a multi-order markov chain framework. In *IEEE transactions on cloud computing*, 2015. doi:10.1109/TCC.2015.2415813.

[Sze18] Jakub Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security*, 3(3):219–234, 2018. ISSN 2509-3436. URL `https://eprint.iacr.org/2016/479.pdf`. doi:10.1007/s41635-018-0046-1.

[VB16] Vanson-Bourne. Uk cloud adoption snapshot and trends for 2016. White Paper 15, Cloud Industry Forum, 2016.

[VC09] Ganesh Venkitachalam and Michael Cohen. Transparent page sharing on commodity operating systems. *Patent US7500048 B1*, 2009.

[VHK14] Owen Vallis, Jordan Hochenbaum, and Arun Kejariwal. A novel technique for long term anomaly detection in the cloud. In *6th USENIX conference on Hot Topics in Cloud Computing*, 2014. URL `https://www.usenix.org/conference/hotcloud14/workshop-program/presentation/vallis`.

[WD12] James Walden and Maureen Doyle. Savi: Static-analysis vulnerability indicator. *Security & Privacy*, 10(3):32–39, June 2012. ISSN 1558-4046. doi:10.1109/MSP.2012.1.

[WL07] Zhenghong Wang and Ruby 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*, pages 494–505. ACM, June 2007. doi:10.1145/1273440.1250723.

[WL08] Zhenghong Wang and Ruby B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture, MICRO-41*, pages 89–93. IEEE, November 2008. doi:10.1109/MICRO.2008.4771781.

[XHH+17] Lei Xu, Jeff Huang, Sungmin Hong, Jialong Zhang, and Guofei Gu. Attacking the brain: Races in the sdn control plane. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 451–468, Vancouver, BC, August 2017. USENIX Association. ISBN 978-1-931971-40-9. URL `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/xu-lei`.

[XX12] Zhifeng Xiao and Yang Xiao. Security and privacy in cloud computing. *IEEE Communications Surveys and Tutorials*, 15-2:843–859, July 2012. doi:10.1109/SURV.2012.060912.00182.

[YF14] Yuval Yarom and Katrina Falkner. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. In *Proceeding of the SEC'14, 23rd USENIX Conference on Security Symposium*, pages 719–732, August 2014. ISBN 978-1-931971-15-7. URL `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom`.

[You50] W. J. Youden. Index for rating diagnostic tests. *Cancer*, 3(1):32–35, 1950. doi:10.1002/1097-0142(1950)3:1<32::AID-CNCR2820030106>3.0.CO;2-3.

[ZJRR12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas 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*, pages 305–316, October 2012. doi:10.1145/2382196.2382230.

[ZJRR14] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceed-*

ings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS'14, pages 990–1003, November 2014. doi:10.1145/2660267.2660356.

[ZLB+12] Yulong Zhang, Min Li, Kun Bai, Meng Yu, and Wanyu Zang. Incentive compatible moving target defense against vm-colocation attacks in clouds. In *Proceeding of the IFIP International Information Security Conference on Information Security and Privacy Research, SEC2012*, pages 388–399. Springer, 2012. doi:10.1007/978-3-642-30436-1_32.

[ZR13] Yinqian Zhang and Michael 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 & communications security, CCS'13*, pages 827–838. ACM, November 2013. doi:10.1145/2508859.2516741.

[ZRZ16] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS'16*, pages 871–882. ACM, October 2016. ISBN 978-1-4503-4139-4. doi:10.1145/2976749.2978324.

[ZZL16] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *Research in Attacks, Intrusions, and Defenses. RAID 2016*, Lecture Notes in Computer Science. Springer, 2016.

Publication List

I Journal Articles

[1] Paundu,A.W., Doudou,F., Kadobayashi,Y., Niswar,M. "DCISION: A Dynamic Code Monitoring Using Static Instrumentation for a Nonintrusive Virtual Machine Observation". International Journal on Electrical Engineering and Informatics. 2019. (UNDER REVIEW)

[2] Paundu,A.W., Doudou,F, Miyamoto,D., Kadobayashi,Y. "Leveraging KVM Events to Detect Cache-based Side Channel Attack in Virtualization Environment". Security and Communication Networks [Special Issue:Emerging and Unconventional: New Attacks and Innovative Detection Techniques]. March 2018. (`https://www.hindawi.com/journals/scn/2018/4216240/`)

II Conference Papers

[3] Paundu,A.W., Okuda,T., Kadobayashi,Y., Yamaguchi,S. Sequence-Based Analysis of Static Probe Instrumentation Data for a VMM-Based Anomaly Detection System. 3rd IEEE International Conference on Cyber Security and Cloud Computing. June 2016. (`https://ieeexplore.ieee.org/document/7545902/`)

[4] Paundu,A.W., Okuda,T., Kadobayashi,Y., Yamaguchi,S. Leveraging static probe instrumentation for VM-based anomaly detection system. ICICS 2015: International Conference on Information and Communications Security. December 2015. (`https://link.springer.com/chapter/10.1007/978-3-319-29814-6_27`)

III Non-Doctoral Program Papers

[5] Niswar,M., Wainalang,S., Ilham,A.A., Zainuddin,Z., Fujaya,Y., Muslimin,Z., Paundu,A.W., Kashihara,S., Fall,D. IoT-based Water Quality Monitor-

ing System for Soft-Shell Crab Farming. 2018 IEEE International Conference on Internet of Things and Intelligence System, IoTAIS 2018. 2018. (https://ieeexplore.ieee.org/document/8600828)

[6] Zainuddin,Z., Paundu,A.W. End user based measurement system for cellular packet data network performance. COMNETSAT 2012: 2012 IEEE International Conference on Communication, Networks and Satellite. 2012. (https://ieeexplore.ieee.org/document/6380792)